University of Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study program/specialization: Computer Science	Spring semester, 2018 Open / ~~Confidential~~
Author: Jon Arne Bø Hovda	…… (signature of author)
Programme coordinator: Krisztian Balog Supervisor(s): Krisztian Balog and Darío Garigliotti	
Title of Master's Thesis: Automatic Entity Typing using Deep Learning	
Credits: 30 ECTS	
Keywords: Neural Networks • Deep Learning • Information Extraction • Clustering and Classification	Number of pages: 81 + supplemental material/other: - Code included in PDF Stavanger, June 15, 2018

Faculty of Science and Technology
Department of Electrical Engineering and Computer Science

Automatic Entity Typing using Deep Learning

Master's Thesis in Computer Science

by

Jon Arne Bø Hovda

Internal Supervisors

Krisztian Balog

Darío Garigliotti

June 15, 2018

Abstract

Knowledge bases contain vast amounts of information about entities and their semantic types. These can be leveraged in a variety of information access tasks like natural language processing and information retrieval. However, knowledge bases are incomplete, emerging entities need to be typed correctly, and existing entities must keep up to date. This is a strenuous task, and so any manual assignment of types is both error-prone and highly inefficient. In this thesis, we address the task of automatically assigning types to entities in a knowledge base.

Existing entity typing methods require great amounts of information about the knowledge base structure and properties, or assume that entity definitions are of a fixed nature. What we propose instead are two neural network architectures, one shallow and one deeper, which take short entity descriptions and, optionally, entity relationships as input. The goal to support knowledge bases with accurate entity typing of both existing and emerging types.

We experiment using the DBpedia knowledge base for evaluation, using two datasets: one reflecting accuracy on typing existing entities, the other on emerging entities. Results show that both our approaches are able to substantially and significantly outperform a state-of-the-art baseline, proving that neural networks can be used to support knowledge bases staying up-to-date and reduce overall incompleteness.

Acknowledgements

I would like to thank my supervisors Krisztian Balog and Darío Garigliotti for the helpful guidance and feedback provided through the work of this thesis.

Contents

Abbreviations

KB	**K**nowledge **B**ase
NER	**N**amed **E**ntity **R**ecognition
NLP	**N**atural **L**anguage **P**rocessing
NN	**N**eural **N**etwork
OWL	**W**eb **O**ntology **L**anguage
RDF	**R**esource **D**escription **F**ramework
URI	**U**niform **R**esource **I**dentifier

Chapter 1

Introduction

Wikipedia is an example community effort for representing facts about the world in one place that is curated by the readers themselves. However, this information is not very well structured, which is where the efforts of DBpedia as a *knowledge base* (KB) and similar projects chimes in. The structure of a KB makes it a powerful tool for describing, linking and defining knowledge through entities and their types.

An *entity* is described by its categories, e.g., a thing, person or country among others. These categories can be described as *types*, e.g., a Wikipedia page where *Norway* is the entity, and `Country` is one of its types. Entities can have different amounts of typing information. some are easily described by just one type, others are more uncertain or ambiguous.

Having correct type assignments makes way for powerful tools. Tasks can exploit this information for information extraction, document classification, natural language processing, and information retrieval. Example queries, e.g., listing *Musicians born in Norway* and *Musicians who are also actors*, can be answered efficiently and with relative ease. However, the typing information associated with entities in the KB is often incomplete, imperfect, or missing altogether. In addition, as new entities emerge on a daily basis, KBs should strive to stay up-to-date to provide relevant responses to its queries.

Existing methods for KB completion [1, 2] support KB incompleteness with good accuracy. Though these methods require great amounts of knowledge of the KB structure in question or assume that entity definitions are of a fixed nature (for example, "Norway **is a** country"). We therefore, set out to solve the task to automatically type entities in a KB.

1.1 Objectives

A primary goal of this thesis is to support KB incompleteness by typing emerging and existing entities. We want to do so while not being overly tied to the KB structure, while also providing a minimal amount of information about the entities themselves. Therefore, we investigate whether we can use short entity descriptions fed to a neural network, with the goal to infer a single, most correct type with high accuracy. We then compare to a state-of-the-art baseline, and formulate the following
RQ1: Can a neural approach, using only entity descriptions, outperform the current state-of-the-art?

Entity relationships might provide valuable information to its type. In other words, an entity connected to other entities with a set of particular types might prove to be an indication of the type of the entity itself. Therefore we also define following
RQ2: Can entity relationship information contribute to type prediction?

When implementing methods to solve entity typing, we investigate two main methods: one approach using a shallow neural network, and the other which is deeper. The motivation is to see whether the added depth further improves typing accuracy, or if the improvement is negligible. Therefore, we finally formulate the following
RQ3: Which of the two proposed solutions perform better?

1.2 Approach and Contributions

In order to address the challenge of automatically assigning a single most correct type to a given entity, we propose two simple fully-connected feedforward neural network architectures. We then experiment with a variety of input entity representations in order to predict a type label.

The following contributions are made by this thesis:

- Two neural network architectures for predicting typing information in a KB.
- Two evaluation sets based on DBpedia 2016-10: one for evaluating performance on established entities, the other focusing on emerging entities.
- An evaluation framework is produced in order to evaluate performance against a ground truth. This framework takes the ontology itself into consideration and rewards typing information according to how close it is to the ground truth.

1.3 Outline

We structure the thesis as follows: Chapter 2 covers key concepts surrounding knowledge bases, schemas, and ontologies. Related methods for automatic entity typing are discussed, including detailed descriptions of baseline methods *SDType* and *Tipalo*. The chapter concludes with a basic introduction to neural networks and distributed representations. We present our design and approach to solving the challenge of KB completion in Chap. 3. Followed up by Chap. 4 on evaluation, where we delve into detail how we evaluate our methods and compare them to a baseline. We present our results and provide a detailed analysis. Finally, in Chap. 5, we conclude the thesis and present future directions for our work.

Chapter 2

Related Work

This chapter presents the main concepts in this thesis. First, a basic introduction to entities and types is provided in Sect. 2.1. Followed up by knowledge repositories in Sect. 2.2, and how they are represented by knowledge bases, using schemas and ontologies, in Sect. 2.3. We give some examples of related entity typing methods, both for typing in context, and in knowledge bases, in Sect. 2.4. Finally, an overview of neural networks and distributed representations are given in Sects. 2.5 and 2.6

2.1 Entities and Types

An **entity** can be anything from an actor like *Keanu Reeves*, to a work of art like the *Mona Lisa*. The simplest explanation is that an entity is something that is uniquely identified, be it an idea, a technology, or a car manufacturer. An entity is often accompanied with a **type**, e.g., *Keanu Reeves* is an instance of type `Actor`, and the *Mona Lisa* is an instance of type `Painting`.[1]

Entities can have several type assignments. *Arnold Schwarzenegger* is a good example seeing that he may be assigned several types, like `Actor`, `Politician`, and `BusinessPerson` among many others. The extent of how many types an entity should have is one of the challenges tied to entity typing (i.e., a single type, or several generic types).

How specific the typing should be is also a challenge. An example is giving a very generic type like `Person` to *Keanu Reeves*, which is not very telling of the entity type. Typing him as `Canadian male actors of Asian descent`[2] might, however, be a bit too specific. These challenges will be further explored in Sect. 2.4.

[1] Types all from DBpedia ontology unless stated otherwise.
[2] Type from YAGO ontology.

2.2 Knowledge Repositories

In order to capture what entities are, and how they relate to each other, it is helpful to have fundamental knowledge which reflects their properties and definitions. This knowledge can have its background in many formats, though it is often categorized and structured in specific ways. We call them *knowledge repositories*, which Balog [3] describes as follows: "A knowledge repository is a catalog of entities along with their descriptions and properties in semi-structured or structured format." A great example of a knowledge repository is Wikipedia.

2.2.1 Wikipedia

Wikipedia is an online encyclopedia and is of no doubt a popular source of information, written and edited by the readers themselves. It is the world's largest community-driven effort, which currently is the fifth most popular website.[3] Wikipedia presents information that is highly relevant and up-to-date, containing over 5.5 million articles in the English version.[4] As stated by Hovy et al. [4], Wikipedia is "[...] the largest and most popular collaborative semi-structured resource of world and linguistic knowledge".

The semi-structured property of Wikipedia is given due to the fact that a page might not cohere to a given template. Some pages might be structured with an abstract, an info-box, table-of-contents, and several sections, while another page might only contain a title and short description. Nevertheless, Wikipedia is linked, and as pages link from one to another, categories and lists form a structure that is useful to describe relations between things and ideas. An example is shown in Fig. 2.1: the page is adhering to the previously mentioned structure, containing an info-box, table of contents, lists, and more.

Due to the fact that Wikipedia is written in a semi-structured format, it is not very machine-readable. Gathering Wikipedia's information into a fully structured format would therefore be very valuable, paving the way for analysis of data and provide an immense corpus available for everyone.

2.3 Knowledge Bases, Schemas and Ontologies

A knowledge base (KB) is used to store information, both structured and unstructured. It is often used to describe facts about entities and their relations. Balog [3] mentions

[3] According to Alexa: https://www.alexa.com/topsites as of 03.03.18
[4] https://meta.wikimedia.org/wiki/List_of_Wikipedias
[5] https://en.wikipedia.org/wiki/University_of_Stavanger

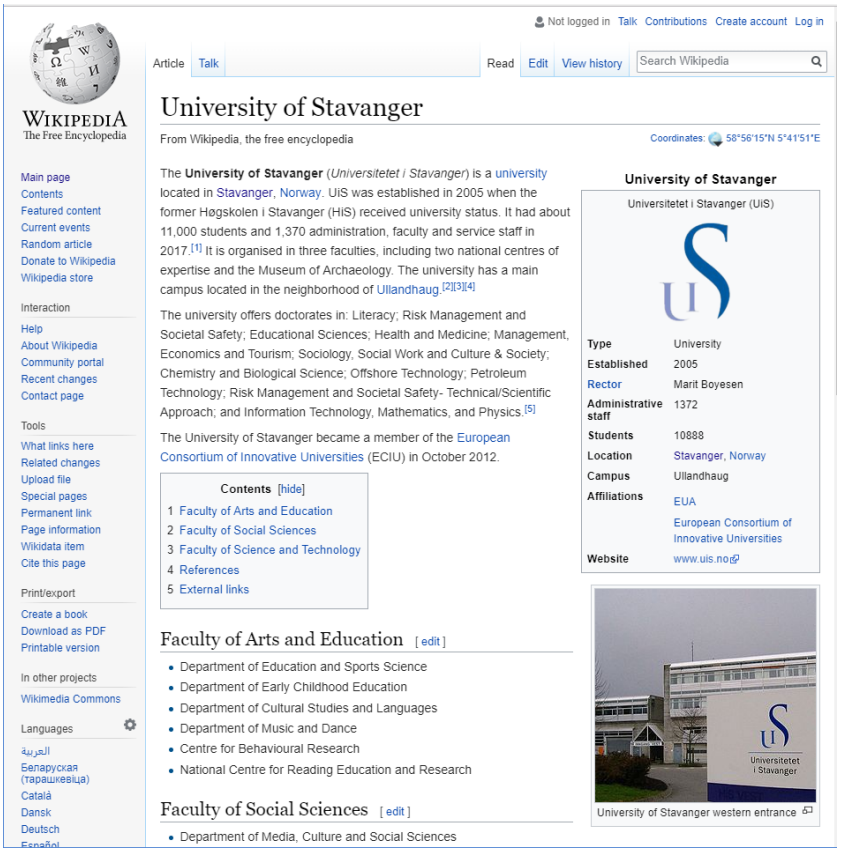

Figure 2.1: Image of the *University of Stavanger* Wikipedia page[5] showing a rich page containing an info-box, table of contents and good structure.

the two layers of a KB: *schema level* and *instance level*. On a schema level, a knowledge model defines relations between types and entities, their classes, sub-classes, and the properties they can have. This knowledge model is often accompanied by a hierarchy, in which the classes are structured. On the instance level, individual entities are described by their names, the attributes they have and their relations with other entities.

One of the earlier uses of a KB is in a two-component knowledge-based system [5]. One component is the KB, which is used to represent facts to describe the world, and the other component is an inference engine, using those facts to arrive at new facts or prove inconsistencies with former facts. Together, they become highly useful for the likes of *expert systems*.

An example of such an expert system is the Cyc project. Started in 1984, the project is the longest ongoing artificial intelligence project [6]. The Cyc project uses handcrafted facts entered into a KB, and these facts are used to further infer new facts by use of logical inference rules [7]. However, it is a challenging project seeing that it is maintained and updated by human-knowledge engineers, making it laborious and error-prone to have a KB which is both up-to-date and correct.

It is challenging for a KB to be complete, as new entities emerge on a daily basis, and missing statements, be it either human-error or machine-error, are bound to take place. A KB derived from a large source of information, which is up-to-date and community-driven, would not necessarily eliminate these issues, but they would be severely reduced. It is here where the effort to use Wikipedia as a knowledge repository comes in, in order to produce a feature-rich KB.

2.3.1 Resource Description Framework

In order to represent knowledge in a KB, it is useful to have a format which is agreed upon by the community to act as a standard. The Resource Description Framework (RDF) is such an effort and is standardized by the World Wide Web Consortium (W3C).[6] RDF is a model for data interchange on the web, and thus describes resources across the web even if the underlying schemas differ. A resource is typically described as follows: *<Subject> <Predicate> <Object>*.

These Subject, Predicate, Object (SPO)-triples, or RDF statements, describe what or how the subject relates to the object. For example, *the car has the color blue* simply describes the car (subject) having the color (predicate) which is blue (object). More specifically, these RDF statements consist of Uniform Resource Identifiers (URI). The *subject* is a resource which is uniquely identified by its URI. The *predicate* is also a URI, describing a relation from the subject, or a subject's property. The *object* is not always a URI, depending on whether it is used to link to another resource or just a literal [3]. In a KB, these triples form the previously mentioned *instance level*. Example triples are given in Listing 1.

Listing 1 RDF triple examples from DBpedia page on Keanu Reeves. URIs are shortened for better visibility.

```
<dbr:Keanu_Reeves> <foaf:name>       "Keanu Reeves"
<dbr:Keanu_Reeves> <dbo:birthDate>   "1964-09-02"
<dbr:Keanu_Reeves> <dbo:birthPlace> <dbr:Lebanon>
<dbr:Keanu_Reeves> <rdf:#type>       <dbo:Agent>
```

2.3.2 Web Ontology Language

While RDF forms the instance level of a KB, the *schema level* is formed by OWL (Web Ontology Language). OWL, along with RDF, is also part of the W3C standardization. The purpose of these schemas is to describe the structure of the knowledge contained

[6] https://www.w3.org/standards/techs/rdf

within for example a KB, making relations between properties and classes more specific. OWL adds semantics to the schema so that a relation like *A isPartnerWith B* implies that *B isPartnerWith A*. Furthermore, this greatly expands the usage of classes, meaning that a relation *B isAncestorOf C* and *C isAncestorOf D* implies that *B isAncestorOf D*. Another relation often used is the *owl:sameAs*, which can map the same entity across several domains, or be used to describe classes that are similar as seen in Listing 2.

Listing 2 OWL example of `sameAs` property.

```
<owl:Class rdf:ID="FootballTeam">
    <owl:sameAs rdf:resource="http://sports.org/US#SoccerTeam"/>
</owl:Class>
```

2.3.3 DBpedia

The motivation behind DBpedia comes from the important role KBs have in the increasing interest of enhanced web intelligence and enterprise search [8]. Previously, KBs were more domain-specific, created and maintained by small amounts of people. These KBs are therefore costly to maintain and difficult to keep up-to-date.

DBpedia is a KB which extracts structured information from Wikipedia, making this information available for everyone to use.[7] It is multilingual and cross-domain, meaning they link to other KBs e.g., YAGO and Wikidata. DBpedia is engaged by the community, and it evolves along with Wikipedia. It is presented in [8], and is further expanded upon in [9].

DBpedia can be leveraged in several ways. Originally, it was implemented to answer queries not easily answered in Wikipedia [9]. Example queries like *find all tunnels longer than 10km* and *list actors who have a background as musicians* are efficiently answered. These queries are possible because of the DBpedia ontology, a tree-like structure where types are stored in a hierarchy.

Some example types in the DBpedia ontology is seen in Fig. 2.2. Here, we see `Thing` as the root node, which we will later see to ignore as it is not helpful in entity typing. `Agent` and `Activity` are examples of *top-level types*, these top-level types are useful for analyzing how types are distributed in an ontology. However, we will soon see that some types are more used than others, which is especially the case in the top-level type `Agent`. Therefore, we will see that when referring to top-level types, we will often ignore `Agent`, and rather use its descendants (e.g., `Person` and `Organisation`) as top-level types instead.

[7] http://wiki.dbpedia.org/

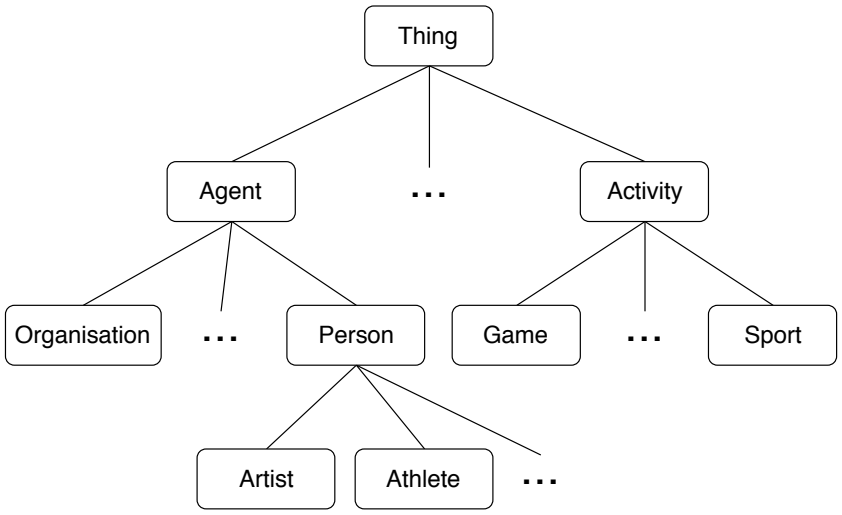

Figure 2.2: Example types in DBpedia ontology structure.

Table 2.1 shows the top-level distribution of the DBpedia dataset *Instance Types*.[8] Here, we can clearly see how some top-level types in DBpedia are significantly more used than others.

DBpedia is useful in many domains, e.g., data integration, topic detection, document ranking, and perhaps most relevant for this thesis: named entity recognition (NER) [9]. More specifically, typing entities in context, and using DBpedia itself for typing entities in KBs. We will come back to typing entities in Sect. 2.4.

2.3.4 Similar Ontologies

There are several similar and related ontologies to DBpedia. Garigliotti and Balog [10] do a comparison between four of them in Table 2.2. Basic statistics about the ontology itself is presented, and also some statistics on how entities are typed in the respective ontologies.

DBpedia is previously presented in Sect. 2.3.3, the difference from DBpedia presented in Table 2.2 is the version used (2015-10 vs. 2016-10 in this thesis). Meaning that some statistics, like the number of types, have changed (from 713 to 760).

