eman ta zabal zazu

Universidad del País Vasco | Euskal Herriko Unibertsitatea

INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Computer Science Degree

Computation

End of Degree Project

# Deep Learning for semantic parsing.

Author

*Edu Vallejo Arguinzoniz*

2020

eman ta zabal zazu

Universidad del País Vasco Euskal Herriko Unibertsitatea

INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Computer Science Degree

Computation

End of Degree Project

# Deep Learning for semantic parsing.

Author

*Edu Vallejo Arguinzoniz*

Instructors

Eneko Agirre and Gorka Azkune

# Abstract

This is the memory of an exploratory research project on techniques for reasoning on text with Deep Learning (DL). To study reasoning we focus on the problem of Natural Language Question-Understanding (NLQU), and in particular in the task of Semantic Parsing, a challenging Natural Language Processing (NLP) task that requires NLQU and even puts todays Deep Learning machinery to the test.

More specifically we provide a discussion about semantic parsing, and in concrete, deep learning techniques for semantic parsing. In our study of semantic parsing, we focus on two central topics: annotation and (deep learning) systems. At a more practical level, we run experiments of a state-of-the-art semantic parsing system a new and innovative semantic parsing dataset called OTTA [Deriu et al., 2020]. Finally, we take the opportunity to learn the details of the system implementation, and we refactor the system to make it suitable (in terms of speed and integration) for future work.

# Contents

**Abstract** **i**

**Contents** **iii**

**List of Figures** **vii**

**List of Tables** **ix**

**1 Introduction** **1**

**2 Background for deep learning in semantic parsing** **5**

2.1 Recurrent Neural Networks . . . . . . . . . . . . . . . . . . . . . . . . . 5

2.1.1 RNNs in NLP . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

2.2 Seq2seq architecture . . . . . . . . . . . . . . . . . . . . . . . . . . 11

2.3 Auto-encoders . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12

**3 Annotation in semantic parsing** **15**

3.1 Why is annotation hard? . . . . . . . . . . . . . . . . . . . . . . . . 15

3.1.1 Annotation task complexity . . . . . . . . . . . . . . . . . . . . 16

3.1.2 Annotation bias . . . . . . . . . . . . . . . . . . . . . . . . . . 16

3.2 ImageNet: The dataset that started it all . . . . . . . . . . . . . . . . . . 17

3.3 Spider: The ImageNet of semantic parsing . . . . . . . . . . . . . . . . . 19

3.3.1 System evaluation . . . . . . . . . . . . . . . . . . . . . . . . 21
3.3.2 A brute-force annotation process . . . . . . . . . . . . . . . . . . . 22
3.4 OTTA: A more productive annotation process . . . . . . . . . . . . . . . 23
3.5 Annotation inverted, using Operation Trees . . . . . . . . . . . . . . . . 24

**4 Systems for semantic parsing 29**
4.1 Seq2seq for semantic parsing . . . . . . . . . . . . . . . . . . . . . . 30
4.2 Type-safe seq2seq, constraining the productions with grammars . . . . . 30
4.3 Grammar-RNN: acknowledging tree structure . . . . . . . . . . . . . . . 32
4.4 Capitalizing on OTTA, improving performance through pretraining . . . . 34

**5 Experiments and results 37**
5.1 Experimental setup . . . . . . . . . . . . . . . . . . . . . . . . . . 38
5.2 Official results . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
5.3 Replication results . . . . . . . . . . . . . . . . . . . . . . . . . . 40
5.4 Results after refactoring . . . . . . . . . . . . . . . . . . . . . . . . 41
5.5 Performance analysis . . . . . . . . . . . . . . . . . . . . . . . . . . 43

**6 Conclusions and future work 45**
6.1 Conclusions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
6.2 Future work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46

**Appendices**

**A GrammarRNN refactoring 51**
A.1 Refactoring the decoder forward pass . . . . . . . . . . . . . . . . . . 52
A.2 Defining a solid machine learning pipeline . . . . . . . . . . . . . . . . 52
A.3 Faster beam search using batching . . . . . . . . . . . . . . . . . . . 53

**B Project objectives report** **55**

B.1 Project description and goals . . . . . . . . . . . . . . . . . . . . . . . 55

B.2 Project planning . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56

B.2.1 WBS diagram . . . . . . . . . . . . . . . . . . . . . . . . . 56

B.2.2 Work units . . . . . . . . . . . . . . . . . . . . . . . . . . . 56

B.2.3 Milestones . . . . . . . . . . . . . . . . . . . . . . . . . . . 59

B.3 Methodology . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59

B.3.1 Meetings . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60

B.3.2 Work place . . . . . . . . . . . . . . . . . . . . . . . . . . . 60

B.4 Risks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60

**Bibliography** **63**

# List of Figures

2.1 Diagram of the inference process in a RNN . . . . . . . . . . . . . . . . . 6

2.2 Diagram of the implementation of the vanilla-rnn cell . . . . . . . . . . . 7

2.3 Diagram of the implementation of the GRU cell . . . . . . . . . . . . . . 8

2.4 Diagram of the implementation of the LSTM cell . . . . . . . . . . . . . 8

2.5 Diagram of the sentence encoding procedure using pretrained GloVe embeddings . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

2.6 Diagram of the sentence decoding procedure when using teacher forcing . 11

2.7 Diagram of the sentence decoding procedure at inference time using the sampling method . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

2.8 Diagram of the use of a seq2seq model that transforms a question to an answer . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12

2.9 Diagram of s simple 2 layer multilayer perceptron auto-encoder. The first layer is a bottleneck that forces the intermediate representation to compress the information enriching it as a consequence. . . . . . . . . . . . . 13

3.1 Diagram showing the AlexNet architecture . . . . . . . . . . . . . . . . . 18

3.2 Progression of the SOTA systems in the ILSVRC challenge . . . . . . . . 18

3.3 A chart showing the relative amount of queries of different types on popular text-to-SQL datasets. The value of 1 in each axis corresponds to the amount of queries of a given type that the dataset with the most queries of such type has. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20

3.4 An annotation sample in Spider. . . . . . . . . . . . . . . . . . . . . . . 23

3.5 Diagram of the annotation process of Spider. . . . . . . . . . . . . . . . . . . 23

3.6 Example of an OT for the query ”Who starred in ’The Notebook’?". . . . . 24

3.7 CFG that all OTs follow . . . . . . . . . . . . . . . . . . . . . . . . . . . 25

3.8 Annotation interface that was used for creating OTTA thought crowd-sourcing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

4.1 The architecture of a seq2seq model where the decoder has been modified to produce rules, which in turn generate a SQL query that can be used to query a database. This particular seq2seq model uses a bidirectional LSTM in the encoder and attention. . . . . . . . . . . . . . . . . . . . . 31

4.2 Abstract syntax tree of a python expression and the time-step in the decoding process that produces each tree node. . . . . . . . . . . . . . . . . . 34

4.3 Unfolded Grammar-RNN aligned with the nodes in the AST that are expanded at each time-step . . . . . . . . . . . . . . . . . . . . . . . . . 34

B.1 Work Breakdown Structure of the project . . . . . . . . . . . . . . . . . 56

# List of Tables

5.1 Number of samples in the fixed test split for each database and difficulty setting. . . . . 38

5.2 Execution Accuracy of the system in different databases and different query complexities as reported in the OTTA paper (cross-validation). . . . 39

5.3 Exact Matching score for two auto-encoder pretraining runs with the original code. . . . . 40

5.4 Execution Accuracy of the original system when trained on top of auto-encoder A taking into account different databases and different query complexities. . . . . 40

5.5 Execution Accuracy of the original system when trained on top of auto-encoder B taking into account different databases and different query complexities. . . . . 41

5.6 Execution Accuracy of the original system when trained without pretraining phase taking into account different databases and different query complexities. . . . . 41

5.7 Exact Matching score for two auto-encoder pretraining runs with the refactored code. . . . . 42

5.8 Execution Accuracy of the refactored system when trained on top of auto-encoder A taking into account different databases and different query complexities. . . . . 42

5.9 Execution Accuracy of the refactored system when trained on top of auto-encoder B taking into account different databases and different query complexities. . . . . 42

5.10 Execution Accuracy of the refactored system when trained without pre-training phase taking into account different databases and different query complexities. . . . . . . . . . . . . . . . . . . . . . . . . . . . 43

5.11 Training speed ($samples/s$) in the pretraining phase, with and without the refactoring improvements and on different hardware platforms and batch-sizes $b$ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43

5.12 Training speed ($samples/s$) in the training phase, with and without the refactoring improvements and on different hardware platforms and batch-sizes $b$ . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43

B.1 Time estimates for each work unit . . . . . . . . . . . . . . . . . . . . . 59

B.2 Deliverables and their deadlines . . . . . . . . . . . . . . . . . . . . . . 59

# CHAPTER 1

# Introduction

Since the very conception of Artificial Intelligence, researchers have coveted to understand and replicate human reasoning. Early attempts tried to model human reasoning as a theorem proving exercise in logic, where axioms would be analogous to knowledge humans leveraged and the proof mechanism could be comparable to the reasoning process [Russell and Norvig, 2002].

It quickly became clear that human reasoning and theorem proving were very different [Stenning and Van Lambalgen, 2012]. For one, humans mostly work with imperfect information and rough estimates, whereas the tools of logic have difficulties capturing information that is not well-defined. Additionally, as more knowledge is added to a world model, the logical reasoning paths grow exponentially and efficient exploration of reasoning paths becomes challenging for any proof system. The fact that humans are able to cope with the endlessly complex world that surrounds us, suggests that we have to perform reasoning in a way other than exhaustive search.

Attempts were made to alleviate the problems, such as using a **fuzzy logic** theory [Zadeh, 1988] that could deal with uncertainty, and proof systems based on heuristics that promised to mimic human intuition in the reasoning process. Despite the efforts, these approaches of uncovering human reasoning found very similar problems, which prevented them to be widely adopted by the scientific community.

More recently, propelled by the advances in the NLP field, reasoning is being studied in the context of language [MacCartney and Manning, 2009]. Language and reasoning have a long story of being studied in conjunction, as it is believed by many that language has a big implication on human thought. However, most of these prior studies were theoretical which limited the testability of hypotheses, halting progress.

From a more practical NLP perspective, reasoning on language is not directly analyzed but instead it is observed in systems that implement NLP tasks that implicitly require reasoning. Many NLP tasks involve reasoning at some level, however to properly research reasoning, the task has to mainly be a reasoning challenge.

Natural language question understanding (NLQU) is a family of tasks that require understanding questions posed in natural language, either to answer them directly or to generate a plan that helps another system answer the questions [Wolfson et al., 2020]. These tasks often require that the system reasons over the relationship between entities in the question. This is specially true if the question is compositional which requires to integrate multiple sources of information in a specific way to answer the question.

Semantic parsing is the task of translating a natural language description into a logical form, a representation of the meaning of the description that a computer can understand [Jia and Liang, 2016a]. Semantic parsing is central to many natural language understanding (NLU) tasks, including NLQU.

Natural language interface to databases (NLIDB) is probably the most popular example of a semantic parsing NLQU task [Reinaldha and Widagdo, 2014]. NLIDB tasks consist in translating a natural language question into a program in a given querying language such that it can be executed in a database to retrieve the answer to the question.

In this context, we define the objectives of this undergraduate theses as the following: 1) Study the literature on semantic parsing, dataset creation and state-of-the-art (SOTA) in deep learning systems for semantic parsing; 2) Install, run and analyze the characteristics of a SOTA system in OTTA, a recently released NLIDB dataset; 3) Refactor the system implementation to prepare it for future research work, and as a way to get familiar with the more practical aspects of DL techniques for semantic parsing.

To study the literature we narrate the intricacies of NLIDB in two important aspects. First we talk about the dataset annotation process, the hardships that are faced when designing effective annotation processes for semantic parsing, and the examples of two modern datasets that have very different approaches to annotation. Spider [Yu et al., 2018] is probably the largest and most complex text-to-SQL dataset available to date; OTTA [Deriu et al., 2020] on the other hand, is a moderate size NLIDB dataset that uses a purpose-made high-level querying language, and introduces an innovative annotation procedure to improve annotation efficiency.

Secondly, we explain how present-day DL methods can be used to implement semantic parsing systems. We begin by demonstrating how current DL methods in NLP (RNNs and seq2seq architectures) can be adapted to perform semantic parsing. We proceed by outlying further innovations specific to this field that enable greater performance, such as grammar based logical form decoding [Krishnamurthy et al., 2017]; and GrammarRNNs [Yin and Neubig, 2017], which are a modification of regular RNNs that are better suited for decoding parse-trees.

Besides giving a depiction of the current state of affairs in semantic parsing, we also perform our own experiments in the context of an ongoing European project called LIHLITH [Agirre Bengoa et al., 2019] (Learning to Interact with Humans by Lifelong Interaction with Humans).

In these experiments we tested a state-of-the-art semantic parsing system on the OTTA dataset (recently developed in the LIHLITH project), and a reimplementation of the system of our own that we developed because we considered that the original implementation was not a good enough base for further developments.

Our refactored system presents several design improvements, which allow it to be easier to operate (because of simplified interfaces) and obtains performance gains of up to 20*x* when in appropriate conditions (running on GPU with a big enough batch-size). We believe that these changes are important if the system is to be used for future work directions.

CHAPTER 2

# Background for deep learning in semantic parsing

To understand the following chapters well, we need to talk about some background knowledge that will become essential in the coming explanations. Because it is not possible to go in depth into all the theoretical aspects that would be required to fully understand the content in the next chapters, the reader is assumed to have an introductory level understanding in basic DL techniques. The reader is referred to [Goodfellow et al., 2016] for a much more complete coverage on DL techniques.

In this chapter we give a bare-bones explanation of some topics relevant for the development of later chapters, these include: recurrent neural networks (RNNs), RNNs for NLP tasks, seq2seq architectures and auto-encoders.

## 2.1 Recurrent Neural Networks

Recurrent neural networks are a family of deep learning architectures that are specialised in processing data of sequential nature [Dupond, 2019]. RNNs are well-suited for a lot of NLP tasks due to the sequential essence of language, and they are used extensively in the field [Socher et al., 2011]. Some examples of the use of RNNs in NLP and language related topics are machine translation [Sutskever et al., 2014a], language modeling [Józefowicz et al., 2016], speech recognition [Sak et al., 2014, Li and Wu, 2014] and speech synthesis [Zen and Sak, 2015] to name a few.

There are many types of RNNs, however they are all based in the same fundamental ideas.

For instance, all RNNs work by processing a sequence of elements one element at a time and every element is processed uniformly.

The interfaces of every RNN are the same. To process any one element, two inputs are needed: a vector representation of the element, and a state vector (also called hidden-state) which encodes all the elements seen so far. With these two inputs a RNN cell will produce the next hidden-state as an output, this hidden-state can be used as a representation for the sequence processed so far and to feed the next RNN cell state.

