Understanding Generalization: Learning Quantifiers and Negation with Neural Tensor Networks

MSc Thesis *(Afstudeerscriptie)*

written by

Michael Repplinger

under the supervision of **Dr. Willem Zuidema**, and submitted to the Board of Examiners in partial fulfillment of the requirements for the degree of

MSc in Logic

at the *Universiteit van Amsterdam*.

Date of the public defense:	**Members of the Thesis Committee:**
August 31, 2017	Dr. Tejaswini Deoskar
	Prof. Dr. Benedikt Löwe (*chair*)
	Dr. Jakub Szymanik
	Dr. Willem Zuidema

INSTITUTE FOR LOGIC, LANGUAGE AND COMPUTATION

Abstract

The following investigation is focused on the intersection of symbolic and distributional approaches to natural language semantics. Broadly speaking, we analyze symbolic approaches to semantics in order to learn how the performance of distributional models can be improved that are applied to compositional language tasks. Specifically then, we set out to discover empirical justification for the often claimed advantage of employing higher-order tensors in semantic vector space models.

Using an earlier natural language inference experiment, and adjusting it towards a more stringent test of generalization performance, we are able to find clear signs in support of the conjectured advantage of tensor models. In our experiments, a clear difference emerges in generalization performance between a conventional matrix-based tree-structured neural network and a tensor-based variant of the architecture.

To discover the cause for this performance difference, we visualize sentence representations of models trained on the semantic task. We find evidence of linearization of complex data in the tensor model, such as hyperplane separation of negated expressions, and the systematic organization of quantified expressions. We then suggest an explanation of our prediction results, by linking the observed properties of the model representations with the logical requirements of the inference task.

Für Paul

Acknowledgments

I want to begin by offering my heartfelt gratitude to Jelle, who really was the ideal advisor. You generously offered your time, your insights, and often enough, solid advice that made me wonder if you knew me better than I knew myself.

Thank you, Ulle, Maria, Tanja, for offering guidance, encouragement, and, a chance.

Then, a major thank you, Mees, for breaking it down for me with patience and clarity.

And thank you, Marco, Dieuwke, Samira, Sander, Bas and Sara for any number of insightful discussions and uplifting conversations during the writing period.

Thank you Ines, and thank you Umberto. You have supported me for such a long time. I am happy to call you my friends.

For your willingness to listen, and for your ever calm support and guidance: thank you, Andre.

To my mother and sisters, Annemay, Ina, Lilly, Lena: thank you for your patience, and thank you for your unconditional love.

Finally – thank you, Gracia. For everything. (Also, that can of Coke made all the difference.)

Contents

Chapter 1

Introduction

Symbolic models of language are fully general; distributional models actually work would be an inappropriate way to introduce a thesis that aims for a nuanced analysis of the two complementary approaches. So we won't.

Natural language semantics as a field of research has existed for centuries, and our understanding of the rules that describe the language we speak have been examined closely, and with increased formal sophistication. The method by which the understanding was gained is grounded on symbolic representations, and to many, it appears the *only* way to gain a well-founded understanding of the subject.

More recently however, a new approach has emerged, in its latest incarnation referred to as *distributional semantics*. The symbolic definitions and descriptions of the previous approach are replaced here by data-driven methods, and a firm statistical foundation. The focus of this thesis will mostly lie on the latter approach, but with a keen interest in gaining additional insights from the former approach.

The major themes of this thesis relates to *compositionality* of language, and the question how compositionality can be accounted for in models of distributional semantics. We will see soon that a related question emerges, regarding the ability or inability of models that solely learn by example, to *generalize* from the cases they have seen to cases that are new to them. The two themes, of compositionality and generalization, will eventually converge and form a more specific question, regarding the role of higher-order tensors in distributional models.

To address these questions, we experimentally evaluate two related models, one based on a conventional function learned from the data, the other being based on a higher-order tensor. We then describe our qualitative analysis of the experimental results, by visualizing and interpreting the representations the models learned from the data.

While we cannot hope to fully and conclusively answer the questions raised above, we hope that our analysis adds some insight into the question how the *compositional* mechanisms encountered in symbolic models can be successfully integrated into distributional models of semantics.

Part I

Background

Chapter 2

Symbolic and Distributional Semantics

This chapter is intended to provide the necessary background to follow the analysis of this thesis. We begin by outlining two approaches to natural language semantics in the symbolic tradition. Next, we introduce some central concepts of linear and multilinear algebra, required to follow the technical details of the models discussed throughout this thesis. The last part of the chapter reviews the literature of distributional semantics. We begin by discussing models of distributional semantics operating at a word level, followed by a review of compositional models, using a broad division of compositional distributional approaches into two classes.

Model Terminology In our analysis we plan to use the following nonstandard terminology, intended to disambiguate two model classes that will be frequently discussed: The *recursive neural network* (Socher, 2014) and the *recurrent neural network* (Elman, 1990) are both commonly abbreviated as 'RNN', in addition to being named similarly. For reasons that will be outlined in Chapter 4, we usually prefer to describe the model of Socher (2014) as a *tree-structured* recurrent neural network, using the abbreviation *tRNN*. Analogously, we abbreviate the related *recursive neural tensor network* as *tRNTN* in our thesis.