Freebase was a large, scalable KB used for structuring general human knowledge [11]. Like DBpedia, it is maintained in a collaborative effort. Though it has since shut down, its data is being transferred to Wikidata, and Google's Knowledge Graph API has since been announced to replace the Freebase API.[9] Freebase data dumps are still actively

[8] Retrieved from https://wiki.dbpedia.org/downloads-2016-10
[9] https://developers.google.com/knowledge-graph/

Table 2.1: Distribution of top-level types in DBpedia *Instance Types* dataset.

Top type	Amount	%
Person	1,243,400	26.07
TimePeriod	1,127,588	23.65
Place	839,987	17.61
Work	496,036	10.40
Species	306,104	6.42
Organisation	285,422	5.98
PersonFunction	171,178	3.59
Event	76,029	1.59
MeanOfTransportation	56,792	1.19
SportsSeason	55,730	1.16
Device	24,161	0.50
ChemicalSubstance	18,033	0.37
Activity	10,631	0.22
Language	9,215	0.19
Biomolecule	7,761	0.16
Disease	6,108	0.12
Food	6,056	0.12
Award	5,549	0.11
EthnicGroup	5,391	0.11
AnatomicalStructure	4,346	0.09
Name	4,345	0.09
UnitOfWork	2,824	0.05
TopicalConcept	1,709	0.03
Holiday	1,138	0.02
Colour	902	0.02
SportCompetitionResult	822	0.02
Currency	391	0.01
GeneLocation	4	<0.01

used, and Freebase statistics in Table 2.2 are retrieved from the public data dump from 2015-03-31.

Yet Another Great Ontology (YAGO) extracts information from Wikipedia, and unifies that information with GeoNames and WordNet [12]. Like DBpedia, the information is available for everyone.[10] YAGO differs from DBpedia in many ways due to entirely different design goals, while DBpedia is intentionally shallow in its class hierarchy, YAGO is comparatively deep. YAGO statistics in Table 2.2 are from YAGO version 3.0.2.

Wikipedia categories represents the Wikipedia category system, which is a graph rather than a type hierarchy. How the conversion from graph to type taxonomy is done, we refer to [10].

The main focus for this section is to present other ontologies, and how they compare to each other. In Table 2.2, it can be seen that DBpedia has the fewest amount of types, while also having the least amount of top-level and leaf-level types. While the height

[10] http://www.mpi-inf.mpg.de/yago-naga/yago/

Table 2.2: Overview of normalized type taxonomies and their statistics. The top block is about the taxonomy itself; the bottom block is about type assignments of entities. Table retrieved from [10].

Type system	DBpedia	Freebase	Wikipedia categories	YAGO
#types	713	1,719	423,636	568,672
#top-level types	22	92	27	61
#leaf-level types	561	1,626	303,956	549,754
height	7	2	35	19
#types used	408	1,626	359,159	314,632
#entities w/ type	4.87M	3.27M	3.52M	2.88M
avg #types/entity	2.8	4.4	20.8	13.4
mode depth	2	2	11	4

of the DBpedia ontology is designed to be intentionally shallow, it is interesting to see Freebase only having a height of 2, quite small when comparing those to the height of Wikipedia categories and YAGO.

When comparing the number of types used, DBpedia is seen only using 408 out of the 713 available types. It is also interesting to see how DBpedia has an average amount of types per entity (avg #types/entity in Table 2.2) of just 2.8, while the other ontologies have significantly more. Therefore, while there are more types in the taxonomy, entities are also typed with an increased amount of types, which can be a consequence of having many types that are similar or indistinguishable.

From these comparisons, DBpedia appears to be more fine-grained and manageable in its size and scope. Thus confirming that it is a good KB for testing our approach to entity typing.

2.4 Automatic Entity Typing

This section presents related and existing methods for entity typing. First, we introduce existing methods for typing entities in context in Sect. 2.4.1. Then, we describe methods for typing entities in KBs in Sect. 2.4.2.

2.4.1 Typing Entities in Context

Though this thesis takes on the challenge of typing entities in knowledge bases, there are similar fields of semantic web research for typing entities in context. Some of these are entity linking and named entity recognition (NER), which Lin et al. [13] define as the

task of identifying named entities in text. NER can for simplicity be divided into two objectives, (1) to find entities in a given text, and (2) typing the entities found in (1). While this thesis resembles objective (2) more, there are some interesting distinctions.

In NER, it is important to type an entity according to the context surrounding it. Therefore, in typing an entity, natural language processing (NLP) is a helpful tool leveraged to find correct typing information. Many NER tasks also differ in how specific the typing should be, some only need to correctly type a scientist as a person in order to get the correct type, other methods produce more fine-grained typing information.

One of the many challenges NER has to solve is to get the correct sense from a text. For example, consider the text *Germany won the world cup of 2014* Is "Germany" supposed to link to the country here, or would it be more correct to link to the German national football team? Similarly, the "world cup of 2014" has to be typed correctly, in this context meaning the FIFA World Cup, but it could just as well be a world cup in another sport, in another context.

An additional NER challenge is the typing of unlinkable entities, this is entities which appear in a text but are not able to link to, e.g., Wikipedia. Lin et al. [13] approach this challenge and calls it the *unlinkable noun phrase problem*. An example of an unlinkable entity can be illustrated by the given text: "Some people think that pineapple juice is good for vitamin C." Some NER tasks correctly type "vitamin C" as a nutrient, but fails to type "pineapple juice" because it does not have a Wikipedia entry, typing it simply as "pineapple" instead. While the unlinkable noun phrase problem is not very relevant to this thesis, the way they type entities is interesting. By using Google Books n-grams, they are able to produce a list which predicts typing information by textual relations and looking at similar linked entities. For example, "Microsoft has released an update to ..." gives a list of similar entities which produces that same or similar text, the list containing entries like Apple, Google, IBM, etc. Thus they are able to predict that these similarities mean Microsoft probably is a business, software developer, organization, and other related types.

A final example is the task of typing emerging entities, that is, entities which are new and might not have a Wikipedia page. Nakashole et al. [14] seek to solve this by typing emerging entities as they get popular in the news and social media. As this thesis seeks to automatically type entities, typing emerging entities not already in a KB is an important feature.

Typing entities in context can therefore be seen as both a different approach from typing entities in KBs and as a similar approach as they share a lot of challenges.

2.4.2 Typing Entities in Knowledge Bases

Here, we present related knowledge base completion methods. We briefly introduce two methods with similar approaches first, followed by a detailed description of the two baseline methods *Tipalo* and *SDType*.

A supervised classification approach on lexico-syntactic patterns is presented in [15]. They train a hierarchy of support vector machines on bag-of-words of short abstracts and Wikipedia categories. Similarly to our approach, they exploit entity short descriptions, but whereas they require several classifiers, we propose a single all-in-one model.

The other approach to KB completion uses corpus-level entity tying. Yaghoobzadeh and Schütze [16] implement a multilayer perceptron approach to assign entity types using word embeddings. While similar in this underlying approach, we correspond to a larger type system (112 FIGER types vs. 760 DBpedia types) and plan to utilize various input entity representations.

We are inspired by these approaches in designing our solution to entity typing using neural networks. What follows is a detailed description of our baseline methods *Tipalo* and *SDType*.

Tipalo

Tipalo is a tool for automatically typing DBpedia entities and is presented in [1]. Its goal is to automatically type an entity given the natural language (NL) definitions of that entity's corresponding Wikipedia page. This way, they do not need to rely on how the Wikipedia page is structured or categorized. The overlying approach is to extract RDF and OWL representations from an entity's abstract. A tool for word sense disambiguation is then used to link the extracted types to WordNet. The method then further aligns these types to WordNet super senses and DUL+DnS Ultralite, which are ontologies used to further generalize the typing results. The resulting method is able to type entities with good accuracy, and is available as a web service.[11]

Figure 2.3 is helpful in order to describe Tipalo. The figure shows a pipeline of the components making up Tipalo, from extracting the abstract from a Wikipedia page, to outputting the typing information.

First in the pipeline is the **definition extractor**. The definition extractor retrieves the Wikipedia abstract from a DBpedia entity in order to determine the definition of that entity. An entity is often described how it is typed by the first sentence of a Wikipedia

[11] http://wit.istc.cnr.it/stlab-tools/tipalo

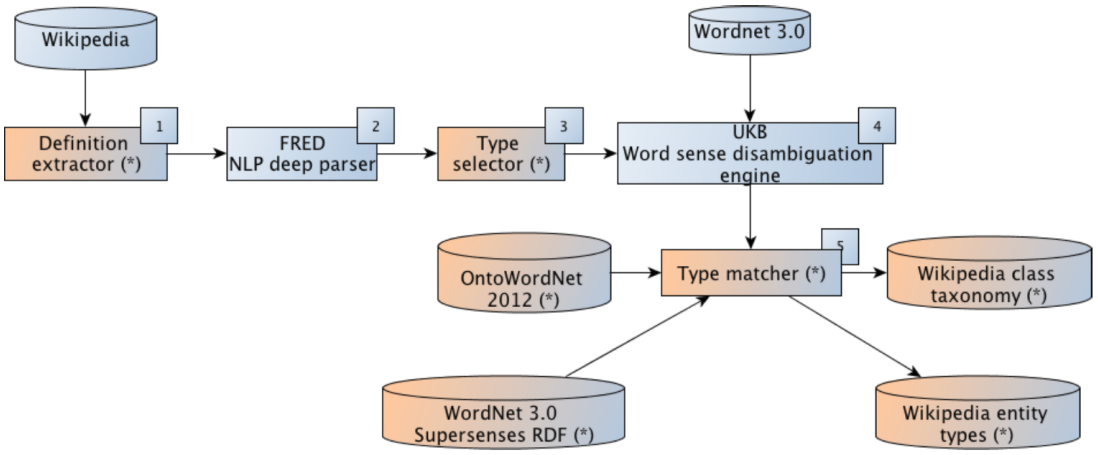

Figure 2.3: Pipeline displaying interconnected parts making up the Tipalo system. Numbers signify the order of execution. (*) denotes components and datasets made for Tipalo. Figure is found in [1].

Keanu Reeves

From Wikipedia, the free encyclopedia

This article is about the Canadian actor. For the Philippine actress, see Keanna Reeves.
"Keanu" redirects here. For other uses, see Keanu (disambiguation).

Keanu Charles Reeves (/kiˈɑːnuː/ *kee-AH-noo*;[1][2] born September 2, 1964) is a Canadian[a] actor, director, producer, and musician. He gained fame for his starring role performances in several blockbuster films, including comedies from the *Bill and Ted* franchise (1989–1991); action thrillers *Point Break* (1991), *Speed* (1994), and the *John Wick* franchise; psychological thriller *The Devil's Advocate* (1997); supernatural thriller *Constantine* (2005); and science fiction/action series *The Matrix* (1999–2003). He has also appeared in dramatic films such as *Dangerous Liaisons* (1988), *My Own Private Idaho* (1991), and *Little Buddha* (1993), as well as the romantic horror *Bram Stoker's Dracula* (1992).

Figure 2.4: Abstract from the Wikipedia page on Keanu Reeves

abstract, though there are exceptions. It might be described in the sentence following the first one, or maybe both sentences are required in order to define what the entity is. Gangemi et al. [1] describe how the definition extractor solves this: "We rely on a set of heuristics based on lexico-syntactic patterns and Wikipedia markup conventions in order to extract such sentences." One of these conventions is how a Wikipedia abstract often uses bold characters on the entity the page is about. Figure 2.4 shows an example of this convention. Furthermore, in [1] they note that the first sentence in an abstract is often of the form: "*bold-name <copula><predicative nominal//predicative adjective>*." From Fig. 2.4 this form is seen: "**Keanu Charles Reeves** (...) is a Canadian actor, director, producer, and musician." However, there are exceptions: The "is a" copula relationship is not always contained in the first sentence. The definition extractor then takes this into account by instead extracting the subsequent sentences containing the "is a" relationship. If there are no bold-named entities, and no "is a" relations are found, the first sentence is simply returned as a base-case.

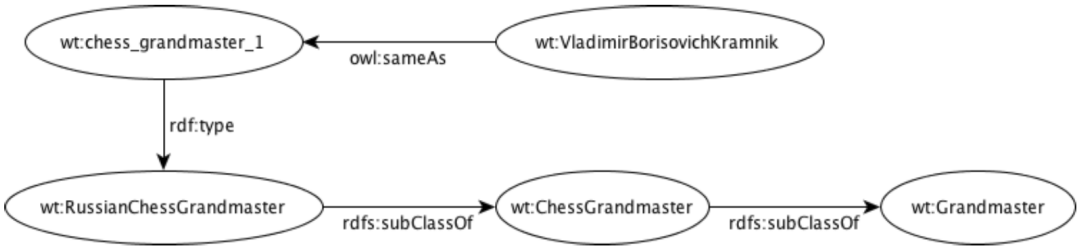

Figure 2.5: FRED output from the Wikipedia definition "Vladimir Borisovich Kramnik is a Russian chess grandmaster." Figure from [1].

The second component is **FRED**, or rather, the **natural language processing deep parser**. Having the definition of an entity extracted from the previous definition extractor, that definition is now to be parsed into a more logical form. This logical form is helpful in order to represent the potential types the entity can have. FRED is a tool that accomplishes this task and is presented in [17]. Briefly explained by Gangemi et al. [1], FRED "... implements a logical interpretation of NL represented in Discourse Representation Theory (DRT)." Its implementation is out of scope from this thesis, however, the important part is that FRED outputs an OWL representation and a taxonomy of types from an entity's NL definition. FRED, like Tipalo, is available as a web service.[12] An example output from FRED is shown in Fig. 2.5.

The third component is the **type selector**. The purpose of the type selector is to figure out which typing information to keep, and which to discard. More specifically, it analyses the output graph from FRED, and through a set of graph patterns, decides whether the graph depicts an individual entity or a class entity. This is an important distinction to make, seeing as the output from FRED is significantly different in those situations. An example here is the FRED output from the NL definition of *chess piece* seen in Fig. 2.6. Comparing Fig. 2.5 and Fig. 2.6 shows two very different graphs, one having only a single `rdf:type` relation, while the other has several. The type selector then has to recognize graph patterns, decide whether it is a class entity or individual entity, and finally select typing information to keep and discard. They identify a set of graph patterns in [1], which follow similar criteria as lexico-syntactic patterns,[13] and are extended with OWL semantics and RDF graph topology.

Ten graph patterns are used in [1], six for identifying individual entities, and four for class entities. An example graph pattern is shown in Table 2.3. A good description is given when they distinguish an individual from a class [1]:

[12] http://wit.istc.cnr.it/stlab-tools/fred

[13] Lexico-syntactic patterns: A string-matching pattern based on text tokens and syntactic structure. From https://en.wiktionary.org/wiki/lexicosyntactic_pattern

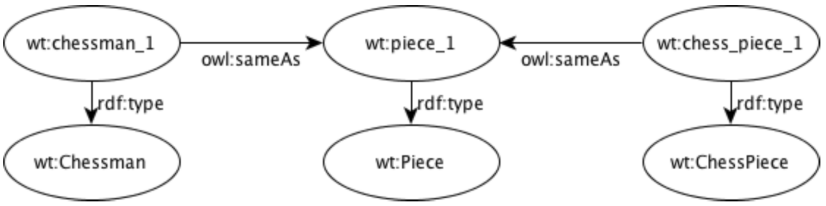

Figure 2.6: FRED output from "Chess pieces, or chessmen, are the pieces deployed on a chessboard to play the game of chess." Figure from [1].

Table 2.3: Graph pattern excerpt from [1].

ID	graph pattern	inferred axioms
gp_4	e **owl:sameAs** x && x **rdf:type** C	e **rdf:type** C

> "Firstly, the type selector distinguishes if an entity is either an individual or a class entity: given an entity e, it is an individual if it participates in a graph pattern of type e **owl:sameAs** x, it is a class if it participates in a graph pattern of type x **rdf:type** e."

The type selector goes through the graph patterns in priority order, starting from one and upwards. Thus, in cases where several graph patterns might be a good fit, it simply selects the graph pattern which is hit first. For example, if gp_4 is found as a match, it is selected with no need to further evaluate gp_5, gp_6, and so on.

Furthermore, the type selector checks if any of the terms having a `rdf:type` relationship can be referenced as a DBpedia entity. This is done in order to achieve higher cohesion, and improve internal linking within DBpedia [1].

The fourth stage is the **UKB Word sense disambiguation engine**. The result from the type selector is the entity's types and their relations. This resulting information must then be gathered into their correct *sense*, hence the need for a word sense disambiguation tool. One way to solve this is to align typing information into WordNet. WordNet is a large lexical database functioning in many ways like a thesaurus.[14] UKB is a tool which performs graph-based word sense disambiguation, lexical relatedness and similarity using an existing knowledge base.[15] Here, a UKB returns a WordNet *synset*[16] which is the best fit considering the context from the entity definition. Though a UKB provides good results in precision and recall, the performance suffers, especially in large datasets such as Wikipedia [1]. An alternative solution is selecting the most frequent WordNet sense instead, providing great performance but with lower precision and recall. The result

[14] https://wordnet.princeton.edu/
[15] http://ixa2.si.ehu.es/ukb/
[16] Synset: A set of one or more synonyms that are interchangeable in some context without changing the truth value of the proposition in which they are embedded. From https://en.wiktionary.org/wiki/synset

from the word sense disambiguation engine is the addition of a WordNet type, given from the corresponding synset to an entity.

Finally is the fifth component, the **type matcher**. From the previous steps, an entity is now accompanied by a WordNet type. The final step links that typing information to other ontologies on the Semantic Web. Gangemi et al. [1] further produces two additional RDF datasets. One of these aligns Wordnet synsets to *super senses*, which are very broad, lexico-semantic categories. The other align synsets with some foundational ontology classes. What this means then, is that the type matcher uses these alignments in order to further produce additional triples. These triples are very generic: where the entity *Chess game*, for example, is aligned with the class **activity**.

To summarize, Tipalo automatically types entities using the NL definitions of the corresponding Wikipedia abstract, Gangemi et al. [1] evaluate Tipalo using a manually annotated golden standard. They report a precision of 0.76, a recall of 0.74 and an F-measure of 0.75.

We intend to use Tipalo as a baseline, as it is a method also making use of short entity descriptions to type an entity. However, the Tipalo service has not been available for the duration of the master's thesis, meaning that SDType will be used as the only baseline. We will see in the following section that SDType is able to outperform Tipalo, therefore, the loss of Tipalo as a baseline function is acceptable.

SDType

Another tool for automatic entity typing is *SDType*. It is presented in [2] and is further explained in [18]. SDType takes advantage of the statistical properties between instances in a knowledge base. A key advantage here is that only the data itself is used, meaning that SDType does not need any external knowledge. SDType can therefore be implemented in different knowledge bases, with the intent to increase the quality of noisy and incomplete typing information [18]. It does so with high accuracy and has since been integrated as part of the latest DBpedia releases.[17]

A knowledge base consists of what they term *A-boxes* and *T-boxes* (presented respectively as instance-level and schema-level in Sect. 2.3). A-boxes are the definition of the instances themselves and the relations between them, while T-boxes is the schema or ontology they are contained in [2]. SDType makes use of the statistical distributions of connecting pairs between A-box resources. If an instance is connected to other instances in certain ways, a type can be inferred based on that information. As Paulheim and Bizer [18] state:

[17] Instance Types Sdtyped Dbo in http://wiki.dbpedia.org/downloads-2016-10

Table 2.4: Distribution of subject and object types for property `dbpebia-owl:location`. Table from [18].