More formally, a RNN at time-step $t$ has hidden-state $h_{t-1}$ and processes an element $x_t$ from a sequence $\boldsymbol{x}$. The RNN computes the next hidden-state $h_t$ as a function $f$ of the previous hidden-state $h_{t-1}$ and the element that is being processed $x_t$, see Figure 2.1

$$h_t = f(x_t, h_{t-1})$$

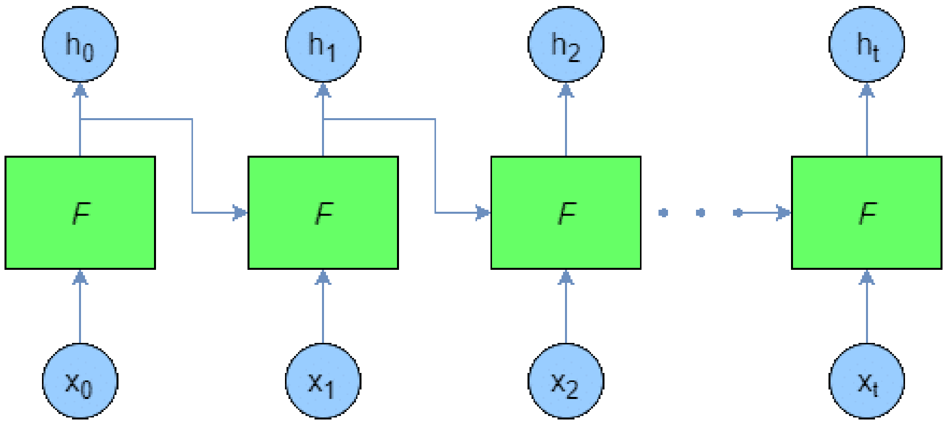


**Figure 2.1:** Diagram of the inference process in a RNN

Source: adventuresinmachinelearning.com

For $f$ to be suitable for DL techniques, it has to be a parametric function $f = f_\theta$ with parameters $\theta$, and $f$ needs to be differentiable with respect to $\theta$, i.e., $\exists \frac{df}{d\theta}$. Differentiability is required if we are to use gradient based methods to optimize the parameters efficiently, which is nowadays the best performing approach if the number of parameters is large.

Finally, $f$ should be a non-linear function so that the capacity of the model can scale with the depth of the deep learning model.

Clearly the choice of $f$ is important and by no means trivial. Different RNN architectures will use different functions, and the choice of the function will give the architecture its character. Some examples of popular RNN architectures are: vanilla-RNNs [Goller and Kuchler, 1996], Gated Recurrent Units (GRUs) [Cho et al., 2014] and Long-Short Term Memory cells (LSTMs) [Hochreiter and Schmidhuber, 1997]. The implementations of $f$ for each architecture are shown in Figures 2.2, 2.3 and 2.4 respectively.

Vanilla-RNNs implement perhaps the simplest $f$ fucntion, a simple affine transformation of the combination of the cell initial-state and the input vector, and a tanh activation function. The combination of these two vectors is usually defined as a concatenation.

$$h_t = \tanh(W_\theta [h_{t-1} : x_t] + b_\theta)$$

Other RNN cells implement more complex functions to compute $h_t$, but usually inherit some aspects from the vanilla-RNN (such as the combination of previous state vector and input before applying a tanh activation function).

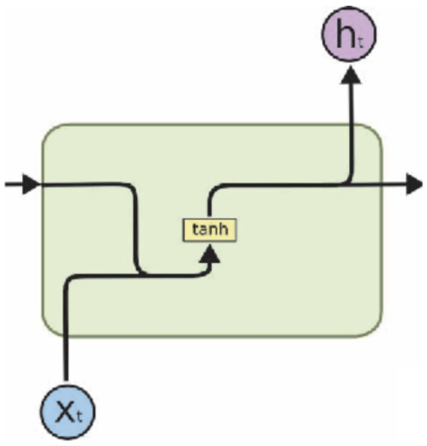


**Figure 2.2:** Diagram of the implementation of the vanilla-rnn cell

Source: towardsdatascience.com

There is a lot that can be said about the different types of RNN architectures, however we consider that this discussion is mostly out of the scope of this document. To summarize the key points: GRUs and LSTMs are newer architectures created to solve the problems that vanilla RNNs have. Namely that vanilla RNNs show convergence problems and instabilities when training, and that vanilla RNNs have the tendency to forget information about far off elements in the sequence, hampering performance.

From this point onward we do not make the distinction between different RNN architec-

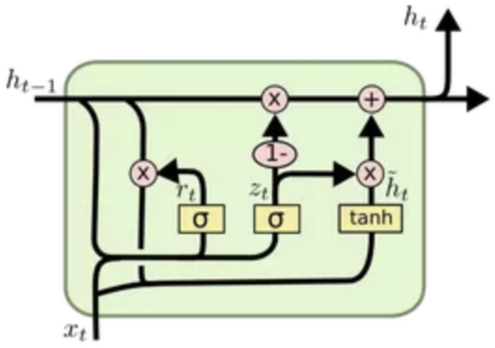


**Figure 2.3:** Diagram of the implementation of the GRU cell

Source: towardsdatascience.com

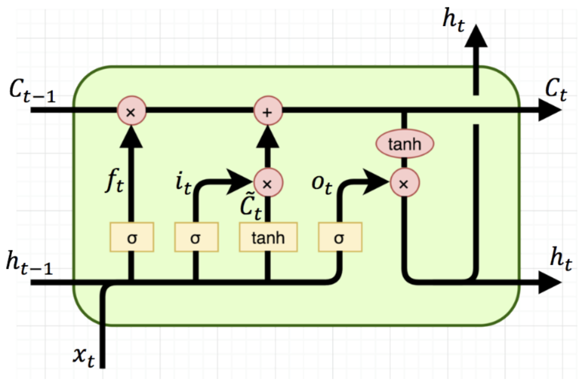


**Figure 2.4:** Diagram of the implementation of the LSTM cell

Source: towardsdatascience.com

tures, we just refer to them as RNNs in general and assume we are using what works best (probably not vanilla-RNNs).

### 2.1.1 RNNs in NLP

To use RNNs (or any kind of DL method) with language, we need to come up with a representation of language that is usable for deep learning.

There are all kinds of text that we may want to process, for example we might want to translate an entire book between languages, or automatically generate the caption of a given image or maybe we want to process dialogue in a dialogue system. These are very different kinds of text with varying levels of length and complexity. Despite the diversity, we would like to have DL machinery that can deal with all kinds of types of text and let the learning procedure compensate for the disparity.

For most purposes, DL methods work with text at the sentence level, that is, sentences are given a (to be learned) vector representation and this representation is used to accomplish tasks.

Representations for sentences can not be learned statically (the representation of every sentence can not be stored in a memory table) because there are a combinatorial (and even infinite) amount of sentences, and we would like the system to generalize to sentences not seen at training time. Because of these limitations, sentences need to be treated as a sequence of smaller units of information, such as n-grams, words, sub-word level tokens or characters.

Unlike with sentences, static representations can be learned for these atomic units of information. Static representations for tokens usually come in the form of token embeddings. Token embeddings are fixed length vectors that are used in place of the token and that do propagate gradients allowing the vectors to be learned at train time through gradient descent methods [Mikolov et al., 2013].

There are two instances that involve sentence representations: encoding a sentence into a vector sentence representation for later use in some task, and decoding a sentence representation into an actual textual sentence.

In the same way, token embeddings can be learned one of two ways: they can either be grouped in a fast indexed lookup table when the deep learning model has textual input, or exist implicitly in a *softmax* layer that assigns probabilities to tokens when the model generates text.

To derive the representation of a sentence from the embeddings of the tokens that make up the sentence, and to recover the tokens that construct a sentence given the representation of the sentence; token embeddings need to be aggregated into the sentence representation, and the sentence representation needs to be broken up into token embeddings.

The RNN architecture provides a mechanism to do both of these things. In the first case, encoding a sentence with a RNN is straightforward, the embedding for each token $x_t$ is fed into the RNN in order, and since the hidden-state $h_t$ at any given time $t$ represents the whole sequence up to the token $x_t$, the last processed hidden-state $h_n$ represents the whole sentence, see Figure 2.5.

Because the last state needs to encode information about the whole sentence into a single vector, it regularly results in a bottleneck. To mitigate this effect a technique called *attention* [Bahdanau et al., 2014] is used often. Attention is a soft-selection mechanism

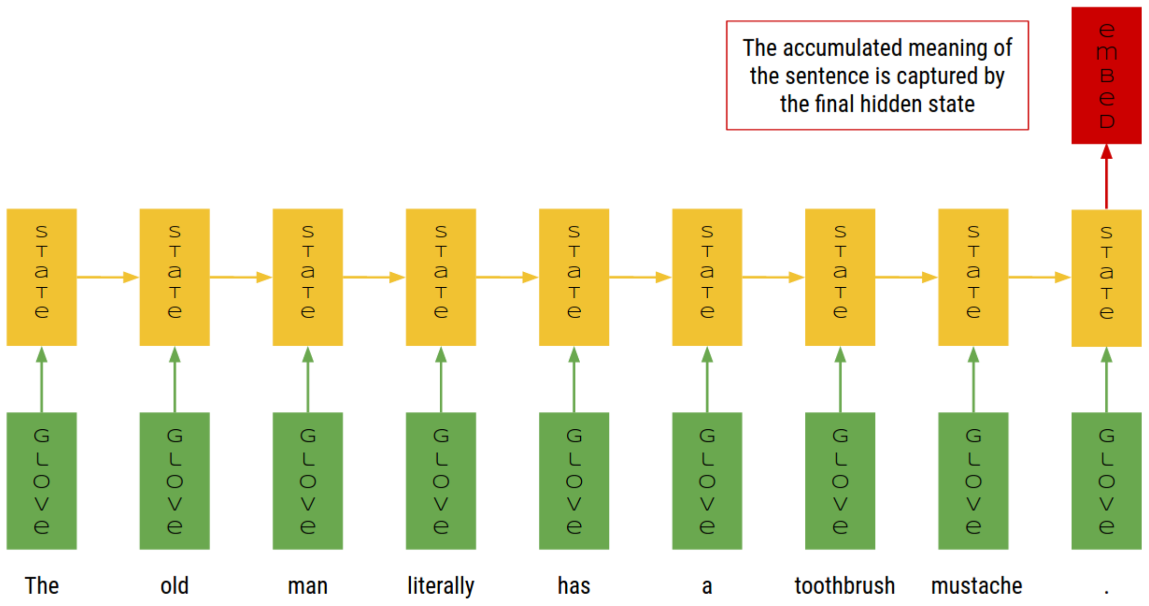

The accumulated meaning of the sentence is captured by the final hidden state
embed
state
glove
The
old
man
literally
has
a
toothbrush
mustache
.

**Figure 2.5:** Diagram of the sentence encoding procedure using pretrained GloVe embeddings

Source: indico.io

that enables using all previous hidden-states of the encoder as the sentence representation and allows the system to focus on the states that are important for the task. The attention mechanism is crucial for good performance in any modern neural architecture, however it is not the main focus of this work to explain the intricacies of attention in detail.

Decoding a sentence representation into the tokens that form it is a bit more complex. It works differently whether we are training the system or using it for inference. In both cases the first RNN cell is initialized with the vector sentence representation and at each time step the corresponding RNN cell is initialized with the hidden-state of the previous RNN just like normal. The part that changes is the sequence $\boldsymbol{x}$ that the RNN processes as input.

When training, the RNN cell at time step $t$ gets fed the token number $t-1$ in the gold standard sentence, which might not be the same token that the previous RNN cell predicted (this technique is called teacher forcing and can be seen in Figure 2.6). When using the trained model for inference however (e.g. in test), because there is no golden-standard available, the output of the previous RNN cells must be used, there are more than one way to do this [Cho, 2016] but we will not get into the details in this report. Figure 2.7 shows decoding by one of the simplest methods for inference time decoding, it just requires feeding the previous most probable token as the next RNN cell input.

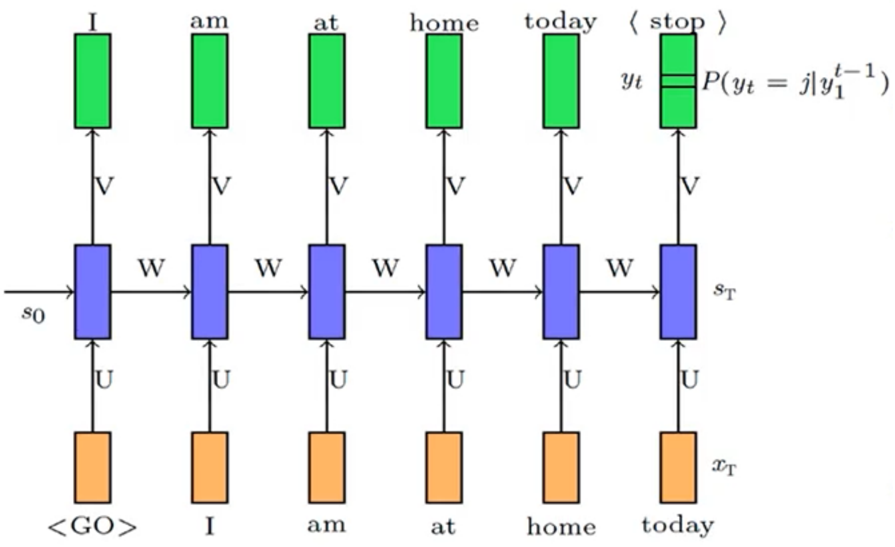


**Figure 2.6:** Diagram of the sentence decoding procedure when using teacher forcing

Source: towardsdatascience.com

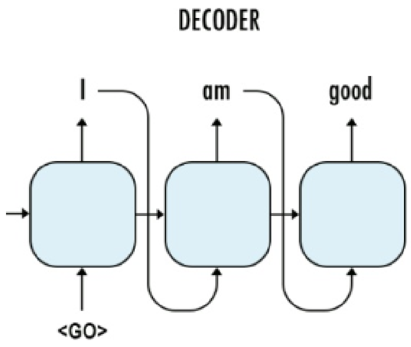


**Figure 2.7:** Diagram of the sentence decoding procedure at inference time using the sampling method

Source: 6chaoran.wordpress.com

## 2.2 Seq2seq architecture

Seq2seq [Sutskever et al., 2014b] is a common DL technique used in NLP (and also in other fields) when dealing with tasks that require transforming data of sequential nature (e.g. a sentence) between representations.

A seq2seq architecture transforms a sequence into another sequence. It does so by employing two connected RNNs: a combination of an encoder and a decoder, see Figure 2.8. A key aspect of seq2seq methods is that the parameters of the whole architecture are learned end-to-end, which means that the gradient of the loss of the decoder, is propagated through the whole computation graph that defines the architecture; even the encoder and the embedding tables at the very beginning of the graph.