2.1 Symbolic Approaches

In this section we will focus on semantic theories in the symbolic tradition. We will sketch, and briefly discuss theories of formal semantics, mainly with the goal to provide the background for our later contrast with distributional approaches to semantics. We then describe NatLog, a computational model of natural language inference, and the symbolic system that generated the training data for the models of Chapters 5 and 6.

2.1.1 Formal Semantics

Tradition demands that discussions of formal semantics begin by citing the well-known principle of compositionality. Despite its abstract nature, the principle can be interpreted as the foundation of modern theories of formal semantics – providing a motivation to discuss it besides tradition.

Principle of Compositionality The *principle of compositionality*, generally associated with the works of Gottlob Frege (Frege, 1892), is commonly phrased along the lines of: "The meaning of a complex expression is determined by the meanings of its constituents and the rules used to combine them." In its abstract form, the principle leaves many aspects of semantic analysis unspecified. Nonetheless, two constraints can be identified: Studying the meaning of language requires analyzing the meaning of individual components, as well as the rules by which we construct larger expressions from smaller ones.

Closely related to compositionality is the notion of *recursion*. Recursively defined processes are widely believed to underly the capacity of speakers of a language to build and understand arbitrary expressions of the language. By recursive *syntactic* processing, speakers can, in principle, build an infinite set of complex expressions from a finite set of simple "building blocks". By a parallel recursive *semantic* process, speakers are then able to express and understand a potential infinitude of distinct meanings.

From Principles to Theories The following lucid description of compositionality is offered by Barbara Partee:

> The compositionality requirement . . . is almost uncontroversial when stated informally. But formalizing it requires an explicit theory of syntax, an explicit theory of semantics, and an explicit theory of the mapping from one to the other.[1]

Based on the general description of a grammar in Montague (1970b), Partee (2014) adds that these "explicit theories" can be understood as syntactic and semantic algebras, together with a homomorphism instantiating the mapping. While still allowing considerable theoretical variation, the principle gained concreteness under this view: The objective of the syntactic algebra is to construct complex expressions (phrases and sentences) from the basic ones (words) by recursive operations, ideally deriving a set of expressions that is identical to the expressions encountered in natural language.[2]

While natural language expressions can be ambiguous, once a (logical) translation has been determined – possibly several distinct translations for an ambiguous expression – by the homomorphism requirement we ensure that any element of the syntactic algebra is mapped to a unique element of the semantic algebra. The latter

[1] Partee (2001). Parenthetical remarks omitted.

[2] If smaller, the grammar is a *fragment*, if larger, it derives expressions not found in reality.

provides expressions with meaning, often through model-theoretic interpretation, while the homomorphism requirement ensure that each expression is assigned exactly *one* such interpretation, thus formally specifying the "meaning functionality" demand expressed abstractly by the original principle.[3]

In this form, the principle admits a general framework of *formal semantics*, semantic theories using the language of mathematics, following the spirit of Frege's principle. Next, we will sketch the basic principles of Montague Semantics – arguably the most influential semantic theory, and the proposal which established the formal understanding of *compositionality* in the sense outlined above.

Montague Semantics *Montague Semantics* or Montague Grammar, developed mostly in a series of seminal papers Montague (1970a,b, 1973), is widely considered to be the first successful attempt to fully formalize the semantics of natural language, or at least some fragment of it.

While the earlier Montague (1970a) showed that the direct interpretation of expressions, without an intermediate logical language, is possible in principle, the system developed in Montague (1970b) and Montague (1973) proved to be more popular, proceeding by translation of natural language expressions into an intermediate logical language, followed by interpretation by model-theoretical semantics. It is the latter approach we will outline in here.

In broad terms, Montague's innovative proposal consists of the compositional translation from natural language to a logical language, such that (syntactic) derivation proceeds in parallel to assignment of semantic representations. This parallel definition ensures that each derivation of a sentence is assigned a unique and appropriate meaning by the theory. The synchronous processing of syntax and semantics is made available by Montague's highly expressive *intensional logic*, a higher-order typed (intensional) logic, enriched by lambda abstraction allowing the construction of appropriate functions at each derivation step.

Our sketch of Montague's system omits several features that were revolutionary at the time, but are now standard techniques of the field, such as generalized quantifiers, or 'extensionalizing' intensional expressions via possible worlds. We refer interested readers to Gamut (1991) or Heim and Kratzer (1998) for thorough introductions to semantic systems in the spirit of Montague's proposal.

Toy Grammar We exemplify the meaning derivation process, outlined in general terms above, by an example sentence and a toy syntax, given in the form of a (simplified) categorial grammar. Table 2.1 describes the syntactic categories and their corresponding semantic types: Names are of syntactic category PN (proper nouns), and semantic type e, a basic type of entities, interpreted as elements of the domain of discourse. Transitive verbs are of category TV, interpreted as a function from entities to a function from entities to truth values, of type $(e \rightarrow (e \rightarrow t))$. Composition of a

[3] Note that by choice of a homomorphism between syntax and semantics, every syntactic expression is assigned just one meaning, however, different expressions can *share* this meaning.

