Technology
Arts Sciences
TH Köln

DLR

Neural Semantic Parsing for Syntax-Aware Code Generation

Masterarbeit zur Erlangung des Master-Grades *Master of Science* im Studiengang Technische Informatik an der Fakultät für Informations-, Medien- und Elektrotechnik der Technischen Hochschule Köln

vorgelegt von:	Artur Baranowski
Matrikel-Nr.:	11130536
Adresse:	Heidestraße 144 51147 Köln
Erstprüfer:	Prof. Dr. Hubert Randerath
Zweitprüfer:	Prof. Dr. Nico Hochgeschwender

Köln, 23. September 2020

Abstract

The task of mapping natural language expressions to logical forms is referred to as semantic parsing. The syntax of logical forms that are based on programming or query languages, such as Python or SQL, is defined by a formal grammar. In this thesis, we present an efficient neural semantic parser that exploits the underlying grammar of logical forms to enforce well-formed expressions. We use an encoder-decoder model for sequence prediction. Syntactically valid programs are guaranteed by means of a bottom-up shift-reduce parser, that keeps track of the set of viable tokens at each decoding step. We show that the proposed model outperforms the standard encoder-decoder model across datasets and is competitive with comparable grammar-guided semantic parsing approaches.

Keywords: semantic parsing, machine learning, natural language processing, code generation, formal grammars, parsing theory

Statutory Declaration

I herewith declare that I have composed the present thesis myself and without use of any other than the cited sources and aids. Sentences or parts of sentences quoted literally are marked as such; other references with regard to the statement and scope are indicated by full details of the publications concerned. The thesis in the same or similar form has not been submitted to any examination body and has not been published. This thesis was not yet, even in part, used in another examination or as a course performance.

Ort, Datum	Rechtsverbindliche Unterschrift

Contents

List of Tables

List of Figures

1 Introduction

> « *The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it.* »
> **Mark Weiser**

1.1 Motivation

As envisioned by Weiser at the turn of the millennium, modern computers, partly by employing advanced user interfaces using speech or gesture recognition technology, enable invisible and ubiquitous interactions within complex environments [1]. Smartphones make the world's knowledge accessible at the touch of a button, anytime, anywhere, as if they were natural extensions of the human mind [2]. Intelligent agents, such as Apple's Siri or Amazon's Echo, perform tasks and provide services to individuals in a personalized way.

Much of this progress is driven by ever-increasing amounts of data and novel approaches to representation and reinforcement learning, in which hidden patterns are extracted from large data corpora. In light of an increasingly complex world with unmanageable amounts of data generated daily, these powerful trainable algorithms are vital for mastering today's challenges. They help us compress and interpret data by showing us which bits of information are relevant and which are not. In some cases, they use and augment large-scale knowledge bases and help scientists and engineers to analyze and understand difficult problems [3].

Extensive efforts at the Institute for Software Technology at the German Aerospace Center (DLR) focus on capturing and analyzing data on software systems and the development processes from which they emerge [4], [5]. Software repositories, mailing lists, or Continuous Integration logs serve as data sources for large databases. Building upon information from those databases, further complex analysis and visualization tools help software engineers understand and evaluate convoluted software architectures [6]. These applications have ease-of-use and accessibility in mind, aiming at shifting the focus from information retrieval to information analysis. However, challenges may arise due to the complexity of unconventional query languages or peculiar database schemas. A conversational interface was developed at DLR to address these issues, aiming at deducing system responses from natural language utterances and gesture interactions [7]. The

conversational system maps user utterances to a set of predefined global intents, which in turn are mapped to corresponding query templates executed on a backend database (see table 1.1). The conversational system is a generic database interface that can serve any client application that implements the interface.

Input Utterance	" Find the class with the highest number of methods inside the bundle `Core Comp`. "

Extracted Intent	`{`
	`  intent: max_number_of_methods,`
	`  entities: { BUNDLE_NAME: "Core Comp" }`
	`}`

Cypher Query Template	`MATCH`
	`  (b:Bundle {name:'@BUNDLE_NAME'})-[]->(c:Class),`
	`  (c)-[d:DECLARES]->(m:Method)`
	`RETURN`
	`  c, COUNT(m) ORDER BY COUNT(m) DESC LIMIT 1`

Table 1.1: Mapping from a natural language utterance to Cypher query template. A conversational system extracts intents and entities from natural language utterances at each dialogue turn and maps the result to a predefined query template.

However, approaching query generation using hand-crafted query templates does not scale well and proves to be inflexible. Each intent and its corresponding query have to be explicitly defined. The system has no way of responding to utterances that are out-of-distribution, that is, utterances that do not correspond to one of the predefined intents. Even worse, since any natural language utterance is inevitably classified and mapped to a particular intent, commands and utterances that are out-of-distribution are still mapped to one of the predefined intents. This leads to unpredictable and perplexing system behavior. A more flexible approach, where queries are generated directly from natural language descriptions, is desirable. It would eliminate the need for intent classification and allow the generation of tailored queries.

Natural Language Understanding (NLU) is the research area concerned with the extraction of structured information from natural language sources such as text or audio files. The main objective of NLU is semantic parsing. In its broadest sense, semantic parsing refers to the extraction of "meaning" from natural language. It aims at delivering granular descriptions of natural language expressions that go beyond simple intent and argument identification. They enable systems to perform complex tasks such as automated reasoning and code generation [8], [9].

These introductory considerations inspire the present thesis work, and the goal pursued is twofold. Firstly, contemporary approaches to code generation based on semantic parsing will be surveyed with the conversational system in mind. These studies are intended to instigate reflections and experiments for examining unexplored aspects of semantic parsing. Secondly, based on previous examinations, the groundwork for improving the conversational system by implementing a practical semantic parser for code generation shall be laid.

1.2 Semantic Parsing

In a nutshell, semantic parsing refers to the task of transforming natural language (NL) expressions into structured representations of their meaning. The basis for this is "Frege's Principle" of semantic compositionality, which states that the meaning of each syntactically complex expression depends on its constituent parts and how they are combined to form sentences [10]. Thus, these representations are usually based on some formal language or system capable of encoding the compositional structure of natural language expressions [11], [12]. For example, by means of logical connectives, conditional statements, quantified variables, and predicates, simple natural language expressions can be encoded in terms of first-order logic (see table 1.2). Such representations are often interchangably referred to as "meaning representations", "programs" or "logical forms". Common applications of semantic parsing include code generation [9], [13], automated reasoning [8], and question answering [14], [15].

Natural Language	" Gizmo loves Greta and Greta loves Gizmo. "
Logical Form	**Loves**(Gizmo, Greta) $\wedge$ **Loves**(Greta, Gizmo)

Table 1.2: The mutual love of Gizmo and Greta can be expressed by using a predicate symbol Loves and the logical connective $\wedge$.

The syntactic structure of meaning representations can be defined by a *formal grammar*, a set of rules that describes how the constituent parts of meaning representations are to be combined to form syntactically valid strings in the language on which they are based. For meaning representations based on programming languages such as Python or SQL, well-defined, efficiently parsable grammars can be provided as a priori knowledge to a semantic parser (see section 2.2.1). Employing such representations is useful for application-specific tasks, such as interfacing with a database. More importantly, they enable stakeholders without appropriate technical backgrounds to use otherwise complicated interfaces. However, due to their relatively simple syntactic structures, programming languages are generally not the right tool to model the compositional structure of natural language expressions.

More expressive "mildly context-sensitive" grammar formalisms, such as Combinatory Categorial Grammars (CCG), can accurately describe linguistic phenomena that cannot be captured by context-free grammars [16]. The basic building blocks of CCGs are "categories" that are composed of both a syntactic and a semantic component. The syntactic component expresses the role that a component has in a sentence. It usually corresponds to a elementary or "combined" grammatical part of speech. The semantic component corresponds to some functional, specifying the type and directionality of the arguments of the category. For example, it is not evident why "Gizmo loves Greta" should translate to **Loves**(Gizmo, Greta) and not to **Loves**(Greta, Gizmo). Only when the hierarchical structure of the sentence is recognized can it be systematically translated. "Gizmo" is a noun phrase and "loves Greta" is a verb phrase, which in turn is composed of the transitive verb "loves" and the object "Greta". CCGs parse these hierarchical structures by composing more complex grammatical categories from simple syntactic elements.

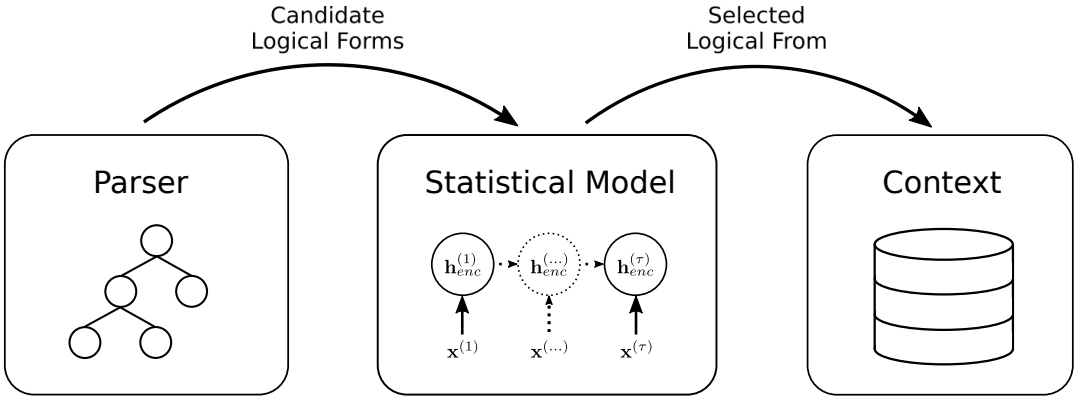


Figure 1.1: Traditional approaches to semantic parsing employ a grammar formalism to parse a set of candidate logical forms from natural language expressions. A statistical model picks the "most likely" logical from and executes it against a context, such as a knowledge base, to obtain a denotation.

The core of a modern semantic parser is a statistical *model* that provides the means to deal with the inherent ambiguity of natural language reliably. In conjunction with some grammar formalism, it defines how exactly natural language expressions are mapped to their corresponding logical forms. Traditional approaches use log-linear models [17] to produce a probability distribution over a set of candidate parses, generated deterministically using grammar formalisms such as CCGs, and choose the highest-scoring logical form. More recent approaches use encoder-decoder models (see section 2.1.5) that string together a single logical form sequentially by predicting its constituent parts piece by piece. In supervised learning settings, statistical models learn to choose or generate logical forms by modeling the statistics of a dataset of input-output examples. This is achieved by employing an optimization algorithm. Models based on neural networks usually employ some variant of gradient-based optimization, such as stochastic gradient descent (section 2.1.1. The "workhorse" behind optimization algorithms for supervised learning is the *backpropagation* algorithm [18].

Usually, predicted logical forms are coupled with an *execution environment* or *context*. For example, in the case of SQL queries, the context may be an underlying database on which predicted queries are executed to obtain the requested information. The results of logical forms that are executed in specific contexts are often referred to as *denotations*. Some semantic parsing approaches learn to produce logical forms exclusively from pairs of natural language utterances and corresponding denotations [15], [19]. In practice, different logical forms that are semantically equivalent yield the same denotations. Thus, instead of prescribing a specific logical form for each natural language expression, the model is free to learn the most appropriate logical form itself when learning from denotations.

1.3 Related Work

Early approaches to semantic parsing predominantly employed rule-based mappings based on pattern-matching [20] or syntactic parsers [21]. Systems based on syntactic parsers were able to reuse common syntactic patterns across multiple domains and achieve a moderate degree of transferability. Some approaches included semantic categories as nonterminal nodes, facilitating the mapping from parse trees to meaning representations by imposing semantic constraints on particular tree nodes [22]. Naturally, the introduction of domain-specific semantic knowledge made it difficult to transfer these systems to other domains.

With increasing computing power and larger quantities of available data, rule-based systems were superseded by probabilistic approaches. Zelle and Mooney were among the first to employ supervised learning strategies in semantic parsing [23]. The proposed model learned to parse meaning representations from a corpus of pairs of natural language utterances and corresponding logical forms. They used an algorithm based on inductive logic programming that learns rules to control a shift-reduce parser's actions. Likewise, Zettlemoyer and Collins proposed a model that learns mappings from natural language sentences to lambda-calculus expressions [11]. They employed combinatory categorial grammars with lexica of word categories, generating a set of possible parses for a given sentence. To resolve this ambiguity, a log-linear model was used for ranking the possible parses according to their likelihood.

More recently, *sequence-to-sequence* models, also referred to as *encoder-decoder* models, have been successfully applied to a wide range of sequence modeling tasks, such as machine translation [24], [25] or text summarization [26]. As the name suggests, they consist of two separate recurrent neural networks, an *encoder* and a *decoder*. The encoder network reads an input sequence and outputs a so-called "context", a vector-valued representation of its input. Conditioned on the context, a decoder network generates outputs from a finite set of elements. Concerning semantic parsing, their main advantage is that they learn mappings from input utterances to meaning representations "end-to-end", without having to rely on intermediate representations. This alleviates the need for manual feature engineering, including the definition of templates and lexica. However, this also excludes explicit a priori knowledge about the compositional structure of logical forms given by templates and lexica. Moreover, the encoder module compresses input information into a fixed-length vector representation, leading to information loss, especially for long sentences. The seminal work of Bahdanau et al. extended the encoder-decoder framework by an "attention" mechanism, almost universally improving model performance across domains and tasks [27]. It enables the decoder to dynamically attend to relevant inputs in encoded sequences at each decoding step instead of relying on "lossy" compressed encoder outputs.

Dong and Lapata use an attention-enhanced encoder-decoder model for parsing logical forms from natural language utterances [12]. Conditioned on a linear combination of input encodings, the decoder generates output tokens sequentially. Problems may arise due to the inherently tree-like structure of logical forms since conventional encoder-

decoder networks model sequential data by design. In order to explicitly model the hierarchical structure of logical forms, Dong and Lapata propose a sequence-to-tree model. The encoder module remains faithful to the standard model proposed by Sutskever et al. [24]. However, the decoder generates logical forms in a top-down manner by introducing a nonterminal vocabulary token "<N>". Initially, the decoder generates a top-level sequence. For each nonterminal token in the sequence, the decoder generates a subsequence conditioned on the parent nonterminal token's latent representation. The decoder continues this procedure until no more sequences contain nonterminal tokens.

Dong and Lapata also noted that parsing mentions of real-world entities that rarely or never appear in a dataset pose a problem for encoder-decoder models. To alleviate this issue, some approaches employ anonymization schemes that replace named entities with typed placeholders [12], [28]. For example, the entity "Gizmo" may be replaced and anonymized by a placeholder "<PERSON>". Another method uses pointer networks [29] and circumvents this issue by copying tokens directly from the input sentence. Pointer networks generate a probability distribution over all encoder inputs, where the most likely input position corresponds to the pointer position. The pointer position may be used for deciding which tokens to copy from the input sequence. Jia and Liang introduce an attention-based copy mechanism that lets the decoder dynamically decide whether to generate a word from its output vocabulary or copy a token from the input sequence [30].

Enforcing grammatical restrictions in the decoder has also been recognized as useful, especially in the circumstances with scarce data [9], [31], [32]. Yin and Neubig propose a neural semantic parser that explicitly encodes the target language syntax as prior knowledge [9]. The model parses entire abstract syntax trees sequentially, starting from the root node and generating tree nodes in depth-first, left-to-right order. Similarly, Xiao et al. take a derivational viewpoint when decoding parse trees [31]. They predict leftmost derivation sequences, each uniquely associated with a corresponding derivation tree. Krishnamurty et al. additionally ensure that decoder predictions satisfy type constraints by providing a type-constrained grammar [32].

Naturally, the question of imposing decoder constraints to ensure well-formed expressions also arises with the problem of code generation. Here, the most recent approaches are dominated by encoder-decoder models with attention as well. Programming languages have a well-defined syntax that usually can be represented by a context-free grammar. While initial attempts ignored these underlying grammar rules [33], subsequent models explicitly included a grammar as a priori knowledge and were able to guarantee syntactically valid code [9], [31]. Rabinovich et al. propose a decoder that employs a separate module for each construct in the grammar [13]. The decoder generates an abstract syntax tree through mutual recursion between modules. At each decoding step, the decoder either generates a symbol or propagates the decoder state to another module.

2 Preliminaries

This chapter discusses the aspects of deep learning and syntactic analysis that form the basis for the semantic parsing approach proposed in this thesis work. The section on deep learning will cover the basic optimization strategy for supervised learning and neural sequence modeling techniques. Concrete models such as recurrent networks and encoder-decoder networks [24] based on them are presented. Reference is made to Goodfellow et al. for an exhaustive presentation of deep learning [34]. The section on syntactic parsing rests upon the presentation of syntactic analysis in the *Dragon Book* [35]. Necessary basics on formal grammars are recapitulated before presenting relevant parsing strategies based on LR parsers.