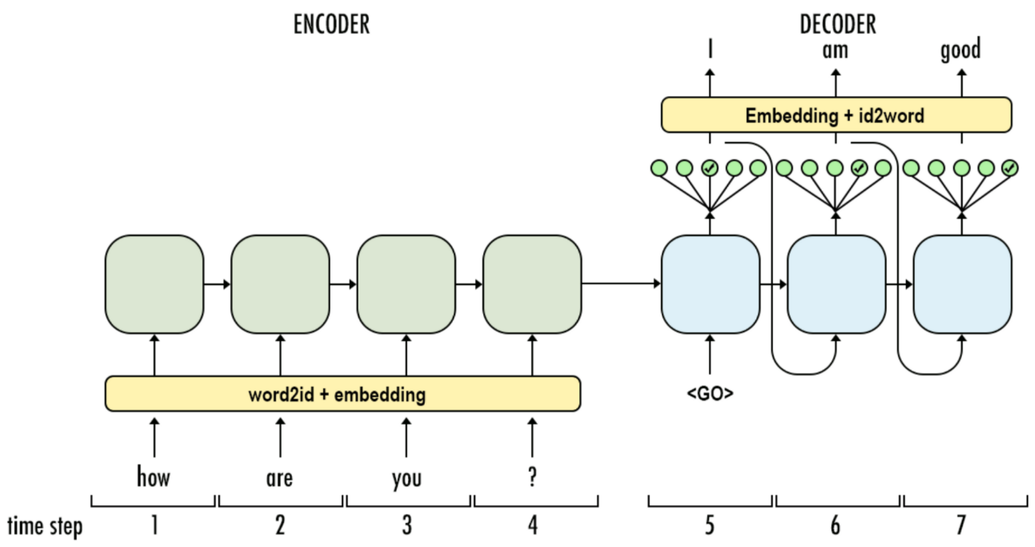


**Figure 2.8:** Diagram of the use of a seq2seq model that transforms a question to an answer

Source: medium.com

Seq2seq architectures have been employed with success in many tasks that involve transforming a natural language fragment into another natural language fragment. Some examples are: machine translation, conversational tasks and text summarization [Sutskever et al., 2014c].

## 2.3 Auto-encoders

An auto-encoder in DL, is a type of encoder-decoder system that is trained so that the output of the system for a given input is as closed to the input as possible [Kramer, 1991]. By training a system to replicate its input in the output, the system needs to learn effective representations for the items in the input that encode as much information as possible about them, if not, the decoder will not be able to resolve what the input was. Because the supervision signal to the auto-encoder comes from the very input, and not from an annotation, auto-encoders are said to be trained in an unsupervised or self-supervised way. The fact that no annotated data is needed to train auto-encoders makes them very flexible and applicable in almost any task.

Auto-encoders have many uses, most uses have to do with the internal representations that auto-encoders learn about the items in the training data. The intermediate representations that auto-encoders learn in the intermediate layers can be used in place of the items they encode, this is specially useful when the auto-encoder architecture has a bottleneck, a section of the underlying computation graph that only allows a limited quantity of information to flow between layers. The intermediate representations learned in a bottlenecked layer of an auto-encoder need to compress as much information as possible about the input in a reduced space, which is usually designed so that it does not have enough space to fully capture all the information about the inputs, forcing the system to learn representations that capture only the most important aspects (see Figure 2.9). As a consequence the learned representations are denser and richer. Some examples of uses for these representations are: dimensionality reduction, denoising (with slight changes to the system) and feature extraction.

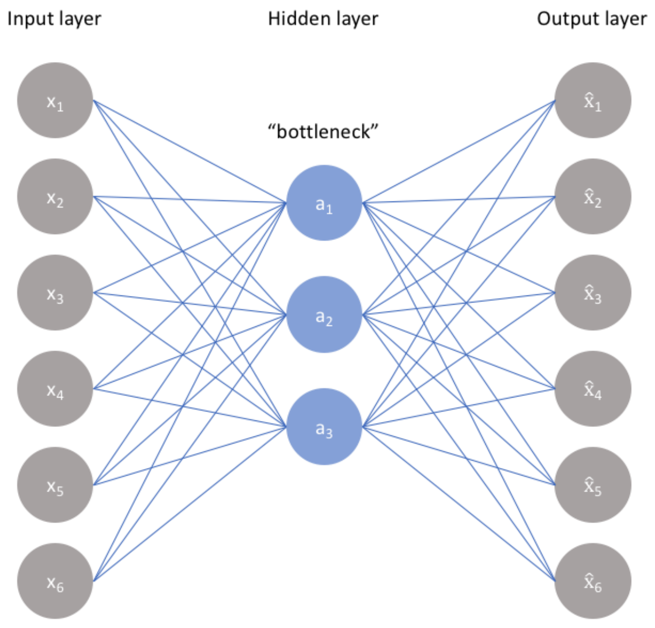


**Figure 2.9:** Diagram of s simple 2 layer multilayer perceptron auto-encoder. The first layer is a bottleneck that forces the intermediate representation to compress the information enriching it as a consequence.

Source: jeremyjordan.me

An auto-encoder, in principle, can be implemented as any DL architecture, the choice of architecture mostly comes down to the type of data that we want to train the auto-encoder on. For a language auto-encoder a seq2seq architecture is often used. For images in computer vision (CV, convolutional layers (learnable image filters) are followed by deconvolutional layers (opposite of convolutional layers). For anything other than those two, a multilayer perceptron with a bottleneck in the middle is used.

CHAPTER 3

# Annotation in semantic parsing

One of the biggest hurdles for progress in the area of semantic parsing is the fact that annotation of large, well curated, labeled corpora is very hard [Deriu et al., 2020]. Even in a day where there are plenty of opportunities to create large datasets fairly cheap, thanks to mass annotation platforms such as Amazon Mechanical Turk (AMT) and the annotation standards that have been set as a consequence, construction of good datasets for semantic parsing tasks still proves to be challenging.

In this chapter we explain the common problems that arise when creating a semantic parsing dataset, which make it hard to cost effectively build a dataset without taking quality compromises. We then describe two modern semantic parsing datasets (Spider and OTTA) that are relevant for this work to see what approach was used to mitigate the common problems, and how they compare.

## 3.1 Why is annotation hard?

Labeled data for semantic parsing tasks usually consists of sentences along logical-forms describing their meaning. For example, in NLIDB tasks such as text-to-SQL, questions posed in natural language are paired with SQL queries to a database, such that the result of the execution of the SQL query over the given database answers the question [Zhong et al., 2017].

### 3.1.1 Annotation task complexity

The problem is that dealing with logical-forms, either writing or reviewing them, is complex in nature, and not something anyone can do. For that reason, experts in the target formal language have to be recruited, which greatly limits the usability of mass annotation platforms such as AMT. At the same time, the time-cost per sample is going to be high (even for experts) compared to other annotation tasks. Finally, it is of especial importance to validate the correctness of the logical-forms, given that small mistakes will happen often, which means that several rounds of annotation are required for validation. As a consequence, construction of big semantic parsing datasets is very expensive in most cases.

### 3.1.2 Annotation bias

A more subtle problem, has to do with the so called annotation bias. Annotation bias is a well recognized problem with any process that requires crowdsourcing [Tommasi et al., 2017], and it is no exception when building semantic parsing datasets. The bias has a lot to do on how the annotation process is framed to the annotator. It is important to make explicit what it is expected from the crowdsourcers, and also to close any potential loopholes that the annotators might use to speed-up annotation at the cost of the quality of the annotations, as annotators usually get earnings per annotation.

To avoid wasting resources in excess, it is crucial to be diligent when designing the crowdsourcing setup. For instance, it is always a good idea to make small budget test runs to see if the annotation process produces the desired data, and launch the main annotation task then. It is also common practice to mix in validation samples to validate the quality of the annotations of crowdsourcers.

For some tasks such as text-to-SQL, the problem is about making a compromise: do we want the dataset to be representative of questions and queries that are used in the real world or would we rather have the questions uniformly capture all the possible queries that can be made to a database? Datasets representative of the real world, will contain simple queries that capture only a small portion of the database data. A dataset composed of simple queries is fine if the goal is to develop and deploy a system. However such a dataset is less interesting from the point of view of research, since the tasks that can be defined on top of it are not as challenging. A side effect of having mostly simple queries is that the system will probably not generalize to potential complex queries. A dataset representative of all the possible queries to the database might not be true to reality, but it presents a challenging task to research text-to-SQL systems, and tests the ability of systems to generalize.

With all this complexity, it is clear that we need to think thoroughly about the annotation design if we want to obtain a good quality dataset without expending more resources than needed. In the following sections the cases of two relevant datasets are reviewed, compared and contrasted.

## 3.2 ImageNet: The dataset that started it all

But, before that, let us make an inspirational detour. In the AI subfield of CV, image-recognition is perhaps one of the most studied tasks. Image recognition consists in determining whether or not images contain some specific object, feature, or activity. It is easy to see why this task is so important, as it is crucial for the development of many CV application [Kamavisdar et al., 2013]. For a very long time this task was addressed using classical CV techniques which involved signal-processing techniques, image filters, and hand-built heuristic paired with feature-based ML algorithms. The datasets at the time were very limited and often, domain specific. This led to systems that would perform well when tested on images similar to those in the training data, but would fail when applied to data that was slightly off the data distribution of the original dataset.

Because of the lack of an all-purpose image-recognition dataset, progress in the field had slowed down. That was before the release of ImageNet [Deng et al., 2009], a massive, multi million image dataset that contains images from a wide range of different domains, and has WordNet as a backbone (a lexical ontology) to provide labels for each image at different levels of coarseness.

Due to the size and complexity of the dataset, systems based on the old methods struggled with the ImageNet task, which involves labeling multiple million images in a couple thousand of classes, a very ambitious proposition at the time. The ImageNet benchmark gets reviewed every year in a very popular challenge known as ILSVRC [Berg et al., 2010] (ImageNet Large Scale Visual Recognition Challenge). This, together with the challenging nature of the task, motivated many research groups to keep improving the algorithms every year. The systems would improve on a yearly basis, but it was not until 2012 that a big leap was made. In the 2012 ILSVRC review, the system by the name of "AlexNet" [Krizhevsky et al., 2012] won (see Figure 3.1). It was a big deal because it was the first

Deep Learning system to get to the top of such a contested leaderboard, and it was able to beat the next system in the leaderboard by a considerable margin.

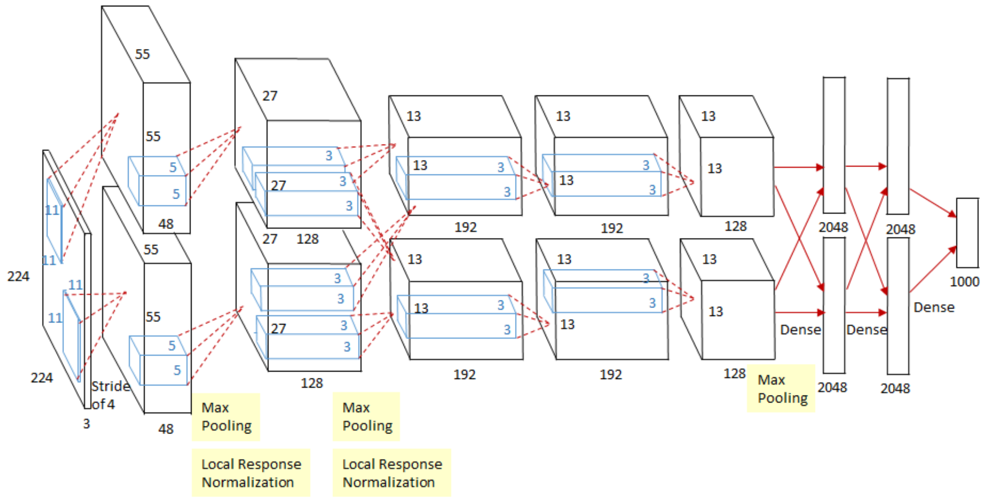


**Figure 3.1:** Diagram showing the AlexNet architecture

Source: Alexnet [Krizhevsky et al., 2012]

After the impressive showing of AlexNet, every winner of the ImageNet contest since, has been a system based on DL; bigger, deeper and more complex each year (see Figure 3.2). At some point it became clear that Deep Learning was the way forward. The Deep Learning phenomenon did not only affect ImageNet or computer vision, as it reached out to almost every subfield of AI, including NLP, and has been for the last decade the main focus of AI.

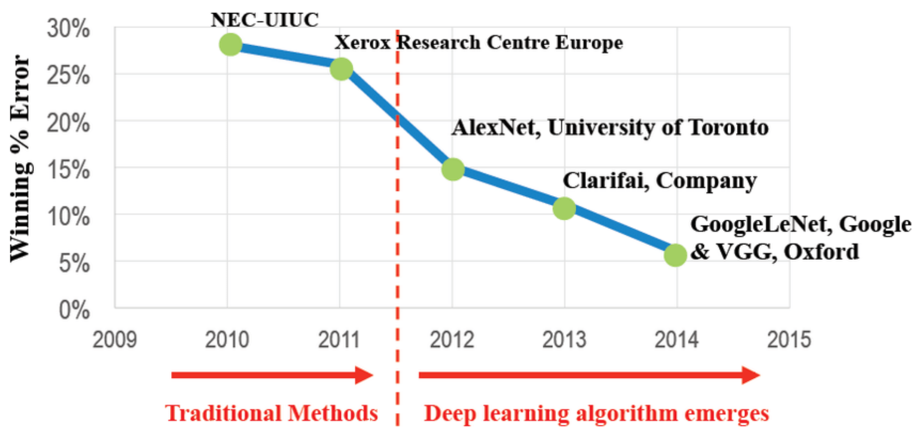


**Figure 3.2:** Progression of the SOTA systems in the ILSVRC challenge

Source: researchgate.net

ImageNet was able to provide an excellent platform for a task that was both of great interest and challenging, and by doing so it spawned a lot of competition and cooperation among researchers. Because of the complexity of the benchmark, the envelope of technology had to be pushed incredibly far and as a result Deep Learning emerged, a very powerful technique that has allowed for rapid progress in every field of study of AI.

Even today ImageNet is still very relevant, not only as a benchmark for some CV tasks (image recognition, image segmentation, scene classification, ...) but also as a resource for pre-training systems before training them for a more specific task involving perception on images [Huh et al., 2016]. Some tasks in which it is common to use ImageNet as a pre-training dataset are: image captioning and visual question answering.

## 3.3 Spider: The ImageNet of semantic parsing

Spider [Yu et al., 2018] is one of the largest text-to-SQL dataset, it contains more than 5,000 SQL queries paired with more than 10,000 questions. The aim of the Spider project is to create a text-to-SQL dataset that mitigates most of the drawbacks that previous datasets in the field have, and provide a platform/benchmark for the development of semantic parsing systems.

Because of the unheard-of amount of samples that this dataset contains compared to previous datasets, and the wide diversity of the queries, it may be said that Spider plays a similar role in semantic parsing to the role ImageNet played in CV.