Type	Subject(%)	Object(%)
owl:Thing	100.0	88.6
dbpedia-owl:Place	69.8	87.6
dbpedia-owl:PopulatedPlace	0.0	84.7
dbpedia-owl:ArchitecturalStructure	50.7	0.0
dbpedia-owl:Settlement	0.0	50.6
dbpedia-owl:Building	34.0	0.0
dbpedia-owl:Organization	29.1	0.0
dbpedia-owl:City	0.0	24.2
...	...	...

"The basic idea of the SDType algorithm is that when observing a certain property in the predicate position of a statement, we can infer the types of the statement's subject and object with certain probabilities."

Table 2.4 is used to give a notion of how SDType takes statistical properties when inferring typing information. Here, the distribution of the DBpedia property `dbpedia-owl:Location` is shown. Given a property that is a location, 100% of the subjects are of type `owl:Thing`, while 69.8% of the subjects are of type `dbpedia-owl:Place`. Similarly, 88.6% of the objects are of type `owl:Thing` and 87.6% of the objects are of type `dbpedia-owl:Place`, and so on. These percentages do not add up to 100% since a resource can have several types. Also, in [18], they describe the observation that not all objects are of type `owl:Thing`. The issue stems from the fact that types in DBpedia only are generated if an info-box is found, meaning that in these cases subjects were created from info-boxes, but objects were not generated from pages with info-boxes.

SDType then makes use of the properties connecting two resources as an indication for their typing information. Paulheim and Bizer [18] define the problem as a *link-based object classification approach*. By using ingoing and outgoing properties from a resource, they can indicate which type a resource should have. Paulheim and Bizer [18] further state: "SDType can be seen as a weighted voting approach, where each property can cast a vote on its object's types, using the statistical distribution to weight its votes." For example, in Table 2.4 and given a triple `x :dbpedia-owl:location :y`, the following conditional probabilities can be assigned:

$$P(\texttt{:x a dbpedia-owl:Place}) = 0.689,$$

$$P(\texttt{:y a dbpedia-owl:Place}) = 0.876.$$

Formally, finding out the likeliness of a type t given a resource containing a certain property *prop*, where *prop* may be an ingoing or outgoing property, is expressed as

follows:

$$P(\,t\,|\,(\exists\, prop\,.\,T\,)).$$

The above probabilities are taken from the statistical distributions from Table 2.4. It is also useful to get a notion of the predictive power of a property, therefore, a weight w_{prop} is assigned. These weights are different for predicting types in the subject and types in the object. In the `dbpeida-owl:location` example above, two weights $w_{dbpedia-owl:location}$ and $w_{dbpedia-owl:location}{}^{-1}$ would be added. These weights are used in order to avoid problems with skewed KBs [18]. A skewed KB may have extensions of types which are significantly more used than others. Without these weights, false typing prediction would occur. Examples of these properties are general purpose ones, such as `rdfs:label` and `owl:sameAs` [18].

To work around this problem, they define property weights w_{prop}. The purpose of property weights are to measure how the property's distribution deviates from the a priori distribution of all the types in the knowledge base [18]. A stronger deviation means it has a higher prediction power of a property. w_{prop} is defined as follows:

$$w_{prop} := \sum_{\text{all types } t} (P(t) - P(t\,|\,(\exists\, prop\,.\,T)))^2.$$

Having conditional properties and property weights, Paulheim and Bizer [18] implement a weighted voting approach. For each property, a vote is cast for the types in the property's distribution, resulting in a likelihood being assigned to each type. Summarizing all these likelihoods then tells the type distribution for a resource. This is a confidence of a resource r having the type t is given by the equation

$$\sum_{\text{all properties prop of } r} P(t(r)\,|\,(\exists\, prop\,.\,T)(r)).$$

They normalize it with a factor v as follows:

$$v := \frac{1}{\sum_{\text{all properties prop of } r} w_{prop}}.$$

The sum over "all properties prop of r" means the properties of all statements that have r either in the subject or the object [18]. They also state the importance of handling subject and objects separately.

Finally, a *confidence threshold* t is added. A type statement resulting with a confidence larger than t is assumed correct. They evaluate SDType by using t values 0.4 and 0.6.

The result is a tool which can correctly type entities, either new ones or correcting faulty existing typing assignments. The basic flow of how SDType works from input data to

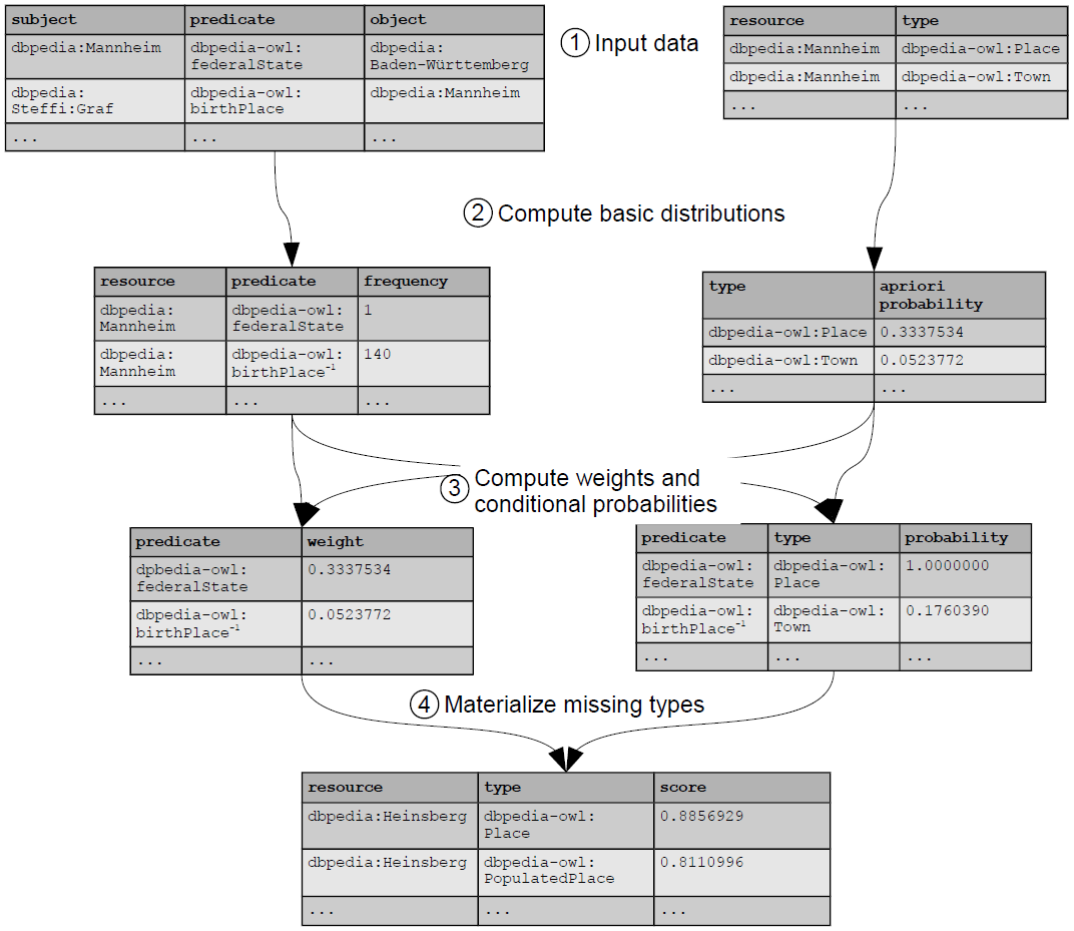

Figure 2.7: SDType type completion visualized as a series of table operations. Figure from [2].

type prediction is shown in Fig. 2.7. They evaluate SDType by using DBpedia as a golden standard and manage to get an F-Measure of up to 0.88. SDType can be applied to any cross-domain KB. Comparing to methods using Wikipedia resources, SDType can type resources with very sparse Wikipedia pages. Even Wikipedia red links can be typed using information from the incoming links [2]. SDType will be used as a baseline in order to compare the result of methods produced in this thesis.

Two different existing methods are now presented, one making use of natural language descriptions of an entity, and one leveraging statistical properties of entity property links. Tipalo proves that natural language definitions of entity descriptions can be used to infer a type, though they do so assuming that entities are described in a consistent way ("... is a ..."), and using only the first few sentences of the entity description.

SDType, on the other hand, shows that using statistical links between entities give a good indication of their type. Though they do so requiring great amounts of knowledge about

the KB structure itself, and how entity properties are linked. Furthermore, SDType does not produce very specific types, which Paulheim and Bizer [2] list as a future work.

Motivated by these two methods strengths and weaknesses, we set out to design an approach which is able to type entities using short entity description only, with the option to additionally provide entity-relationship data. In doing so, entities can be typed in a more flexible way, and without the need of great knowledge of the KB in question. Next, we describe the basics of neural networks, with the intent to use them for entity typing.

2.5 Neural Networks

Neural networks (NNs) have seen a continuous rise in popularity and usage in the last years, and due to a significant rise in computational power, they can solve more advanced and complex problems. NNs are inspired by the human biology of the brain, where neurons are able to learn and use past knowledge to recognize new or similar concepts [6]. Usually, a NN consists of several layers, including input layers, output layers, and one or several hidden layers. An input layer is responsible to receive data, which then sends that data to the hidden layers. Hidden layers are tasked to transform input through non-linear functions, in order to get a more abstract representation of that data [19]. Finally, the output layer transforms data from the hidden layers to an output format, which either in classification is a set of label scores, or other problems a binary true or false.

A motivation for using NNs to solve the task of typing entities based on entity descriptions is how well NNs perform in natural language processing (NLP) tasks in general. According to Goldberg [20], due to advances made recently by NN research, NNs are now implemented in many NLP tasks. In solving those tasks with great results, they become state-of-the-art approaches.

Following are more defined descriptions of NN layers useful for understanding why NNs are able to solve NLP tasks.

Input Layer

In general, NN input layer is used to represent data, doing so in a format that makes it possible for a model to learn anything about that input, and distinguish it from other inputs. There are many ways to provide text as input for an input layer, i.e., characters, words, even whole sentences or documents. However, for the sake of generalization,

the most flexible approach is to make use of the distributed representation of word embeddings, which we describe in Sect. 2.6.

For making use of distributed representations in the input layer, we must feed the input layer with vector representations of entity descriptions. These vectors have size d dimensions, and the size of the vectors often decide the size of the input layers (amount of neurons). In other words, if using word embeddings with a word2vec model that outputs 300-dimensional vectors, the size d of the input layer would likely also be 300, which is seen in the input layer of Fig. 2.8.

Hidden Layer

There are many ways for a NN to connect its hidden layers, some involve convolutional architecture, others involve a recurrent architecture. Here, we describe the **fully connected feed-forward** approach for structuring the hidden layers. Each neuron in a hidden layer h_i is connected to every neuron in the following layer h_{i+1}. The transformation from one layer to the next is described in a simple way in [19], where each layer in a model does the transformation:

$$h_i = f(\mathrm{W}_i \mathrm{h}_{i-1}), \tag{2.1}$$

the previous layer h_{i-1} is used to apply a linear transformation using matrix W_i, then applying an *activation* function f, one of which is *ReLU* (rectified linear unit). ReLU activation is seen as:

$$f(x) = max(x, 0), \tag{2.2}$$

where x is input data. Meaning that ReLU outputs 0 if the input is less than 0, otherwise it outputs the raw input. Hidden layers are seen in Fig. 2.8 as the two middle layers of size 512.

Output Layer

The output layer is responsible for generating output depending on the states of the hidden layers. It is, like the hidden layers, often fully connected where each neuron represents a possible outcome depending on the task. For classification, each neuron is often tied to a possible label. More specifically, the output layer produces a probability distribution, where the most probable label has the highest probability. Usually, the output is transformed to contain values in [0, 1] while also the sum of the outputs equal

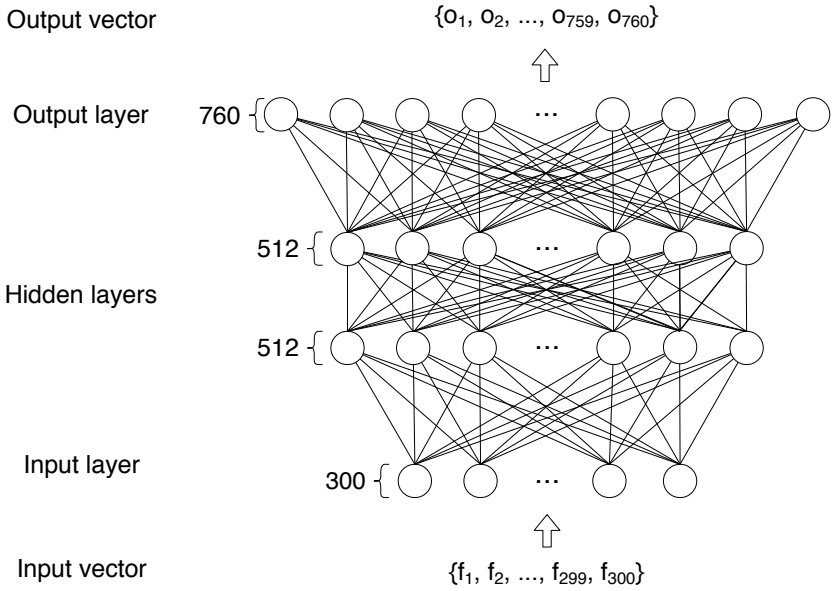

Figure 2.8: A fully connected feed-forward architecture. From the bottom, an 300-dimensional input vector is put onto an input layer, followed by two hidden layers of size 512, and finally an output layer of size 760 that outputs the final output vector of same size.

to 1. An example is the *softmax* function

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}, \tag{2.3}$$

where z is the input vector to the output layer, and j indexes the output units $j = 1, 2, ..., K$. In Fig. 2.8, the output layer and resulting output vector is seen in the top. both with size 760.

Training a Neural Network

In training a NN, the model trains on some input data often named the *training* set. A training set contains labeled data, that is, data where the result is known. In the scope of this thesis, training data are entities or entity descriptions, and their types are labels.

In order to evaluate a model, it is useful to have some data that is *unseen*. Unseen data is never learned by the model, and is used to evaluate a model's performance on data it has never seen before. This data is often a subset removed from the training data itself.

It is vitally important for a model to evaluate on unseen data, in order to prevent a model from *overfitting*. Overfitting is an issue when a model trains on some data and is able to predict that data with good accuracy (up to 100% accuracy). However, that model might then only be able to have good performance on training data, and performance on any unseen data will often suffer.

Dropout is a technique used to alleviate the challenge of overfitting. When inserted between two layers, e.g., input layer and the first hidden layer, a dropout layer will choose to remove some of the input neurons with a probability p. A model will then be less dependent on the full training data, and reduce the overall effect of overfitting.

A model is trained for several *epochs*. An epoch is when a model has processed all training data. The more epochs a model is trained, the more weights and biases are able to update and improve prediction accuracy.

For a model to improve and update weights and biases, a model is provided an *optimizer*. The responsibility of the optimizer is to update these weights and biases in order to minimize a *loss* function. Namely, when predicting data and comparing to the ground truth labels, the model attempts to adjust, or optimize, parameters with the goal to predict those ground truth labels. This operation is called *backpropagation* and is when a model "reverses" through the network, updating parameters along the way. The goal to minimize the error of the model, and is accomplished by changing the model parameters such that it reaches a minimum error value. This value is found by moving down slopes, therefore the name *gradient descent*.

An example optimizer is *stochastic gradient descent* (SGD), and a loss function example is categorical cross entropy. These are more advanced to explain in a related works section on entity typing, therefore we refer to more in-depth resources like [6].

Now that NNs are introduced, we describe distributed representations and how they can be used to feed short entity descriptions.

2.6 Distributed Representations

In order to get an idea of how to use entity descriptions with the intent to feed them into a neural network, we describe the topic of distributed representations.

Representation of data is highly significant in the domain of deep learning and is important in order to get relevant results and good performance [21]. Feature-based data representations have previously been the standard way to provide input for a model to learn. However, these features are often manually created, and therefore can be inefficient, time costly, and end up being tied to the applied domain.

A better way would be for these features to be automatically found so that important factors are extracted, and less significant factors are ignored. This representation learning is highly efficient, and neural networks (NNs) are often used in order to learn those

representations. In the scope of this thesis, we seek to learn representations of entity descriptions, and so a NN approach is employed for that task.

Neural networks can be generalized to have two different representations, (1) *local representation*, and (2) *distributed representation* [22]. In (1), a concept might be represented by one neuron, and one neuron representing a concept. While this local representation approach is easy to understand and implement, it does not support generalization, which is important for a model to be applicable for unseen data. Compared to (2), where a concept is represented by many neurons, and more importantly, the pattern of activity across those neurons. While more complex to understand and implement, the result is a more generalized approach, where the model is able to pick out similar concepts, meaning they have similar representations. In this global, more generalized approach, a model presented with unseen data, even with an unseen concept, might be able to infer something by its similarity to other concepts.

2.6.1 Word Embeddings

A common approach for representing words in a text or corpus is using a highly dimensional, sparse vector. This is the frequency of the word itself, often accompanied by the frequency of the neighboring words in order to get a sense of the context. This high dimensional sparse vector does not computationally do well in neural networks (a vocabulary of 3 million unique words would result in 3 million-dimensional vectors). Instead, it would be favorable to have more generic, dense and low-dimensional vectors, as stated by Goldberg [23]:

> "One of the benefits of using dense and low-dimensional vectors is computational: the majority of neural network toolkits do not play well with very high-dimensional, sparse vectors. [...] The main benefit of the dense representations is generalization power: if we believe some features may provide similar clues, it is worthwhile to provide a representation that is able to capture these similarities."

Word embeddings then, are a type of distributed representation for text, and allows for representing words with similar meaning in a way so that they have similar representations [24]. They result to be rich, dense, low dimensional vectors favorable by neural networks. These word embeddings are learned by a model on some text or corpus, and while there are many different approaches for doing so, we opt for using *word2vec*.

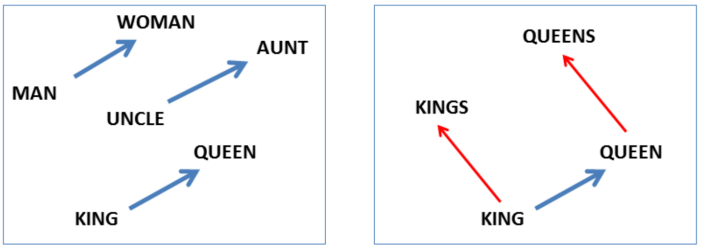

Figure 2.9: Example use of word embeddings from [25]. Left panel highlights relations between genders. Right panel illustrates the same gender relation (blue), but also projecting relations between singular and plural words (red).

Word2Vec

Mikolov et al. [24] presented in 2013 a method to learn word embeddings named *word2vec*. Early on it was an attempt to efficiently use neural networks in order to train word embedding models. Two learning models are introduced for learning word embeddings, a continuous Bag-of-Words (CBOW) model, and a continuous Skip-Gram model. The CBOW model learns the embeddings by predicting the current word based on its context, while the Skip-Gram model predicts the surrounding words of a given word.

There are two ways to make use of word2vec word embeddings, either by training a model on a corpus of text or by retrieving a pre-trained model extensively trained on a popular dataset, like the *Google News* dataset. Alternatively, a pre-trained model can also be used to further learn a new corpus of text.