2.1 Deep Learning

Many artificial intelligence applications are based on hand-engineered features that provide a basis for performing a specific task. For example, such a task may be predicting the price of a house based on a particular set of features, such as the number of rooms, location, age, or condition. In practice, for more challenging tasks, such as object recognition in images, it is difficult to determine an appropriate set of features. For example, recognizing a cats' features from raw pixel data, such as a simplified arrangement of specific geometric shapes corresponding to ears, legs, etc., is itself a task complicated enough to justify the use of machine learning algorithms. *Deep Learning* is precisely the tool for learning both the mapping from features to outputs as well as the features themselves.

The fundamental building blocks of conventional deep learning models are arrays of linear predictor functions combining a set of coefficients with latent features to produce output values (equation 2.1). Given a vector of real-valued input features $\mathbf{h} \in \mathbb{R}^d$, the function weighs each input feature with a coefficient w_i, $i \in \mathbb{N} : 1 \leq i \leq d$. A bias term b represents the function response to an all-zero input. The weights $\mathbf{w}$ and bias b can be learned by means of a *learning algorithm* (see section 2.1.1). For convenience, we denote all learnable parameters of some model f by Θ. In combination with an (usually non-linear) activation function $a(z)$ they constitute the most elementary neural network algorithm, called *perceptron* [36] (see equation 2.2).

$$z(\mathbf{h}) = h_1 w_1 + ... + h_d w_d + b \tag{2.1}$$

$$f(\mathbf{h};\, \Theta) = a(z(\mathbf{x})) \tag{2.2}$$

We can learn mappings $f : \mathbb{R}^{d_1} \rightarrow \mathbb{R}^{d_2}$ by "stacking" multiple perceptrons with distinct parameters in a d_2-dimensional vector. Given an input $\mathbf{h}_0$, the model then outputs some transformed representation $\mathbf{h}_1$.

$$f(\mathbf{h}_0;\ \Theta) = \begin{bmatrix} f_1(\mathbf{h}_0;\ \theta_1) \\ ... \\ f_{d_2}(\mathbf{h}_0;\ \theta_{d_2}) \end{bmatrix} = \mathbf{h}_1 \tag{2.3}$$

The term *deep learning* is due to the practice of designing "deep" chained models with multiple such functions. Each function is referred to as a *layer* of the model, and L is the number of layers in the model (equation 2.3).

$$f(\mathbf{h}_0;\ \Theta) = f^{[L]}(\ f^{[L-1]}(\ f^{[L-2]}(...);\ \theta^{[L-1]});\ \theta^{L}) = \mathbf{h}_L \tag{2.4}$$

Equation 2.3 is an example of a *multilayer perceptron*, or *feed-forward neural network*. The "depth" of such a model is given by the number of functions in the chain. The n-th function is refered to as *n-th hidden layer* of model. The n-th output vector $\mathbf{h}_n$ is referred to as the *n-th hidden state* of model and $\mathbf{h}_L$ is its output. The initial hidden state $\mathbf{h}_0$ often corresponds to the raw input data, for example, the pixel values of an image of a cat or a dog. The final hidden state $\mathbf{h}_L$ is usually used for inference. For example, in multiclass classification, each element in $\mathbf{h}_L$ may correspond to an object label, "cat" or "dog", and the maximal element in $\mathbf{h}_L$ is the predicted label.

By learning an appropriate set of weights using a learning algorithm (section 2.1.1), the initial layers can extract basic patterns in the input data. Subsequent layers receive these basic representations as input and compose more complex patterns from them. This allows deep learning models to learn very complex input-output relationships such as a mapping from raw image pixel data to an object label. Commonly, the input-output behavior of a function is learned by means of *supervised learning* [37]. In supervised learning settings, a function f is modeled and approximated by a function f^* from a dataset of input-output pairs that are examples of the true behavior of f.

2.1.1 Optimization for Supervised Learning

Neural network algorithms generally involve an optimization procedure. It enables them to "learn" the statistics of a data-generating distribution from a dataset $\mathbb{D} = (\mathbf{d}^{(1)}, ..., \mathbf{d}^{(\tau)})$ of samples drawn from the same distribution. This is achieved by means of minimizing or maximizing a function J of the parameters Θ, called *loss function* or *objective function*. In case of supervised learning, each dataset element $\mathbf{d} \in \mathbb{D}$ is a pair $\mathbf{d} = (\mathbf{x}, \mathbf{y})$, where $\mathbf{x}$ is the input and $\mathbf{y}$ is the corresponding output of the function f we are trying to model. Usually, the loss J is defined as the sum of an per-example loss L (see equation 2.5). The arguments to L are the model prediction $\hat{\mathbf{y}} = f(\mathbf{x};\ \Theta)$ for an input $\mathbf{x}$ and the true observed label $\mathbf{y}$. Informally, L will assign a "cost" to model predictions $\hat{\mathbf{y}}$ depending on how well they approximate the corresponding true label $\mathbf{y}$. Each dataset

element $\mathbf{d} \in \mathbb{D}$ may be viewed as being drawn from a true conditional probability distribution $P_g(\mathbf{Y} \mid \mathbf{X})$. Each pair of input sample $\mathbf{x} \in \mathbf{X}$ and output sample $\mathbf{y} \in \mathbf{Y}$ are instances the jointly distributed random variables $\mathbf{X}$ and $\mathbf{Y}$.

$$J(\Theta) = \frac{1}{\tau} \sum_{i=1}^{\tau} L(f(\mathbf{x}^{(i)}; \Theta), \mathbf{y}^{(i)}) \tag{2.5}$$

The goal is to find a set of parameters Θ that minimizes the loss J. The specific loss function is application dependent. However, the optimization algorithm is most commonly based on maximizing the likelihood of the empirical distribution $\hat{P}_g$ defined by a dataset $\mathbb{D}$, drawn from P_g [38]. The model f defines a family of conditional distributions $P_m(\mathbf{Y} \mid \mathbf{X}; \Theta)$, indexed by Θ. Provided the true distribution P_g, or at least a useful approximation of it, lies within the family of distributions P_m, a maximum likelihood estimator can be used for finding optimal parameters with respect to a dataset $\mathbb{D}$ (equation 2.6). In practice, the product over many probabilities is prone to numerical underflow. A more convenient equivalent representation of equation 2.6 takes sum of logarithms of likelihoods (equation 2.7). Since the logarithm is a monotonic increasing function, the parameters Θ that maximize equation 2.6 also maximize equation 2.7. In effect, equations 2.6 and 2.7 maximize the likelihood of the observed data in $\mathbb{D}$.

$$\Theta^*_{MLE} = \arg\max_{\Theta} \prod_{i=1}^{\tau} P_m(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}; \Theta) \tag{2.6}$$

$$\Theta^*_{MLE} = \arg\max_{\Theta} \sum_{i=1}^{\tau} log[P_m(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}; \Theta)] \tag{2.7}$$

From equation 2.7 it is evident that maximizing the likelihood of the dataset is equivalent to minimizing the Kullback-Leibler (KL) divergence between the empirical distribution $\hat{P}_g$ and the model distribution P_m (equation 2.8) [38]. The empirical distribution is not a function of Θ. Thus, minimizing the KL divergence is equivalent to minimizing the negative log-likelihood, or *cross-entropy* (see equation 2.10).

$$\Theta^*_{KLD} = \arg\min_{\Theta} \sum_{i=1}^{\tau} log[\hat{P}_g(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)})] - log[P_m(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}; \Theta)] \tag{2.8}$$

$$\Theta^*_{NLL} = \arg\min_{\Theta} - \sum_{i=1}^{\tau} log[P_m(\mathbf{y}^{(i)} \mid \mathbf{x}^{(i)}; \Theta)] \tag{2.9}$$

Minimizing the negative log-likelihood is a widespread objective for sequence models. Assuming $\mathbf{X}$ and $\mathbf{Y}$ are discrete random variables, P_g is a categorical distribution. Minimizing the negative log-likelihood with respect to Θ is then equivalent to minimizing the categorical cross-entropy between the truly observed labels $\mathbf{y}$ in the empirical distribution and the predicted labels $\hat{\mathbf{y}}$ in the model distribution. We obtain the definition of negative log-likelihood loss in equation 2.10. The model output $\hat{\mathbf{y}}$ is a probability distribution over the discrete set of outcomes (for example the labels “cat” and “dog”). To ensure

the model output "mimicks" such a probability distribution, the L-th, final layer of the model must be a map $f^{[L]}: \mathbb{R}^{d_{L-1}} \rightarrow \mathbb{R}^{d_L}$, where d_L is the number of possible categories the random variable $\mathbf{Y}$ may take on. The output of the linear predictor functions (see equation 2.1) in the final layer are then normalized to a probability distribution using a softmax (equation 2.11) as activation function. Each position in the output vector $\hat{\mathbf{y}}$, associated with a particular outcome, is interpreted as the probability of that outcome. The true target label $\mathbf{y}$ may also be interpreted as a probability distribution with all of its probability mass concentrated in the position corresponding to the observed outcome.

$$L(f(\mathbf{x}^{(i)};\, \Theta),\, \mathbf{y}^{(i)}) = NLLL(\hat{\mathbf{y}}^{(i)},\, \mathbf{y}^{(i)}) = -\sum_{c=1}^{C} \mathbf{y}_c^{(i)} \cdot log(\hat{\mathbf{y}}_c^{(i)}) \tag{2.10}$$

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{c=1}^{C} e^{z_c}} \tag{2.11}$$

Thus defined, the objective function J can be iteratively minimized by computing its gradient with respect to the parameters Θ (equation 2.12). The gradient is a vector indicating the direction of steepest ascent at any given point on the surface defined by $J(\Theta)$, the negative gradient is the vector of steepest descent. Using the gradients, we can define a parameter update rule, where we take a small step in the direction of steepest descent at each iteration (equation 2.13). The magnitude of the step is defined by a parameter α. This optimization algorithm is referred to as *gradient descent*. The gradients can be efficiently computed using the *backpropagation* algorithm. We refer to Goodfellow et al. [18] for a presentation of the backpropagation algorithm. Most machine learning frame-works, such as PyTorch and TensorFlow implement differentiation engines that compute gradients automatically [39], [40].

$$\nabla_\Theta J(\Theta) = \frac{1}{\tau} \sum_{i=1}^{\tau} \nabla_\Theta L(f(\mathbf{x}^{(i)};\, \Theta),\, \mathbf{y}^{(i)}) \tag{2.12}$$

$$\Theta' = \Theta - \alpha \cdot \nabla_\Theta J(\Theta) \tag{2.13}$$

Calculating the gradients over the entire dataset $\mathbb{D}$ requires the evaluation of the model f on every example in the dataset. By employing *stochastic gradient descent* (SGD), the computational cost of the gradient descent algorithm can be reduced. The factor $\frac{1}{\tau}$ in equation 2.12 can be interpreted as computing an expectation. The expected gradient can be approximated by a small subsample of $\mathbb{D}$, called a *minibatch*. The advantage of SGD is that a fixed minibatch size can be defined. Then, the computational cost of performing an SGD update does not increase with the size of the training dataset. Naturally, for a fixed minibatch size m, the sample becomes less reliable as the dataset size increases.

2.1.2 Language Models

Many deep learning problems, such as neural machine translation and code generation, require processing of sequential data. Modeling the joint distribution of many

discrete random variables in a sequence, for example the words in a sentence, quickly becomes a challenge because of combinatorial explosion, i.e. the *curse of dimensionality* [41]. Consider a sequence of 10 consecutive elements drawn from a vocabulary V of size $|V|$. Naively, there are potentially $|V|^{10}$ combinations. Often, however, elements that are in close proximity in the sequence are statistically more dependent. A statistical *language model* takes advantage of this fact by modeling the conditional probability of a sequence element given all previous sequence elements. Then, the probability of a sequence is given by the chain rule of probability.

$$P(w_1, w_2, ..., w_{\tau-1}, w_\tau) = P(w_1)P(w_2 \mid w_1) \dots P(w_\tau \mid w_1, ..., w_{\tau-1}) \tag{2.14}$$

A *neural* language model uses continuous space embeddings of vocabulary elements that help alleviate the curse of dimensionality (see section 2.1.3). A model $f(\mathbf{x}; \Theta)$, taking sequential inputs $\mathbf{x} = (x_1, ..., x_\tau)$, and producing sequential outputs $\hat{\mathbf{y}} = (y_1, ..., y_\kappa)$, defines a probability distribution $P(y_i \mid \mathbf{x}, y_{<i}; \Theta)$ for each output sequence element y_i, $i \in \mathbb{N} : 1 \leq i \leq \kappa$. This conditional distribution can be learned by minimizing the negative log-likelihood for each output sequence element, as described in section 2.1.1. Note that this approach maximizes the likelihood for each sequence element locally. This "greedy" maximization approach does not maximize the likelihood of the sequence. The most likely sequence for a chain of probabilites, as described in equation 2.14, can be found by exhaustive search, which is usually computationally intractable. Instead heuristic search algorithms approximating an optimal solution, such as *beam search*, are usually employed [42].

2.1.3 Continuous Space Embeddings

Inputs must be encoded in a vector-valued representation in order to be processed by a neural network. Assuming inputs are drawn from a finite set V, called *vocabulary*, each element can be uniquely represented by a "one-hot" vector $e \in \mathbb{R}^{|V|}$, with a single non-zero element. However, since vocabulary sizes, especially for natural languages, are typically very large, this entails the "curse of dimensionality" [41]. Thus, one-hot encodings are usually impractical.

The field of distributional semantics is concerned with quantifying semantic similarities between words from their distribution in big corpora of language data. Distributional semantics offers a framework for obtaining dense (distributed) vector representations, where words are embedded into a vector space of dimensionality $d_e \ll |V|$. The assumption is that words with semantic proximity have similar distributions. This notion is encapsulated in John Firth's famous quote: "You shall know a word by the company it keeps" [43].

Local context window techniques, such as skip-grams [44], and global matrix factorization techniques, such as Latent Semantic Analysis (LSA) [45], are the main methods for obtaining word vectors. Word vectors from skip-gram models, such as WORD2VEC [44], can be viewed as predicting words that are likely to occur in a specific window c of a given center word w_0. Thus, words are represented in terms of neighboring words in

close proximity. On the other hand, matrix factorization techniques compute global co-occurrence matrices for a specific scope, for example, on a per-document basis. The global co-occurrence counts in the resulting matrix columns corresponding to each word form the words' vector representations. They are projected into lower-dimensional subspaces using factorization techniques such as Singular Value Decomposition.

Matrix factorization methods exploit the underlying language corpus's statistics more efficiently since they leverage global co-occurrence counts. In contrast, skip-gram models iterate over a language corpus window-by-window and perform separate parameter updates whenever the same words co-occur. However, empirical results show that window-based methods perform very well on word analogy tasks [46]. This is in line with the observation that solving analogies using simple linear translations yields surprisingly accurate results. For example, given the embedding e_{Russia} for the word "Russia", computing $e_{Russia} - e_{Moscow} + e_{Tokyo}$ may approximately yield the word embedding e_{Japan} for "Japan". The GloVe model [47] combines the strengths of both approaches.

2.1.4 Recurrent Neural Networks

Feed-forward neural networks (see equation 2.4) have limitations in that they only process inputs of fixed length. However, some input data, such as sentences in the English language, are inherently variable in length. Moreover, the individual functions in each layer do not share parameters. At each position, different features are learned that are specific to the respective input. Therefore, the model is sensitive to the ordering of the input. For example, words in natural language utterances such as "You must have patience" may be rearranged to "Patience you must have" while still retaining their meaning. A model that is robust under the assumption of rearranged inputs is desirable.

To process sequential input data, which may or may not be of fixed-length, *recurrent neural networks* (RNN) may be used. In contrast to feedforward networks, inputs are processed sequentially in several time steps. Feedback connections allow results from previous time steps to be fed back into the network, thus enabling sharing parameters across the input sequence. In practice, many RNN architectures are conceivable. Any network that allows for feedback connections may be referred to as recurrent. In the simplest case, a single-layer RNN is formalized as producing a sequence of hidden states recursively.

$$\mathbf{h}^{(t)} = f(\mathbf{h}^{(t-1)}, \mathbf{x}^{(t)}; \Theta) \tag{2.15}$$

An input sequence of values may be represented by $\mathbf{x} \in \mathbb{R}^{d_x \times \tau}$, where d_x is the dimension of an input tokens vector-valued embedding (see section 2.1.3) and τ is the length of the input sequence. Note that $\mathbf{x}$ represents a single input example, now comprised of τ tokens instead of a single vector and $\mathbf{x}^{(t)}$ is the t-th token in the sequence (see figure 2.1). As implied by equation 2.15, the network reuses its parameters θ and thus shares parameters across input positions 1, ..., τ. In most cases the model will include an output layer which generates an output at each time step t or at the end of the sequence. Since the final hidden state $\mathbf{h}^{(\tau)}$ of the input sequence is of fixed length and the input sequence

may be of variable length, $\mathbf{h}^{(\tau)}$ can be viewed as a lossy compression of $\mathbf{x}$. This may cause problems for long sequences with long-range dependencies. An encoder-decoder models with *attention* [27], as described in section 2.1.6, can mitigate these issues.