Some of the key features that characterize Spider consist in but are not limited to:

- Number of SQL queries: Because of the cost of annotating SQL queries, most dataset authors choose to collect few SQL queries and augment the dataset by paraphrasing multiple times the question that corresponds to each logical form. Spider also uses paraphrasing to augment the size of the dataset, but limits the amount of repetitions of the same SQL query in the dataset to 2. Spider also contains more SQL queries to begin with, so the paraphrasing is more acceptable.

- Complexity of SQL queries: Also because the cost of annotation and because sometimes the database schema does not allow for complex queries (because the relations among the tables in the schema might be simple), datasets often only include very limited and almost trivial sets of queries. Spider contains many complex queries involving all kinds of common SQL patterns.

- Cross-domain: One of the biggest problems with datasets that preceded Spider is that queries were based on one or few database schemes. Having many schemes to pose the queries against (200 in the case of Spider) allows to train a system that better generalize to unseen database schemes, if trained in a zero-shot setting. Having cross-domain queries also makes the task more challenging which has the effect of empowering research in the field.

- Variety of queries: A side-effect of gathering samples from multiple databases of different kinds, is that the type of queries are also diverse. Spider contains both simple and complex queries, coming from simple and complex schemes, spanning very different domains and use-cases. Figure 3.3 shows how the variety of the queries in Spider compare to other popular text-to-SQL datasets.

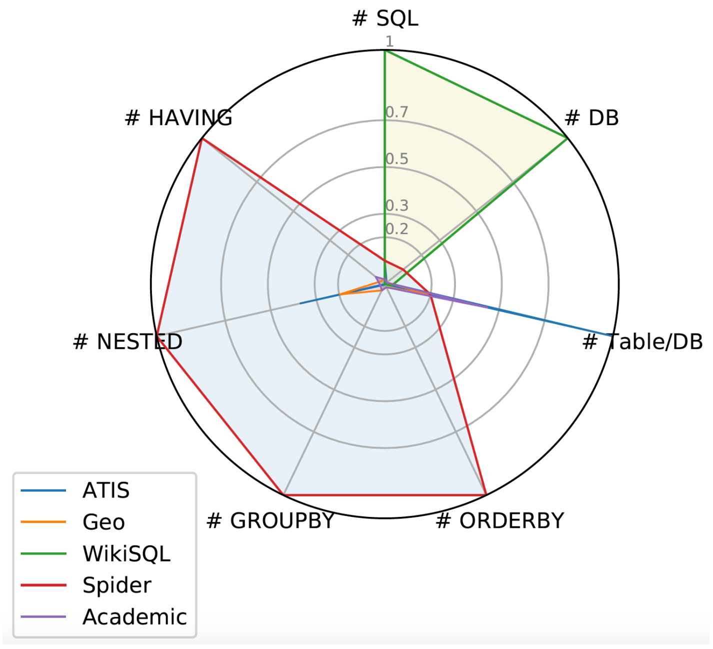


**Figure 3.3:** A chart showing the relative amount of queries of different types on popular text-to-SQL datasets. The value of 1 in each axis corresponds to the amount of queries of a given type that the dataset with the most queries of such type has.

Source: Spider [Yu et al., 2018]

The task proposed on top of Spider is in principle very similar to a standard text-to-SQL task. In essence, it consists in generating SQL queries from questions. However, because of the characteristics of this dataset in particular, there are a couple of differences, two to be precise.

First, unlike most other text-to-SQL datasets, SQL queries are not shared between the train and test splits. In previous work, each SQL query would appear multiple times in the dataset but each time it would be paired with a different question (with the same meaning). Two samples would be considered different even if the SQL queries were the same, as long as the questions that were paired with the queries would be different. Because of this design, it is possible to have the same SQL queries appear in both train and test splits, albeit with different paired questions.

Because queries would be shared between splits and the fact that the dataset contained few queries, allowed top performing systems to memorize the SQL queries, and use few lexical features about the question to resolve which memorized SQL query should be generated completely defeating the purpose of the task. The memorization phenomenon on the leading systems was considered before the release of Spider [Popescu et al., 2004], and the low performance of state-of-the-art systems on Spider made the suspicions clear.

Besides separating the queries so that there are no two equal queries from each database schema in both splits, Spider also reserves some database schemes solely for the test split, meaning that the database schemes in the test could not have been observed by the system at train time. Because of that, Spider requires to explicitly provide the database schema to the system, as opposed to learning the schema implicitly during train-time. This setting forces the system to understand the schema and to reason over the question even more so than with the standard split.

### 3.3.1 System evaluation

Since the text-to-SQL task is generative, evaluating systems is somewhat complicated. If you factor in the fact that the objects of generation are executable programs, evaluation becomes even trickier. There is no right way to evaluate semantic parsing systems since there is no easy approach to tell whether the output of the system is correct or not, unless the output is identical to the annotated in a gold standard in which case we know it is correct. However for many tasks including text-to-SQL, the gold standard might only be one of many possible correct outputs, and determining if the output generated by the system is equivalent to it, is as hard or harder than the task itself (theoretically speaking, proving that two SQL programs are the same is undecidable).

Even if it is not a perfect solution, Spider settles with three evaluation metrics that when jointly considered, give some idea of the actual performance of the system:

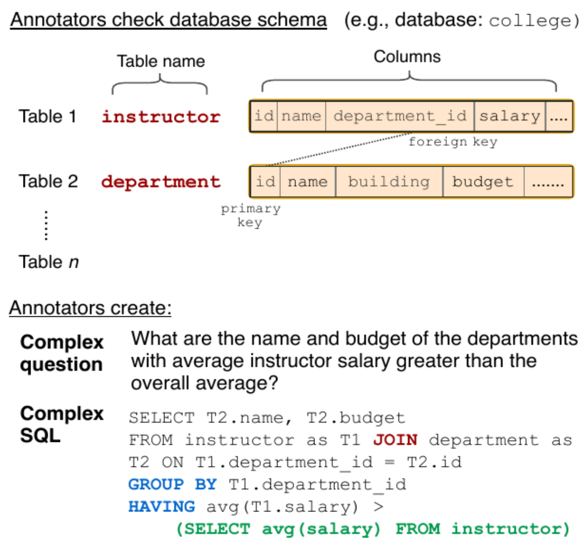
- **Component Matching:** The generated output is decomposed into its parts (each clause of the query), and the set of tokens in the components are compared without considering the order of appearance (because order does not matter within each clause). Each component is either equal or different (without in-between values) and the **Component Matching** score of each sample is the average of the components that are equal. This metric gives a notion of how close the output of the system is, to the annotated gold standard. However, it only compares equality at a token level, so it does not capture whether the query is correct from a meaning point of view.
- **Exact Matching:** like Component Matching, but instead of averaging the correct components of each sample, it requires all the components to be equal (a perfect Component Matching score) for the output to count as correct 1, otherwise the output is incorrect. This more strict method of evaluation gives a lower-bound on the systems performance since it only counts the outputs that are equivalent at the surface level, but misses out on programs that always give the same answer when executed.
- **Execution Accuracy:** the output is executed on a SQL engine and the results are compared to the results of executing the gold standard query. This will cover the queries that are equivalent but, unfortunately, because each database contains limited data, will also consider valid queries that are not equivalent but do produce the same results when run on the available data. Because of the inclusive nature of the metric (all correct queries are accepted but some incorrect queries are accepted too), the score will be an upper-bound on the real performance of the system.

Even if we can not obtain true system performance figures with the former methods, we can draw a picture of the output quality of a system, and more importantly, we can compare different systems which is essential for any development to happen.

### 3.3.2 A brute-force annotation process

Spider does not come without its problems though. First off, the annotation process could be better engineered. The process consists of a group of 11 college CS students writing 20-50 original question-SQL pairs for each dataset. After that the dataset is reviewed and a paraphrase of the question is added for each pair (see Figure 3.4).

There are two problems with this system: for one, the procedure took a grand-total of about 1000 person hours, which at 5600 produced queries, corresponds to 10 minutes per


Annotators check database schema (e.g., database: college)
Table name
Columns
Table 1 instructor
id name department_id salary ....
foreign key
Table 2 department
id name building budget .......
primary key
Table n
Annotators create:
Complex question
What are the name and budget of the departments with average instructor salary greater than the overall average?
Complex SQL

```
SELECT T2.name, T2.budget
FROM instructor as T1 JOIN department as
T2 ON T1.department_id = T2.id
GROUP BY T1.department_id
HAVING avg(T1.salary) >
    (SELECT avg(salary) FROM instructor)
```

**Figure 3.4:** An annotation sample in Spider.

Source: Spider [Yu et al., 2018]

query on average, too inefficient to be applied in potentially larger datasets, or projects with a small budget (see Figure 3.5). The second problem is that the queries do not cover the whole range of the database data because they are limited by the creativity of the students.

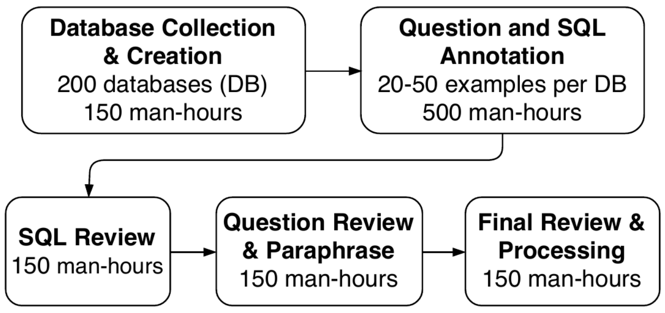


**Figure 3.5:** Diagram of the annotation process of Spider.

Source: Spider [Yu et al., 2018]

## 3.4 OTTA: A more productive annotation process

OTTA (Operation Trees and Token Assignment) [Deriu et al., 2020] is yet another semantic parsing dataset. While not as large as Spider, this dataset serves to showcase a clever annotation procedure that improves Spiders brute-force approach in several ways.

For one, it greatly speeds up the average annotation speed, from 5 minutes per sample to less than 2 minutes, a 3 fold improvement which dramatically cuts down costs and time. It also lowers the barrier of entry for annotators, which makes recruiting easier and crowdsourcing possible. Finally, the annotation procedure guarantees that queries covering the whole range of data on the databases are generated, unlike in Spider which is biased by the imagination of the students.

## 3.5 Annotation inverted, using Operation Trees

In order to implement the annotation process design, OTTA makes use of a custom intermediate representation called Operation Tree (OT) rather than SQL, an example is shown in Figure 3.6. An OT is a very simple tree-like representation for a query plan in a database, it is specially useful since there is a very simple and complete context free grammar (CFG) that captures the syntax of all possible OTs (see Figure 3.7).

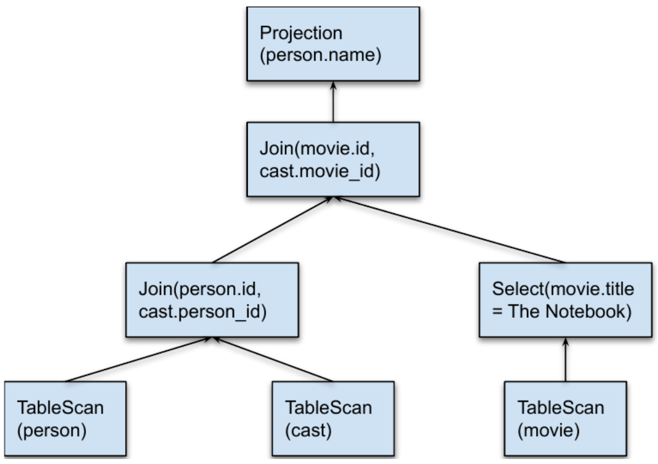
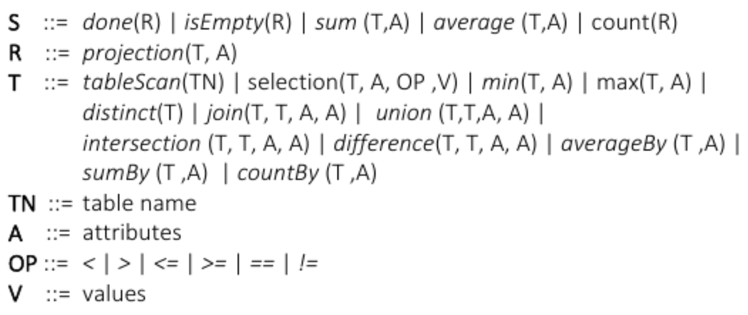

Projection (person.name)
Join(movie.id, cast.movie_id)
Join(person.id, cast.person_id)
Select(movie.title = The Notebook)
TableScan (person)
TableScan (cast)
TableScan (movie)

**Figure 3.6:** Example of an OT for the query ”Who starred in ’The Notebook’?".

Source: OTTA [Deriu et al., 2020]

The CFG allows any conceivable OT to be generated at will, which makes inverting the annotation process possible. Instead of asking the annotators to write logical forms for some given questions (or coming up with both, question and logical form), we can generate the OTs from the CFG and ask the annotators what question does each OT answer to. Even if there are some problems (such as meaningless OTs) with this procedure, the annotators no longer have to come up with the logical forms, which is considered to be the task that most hinders performance.



```
S   ::= done(R) | isEmpty(R) | sum (T,A) | average (T,A) | count(R)
R   ::= projection(T, A)
T   ::= tableScan(TN) | selection(T, A, OP ,V) | min(T, A) | max(T, A) |
        distinct(T) | join(T, T, A, A) |  union (T,T,A, A) |
        intersection (T, T, A, A) | difference(T, T, A, A) | averageBy (T ,A) |
        sumBy (T ,A)  | countBy (T ,A)
TN  ::= table name
A   ::= attributes
OP  ::= < | > | <= | >= | == | !=
V   ::= values
```

**Figure 3.7:** CFG that all OTs follow

Source: OTTA [Deriu et al., 2020]

Removing the annotation of logical forms from the annotation procedure has many benefits. The most obvious improvement, is that it is easier for crowdsourcers to come up with correct questions for OTs than it is to come up with proper OTs for questions, speeding up annotation as a consequence. The OT specification is also very simple, allowing people unfamiliar with it to take part in annotation. For this to be feasible, a few training or/and validation annotation rounds are pertinent, which requires to have a good annotation interface 3.8.

Another beneficial effect of the method, is that since the OTs are generated in a procedural manner, the distribution of queries can be controlled, which makes possible estimating what ranges of data in the database are covered by the queries. This fact can be exploited to design annotation tasks in a way that the whole range of data in the database is covered with uniform probability.

The use of an intermediate representation has additional benefits too. One such benefit, is that systems have been showed to perform better with more abstract query plan representations than the SQL standard [Guo et al., 2019]. A possible reason for this is that SQL contains a lot of implementation details that are unique to SQL and are not related to the semantic interpretation of the question, producing in that way a mismatch. Because of the mismatch, text-to-SQL systems have to learn how to implement a query plan in SQL on top of learning to give a semantic interpretation to the question. By using a more abstract formal language closer to the actual query plan, the gap is bridged between the conceptual task of reasoning about text (which is the focus of semantic parsing) and the specific task of converting utterances to database queries in a concrete language.