Class	Syntax	Semantics	Expressions
		basic	
S	S	t	sentence
PN	PN	e	proper noun (names)
		complex	
N	S/PN	$e \rightarrow t$	common noun (plural)
ITV	S\N	$e \rightarrow t$	intransitive verb
TV	(S\N)/N	$e \rightarrow (e \rightarrow t)$	transitive verb
ADJ	N/N	$(e \rightarrow t) \rightarrow (e \rightarrow t)$	adjective (predicative)
CON	(N\N)/N	$(e \rightarrow t) \rightarrow ((e \rightarrow t) \rightarrow (e \rightarrow t))$	conjunction (nominal)

Table 2.1: Correspondence between syntax and semantics

transitive verb with its object results in category VP (a verb phrase), interpreted as a function from individuals to truth values, $(e \rightarrow t)$. Finally, the full sentence is of category S, of basic type t of truth values, and interpreted as 'true' or 'false'.

The left side of Fig. 2.1 is a parse tree of the example sentence "Gottlob loves Yoshua", under the syntactic rules expressed by the complex syntactic types in Table 2.1. Analogously, the right tree shows the result of assigning each node on the left a meaning. The meaning of the names are logical constants, interpreted as domain elements, while the meaning of the verb is a lambda expression over two individual evaluating whether they stand in the relation denoted by the relational constant. These meaning assignments are lexical, while the assignments of the remaining nodes result from rules that are the semantic analog to the syntactic phrasal rules. They correspond here to function-argument application, first for the object of the transitive verb, then the subject. The phrasal nodes on the right also contain the expressions resulting from *beta reduction*, replacing free occurrences of a parameter variable with the argument value. The topmost node, representing the meaning of the entire sentence, yields 'true' if the denoted individuals stand in the denoted relation, and 'false' otherwise.

Note that the bidirectional arrow between trees should not be read as saying that every sentence has a unique logical translation, and therefore, a unique meaning. It is intended to show that sentences, given a particular *syntactic derivation* are assigned a unique interpretation.

Criticism and Alternatives The field of modern formal semantics increased the formal understanding of language meaning substantially over the past decades. Nonetheless, criticism of the approach can be raised, of which we give two examples.

One argument relates to the observation that formal semantics is mainly concerned with *structural*, compositional aspects of meaning. Analysis of the meaning basis, i.e. lexical content, has arguably received less attention. In Section 2.2 we will see that major progress in this area was recently made by non-symbolic machine

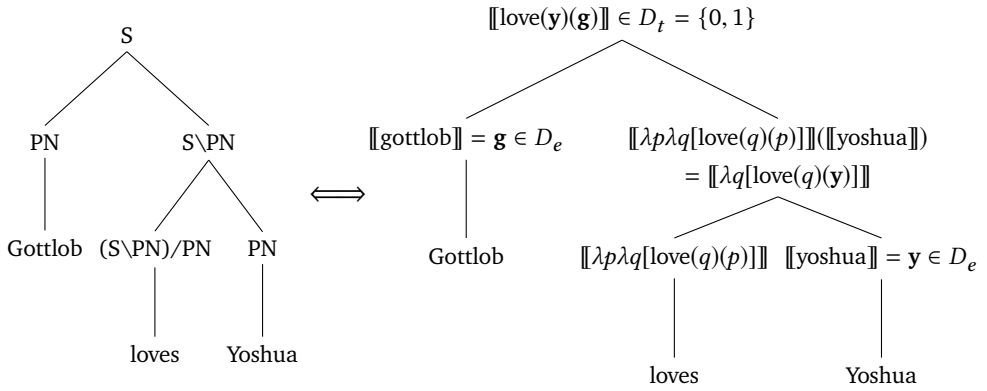

Figure 2.1: Translation to logical language

learning models, raising the question whether similar approaches could be applied to compositional meaning analysis.

Another point concerns the limited integration of individual semantic theories. Montague's original proposal was a "method of fragments" – identifying a well-delineated syntactic fragment of the language, then formally describing this fragment. Modern semantic theories mostly no longer take this approach, opting instead for the descriptions of particular *semantic phenomena* that are not confined to a syntactic *fragment*. Consequently, no part of the language can be set aside and semantically described in *completion*, which is a requirement however for *computational* models of symbolic semantics (Partee, 2001).

The theoretical question whether there are viable alternatives to symbolically grounded semantics is part of a long-running debate, for example, in the discussion of the *binding problem*. We will not attempt to directly answer this complex question here, but will indirectly address it throughout our analysis.

2.1.2 Natural Logic and Language Inference

Natural logic generally describes logical systems encoding reasoning patterns encountered in natural language, in contrast to classically defined logic, centered around the notion of formal validity of inference, or *logical reasoning*. The term *natural logic*, and the framing of the problem in a modern logical context, is due to Lakoff (1970). One application of natural logics is the modeling of *natural language inference* (NLI), automated systems performing reasoning tasks over natural language expressions.