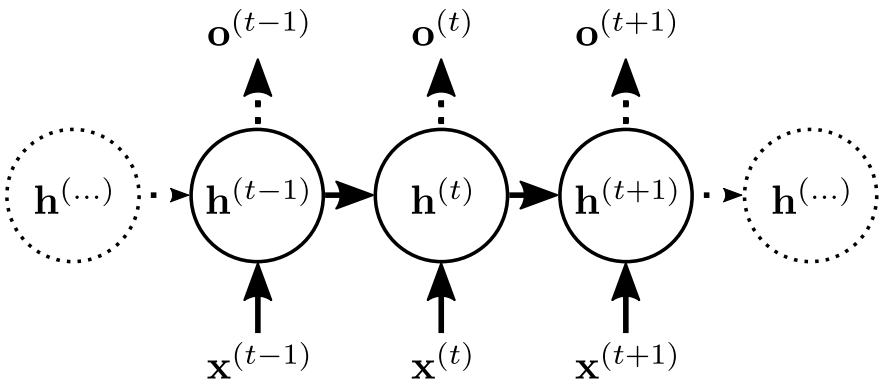


Figure 2.1: Basic single-layer recurrent neural network model. Each node represents a layer operation (similar to equation 2.3, except the additional token from the input sequence) with identical parameters θ, but different inputs. At each time step t the previous hidden state $\mathbf{h}^{(t-1)}$ and the sequence token $\mathbf{x}^{(t)}$ is fed into the layer. The network can act as a sequence transducer: at each time step, an output token $\mathbf{o}^{(t)}$ may be produced.

Bidirectional RNN

The presented RNN architecture leverages information from the current input $x^{(t)}$ and past input token $x^{(i)}$ at time steps $i < t$ to make a prediction $o^{(t)}$. However, some applications require information from the whole sequence. Whether the phrase "The appearance of Fibonacci sequences in nature is fascinating" refers to a mathematical object or an italian mathematician depends on input tokens that follow the word "Fibonacci". *Bidirectional* recurrent neural networks (BRNN) are employed in such cases [48]. They consist of a forward and a backward RNN. The forward RNN moves forward through the input sequence to compute the hidden states $\overrightarrow{\mathbf{h}}^{(t)}$. The backward RNN moves backward through the sequence to compute the hidden states $\overleftarrow{\mathbf{h}}^{(t)}$. The hidden state at time step t is defined as the concatenation $\mathbf{h}^{(t)} = [(\overrightarrow{\mathbf{h}}^{(t)})^\top; (\overleftarrow{\mathbf{h}}^{(\tau-t)})^\top]$ of the forward and backward hidden states.

Gated Recurrent Neural Networks

During backpropagation, gradients propagated over a long sequence of steps tend to become very small and vanish [27]. Therefore, basic RNNs often fail to model long-range dependencies such as in phrases like "The cats, hoping for food and ..., are scratching at my door". The word "are" depends on the plural inflection of the word "cats". However, a long subordinate clause separates these words. Recurrent neural networks have been shown to exhibit problems with such long-range dependencies because they weight recent input information more strongly [27]. Gated recurrent neural networks, such as *Long Short-Term Memory Networks* (LSTM) [49] and *Gated Recurrent Units* (GRU) [25] have proven to be useful in solving long-range dependencies. Both LSTMs and GRUs extend regular RNNs by additional neural network layers, so-called "gates", that interact with each other in a special way. In LSTMs, besides the hidden state, a so-called *cell state* is

produced, which acts as a kind of memory for the recurrent neural network. Depending on the current input token, the cell state is updated through an *update gate*, prompting the RNN to keep the input information in memory. A *forget gate* removes information from the cell state if it becomes redundant. This mechanism allows the LSTM to keep track of specific information, such as an internal representation of the word "cats", in memory and "recall" it later in the sequence. We refer to Hochreiter and Schmidhuber [49] for a thorough introduction to LSTMs.

2.1.5 Encoder-Decoder Models

In addition to the ability to process variable-length input, many sequence transduction tasks, such as semantic parsing or machine translation, require the generation of variable-length output. The *encoder-decoder* model (also referred to as *sequence-to-sequence* model) proposed by Sutskever et al. and Cho et al. [24], [25] permits to process inputs of variable length, where input sequences $\mathbf{x} \in \mathbb{R}^{d_x \times \tau}$ of length τ and output sequences $\hat{\mathbf{y}} \in \mathbb{R}^{d_y \times \kappa}$ of length κ may be of different length $\tau \neq \kappa$.

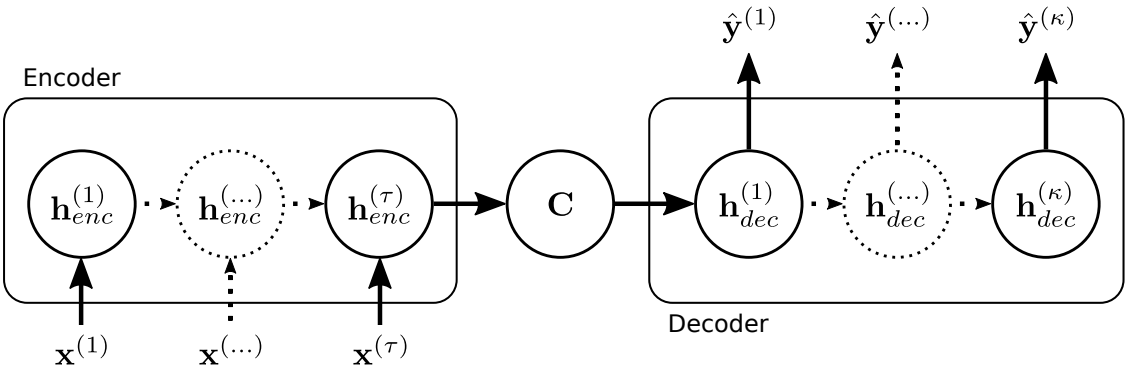


Figure 2.2: A encoder-decoder neural network composed of an encoder RNN, processing input tokens $(\mathbf{x}^{(1)}, ..., \mathbf{x}^{(\tau)})$ and a decoder RNN generating output tokens $(\hat{\mathbf{y}}^{(1)}, ..., \hat{\mathbf{y}}^{(\kappa)})$.

The architecture consists of two jointly trained RNNs. An encoder RNN processes all input tokens to produce a so-called context vector $\mathbf{C}$. Usually $\mathbf{C} \in \mathbb{R}^{d_{enc}}$ and $\mathbf{C} = \mathbf{h}_{enc}^{(\tau)}$, but the dimensions of the hidden states d_{enc} and d_{dec} do not have to be equal necessarily. A decoder RNN conditioned on the context vector $\mathbf{C}$, with $\mathbf{h}_{dec}^{(0)} = \mathbf{C}$, then emits an output sequence $\hat{\mathbf{y}}$.

2.1.6 Attention Mechanism

As noted in section 2.1.4, encoder-decoder models compress all input information into a fixed-length vector $\mathbf{h}_{enc}^{(\tau)}$. Since input sequences $\mathbf{x} \in \mathbb{R}^{d_x \times \tau}$ can be arbitrarily long, the representation $\mathbf{h}_{enc}^{(\tau)}$ of an input sequence becomes increasingly lossy with longer input sequences. This can be seen, for example, in neural machine translation, in increasingly low translation quality for longer sentences [25]. This issue can be mitigated by using information from *all* encoder hidden states $(\mathbf{h}_{enc}^{(1)}, ..., \mathbf{h}_{enc}^{(\tau)})$. By providing an *attention* mechanism, as proposed by Bahdanau et al. [27], the decoder can "construct" a context vector $\mathbf{c}$ from all encoder hidden states adaptively. At each decoding step, the decoder takes a linear combination of the encoder hidden states as context (see figure 2.3). Each encoder

hidden state is weighed by a scalar, derived from an *alignment model*.

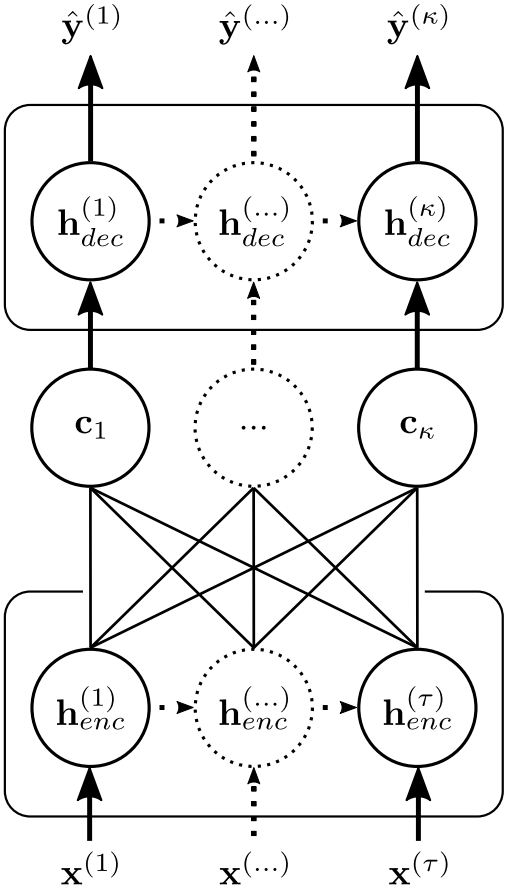


Figure 2.3: Encoder-decoder model with attention. For each time step at the decoder, a separate context vector $\mathbf{c}_i$ is computed, representing the alignment to the hidden state expected to best encode the information needed to decode the next output token.

Concretely, By using a bidirectional RNN (see section 2.1.4), each hidden state $\mathbf{h}_{enc}^{(t)} = [(\overrightarrow{\mathbf{h}}_{enc}^{(t)})^\top; (\overleftarrow{\mathbf{h}}_{enc}^{(\tau-t)})^\top]$ encodes information from the entire input sequence, with a focus on time step t. For each decoding step $i \in \mathbb{N} \cap [1, \kappa]$, the decoder computes a separate context vector $\mathbf{c}_i$ as the weighted sum of the encoder hidden states $\mathbf{h}_{enc}$ (equation 2.16). The weights $\boldsymbol{\alpha}_i$ at decoding step i are computed using an alignment model f_a (equation 2.18, with trainable parameters γ. The weights are obtained by normalizing the output of the alignment model by a softmax function (equation 2.17), such that $\sum_{t=1}^{\tau} \alpha_{it} = 1$.

$$\mathbf{c}_i = \sum_{t=1}^{\tau} \alpha_{it} \mathbf{h}_{enc}^{(t)} \tag{2.16}$$

$$\alpha_{it} = \frac{\exp(e_{it})}{\sum_{j=1}^{\tau} \exp(e_{ij})} \tag{2.17}$$

$$e_{it} = f_a(\mathbf{h}_{dec}^{(i-1)},\ \mathbf{h}_{enc}^{(t)};\ \gamma) \tag{2.18}$$

f_a can be defined, for example, as a simple single-layer feed-forward network. This allows the alignment model, the encoder RNN and the decoder RNN to be trained jointly.

Intuitively, f_a learns which input position $\mathbf{x}^{(t)}$, represented by $\mathbf{h}_{enc}^{(t)}$, a target output $\mathbf{y}^{(i)}$ is *aligned* to, as it will assign a high weight α_{it} to that encoder hidden state. For example, a neural machine translation model, translating the sentence "*The European Union is located primarily in Europe*" to "*L'Union européenne est principalement située en Europe*" might infer that the third output token "*européenne*" is translated from the second input token "*European*" and thus, aligns the context to the second encoder hidden state.

2.2 Syntactic Parsing

The parts of the compiler toolchain relevant for the practical purposes of this work are *Lexical Analyzers* and *Syntactic Parsers*, as they help us recognize some string s of input tokens as being a sentence of a language $L(G)$ generated by grammar G. Lexical analyzers, or *lexers*, take a raw string of characters containing a program's source code as input and output a stream of tokens that is sent to a parser for further processing. The lexical analyzer first identifies *lexemes* in the source program by matching it with predefined patterns. Patterns describe the forms that particular tokens may take and generally correspond to some regular expression. If a match is found, the lexer generates a token. A token's type corresponds to the lexical unit matched, and the value of a token is an instance of the associated pattern. In practice, lexers are automatically generated and employ finite-state automata for recognizing lexemes. For example, the lexer generator Lex defines a special notation for specifying regular expressions describing tokens. Lex takes this specification as input and generates a finite state automaton that recognizes the specified patterns [50]. The token stream thus created, is passed on to the parser for syntactic analysis. The reader may refer to Aho et al. [51] for a thorough exposition to lexical analysis.

Syntactic analysis is performed by a *parser* that verifies if some token stream s received from a lexer is part of a language $L(G)$ generated by a grammar G (see section 2.2.1). As the parser recognizes the completion of a production in the grammar, it constructs a *parse tree* that can be passed on to subsequent processing stages. Parsing approaches may be distinguished as being either top-down or bottom-up. Top-down parsers build parse trees starting from the tree's root node, whereas bottom-up parsers begin parsing at the terminal leaves of the tree. Practical parsers usually only parse subsets of context-free grammars. This is the case for LL(k) (left-to-right, leftmost derivation) parsers, top-down parsers with k tokens of lookahead. The lookahead defines how many tokens in the input buffer the parser can use to decide which production to apply. On the other hand, LR(k) (left-to-right, rightmost derivation) parsers construct parse trees bottom-up. LR(k) parsers are of great practical significance since they parse all deterministic context-free grammars in linear time and are relatively easy to implement.

Any form of semantic analysis, like type checking or object binding, is ignored in this thesis and left for future work. In the following, section 2.2.1 reviews the basic properties and terminology of context-free grammars. Next, section 2.2.2 discusses shift-reduced parsing, a parsing strategy that is fundamental to many bottom-up parsing algorithms. Finally, the sections 2.2.3 and 2.2.4 recapitulate prevalent parsing concepts based on LR parsing.

2.2.1 Context-Free Grammars

For the discussion of syntactic parsing, it is useful to recall a few basic definitions and properties of formal grammars, especially context-free grammars (CFGs). A context-free grammar is a set of recursive rules used to generate a context-free language.

Formally, a CFG is described by a 4-tuple $(N, T, P, \langle S \rangle)$:

— N is a finite set of *nonterminal* symbols, also referred to as *syntactic variables*, representing "placeholders" for patterns of terminal and nonterminal symbols that can be generated by them.
— T is a finite set of *terminal* symbols. These are the symbols that actually make up sentences of the language L generated by the CFG. V and T are disjoint sets.
— P is a finite set of *productions*. Productions are used to *derive* sentences of a language. Each production consists of a *left-hand side* consisting of a nonterminal symbol and a *right-hand side* consisting of a string of terminal and nonterminal symbols. The right-hand side describes the sequence with which the nonterminal symbol on the left-hand side can be replaced in the grammar.
— $\langle E \rangle$ is the start symbol of the grammar. It is a nonterminal symbol that derives all sentences of the CFG.

All production rules in context-free grammars are of the form $\langle A \rangle \rightarrow \alpha$. A standard example of a CFG is the so-called "expression grammar" [52] in figure 2.19. Uppercase letters in angle brackets, such as $\langle E \rangle$, represent nonterminal symbols. Uppercase letters, such as X, represent a single terminal or nonterminal symbol. Boldface strings, such as **id**, represent terminal symbols, and terms in single quotes represent literal terminals. Lowercase Greek letters, such as α or β, refer to strings of terminal and nonterminal grammar symbols. Lowercase alphabet letters, such as x and y, refer to strings of terminal symbols. We will refer to this example and notation throughout this section.

$$\begin{aligned} \langle E \rangle &\rightarrow \langle E \rangle \text{ `+' } \langle T \rangle \mid \langle T \rangle \\ \langle T \rangle &\rightarrow \langle T \rangle \text{ `*' } \langle F \rangle \mid \langle F \rangle \\ \langle F \rangle &\rightarrow \text{ `(' } \langle E \rangle \text{ `)' } \mid \textbf{id} \end{aligned} \tag{2.19}$$

We can use the grammar rules to *derive* strings of grammar symbols (see figure 2.20). In the expression grammar 2.19, $\langle E \rangle$ represents the start rule. We use the symbol $\Rightarrow$, to indicate one derivation step. For example, $\langle E \rangle \Rightarrow \langle E \rangle$ `+' $\langle T \rangle$ derives $\alpha = \langle E \rangle$ `+' $\langle T \rangle$ in one step. The symbol $\overset{*}{\Rightarrow}$ represents the reflexive, transitive closure of $\Rightarrow$, indicating that a string of grammar symbols can be derived in zero or more steps. A sequence of grammar symbols $\langle S \rangle \overset{*}{\Rightarrow} \alpha$, that can be derived from the start symbol S, is called *sentential form* of a grammar G. A sequence of terminals $\langle S \rangle \overset{*}{\Rightarrow} x$ derived from the start symbol is called *sentence* of G. The *language* $L(G)$ of a context-free grammar $G = (N, T, P, \langle S \rangle)$ is defined by the set of all sentences that can be derived from the start symbol S and we say the grammar G generates the language $L(G)$. Formally, $L(G) = \{ x \mid x \in T \ \wedge \ \langle S \rangle \overset{*}{\Rightarrow} x \}$.

$$\begin{aligned} &\langle E \rangle \Rightarrow \langle E \rangle \text{ `+' } \langle T \rangle \Rightarrow \langle E \rangle \text{ `+' } \langle F \rangle \Rightarrow \langle E \rangle \text{ `+' } \textbf{id} \Rightarrow \langle T \rangle \text{ `+' } \textbf{id} \\ &\Rightarrow \langle F \rangle \text{ `+' } \textbf{id} \Rightarrow \textbf{id} \text{ `+' } \textbf{id} \end{aligned} \tag{2.20}$$

$$\langle E \rangle \quad \overset{*}{\Rightarrow} \quad \mathbf{id}\ \text{`}+\text{'}\ \mathbf{id} \tag{2.21}$$

The derivation in 2.20 is an example of a *rightmost* derivation. In rightmost derivations, the rightmost nonterminal symbols are always replaced first. In contrast, in *leftmost* derivations, the leftmost nonterminal symbol is always replaced first. Each derivation γ in a rightmost (leftmost) derivation sequence is called *right-sentential* (*left-sentential*) form. The derivation sequence imposes a hierarchical structure, referred to as *parse tree*.