Some interesting examples from using these vectors, is to compute the similarity or dissimilarity between words. A popular example is to find the word 'queen', by using the set of words 'man', 'woman', 'king'. Given a pre-trained word2vec model v, one can compute $v(\text{'woman'}) + v(\text{'king'}) - v(\text{'man'})$, which returns a set of ranked results, the top result being 'queen'. Another example is when finding the capital of a country, e.g., by finding the capital of China, one can compute $v(\text{'Rome'}) - v(\text{'Italy'}) + v(\text{'China'})$, and get the top result being 'Beijing'. Other examples are available in [25], and one of those are presented in Fig. 2.9.

We have now presented word embeddings, with word2vec being one approach for learning them. By using a word embedding model to represent short entity descriptions, we have rich low-dimensional vectors which we can use for our neural network.

Now that we have established basic information on entities, types, KBs, related entity typing methods, neural networks and distributed representations, we can go ahead and design a new approach. An approach which takes advantage of word embeddings from

[17] https://code.google.com/archive/p/word2vec/

short entity descriptions in order to type entities in KBs with NNs, the end goal to support KBs with incompleteness.

Chapter 3

Approach

In this chapter, we present our approach to designing and implementing a neural network with the goal to predict a single type given an entity and one or several inputs. First, we present the design of our neural network models in Sect. 3.1, followed by how we design our input components in Sect. 3.2. Finally, we present our code implementation in Sect. 3.3.

3.1 Architecture Design

We describe the design of two neural network models for automatically typing entities in a knowledge base.

3.1.1 Design Overview

Our approach is based on a multilayer percepteron (MLP), a simple neural network (NN) architecture consisting on vector representations of entities as inputs, and a softmax operation on the output layer to obtain a probability distribution among all types.

This model is simple yet also flexible to account for combining various input representations possibly of different dimensions, as shown in [16], where a similar architecture is used for fine-grained typing of entity mentions in a corpus.

3.1.2 Architectures

We present two architectures, NeuType1 and NeuType2.

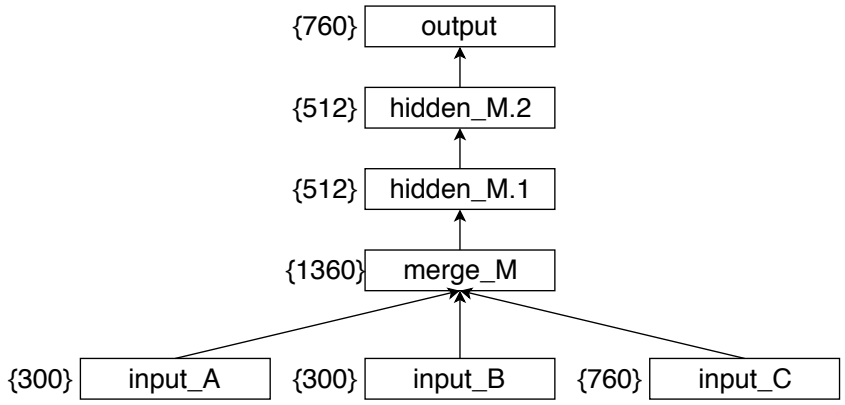

Figure 3.1: NeuType1. Arrows indicate fully-connected layers, and number of nodes in a layer are within brackets.

NeuType1

The first architecture is NeuType1, and is presented in Fig. 3.1. It consists of a fully-connected feedforward neural network, and is able to handle different entity vector representations, which are given by `input_A`, `input_B` and `input_C` input components (cf. Sect. 3.2). A merge layer `merge_M` concatenates the available inputs into hidden layers `hidden_M.1` and `hidden_M.2`. The outputs are transformed by softmax into a probability distribution across all possible 760 type labels in DBpedia ontology.[1] This model resembles the simple learning framework introduced by Le and Mikolov [26], where a neural classifier is applied on a merging of multiple input vectors.

NeuType2

A deeper NN architecture is NeuType2, which is depicted in Fig. 3.2. Unlike in NeuType1, here each input component is firstly fully connected to its own stack of hidden layers. In this way, its depth allows it to capture better each input entity representation before combining them by vector concatenation. Similar deep merging networks have proven to be effective versus another textual inputs composition for classification tasks [27].

3.1.3 Output

When defining the model output, we are interested in finding the single most correct type. We therefore address the problem as a multiclass, single-label classification task, and return the type with highest probability.

[1] We discard the `<owl:Thing>` root type, which is meaningless for our task.

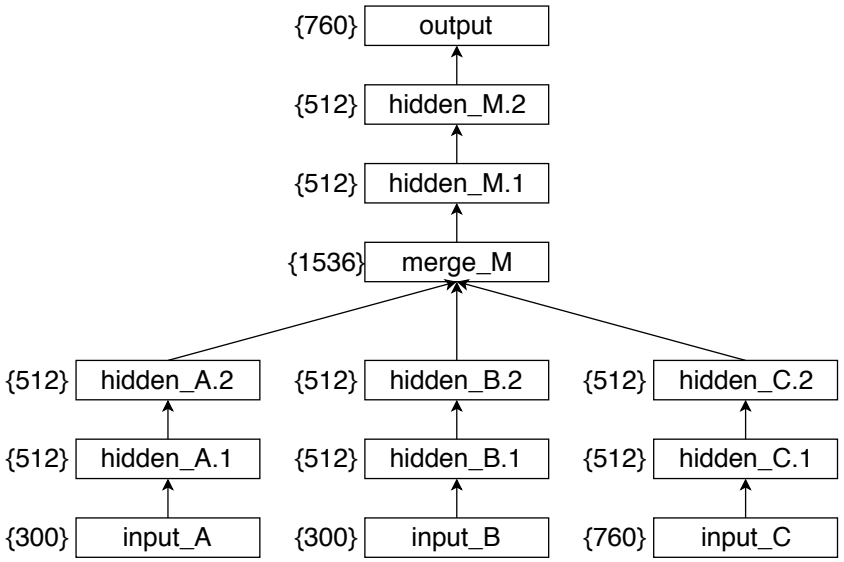

Figure 3.2: NeuType2. Arrows indicate fully-connected layers, and number of nodes in a layer are within brackets.

3.2 Input Components

We consider three input components: $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$. Each of these input vector spaces aim to represent a particular information component of an entity.

Input $\boldsymbol{A}$

Component $\boldsymbol{A}$ is the main input representation, and consists of word embeddings of short entity descriptions. Specifically, for an entity e we retrieve its short description s_e in DBpedia. We then assign to each token w in s_e its 300-dimensional vector $\boldsymbol{x}$ in the *word2vec* pre-trained word embeddings, obtained by the approach presented in [24] on the *Google News* dataset[2] as follows:

$$\boldsymbol{x} = \begin{cases} \boldsymbol{v}_w, & \text{if } w \text{ in } word2vec\_model \\ \boldsymbol{0}, & \text{otherwise.} \end{cases}$$

$\boldsymbol{x}$ then is zero-valued for words not found in the *word2vec* model.

Input $\boldsymbol{A}$ is simply the centroid $\boldsymbol{c}_e$ of these word embeddings for e. This approach for the example entity *Machine_learning* is seen in Fig. 3.3.

[2] *GoogleNews-vectors-negative300.bin.gz* from https://code.google.com/archive/p/word2vec/.

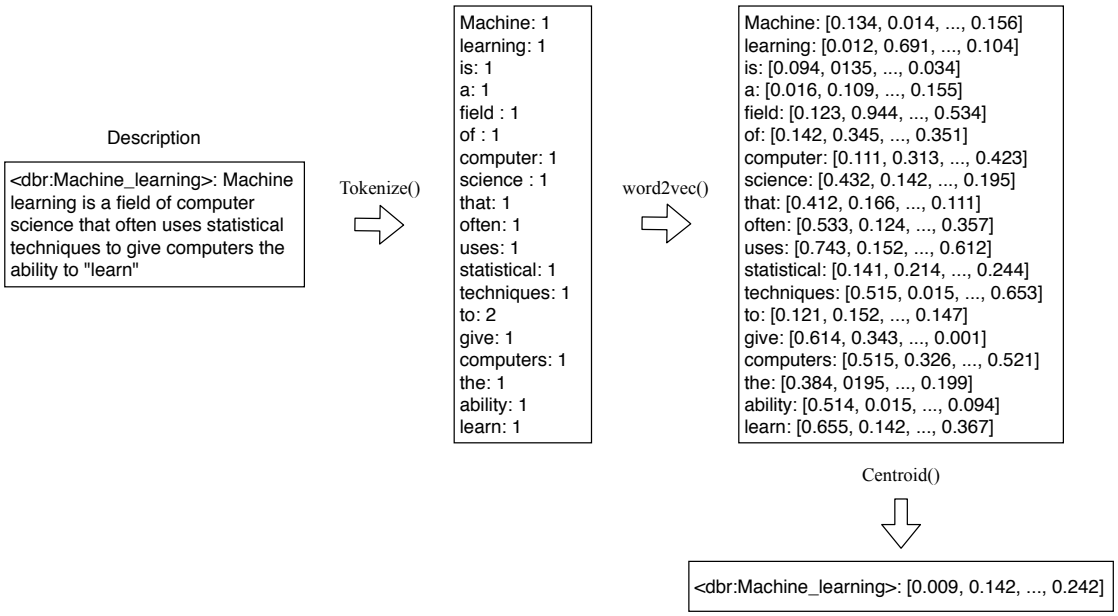

Figure 3.3: Illustration of generating input $\boldsymbol{A}$ for entity *Machine_learning*. Values are random and for representation purposes only.

Input $\boldsymbol{B}$

Component $\boldsymbol{B}$ comprises the short descriptions of the related entities of an entity. Given e, we retrieve the set of related entites R_e, and obtain for each $e' \in R_e$ the centroid c'_e of word embeddings in its short description as before. Some related entities might not have a short description, and therefore we initialize them with zero values:

$$c'_e = \begin{cases} c'_e, & \text{if } e \text{ has a short description} \\ 0, & \text{otherwise.} \end{cases}$$

We define $\boldsymbol{B}$ as the centroid of these related entities' centroid $avg_{e' \in R_e} \boldsymbol{c'_e}$. The generation of this input component for the same entity example is presented in Fig. 3.4.

Input $\boldsymbol{C}$

Finally, component $\boldsymbol{C}$ represents the frequency of the types of related entities. Formally, given an entity e and its related entities R_e, the type frequency vector of related entities is defined as $(f_1, ..., f_n)$ where f_i counts how many entities in R_e are assigned to type t_i, for each of the n labels in the universe of types. Figure 3.5 illustrates the steps for generating input $\boldsymbol{C}$.

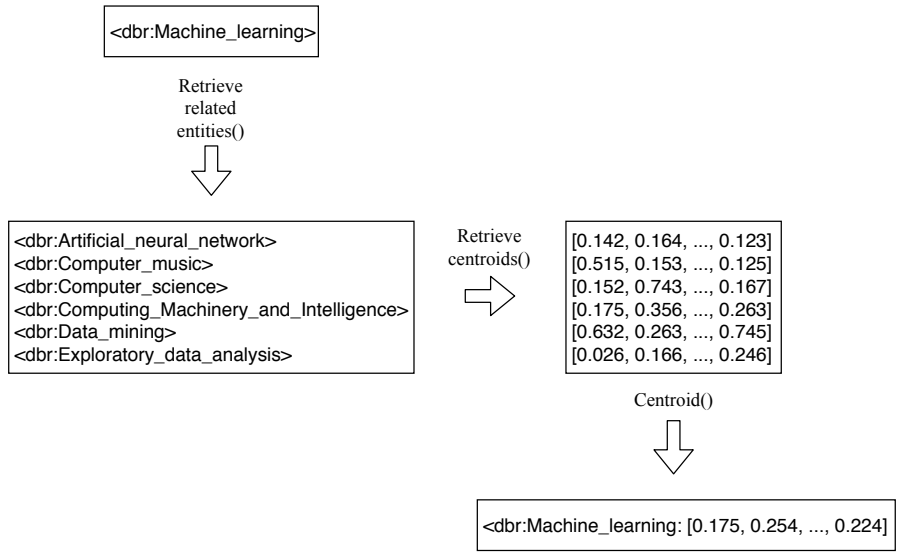

Figure 3.4: Illustration of generating input $\boldsymbol{B}$ for entity *Machine_learning*. Values are random and for representation purposes only.

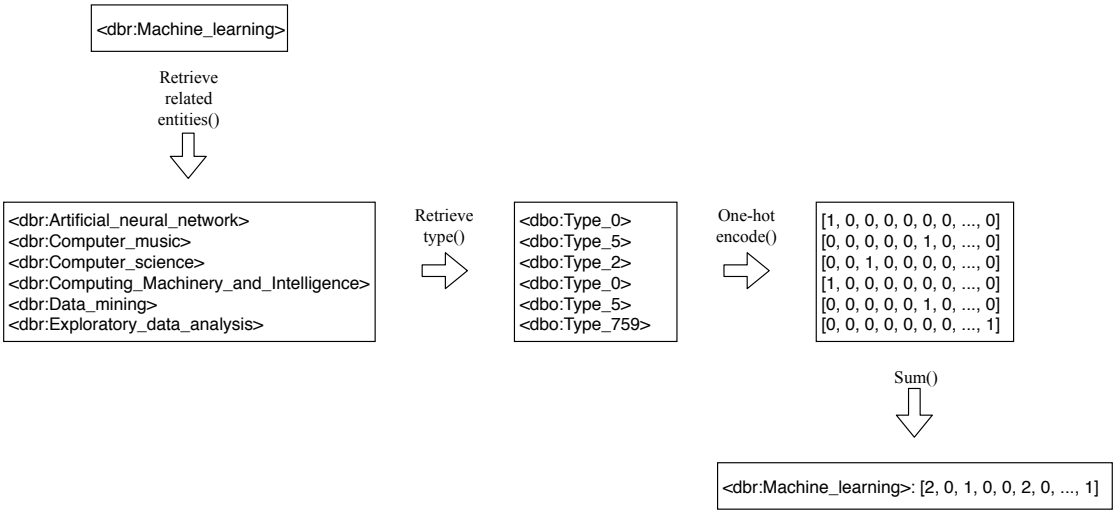

Figure 3.5: Illustration of generating input $\boldsymbol{C}$ for entity *Machine_learning*. Values are random and for representation purposes only.

We denote with plus (+) the fact that more than a single input component is provided to the model. For example, $\boldsymbol{A}$ + $\boldsymbol{B}$ means that $\boldsymbol{A}$ and $\boldsymbol{B}$ are provided, meanwhile $\boldsymbol{C}$ is ignored.

Our inputs then for an example entity is summarized in Table 3.1.

3.3 Neural Network Implementation

In this section we give some insight into the implementation of the NNs themselves. We walk through the code used to train our models, and show how previously saved

Table 3.1: Input summary of components $\boldsymbol{A}$, $\boldsymbol{B}$ and $\boldsymbol{C}$ for entity *Long_Man*. Subscripts denote vector dimensions.

Entity	*Long_Man*
$\boldsymbol{A}$	$[-0.03631253, -0.06072263, 0.0469383, ..., -0.00998038]_{300}$
$\boldsymbol{B}$	$[-0.00652788, -0.0305521, 0.06728973, ..., -0.00559675]_{300}$
$\boldsymbol{C}$	$[0., 0., 2., 0., 0., 1., 0., ..., 0.]_{760}$

models can be loaded in order to produce type predictions. Python version 3.6.5 is used throughout the work of this thesis.

First of all, we load input files using *Pandas*[3] for entity-types assignments, and *Numpy*[4] for our inputs $\boldsymbol{A}$, $\boldsymbol{B}$ and $\boldsymbol{C}$. Pandas is a high-performing data structure and analysis tool that is easy to use, while Numpy is a fundamental package for scientific computing with Python.

Listing 3 Loading individual input files for the neural network model.

```
import pandas as pd
import numpy as np

TRAIN_ent_typ = pd.read_csv(TRAIN_path + "ET.tsv", header=0, sep="\t")
TRAIN_cent_abs = np.loadtxt(TRAIN_path + "ABS.tsv", skiprows=1)
TRAIN_cent_rel = np.loadtxt(TRAIN_path + "REL.tsv", skiprows=1)
TRAIN_typ_freq = np.loadtxt(TRAIN_path + "TYP.tsv", skiprows=1)
```

Listing 3 shows how we opt to load input files for all our inputs. We use Numpy for loading our vectors as it is able to retain the vector shapes without the need for additional post-processing.

Moving on, we encode our types in order for our neural network to use them as labels, i.e., converting them to numbers such that every type maps to a corresponding number. The process is seen in Listing 4. We use *scikit-learn*[5], a powerful tool for data mining and analysis, which has great functionality for encoding and decoding labels. We remove the root node `Thing` as stated earlier, and further motivate in Chap. 4.

With inputs loaded and labels encoded, we further present how models are created. We train and implement our models using *Keras*[6], a high-level API used on top of the machine learning framework *TensorFlow*[7]. We go into greater detail about Keras

[3] https://pandas.pydata.org/
[4] http://www.numpy.org/
[5] http://scikit-learn.org
[6] https://keras.io/
[7] https://www.tensorflow.org/

Listing 4 Label encoding for all available types in the DBpedia ontology.

```
from sklearn.preprocessing import LabelEncoder
from type_taxonomy import TypeTaxonomy

all_types = list(type_taxonomy.get_types_all())
all_types.remove('<owl:Thing>')

encoder = LabelEncoder()
encoder.fit(all_types)

y_train = encoder.transform(TRAIN_ent_typ['Type'])
```

and TensorFlow in Sect. 4.1.5. We demonstrate how we create models NeuType1 and NeuType2 in Listing 5.

After constructing a model, we train it as seen in Listing 6. Early stopping is used, which stops training after no improvement has been seen for a set amount of epochs. However, early stopping only stops training, and does not return the best model, which is why we use `ModelCheckpoint` to store the best performing model.

Listing 7 shows how a model can be loaded in order to produce predictions for a set of entities and their input vectors.