Natural logic is linked to the *monotonicity* properties of generalized quantifiers. We illustrate monotonicity by example: "No European sings" entails "No German sings", as well as: "No European sings loudly" – when replacing a general expression by a more specific one, entailment holds. The property of admitting entailment from *general* to *specific* is referred to as *downward monotonicity*. Under the perspective of generalized quantifiers, we can interpret 'no' as relating its two arguments, 'European' and 'sing'. Downward entailment holds for either of them, and we say that 'no' is downward

Symbol	Name	Example	Interpretation
$x \equiv y$	equivalence	(couch, sofa)	$X = Y$
$x \sqsubset y$	forward entailment (*excl.*)	(crow, bird)	$X \subset Y$
$x \sqsupset y$	reverse entailment (*excl.*)	(bird, crow)	$X \supset Y$
$x \wedge y$	negation	(able, unable)	$X \cap Y = \emptyset \wedge X \cup Y = \mathcal{U}$
$x \mid y$	alternation	(cat, dog)	$X \cap Y = \emptyset \wedge X \cup Y \neq \mathcal{U}$
$x \smile y$	cover	(animal, non-dog)	$X \cap Y \neq \emptyset \wedge X \cup Y = \mathcal{U}$
$x \# y$	independence	(hungry, hippo)	*all other cases*

Table 2.2: The seven basic relations of MacCartney's logic[4]

monotone *in both arguments*. Consider next that "Every European sings" entails "Every German sings", but does not entail "Every European sings loudly". However, "Every European sings loudly" entails "Every European sings" – we say 'every' is downward monotone in its first argument and *upward monotone* in its second argument.

MacCartney Logic and NatLog The natural logic developed in MacCartney (2009) is described by the author as an extension of the monotonicity calculus by Valencia (1991) and van Benthem (1991). The logical model has been implemented computationally as the *NatLog* system, outlined for example in MacCartney and Manning (2009). The inference mechanism of MacCartney's natural logic operates directly on natural language expressions, without intermediate translation to a logical language – similar to the original Montague framework mentioned in Section 2.1.1.

Traditionally, inference of language is modeled in terms of two relations, entailment, and contradiction, sometimes adding a third, independence. MacCartney's system defines seven relations, in order to adequately cover the entire spectrum of natural language inferences. The seven basic relations, with examples and their set-theoretic definitions, are listed in Table 2.2. Some relations are familiar (equivalence, entailment, independence). Note that entailment is split over two relations, with two corresponding symbols: forward entailment and reverse entailment. Next, 'negation' relates mutually exclusive concepts whose union covers the entire domain; 'alternation' relates mutually exclusive items whose union does not cover the domain; 'cover' relates overlapping concepts whose union covers the domain.

The basic relations are related to monotonicity by the *projective properties* of expressions. For example, "fail to $\boldsymbol{x}$" is downward monotone: "fail to sing" entails "fail to sing loudly". Its projective property with respect to $\sqsubset$, the entailment relation inside the logic, is characterized by a direction reversal: "Sing loudly" $\sqsubset$ 'sing', but: "fail to sing loudly" $\sqsupset$ "fail to sing". In total, these projective properties of expressions over the basic relations characterize how *context* influences the *composition* of inference – in a sense which will be made more precise next.

[4] Table adapted from MacCartney and Manning (2009).

Compositional Entailment The system described so far only derives direct lexical instances of entailment, i.e. from one word to another. Extension to the (propositional) sentence level is intuitively straightforward, by interpretation of propositions as sets of possible worlds. For instance, "all dogs bark" and "not all dogs bark" are related by the basic relation 'negation'; their sets of worlds are disjoint, and the union of their worlds contains all possible worlds (by the law of excluded middle).

The next extension consists of adding compositionality to the system, to allow inferences beyond a single step. To this end, the logic is equipped with a simple proof calculus. Compositional derivations proceed via a sequence of *edit* operations, i.e. an algorithmic replacement process, which encodes both lexical knowledge and the monotonicity contributions by the context, e.g. of quantified subexpressions. The atomic edit operations are:

substitutions are one-for-one lexical replacements yielding the relation that holds between the replaced items, e.g. 'sofa' $\mapsto$ 'couch' yields: $\equiv$

deletions replace a complex expression by a simpler one, usually on items under entailment, e.g. "red car" $\mapsto$ 'car' yields: $\sqsubset$

insertions are defined symmetrically to deletions, so for items under reverse entailment, e.g. 'car' $\mapsto$ "red car" yields: $\sqsupset$

Given a premise and a hypothesis, complex inferences are performed as a series of these edits, in which the edit sequence transforms the premise into the hypothesis. The (predicted) *relation* holding between the premise and hypothesis is then determined as follows: Each edit of the sequence is associated with a relation (e.g. the relation that held between two words that were substituted). These chained edits are then joined to form (ideally) a single predicted relation. The main complexity of this process is to determine how relations are joined, given their context. To this end, the process is defined as a function of the monotonicity or projective properties described above, ensuring that the outcome of joining takes into account the monotonicity properties of the context in which the edit that generated the relation took place.