2.2.2 Shift-Reduce Parsing

All LR(k) parsers and their modifications are based on a bottom-up parsing technique called *shift-reduce* parsing. A shift-reduce parser maintains a stack holding grammar symbols and an input buffer with the stream of tokens obtained from a lexer. The input buffer is read from left-to-right, token by token. Each processed token is shifted onto the stack. When the parser recognizes a sequence of symbols α on top of the stack that corresponds to some production in the grammar, it reduces α to the nonterminal variable in the head of that production. This procedure is repeated until the parser has reduced the start symbol $\langle S \rangle$ of the grammar, and the input buffer is empty. In that case, the input buffer is verified as a sentence of the language $L(G)$ generated by G. Otherwise, the input is faulty and not accepted.

Stack	**Input**	**Action**
$	**id** + **id** $	SHIFT
$ **id**	+ **id** $	REDUCE, $\langle F \rangle \rightarrow$ **id**
$ $\langle F \rangle$	+ **id** $	REDUCE, $\langle T \rangle \rightarrow \langle F \rangle$
$ $\langle T \rangle$	+ **id** $	REDUCE, $\langle E \rangle \rightarrow \langle T \rangle$
$ $\langle E \rangle$	+ **id** $	SHIFT
$ $\langle E \rangle$ +	**id** $	SHIFT
$ $\langle E \rangle$ + **id**	$	REDUCE, $\langle F \rangle \rightarrow$ **id**
$ $\langle E \rangle$ + $\langle F \rangle$	$	REDUCE, $\langle T \rangle \rightarrow \langle F \rangle$
$ $\langle E \rangle$ + $\langle T \rangle$	$	REDUCE, $\langle E \rangle \rightarrow \langle E \rangle$ ' + ' $\langle T \rangle$
$ $\langle E \rangle$	$	ACCEPT

Table 2.1: Shift-reduce parsing example.

By means of *handle pruning*, a shift-reduce parser produces a rightmost derivation sequence in reverse. Consider a rightmost derivation sequence as in figure 2.22. A handle is a production $\langle A \rangle \rightarrow \beta$, where β appears at some position in a right-sentential form γ_i, such such that replacing β with $\langle A \rangle$ produces the previous right-sentential form γ_{i-1} in the rightmost derivation sequence. In other words, when $\langle S \rangle \underset{rm}{\overset{*}{\Rightarrow}} \alpha \langle A \rangle x \underset{rm}{\Rightarrow} \alpha \beta x$, then $\langle A \rangle \rightarrow \beta$ is a handle of $\alpha \beta x$. Note, that x in $\alpha \beta x$ is required to be a sequence of

terminal symbols.

$$\langle S \rangle \Rightarrow \gamma_1 \Rightarrow \ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = x \tag{2.22}$$

Note also, how each reduction in table 2.1 reverses a derivation in the sequence shown in figure 2.20. For an unambiguous grammar G, every right-sentential form has exactly one handle (i.e., a unique rightmost derivation). Then, for each right-sentential form γ_i in the rightmost derivation sequence, we can locate the body of the handle β_i and recover the right-sentential form γ_{i-1}, until we have recovered the start symbol $\langle S \rangle$. Deciding when to reduce and by which production to reduce are the critical decisions a shift-reduce parser has to make.

2.2.3 LR Parsing

LR(k) parsing is the most popular parsing technique in modern compilers. As a type of shift-reduce parser, LR parsers construct a rightmost derivation sequence in reverse during a left-to-right scan of the input sequence. An LR(k) parser uses k tokens of lookahead to make parsing decisions. This section is limited to the derivation of the canonical LR(0) item set. We refer to Aho et al. [53] for a derivation of the LR(0) and LR(1) parse tables.

In order to make parsing decisions, LR(k) parsers rely on so-called *items*. An LR(0) item is a string of the form $\langle A \rangle \rightarrow \alpha \cdot \beta$, with $\langle A \rangle \rightarrow \alpha\,\beta$ being a production in a grammar G. The dot indicates that the parser has already consumed a string of symbols derivable from α and expects a string derivable from β. An item $\langle A \rangle \rightarrow \alpha\,\beta \cdot$ indicates that a string of symbols $\alpha\,\beta$ derivable from $\langle A \rangle$ has been consumed and $\alpha\,\beta$ may be reduced to $\langle A \rangle$.

Algorithm 1 : CLOSURE(I_s)

INPUT Kernel items I_s of some state s.
OUTPUT Closure of I_s.

```
1: function CLOSURE(I_s)
2:     I_c ← ∅
3:     while |I_c| ≠ |I_s| do
4:         I_c ← I_s
5:         for each [ A → α · Bβ ] ∈ I_s do
6:             for each [ B → γ ] ∈ G' do
7:                 I_s ← I_s ∪ {[ B → ·γ ]}
8:     return I_s
```

LR(0) parsers employ a finite state automation based on a collection of *item sets*, called the *canonical* LR(0) collection. The canonical LR(0) collection for a grammar G can be derived using two functions CLOSURE and GOTO, and an augmented grammar G'. The augmented grammar defines a new start rule $\langle S' \rangle \rightarrow \langle S \rangle$. Reduction by this particular production indicates acceptance of the input string.

Assume I is a set of items for a grammar G. Closure(I) adds every item in I to the closure. Next, if $A \rightarrow \alpha \cdot B\beta$ is an item in Closure(I), and $B \rightarrow \gamma$ is a rule in in the grammar G', $B \rightarrow \cdot\gamma$ is added to the closure. This process is repeated until no more items are added to the closure (see algorithm 1).

Algorithm 2 : Goto(I_s, X)

Input Item set I_s of some state s.
A grammar symbol $X \in (N \cup T)$
Output Closure of I_s.

```
1: function Goto(I_s, X)
2:     I_g ← ∅
3:     for each [ A → α · Xβ ] ∈ I_s do
4:         I_g ← I_g ∪ {[ A → αX · β ]}
5:     return Closure(I_g)
```

The Goto(I_s, X) function defines the state transitions in the LR(0) state machine. I_s is a set of items (representing a state in the LR(0) automaton) and X is a grammar symbol. The "transition" to another state (item set) is performed on the grammar symbol X. Goto(I_s, X) is the closure of all items $A \rightarrow \alpha X \cdot \beta$ for all items $A \rightarrow \alpha \cdot X\beta$ in I_s (see algorithm 2).

Algorithm 3 : Collection(G')

Input Augmented grammar G'.
Output Canonical collection of LR(0) items C.

```
1: function Collection(G')
2:     C ← Closure({[ S' → ·S ]})
3:     C_t ← ∅
4:     while|C_t| ≠ |C| do
5:         C_t ← C
6:         for each I_s ∈ C do
7:             for each grammar symbol X do
8:                 if Goto(I_s, X) ≠ ∅ then
9:                     C ← Goto(I_s, X) ∪ C
10:    return C
```

Using the Closure and Goto functions, we can compute the canonical collection of LR(0) items for an augmented grammar G' (see algorithm 3).

2.2.4 LALR Parser

The LALR (*lookahead*-LR) parsers are simplifications of canonical LR parsers that are very popular in practical applications because of their favourable memory requirements. For example, GNU Bison generates a LALR(1) parser by default [54]. The advantage of LALR(1) parsers is that the LALR(1) automaton uses much fewer states than the LR(1) automaton while retaining most of the expressive power of LR(1) parsers. In fact, the

number of LALR(1) item sets is equal to the number of LR(0) item sets for a grammar G. A LALR(1) parser can be formed by constructing a LR(1) parser and "merging" states that have identical LR(0) items, i.e. states with identical kernel sets (but potentially different FOLLOW sets). A context-free grammar that can be parsed with a LALR(1) parser is said to be LALR(1). The LR(1) grammars are a proper superset of LALR(1) grammars. Therefore, a LALR(1) parser cannot parse all deterministic context-free grammars. This is because applying a LALR(1) parser to an otherwise unambiguous LR(1) grammar may introduce REDUCE/REDUCE conflicts. However, most practical syntactic structures in programming languages can be conveniently described using a LALR(1) grammar [53]. Reference is made to Aho et al. [35] for a detailed presentation of efficient algorithms for the construction of LALR(1) parse tables. Instead, we give an example of a context-free grammar that is LR(1), but not LALR(1), and show how merging LR(1) states can cause REDUCE/REDUCE conflicts.

Consider the following grammar:

$$\begin{array}{lcl} S' & \rightarrow & S \\ S & \rightarrow & aAd \mid bBd \mid aBe \mid bAe \\ A & \rightarrow & c \\ B & \rightarrow & c \end{array} \tag{2.23}$$

We compute the initial item set $I_0 = \textsc{Closure}(\{[\, S' \rightarrow \cdot S,\ \$\,]\})$. Then, we obtain the item set I_{ac} by applying $\textsc{Goto}(\textsc{Goto}(I_0, a), c)$. Similarly, the item set I_{bc} is obtained by applying $\textsc{Goto}(\textsc{Goto}(I_0, b), c)$. The LR(0) item sets I_{ac} and I_{bc} are identical. Therefore, the item sets are merged by taking the union $I_m = I_{ac} \cup I_{bc}$ of the LR(1) items. In the resulting state I_m the lookahead sets are no longer disjoint, since there is more than one REDUCE item on both e and d. Thus, while the LR(1) grammar is unambiguous (the states remain seperate in the LR(1) automaton), a REDUCE/REDUCE conflict occurs in the LALR(1) parser.

$$\begin{array}{llll} I_0: & S' \rightarrow \cdot S,\ \$ & I_{ac}: \quad A \rightarrow c\,\cdot,\ e & I_{bc}: \quad A \rightarrow c\,\cdot,\ d \qquad I_m: \quad A \rightarrow c\,\cdot,\ e \\ & S \rightarrow \cdot aAd,\ \$ & \qquad\quad B \rightarrow c\,\cdot,\ d & \qquad\quad B \rightarrow c\,\cdot,\ e \qquad\quad A \rightarrow c\,\cdot,\ d \\ & S \rightarrow \cdot bBd,\ \$ & & \qquad\qquad\qquad\quad B \rightarrow c\,\cdot,\ d \\ & S \rightarrow \cdot aBe,\ \$ & & \qquad\qquad\qquad\quad B \rightarrow c\,\cdot,\ e \\ & S \rightarrow \cdot bAe,\ \$ & & \end{array} \tag{2.24}$$

3 Implementation

This chapter is a detailed account of our approach to semantic parsing. First, the objectives of the work and the general strategy are formalized in section 3.1. In section 3.2 we introduce NL2PL, a neural semantic parsing tool that fulfills the requirements formalized in section 3.1. We provide a broad overview of the tool architecture and informally introduce our semantic parser's core components. Next, section 3.3 describes the preprocessing applied to input data before training and inference. Finally, our model's main components and the algorithms used for training and inference are described in detail in section 3.4.

3.1 Problem Statement

Based on the concerns expressed in section 1.1 regarding template-based query generation schemes and the current state of the art in semantic parsing (see section 1.3), the prerequisites and requirements for this thesis work are formalized. We empathize that generating queries from natural language descriptions is an instance of semantic parsing for *code generation*. Therefore, it is assumed that the syntax of target meaning representations (programs written in a particular programming language) can be represented by a context-free grammar (see section 2.2.1).

1) Statistical Model

As supervised learning approaches to semantic parsing have been established as state of the art [12], [55], a supervised learning setting is adopted in this work (see section 2.1.1). A dataset $\mathbb{D}_s = [(\mathbf{x}^{(1)}, \mathbf{y}^{(1)}), ..., (\mathbf{x}^{(\kappa)}, \mathbf{y}^{(\kappa)})]$, sampled from a data-generating distribution P_D, with κ pairs of natural language descriptions $\mathbb{X}_s = (\mathbf{x}^{(1)}, ..., \mathbf{x}^{(\kappa)})$ and corresponding program code snippets $\mathbb{Y}_s = (\mathbf{y}^{(1)}, ..., \mathbf{y}^{(\kappa)})$, is given. The i-th pair in the dataset is given as $\mathbf{d}^{(i)} = (\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$. Let f be defined as a function, with learnable parameters Θ, that models the relationship between inputs and outputs for any sample $\mathbf{d} \in \mathbb{D}_s$.

$$\mathbf{y} = f(\mathbf{x}; \Theta) \tag{3.1}$$

The fundamental goal of this thesis is to define a model f and find a parametrization f^* that approximates the relationship between inputs $\mathbf{x}^{(i)}$ and outputs $\mathbf{y}^{(i)}$ as observed in a dataset $\mathbb{D}_s$ containing natural language descriptions and corresponding program code snippets.

$$\mathbf{y}^{(i)} \approx f^*(\mathbf{x}^{(i)}; \Theta) \quad \text{for} \quad i = 1, ..., \kappa \tag{3.2}$$

It is assumed that f^* generalizes to "unseen" samples $\mathbf{d}$, drawn from P_D, with $\mathbf{d} \notin \mathbb{D}_s$. Furthermore, it is assumed that any input $\mathbf{x} = (x_1, ..., x_\tau)$ is a τ-tuple of arbitrary length τ and any output $\mathbf{y} = (y_1, ..., y_\eta)$ is an η-tuple of arbitrary length η. Each input tuple element x shall be part of some finite set of elements V_{in} and each output tuple element shall be part of some finite set of elements V_{out}. Concretely, the elements of V_{in} are assumed to be the words of some natural language, such as English. An input τ-tuple $\mathbf{x}$ is a sequence of words, i.e., a sentence of the English language. The elements of V_{out} are the atomic parts of some programming language string, for example, it's tokens or lexemes (see section 2.2).

2) Code Generation

Enforcing grammatical constraints has been found to be beneficial in the context of code generation [9], [32]. Therefore, as an additional requirement to the statistical model f, every output η-tuple $\mathbf{y}$ must be derivable from a context-free grammar G, which is provided to the model as a priori knowledge. We formalize this constraint by explicitly requiring the image of f to be exactly $L(G)$, the language generated by the grammar.

$$\begin{aligned} &f : \mathbb{X} \to \mathbb{Y} \\ &Im(f) = L(G) \subseteq \mathbb{Y} \end{aligned} \tag{3.3}$$

Intuitively, this means we require the model f to output η-tuples that are syntactically valid under a grammar G.

3.2 NL2PL

We present NL2PL (*natural-language-to-programming-language*), an experimental semantic parsing framework that uses a encoder-decoder network and a LALR(1) parser (see section 2.2.4) to generate program code from natural language descriptions. NL2PL meets the requirements set in section 3.1, as it generates a semantic parser from a dataset of natural language descriptions and corresponding target programs. A LALR(1) grammar describing the syntax of the target programs ensures the generation of syntactically valid programs. This assumes that the syntax of the target language can be described by a LALR(1) grammar. Table 3.1 summarizes the the input and output data of NL2PL.

The configurable model hypotheses featured in NL2PL are variants of the standard encoder-decoder model presented in section 2.1.5. See Appendix A for a list of configuration options. A LALR(1) parser generated from an input LALR(1) grammar G ensures the decoder generates a syntactically valid token sequence during inference. At each decoding step, we derive the set of possible terminal symbols from the current parser state (item set). All tokens in the output vocabulary V_{out} that is an instance of an expected

terminal symbol may be generated by the decoder. The decoder produces a probability distribution over the output vocabulary V_{out}. Finally, we choose the token with the highest probability that is also in the set of expected tokens. These are the two principal components of the proposed semantic parser. Section 3.4 gives a detailed description of the used algorithms for training and inference.

NL2PL provides several tools to facilitate the processing of input and output data, the generation and training of a semantic parser, and its use. The functionality of the tool is divided into three main scripts:

— `preprocess.py`: Takes a dataset $\mathbb{D}_s$ of pairs of natural language descriptions and corresponding programs and a LALR(1) grammar. Generates the input and output vocabularies of the parser model and applies all preprocessing steps to the dataset before training (see section 3.3).
— `train.py`: Used for training a neural encoder-decoder network that produces a probability distribution over the output vocabulary. Takes the preprocessed dataset and a model hypothesis, specifying the embedding dimensions (section 2.1.3), the number of hidden units (section 2.1), layers, etc. outputs a parser model that can be used for inference.
— `translate.py`: Takes a trained encoder-decoder model. Runs the model as a translation service that accepts natural language descriptions and returns the predicted, syntactically valid program. Includes evaluation routines for evaluating a model on the test splits of datasets (chapter 4).