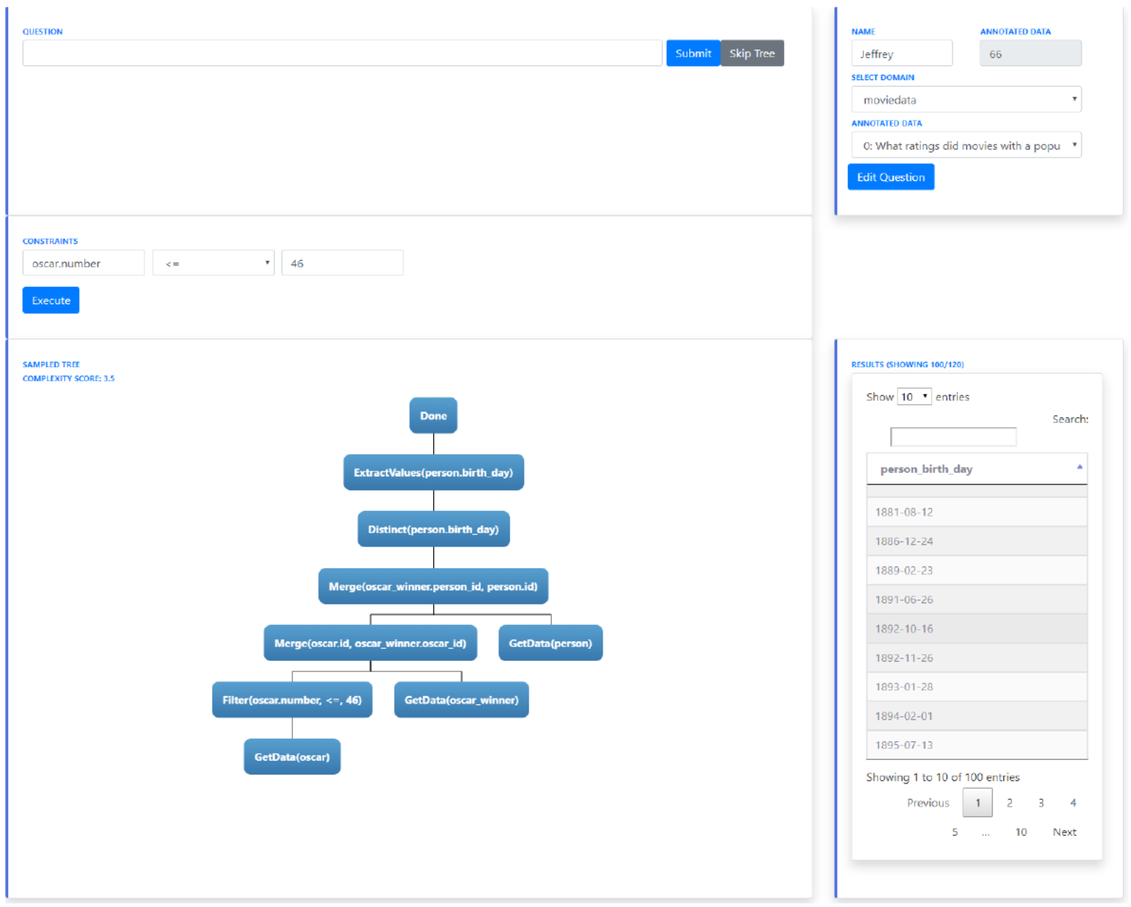
From the point of view of building a system, having a CFG that generates valid OTs from potentially the same data distribution as the distribution of the annotated pairs, can be a very powerful tool if used in a clever way. For example, a system or some parts of the system can be pretrained with as many sampled OTs as one desires before using the annotated trees in the training phase.

By pretraining the system on a large amount of non-annotated trees, the system can learn all about the formal language and its implementation details at the pretraining stage, without using the annotated samples which are very valuable compared with the ones we sample from the CFG [Kociský et al., 2016].

The goal is for the system to have experience with the formal language by the time it starts training with the annotated data. This way, instead of learning all the knowledge needed to generate logical forms (the implementation details of the formal language), it only needs to learn how to map a text description to a logical form (which is hard by itself) from the annotated data. By reducing the things that the system needs to learn from the relatively few annotated examples, the hope is that sample efficiency is boosted and as a consequence a better performing system can be trained for the same amount of annotated training data.

The authors of OTTA do not define a task *per se* because the focus of the contribution is on the annotation procedure. However, they do showcase a baseline system based on a state-of-the-art semantic parsing DL architecture, to prove that the dataset can be used to define a challenging task.

In their experimental setup, they evaluate the system using 5-fold cross-validation on the whole dataset, and for each fold they follow the same evaluation criteria as Spider. For the evaluation metrics, **Exact Matching** and **Execution Accuracy** are used which give lower and upper bounds for the actual performance of the system, **Component Matching** can not be used since it is a SQL specific metric, and even if it was adapted for OTs the results would not be comparable.


QUESTION
Submit
Skip Tree
Edit Question
Execute
Done
ExtractValues(person.birth_day)
Distinct(person.birth_day)
Merge(oscar_winner.person_id, person.id)
Merge(oscar.id, oscar_winner.oscar_id)
GetData(person)
Filter(oscar.number, <=, 46)
GetData(oscar_winner)
GetData(oscar)
Previous
Next

NAME: Jeffrey

ANNOTATED DATA: 66

SELECT DOMAIN: moviedata

ANNOTATED DATA: 0: What ratings did movies with a popu

CONSTRAINTS: oscar.number <= 46

SAMPLED TREE

COMPLEXITY SCORE: 3.5

RESULTS (SHOWING 100/120)

Show 10 entries

Search:

| person_birth_day |
|---|
| 1881-08-12 |
| 1886-12-24 |
| 1889-02-23 |
| 1891-06-26 |
| 1892-10-16 |
| 1892-11-26 |
| 1893-01-28 |
| 1894-02-01 |
| 1895-07-13 |

Showing 1 to 10 of 100 entries

1 2 3 4 5 ... 10

**Figure 3.8:** Annotation interface that was used for creating OTTA thought crowdsourcing

Source: OTTA [Deriu et al., 2020]

CHAPTER 4

# Systems for semantic parsing

When semantic parsing first started to be studied with a data-driven focus, the problem of parsing natural language text into a program (in a formal language) was treated as an instance of conditional language generation [Jia and Liang, 2016b], where the output was generated conditioned on the input.

This approach to semantic parsing had great success in the beginning. Language generation was already well established, and as a consequence there was a lot written as a foundation for future work. It was easy to take standard tools from the field (like recurrent neural networks and sequence to sequence architectures), and apply them to semantic parsing with minor tweaks [Iyer et al., 2017].

The language generation approaches could cope with the simpler datasets that were available early on. However, as harder datasets and tasks were proposed (such as Spider), it became clear that more specialised machinery was required [Krishnamurthy et al., 2017].

In this chapter, the evolution of the state-of-the-art in semantic parsing systems is described. All systems are based on the sequence to sequence (seq2seq) approach from language generation, which we explained in chapter 2 alongside recurrent neural networks (RNNs). First we explain how the seq2seq technique, which consumes and generates language, can be modified to generate logical forms and do semantic parsing instead. In followup sections, the innovations from this original architecture are motivated and explained in detail.

## 4.1 Seq2seq for semantic parsing

We have seen how seq2seq techniques work for tasks with natural language text as both input and output. In semantic parsing, the input sequence is going to be a natural language description and the output sequence a program in some formal language. To adapt seq2seq to semantic parsing, we need only modify the decoder so that it decodes logical forms instead of natural language sentences. Logical forms can be decoded sequentially [Iyer et al., 2017], in the same way as natural language sentences, however there are problems that prevent this approach from performing well.

Most important is the fact that the output vocabulary is often large and very heterogeneous. This is because the output vocabulary needs to include tokens from different sources such as: table-names, column-names, reserved words in the formal language as well as the whole input vocabulary (there might be string literals that need to be included in the output program).

Even if we are able to consider most of the words needed to generate queries in the output vocabulary, we would still experience low performance if we decoded each token the standard way (i.e. by selecting a token from the whole vocabulary at each time-step). The low performance is caused by merging these very different sources of lexicon on a single vocabulary. The fact that we are treating all tokens from different sources the same way, fails to capture the very specific nature of some tokens (such as language reserved words) and to make a distinction between entities (such as table or column names) and query values in the text utterance (such as a id numbers, dates, person names, ...).

In the following sections recent innovations in the field that boost the systems performance by tackling the aforementioned problems are outlined.

## 4.2 Type-safe seq2seq, constraining the productions with grammars

One elemental problem of the previous approach has to do with the fact that all the words from the output vocabulary are treated equally. At every time-step we ask the decoder to choose the next word from the whole vocabulary, however, in most cases when dealing with a formal language, only a few words from the vocabulary are valid given a previous tokens.

By failing to include information about the target language as prior knowledge, the system needs to learn this information at train time as well, lowering sample efficiency and limiting the performance of the end system as a result.

Type-safe approaches ensure that the system always produces syntactically valid outputs. With a sequential production mechanism like seq2seq, the system needs to know what productions are valid at each time-step. One way of explicitly including knowledge about the formal language structure, is using a context-free grammar that defines the syntax of the logical forms [Krishnamurthy et al., 2017].

Instead of starting from an empty sequence and building the program left to right by producing the next token given all previous tokens. We are going to start the program from the root of the grammar, and at each time-step we are going to select a rule to expand the leftmost non-terminal. Spatially, the program is being generated in a tree-like fashion by recursively branching from non-terminal tokens (that represent nodes of the tree) until terminals are produced (the leaves of the tree). A seq2seq model that decodes rules is shown in Figure 4.1 .

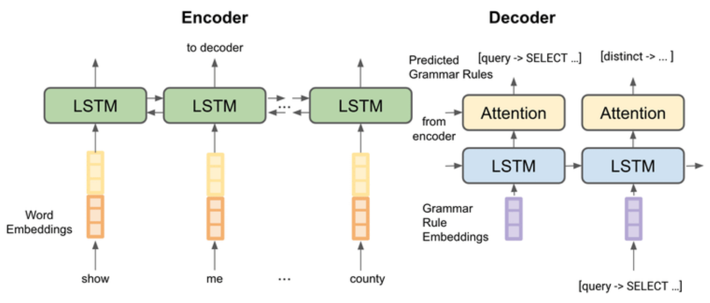


**Figure 4.1:** The architecture of a seq2seq model where the decoder has been modified to produce rules, which in turn generate a SQL query that can be used to query a database. This particular seq2seq model uses a bidirectional LSTM in the encoder and attention.

Source: Neural Semantic Parsing with Type Constraints [Krishnamurthy et al., 2017]

Because each non-terminal usually only has a few rules to expand, the decoder only needs to choose between the few type-correct choices and does not need to worry about incorrect options that would only introduce noise. Having a reduced set of choices greatly eases the task of the decoder which directly results in better performance.

## 4.3 Grammar-RNN: acknowledging tree structure

Using grammars and rule productions are big architectural improvements for semantic parsing. Despite this, new problems are introduced with the previous system formulation.

One such problem, is that the architecture is still sequential in essence while the production mechanism is now recursive, this results in a mismatch that can have negative consequences.

For instance, lets assume that a rule produces two or more non-terminals. The system then goes into expanding the leftmost non-terminal and keeps working on the left branch for many iterations until there are terminals at all the leaves of the left sub-tree. By the time the decoder starts decoding the next non-terminal of the original production, it might have already spent too much time on the leftmost sub-tree, potentially forgetting or disrupting any information that the decoder hidden-state would encode about the parent of the next non-terminal to expand. This is of-course undesirable, and has been shown to harm performance, especially when generating deep, complex sets of trees.

One way around this phenomenon, is to augment the decoder to keep track of the hidden states at each iteration and store the states in a tree structure that mimics that of the parse [Yin and Neubig, 2017] (can be implemented with a lookup table for the hidden states, and a pointer table that stores the parent of each node).

The system then needs to keep track of the position in the tree of the next node to expand. When it is time to expand a node, the decoder (implemented as an RNN) takes the previous computed hidden state and the input to the decoder, like usual, but it also retrieves additional information about the parent node in the parse tree. By combining these two sources of information (sequential and recursive), the decoder can make a more informative choice on the rule to use to expand the next non-terminal, often seen as a boost in performance.

To formalize this procedure, lets consider how the rule $y_t$ at time-step $t$ is decoded. To compute the rule production $y_t$ at time step $t$, we apply the the appropriate softmax layer (a softmax that only considers type-safe productions) to the hidden-state $h_t$ at time $t$.

$$z_{tj} = W_a h_t + b_a$$

$$y_t = \frac{e^{z_{tj}}}{\sum_{k=1}^{K_a} e^{z_{tk}}}$$

where $W_a, b_a$ are the weights and biases of the corresponding softmax layer and $K_a$ is the amount of elements in the softmax.

To compute $h_t$ we need the previous hidden-state and the embedding of the last produced rule $a_{t-1}$ at time-step $t$ (similar to a text decoder explained in "RNNs for NLP"), as in an ordinary RNN. Besides those two, we also want to include additional information about the parent non-terminal. There are many aspects about the parent that can be included as information, however, most systems that follow this design just include the hidden state of the parent $h_{P_t}$ and the embedding of the rule that produced the parent $a_{P_t}$. We call $P$ the pointer table that contains the indices $P_k$ of the parent node for each node $k$, with $k \leq t$. Note that $P_t < t$ has to be satisfied since the parent node is always produced before any of its children.

To use an RNN cell to process these different sources of information, we need to combine all the tensors to produce a total of two tensors, one will be fed as initial state to the RNN cell and the other as input. The initial-state of the RNN is usually the hidden-state $h_{t-1}$ produced by the previous RNN cell, all other tensors are concatenated to create the input tensor to the RNN. With all the details established, we can finally define how the hidden-state $h_t$ is computed in a given time-step $t$:

$$h_t = RNN([a_{t-1} : a_{P_t} : h_{P_t}], h_{t-1})$$

The RNN cell architecture can be any, as long as the interfaces only require a hidden state and an input. State-of-the-art systems will use additional information sources, such as attention vectors and node embeddings of the parent terminal (not the same as rule embeddings).

Figures 4.2 and 4.3 show the procedure of decoding a python program that sorts the list *my_list* in descending order using a RNN augmented with information from the tree structure. The continuous lines in Figure 4.3 correspond to the sequential flow of state in a traditional RNN that just decodes grammar rules and does not leverage tree structure. The dashed lines correspond to additional information provided to the decoder about the parent node state. In this example the production at time step $t_{14}$ happens long after its parent node gets produced (at $t_4$), without the additional information from the parent state, the RNN needs to remember information for a long period of time.