Listing 5 Functions for creating models NeuType1 and NeuType2 using Keras. `get_simple_model_A` corresponds to NeuType1, while `get_complex_model_ABC` corresponds to NeuType2.

```
from keras import losses, optimizers
from keras.models import Model
from keras.layers import Dense, Input, concatenate

def get_simple_model_A():
    input_1 = Input(shape=(300,))

    x = Dense(512, activation='relu')(input_1)
    x = Dense(512, activation='relu')(x)

    output = Dense(num_classes=760, activation='softmax')(x)

    model = Model(input_1, output)

    sgd = optimizers.SGD(lr=0.1)
    model.compile(loss=losses.sparse_categorical_crossentropy,
     ↪  optimizer=sgd, metrics=['accuracy'])
    return model

def get_complex_model_ABC():
    input_1 = Input(shape=(300,))
    input_2 = Input(shape=(300,))
    input_3 = Input(shape=(760,))

    x_1 = Dense(512, activation='relu')(input_1)
    x_1 = Dense(512, activation='relu')(x_1)

    x_2 = Dense(512, activation='relu')(input_2)
    x_2 = Dense(512, activation='relu')(x_2)

    x_3 = Dense(512, activation='relu')(input_3)
    x_3 = Dense(512, activation='relu')(x_3)

    x = concatenate([x_1, x_2, x_3])
    x = Dense(512, activation='relu')(x)
    x = Dense(512, activation='relu')(x)

    output = Dense(760, activation='softmax')(x)

    model = Model([input_1, input_2, input_3], output)

    sgd = optimizers.SGD(lr=0.1)
    model.compile(loss=losses.sparse_categorical_crossentropy,
     ↪  optimizer=sgd, metrics=['accuracy'])
    return model
```

Listing 6 Training a model using early stopping on some input `inputs`, with ground truth labels `y_train`.

```
from keras.callbacks import EarlyStopping, ModelCheckpoint

model_path = "Models/" + config + ".h5"
model = model_fn()

earlystop = EarlyStopping(monitor='val_acc', min_delta=0.0001,
↪  patience=5, mode='auto')
modelcheck = ModelCheckpoint(filepath=model_path, monitor='val_acc',
↪  save_best_only=True)
history = model.fit(inputs, y_train,
                        batch_size=128,
                        epochs=50,
                        callbacks=[earlystop, modelcheck],
                        validation_split=0.1)
```

Listing 7 Predicting labels for a set of entities using a loaded model.

```
from keras.models import load_model

model = load_model(model_path)

text_labels = encoder.classes_

y_A_softmax = model.predict(input_vectors)
predicted_A_labels = [text_labels[np.argmax(i)] for i in y_A_softmax]

preds_A = {}
for x, pred in zip(entities, predicted_A_labels):
    preds_A[x] = pred
```

Chapter 4

Evaluation

This chapter describes how we perform our experiments. First, we describe how to evaluate our methods, and how we process our datasets in Sect. 4.1. We then present our results and analysis in Sect. 4.2.

4.1 Experimental Setup

For evaluating our proposed approach, we present our evaluation metrics in Sect. 4.1.1, where we also show how we use these metrics and how we take into account the ontology itself. Furthermore, in Sect. 4.1.2 we describe how we draw pseudo-random balance test data, and use this method to create two evaluation datasets Dataset 1 and Dataset 2 in Sect. 4.1.3. We describe how we retrieve the background datasets used in this thesis in Sect. 4.1.4, and finally extend our neural network experimentation setup in Sect. 4.1.5.

4.1.1 Evaluation Metrics

In order to evaluate entity-type assignments, we employ a rank-based evaluation for finding the most correct entity-type. Since we are interested in predicting a single entity type, we use normalized Discounted Cumulative Gain at rank 1 (NDCG@1) as our evaluation metric. We consider different ways of computing gain according to a distance $d(t_a, t_g)$ between an assigned entity type t_a and a ground truth type t_g. The motivation is to take into account the hierarchy of types [28]. For example, predicting type `Person` for a correct type `Athlete` is a less severe error than predicting `Scientist`.

Using *strict* scoring, the following strict gain scoring is defined:

$$score = \begin{cases} 1, & \text{if } d(t_a, t_g) = 0 \\ 0, & \text{otherwise.} \end{cases}$$

In this way, if $d(t_a, t_g) = 0$, $t_a = t_g$ and it is rewarded a score of 1. We note that strict scoring is equivalent to classification accuracy.

The strict scoring method has a drawback considering the fact that near misses are not rewarded. If an entity is typed as `Artist` but is typed as `Actor` in the ground truth (meaning a distance of 1 is returned), the typing is considered a miss and rewarded a score of 0 by the strict scoring method.

Instead, in evaluating type assignments in an ontology, it is more interesting to reward near misses with a score between 0 and 1 judging by the distance of the miss. In order to account for this, Balog and Neumayer [28] present two *lenient* scoring methods. The distance function is now turned into a gain measure, where two decay functions are presented, one linear and one exponential. *Linear* gain is defined as

$$G(t) = 1 - \frac{d(t_a, t_g)}{h}, \tag{4.1}$$

where h is the depth of the ontology ($h = 6$ in DBpedia 2016-10). *Exponential* gain, instead, is defined as

$$G(t) = b^{-d(t_a, t_g)}, \tag{4.2}$$

where b is the base of the exponent (set to 2 in this thesis).

Now, a correct type is rewarded with a score of 1, and a miss is scored by the severity of the miss. A score of 0 is assigned if $d(t_0, t_1) = \infty$, meaning the different types are not on the same path, e.g., typing an entity as `Organization` to a `Person`.

However, the lenient scoring methods presented above does not account for sibling types. In the deeper layers of the ontology, some types become less severe to misclassification, therefore we might want to reward a type being typed among the siblings of the ground truth. Hence, a scoring method could reward an exact match, that also rewards descendant/ancestors, and finally rewards a sibling type taking into consideration the type's depth in the ontology. The motivation behind this is to not reward sibling types in the top levels, as they are very different from each other, but reward the deeper, more interchangeable sibling types. Examples showing these sibling types is shown in Fig. 4.1, while an `OfficePerson` is different from `Politician`, the difference is not as large as `Person` and `Organisation`, which are also sibling types. In this case, `OfficePerson` will be rewarded with a higher score than zero, while `Person` will not be rewarded.

Table 4.1: Summary of proposed evaluation metrics. Columns represent whether a type is rewarded if it is an exact match, if it is among the descendants/ancestors, or among sibling types.

Scoring Method	Exact type	Descendant/Ancestor	Siblings
Strict	X		
Lenient	X	X	
w/siblings	X	X	X

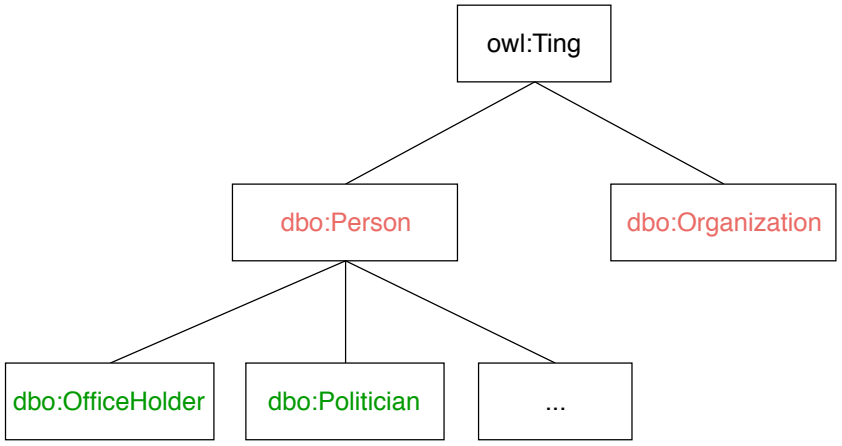

Figure 4.1: Representation of misclassifying types in an ontology tree. The sibling pair in red are distinguishable, and thus should not be rewarded a score if typed as one or the other. However, the sibling pair in green are more interchangeable, and thus could be rewarded for typing in the correct branch.

We do not implement a scoring method considering sibling types in this thesis, as the DBpedia ontology has a height of 6 (disregarding `Thing`), and therefore, the effect of these sibling types would not be very significant. However, it could be a meaningful evaluation metric to use in deeper ontologies like YAGO, where the height is 19 (as presented in Sect. 2.3.4), and the amount of possible types is larger (760 in DBpedia vs. 568,672 in YAGO).

Evaluation Framework

The evaluation framework takes a set of entity type assignments and compares them to a given ground truth. The framework implements the scoring methodology seen in Sect. 4.1.1, and produces a final score reflecting accuracy.

In order to evaluate entity-type pairs when a different scoring method than *strict* is used, the ontology has to be taken into consideration in order to retrieve the path and depth of the types. The ontology is therefore stored as a tree based on the DBpedia ontology.[1] The ontology can be retrieved as a `.nt` file, which we further post-process to

[1] http://wiki.dbpedia.org/downloads-2016-10#dbpedia-ontology

only contain triples with the predicate `rdfs:subClassOf`. The result is a tree structure of the DBpedia ontology, making it possible to retrieve depth, height, leaf nodes, parent nodes and child nodes. An excerpt of the ontology stored by the evaluation framework is shown in Listing 8.

Listing 8 Output example from DBpedia ontology tree consisting of `rdfs:subClassOf` relations. A node is indented when it is a child node.

```
<owl:Thing>
    <dbo:Activity>
        <dbo:Game>
            <dbo:BoardGame>
            <dbo:CardGame>
        <dbo:Sales>
        <dbo:Sport>
            <dbo:Athletics>
            <dbo:Boxing>
        <dbo:Employer>
...
```

4.1.2 Generating Balanced Test Data

In order to evaluate a method's performance, we create a randomly drawn subset of data which seeks to approximate key characteristics of the source data. We create datasets containing N "random" entities. We say "random" seeing that during the work of the thesis, we found that a purely random approach would not be suitable for generating good and fair evaluation data. There is a bias in some of the top-level branches as seen earlier in Table. 2.1. While randomly drawing N entities would reflect this bias, we risk ending up with evaluation data that is mainly contained in these top-level branches, potentially missing information contained in the less populated top-level types. We would then get no indication of how a method performs on different top-level branches, and the evaluation itself end up being biased.

It is in our interest then to draw a random subset of evaluation data that still retains the characteristics of the underlying distribution, while also having a broad representation containing data in as many top-level types as possible. The result is a pseudo-random "balanced" dataset.

With the pseudo-random approach in Algorithm 4.1, we achieve the goal of having a broad top-level representation, while still keeping the characteristics of the distribution. Note that we optionally reserve an entity r. The motivation behind this reservation is to prioritize a rich training set over rich evaluation sets. In other words, if we do not do this reservation, we risk evaluation sets containing all entities assigned to top-level types

Algorithm 4.1 Drawing entities in a pseudo-balanced approach given a set of entities E, set of top-level types T_{top}, total amount to draw N and minimal amount to draw from each top-type M.

Input: E ▷ Set of entities
Input: T_{top} ▷ Set of top-level types
Input: M ▷ Minimum amount to sample from each top-level type
Input: N ▷ Total amount to sample
Output: *sample* ▷ Set of pseudo-random drawn entities

for $t \in T_{top}$ **do**
  $E_t \leftarrow \{e \in E : \text{type of } e \text{ is } t\}$
  if $|E_t| < M$ **then**
    $r \leftarrow \{e : e \in E_t, \text{randomly sampled}\}$ ▷ Reserve r for training
    $E_t \leftarrow E_t \setminus r$
    $sample_t \leftarrow \{e_1, ..., e_{|E_t|} : e_i \in E_t\}$ ▷ Sample the rest for evaluation
  else
    $sample_t \leftarrow \{e_1, ..., e_M : e_i \in E_t, \text{randomly sampled}\}$
  end if
  $sample \leftarrow sample \cup sample_t$
end for
$filler \leftarrow \{e_1, ..., e_{N-|sample|} : e_i \in E, \text{randomly sampled}\}$
$sample \leftarrow sample \cup filler$

(i.e., top-level types with less than 10 assigned entities). Consequently, a model might never see these types in training, therefore producing "impossible" types to predict in an evaluation. By reserving r then, we avoid the situation of impossible types.

Now that we have designed an approach for drawing entities, we set out to define two datasets used for evaluating our model's performance.

4.1.3 Evaluation Datasets

We evaluate performance using two datasets, Dataset 1 and Dataset 2. Dataset 1 is used for evaluating performance on entity-type assignments already in the KB, while Dataset 2 evaluates performance on emerging entities not seen previously by the KB. The specific constraints are listed here:

- **Dataset 1**: We generate this set by drawing entities that have type assignments ("instance types") in DBpedia, using the balanced pseudo-random approach previously described. We apply the following additional constraints for entities:
  1. They must have types predicted by SDType, to facilitate comparison, and
  2. They must have a short description in DBpedia.

Table 4.2: Amount of entities in each corresponding dataset. Data from DBpedia 2016-10 release.

Dataset	Entity-type assignments	Unique entities	Duplicate assignments
Instance types	5,150,432	5,044,223	106,209
SDTyped	2,608,853	2,608,853	0

Each entity is then labeled with a single, most specific type in DBpedia instance types.

- **Dataset 2**: We select emerging entities by picking entities with types in DBpedia Live,[2] such that these entities do not have types in DBpedia instance types. Similarly, we also require that:

  1. Entities have types predicted by SDType, and
  2. Entities have a short abstract in DBpedia Live.

  We use the pseudo-random approach previously described, excepting that the optional reservation of training instances is not applied here.

Next, we describe in detail our approach for retrieving our background datasets instance types and SDTypes.

4.1.4 Background Datasets

Two datasets primarily make up the underlying background data in this thesis. The background datasets are obtained from the DBpedia 2016-10 release,[3] more specifically, the *Instance Types* and *Instance Types Sdtyped Dbo* entity-type assignments. For simplicity, these datasets will be referred to as *Instance Types* and *SDTyped* respectively.

Listing 9 Triple form used in DBpedia.

```
$object rdf:type $class
```

Instance Types dataset is described by DBpedia as triples on the form seen in Listing 9. These triples are produced from DBpedia's mapping-based extraction of Wikipedia and are used to describe entity-type relationships in DBpedia on the form of `rdf:type` triples.

Table 4.2 shows the number of entities contained in each dataset. 5,150,432 entity-type assignments are contained in the Instance Types set, where 5,044,223 of these are unique

[2] Retrieved through SPARQL endpoint http://dbpedia-live.openlinksw.com/sparql.
[3] http://wiki.dbpedia.org/downloads-2016-10

Table 4.3: Amount of entities in each corresponding dataset with `owl:Thing` filtered out. Data from DBpedia 2016-10 release.

Dataset	# of entity-type assigments	Without `owl:Thing`
Instance types	5,044,223	4,767,652
SDTyped	2,608,853	2,608,853

entities. There are 106,209 assignments which are filtered out as a result of entities having more than one typing assignment. An example of these typing assignments is seen in Listing 10. In deciding which of these type pairs to keep, we see that they are all on the same path and keep the most specific type. Meaning that in Listing 10 `MotorSportsSeason` is kept.

Listing 10 Example of an entity `Aston_martin__1931__1` with several types found in *instance types*.

```
<dbr:Aston_Martin__1931__1> <rdf:type> <dbo:MotorsportSeason>
<dbr:Aston_Martin__1931__1> <rdf:type> <dbo:SportsSeason>
```

The **SDTyped** dataset is described by DBpedia as follows:[4]

> "The SDType heuristic can extract probable type information in large, cross-domain databases on noisy data. This is its result for DBpedia which supplements the normally gathered instance types. This set is its result for DBpedia where the inferred type has an equivalent in the DBpedia ontology."

The set contains 2,608,853 entity-type assignments, each with a single assigned type. The triples are presented on the same form as in Instance Types. This dataset is highly relevant as it makes it possible to compare and evaluate SDType against our methods.

Instance Types dataset includes entity-type assignments where an entity is typed as `Thing`. This type information does not tell anything specific about an entity, which is further reinforced by the ontology where `Thing` is the root node. Every type then is a descendant of `Thing`, and entity-type pairs containing `Thing` are therefore excluded. Table 4.3 shows the result of this filtering; the amount of entity-type assignments in Instance Types are now reduced to 4,767,652. SDTyped remains unchanged as there are no entities typed to `Thing`.

We further produce the following datasets, Table 4.4 highlights the main characteristics of them. First, the dataset *Intersection* is produced, containing the intersection of Instance

[4] http://wiki.dbpedia.org/services-resources/documentation/datasets#InstanceTypesSdtypedDbo

Table 4.4: Datasets produced and used by this thesis. Dom_{ins} is the domain of entities in Instance Types, and Dom_{sdt} is the domain of entities in SDTyped.

Dataset	Set operation	Entities
Intersection	$Dom_{ins} \cap Dom_{sdt}$	2,165,602
SDTonly	$Dom_{sdt} - Dom_{ins}$	443,251
SDTonly typed live	$SDTonly \exists$ type in dbplive	12,936

Table 4.5: Comparing datasets with the added requirement that an entity must have a short abstract in *Short Abstracts*, Dom_{sa} is the set of entities in *Short Abstracts*.

Dataset	Previous count	With abstract
Dom_{sa}	-	4,935,272
Intersection $\cap Dom_{sa}$	2,165,602	846,459
SDTonly typed live $\cap Dom_{sa}$	12,936	11,021
Instance Types $\cap Dom_{sa}$	4,766,652	3,047,794

Types and SDTyped, i.e., the 2,165,602 entities that both datasets have in common. Second is *SDTonly*, this is the set of SDTyped entities that do not appear in Instance Types, therefore making a set which is uniquely typed by SDType. *SDTonly typed live* is a dataset containing entities typed by SDTonly, that has since the DBpedia 2016-10 release gotten a type on DBpedia live.[5]

We have now explained the process of retrieving entity-type assignments themselves. Moving on, we describe the procedure to retrieve short entity descriptions.

Abstract Datasets

We have earlier described our intent to use short entity descriptions. They are retrieved from *Short Abstracts* DBpedia dataset.[6] *Short Abstracts* consists of entity-abstract pairs where each abstract has roughly 600 characters from that entity's abstract section on Wikipedia.

Before we can process the short abstracts, we will have to map the entities in the short abstracts dataset to our existing datasets in Sect. 4.1.4. Recall Table 4.4, where entities were constrained to not have type `Thing`. We now have an additional constraint, namely that each of these entity-type assignments must have a short abstract in *Short Abstracts* dataset. Table 4.5 presents the result of this added constraint, and while there is a large reduction of entities in *Intersection* and *Train*, we shall see that this added constraint actually helps to clean up the data.

[5] Using DBpedia SPARQL endpoint `http://dbpedia-live.openlinksw.com/sparql` extracted at 06.03.18

[6] *Short Abstracts* from `https://wiki.dbpedia.org/downloads-2016-10`

The reduction of entities seen in Table 4.5 is substantial, and highlights some characteristics of the DBpedia *Instance Types* dataset. Recall from Sect. 4.1.4 that *Intersection* is a product of that dataset. Let e_{ins} be the set of entities in *Instance Types*, and e_{abs} be the set of entities with an abstract in *Short Abstracts*. We list some reasons for the reduction in amounts of entities:

- Entities in e_{ins} might not have a corresponding abstract in e_{abs}, or the abstract might be in another language.
- We do not account for redirect pages, meaning that an entity in e_{ins} might map to an entity in e_{abs} through redirection.
- Many entities in e_{ins} are typed as `CareerStation`, which is like a period of time in the career of the entity. An example is a football player, who would contain several entries representing the player's transfer from one club to another; see example in Listing 11.
- About 10 abstracts in Dom_{abs} are empty.

Listing 11 Entity-type assignments found in *Instance Types* where a person might have several entities for different stages in their career.

```
<dbr:Ander_Bardají> <rdf:type>  <dbo:SoccerPlayer>
<dbr:Ander_Bardají__1>  <rdf:type>  <dbo:CareerStation>
<dbr:Ander_Bardají__2>  <rdf:type>  <dbo:CareerStation>
<dbr:Ander_Bardají__3>  <rdf:type>  <dbo:CareerStation>
<dbr:Ander_Bardají__4>  <rdf:type>  <dbo:CareerStation>
<dbr:Ander_Bardají__5>  <rdf:type>  <dbo:CareerStation>
```

To summarize, we have now produced three different datasets containing entity-type assignments which all have a corresponding abstract. Though we do lose potential training data, the resulting data is cleaner as each entity is mapped to a type with a rich description attached. We summarize and list key characteristics of each dataset here:

- Intersection is the set of entities in both Dom_{ins} and Dom_{sdt}, meaning that we can get an *instance type* and a *SDType* for each entity.
  - *Dataset 1* consists of 1,000 pseudo-randomly drawn from *Intersection* used for evaluation.
- SDTonly typed live contains entities typed by *SDType* which does not have a *instance type* in the DBpedia 2016-10 release, but has since gotten a *live type* retrieved from DBpedia live.