Input
— A LALR(1) grammar G. — Dataset $\mathbb{D}_s$, drawn from a data-generating distribution P_D, with pairs of natural language expressions $\mathbf{x}$ and logical forms $\mathbf{y}$, where $\mathbf{y} \in L(G)$. — A model hypothesis $f(\mathbf{x}; \Theta)$.
Output
— Input vocabulary V_{in} of natural language input tokens. — Output vocabulary V_{out} of programming language output tokens. — A statistical model f^* that approximates the data generating distribution P_D.

Table 3.1: Inputs and outputs of NL2PL.

NL2PL is implemented in Python. It depends on the parser generator Lark [56] and the machine learning framework PyTorch [57].

Lark

Lark is parser generator that implements both the LALR(1) (see section 2.2.4) and SPPF-Earley [58] parsing algorithms. Using Lark, parse trees are automatically recovered during parsing with a LALR(1) parser. We use Lark, because it is well documented,

open-source, and natively implemented in Python. This greatly facilitates making the necessary modifications to the LALR(1) parsing routines.

PyTorch

PyTorch is an open-source machine learning library based on Torch [59]. It implements high-level tensor computations supported by hardware acceleration and numerous auxiliary functions. Moreover, PyTorch implements an automatic differentation engine. Tensor computations are recorded in a computational graph that is used to compute the gradients of the objective function (see section 2.1.1) automatically. We use PyTorch because it has an accessible API that encourages experimentation and has been proven to be a reliable, production-ready machine learning framework.

As indicated in table 3.1, the tool requires a Lark grammar and a dataset of input-output pairs to generate a semantic parser, where the outputs are sentences in the language specified by the grammar. Lark grammars are based on a syntax that is very close to standard EBNF notation. The reader may refer to the Lark grammar reference [60] for detailed documentation on the grammar notation for Lark grammars.

3.3 Preprocessing

Before training a model, the input grammar and dataset are preprocessed in various ways. Preprocessing is done to accelerate training and simplify inference. The preprocessing stage yields the input and output vocabularies for the encoder-decoder model to be trained verifies that the target examples in the dataset can be derived from the provided grammar, and extracts a set of properties used during training from each dataset example. Preprocessing is performed by the `preprocess.py` script. A dataset $\mathbb{D}_s$ is expected to consist of parallel text files, where one text file contains one input example per line, and the other text file contains the target program code string in the corresponding line. Distinct training, validation, and test splits of datasets may be provided (see section 4.2). All output examples in all datasets must be derivable from a single *LALR*(1) grammar specification provided as an argument to the preprocessing script. If a dataset is split into training, validation, and test data, the script will yield the following files:

— `<user_defined_name>.train.pt`: Preprocessed dataset split for training.
— `<user_defined_name>.dev.pt`: Preprocessed dataset split used for validating and optimizing hyperparameter configurations during training.
— `<user_defined_name>.test.pt`: Preprocessed dataset split used for testing model performance after training.
— `<user_defined_name>.lang.pt`: File containing the grammar and vocabularies.

In the following, we describe how input and output vocabularies are derived from the input datasets and all preprocessing steps applied to the datasets.

Input and Output Vocabularies

Input and output vocabularies are exclusively derived from an input dataset $\mathbb{D}_s = (\mathbb{X}_s, \mathbb{Y}_s)$. The input vocabulary is derived from the input examples $\mathbb{X}_s$. Each input item in the dataset is assumed to be a natural language string. First, each such input string is normalized by setting all uppercase words in the text in lowercase and padding punctuation marks with spaces. Moreover, special meta-symbols are added to the input text (see table 3.4). The set of whitespace delimited substrings in each input text in the dataset then constitues input vocabulary V_{in} = [(0, '<PAD>'), (1, '<SOS>'), ..., (τ − 1, 'zebra'), (τ, 'zoo')]. Each item in the vocabulary is uniquely paired with an integer, such that the input string is represented by a sequence of integers.

Original String	"What is the capital of Germany?"
Normalized String	"<SOS> what is the capital of germany ? <EOS>"
Integer Sequence	[1, 789, 125, 46, 4568, 684, 8954, 387, 2]

Table 3.2: Ficticious example of preprocessing of natural language input strings.

The output vocabulary is derived from the target source code examples $\mathbb{Y}_s$. Each output example is assumed to be a sentence of the language $L(G)$ generated by the input LALR(1) grammar G. Using the parser generator Lark we generate a lexical analyzer and LALR(1) parser from the grammar G. The source code is then tokenized using the lexical analyzer, such that a token sequence s is obtained that represents the source code. The parser verifies that $s \in L(G)$. If not, the input-output pair is discarded from the dataset and not considered. The set of tokens obtained in this way for all output examples forms the output vocabulary V_{out}. Like in the input vocabulary, each token in the output vocabulary is uniquely paired with and represented as an integer. Additionally, the special meta-symbols in table 3.4 are included in the vocabulary.

Field	Description
source	The original natural language source string.
target	The original target program code string.
source_idx	The list of integer indices for each word in the normalized natural language string, according to the input vocabulary V_{in}.
target_idx	The list of "compressed" integer indices for each token in the tokenized program code, according to the output vocabulary V_{out}.
source_len	The length of source_idx.
target_len	The length of target_idx.

Table 3.3: Fields generated per example during preprocessing.

Data Preprocessing

After the input and output vocabularies have been created, all dataset splits are preprocessed in terms of those vocabularies. Each input-output pair is inserted into a data

structure with multiple fields. This data structure represents a dataset example and is stored in the respective dataset split file . The data structure holds the fields denoted in table 3.3.

Meta Symbol	Description
<SOS>	*Start-of-sequence* symbol. Each input and output sequence begins with this symbol and initiates the encoder and decoder algorithms.
<EOS>	*End-of-sequence* symbol. Indicates to the encoder algorithm that the whole input sequence has been consumed. The decoder algorithm implicitly stops generating tokens when the output has been reduced to the start rule of the grammar.
<PAD>	*Padding* symbol. Neural network training can be accelerated substantially by *batching* multiple examples in a single matrix (see section 2.1.1). However, natural language texts are expected to be of varying length. In order to have equal length texts within a batch, each input sequence shorter than the maximal length input item is padded with the padding symbol.
<UNK>	*Unknown* symbol. During inference, words may occur in natural language strings that did not appear in the dataset used for deriving the vocabularies. Such words are replaced by this special symbol.

Table 3.4: Meta symbols in vocabularies.

3.4 Model

The two main variants of models used for generating a semantic parser are encoder-decoder models (section 2.1.5) and attention-based encoder-decoder models (section 2.1.6). During decoding, encoder-decoder models are assisted by a LALR(1) parser that ensures valid tokens are generated at each decoding step. The full decoder algorithm is described in section 3.4.1. The training algorithm learns a language model as described in section 2.1.1 and 2.1.2. We describe the training algorithm in section 3.4.2.

3.4.1 Inference

Inference on natural language descriptions of programs can be performed using the `translate.py` script. See Appendix A for a list of arguments. See table 3.5 for a description of inputs and outpus of the `translate.py` script.

First, a LALR(1) parser is generated from the provided grammar G. Natural language strings are preprocessed as described in section 3.3 before inference. An integer sequence $\mathbf{x}$, where each integer represents an element in V_{in} is fed into the encoder RNN as described in section 2.1.5. Conditioned on the encoder's output context vector, the decoder RNN generates a sequence of integers, where each integer is an element of V_{out}. Using the LALR(1) parser, before each decoding step, a set of viable tokens $E \subseteq V_{out}$ is determined. The decoder generates a probability distribution over all output vocabulary tokens in V_{out}. The most likely token $e \in E$ is generated (see algorithm 4).

Input
— Input vocabulary V_{in} of natural language input tokens. — Output vocabulary V_{out} of programming language output tokens. — A LALR(1) grammar G. — A trained statistical model f^*, with $f^* : \mathbb{X} \rightarrow \mathbb{Y}$. — An input natural language string.
Output
— A token sequence $\mathbf{y} = (y_1, ..., y_\eta)$, with $\mathbf{y} \in L(G)$ and each sequence item $y_i \in V_{out}$.

Table 3.5: Inputs and outputs of `translate.py`.

The set of viable tokens can easily be determined by computing Closure over the kernel items I_k corresponding to the current parser state. For each item $A \rightarrow \alpha \cdot B\beta$ in Closure(I_k) we add the grammar symbol B to a set E_{NT}. Then, we compute the set of expected terminal symbols E_T by taking the the intersection $E_{NT} \cap T$. The set of expected tokens E_t then corresponds to all tokens in the output vocabulary V_{out} with type $t \in E_T$ (see algorithm 5).

Algorithm 4 : LALR(1) Assisted Decoding Algorithm

$a \leftarrow$ Beginning of sequence token t = `Token(SOS, '<SOS>')`.
$s \leftarrow$ State on top of state stack S.
$H \leftarrow$ Encode($[x_1, x_2, ..., x_{\tau-1}, x_\tau]$)
$I_s \leftarrow$ Closure($\{[S' \rightarrow \cdot S, \$]\}$)
$E_t \leftarrow$ Expected(I_s)
$H, a \leftarrow$ Decode(H, E_t, a)
while Action(s, a) $\neq$ `accept` **do**
  if Action(s, a) = `shift` t **then**
    Push state t onto stack S.
    $I_s \leftarrow$ Goto(I_s, a)
    $E_t \leftarrow$ Expected(I_s)
    $H, a \leftarrow$ Decode(H, E_t, a)
  else if Action(s, a) = `reduce` $[A \rightarrow \beta]$ **then**
    Pop $|\beta|$ symbols off the stack S.
  $s \leftarrow$ State on top of state stack S.

The parser invokes the decoder module only when$|E_t| > 1$, i.e., when the parser has a decision to make between two or more tokens to parse in the next step. If$|E_t| = 1$, with a single expected token t, Goto(I_k, t) is computed and the resulting state is shifted onto the stack without invoking the decoder module. If$|E_t| > 1$, the algorithm invokes the decoder to determine the most likely token in the set t. Again, Goto(I_k, t) is computed to yield the next parser state.

Algorithm 5 : Function EXPECTED for computing set of expected tokens.

```
I_s ← LALR(1) item set for current state s.
T ← Terminal symbols T of grammar G.
V_t ← Token vocabulary.
function EXPECTED(I_s)
    E_t ← ∅
    for [A → α · Bβ] ∈ I_s do
        if B ∈ T then
            B_t ← All tokens in V_t of type B.
            E_t ← B_t ∪ E_t
    return E_t
```

The algorithm does not require an explicit end of sequence symbol (see table 3.4). The algorithm terminates when the parser recognizes a reduction by the start rule $\langle S' \rangle \rightarrow \langle S \rangle$ of the augmented grammar G' (see section 2.2.3).

3.4.2 Training

A semantic parser can be generated using the `train.py` script. See Appendix A for a list of arguments. See table 3.6 for a description of inputs and outpus of of the `train.py` script.

INPUT
— Input vocabulary V_{in} of natural language input tokens. — Output vocabulary V_{out} of programming language output tokens. — Dataset $\mathbb{D}_s$, drawn from a data-generating distribution P_D, with pairs of natural language expressions $\mathbf{x}$ and logical forms $\mathbf{y}$, where $\mathbf{y} \in L(G)$. — Configuration data specifying the hyperparameters of the model to be trained.
OUTPUT
— A statistical model f^* that approximates the data generating distribution P_D.

Table 3.6: Inputs and outputs of `train.py`.

The training algorithm learns a language model as described in section 2.1.1 and 2.1.2. A LALR(1) parser, as described in section 3.4.1, is not directly used to ensure syntactically valid programs. If the algorithm 4 would be used during training, each training example had to be processed individually. This would preclude the batching of multiple training examples and make training computationally very expensive. Thus, during training, the decoder may generate outputs $\hat{\mathbf{y}} \notin L(G)$.

However, as described in section 3.4.1, the decoder is only invoked when the set of expected tokens contains more than one token, causing a mismatch between the token sequences learned by the model during training and the sequences generated during inference. Effectively, the decoder has to predict the complete ground truth target sequence $\mathbf{y}$, whereas during inference, the decoder only predicts tokens when the set of viable

tokens is ambiguous, i.e., greater than one.

To account for this mismatch, ground truth target sequences $\mathbf{y}$ are preprocessed in a particular way. During the preprocessing stage, a LALR(1) parser is generated from a provided grammar G. All ground truth target programs in an input dataset are parsed using the generated parser. During each parsing step, the set of viable tokens in the output vocabulary V_{out} is determined using algorithm 5. If the set of viable tokens is greater than one, the index of this token is appended to `target_idx` (see table 3.3), otherwise not. This way, the decoder does not have to learn the tokens during training that would be automatically generated by the parser during inference.

4 Evaluation

In the following, the semantic parser proposed in chapter 3 shall be evaluated. Since a supervised learning algorithm was chosen, datasets of pairs of natural language expressions and corresponding logical forms are necessary for training and evaluating the model. Available datasets for semantic parsing are examined and evaluated for their suitability in section 4.1. In section 4.2 the evaluation methododology and metrics used are documented. Finally, the evaluation results are presented in section 4.3 and analyzed in section 4.4.

4.1 Datasets

The aim of evaluating supervised deep learning models is to measure how well they *generalize* to unseen examples from an unknown, but true distribution. Neural models are particularly "data-hungry"; a key challenge is to obtain sufficiently large datasets of input-output pairs from that distribution. Datasets designed explicitly for semantic parsing are usually either written manually, and relatively small [23], or they are generated automatically and have low complexity regarding their logical forms. [61]. Most datasets are limited to a particular domain, such as restaurant booking or travel information. They are annotated with well-defined meaning representations in languages such as Prolog or SQL. Table 4.2 gives an overview of datasets commonly used as benchmarks for semantic parsing. In order to assess how well machine learning algorithms perform, datasets are commonly divided into disjoint training and test sets (see section 4.2). A model is trained exclusively on training set examples and, after training, evaluated on test set examples to probe its generalizability.

Example Utterance	"What cities are located in California?"

Example Query	`SELECT CITYalias0.CITY_NAME`
	`FROM CITY AS CITYalias0`
	`WHERE CITYalias0.STATE_NAME = "california" ;`

SQL Pattern	`SELECT <table_alias>.<field>`
	`FROM <table> AS <table_alias>`
	`WHERE <table_alias>.<field> = "<literal>" ;`

Table 4.1: A query example using the standardization scheme as proposed by Dollak et al. [55] In the corresponding SQL pattern, specific table names, field names and values are substituted with generic placeholders.

Finegan-Dollak et al. [55] have conducted a study on the statistics of various corpora

relevant for semantic parsing. Based on several redundancy and complexity measures, the datasets listed in table 4.3 were evaluated with regard to their characteristics. They identified various problems with the current evaluation methodology and made valuable improvements to the datasets. Most notably, these improvements include the canonicalization of logical forms and variable annotation.

The canonicalization of the logical forms is motivated by improving comparability between datasets. Logical forms across datasets are translated into SQL queries that use a consistent writing style (see table 4.1). Fields in `SELECT` clauses, tables in `FROM` clauses and constraints in `WHERE` clauses are ordered alphabetically. Table aliases follow the pattern `<TABLE_NAME>alias<N>`, where `N` indicates the `N`-th use of a table `<TABLE_NAME>`. Capitalization and spacing between symbols are standardized.

Dataset	Description
GeoQuery (Zelle and Mooney, 1996 [23])	Pairs of nautral language questions about the geography of the US. Annotated with Prolog queries.
ATIS (Price, 1990 [62])	Natural language questions regarding airline travel information. Datasets available annotated with SQL queries and Prolog statements.
Advising (Finegan-Dollak et al., 2018 [55])	Dataset of questions regarding course information at the University of Michigan. Annotated with SQL queries.
Scholar (Iyer et al., 2017 [63])	Natural language questions about academic publications. SQL queries were automatically generated and validated by checking if they yield the correct output.
Restaurants (Popescu et al., 2003 [64])	Dataset with questions about restaurant booking. Annotated with SQL queries.
Academic (Li and Jagadish, 2014 [65])	Questions on the Microsoft Academic Search database. Same domain as Scholar, but different database schema. Annotated with SQL queries.
Yelp (Yaghmazadeh et al., 2017 [66])	Dataset pairing natural language questions about business ratings and user reviews with SQL queries.
IMDB (Yaghmazadeh et al., 2017 [66])	Natural language questions about the Internet Movie Database paired with SQL queries.
WikiSQL (Zhong et al., 2018 [61])	Automatically generated pairs of questions and corresponding SQL queries on individual Wikipedia tables. Automatically generated questions questions are replaced by crowd-sourced paraphrases.
CoNaLa (Yin et al., 2018 [67])	Curated pairs of natural language descriptions and corresponding code snippets in Java and Python crawled from Stack Overflow.
IFTTT (Quirk et al. 2015 [68])	Dataset of natural language expressions paired with simple "If-This-Then-That" statements that can be mapped to executable code.