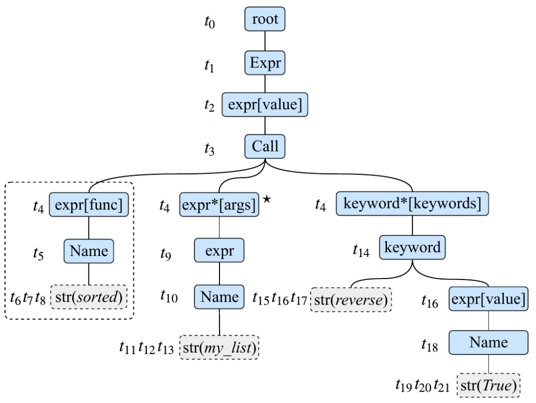


**Figure 4.2:** Abstract syntax tree of a python expression and the time-step in the decoding process that produces each tree node.

Source: A Syntactic Neural Model for General-Purpose Code Generation [Yin and Neubig, 2017]

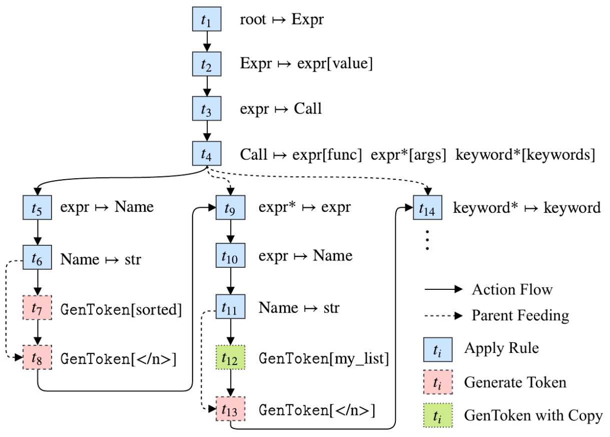


**Figure 4.3:** Unfolded Grammar-RNN aligned with the nodes in the AST that are expanded at each time-step

Source: A Syntactic Neural Model for General-Purpose Code Generation [Yin and Neubig, 2017]

## 4.4 Capitalizing on OTTA, improving performance through pre-training

It seems like most of the complexity in semantic parsing stems from handling logical forms. This is the case for both annotation and when designing a system for semantic parsing.

In annotation, annotators need to take a lot of time comparatively to come up with correct logical forms (compared to annotating in natural language, for example), and additional measures have to be taken to ensure that the annotation quality is acceptable.

From the point of view of system design, various sources of lexicon have to be carefully integrated. Additionally, the system needs to be designed for type-safety for performance to be acceptable.

Given that dealing with logical forms is the difficult part for the system, we would like to have as much supervision on the logical forms as possible. Collecting logical forms from humans is usually also the hard part for annotation, which limits the amount of samples that the system can see at train time. However, because of the way that the OTTA annotation task is designed, valid trees can be randomly sampled from the OT CFG with the same probability distribution as the trees that have been annotated by hand (since these trees have also been sampled).

Since we can sample as many trees as we want for practically zero cost, we can design the system training procedure to exploit this fact. For example, we could initialize the weights of the decoder with many non-annotated trees to make sure it learns good representations for trees before we train the system in the actual semantic parsing task.

A possible way to implement this pretraining, would be to train a tree auto-encoder with a lot of samples until the decoder can effectively decode a tree given a latent representation for it [Kociský et al., 2016]. Then, the decoder part of the model can be reused in the final text-to-tree system. The decoder will already know how to map midway representations to trees, so the only thing that is left is for the text encoder to learn to map the text to the same representations that the tree encoder produced for the corresponding trees. The weights of the decoder could be left frozen, however for better results it is advisable to continue learning the weights of the decoder and the encoder jointly.

# CHAPTER 5

# Experiments and results

In this chapter we describe experiments with a Grammar-RNN system on the OTTA dataset. We first review the official results from the OTTA dataset paper which uses this system as a state-of-the-art baseline. The results are based on cross-validation. We decided to set a more common evaluation setting, with train and test splits. We thus give our own set of results for the same exact system (running the same code and same model hyper-parameters) using the train/test split. Finally we run the same experiments on a refactored version of the deep learning model that we created to speed up the system in every aspect (training, evaluation and inference) by making better use of modern hardware resources such as graphic processing units (GPUs) and vector instructions in modern central processing units (CPUs). We report both task performance and efficiency numbers for the modified system.

Our results, while not directly comparable to the official results on the paper, are representative of the systems performance on a more common evaluation setting for a deep learning task. We also include further measurements of the results that are not present in the OTTA [Deriu et al., 2020] paper that are useful to better grasp the behaviour of the system; these include variability measures and results for the intermediate pretraining step.

Besides replicating results, we gathered supplementary data on the systems performance with different pretraining warm up runs and without warm up step at all (as outlined in chapter 4). This additional data allows to study the impact that the pretraining stage has on the systems final task performance.

## 5.1 Experimental setup

The evaluated systems are the Grammar-RNN deep learning model used in the OTTA paper as a baseline, and our refactored version of the same original system. Both have been trained in the settings used in the OTTA paper, that is: embedding-size $o = 128$, GRU hidden-size $h = 256$ and batch-size $b = 64$, the auto-encoder (in the pretraining phase) was trained for 3 epochs on a fixed set of sampled trees and the text-to-tree model for 100 epochs using early stopping. For evaluation beam search was used for sampling, with a beam size of $K = 15$.

The systems performance is evaluated according to the **Execution Accuracy** metric established in chapter 3, for the auto-encoder results we use **Exact Matching** instead.

In relation to the evaluation methodology, the results from the paper are obtained following a 5-fold cross-validation method, while our experiments have been carried out using a standard fixed train-test split strategy, with a split ratio of 0.8 train and 0.2 test (equal to a single fold in 5-fold cross-validation), the statistics for the test split can be observed in table 5.1.

| | Easy | Medium | Hard | Extra Hard | All |
|---|---|---|---|---|---|
| Moviedata | 78 | 74 | 53 | 58 | 139 |
| Chinook | 107 | 104 | 70 | 37 | 318 |
| College | 37 | 42 | 42 | 26 | 147 |
| Driving School | 58 | 32 | 13 | 10 | 113 |
| Formula 1 | 52 | 35 | 23 | 4 | 114 |

**Table 5.1:** Number of samples in the fixed test split for each database and difficulty setting.

There are two reasons to change the evaluation methodology: one is that fixed splits are commonly used to define deep learning tasks, as they make comparing systems fair. The other reason is that cross-validation evaluation is computationally very expensive, and using a fixed split allows us to perform many more experiments in the same amount of time.

By taking advantage of the faster evaluation speed, we were able to perform multiple runs of each experiment so as to compute statistically more meaningful results, by taking the mean of multiple runs and calculating the standard deviation between runs.

We report 3 sets of results in total, the first two sets of results correspond to an auto-encoder warm up run followed by 5 text-to-tree training and evaluation runs, all of which make use of the decoder weights learned in the one pretraining run. The last set also consists of 5 text-to-tree training and evaluation runs but entirely skips the pretraining and learns the decoder weights from scratch. By including two sets of results obtained with different pretraining runs and one without pretraining at all, we hope to give an overall intuition of the importance of this phase.

Each run involves training and testing a different system for each of the 5 database schemes available in OTTA.

## 5.2 Official results

Table 5.2 shows the results of the Grammar-RNN system reported in the OTTA paper.

| | Easy | Medium | Hard | Extra Hard | Weighted Average |
|---|---|---|---|---|---|
| Moviedata | 0.645 | 0.619 | 0.437 | 0.108 | 0.475 |
| Chinook | 0.610 | 0.442 | 0.396 | 0.482 | 0.473 |
| College | 0.525 | 0.739 | 0.294 | 0.077 | 0.468 |
| Driving School | 0.518 | 0.272 | 0.611 | 0.187 | 0.451 |
| Formula 1 | 0.355 | 0.075 | 0.0 | 0.0 | 0.263 |

**Table 5.2:** Execution Accuracy of the system in different databases and different query complexities as reported in the OTTA paper (cross-validation).

The aim of these results was to show that the dataset was challenging, which seems to be the case since the scores are not very high (perfect score of 1.00), specially considering that **Execution Accuracy** is an optimistic estimate of the actual performance of the system, and that a separate model was trained for each database which should make the task easier than if we only trained one model for all databases, or even considered a database split were some databases at test time are unobserved in the training.

There is distinctively very weak performance with the "Extra Hard" type queries, which was expected. The "Formula 1" database seems to be particularly challenging, which invites for further research into what is making the system lose performance in this specific database.

## 5.3 Replication results

In this section we present the results obtained in the train/test split with the original dataset code without any changes. We first show the results of the auto-encoder. Table 5.3 shows the Exact Matching scores obtained when evaluating the auto-encoders (A and B) learned in two different pretraining runs.

| | A | B |
|---|---|---|
| Moviedata | 0.336 | 0.185 |
| Chinook | 0.205 | 0.171 |
| College | 0.000 | 0.043 |
| Driving School | 0.000 | 0.003 |
| Formula 1 | 0.000 | 0.000 |

**Table 5.3:** Exact Matching score for two auto-encoder pretraining runs with the original code.

The auto-encoder results show great variability in the first two databases and very low performance on the last 3 databases. A possible explanation for these results is that the Exact Matching is hard. We also think that the auto-encoder training did not converge in the 3 training epochs, as we saw greater performance (up to 0.7 Exact Matching) on some of the databases in a few runs we did with more epochs. However we did not perform hyper-parameter exploration as the results would not be comparable with those reported in the paper and it would just spur more experimenting.

The following tables, table 5.4, 5.5 and 5.6, show the mean and standard deviation of scores of 5 training runs of the text-to-tree models. The tables correspond to the scores obtained when training the model based on pretraining of auto-encoder A, B and no pre-training.

| | Easy | Medium | Hard | Extra Hard | Weighted Average |
|---|---|---|---|---|---|
| Moviedata | 0.754 ±0.025 | 0.627 ±0.047 | 0.615 ±0.043 | 0.221 ±0.026 | 0.573 ±0.014 |
| Chinook | 0.015 ±0.014 | 0.085 ±0.087 | 0.066 ±0.090 | 0.027 ±0.060 | 0.050 ±0.050 |
| College | 0.714 ±0.037 | 0.733 ±0.057 | 0.491 ±0.049 | 0.200 ±0.110 | 0.565 ±0.026 |
| Driving School | 0.596 ±0.038 | 0.419 ±0.057 | 0.354 ±0.042 | 0.340 ±0.167 | 0.496 ±0.014 |
| Formula 1 | 0.427 ±0.069 | 0.143 ±0.073 | 0.043 ±0.044 | 0.000 ±0.000 | 0.247 ±0.032 |

**Table 5.4:** Execution Accuracy of the original system when trained on top of auto-encoder A taking into account different databases and different query complexities.

| | Easy | Medium | Hard | Extra Hard | Weighted Average |
|---|---|---|---|---|---|
| Moviedata | 0.751 ±0.046 | 0.584 ±0.022 | 0.604 ±0.019 | 0.259 ±0.066 | 0.567 ±0.022 |
| Chinook | 0.023 ±0.017 | 0.077 ±0.061 | 0.066 ±0.073 | 0.011 ±0.024 | 0.048 ±0.038 |
| College | 0.723 ±0.035 | 0.704 ±0.057 | 0.514 ±0.071 | 0.285 ±0.075 | 0.581 ±0.023 |
| Driving School | 0.579 ±0.085 | 0.356 ±0.057 | 0.354 ±0.069 | 0.280 ±0.084 | 0.464 ±0.049 |
| Formula 1 | 0.450 ±0.052 | 0.200 ±0.041 | 0.026 ±0.024 | 0.000 ±0.000 | 0.272 ±0.030 |

**Table 5.5:** Execution Accuracy of the original system when trained on top of auto-encoder B taking into account different databases and different query complexities.

| | Easy | Medium | Hard | Extra Hard | Weighted Average |
|---|---|---|---|---|---|
| Moviedata | 0.718 ±0.038 | 0.559 ±0.052 | 0.566 ±0.042 | 0.221 ±0.039 | 0.532 ±0.014 |
| Chinook | 0.053 ±0.077 | 0.062 ±0.082 | 0.057 ±0.073 | 0.000 ±0.000 | 0.039 ±0.044 |
| College | 0.649 ±0.043 | 0.709 ±0.031 | 0.557 ±0.036 | 0.277 ±0.096 | 0.574 ±0.013 |
| Driving School | 0.569 ±0.091 | 0.319 ±0.081 | 0.262 ±0.042 | 0.260 ±0.089 | 0.435 ±0.050 |
| Formula 1 | 0.338 ±0.104 | 0.149 ±0.055 | 0.017 ±0.039 | 0.000 ±0.000 | 0.204 ±0.064 |

**Table 5.6:** Execution Accuracy of the original system when trained without pretraining phase taking into account different databases and different query complexities.

The obtained results are overall substantially higher than the results reported in the paper. This is the case for most databases except for the "Chinook" dataset which had very bad results on all cases (a score of almost 0). The "Formula 1" dataset scores are very similar to those reported in the paper.

In terms of how the system performance correlates with the pretraining quality, the results were slightly better when training using a better performing auto-encoder (A), as shown by the slightly better results of table 5.4 vs 5.5. On the other hand, the results in Table 5.6 indicate that pretraining with the auto-encoder is beneficial.

## 5.4 Results after refactoring

In this section we show the same figures we presented for the original model, but when using our refactored version of the model. Table 5.7 is again the Exact Matching scores obtained when evaluating the auto-encoders.

The auto-encoder results of the refactored system are slightly better, however it is not possible to tell if this is due to chance or not with just two runs. Despite the higher overall scores, the relative scoring of the auto-encoder in different databases seems to be similar to that of the initial model.

| | A | B |
|---|---|---|
| Moviedata | 0.308 | 0.454 |
| Chinook | 0.204 | 0.242 |
| College | 0.051 | 0.029 |
| Driving School | 0.014 | 0.000 |
| Formula 1 | 0.000 | 0.004 |

**Table 5.7:** Exact Matching score for two auto-encoder pretraining runs with the refactored code.

Just like in the previous section, tables 5.8, 5.9 and 5.10 show the results of the system when using auto-encoder A, B or no auto-encoder as a pre-training, but using the refactored model.

| | Easy | Medium | Hard | Extra Hard | Weighted Average |
|---|---|---|---|---|---|
| Moviedata | 0.721 ±0.030 | 0.562 ±0.061 | 0.574 ±0.044 | 0.255 ±0.028 | 0.544 ±0.011 |
| Chinook | 0.002 ±0.004 | 0.004 ±0.008 | 0.011 ±0.019 | 0.000 ±0.000 | 0.004 ±0.006 |
| College | 0.617 ±0.064 | 0.662 ±0.062 | 0.443 ±0.113 | 0.200 ±0.088 | 0.506 ±0.055 |
| Driving School | 0.514 ±0.048 | 0.350 ±0.164 | 0.246 ±0.100 | 0.300 ±0.141 | 0.418 ±0.042 |
| Formula 1 | 0.165 ±0.062 | 0.120 ±0.042 | 0.000 ±0.000 | 0.000 ±0.000 | 0.112 ±0.036 |

**Table 5.8:** Execution Accuracy of the refactored system when trained on top of auto-encoder A taking into account different databases and different query complexities.

| | Easy | Medium | Hard | Extra Hard | Weighted Average |
|---|---|---|---|---|---|
| Moviedata | 0.743 ±0.031 | 0.584 ±0.063 | 0.562 ±0.021 | 0.217 ±0.042 | 0.546 ±0.027 |
| Chinook | 0.000 ±0.000 | 0.000 ±0.000 | 0.009 ±0.019 | 0.000 ±0.000 | 0.002 ±0.004 |
| College | 0.676 ±0.043 | 0.633 ±0.076 | 0.410 ±0.072 | 0.215 ±0.104 | 0.506 ±0.064 |
| Driving School | 0.586 ±0.021 | 0.331 ±0.052 | 0.323 ±0.127 | 0.220 ±0.084 | 0.451 ±0.025 |
| Formula 1 | 0.208 ±0.052 | 0.166 ±0.024 | 0.000 ±0.000 | 0.000 ±0.000 | 0.146 ±0.031 |

**Table 5.9:** Execution Accuracy of the refactored system when trained on top of auto-encoder B taking into account different databases and different query complexities.