Limitations of the Logic MacCartney and Manning (2009) evaluate an implementation of the logic on several semantic tasks, and performance on the tasks suggests that the edit-based proof calculus is sufficient for the primary aim of the system of modeling natural language inference. Seen as a formal logic however, the authors note that the deductive power of the system is strictly lower than that of classical first-order logic. For one, the system does not allow combination of multiple premises, i.e. the classical syllogisms based on two premises and one conclusion are not directly representable in the system.

Another limitation, more relevant in the context of our own analysis, is that MacCartney's natural logic turns out to be semantically incomplete. MacCartney (2009) shows that there is no (deterministic) way to derive an edit sequence establishing the *duality* of existential and universal quantifier.

This duality, commonly known as the *de Morgan's laws for quantifiers* in classical first-order logic, expresses:

$$\forall x P(x) \equiv \neg[\exists \neg P(x)] \tag{2.1}$$

$$\exists x P(x) \equiv \neg[\forall \neg P(x)] \tag{2.2}$$

This deductive limitation in terms of quantifier equivalences will gain practical relevance in Chapter 6, where its effect was instrumental for one of the results.

2.2 Distributional Semantics

We turn now to the models of distributional semantics, explaining the rationale behind the models and discuss their advantages and limitations.

The following sections presuppose some basic knowledge of linear algebra and tensor calculus. For readers who want to refresh their memory, a brief linear algebra primer is provided in Appendix A. In this primer we also introduce the basic concepts behind higher-order tensors, and briefly touch on a few related points, such as tensor factorization and the difference between the definition of tensors in the machine learning literature and in pure mathematics.

Distributional Hypothesis Models of *distributional semantics*, or semantic vector space models, are a class of semantical models based on statistical and machine learning techniques, usually understood as realizations of the *distributional hypothesis*. Following Firth (1957), the hypothesis is often expressed as: "A word is characterized by the company it keeps". The related idea, of language meaning being defined by language usage, can be traced back further to a central concept of Wittgenstein (1953). Paraphrasing, the hypothesis states that *semantic similarity* can be expressed by *distributional similarity*, suggesting that statistical analysis of language is an effective way of determining word meaning.[5]

Spatial Representation While the above suggests a method to gather semantic information, we also require a formal system to *represent* the information. All models discussed here are *vector space* models, in which words correspond to elements of that space, i.e. each word is represented by a vector. Given this representation, semantic similarity between words can be defined in terms of distance between their vector representations. More generally, a vector space representation enables spatial or geometric interpretation of semantic meaning, providing the semantic theory with the complete formal machinery of (finite) vector spaces. In particular, a natural extension of *compositional* semantics is provided, leading to the models discussed in Chapter 3.

[5] We will sometimes refer to the methods of this section as belonging to *contextual* distributional semantics, in contrast to *compositional* distributional semantics.

	animal	large	small	USB
MOUSE	8	2	4	3
ELEPHANT	9	5	1	0
CAR	1	4	3	0

Table 2.3: Idealized context-counts for MOUSE, ELEPHANT and CAR

2.2.1 A Simple Example Space

Table 2.3 is an idealized illustration of co-occurrence counts and a corresponding semantic space for three nouns, 'mouse', 'elephant' and 'car'. Values in this table indicate how often a target word, i.e. the word we aim to represent as a vector, appeared near certain other words in a corpus. For example, 'mouse' and 'animal' co-occurred eight times, while 'car' and 'animal' did so only once. Based on these hypothetical counts, our semantic space would be four-dimensional, with vectors consisting of raw co-occurrence counts. In reality, dimensionality is usually in the order of hundreds, and raw counts are transformed into (weighted) frequency values.

Using *cosine similarity* defined in Appendix A on pairs of these word vectors, we can calculate their semantic similarity: 0.86 (mouse, elephant), 0.57 (mouse, car), 0.61 (elephant, car). This would then represent, as intended, that 'mouse' and 'elephant' are more similar to each other than either of the two is to 'car' (animals vs. non-animal), and that 'car' is (slightly) more similar to 'elephant' than to 'mouse' (being somewhat more similar in size). Note also the co-occurrence of 'mouse' with a seemingly unrelated word, 'USB', intended to illustrate the problem of *polysemy*. In most models, the vector representation of 'mouse' would be a 'mix' of contexts where 'mouse' refers to a rodent and contexts where the word refers to a computer component.

2.2.2 Models and Parameters

We now present some important choices and parameters in the definition of distributional semantic models, exemplifying these choices by relevant models implementing them in certain ways.

Context Several parameters and modeling choice are left to be defined in specifying a full semantic space model. First, the definition of what constitutes *context*. Commonly, this context is defined as "the N neighboring words of the target word", where N is the *context window*, another parameter. An early successful implementation of this *classical* context definition, populating vectors by (weighted) co-occurrence counts of neighboring words, is the *Hyperspace Analog to Language* (HAL) system by Lund and Burgess (1996). However, other choices are possible, for example, context being defined on a character, i.e. sub-word level, to allow sharing of contexts across morphologically similar words Bojanowski et al. (2016). Other models, such as (LDA)

topic models, use an entire document as context, deriving a low number of (latent) topics which are defined by distributions over words.