Table 4.2: Common datasets used as benchmarks for evaluating semantic parsers.

Furthermore, Finegan-Dollak et al. provide the means for accurate entity anonymization in logical forms through explicit variable annotation. It is common for literals in logical forms to be mentioned in the corresponding natural language utterance. For ex-

ample, in table 4.1, the entity "California" in the example utterance reappears as literal value in a `WHERE` constraint of the example program. In the datasets provided by Finegan-Dollak et al., literals are replaced by typed variables that refer to entities in the input utterance. The literal values are recorded so that the original dataset can be restored. Entity anonymization abstracts from specific values in logical forms and can give a clearer picture of how well a model can predict not the content but the *structure* of a program.

Statistics

Finegan-Dollak et al. [55] collected statistics on various datasets, listed in Table 4.1, and gathered various redundancy and complexity measures from which characteristic properties can be derived. Most notably, dataset size does not necessarily imply greater dataset variability. For example, by raw sample count ("question count"), WikiSql is by an order of magnitude larger than ATIS. However, WikiSQL is a dataset with high redundancy. In fact, more than half of all logical forms (42,816) follow a simple `SELECT-FROM-WHERE` pattern, with a single colum reference and constraint (see Table 4.4). Other queries are similarly simply structured, mostly adding just one more constraint or using an aggregation function, such as `SUM(*)` or `COUNT(*)`, in the `SELECT` clause. In contrast, ATIS and GeoQuery feature comparatively complex queries. For instance, nested queries are prevalent in both datasets. ATIS and GeoQuery have an average nesting depth of 1.39 and 2.03, respectively, with a maximal nesting depth of 8 and 7. Also, queries from both datasets typically refer to multiple tables and table columns, use `JOINS` and other SQL language features, and use multiple constraints routinely. The sample queries in table 4.4 give an approximate impression of their relative complexity. ATIS and GeoQuery also have much more favorable redundancy measures. Even though it only has 5280 samples, compared to 80654 samples in WikiSQL, ATIS features a greater variety of query patterns (751) compared to WikiSQL (488).

	Redundancy Measures						Complexity Measures							
		Unique		Queries / pattern			Tables / query		Unique tables / query		SELECTs / query		Nesting Depth	
	Question count	query count	[1]/[2]	μ	Max	Pattern count	μ	Max	μ	Max	μ	Max	μ	Max
Advising	4570	211	21.7	20.3	90	174	3.2	9	3.0	9	1.23	6	1.18	4
ATIS	5280	947	5.6	7.0	870	751	6.4	32	3.8	12	1.79	8	1.39	8
GeoQuery	877	246	3.6	8.9	327	98	1.4	5	1.1	4	1.77	8	2.03	7
Restaurants	378	23	16.4	22.2	81	17	2.6	5	2.3	4	1.17	2	1.17	2
Scholar	817	193	4.2	5.6	71	146	3.3	6	3.2	6	1.02	2	1.02	2
Academic	196	185	1.1	2.1	12	92	3.2	10	3	6	1.04	3	1.04	2
IMDB	131	89	1.5	2.5	21	52	1.9	5	1.9	5	1.01	2	1.01	2
Yelp	128	110	1.2	1.4	11	89	2.2	4	2	4	1	1	1	1
WikiSQL	80,654	77,840	1.0	165.3	42,816	488	1	1	1	1	1	1	1	1

Table 4.3: Statistics of the standardized datasets by Finegan-Dollak et al. [55], based on logical forms written in SQL. [1] is "Question count" and [2] is "Unique query". Datasets above the first dashed horizontal line are hand-made from the NLP community, below are hand-made datasets from the DB community. Datasets below the second dashed horizontal line are automatically generated.

Learning a semantic parser for code generation requires great variability and complexity in the target meaning representations. Moreover, a supervised learning approach, as taken in this work, requires big dataset sizes. ATIS, GeoQuery and Advising provide

the best balance between dataset size and query complexity. ADVISING and ATIS are similar in size, but ATIS is slightly more complex. Moreover, ATIS and GEOQUERY are more widely used as benchmarks for semantic parsers throughout the literature. Therefore, we evaluate the proposed model on ATIS and GEOQUERY.

Dataset	Example

WIKISQL	`SELECT TABLEalias0.POSITION_FIELD`
	`FROM TABLE AS TABLEalias0`
	`WHERE TABLEalias0.CLUB_TEAM_FIELD = "Baylor" ;`

GEOQUERY	`SELECT HIGHLOWalias0.LOWEST_POINT`
	`FROM HIGHLOW AS HIGHLOWalias0`
	`WHERE HIGHLOWalias0.STATE_NAME IN`
	`( SELECT RIVERalias0.TRAVERSE`
	`FROM RIVER AS RIVERalias0`
	`WHERE RIVERalias0.RIVER_NAME = "mississippi" )`
	`ORDER BY HIGHLOWalias0.LOWEST_ELEVATION LIMIT 1 ;`

ATIS	`SELECT DISTINCT FLIGHTalias0.FLIGHT_ID`
	`FROM AIRPORT_SERVICE AS AIRPORT_SERVICEalias0 ,`
	`AIRPORT_SERVICE AS AIRPORT_SERVICEalias1 ,`
	`CITY AS CITYalias0 , CITY AS CITYalias1 ,`
	`FLIGHT AS FLIGHTalias0`
	`WHERE CITYalias0.CITY_CODE = AIRPORT_SERVICEalias0.CITY_CODE`
	`AND CITYalias0.CITY_NAME = "ATLANTA"`
	`AND CITYalias1.CITY_CODE = AIRPORT_SERVICEalias1.CITY_CODE`
	`AND CITYalias1.CITY_NAME = "BALTIMORE"`
	`AND FLIGHTalias0.FROM = AIRPORT_SERVICEalias0.AIRPORT_CODE`
	`AND FLIGHTalias0.TO = AIRPORT_SERVICEalias1.AIRPORT_CODE ;`

Table 4.4: Example queries taken from WIKISQL, GEOQUERY and ATIS demonstrating the relative complexity of the queries.

4.2 Setup and Metrics

We use the same dataset splits as Finegan-Dollak et al. [55] for evaluating the proposed model (see table 4.5). A basic encoder-decoder model and an attentional encoder-decoder model, as reported by Finegan-Dollak et al., serves as the baseline for comparison. Moreover, we compare our model to the SEQ2TREE model proposed by Dong and Lapata [12]. We used the validation sets for optimizing hyperparameters (such as hidden or embedding dimensions) and evaluated accuracy once on the test splits of the datasets.

	Training	Validation	Test
GEOQUERY	552	49	279
ATIS	4344	486	442

Table 4.5: Dataset splits used for GEOQUERY and ATIS.

The main metrics used are classification accuracy *CA* and *exact match* classification accuracy *EM*. Exact match classification accuracy measures the percentage of predicted test set examples that match the target ground truth program, i.e., the predicted token sequences identical to the token sequences in the test set examples (equation 4.1). Classi-

fication accuracy measures the total number of individual tokens at each decoding step that were correctly predicted (equation 4.2). The final output of the training algorithm is the model with the highest validation set exact match accuracy during training. The model is tested using parser assistance as described in section 3.4.1.

$$EM = \frac{\text{Correctly predicted test set examples}}{\text{Total number of test set examples}} \tag{4.1}$$

$$CA = \frac{\text{Correctly predicted test set tokens}}{\text{Total number of test set tokens}} \tag{4.2}$$

Moreover, training and validation classification and exact match classification accuracy are measured to quantify the generalizability of the generated semantic parser. High training and validation accuracy combined with low test accuracy are indicative of *overfitting*, whereas low training and validation accuracy combined with low test accuracy are indicative of the model *underfitting* the data [69].

For our trials we use the canonicalized GeoQuery and ATIS datasets provided by Finegan-Dollak et al. [55] We run trials without anonymized entities and with anonymized entities (see section 4.1 and table 4.1). We refer to trials with the standard dataset, i.e., the trials without anonymized entities, as *standard* trials. Trials with anonymized entities, where the algorithm is tasked with predicting only the token type and the correct token value is inserted by an "all-knowing" oracle, are referred to as *oracle* trials.

4.3 Results

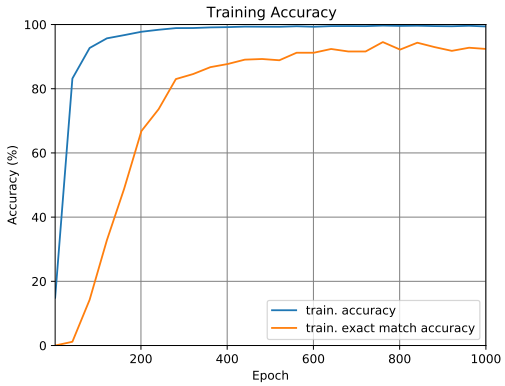
In table 4.6 we present the results of the evaluation. We evaluated both attentional encoder-decoder models with parser assistance and encoder-decoder models without attention. The full model configuration for each trial are given in Appendix B.

	GeoQuery		**ATIS**	
	Standard	**Oracle**	**Standard**	**Oracle**
Ours	34%	63%	9%	48%
+ Attention	51%	69%	33%	55%
Finegan-Dollak et al.	27%	49%	8%	14%
+ Attention	63%	73%	46%	57%
Dong & Lapata	62%	68%	46%	56%

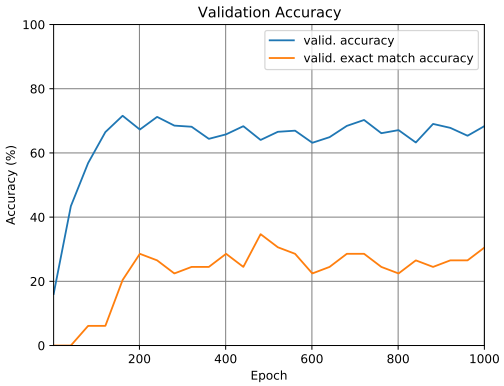
Table 4.6: Exact match accuracy on dataset test splits for GeoQuery and ATIS.

Table 4.6 shows the results for exact match accuracy. The baseline model is the encoder-decoder model (as described in section 2.1.5) by Finegan-Dollak et al. [55] without attention. We also run a trial with attention for both models. Additionally, we compare our model with the attentional Seq2Tree model by Dong and Lapata [12].

GeoQuery Standard, without Attention



(a) Training accuracy (blue) and training exact match accuracy (orange) in percent over 1000 epochs.

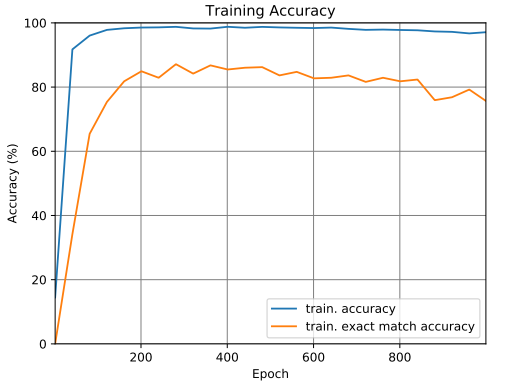


(b) Validation accuracy (blue) and validation exact match accuracy (orange) in percent over 1000 epochs.

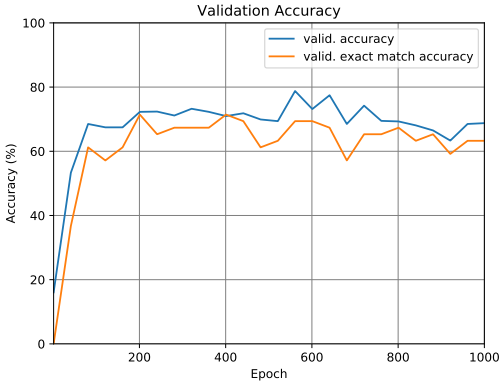
Figure 4.1: Training results for trial on the GeoQuery dataset with standard entities and without attention.

Figure 4.1a shows the training accuracy and training exact match accuracy. Training accuracy at epoch 1000 is 99.561%. Training exact match accuracy at epoch 1000 is 92.188%. Figure 4.1b shows the validation accuracy and validation exact match accuracy. Validation accuracy at epoch 1000 is 67.632%. Validation exact match accuracy at epoch 1000 is 30.612%. Best validation exact match accuracy was achieved in epoch 1173 with 38.776%. Test accuracy is 66.445% and test exact match accuracy is 34.050% (see table 4.6).

GeoQuery Oracle, without Attention



(a) Training accuracy (blue) and training exact match accuracy (orange) in percent over 1000 epochs.

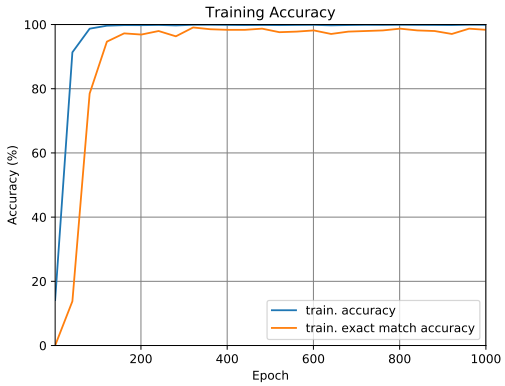


(b) Validation accuracy (blue) and validation exact match accuracy (orange) in percent over 1000 epochs.

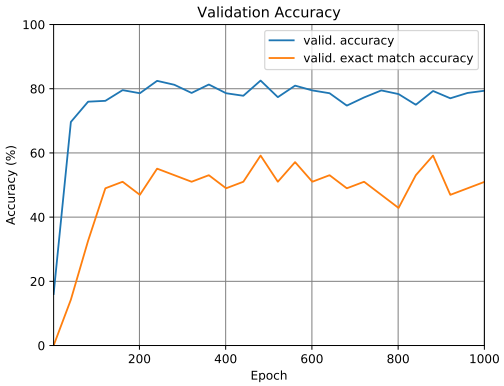
Figure 4.2: Training results for trial on the GeoQuery dataset with oracle entities and without attention.

Figure 4.2a shows the training accuracy and training exact match accuracy. Training accuracy at epoch 1000 is 97.017%. Training exact match accuracy at epoch 1000 is 77.022%. Figure 4.2b shows the validation accuracy and validation exact match accuracy. Validation accuracy at epoch 1000 is 67.368%. Validation exact match accuracy at epoch 1000 is 61.224%. Best validation exact match accuracy was achieved in epoch 296 with 73.469%. Test accuracy is 67.853% and test exact match accuracy is 63.799%.

GeoQuery Standard, with Attention



(a) Training accuracy (blue) and training exact match accuracy (orange) in percent over 1000 epochs.

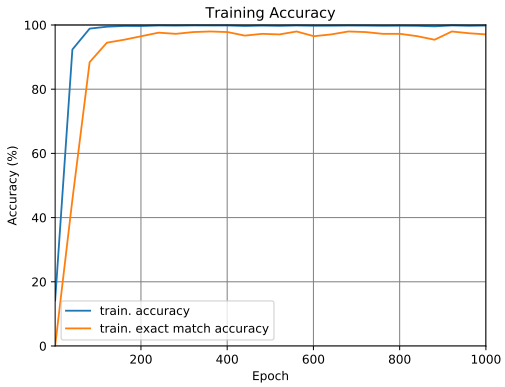


(b) Validation accuracy (blue) and validation exact match accuracy (orange) in percent over 1000 epochs.

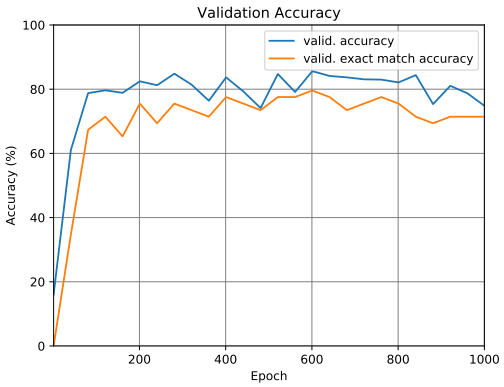
Figure 4.3: Training results for trial on the GeoQuery dataset with standard entities and attention.

Figure 4.3a shows the training accuracy and training exact match accuracy. Training accuracy at epoch 1000 is 99.899%. Training exact match accuracy at epoch 1000 is 98.713%. Figure 4.3b shows the validation accuracy and validation exact match accuracy. Validation accuracy at epoch 1000 is 82.807%. Validation exact match accuracy at epoch 1000 is 57.143%. Best validation exact match accuracy was achieved in epoch 478 with 63.265%. Test accuracy is 74.712% and test exact match accuracy is 51.613%.

GeoQuery Oracle, with Attention



(a) Training accuracy (blue) and training exact match accuracy (orange) in percent over 1000 epochs.

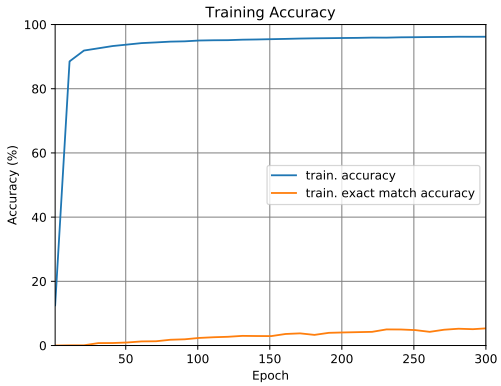


(b) Validation accuracy (blue) and validation exact match accuracy (orange) in percent over 1000 epochs.