Table 4.6: Statistics on the *Related Entities* dataset.

# of page links	Min	Max	Avg
12,843,699	1	9,441	14.29

  - *Dataset 2*: consists of 1,000 pseudo-randomly drawn from *SDTonly typed live* also used for evaluation.

- *Train*, which is our training data, contain all entities in $Dom_{ins} \cup Dom_{sa}$ except for the entities in Dataset 1.

We now have data required for input component $\boldsymbol{A}$ through abstract datasets. However, inputs $\boldsymbol{B}$ and $\boldsymbol{C}$ are dependent on knowledge of how entities are related to each other. We therefore further describe how we retrieve these relations in the *related entities dataset*.

Related Entities Dataset

Entities on Wikipedia are connected to each other through links on a given entity's Wikipedia page. These links are extracted to DBpedia and is available in the *Page Links* dataset[7]. The dataset is described on DBpedia as follows:

> "Dataset containing internal links between DBpedia instances. The dataset was created from the internal links between Wikipedia articles. The dataset might be useful for structural analysis, data mining or for ranking DBpedia instances using Page Rank or similar algorithms."

We parse the dataset, storing information about entity relationships in a dictionary, i.e., $x : \{y\}$. where y are the related entities of entity x as seen in Listing 12. We present some basic statistics in Table 4.6. Here, we see that on average there are 14 links between entities, therefore providing an acceptable amount of relationship data.

Listing 12 Example related entities after parsing the DBpedia *Page Links* dataset.

```
"<dbr:Vikings>":    ['<dbr:Norwegian_language>',
                     '<dbr:Swedish_language>',
                     '<dbr:Old_Norse>',
                     '<dbr:Norsemen>',
                     '<dbr:Old_Norse_language>',
                     '<dbr:Raid_(military)>',
                     ...]
```

[7] *Page Links* from https://wiki.dbpedia.org/downloads-2016-10

Table 4.7: Computer specification used for training models.

Component	
CPU	Intel Xeon CPU E5-2650 v4 @ 2.20GHz
GPU	NVIDIA Tesla P100 12GB
RAM	256GB

The result is a dataset we can use to get information about relationships between entities. Combined with the abstract dataset, we have the required data for input component $\boldsymbol{B}$, where we want to describe related entities using their short abstracts. Next, we briefly describe the details of retrieving type frequencies for input $\boldsymbol{C}$.

Type Frequencies Dataset

Analyzing the distribution of related entity types might give an indication to the type of the entity in question. Inspired by SDType's approach in Sect. 2.4.2, if an entity x is connected to several other entities typed having type `Place`, x probably is a `Place` or `Location` itself.

The type frequency vector seen in input component $\boldsymbol{C}$ then is simply counting, for each entity, the number of types the related entities are assigned to. These types are retrieved from *Instance Types* dataset.

4.1.5 Experimenting with Neural Networks

Setup

All instances of model training in this thesis are done on a computer located at the University of Stavanger (UiS). The specifications of this computer are listed in Table 4.7. However, these specifications are not required for training these models, as any modern desktop computer with a dedicated graphics processing unit (GPU) should be able to perform the same job. The only limitations are that the GPU should have at least 6GB of random access memory (RAM) in order to store the models and the batch of input data. Furthermore, if the input data to the model is to fit in main memory, the main memory size should reflect this (all input data found to be about 32GB in this thesis), otherwise, one can implement batching and read data from the hard drive instead.

We present the average training time for models NeuType1 $\boldsymbol{A}$ and NeuType2 $\boldsymbol{A} + \boldsymbol{B} + \boldsymbol{C}$ in Table 4.8.

Table 4.8: Training time per model. E is the average amount of epochs and T is average time required.

Model	E	T/E	T_{total}
NeuType1 $\boldsymbol{A}$	37.8	110s	1h 9m
NeuType2 $\boldsymbol{A}$ + $\boldsymbol{B}$ + $\boldsymbol{C}$	20.4	170s	57m

Training is done using Keras[8] [29], a high-level API capable of running on top of several machine learning frameworks like TensorFlow and Theano. Keras focuses on simplicity, and its mission is to go from idea to result with the least possible delay. For this thesis, we use Keras version 2.1.6 running on top of TensorFlow.

TensorFlow[9] is an open-source machine learning framework which is flexible and has high performance for numerical computations. It was originally designed and developed by the Google Brain team within the Google AI organization. We use TensorFlow version 1.8.

Input for our models are stored as individual files per input for flexibility and memory management i.e., input $\boldsymbol{A}$ is stored in one file and input $\boldsymbol{B}$ is stored in another file. These input files can then be read and processed when they are needed. The files themselves contain rows where each row corresponds to the entity the row data represents.

Furthermore, we use a validation split of 0.1, meaning that 10% of the training data is used for evaluating the model during training. This is not to be confused with the evaluation datasets produced in Chap. 4.

Parameter Search

We optimize NeuType2 with all three input components, by experimenting for several parameter settings as follows. We do a learning rate sweep search in $\{10^{-k} : k \in 1, 2, 3\}$, and a finer search in the interval [0.05, 0.5] with 0.05-long steps. Results from both learning rate searches are presented in Figs. A.2 and A.3.

We try different optimizers, specifically stochastic gradient descent (SGD), SGD with momentum, and Adam. Findings are presented in Fig. A.4.

Furthermore, we test adding dropout in three network positions (before `hidden_M.1`, after `hidden_M.2`, between `hidden_M.1` and `hidden_M.2`), each using probability $p \in \{0.2, 0.4\}$. Results from dropout positioning are seen in Fig A.1. Finally, we search for hidden layer

[8] https://keras.io/
[9] https://www.tensorflow.org/

size $h \in \{128, 256, 512, 768, 1024\}$, all with ReLU activation nodes. Categorical cross entropy is used as loss function.

Parameter Settings

We found the following parameter settings to perform best: SGD optimizer with a learning rate of 0.1, and no dropout. All hidden layers have the size set to 512. We then train all the models for both architectures using these parameter settings.

For inputs different than $\boldsymbol{A} + \boldsymbol{B} + \boldsymbol{C}$, we ignore the missing input portion(s) from the model. As an example, in input configuration $\boldsymbol{A} + \boldsymbol{B}$, we remove `input_C` in Fig. 3.1, and remove `input_C`, `hidden_C.1`, and `hidden_C.2` in Fig. 3.2.

4.2 Results and Analysis

4.2.1 Main Results and Evaluation

With our experiments, we seek to answer the earlier mentioned research questions (RQs), we repeat them here in more detail:

- **RQ1**: Can a neural approach, using only entity descriptions, outperform the current state of the art (SDType), which is based on heuristic link-based type inference?
- **RQ2**: Can entity relationship information contribute to type prediction?
- **RQ3**: Which of the two proposed neural architectures (NeuType1 vs. NeuType2) perform better?

Table 4.9 presents the main results. We evaluate both NeuType1 and NeuType2 using all combinations of input components. Scores reported for each neural model are averaged from 5 independent training sessions. In each session, a model is trained for a maximum of 50 epochs, with early stopping implemented in order to prevent overfitting. Early stopping is configured to stop training when no improvement is observed for 5 consecutive epochs.

We test statistical significance against the baseline using a two-tailed paired t-test at $p < 0.05$ and $p < 0.001$, denoted by †and ‡, respectively. We also test statistical significance of each model in NeuType2 versus the corresponding one in NeuType1, using again a two-tailed paired t-test at $p < 0.05$ and $p < 0.001$, denoted by $^{\square}$ and $^{\lozenge}$, respectively.

Table 4.9: Entity typing results. *s* denotes the standard deviation for the averaged performances of the neural models.

Model	Dataset 1						Dataset 2					
	Strict		Linear		Exponential		Strict		Linear		Exponential	
	NDCG@1	*s*	NDCG@1	*s*	NDCG@1	*s*	NDCG@1	*s*	NDCG@1	*s*	NDCG@1	*s*
SDType	0.8020	-	0.8562	-	0.8331	-	0.6970	-	0.7873	-	0.7451	-
NeuType1												
$\boldsymbol{A}$	0.8578‡	0.0025	0.8980‡	0.0028	0.8796‡	0.0028	**0.7870**‡	0.0039	**0.8617**‡	0.0045	**0.8272**‡	0.0042
$\boldsymbol{B}$	0.7722	0.0051	0.8028	0.0049	0.7895	0.0047	0.3664	0.0051	0.5381	0.0033	0.4419	0.0039
$\boldsymbol{C}$	0.7222	0.0050	0.7571	0.0043	0.7414	0.0048	0.2950	0.0054	0.3781	0.0044	0.3341	0.0045
$\boldsymbol{A}+\boldsymbol{B}$	**0.8864**‡	0.0055	**0.9193**‡	0.0035	**0.9045**‡	0.0043	0.7700‡	0.0049	0.8558‡	0.0042	0.8164‡	0.0040
$\boldsymbol{A}+\boldsymbol{C}$	0.8766‡	0.0057	0.9072‡	0.0066	0.8935‡	0.0063	0.7532‡	0.0092	0.8355‡	0.0074	0.7974‡	0.0080
$\boldsymbol{B}+\boldsymbol{C}$	0.7828	0.0098	0.8157	0.0098	0.8006	0.0096	0.3756	0.0181	0.5251	0.0718	0.4430	0.0406
$\boldsymbol{A}+\boldsymbol{B}+\boldsymbol{C}$	0.8748‡	0.0090	0.9074‡	0.0091	0.8930‡	0.0091	0.7462‡	0.0058	0.8354‡	0.0030	0.7944‡	0.0035
NeuType2												
$\boldsymbol{A}$	0.8558‡	0.0029	0.8956‡	0.0028	0.8777‡	0.0026	**0.7816**‡	0.0050	0.8587‡	0.0057	**0.8230**‡	0.0056
$\boldsymbol{B}$	0.7788	0.0050	0.8070	0.0050	0.7947	0.0049	0.3696	0.0130	0.5453	0.0116	0.4475	0.0125
$\boldsymbol{C}$	0.7224	0.0060	0.7586	0.0066	0.7424	0.0061	0.2854	0.0060	0.3690	0.0055	0.3244	0.0055
$\boldsymbol{A}+\boldsymbol{B}$	0.8896‡	0.0026	0.9219‡	0.0030	0.9074‡	0.0025	0.7766‡	0.0057	**0.8600**‡	0.0041	0.8216‡	0.0046
$\boldsymbol{A}+\boldsymbol{C}$	0.8926$^{\ddagger\diamond}$	0.0046	0.9256$^{\ddagger\diamond}$	0.0034	0.9108$^{\ddagger\diamond}$	0.0034	0.7670$^{\ddagger\square}$	0.0068	0.8490$^{\ddagger\square}$	0.0047	0.8107$^{\ddagger\diamond}$	0.0055
$\boldsymbol{B}+\boldsymbol{C}$	0.8134$^{\diamond}$	0.0068	0.8431$^{\diamond}$	0.0068	0.8299$^{\diamond}$	0.0068	0.3802	0.0242	0.5286	0.0777	0.4470	0.0464
$\boldsymbol{A}+\boldsymbol{B}+\boldsymbol{C}$	**0.8958**$^{\ddagger\diamond}$	0.0027	**0.9284**$^{\ddagger\diamond}$	0.0033	**0.9138**$^{\ddagger\diamond}$	0.0026	0.7556‡	0.0108	0.8487$^{\ddagger\diamond}$	0.0076	0.8056$^{\ddagger\diamond}$	0.0092

In answering our RQs, we refer to our main results in Table 4.9.

RQ1. For answering our first research question, we compare input configuration $\boldsymbol{A}$ in both NeuType1 and NeuType2 against the baseline method SDType. In both architectures, it is clear that the neural approach using short entity descriptions is able to outperform the baseline method in both Dataset 1 and Dataset 2 across all evaluation measures.

RQ2. Our second research question considers the effect of adding optional inputs $\boldsymbol{B}$ and $\boldsymbol{C}$. Observing results on Dataset 1, performances improve in both architectures when including optional inputs. Specifically, in NeuType1, using inputs $\boldsymbol{A}+\boldsymbol{B}$ is the best performing model, thus proving that the neural transformations properly capture both components of similar structure (i.e., centroids of word embeddings). In NeuType2, configuration $\boldsymbol{A}+\boldsymbol{B}+\boldsymbol{C}$ has the highest performance, as another evidence of the benefits of representations of related entities. When comparing NeuType1 to NeuType2, the additional hidden layers per input rewards the optional inputs significantly.

In evaluating the addition of optional inputs on Dataset 2, we see that these actually deteriorate performance compared to using only short entity descriptions. Recall that this dataset represents emerging entities, and therefore an entity might not have enough relationships compared to entities in Dataset 1. Consequently, inputs $\boldsymbol{B}$ and $\boldsymbol{C}$ suffer, and does not contain the same rich data as in Dataset 1.

RQ3. To answer the final research question, we compare NeuType1 to NeuType2. In Dataset 1, input $\boldsymbol{A}+\boldsymbol{B}+\boldsymbol{C}$ in NeuType2 has a slightly better score than $\boldsymbol{A}+\boldsymbol{B}$ in NeuType1. It is also interesting to note that NeuType2 almost scores just as high using only $\boldsymbol{A}+\boldsymbol{C}$. On the other hand, when comparing on Dataset 2 it is clear that short entity descriptions from $\boldsymbol{A}$ are more valuable in the case of sparse relationship data in both $\boldsymbol{B}$

Table 4.10: Confusion matrix comparing entity-type assignments using NeuType1 with input ***A*** against SDType.

		SDType	
		Correct	Incorrect
NeuType1	Correct	721	140
	Incorrect	81	**58**

and ***C***. Here, NeuType1 provides mostly almost identical scores to NeuType2, and thus NeuType1 is preferable when considering time and resources required for training the model.

The only configurations performing worse than the baseline, are all configurations not including input ***A***, which further confirms the importance of short entity descriptions of the entities themselves.

4.2.2 Analyzing Predictions

Beyond the RQs, we provide some analysis on predictions themselves. Specifically, we want to investigate the cases where both SDType and our method were unable to produce a correct type. The motivation for this is to see if there are any patterns to incorrect predictions, e.g., is the entity's typing ambiguous, or are the ground truth labels at fault?

In order to present predictions from our model, we use the best performing one out of the 5 averaged models presented for each configuration in Table 4.9. Unless stated otherwise, all following results use best performing model when referring to NeuType1 or NeuType2. In addition, for the following results, we base our evaluation solely on Dataset 1.

We compare type predictions from NeuType1 using input ***A*** to SDType in Table 4.10. We indicate that a prediction is correct by comparing it to the ground truth using strict evaluation. A prediction is incorrect if the predicted type is different from the ground truth.

Table 4.11 shows a selection of 20 entities and their predicted types which both methods predicted incorrectly out of a total of 58 incorrect predictions in Table 4.10. Predictions are produced by NeuType1 using input ***A*** and SDType and ground truth types are from instance types dataset. We note some interesting observations. Many of these incorrect predictions are rewarded by lenient gain measures, i.e., `Politician` in ground truth (GT) and either SDType or NeuType1 predicting `Person`. For example, *Mark Penn* is typed as `Politician` in GT, though looking up on Wikipedia[10] we see that he is more a political strategist, pollster, and author. Therefore, it can be seen that no specific

[10] https://en.wikipedia.org/wiki/Mark_Penn

Table 4.11: Comparing 20 entity-type assignments where both NeuType1 using input $\boldsymbol{A}$ and SDType produced an incorrect type. All 58 incorrect predictions are presented in Table A.1.

Entity	GT	SDType	NeuType1
Afterschool_Alliance	Non-ProfitOrganisation	PersonFunction	Organisation
Charles_L._Lewis_(California_politician)	Politician	Person	OfficeHolder
Cremaster_muscle	Muscle	AnatomicalStructure	AnatomicalStructure
David_Christopherson	MemberOfParliament	TimePeriod	Politician
Earth_Day	Holiday	GivenName	Convention
Edgar,_King_of_Scotland	Monarch	PersonFunction	Royalty
Eustachy_Erazm_Sanguszko	Noble	Person	MilitaryPerson
Fernando_de_Santiago_y_Díaz	PrimeMinister	OfficeHolder	Person
Gordon_Davidson_(politician)	MemberOfParliament	TimePeriod	Politician
Group_Voyagers	BusCompany	PersonFunction	Company
Gudensberg	Town	Settlement	Settlement
Isaac_Komnenos_(son_of_John_II)	Noble	Royalty	Monarch
Lateral_cricoarytenoid_muscle	Muscle	AnatomicalStructure	Nerve
Lena_Videkull	SoccerManager	SoccerPlayer	SoccerPlayer
Lyman_Trumbull	Senator	OfficeHolder	OfficeHolder
Mark_Penn	Politician	Person	Person
Orange_County_Ramblers	CanadianFootballTeam	SoccerClubSeason	NationalFootballLeagueSeason
Shamsuddin_Ilyas_Shah	Royalty	PersonFunction	Country
SoulPancake	Website	Book	Company
University_of_St_Andrews_Athletic_Union	University	Sport	Organisation
William_W._Woollcott	BusinessPerson	Person	Person

Table 4.12: Confusion matrix comparing entity-type assignments using NeuType2 with input $\boldsymbol{A} + \boldsymbol{B} + \boldsymbol{C}$ against NeuType1 with input $\boldsymbol{A}$.

		NeuType1	
		Correct	Incorrect
NeuType2	Correct	828	72
	Incorrect	33	**67**

type can realistically be given, thus `Person` is predicted. This observation is seen for numerous GT types e.g., `Politician`, `Royalty`, `Noble`, all sub-types of `Person`. These observations confirm that entities are ambiguous in nature and that sometimes, no single type can easily describe the entity. The same type ambiguity is evident in types `Town` and `Settlement`.

Table 4.12 shows another confusion matrix, only now we compare NeuType1 with input $\boldsymbol{A}$ to NeuType2 $\boldsymbol{A} + \boldsymbol{B} + \boldsymbol{C}$. We take a look at entity-type pair predictions where both NeuType1 and NeuType2 were incorrect. There are 67 entities both methods typed incorrectly, 20 of which are presented in Table 4.13. Here, we see both methods unable to predict types like `Enzyme` and `Muscle`, producing types close in the ontology instead, either sibling types or parent types as seen previously. However, let us focus on predictions where the GT label is another category entirely.

First, when looking at *SoulPancake*, GT has `Website` as the most specific type, however, looking it up on Wikipedia[11] it is more appropriately a `Company`, which our methods both predicted correctly.

[11] https://en.wikipedia.org/wiki/SoulPancake

Table 4.13: Comparing 20 entity-type assignments where both NeuType2 with input $\boldsymbol{A} + \boldsymbol{B} + \boldsymbol{C}$ and NeuType1 with input $\boldsymbol{A}$ produced an incorrect type. All 67 predictions presented in Table A.2.