Similarity Another modeling choice, although not strictly part of the "meaning extraction" architecture itself, is the *similarity* or *distance* metric. Choosing a particular metric, like cosine similarity, implicitly encodes certain assumptions about what constitutes semantic similarity, in addition to being motivated by statistical considerations.

Weighting While all models, in some form, derive semantic information from the analysis of lexical co-occurrence, i.e. are based on *context counting*, most do not use unprocessed counts to populate the vector components, but instead transform them by *weighting schemes*. Intuitively, these processing steps can be seen as adjusting counts for word frequency, giving more weight to words that are rare but informative. For example, a word that occurs frequently and across several unrelated meaning contexts, is semantically less informative than a word that occurs infrequently, and only within a small number of related contexts. Weighting is aimed at discounting the contribution of the former words, while giving more weight to the latter. A popular method used for this purpose is *pointwise mutual information* (PMI).

Dimensionality Several approaches perform *dimensionality reduction* on the derived representations. A common processing pipeline, described for example in Baroni et al. (2014b), roughly proceeds as follows: construct co-occurrence matrices from corpus analysis, re-weight counts by PMI, compress matrices by non-negative matrix factorization (NMF) or singular value decomposition (SVD). The dimensionality reduction step is motivated by two main concerns: computational efficiency, and possibly greater *generalization* capacity of the model Levy and Goldberg (2014). The latter effect can result from merging (similar) contexts, i.e. associating a word with contexts of similar words even though it might have never appeared directly in these contexts itself, thus allowing the model to uncover additional similarities.

Objective A fundamental choice not explicitly mentioned so far is the *objective* of the algorithm analyzing the data. The classical method, described above, is aimed at *counts* of context items of a given word. A recent class of models, neural word embeddings, instead aims to *predict*, either (the most likely) word given a context, or the (most likely) context for a given word. We will discuss the difference between objectives in the next section.

2.2.3 Neural Word Embeddings

Neural network-based word representations gained popularity in recent years. We will sketch the idea behind these models, and mention some reasons brought forward to explain their success.

Sparse vs. Dense Word-level models of distributional semantics can be distinguished by the type of their vectorial representation. First, *sparse* representations – high-dimensional spaces in which dimensions directly correspond to contexts, consequently leading to vector representations that are numerically *sparse* across components, since a word does not appear in every context. The classical counting approaches, possibly including count re-weighting, but excluding dimensionality reduction, would fall into this category. Second, *dense* representation – (usually) lower-dimensional spaces, in which dimensions are the result of compressing (classically) derived counts by dimensionality reduction, or are constructed densely immediately, i.e. as *embeddings*.

History The latter approach is taken by models referred to as neural language models of *word embeddings*. These models are usually traced back to early suggestions by Hinton (1986), and Bengio et al. (2003), who applied a (regular) neural architecture to the task of deriving word meanings. Embeddings gained an immense boost in popularity after the proposals by Mikolov et al. (2013b) and Mikolov et al. (2013a), implemented as `word2vec`. These methods can be understood as feedforward neural networks without hidden layers and nonlinear activation functions, which the authors identified as computational bottlenecks of the previous models.

word2vec The embeddings created by word2vec consist of two distinct, but closely related algorithms. The first model, *Continuous Bag-of-Words* (CBOW), learns to *predict a word*, given a context of surrounding words. A "projection layer" turns discretely encoded (sparse) word representations into continuous (dense) representations, which are then fed into a matrix – shared for all context words regardless of their position, hence the name "bag-of-words" – for an output prediction of the most likely middle word for a given context. The second models is trained to predict in the opposite direction; given a word, the goal is to maximize the log probability of its surrounding context, i.e. the model learns to *predict the context*, given an individual word. Note that the context words do not need to appear as a consecutive sequence in the corpus, i.e. the total number of context words is fixed by a parameter, but individual words can be *skipped* – referred to by the algorithm's name. The above sketch of the algorithms only describes the "bare-bones" method, omitting for example *negative sampling*, which can roughly be understood as an approximation of the softmax function needed for the conditional probability training objective.

Embedding Advantages In contrast to previous proposals of neural embeddings, the word2vec models were developed with a focus on computational efficiency, based on several optimization steps (e.g. the removal of a hidden layer, negative sampling instead of softmax). As a result, word models could be trained on much larger datasets than before. For example, word2vec contains a pre-trained word model, trained on a 100 billion word subset of the Google News dataset. The popularity of these models can partly be explained by their importance for models of *deep learning*, which owe at least part of their success in recent years to the use of pre-trained embeddings. Neural