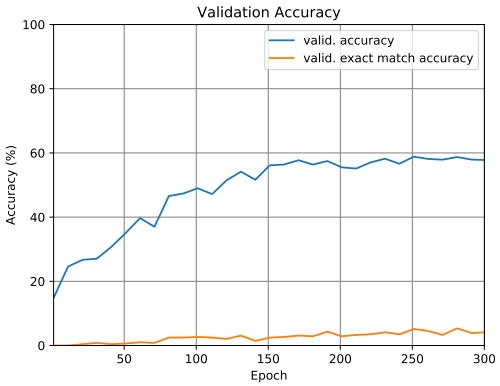
Figure 4.4: Training results for trial on the GeoQuery dataset with oracle entities and attention.

Figure 4.4a shows the training accuracy and training exact match accuracy. Training accuracy at epoch 1000 is 99.807%. Training exact match accuracy at epoch 1000 is 96.324%. Figure 4.4b shows the validation accuracy and validation exact match accuracy. Validation accuracy at epoch 1000 is 71.842%. Validation exact match accuracy at epoch 1000 is 73.469%. Best validation exact match accuracy was achieved in epoch 489 with 81.633%. Test accuracy is 74.939% and test exact match accuracy is 69.176%.

ATIS Standard, without Attention



(a) Training accuracy (blue) and training exact match accuracy (orange) in percent over 300 epochs.

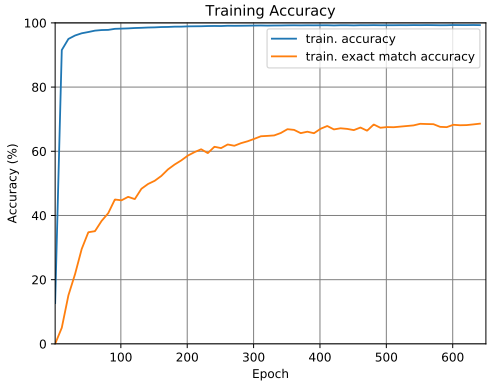


(b) Validation accuracy (blue) and validation exact match accuracy (orange) in percent over 300 epochs.

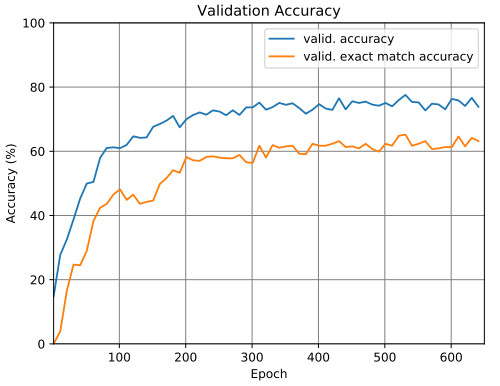
Figure 4.5: Training results for trial on the ATIS dataset with standard entities and without attention. Due to an error during logging only data up to epoch 300 is available.

Figure 4.5a shows the training accuracy and training exact match accuracy. Training accuracy at epoch 300 is 96.220%. Training exact match accuracy at epoch 300 is 5.682%. Figure 4.5b shows the validation accuracy and validation exact match accuracy. Validation accuracy at epoch 300 is 57.954%. Validation exact match accuracy at epoch 300 is 3.292%. Best validation exact match accuracy was achieved in epoch 983. Test accuracy is 38.729% and test exact match accuracy is 9.050%.

ATIS Oracle, without Attention



(a) Training accuracy (blue) and training exact match accuracy (orange) in percent over 650 epochs.

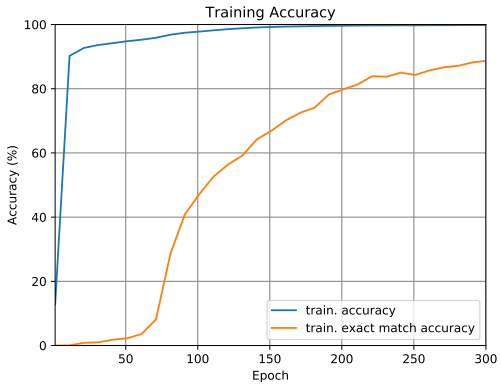


(b) Validation accuracy (blue) and validation exact match accuracy (orange) in percent over 650 epochs.

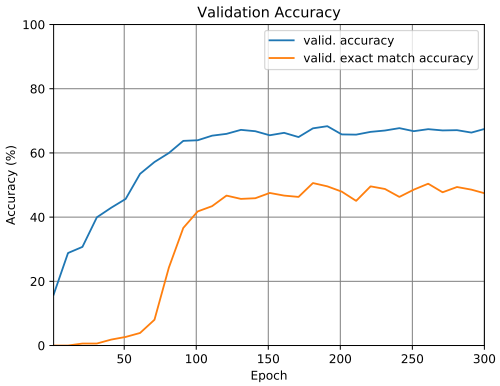
Figure 4.6: Training results for trial on the ATIS dataset with oracle entities and without attention.

Figure 4.6a shows the training accuracy and training exact match accuracy. Training accuracy at epoch 650 is 99.292%. Training exact match accuracy at epoch 650 is 68.063%. Figure 4.6b shows the validation accuracy and validation exact match accuracy. Validation accuracy at epoch 650 is 74.822%. Validation exact match accuracy at epoch 650 is 64.609%. Best validation exact match accuracy was achieved in epoch 628 with 65.432%. Test accuracy is 47.130% and test exact match accuracy is 48.993%.

ATIS Standard, with Attention



(a) Training accuracy (blue) and training exact match accuracy (orange) in percent over 300 epochs.

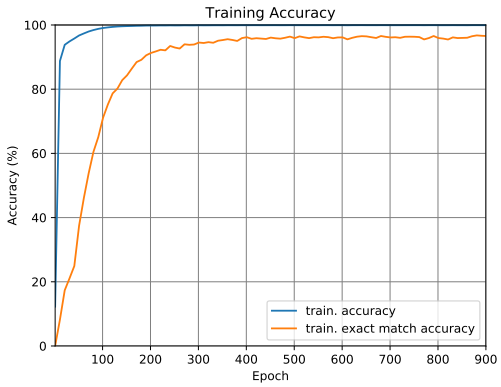


(b) Validation accuracy (blue) and validation exact match accuracy (orange) in percent over 300 epochs.

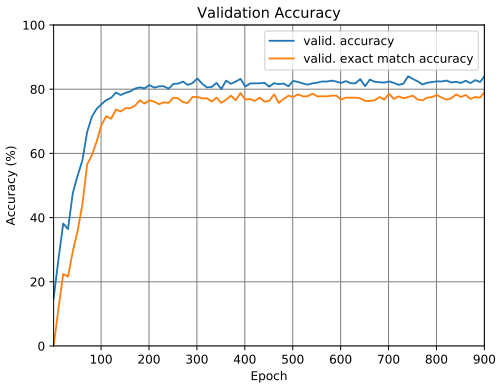
Figure 4.7: Training results for trial on the ATIS dataset with standard entities and attention.

Figure 4.7a shows the training accuracy and training exact match accuracy. Training accuracy at epoch 300 is 99.812%. Training exact match accuracy at epoch 300 is 88.744%. Figure 4.7b shows the validation accuracy and validation exact match accuracy. Validation accuracy at epoch 300 is 66.207%. Validation exact match accuracy at epoch 300 is 48.765%. Best validation exact match accuracy was achieved in epoch 181 with 50.617%. Test accuracy is 40.930% and test exact match accuracy is 33.032%.

ATIS Oracle, with Attention



(a) Training accuracy (blue) and training exact match accuracy (orange) in percent over 900 epochs.



(b) Validation accuracy (blue) and validation exact match accuracy (orange) in percent over 900 epochs.

Figure 4.8: Training results for trial on the ATIS dataset with oracle entities and attention.

Figure 4.8a shows the training accuracy and training exact match accuracy. Training accuracy at epoch 900 is 99.929%. Training exact match accuracy at epoch 900 is 96.236%. Figure 4.8b shows the validation accuracy and validation exact match accuracy. Validation accuracy at epoch 900 is 83.095%. Validation exact match accuracy at epoch 900 is 78.189%. Best validation exact match accuracy was achieved in epoch 395 with 79.424%. Test accuracy is 47.159% and test exact match accuracy is 55.257%.

4.4 Analysis

The results in table 4.6 show that the proposed approach to semantic parsing is competitive with comparable models. Most notably, our approach yields significant improvements compared to the regular encoder-decoder baseline model, as reported by Finegan-Dollak et al. [55] when entities are anonymized (oracle condition) and no attention mechanism is applied. Here, our encoder-decoder model with parser assistance yields an exact match accuracy of 63% on the GeoQuery dataset and 48% on the ATIS dataset. In contrast, the baseline encoder-decoder model by Finegan-Dollak et al. achieves 49% on the GeoQuery dataset and only 14% on the ATIS dataset. Thus, our model achieves a relative improvement of 14% and 34%, respectively.

However, on the standard datasets without entity anonymization, the improvements in the trials without attention are insignificant. Our approach yields an exact match accuracy of 34% on the GeoQuery dataset and 9% on the ATIS dataset. The baseline model by Finegan-Dollak et al. achieves 27% on the GeoQuery dataset and 8% on the ATIS dataset. This indicates that the proposed semantic parser's main utility lies in recognizing the *structure* of a program more reliably. However, when the model is tasked with resolving not only entity types, as in the oracle condition, but also concrete entity values, it struggles as much as the baseline encoder-decoder model by Finegan-Dollak et al. This is also substantiated by the validation accuracy plots for the standard trials without entity anonymization in figures 4.1b and 4.5b. While reasonable validation accuracy on both the GeoQuery and ATIS datasets are achieved, they do not translate to comparable exact match accuracy. In contrast, in the validation accuracy plots in figures 4.2b and 4.6b for the trials with entity anonymization the curves for validation accuracy and exact match validation accuracy show a very similar course.

Applying an attention mechanism to both our approach and the baseline model by Finegan-Dollak et al. further relativizes the improvements. In the trial with anonymized entities our approach achieves 69% exact match accuracy on the GeoQuery dataset and 55% on the ATIS dataset. The attentional encoder-decoder model by Finegan-Dollak et al. (73% for GeoQuery and 57% for ATIS) and the Seq2Tree model by Dong and Lapata [12] (68% for GeoQuery and 56% for ATIS) achieve comparable results. Notably, the improvement by employing an attention mechanism is relatively small for our approach compared to the improvement the standard encoder-decoder model by Finegan-Dollak et al. obtains from applying attention. We only gain 6% in the GeoQuery trial with anonymized entities and 7% in the ATIS trial. In contrast, Finegan-Dollak et al. improve by 24% in the GeoQuery trial and by 43% in the ATIS trial.

This indicates that the parser assisted decoder proposed in this work has a similar effect as applying attention to a standard encoder-decoder model. Bahdanau et al. [27] assert that standard encoder-decoder models (see section 2.1.4) degrade in performance with increasing input and output sequence length since the context vector is of fixed size and stores the same amount of information for short and long sequences. Attention mechanisms alleviate this issue by dynamically constructing a separate context vector for each decoding step (see section 2.1.5). As described in section 3.4.1, the parser assisted

decoding algorithm does not invoke the decoder when the set of expected tokens is of size one, effectively "compressing" the output sequence and offloading token generation to the parser. The decoder has to generate much shorter sequences, leading to less performance degradation due to long output sequences.

Finally, our model slightly underperforms on the standard datasets without entity anonymization when using an attention mechanism. On the GeoQuery dataset, our semantic parser only achieves 51% exact match accuracy, while Finegan-Dollak et al. achieve 63%. On the ATIS dataset, we achieve 33%, while Finegan-Dollak et al. achieve 46%. The reasons for this difference could not be identified. One possible reason for this discrepancy is a poor choice of hyperparameters (model configuration). Further trials are necessary in order to optimize model configurations for these particular trials.

5 Conclusion

In this thesis parser assisted approaches to semantic parsing were examined. We hypothesized that a bottom-up shift-reduce parser that enforces syntactically valid logical forms might improve prediction accuracy for semantic parsers based on encoder-decoder models. This assertion was informed by the fact that encoder-decoder models perform better on shorter sequences [27]. Parser models based on shift-reduce parsers help enforce syntactically valid logical forms while allowing outputs to be modeled as sequences, simplifying the architecture and training procedure compared to the SEQ2TREE model proposed by Dong and Lapata [12]. Our approach to semantic parsing can be interpreted as predicting only the leaves of a parse tree. Approaches that take a derivational view during the construction of logical forms [31] or try to predict whole parse trees [12] necessarily must predict longer sequences. Enforcing syntax contraints has also proven to be beneficial (see section 1.3). We have shown that our approach to grammar-guided syntax aware semantic parsing is competitive with comparable approaches. Notably, our semantic parsing model has an advantage over standard encoder-decoder models without attention. It was shown that, especially in trials with anonymized entities, our model outperforms standard encoder-decoder models by a large margin. This indicates that our parser assisted decoder can predict the *structure* of programs much more reliably than standard encoder-decoder models. We conclude that parsing techniques based on shift-reduce parsing are a promising avenue for enhancing encoder-decoder semantic parsers for code generation.

5.1 Future Work

The proposed model serves as a solid foundation for other advanced approaches. In our approach, syntax constraints are not considered during training. The LALR(1) parser only assists in generating logical forms during inference. This approach was chosen to favor increased training efficiency, since employing the probabilistic parser during training time would preclude any simple form of batching. However, this mismatch between code generation during training and inference may lead to suboptimal utilization of the syntax information encoded in the grammar. The most direct approach would track the expected tokens for each training example during training. However, this would incur a lot of computational costs, since each training example would have to be parsed individually.

During inference, mechanisms for enforcing semantic constraints may be considered. One such mechanism is referred to as *execution guidance* [70]. The basic idea of execution

guidance provides a mechanism to ensure logical forms are *executable* in a given context. Assuming SQL queries are generated, the predicted queries may be executed on the target database. Any runtime errors that may occur indicate a semantic error. For example, a comparison by an operator such as ‘ > ‘, between a column of type string and a column of type float, would cause a runtime error. Purely syntactic analysis cannot preclude such errors. By executing partial (executable) logical forms on the target context and applying appropriate corrections, such semantic errors may be avoided.

As proposed by Jia and Liang [30], a coping mechanism has been proven to be beneficial for resolving entities and literals in logical forms. Copying mechanisms usually employ pointer networks [29] that indicate which parts of the input sequence should be copied and inserted into the output sequence. The disadvantage is that it is assumed that every literal value to be generated in the output appears in the input sequence, which might not be the case with complex logical forms. Hybrid *pointer-generator* networks [26] enable the model to dynamically decide whether to copy tokens from the input sequence or generate them from their output vocabulary. This approach liberates the decoder from any constraint prescribing from which source any tokens should be generated.

Finally, neural network architectures different from encoder-decoder models may be explored. So-called *transformer* models [71] achieve state-of-the-art results in a variety of sequence modeling tasks [72], [73]. Like encoder-decoder models, the general transformer architecture relies on an encoder and a decoder. They employ an attention-mechanism, referred to as *self-attention*, that does not rely on recurrent neural networks. This enables them to process sequential input information in parallel, while still generating output tokens sequentially. Their main advantage is that they enable advanced language representation models, such as BERT [74], that can be used to pre-train contextual representations on corpora of unlabeled data. Especially supervised semantic parsing approaches can benefit from such pre-training on large unlabeled corpora since data sparsity is a significant bottleneck in training semantic parsers.

Bibliography

[1] M. Weiser, "The Computer for the 21st Century," *Scientific American*, vol. 265, no. 3, pp. 66–75, Jan. 1991.

[2] A. Clark and D. Chalmers, "The Extended Mind," *Analysis*, vol. 58, no. 1, pp. 7–19, Jan. 1998.

[3] S. Chen, J. Wang, X. Feng, F. Jiang, B. Qin, and C.-Y. Lin, "Enhancing Neural Data-To-Text Generation Models with External Background Knowledge," Hong Kong, China: Association for Computational Linguistics, Nov. 2019.

[4] D. Heidrich and A. Schreiber, "Visualization of a Software System in Virtual Reality," in *MuC'19 Proceedings of Mensch und Computer 2019*, F. Alt, A. Bulling, and T. Döring, Eds., ser. Tagungsband MuC 2019, vol. 134, ACM, Sep. 2019, pp. 905–907.

[5] A. Baranowski, P. Seipel, and A. Schreiber, "Visualizing and exploring OSGi-based Software architectures in augmented reality," in *24th ACM Symposium on Virtual Reality Software and Technology*, ser. VRST '18, ACM, Nov. 2018, 62:1–62:2.

[6] A. Schreiber, M. Misiak, P. Seipel, A. Baranowski, and L. Nafeie, "Visualization of Software Architectures in Virtual Reality and Augmented Reality," in *2019 IEEE Aerospace Conference*, ser. IEEE Aerospace Conference Proceedings, Jun. 2019, pp. 1–12.