Entity	GT	NeuType1	NeuType2
Afterschool_Alliance	Non-ProfitOrganisation	Organisation	Organisation
Chopped_and_screwed	MusicGenre	Single	Single
Cremaster_muscle	Muscle	AnatomicalStructure	AnatomicalStructure
Cyclooxygenase	Enzyme	Drug	Drug
Earth_Day	Holiday	Convention	Convention
Edgar,_King_of_Scotland	Monarch	Royalty	Royalty
Feldgrau	Colour	MilitaryUnit	MilitaryUnit
Group_Voyagers	BusCompany	Company	Company
Gudensberg	Town	Settlement	Settlement
Ice_hockey_at_the_2014_Winter_Olympics_–_...	OlympicEvent	IceHockeyLeague	OlympicResult
Lateral_cricoarytenoid_muscle	Muscle	Nerve	Nerve
Oktoberfest_celebrations	Holiday	Convention	Convention
Orange_County_Ramblers	CanadianFootballTeam	NationalFootballLeagueSeason	FootballMatch
Polynucleotide_phosphorylase	Enzyme	Protein	Protein
Protein_kinase_A	Enzyme	Protein	Protein
Ribonuclease	Protein	Enzyme	Enzyme
Shamsuddin_Ilyas_Shah	Royalty	Country	Monarch
SoulPancake	Website	Company	Company
Testosterone	Drug	Disease	AnatomicalStructure
Tuvalu_at_the_Pacific_Games	Event	SportsEvent	SoccerTournament
University_of_St_Andrews_Athletic_Union	University	Organisation	Organisation

Another interesting example is `Feldgrau`, which GT types as `Colour`, and is indeed a color used on military uniforms. It is no wonder then, that our method misclassify this as *MilitaryUnit*. Let us take a look at the short-entity description of *Feldgrau*:[12]

> "Feldgrau (English: field-grey, also field grey) has been the official basic color of military uniforms of the German armed forces since the early 20th century until 1945 or 1989 respectively. [...]. Metaphorically, feldgrau used to refer to the armies of Germany (the Imperial German Army and the Heer [en: ground forces, or army] component of the Reichswehr and the Wehrmacht)."

The most interesting thing to note is the latter part of the description, that it is metaphorically used to refer to a type of army. Therefore, while not managing to correctly type this entity as `Colour`, we see that it is not exactly wrong either for typing it as `MilitaryUnit`.

A final example to take out of Table 4.13 is *Testosterone*. The GT label for this is `Drug`, which strictly is incorrect. Thus showing that while our evaluation GT labels are mostly good, there are edge cases where they are incorrect. Neither NeuType1 nor NeuType2 are able to infer a better type with `Disease` and `AnatomicalStructure` respectively, though NeuType2 is slightly more correct and closer to the most correct type `Hormone`.

[12] https://en.wikipedia.org/wiki/Feldgrau

Table 4.14: Presenting top-level types according to the ground truth in Dataset 1. SDType and NeuType1 using input $\boldsymbol{A}$ is included showing their accuracy to predict the corresponding top-level type using strict evaluation. The top-level type distribution originally in Instance Types is included for in *InsTypes*.

Top type	acc. SDType	acc. NeuType1	% in Dataset1	% in InsTypes
Person	0.74	0.80	30.90	26.10
Work	0.97	0.91	16.80	10.40
Place	0.70	0.86	14.50	17.60
Organisation	0.86	0.88	11.10	5.90
Species	0.49	0.96	4.50	6.42
Event	0.96	0.89	2.70	1.59
SportsSeason	0.22	0.94	1.80	1.16
MeanOfTransportation	0.87	1.00	1.50	1.19
Holiday	0.73	0.80	1.50	0.02
TopicalConcept	1.00	0.79	1.40	0.03
EthnicGroup	1.00	1.00	1.20	0.11
Disease	1.00	0.83	1.20	0.12
Language	1.00	0.83	1.20	0.19
Food	1.00	0.91	1.10	0.12
ChemicalSubstance	0.55	0.82	1.10	0.37
Activity	0.90	0.90	1.00	0.22
AnatomicalStructure	0.70	0.70	1.00	0.09
Biomolecule	1.00	0.60	1.00	0.16
Device	1.00	1.00	1.00	0.50
SportCompetitionResult	1.00	1.00	1.00	0.02
Currency	1.00	0.90	1.00	0.01
Colour	1.00	0.89	0.90	0.02
Name	1.00	1.00	0.60	0.09

4.2.3 Top-level Accuracy

For the following tables and figures, we compare top-level type predictions between SDType and NeuType1 using input $\boldsymbol{A}$. First, we present Table 4.14, where we compare each method's ability to predict types in each top-level branch using strict evaluation. The top-level type distribution of Dataset 1 is also presented.

Comparing top-level distribution between the original Instance Types dataset and Dataset 1, we see that the problem of typing *Testosterone* can be attributed to the fact that there simply is not enough training data (`AnatomicalStructure` accounts for less than 0.1% of the training data/Instance Types). Consequently, we see a trend in most of the top-level types with a low population where SDType has better performance.

Following up is Fig. 4.2, where we plot the difference in top-level accuracy between NeuType1 using input $\boldsymbol{A}$ and SDType using strict evaluation metric.

We want to see in which top-level types our methods perform better, and in which cases SDType has better accuracy. This is better represented in Fig. 4.2, where the difference in accuracy is plotted for each top-level type. We can see that SDType has better

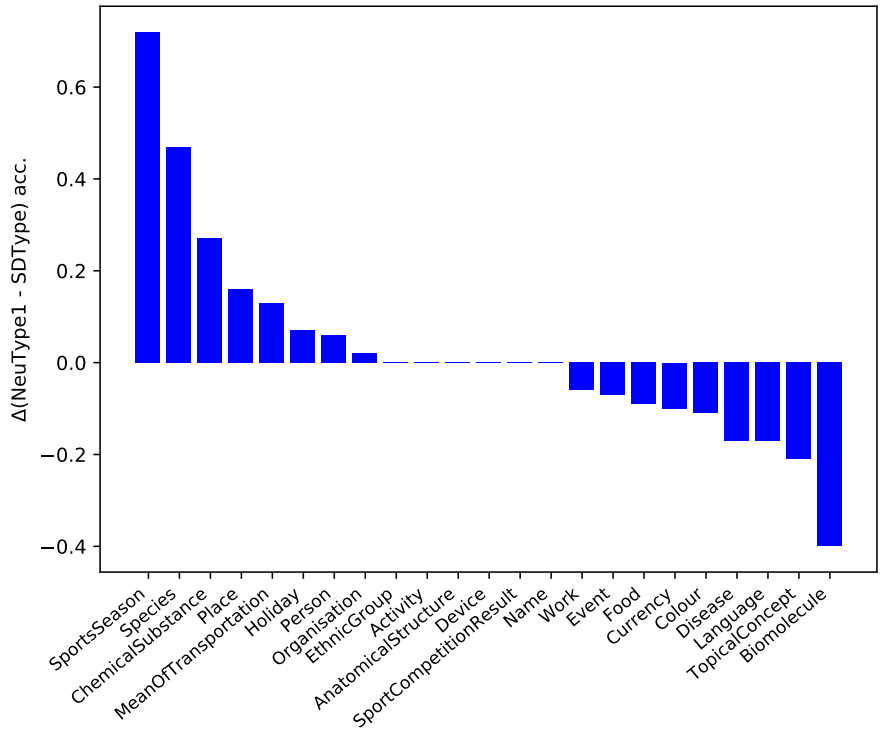

Figure 4.2: Differences Δ strict scoring in the corresponding top-level types between NeuType1 using input $\boldsymbol{A}$ and SDType.

performance in top-level types `Biomolecuile`, `TopicalConcept`, and `Language` among other top-level types, though seeing as they account for 1.4% and less in Dataset 1, this can be attributed to a lack of training data in these top-level types. When looking at more populated top-level types, where SDType has better performance, we find `Work` and `Event`, however, this difference is minimal (< 0.1).

Table 4.15 shows difference between NeuType1 using input $\boldsymbol{A}$ and SDType using all evaluation metrics, both strict and lenient. We plot the difference of lenient scoring method using linear gain measure in Fig. 4.3.

Paulheim and Bizer [2] stated in their paper on SDType that the types they infer are often generic and high up in the type hierarchy. We confirm this when we compare accuracy in top-level types between NeuType1 and SDType in Table 4.15 including lenient gain measures. The corresponding difference plot is seen using linear gain measure in Fig 4.3. We see an overall increase in difference favoring SDType in most top-level types, including a reduction in score for top-level types favoring NeuType1. Therefore confirming that SDType makes good use of lenient gain measures, though NeuType1 still has a better score in the most popular top-level types (e.g., `Person` and `Place`).

Table 4.15: Presenting top-level types according to the ground truth in Dataset 1. Δ denotes difference (NeuType1- SDType).

Top type	Δ strict acc.	Δ linear acc.	Δ exp. acc.	% in Dataset1
Person	0.06	0.07	0.07	30.90
Work	-0.06	-0.06	-0.06	16.80
Place	0.17	0.08	0.08	14.50
Organisation	0.03	0.04	0.04	11.10
Species	0.47	0.45	0.45	4.50
Event	-0.07	-0.06	-0.06	2.70
SportsSeason	0.72	0.72	0.72	1.80
MeanOfTransportation	0.13	0.13	0.13	1.50
Holiday	0.07	0.07	0.07	1.50
TopicalConcept	-0.21	-0.21	-0.21	1.40
EthnicGroup	0.00	0.00	0.00	1.20
Disease	-0.17	-0.17	-0.17	1.20
Language	-0.17	-0.17	-0.17	1.20
Food	-0.09	-0.09	-0.09	1.10
ChemicalSubstance	0.27	0.27	0.27	1.10
Activity	0.00	0.00	0.00	1.00
AnatomicalStructure	0.00	-0.05	-0.05	1.00
Biomolecule	-0.40	-0.40	-0.40	1.00
Device	0.00	0.00	0.00	1.00
SportCompetitionResult	0.00	0.00	0.00	1.00
Currency	-0.10	-0.10	-0.10	1.00
Colour	-0.11	-0.11	-0.11	0.90
Name	0.00	0.00	0.00	0.60

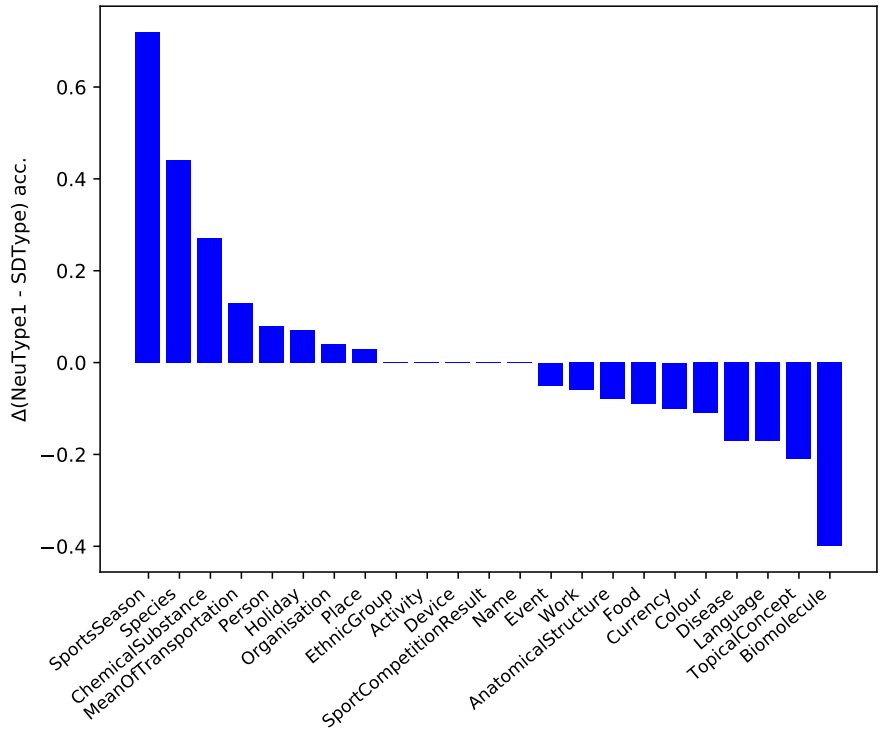

Figure 4.3: Differences Δ linear scoring in the corresponding top-level types between NeuType1 using input $\boldsymbol{A}$ and SDType.

Chapter 5

Conclusions and Future Directions

This chapter concludes our thesis. Main conclusions and answers to our research questions are presented in Sect. 5.1. Future directions are finally presented in Sect. 5.2.

5.1 Conclusions

In this thesis, we have addressed the problem of automatically assigning a type to a given entity in a knowledge base, stating the importance for knowledge bases to update and reflect relevant information on emerging and changing entities. We investigated how related methods solve this problem, and described the two methods *Tipalo* and *SDType* in detail. After identifying these method's drawbacks, we set out to design an approach only requiring short entity descriptions, with optional relationship data.

We have proposed two simple neural network architectures, a shallow network NeuType1, and a deeper network NeuType2. In these architectures we have experimented with a variety of input entity representations, be it only short entity descriptions themselves, exploiting entity-relationship data, or combining these in different ways.

We have evaluated our models using test datasets Dataset 1 and Dataset 2. Here we identified that some top-level types were overrepresented compared to others. In order to counteract this, we designed a pseudo-balanced random approach to draw entities. The resulting evaluation datasets are fairer and contain entities in as many top-level branches as possible.

A main finding of this work is that even these simple neural network approaches, relying on limited input, are able to provide a substantial and significant improvement over the existing state-of-the-art, data-intensive method.

We set out to answer three research questions, we conclude them here.

In **RQ1**, we asked whether a neural approach can, using only entity descriptions, outperform the current state of the art (SDType). We show that by using a word embedding approach feeding short entity descriptions into a neural network, we are able to predict entity types with good accuracy. Neural networks are able to make good use of the centroid word embeddings, therefore showing that the centroid word embedding of a text tells something about what the *concept* of the text presents. This concept then is more than able to indicate a correct type, with an impressive accuracy outperforming the state-of-the-art baseline SDType on our evaluation sets.

In **RQ2**, we asked if entity relationship information can contribute to type prediction. Our results show that accuracy is generally improved given that the relationship data is rich. Sparse relationship data, on the other hand, reduce accuracy. This is an important consideration when adding the optional relationship information.

We proposed the architectures NeuType1 and NeuType2. In **RQ3** we asked which of the proposed architectures perform better. The results show that given rich relationship data, NeuType2 generally performs better. Should relationship data not be available, or the data itself sparse, then NeuType1 is better at predicting correct types using only the short entity descriptions themselves.

In summary, our method is able to support knowledge bases with emerging and existing entities. The knowledge bases as a result reduce overall incompleteness, and reflect knowledge about entities and their types more accurately. Compared to the baseline method, our method is more flexible and not overly tied to the knowledge base structure.

5.2 Future Directions

For future work, it would first be interesting to see how our approach performs on other KBs, like Freebase or YAGO. These ontologies are very different in design, and seeing how our models would perform in these could perhaps give us other ideas and approaches.

Secondly, we would like to explore alternative network architectures and input representations. Deep learning is a broad subject, and there are many techniques to try out. Other approaches to model design would be interesting to investigate, like convolutional networks, or maybe even combining different approaches inside the same model.

Other classification approaches could be tried out. Instead of predicting the type for a given entity, we could rather predict the probability of an entity producing that given type. In other words, calculating $P(e|t)$ instead of $P(t|e)$ for an entity e and type t.

Using word embeddings has proven valuable in this thesis, but more data might be extracted from them. We want to investigate other ways to get word embeddings from phrases instead of words (i.e., 'machine learning' instead of 'machine' and 'learning' as separate words). Another way to use word embeddings would perhaps also be to train a new or existing model on our vocabulary of words so that no words would go amiss.

Appendix A

Extended Tables and Figures

A.1 Tables

Table A.1: Comparing entity-typing assignments where both NeuType1 using input ***A*** and SDType produced an incorrect type.

Entity	GT	Neutype1	Neutype2
Will_Payne_(actor)	Actor	Person	Person
Paul_Valentine	Actor	Person	Person
Lokhvytsia_Raion	AdministrativeRegion	Settlement	Settlement
Kent_Austin	AmericanFootballPlayer	GridironFootballPlayer	GridironFootballPlayer
PTT_(Turkey)	Bank	Company	Company
Group_Voyagers	BusCompany	Company	Company
William_W._Woollcott	BusinessPerson	Person	MusicalArtist
Orange_County_Ramblers	CanadianFootballTeam	NationalFootballLeagueSeason	FootballMatch
Montrose,_Michigan	City	Town	Town
Avilés	City	Settlement	Settlement
Feldgrau	Colour	MilitaryUnit	MilitaryUnit
National_Film_Development_Corporation_of_...	Company	GovernmentAgency	Organisation
Iron_Lore_Entertainment	Company	VideoGame	VideoGame
Testosterone	Drug	Disease	AnatomicalStructure
Polynucleotide_phosphorylase	Enzyme	Protein	Protein
Protein_kinase_A	Enzyme	Protein	Protein
Cyclooxygenase	Enzyme	Drug	Drug
Tuvalu_at_the_Pacific_Games	Event	SportsEvent	SoccerTournament
Cristóbal_Vaca_de_Castro	Governor	OfficeHolder	Person
John_W._Carlin	Governor	OfficeHolder	OfficeHolder
Oktoberfest_celebrations	Holiday	Convention	Convention
Earth_Day	Holiday	Convention	Convention
Wenrohronon	Language	MilitaryConflict	MilitaryConflict
Abdul_Fatah_Younis	MilitaryPerson	OfficeHolder	OfficeHolder
Edgar,_King_of_Scotland	Monarch	Royalty	Royalty
Pic_del_Port_Vell	Mountain	Settlement	SkiArea
Lateral_cricoarytenoid_muscle	Muscle	Nerve	Nerve
Cremaster_muscle	Muscle	AnatomicalStructure	AnatomicalStructure
Chopped_and_screwed	MusicGenre	Single	Single
Reveal_(rapper)	MusicalArtist	Person	Person
Afterschool_Alliance	Non-ProfitOrganisation	Organisation	Organisation
The_Document_Foundation	Non-ProfitOrganisation	Organisation	Organisation
William_Berrian_Vail	OfficeHolder	Politician	MemberOfParliament
William_Cornwallis	OfficeHolder	MilitaryPerson	MilitaryPerson
Ice_hockey_at_the_2014_Winter_Olympics_–_...	OlympicEvent	IceHockeyLeague	OlympicResult
Suez_Canal_Authority	Organisation	Company	Company
Achyuta_Samanta	Person	University	University
Bill_Richards_(musician)	Person	MusicalArtist	MusicalArtist
Rafael_Pineda_Ponce	Person	OfficeHolder	OfficeHolder
Herbert_Reinecker	Person	Writer	Writer
Jesse_More_Greenman	Person	Scientist	Scientist
Asis_Datta	Person	Scientist	Scientist
Yuknoom_Took'_K'awiil	Person	Monarch	Monarch
Martín_Perna	Person	Artist	Artist
Yugendran	Person	MusicalArtist	MusicalArtist
Jane_Gazzo	Person	MusicalArtist	MusicalArtist
Mark_Penn	Politician	Person	Person
Steve_Laffey	Politician	Person	Person
Tom_Horne	Politician	OfficeHolder	OfficeHolder
Mirwais_Yasini	Politician	OfficeHolder	Person
Fernando_de_Santiago_y_Díaz	PrimeMinister	Person	OfficeHolder
Ribonuclease	Protein	Enzyme	Enzyme
Shamsuddin_Ilyas_Shah	Royalty	Country	Monarch
Pulcheria	Saint	Royalty	Monarch
Cruise_(song)	Single	Band	Band
Steve_Ketteridge	SoccerManager	SoccerPlayer	SoccerPlayer
Lena_Videkull	SoccerManager	SoccerPlayer	SoccerPlayer
Marcellin_Gaha_Djiadeu	SoccerManager	SoccerPlayer	SoccerPlayer
Genki_o_Dashite	Song	Single	Single
Mendrisio_railway_station	Station	RailwayStation	RailwayStation
Rihanna_777_Documentary..._7Countries7Day...	TelevisionShow	Album	Album
Locomotion_(TV_channel)	TelevisionStation	Company	Company
Gudensberg	Town	Settlement	Settlement
Tashkurgan_Town	Town	Settlement	Settlement
University_of_St_Andrews_Athletic_Union	University	Organisation	Organisation
SoulPancake	Website	Company	Company
Mahbod_Seraji	Writer	Book	Book

Table A.2: Comparing entity-typing assignments where both NeuType1 using input $\boldsymbol{A}$ and SDType produced an incorrect type.