We can deduce from the results that the refactored system is slightly less performant than the original one. The best results (auto-encoder A, table 5.8) are 3 points lower in Moviedata, 5 points lower in Chinook, 8 points lower in College and 7 points lower in Driving School and 13 point slower in Formula 1. The hypotheses about the correlation of the auto-encoder performance with the end-system performance do still apply with the refactored system, and the behaviour across the databases is similar to the original system. The qualitative similarities between the two sets of scores makes us believe that the refac-

| | Easy | Medium | Hard | Extra Hard | Weighted Average |
|---|---|---|---|---|---|
| Moviedata | 0.669 ±0.052 | 0.524 ±0.034 | 0.589 ±0.075 | 0.231 ±0.031 | 0.506 ±0.024 |
| Chinook | 0.000 ±0.000 | 0.000 ±0.000 | 0.000 ±0.000 | 0.000 ±0.000 | 0.000 ±0.00 |
| College | 0.600 ±0.062 | 0.638 ±0.097 | 0.348 ±0.080 | 0.115 ±0.072 | 0.453 ±0.025 |
| Driving School | 0.486 ±0.041 | 0.325 ±0.082 | 0.246 ±0.100 | 0.280 ±0.084 | 0.395 ±0.055 |
| Formula 1 | 0.208 ±0.079 | 0.126 ±0.092 | 0.009 ±0.019 | 0.000 ±0.000 | 0.135 ±0.053 |

**Table 5.10:** Execution Accuracy of the refactored system when trained without pretraining phase taking into account different databases and different query complexities.

tored system might contain a functional difference which is causing a little performance to leak.

## 5.5 Performance analysis

In this section we compare the training speeds of the original and the refactored systems, both in the auto-encoder warm up (table 5.11) and text-to-tree (table 5.12) training phases. We test both systems running in CPU and GPU and in a range of batch-size settings. The speed is measured in the average amount of samples per second processed in an epoch for the Moviedata database.

| | b=32 | b=64 | b=128 | b=256 | b=512 |
|---|---|---|---|---|---|
| Original CPU | 26.1 | 30.6 | 32.7 | 33.8 | 34.2 |
| Original GPU | 25.3 | 27.1 | 27.8 | 28.1 | 28.3 |
| Refactored CPU | 34.5 | 47.6 | 57.4 | 68.3 | 76.9 |
| Refactored GPU | 58.7 | 114.0 | 225.9 | 437.8 | 817.8 |

**Table 5.11:** Training speed ($samples/s$) in the pretraining phase, with and without the refactoring improvements and on different hardware platforms and batch-sizes $b$

| | b=32 | b=64 | b=128 | b=256 | b=512 |
|---|---|---|---|---|---|
| Original CPU | 37.5 | 47.2 | 49.8 | 53.7 | 57.3 |
| Original GPU | N/A | N/A | N/A | N/A | N/A |
| Refactored CPU | 45.3 | 66.4 | 91.9 | 118.4 | 154.5 |
| Refactored GPU | 74.7 | 140.6 | 281.3 | 549.8 | 1021.7 |

**Table 5.12:** Training speed ($samples/s$) in the training phase, with and without the refactoring improvements and on different hardware platforms and batch-sizes $b$

The results show that the refactored system can speed up the training by a factor of up to 20*x* when using a big enough batch-size. Running at the training settings used in the paper, the refactored systems yields 3 to 4 times faster training. Note that the original system was not prepared to be run on GPU, not only because it did not scale well running the auto-encoder but because it was not able to run at all the text-to-tree in the GPU.

# CHAPTER 6

# Conclusions and future work

In this chapter the conclusions of the project are looked at and directions for future work are proposed.

## 6.1 Conclusions

All in all, we have met all the objectives set in this undergraduate theses: 1) we have studied the literature on semantic parsing, dataset creation and state-of-the-art in deep learning systems; 2) We have installed, run and analyzed the characteristics of a state-of-the-art system in OTTA; 3) We have refactored the system with improvements to design and speed.

In this project we have explored the emerging field of semantic parsing, and in particular the task of NLIDB. To develop a strong understanding of the field: we have looked at the annotation process, we have discussed how DL can be applied to semantic parsing tasks, we have refactored a GrammarRNN system to increase its speed, and we have run experiments about these GrammarRNN systems in a NLIDB dataset called OTTA.

Regarding annotation, we explained the reasons why the annotation of semantic parsing datasets is challenging, namely that workforce for this type of annotation is scarce and that it is hard to get high annotation speeds; and we took the opportunity to compare the case of two very different NLIDB datasets, Spider and OTTA.

As far as DL systems for semantic parsing, we first briefly went through some of the background knowledge in DL and NLP that is necessary to understand the semantic parsing machinery. With the background knowledge covered, we explained how a common NLP technique (seq2seq) could be adapted to perform semantic-parsing. We concluded the discussion on semantic parsing systems talking about the innovations in DL techniques for semantic parsing (type-safe decoding and GrammarRNNs) that led to the current state-of-the-art.

About system refactoring, many parts of the original system were completely overhauled to accommodate the use of parallel hardware resources better. As a result, we were able to run the whole system in GPU with minor bottlenecks, which resulted in up to a 20 times speed-up compared with the original system. The refactored system however, showed measurably lower results, which led us to believe that a small bug was introduced somewhere in the refactoring process.

Multiple experiments were ran to compare the refactored system to the original, which helped to reach to the mentioned conclusions about the refactored system. Additional experimentation was also performed that led to evidence that initializing the weights of the GrammarRNN decoder by pre-training it as a part of a tree auto-encoder, increases task performance measurably. However the experiments were not able to conclude the extent to which the quality of the pre-training correlated with the performance of the GrammarRNN.

## 6.2 Future work

There are several avenues for future work to continue work beyond the scope of this undergraduate thesis, as follows:

- **Exploration into the results discrepancies of the refactored system:** As observed in the results section, the results of the refactored system are statistically distinct from the results of the original system. It stands to reason that there are some unintended functional differences which prevent the two systems from being functionally equivalent. Identifying these would give us some insight on the system, and what small details are important to get right, to obtain the best results possible.

- **Deeper analysis of the replicated results:** From the results we have also learned that our evaluation set-up produces very different results to the reference results reported in the OTTA paper. In most databases the replicated results are considerably superior, at the same time, there is a database (Chinook) in which the replicated results are very poor while the paper results for that database are at the same level of other databases. A deeper analysis of replication methodology should allow to uncover why this is the case.

- **Analysis of the performance in the Formula 1 database:** Performance of the system on the Formula 1 database is significantly lower than in any other database across all result tables. Discovering what aspect of this database is responsible for this drop in performance would be helpful to better understand the limitations and the behaviour of the system.
- **Study of the correlation between pre-training quality and system performance:** In our experiments we concluded that pre-training the GrammarRNN decoder translated to better task performance, however we left open the question regarding the importance of this phase in the whole training process. Understanding the correlation between the pre-training quality and the performance of the system when put together, can help design more effective training procedures.

Thanks to the refactoring, the system is now a good open base to pursue the following open research questions:

- **Language model as text encoder:** Language models (LM) [Bengio et al., 2003], such as BERT [Devlin et al., 2018], are DL models that have been trained on very large corpora of text in self-supervised tasks. Because of the massive amount of training data and (usually) the size of the models, LMs learn a very rich representation of language. The learned representation of language, make LMs exceptionally good for NLP tasks and they have become a popular tool to boost performance in tasks involving language. It would be interesting to see if performance of the GrammarRNN model can be boosted by replacing the RNN text encoder with a state-of-the-art LM.
- **Cross-domain NLIDB:** In the same way that Spider defines a cross-domain semantic parsing task by separating datasets in the train/test split, a similar task could be considered on top of OTTA (albeit with only 5 databases compared to Spiders 200). Such a task would require major additions to the GrammarRNN system since it would have to leverage database schemes as input, whereas now the schema is the same for each system and is handled implicitly.

# Appendices

APPENDIX A

# GrammarRNN refactoring

One of our main contributions to the LIHLITH project was refactoring the GrammarRNN system. The objective of refactoring the system was to have more solid groundwork to explore future work directions involving the GrammarRNN system. These directions would include: more experimentation, changes to the system involving a better text encoder based on language models and/or the addition of a database schema encoder to enable the system to perform domain transfer.

We considered that refactoring the system was necessary because the original design was made *ad hoc* for demonstrating the challenging nature of the dataset, and not for implementing a system which would be iterated on. Our design, on the contrary, has more resemblance to that of a production DL system as it makes extensive use of common patterns found in exemplary DL systems. As a byproduct of the change in design, the speed of the system improved by at least 3 times and as much as 20 times.

Refactoring the system also helped us get hands-on experience with it, and made us learn a lot of theoretical and practical aspects about these methods, which was also one of the main goals of the project.

In this chapter we narrate the most important changes made to the system in the refactoring process. For reference, both the original system and the refactored are written in python on top of the **PyTorch** DL library.

## A.1 Refactoring the decoder forward pass

Originally, the forward pass procedure of the decoder was implemented with a mix of tensor operations and python code. The tensor operations scale very well with parallel hardware resources, however the python code is sequential and must be run through a interpreter which makes it comparatively very slow. Because the proportion of time spent on parallel operations was not high enough, the scalability of the whole system to parallel hardware was hard-limited by the sequential python code. This was specially the case for the softmax layer selection implementation, lets recapitulate what the softmax layer selection is for.

In a type-safe seq2seq architecture we need to learn different softmax output layers, each softmax layer corresponds to a restricted set of productions that are valid in a given state of the generation process. The softmax layer that needs to be used has to be selected depending on the non-terminal that is going to be expanded. To implement this layer selection mechanism the original system would run through all the samples in the batch with a *for* loop, would check the type of non-terminal with a series of *if* statements and would apply the corresponding softmax layer one sample at a time. Besides introducing serial code overhead, computing softmax activations in a one by one sample basis parallelizes worse than if a whole batch of samples would be processed all at once.

The improvement we propose for this implementation works by assigning a type-id to each node (which we preprocess for training) unique for each type of node. These type-ids are then stored in tensors which can be accessed fast and in parallel (all the ids of a batch at once) and can be used to generate masks, which in turn are used to selectively apply the softmax layers (all nodes of the same type at once) to the samples. Profiling the system before and after shows that the forward pass is sped up by up to a factor of 20.

## A.2 Defining a solid machine learning pipeline

Another inconvenience of the original design is that there is no clear machine learning (ML) pipeline. A ML pipeline is a way of structuring a ML system to simplify the workflow of training, validating and operating the system. A ML pipeline typically consists of a series of well defined steps that are performed one after the other. ML pipeline steps typically include: preprocessing, training and evaluating.

The initial system had something similar to a pipeline structure but mixed some of the steps together which caused inefficiencies and loss of flexibility (because the design was to integrated). We defined a proper 3 step pipeline with preprocessing, training and evaluation steps. We took this opportunity to do more work in the preprocessing step by precomputing as much of the data as possible (node ids, id masks, parent ids, ...), this led to a small bump in performance of a couple of samples per second (depending on batch-size) in the training and testing phases.

## A.3 Faster beam search using batching

A big limitation of the system since the start, was that evaluating the system was very slow. This was specially a problem because we wanted to do early-stopping based on the development performance of the model, which requires to evaluate the model every few epochs. As a partial solution, we initially changed the evaluation frequency from evaluating every epoch to evaluating every five epochs, reducing the amount of system evaluations by a factor of 5. However evaluation was still expensive and we did not want to reduce the evaluation frequency further, so we ultimately decided to refactor the code responsible for the evaluation in the hopes of improving performance.

The main reason for the poor speed of evaluation was the use of beam search for sampling, which was implemented without speed in mind. The first beam search version only supported sampling the tree corresponding to one single question, which is less than ideal for scaling with the use of parallel hardware. We modified the implementation in order to applying beam search in batches of test data. Because the weights do not need to be updated in the test phase, batch-size is not a concern for the amount of gradient descent steps taken per epoch, that is why we were able to increase batch-size up to $b = 512$ using a GPU with a moderate frame-buffer size of 11GB. The change to batched beam search yielded a performance boost of $3x$.

# APPENDIX B

# Project objectives report

In this chapter we define the project in a explicit way. The project definition covers: an overall description of the project, the concrete goals of the project, and the planning and the methodology to achieve set goals. Finally we give a list of identified risks that can compromise the goal.

## B.1 Project description and goals

The main objective of this project is to get initiated in the line of research of deep learning techniques for semantic parsing. The concrete goals of the project are aligned to achieve this main objective. These concrete goals are: 1) Study the literature on semantic parsing, dataset creation and state-of-the-art (SOTA) in deep learning systems for semantic parsing; 2) Install, run and analyze the characteristics of a SOTA system in OTTA [Deriu et al., 2020], a recently released NLIDB dataset; 3) Refactor the system implementation to prepare it for future research work, and as a way to get familiar with the more practical aspects of DL techniques for semantic parsing. By accomplishing the aforementioned goals we expect that we end up in a position to pursue future research work.