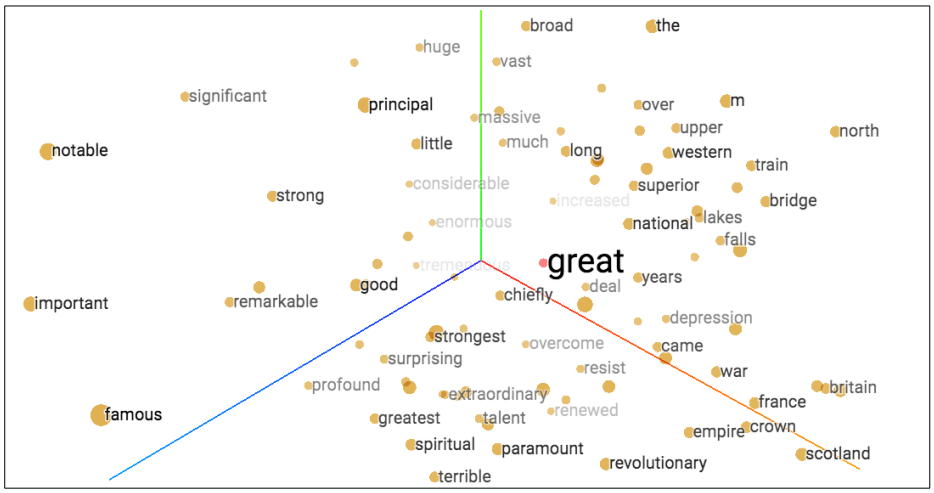

Figure 2.2: word2vec embedding of *great* and neighboring embeddings[6]

embeddings sparked additional interest due to linguistic regularities seen in these models, suggesting an interpretation in terms of semantic features. We will discuss these regularities in the following paragraph.

It is currently an open question whether the representations of neural word models are substantially different from those of the classical word models. A study by Baroni et al. (2014b) concludes that (prediction-based) neural embeddings generally outperform the classical (count-based) methods. This result has been challenged by Levy and Goldberg (2014), which formally establish a close correspondence between the neural models and models factorizing PMI matrices, i.e. claiming that prediction models are equivalent to counting models. Further empirical evidence for this claim was provided in Levy et al. (2015), showing that counting models can achieve performance similar to embeddings by improved hyperparameter choices. In response to these rebuttals, Arora et al. (2015) argue formally and empirically that embedding models, like those of word2vec, are not reducible to matrix factorization if one makes realistic assumptions regarding model dimensionality. Further, these models *impose* the linear structure on the data by their *low-dimensional*, *nonlinear* architecture, suggesting that embedding models are more powerful than classical higher-dimensional approaches.[7]

Linguistic Regularities Neural embeddings gained widespread attention for their representation of relatively complex syntactic and semantic relational information, at odds with linguistic expectations of a syntactically shallow, unsupervised model. Further, these linguistic relations can be extracted from the model by simple linear

[6] 1st/2nd/3rd principal component of 200D word2vec embeddings. Plot generated with TensorFlow Embedding Projector (Smilkov et al., 2016).

[7] While word2vec models contain no hidden layers with nonlinearities, they are nonlinear models due to their training objective, log of (approximated) softmax.

operations over representations, suggesting that embeddings contain non-trivial linear structure.

Several well-known examples of such relations were presented in Mikolov et al. (2013c). Here, the authors observe that a relation-specific, constant *vector offset* exists between the vector representations of word pairs standing in the given relation. Based on this offset, the embedding space can be queried for an answer to *analogy questions* of the form: "word 1 is to word 2 as word 3 is to word 4", where words 1, 2, 3 are given, and word 4 needs to be found in order to answer the question. Expressed in terms of vector space operations, using cosine similarity to measure semantic similarity, the analogy query is given by:

$$\text{argmax}_{v_4} \left(\cos\left(v_4,\ v_3 - v_1 + v_2\right)\right) \tag{2.3}$$

The following are some well-known results derived by this query:

$$v_{\text{king}} - v_{\text{man}} + v_{\text{woman}} \approx v_{\text{queen}}$$
$$v_{\text{Paris}} - v_{\text{France}} + v_{\text{Italy}} \approx v_{\text{Rome}}$$
$$v_{\text{apple}} - v_{\text{apples}} \approx v_{\text{car}} - v_{\text{cars}}$$

The first result can be understood as the vector form of the analogy "*man* is to *woman* as *king* is to *queen*", expressed in terms of basic linear algebraic operations. This example has been interpreted as evidence that embeddings contain structure encoding a *gender* relation or feature. Similarly, the next two examples suggest the model learned to relate countries and their capitals, as well as a *syntactic* relation between singular and plural word forms.

2.2.4 Limitations and Relation to Compositionality

While the relevance of the previous results should not be easily dismissed, it seems doubtful whether purely context-based models can model meaning in its full generality, over arbitrary types of expressions.

Limits of Context Learning An apparent limitation relates to the meaning of sentences, widely believed to require accounting for *compositionality*. A purely distributional, non-compositional approach to sentence meaning is likely to face serious obstacles due to an *unanalyzed* treatment of sentences or phrases. *Sparsity of data* is one such obstacle. Individual words occur abundantly across different contexts, but the frequency of combinations of words declines with the length of the sequence. While short phrases still occur frequently enough to enable some form of meaning derivation by contextual analysis alone, full sentences occur too rarely for a context-based approach aimed at general sentence meaning.