Entity	GT	Neutype1	Neutype2
Will_Payne_(actor)	Actor	Person	Person
Paul_Valentine	Actor	Person	Person
Lokhvytsia_Raion	AdministrativeRegion	Settlement	Settlement
Kent_Austin	AmericanFootballPlayer	GridironFootballPlayer	GridironFootballPlayer
PTT_(Turkey)	Bank	Company	Company
Group_Voyagers	BusCompany	Company	Company
William_W._Woollcott	BusinessPerson	Person	MusicalArtist
Orange_County_Ramblers	CanadianFootballTeam	NationalFootballLeagueSeason	FootballMatch
Montrose,_Michigan	City	Town	Town
Avilés	City	Settlement	Settlement
Feldgrau	Colour	MilitaryUnit	MilitaryUnit
National_Film_Development_Corporation_of_...	Company	GovernmentAgency	Organisation
Iron_Lore_Entertainment	Company	VideoGame	VideoGame
Testosterone	Drug	Disease	AnatomicalStructure
Cyclooxygenase	Enzyme	Drug	Drug
Protein_kinase_A	Enzyme	Protein	Protein
Polynucleotide_phosphorylase	Enzyme	Protein	Protein
Tuvalu_at_the_Pacific_Games	Event	SportsEvent	SoccerTournament
Cristóbal_Vaca_de_Castro	Governor	OfficeHolder	Person
John_W._Carlin	Governor	OfficeHolder	OfficeHolder
Oktoberfest_celebrations	Holiday	Convention	Convention
Earth_Day	Holiday	Convention	Convention
Wenrohronon	Language	MilitaryConflict	MilitaryConflict
Abdul_Fatah_Younis	MilitaryPerson	OfficeHolder	OfficeHolder
Edgar,_King_of_Scotland	Monarch	Royalty	Royalty
Pic_del_Port_Vell	Mountain	Settlement	SkiArea
Lateral_cricoarytenoid_muscle	Muscle	Nerve	Nerve
Cremaster_muscle	Muscle	AnatomicalStructure	AnatomicalStructure
Chopped_and_screwed	MusicGenre	Single	Single
Reveal_(rapper)	MusicalArtist	Person	Person
Afterschool_Alliance	Non-ProfitOrganisation	Organisation	Organisation
The_Document_Foundation	Non-ProfitOrganisation	Organisation	Organisation
William_Cornwallis	OfficeHolder	MilitaryPerson	MilitaryPerson
William_Berrian_Vail	OfficeHolder	Politician	MemberOfParliament
Ice_hockey_at_the_2014_Winter_Olympics_–_...	OlympicEvent	IceHockeyLeague	OlympicResult
Suez_Canal_Authority	Organisation	Company	Company
Achyuta_Samanta	Person	University	University
Bill_Richards_(musician)	Person	MusicalArtist	MusicalArtist
Rafael_Pineda_Ponce	Person	OfficeHolder	OfficeHolder
Herbert_Reinecker	Person	Writer	Writer
Jesse_More_Greenman	Person	Scientist	Scientist
Asis_Datta	Person	Scientist	Scientist
Yuknoom_Took'_K'awiil	Person	Monarch	Monarch
Martín_Perna	Person	Artist	Artist
Yugendran	Person	MusicalArtist	MusicalArtist
Jane_Gazzo	Person	MusicalArtist	MusicalArtist
Tom_Horne	Politician	OfficeHolder	OfficeHolder
Mirwais_Yasini	Politician	OfficeHolder	Person
Steve_Laffey	Politician	Person	Person
Mark_Penn	Politician	Person	Person
Fernando_de_Santiago_y_Díaz	PrimeMinister	Person	OfficeHolder
Ribonuclease	Protein	Enzyme	Enzyme
Shamsuddin_Ilyas_Shah	Royalty	Country	Monarch
Pulcheria	Saint	Royalty	Monarch
Cruise_(song)	Single	Band	Band
Lena_Videkull	SoccerManager	SoccerPlayer	SoccerPlayer
Steve_Ketteridge	SoccerManager	SoccerPlayer	SoccerPlayer
Marcellin_Gaha_Djiadeu	SoccerManager	SoccerPlayer	SoccerPlayer
Genki_o_Dashite	Song	Single	Single
Mendrisio_railway_station	Station	RailwayStation	RailwayStation
Rihanna_777_Documentary..._7Countries7Day...	TelevisionShow	Album	Album
Locomotion_(TV_channel)	TelevisionStation	Company	Company
Tashkurgan_Town	Town	Settlement	Settlement
Gudensberg	Town	Settlement	Settlement
University_of_St_Andrews_Athletic_Union	University	Organisation	Organisation
SoulPancake	Website	Company	Company
Mahbod_Seraji	Writer	Book	Book

A.2 Figures

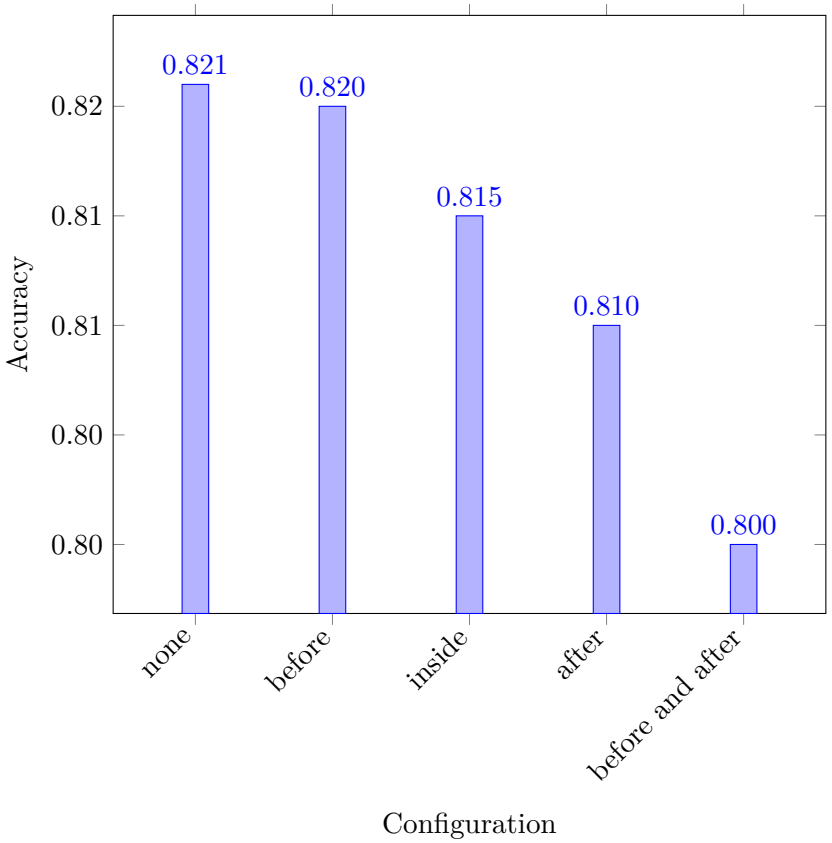

Figure A.1: Accuracy of different dropout configurations. Each configuration is evaluated on Dataset 1 using strict scoring.

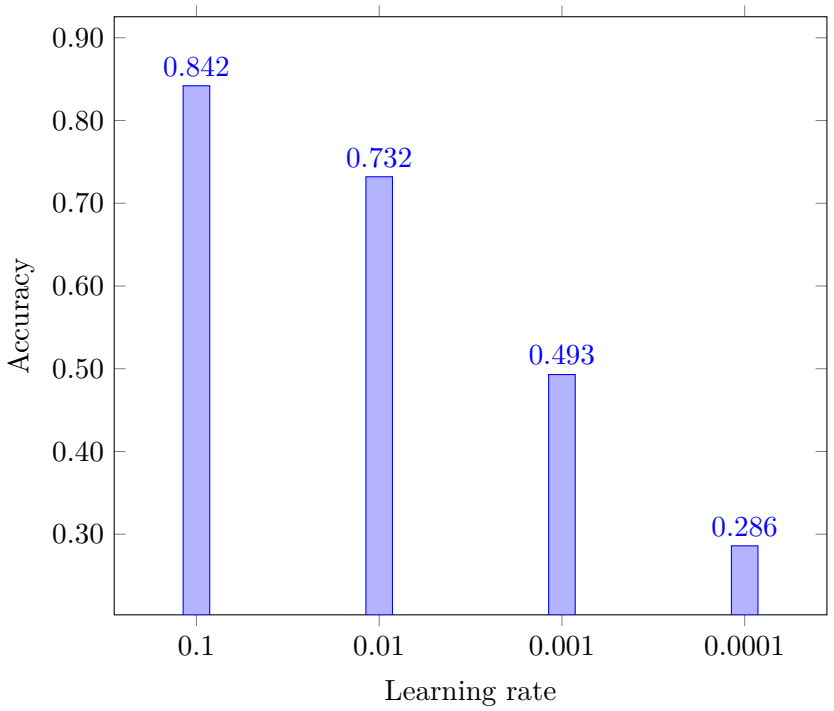

Figure A.2: Results from a rough learning rate search. Each configuration is evaluated on Dataset 1 using strict scoring.

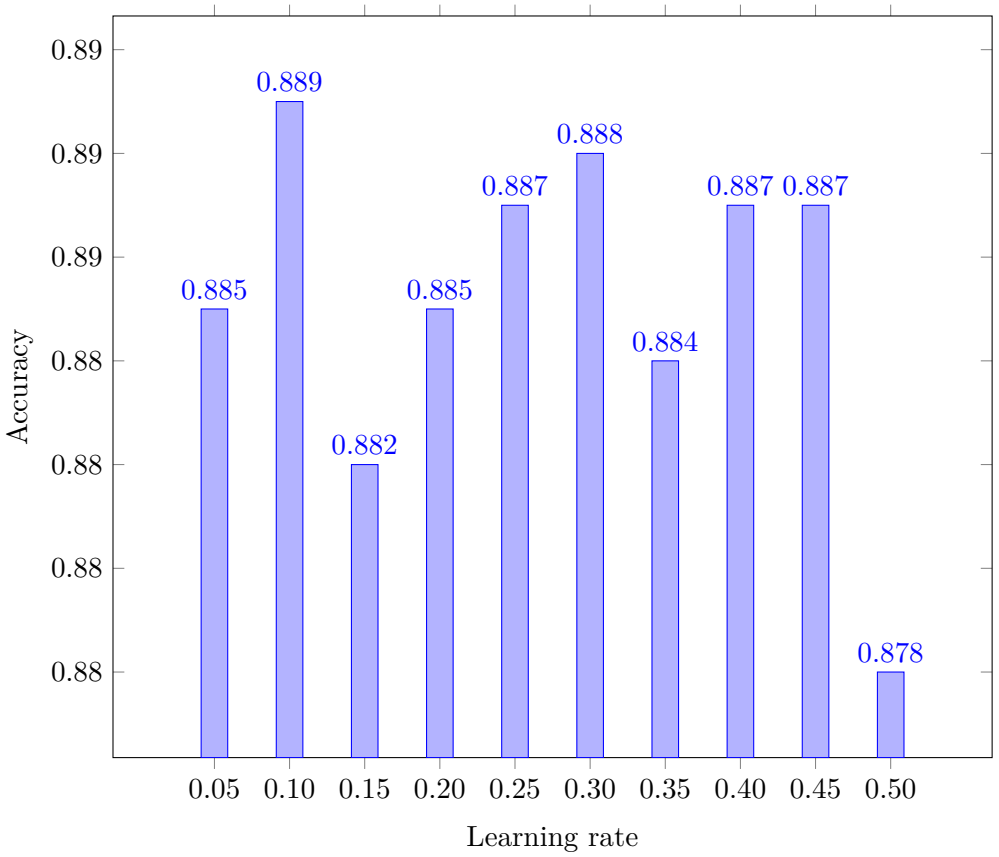

Figure A.3: Results from a fine learning rate search. Each configuration is evaluated on Dataset 1 using strict scoring.

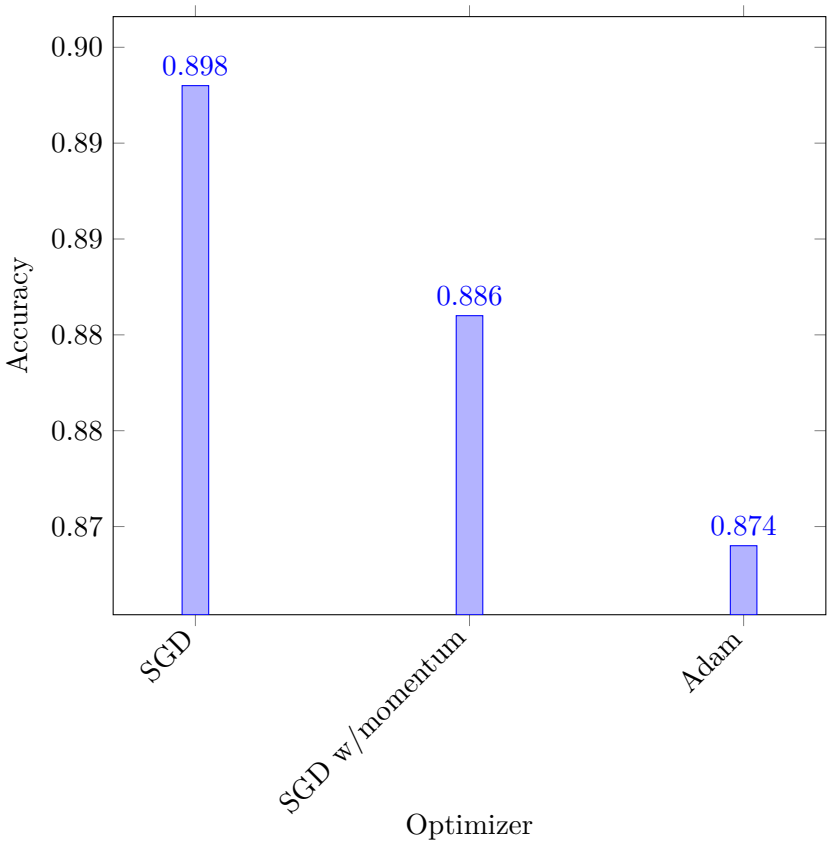

Figure A.4: Accuracy of different dropout configurations. Each configuration is evaluated on Dataset 1 using strict scoring.

Bibliography

[1] Aldo Gangemi, Andrea Giovanni Nuzzolese, Valentina Presutti, Francesco Draicchio, Alberto Musetti, and Paolo Ciancarini. Automatic typing of DBpedia entities. In *International Semantic Web Conference*, pages 65–81. Springer, 2012.

[2] Heiko Paulheim and Christian Bizer. Type inference on noisy rdf data. In *International Semantic Web Conference*, pages 510–525. Springer, 2013.

[3] Krisztian Balog. Entity-oriented search. Springer, 2018.

[4] Eduard Hovy, Roberto Navigli, and Simone Paolo Ponzetto. Collaboratively built semi-structured content and Artificial Intelligence: The story so far. *Artificial Intelligence*, 194:2–27, 2013.

[5] Frederick Hayes-Roth, Donald Waterman, and Douglas Lenat. Building expert systems. 1984.

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[7] Douglas B Lenat. Cyc: A large-scale investment in knowledge infrastructure. *Communications of the ACM*, 38(11):33–38, 1995.

[8] Christian Bizer, Jens Lehmann, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. DBpedia - a crystallization point for the web of data. *Web Semantics: science, services and agents on the world wide web*, 7(3):154–165, 2009.

[9] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick Van Kleef, Sören Auer, et al. Dbpedia - a large-scale, multilingual knowledge base extracted from Wikipedia. *Semantic Web*, 6(2):167–195, 2015.

[10] Darío Garigliotti and Krisztian Balog. On type-aware entity retrieval. In *Proceedings of the ACM SIGIR International Conference on Theory of Information Retrieval*, ICTIR '17, pages 27–34. ACM, 2017. doi: 10.1145/3121050.3121054.

[11] 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. AcM, 2008.

[12] 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. ACM, 2007.

[13] Thomas Lin, Oren Etzioni, et al. No noun phrase left behind: detecting and typing unlinkable entities. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pages 893–903. Association for Computational Linguistics, 2012.

[14] Ndapandula Nakashole, Tomasz Tylenda, and Gerhard Weikum. Fine-grained semantic typing of emerging entities. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 1488–1497, 2013.

[15] Tomáš Kliegr and Ondřej Zamazal. Lhd 2.0: A text mining approach to typing entities in knowledge graphs. *Web Semantics: Science, Services and Agents on the World Wide Web*, 39:47–61, 2016.

[16] Yadollah Yaghoobzadeh and Hinrich Schütze. Corpus-level fine-grained entity typing using contextual information. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 715–725, 2015.

[17] Valentina Presutti, Francesco Draicchio, and Aldo Gangemi. Knowledge extraction based on discourse representation theory and linguistic frames. In *International Conference on Knowledge Engineering and Knowledge Management*, pages 114–129. Springer, 2012.

[18] Heiko Paulheim and Christian Bizer. Improving the quality of linked data using statistical distributions. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 10(2):63–86, 2014.

[19] Yadollah Yaghoobzadeh. *Distributed representations for fine-grained entity typing.* PhD thesis, lmu, 2017.

[20] Yoav Goldberg. A primer on neural network models for natural language processing. *Journal of Artificial Intelligence Research*, 57:345–420, 2016.

[21] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

[22] Geoffrey E Hinton. Distributed representations. 1984.

[23] Yoav Goldberg. *Neural Network Methods in Natural Language Processing*. Morgan & Claypool Publishers, 2017.

[24] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Proc. of NIPS*, pages 3111–3119, 2013.

[25] Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. Linguistic regularities in continuous space word representations. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 746–751, 2013.

[26] Quoc V. Le and Tomas Mikolov. Distributed representations of sentences and documents. In *Proc. of ICML*, volume 32, pages 1188–1196, 2014.

[27] Mohit Iyyer, Varun Manjunatha, Jordan Boyd-Graber, and Hal Daumé III. Deep unordered composition rivals syntactic methods for text classification. In *Proc. of ACL-IJCNLP*, pages 1681–1691, 2015.

[28] Krisztian Balog and Robert Neumayer. Hierarchical target type identification for entity-oriented queries. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 2391–2394. ACM, 2012.

[29] François Chollet et al. Keras. https://keras.io, 2015.