## B.2 Project planning

### B.2.1 WBS diagram

We used a Work Breakdown Structure (WBS) to outline all the work that this project entails (Figure B.1). The project time estimates can be seen in Table B.1,

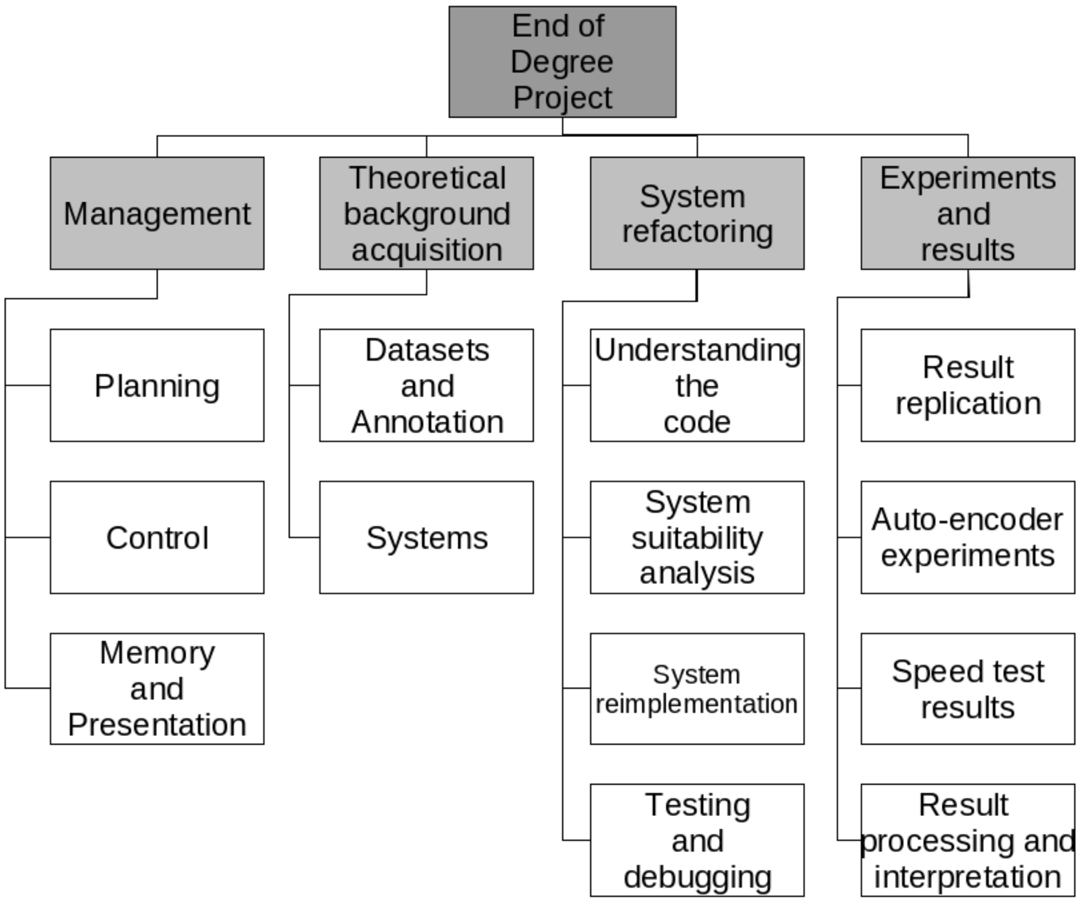


**Figure B.1:** Work Breakdown Structure of the project

### B.2.2 Work units

Next, we give brief descriptions of the work units and estimations of the time amount that each work unit we believe it will take. Because of the highly uncertain nature of some of these tasks (even though we tried to segregate the responsibilities in work units as much as possible) it is a possibility that the estimates could have great error.

### Planning

The work that is concerned with planning the project: what the project is about, what are the goals, what are the due tasks, how to carry the project out, what milestones have to be achieved and when are the deadline for those milestones, and what are the risks incurred in this project. The result being this report.

### Control

The duty of keeping the project focused and aligned with the goals through time, and making sure it stays according to plan or the plan evolves to refocus the project. This work is mostly performed in control meetings held every week and email communications.

### Memory and presentation

Writing the memory of the project that encompasses the work done in an expository way and the preparation of the defense of set work as a presentation which incurs material for the presentation.

### Datasets and annotation

Study of the literature having to do with semantic parsing datasets and annotation processes.

### Systems

Study of the literature having to do with deep learning semantic parsing systems. We will specially focus on the evolution of the methods and the current state-of-the-art.

### Understanding the code

Building an understanding of the implementation of one of the state-of-the-art systems. The understanding should be enough to iterate upon the design of system to implement potential improvements.

### System suitability analysis

Analyzing if the system is suitable for our requirements and what changes need to be made to cater it to our needs.

### System reimplementation

Reimplementing some aspects of the system according to the results of the suitability analysis, to meet our needs.

### Testing and debugging

Testing the refactored system from the functional aspect of correctness (the system still works and does what is intended to) and the aspects that the refactoring process touches.

### Result replication

Try to replicate the results from the OTTA project (which the state-of-the-art system is taken from) in both the original and refactored system.

### Auto-encoder experiments

Experiment with the auto-encoder pretraining stage, which is notoriously different when using the OTTA dataset. Analyze the benefits of pretraining an auto-encoder.

### Speed test results

Gather the results of speed comparisons between the original system and the refactored system.

### Result processing and interpretation

Process the results to get meaningful metrics (mean, stdev, ...), and give an interpretation to the results.

| **Work-unit** | **Time estimate (hours)** |
|---|---|
| **Management** | **150** |
| Planning | 5 |
| Control | 45 |
| Memory and presentation | 100 |
| **Theoretical background acquisition** | **100** |
| Datasets and annotation | 40 |
| Systems | 60 |
| **System refactoring** | **200** |
| Understanding the code | 15 |
| System suitability analysis | 5 |
| System reimplementation | 150 |
| Testing and debugging | 30 |
| **Experiments and results** | **50** |
| Result replication | 10 |
| Auto-encoder experiments | 20 |
| Speed test results | 5 |
| Result processing and interpretation | 15 |
| **Total** | **500** |

**Table B.1:** Time estimates for each work unit

### B.2.3 Milestones

Table B.2 shows the deadline dates for the deliverables.

| Deliverable | Date |
|---|---|
| Implementation | 2020-06-21 |
| Memory | 2020-06-21 |
| Presentation | 2020-06-29 |

**Table B.2:** Deliverables and their deadlines

## B.3 Methodology

This undergraduate thesis is carried out as a project of the IXA natural language processing research group. In particular the project is developed in the context of a European project called LIHILITH which the group is currently involved with. The student is receiving support from three IXA instructors (Eneko Agirre, Gorka Azkune and Aitor Soroa), two of which figure as project instructors. And has keep in contact and received support from the main author (Jan Deriu) of the OTTA project (which is itself part of LIHLITH). The student has had access to hardware resources in the form of servers nodes with CPU and GPU capabilities, also provided by the research group IXA.

### B.3.1 Meetings

Regular meetings are arranged in a fixed time slot every week with the three IXA instructors, the objective of these meetings is controlling the progress of the project. Additional meetings are arranged on-the-go with Jan Deriu to resolve specific doubts about their system. As the deadline gets closer the frequency of meetings will increase and the meetings will be arranged on demand.

### B.3.2 Work place

The student was assigned an office desk in one of the IXA offices in the faculty building. However, because of the COVID-19 situation the student is forced to work from home.

## B.4 Risks

Given the size and scope of the project, and the fact that there are many uncertainties (as is natural in a research project) the project incurs quite some risk. What follows is a list of sources of risk that were identified as the project started.

- **Research and novelty factors:** The project has a research component to it which is always a source of uncertainty. This makes hard coming up with accurate estimates to some of the work-units. This is specially the case because this area of semantic parsing in specific is new to the student and some of the instructors.
- **COVID-19 situation:** Also a source of uncertainty is the current situation of COVID-19, which can also have a big effect in the morale and routine, potentially becoming a detriment to the project.
- **Compute capabilities:** Deep learning is notorious for requiring high amounts of compute and memory resources to be applied. The student is given access to plenty

of hardware resources in the form of powerful server nodes. These capabilities however, need to be shared between many other researchers and it is difficult to say when the resources will be free and when the demand for them will be high.

# Bibliography

[Agirre Bengoa et al., 2019] Agirre Bengoa, E., Otegi, A., Pradel, C., Rosset, S., Peñas Padilla, A., and Cieliebak, M. (2019). Lihlith: Learning to interact with humans by lifelong interaction with humans.

[Bahdanau et al., 2014] Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate.

[Bengio et al., 2003] Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. (2003). A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155.

[Berg et al., 2010] Berg, A., Deng, J., and Fei-Fei, L. (2010). Large scale visual recognition challenge (ilsvrc), 2010. *URL http://www. image-net. org/challenges/LSVRC*, 3.

[Cho, 2016] Cho, K. (2016). Noisy parallel approximate decoding for conditional recurrent language model. *CoRR*, abs/1605.03835.

[Cho et al., 2014] Cho, K., van Merrienboer, B., Bahdanau, D., and Bengio, Y. (2014). On the properties of neural machine translation: Encoder-decoder approaches. *CoRR*, abs/1409.1259.

[Deng et al., 2009] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee.

[Deriu et al., 2020] Deriu, J., Mlynchyk, K., Schläpfer, P., Rodrigo, A., von Grünigen, D., Kaiser, N., Stockinger, K., Agirre, E., and Cieliebak, M. (2020). A methodology for creating question answering corpora using inverse data annotation. *arXiv preprint arXiv:2004.07633*.

[Devlin et al., 2018] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

[Dupond, 2019] Dupond, S. (2019). A thorough review on the current advance of neural network structures. *Annual Reviews in Control*, 14:200–230.

[Goller and Kuchler, 1996] Goller, C. and Kuchler, A. (1996). Learning task-dependent distributed representations by backpropagation through structure. In *Proceedings of International Conference on Neural Networks (ICNN'96)*, volume 1, pages 347–352 vol.1.

[Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep learning*. MIT press.

[Guo et al., 2019] Guo, J., Zhan, Z., Gao, Y., Xiao, Y., Lou, J., Liu, T., and Zhang, D. (2019). Towards complex text-to-sql in cross-domain database with intermediate representation. *CoRR*, abs/1905.08205.

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

[Huh et al., 2016] Huh, M., Agrawal, P., and Efros, A. A. (2016). What makes imagenet good for transfer learning? *CoRR*, abs/1608.08614.

[Iyer et al., 2017] Iyer, S., Konstas, I., Cheung, A., Krishnamurthy, J., and Zettlemoyer, L. (2017). Learning a neural semantic parser from user feedback. *CoRR*, abs/1704.08760.

[Jia and Liang, 2016a] Jia, R. and Liang, P. (2016a). Data recombination for neural semantic parsing.

[Jia and Liang, 2016b] Jia, R. and Liang, P. (2016b). Data recombination for neural semantic parsing. *CoRR*, abs/1606.03622.

[Józefowicz et al., 2016] Józefowicz, R., Vinyals, O., Schuster, M., Shazeer, N., and Wu, Y. (2016). Exploring the limits of language modeling. *CoRR*, abs/1602.02410.

[Kamavisdar et al., 2013] Kamavisdar, P., Saluja, S., and Agrawal, S. (2013). A survey on image classification approaches and techniques. *International Journal of Advanced Research in Computer and Communication Engineering*, 2(1):1005–1009.

[Kociský et al., 2016] Kociský, T., Melis, G., Grefenstette, E., Dyer, C., Ling, W., Blunsom, P., and Hermann, K. M. (2016). Semantic parsing with semi-supervised sequential autoencoders. *CoRR*, abs/1609.09315.

[Kramer, 1991] Kramer, M. A. (1991). Nonlinear principal component analysis using autoassociative neural networks. *AIChE Journal*, 37(2):233–243.

[Krishnamurthy et al., 2017] Krishnamurthy, J., Dasigi, P., and Gardner, M. (2017). Neural semantic parsing with type constraints for semi-structured tables. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1516–1526.

[Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.

[Li and Wu, 2014] Li, X. and Wu, X. (2014). Constructing long short-term memory based deep recurrent neural networks for large vocabulary speech recognition. *CoRR*, abs/1410.4281.

[MacCartney and Manning, 2009] MacCartney, B. and Manning, C. D. (2009). *Natural language inference*. Citeseer.

[Mikolov et al., 2013] Mikolov, T., Sutskever, I., Chen, K., Corrado, G., and Dean, J. (2013). Distributed representations of words and phrases and their compositionality. *CoRR*, abs/1310.4546.

[Popescu et al., 2004] Popescu, A.-M., Armanasu, A., Etzioni, O., Ko, D., and Yates, A. (2004). Modern natural language interfaces to databases: Composing statistical parsing with semantic tractability. In *Proceedings of the 20th International Conference on Computational Linguistics*, COLING '04, page 141–es, USA. Association for Computational Linguistics.

[Reinaldha and Widagdo, 2014] Reinaldha, F. and Widagdo, T. E. (2014). Natural language interfaces to database (nlidb): Question handling and unit conversion. In *2014 International Conference on Data and Software Engineering (ICODSE)*, pages 1–6.

[Russell and Norvig, 2002] Russell, S. and Norvig, P. (2002). Artificial intelligence: a modern approach.

[Sak et al., 2014] Sak, H., Senior, A. W., and Beaufays, F. (2014). Long short-term memory recurrent neural network architectures for large scale acoustic modeling.

[Socher et al., 2011] Socher, R., Lin, C. C., Manning, C., and Ng, A. Y. (2011). Parsing natural scenes and natural language with recursive neural networks. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 129–136.

[Stenning and Van Lambalgen, 2012] Stenning, K. and Van Lambalgen, M. (2012). *Human reasoning and cognitive science*. MIT Press.

[Sutskever et al., 2014a] Sutskever, I., Vinyals, O., and Le, Q. V. (2014a). Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215.

[Sutskever et al., 2014b] Sutskever, I., Vinyals, O., and Le, Q. V. (2014b). Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215.

[Sutskever et al., 2014c] Sutskever, I., Vinyals, O., and Le, Q. V. (2014c). Sequence to sequence learning with neural networks. In Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N. D., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 27*, pages 3104–3112. Curran Associates, Inc.

[Tommasi et al., 2017] Tommasi, T., Patricia, N., Caputo, B., and Tuytelaars, T. (2017). *A Deeper Look at Dataset Bias*, pages 37–55. Springer International Publishing, Cham.

[Wolfson et al., 2020] Wolfson, T., Geva, M., Gupta, A., Gardner, M., Goldberg, Y., Deutch, D., and Berant, J. (2020). Break it down: A question understanding benchmark. *Transactions of the Association for Computational Linguistics*, 8:183–198.

[Yin and Neubig, 2017] Yin, P. and Neubig, G. (2017). A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696*.

[Yu et al., 2018] Yu, T., Zhang, R., Yang, K., Yasunaga, M., Wang, D., Li, Z., Ma, J., Li, I., Yao, Q., Roman, S., et al. (2018). Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *arXiv preprint arXiv:1809.08887*.

[Zadeh, 1988] Zadeh, L. A. (1988). Fuzzy logic. *Computer*, 21(4):83–93.

[Zen and Sak, 2015] Zen, H. and Sak, H. (2015). Unidirectional long short-term memory recurrent neural network with recurrent output layer for low-latency speech synthesis. pages 4470–4474.

[Zhong et al., 2017] Zhong, V., Xiong, C., and Socher, R. (2017). Seq2sql: Generating structured queries from natural language using reinforcement learning. *CoRR*, abs/1709.00103.