[7] P. Seipel, A. Stock, S. Santhanam, A. Baranowski, N. Hochgeschwender, and A. Schreiber, "Speak to your Software Visualization—Exploring Component-Based Software Architectures in Augmented Reality with a Conversational Interface," in *2019 Working Conference on Software Visualization (VISSOFT)*, Sep. 2019, pp. 78–82.

[8] C. Kaliszyk, J. Urban, and J. Vyskočil, "Automating Formalization by Statistical and Semantic Parsing of Mathematics," in *Interactive Theorem Proving*, M. Ayala-Rincón and C. A. Muñoz, Eds., Springer International Publishing, 2017, pp. 12–27.

[9] P. Yin and G. Neubig, "A Syntactic Neural Model for General-Purpose Code Generation," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Vancouver, Canada: Association for Computational Linguistics, Jul. 2017, pp. 440–450.

[10] F. Pelletier, "The Principle of Semantic Compositionality," *Topoi*, vol. 13, pp. 11–24, Mar. 1994.

[11] L. Zettlemoyer and M. Collins, "Online Learning of Relaxed CCG Grammars for Parsing to Logical Form," in *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, Prague, Czech Republic: Association for Computational Linguistics, Jun. 2007, pp. 678–687.

[12] L. Dong and M. Lapata, "Language to Logical Form with Neural Attention," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 33–43.

[13] M. Rabinovich, M. Stern, and D. Klein, "Abstract Syntax Networks for Code Generation and Semantic Parsing," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Vancouver, Canada: Association for Computational Linguistics, Jul. 2017, pp. 1139–1149.

[14] D. Guo, D. Tang, N. Duan, M. Zhou, and J. Yin, "Dialog-to-Action: Conversational Question Answering Over a Large-Scale Knowledge Base," in *Advances in Neural Information Processing Systems 31*, Curran Associates, Inc., 2018, pp. 2942–2951.

[15] J. Berant, A. Chou, R. Frostig, and P. Liang, "Semantic Parsing on Freebase from Question-Answer Pairs," in *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, Seattle, Washington, USA: Association for Computational Linguistics, Oct. 2013, pp. 1533–1544.

[16] Y. Artzi, N. FitzGerald, and L. Zettlemoyer, "Semantic Parsing with Combinatory Categorial Grammars," in *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Tutorials)*, Sofia, Bulgaria: Association for Computational Linguistics, Aug. 2013, p. 2.

[17] S. Clark and J. R. Curran, "Parsing the WSJ Using CCG and Log-Linear Models," in *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL-04)*, Barcelona, Spain, Jul. 2004, pp. 103–110.

[18] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, pp. 200–217, `http://www.deeplearningbook.org`.

[19] P. Pasupat and P. Liang, "Inferring Logical Forms From Denotations," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 23–32.

[20] T. Johnson, "Natural Language Computing: The Commercial Applications," *Knowledge Eng. Review*, vol. 1, pp. 11–23, 1984.

[21] W. A. Woods, "Progress in Natural Language Understanding: An Application to Lunar Geology," in *Proceedings of the June 4-8, 1973, National Computer Conference and Exposition*, ser. AFIPS '73, New York, New York: Association for Computing Machinery, 1973, pp. 441–450.

[22] G. G. Hendrix, E. D. Sacerdoti, D. Sagalowicz, and J. Slocum, "Developing a Natural Language Interface to Complex Data," *ACM Trans. Database Syst.*, vol. 3, no. 2, pp. 105–147, Jun. 1978.

[23] J. M. Zelle and R. J. Mooney, "Learning to Parse Database Queries Using Inductive Logic Programming," in *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, 1996, pp. 1050–1055.

[24] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to Sequence Learning with Neural Networks," in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'14, Montreal, Canada: MIT Press, 2014, pp. 3104–3112.

[25] K. Cho, B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1724–1734.

[26] A. See, P. Liu, and C. Manning, "Get To The Point: Summarization with Pointer-Generator Networks," in *Association for Computational Linguistics*, 2017.

[27] D. Bahdanau, K. Cho, and Y. Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate," 2014.

[28] A. Suhr, S. Iyer, and Y. Artzi, "Learning to Map Context-Dependent Sentences to Executable Formal Queries," in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, New Orleans, Louisiana: Association for Computational Linguistics, Jun. 2018, pp. 2238–2249.

[29] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer Networks," in *Advances in Neural Information Processing Systems 28*, Curran Associates, Inc., 2015, pp. 2692–2700.

[30] R. Jia and P. Liang, "Data Recombination for Neural Semantic Parsing," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 12–22.

[31] C. Xiao, M. Dymetman, and C. Gardent, "Sequence-based Structured Prediction for Semantic Parsing," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 1341–1350.

[32] J. Krishnamurthy, P. Dasigi, and M. Gardner, "Neural Semantic Parsing with Type Constraints for Semi-Structured Tables," in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, Copenhagen, Denmark: Association for Computational Linguistics, Sep. 2017, pp. 1516–1526.

[33] W. Ling, P. Blunsom, E. Grefenstette, K. M. Hermann, T. Kočiský, F. Wang, and A. Senior, "Latent Predictor Networks for Code Generation," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 599–609.

[34] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, `http://www.deeplearningbook.org`.

[35] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[36] Y. Freund and R. E. Schapire, "Large Margin Classification Using the Perceptron Algorithm," in *Proceedings of the Eleventh Annual Conference on Computational Learning Theory*, ser. COLT' 98, Madison, Wisconsin, USA: Association for Computing Machinery, 1998, pp. 209–217.

[37] A. Dey, "Machine Learning Algorithms: A Review," 2016.

[38] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, pp. 129–132, `http://www.deeplearningbook.org`.

[39] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in PyTorch," 2017.

[40] Martin Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*, Software available from tensorflow.org, 2015. [Online]. Available: `https://www.tensorflow.org/`.

[41] S. Bengio and Y. Bengio, "Taking on the Curse of Dimensionality in Joint Distributions Using Neural Networks," *Trans. Neur. Netw.*, vol. 11, no. 3, pp. 550–557, May 2000.

[42] S. Wiseman and A. M. Rush, "Sequence-to-Sequence Learning as Beam-Search Optimization," in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, Austin, Texas: Association for Computational Linguistics, Nov. 2016, pp. 1296–1306.

[43] J. R. Firth, "A synopsis of lingusitic theory. Studies in linguistic analysis.," 1957.

[44] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed Representations of Words and Phrases and Their Compositionality," in *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'13, Lake Tahoe, Nevada: Curran Associates Inc., 2013, pp. 3111–3119.

[45] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American Society for Information Science*, vol. 41, no. 6, pp. 391–407, 1990.

[46] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient Estimation of Word Representations in Vector Space," in *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2013.

[47] J. Pennington, R. Socher, and C. D. Manning, "GloVe: Global Vectors for Word Representation," in *Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1532–1543.

[48] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.

[49] S. Hochreiter and J. Schmidhuber, "Long Short-term Memory," *Neural computation*, vol. 9, pp. 1735–80, Dec. 1997.

[50] M. E. Lesk and E. Schmidt, "Lex—a Lexical Analyzer Generator," in *UNIX Vol. II: Research System (10th Ed.)* USA: W. B. Saunders Company, 1990, pp. 375–387.

[51] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006, pp. 109–190.

[52] ——, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006, pp. 191–209.

[53] ——, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006, pp. 259–278.

[54] *Bison, Version 2.1*, GNU Software Foundation. [Online]. Available: `http://www.gnu.org/software/bison/bison.html`.

[55] C. Finegan-Dollak, J. K. Kummerfeld, L. Zhang, K. Ramanathan, S. Sadasivam, R. Zhang, and D. Radev, "Improving Text-to-SQL Evaluation Methodology," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, Melbourne, Australia: Association for Computational Linguistics, Jul. 2018, pp. 351–360.

[56] E. Shinan, *Lark - a parsing toolkit for Python*, `https://github.com/lark-parser/lark`, 2020.

[57] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 8024–8035.

[58] J. Earley, "An Efficient Context-Free Parsing Algorithm," *Commun. ACM*, vol. 13, no. 2, pp. 94–102, Feb. 1970.

[59] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A Matlab-like Environment for Machine Learning," in *BigLearn, NIPS Workshop*, 2011.

[60] E. Shinan, *Lark Grammar Reference*, `https://lark-parser.readthedocs.io/en/latest/grammar.html`, 2020.

[61] V. Zhong, C. Xiong, and R. Socher, "Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning," *ArXiv*, vol. abs/1709.00103, 2018.

[62] P. J. Price, "Evaluation of Spoken Language Systems: The ATIS Domain," in *Proceedings of the Workshop on Speech and Natural Language*, ser. HLT '90, Hidden Valley, Pennsylvania: Association for Computational Linguistics, 1990, pp. 91–95.

[63] S. Iyer, I. Konstas, A. Cheung, J. Krishnamurthy, and L. Zettlemoyer, "Learning a Neural Semantic Parser from User Feedback," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Vancouver, Canada: Association for Computational Linguistics, Jul. 2017, pp. 963–973.

[64] A.-M. Popescu, O. Etzioni, and H. Kautz, "Towards a Theory of Natural Language Interfaces to Databases," in *Proceedings of the 8th International Conference on Intelligent User Interfaces*, ser. IUI '03, Miami, Florida, USA: Association for Computing Machinery, 2003, pp. 149–157.

[65] F. Li and H. V. Jagadish, "Constructing an Interactive Natural Language Interface for Relational Databases," *Proc. VLDB Endow.*, vol. 8, no. 1, pp. 73–84, Sep. 2014.

[66] N. Yaghmazadeh, Y. Wang, I. Dillig, and T. Dillig, "Type- and Content-Driven Synthesis of SQL Queries from Natural Language," Feb. 2017.

[67] P. Yin, B. Deng, E. Chen, B. Vasilescu, and G. Neubig, "Learning to Mine Aligned Code and Natural Language Pairs from Stack Overflow," in *International Conference on Mining Software Repositories*, ser. MSR, ACM, 2018, pp. 476–486.

[68] C. Quirk, R. Mooney, and M. Galley, "Language to Code: Learning Semantic Parsers for If-This-Then-That Recipes," in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, Beijing, China: Association for Computational Linguistics, Jul. 2015, pp. 878–888.

[69] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, pp. 108–114, `http://www.deeplearningbook.org`.

[70] C. Wang, K. Tatwawadi, M. Brockschmidt, P.-S. Huang, Y. Mao, O. Polozov, and R. Singh, *Robust Text-to-SQL Generation with Execution-Guided Decoding*, 2018.

[71] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is All you Need," in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., Curran Associates, Inc., 2017, pp. 5998–6008.

[72] M. Ott, S. Edunov, D. Grangier, and M. Auli, "Scaling Neural Machine Translation," in *Proceedings of the Third Conference on Machine Translation: Research Papers*, Brussels, Belgium: Association for Computational Linguistics, Oct. 2018, pp. 1–9.

[73] B. Wang, R. Shin, X. Liu, O. Polozov, and M. Richardson, *RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers*, 2019.

[74] I. Turc, M.-W. Chang, K. Lee, and K. Toutanova, "Well-Read Students Learn Better: On the Importance of Pre-training Compact Models," *arXiv preprint arXiv:1908.08962v2*, 2019.

Appendix

Appendix A

Documentation of `preprocess.py` arguments

Argument	Description
`grammar`	Lark grammar for parsing target samples.
`start`	The start rule of the grammar. Defaults to "start".
`src_train`	Training dataset source samples.
`tgt_train`	Training dataset target samples.
`src_test`	Test dataset source samples.
`tgt_test`	Test dataset target samples.
`src_dev`	Validation dataset source samples.
`tgt_dev`	Validation dataset target samples.
`save_data`	Path and name for saving preprocessed data.
`check`	Check whether target examples are valid programs.

Documentation of `train.py` arguments

Argument	Description
`data`	The input datasets and vocabularies.
`save`	The name under which the model will be saved.
`out`	The file in which training info is logged.
`validate`	Whether to validate training progress on the dev split.
`early_stop`	Stop training when validation accuracy has not improved since the specified number of iterations.
`best_gold`	Save model with best validation gold accuracy when validating.

`epochs`	Number of training iterations.
`batch_size`	Number of samples to batch.
`learning_rate`	Learning rate for SGD optimizer.
`gradient_clip`	Clipping to prevent exploding gradients.
`attention`	Attention mechanism according to Bahdanau.
`copy`	Copy attention for copying tokens from the input sentence.
`stack_encoding`	Value stack encodings used during decoding.
`stack_emb_size`	Dimension of embedding vector for stack encoder.
`stack_hidden_size`	Dimension of stack encoder hidden state.
`stack_dropout`	Dropout applied to stack encoder embeddings.
`layers`	Number of layers to use for encoder and decoder.
`enc_emb_size`	Dimension of embedding vector for encoder.
`dec_emb_size`	Dimension of embedding vector for decoder.
`enc_hidden_size`	Dimension of encoder hidden state.
`dec_hidden_size`	Dimension of decoder hidden state.
`enc_rnn_dropout`	Dropout applied to encoder outputs and hidden states.
`dec_rnn_dropout`	Dropout applied to decoder outputs and hidden states.
`enc_emb_dropout`	Dropout applied to encoder embeddings.
`dec_emb_dropout`	Dropout applied to decoder embeddings.
`teacher_forcing`	Ratio of decoder's own predictions and true target values used during training.
`bidirectional`	Set encoder to compute forward and backward hidden states.

Documentation of `translate.py` arguments

Argument	**Description**
`model`	The model file to use for translation.
`eval`	The test dataset to evaluate.
`out`	The logging file.
`beam_width`	The beam with for the parser decoder. Defaults to greedy search.
`no_parser`	Turns off parser-assisted decoding.

Appendix B

Configuration for GeoQuery Standard, without Attention

Argument	Value
attention	False
bidirectional	False
batch_size	64
layers	1
learning_rate	0.1
enc_hidden_size	224
dec_hidden_size	224
enc_emb_size	128
dec_emb_size	128
enc_rnn_dropout	0.05
dec_rnn_dropout	0.05
enc_emb_dropout	0.1
dec_emb_dropout	0.1
teacher_forcing	0.8

Configuration for GeoQuery Oracle, without Attention

Argument	Value
attention	False
bidirectional	False
batch_size	32
layers	1
learning_rate	0.1
enc_hidden_size	128
dec_hidden_size	128
enc_emb_size	92
dec_emb_size	92

enc_rnn_dropout	0.05
dec_rnn_dropout	0.05
enc_emb_dropout	0.1
dec_emb_dropout	0.1
teacher_forcing	0.8

Configuration for GeoQuery Standard, with Attention

Argument	Value
attention	True
bidirectional	True
batch_size	32
layers	1
learning_rate	0.1
enc_hidden_size	128
dec_hidden_size	128
enc_emb_size	96
dec_emb_size	96
enc_rnn_dropout	0.05
dec_rnn_dropout	0.05
enc_emb_dropout	0.1
dec_emb_dropout	0.1
teacher_forcing	0.8

Configuration for GeoQuery Oracle, with Attention

Argument	Value
attention	True
bidirectional	True
batch_size	32

layers	1
learning_rate	0.1
enc_hidden_size	96
dec_hidden_size	96
enc_emb_size	64
dec_emb_size	64
enc_rnn_dropout	0.05
dec_rnn_dropout	0.05
enc_emb_dropout	0.1
dec_emb_dropout	0.1
teacher_forcing	0.8

Configuration for ATIS Standard, without Attention

Argument	**Value**
attention	False
bidirectional	False
batch_size	128
layers	2
learning_rate	0.1
enc_hidden_size	512
dec_hidden_size	512
enc_emb_size	256
dec_emb_size	256
enc_rnn_dropout	0.05
dec_rnn_dropout	0.05
enc_emb_dropout	0.1
dec_emb_dropout	0.1
teacher_forcing	0.8

Configuration for ATIS Oracle, without Attention

Argument	Value
attention	False
bidirectional	False
batch_size	128
layers	2
learning_rate	0.1
enc_hidden_size	512
dec_hidden_size	512
enc_emb_size	256
dec_emb_size	256
enc_rnn_dropout	0.05
dec_rnn_dropout	0.05
enc_emb_dropout	0.1
dec_emb_dropout	0.1
teacher_forcing	0.8

Configuration for ATIS Standard, with Attention

Argument	Value
attention	True
bidirectional	True
batch_size	100
layers	2
learning_rate	0.1
enc_hidden_size	256
dec_hidden_size	256
enc_emb_size	128
dec_emb_size	128
enc_rnn_dropout	0.05

dec_rnn_dropout	0.05
enc_emb_dropout	0.1
dec_emb_dropout	0.1
teacher_forcing	0.9

Configuration for ATIS Oracle, with Attention

Argument	Value
attention	True
bidirectional	True
batch_size	128
layers	2
learning_rate	0.1
enc_hidden_size	256
dec_hidden_size	256
enc_emb_size	128
dec_emb_size	128
enc_rnn_dropout	0.1
dec_rnn_dropout	0.1
enc_emb_dropout	0.2
dec_emb_dropout	0.2
teacher_forcing	0.8