Another obstacle concerns meaning derivation of *logical operators*, e.g. negation. To learn the meaning of (sentential) negation by context analysis alone, its logical meaning would have to be fully expressed in the context distributions of sentences and their negation. Aside from intuitive and formal semantic objections, such an

assumption is countered empirically. Researching distributional patterns of *contrasting word pairs*, Mohammad et al. (2013) show that highly contrasting items (including aspectual negation) occur in very similar contexts – indicating that logical negation cannot be learned by context analysis alone.[8]

Generalization Capacity Finally, another concern relates to the capacity of speakers of *generalization* over meanings of expressions. Human speakers frequently encounter sentences they have never heard before, yet, they easily derive their meaning by knowing the meaning of the individual components (words) and how to combine them systematically (the rules of composition). It seems clear to us that the purely contextual distributional models are capable of *some* generalization, considering empirical evidence such as the structure exposed by the analogy results, or formal results like the *linearizing* effect of embeddings shown by Arora et al. (2015). It seems similarly evident to us however that distributional approaches without any mechanism accounting for compositionality are unlikely to be capable of *full* generalization. Otherwise, the meaning of complex expressions of arbitrary depth and structural complexity would be fully defined by their usage distributions. This notion however is both at odds with linguistic theory and unwarranted from a statistical perspective, considering the aforementioned problem of data sparsity with respect to sentence length.

Combination of Approaches The limitations outlined here seem to motivate compositional extensions of the distributional approaches. At the same time, the success in some aspects of meaning derivation of these models is undeniable. Distributional approaches constitute a major step forward in the derivation of lexical meaning – an area of research comparably underappreciated by symbolic theories of meaning. It also seems to be widely accepted that several breakthrough results achieved by deep learning models in recent years were made possible by including (pre-trained) word representations. The goal therefore seems to be to *combine* distributional models with compositional processing. Under this view, the question: "What are the limitations of distributional models in deriving sentence meaning?", could perhaps be rephrased as: "What are the limitations of current distributional models when used as *input* for further *compositional processing*?"

[8] Kruszewski et al. (2017) confirm that logical negation eludes context analysis, but show that *conversational* negation – negation under a restricted set of alternatives – is susceptible to it.

Chapter 3

Compositional Distributional Semantics

In the previous chapter, we saw an outline of the symbolic approach to compositionality, as well as the areas of success and limitations of context-based models of Section 2.2. Next, we will turn to the literature on *compositional distributional semantics*. In broad terms, the models of this approach extend the contextual methods of distributional semantics by elements that allow for the derivation of phrasal and sentential meaning.

3.1 Two Classes of Compositional Approaches

Before continuing our review, we first propose a (nonexhaustive) classification of recent compositional approaches to distributional semantics. The classification, far from covering the entire field, is merely intended as a lens through which we suggest viewing the literature. It will allow us to focus the discussion on two prominent research directions, and maximize the contrast of their comparison. We suggest then a division along the following classes:

type-based (or *typed*) approaches, with models of this class employing higher-order tensors under an explicit syntax-semantics correspondence

approximative approaches, generally relying on machine learning techniques to indirectly represent and approximate the effects of composition

Examples of the type-based approach are the framework of Baroni et al. (2014a), or the approach outlined in Coecke et al. (2010). We refer to this class of models as *type-based* approaches, because they can be understood as a translation of symbolic semantics to a vector space framework, by defining a correspondence between syntactic categories, semantic types and tensor order.

The second research direction we identify are models that account for compositionality by selectively including insights from linguistic theory, where such insights are interpreted from a machine learning perspective. We suggest that models like Elman (1990) or Socher et al. (2013b) belong to this class. Where the type-based approach

can be seen as explicitly stating the compositional rules, models of the second approach are aimed instead at *learning* the parameters of composition, generally through some form of approximation.[1]

Before reviewing the literature under the view of this classification, we will discuss two studies that do not fall under it, but which had a major impact on the entire field of compositional distributional semantics.

3.2 Types of Composition Functions

This section discusses two landmark articles, Mitchell and Lapata (2008) and Mitchell and Lapata (2010), which frequently serve as a starting point in the literature of compositional methods in distributional semantics. These early investigations of compositionality should be seen in the context of another highly influential study, Turney and Pantel (2010), published at around the same time. Where the studies of Mitchell and Lapata provide an overview of composition methods, the study of Turney and Pantel (2010) covers the entire field of distributional semantics, classifying and discussing models, their methods and underlying assumptions. Up until the point when the embedding models of Section 2.2.3 rose to prominence, the study of Turney and Pantel provided a near-complete overview of the methods of (count-based) distributional semantics.

Shared Model Features Input to all models consists of word vectors constructed by contextual word models, as described in Section 2.2. In the evaluation of Mitchell and Lapata (2010) *composition* is implemented as the application of a function mapping two vectors (the composition input) to another vector (the composition output), in a single composition step. Recursive extensions of the following definitions seem to be straightforward, but were not investigated, since the experimental setup only requires a fixed number of composition steps.

3.2.1 Additive Composition

The general definition of the *additive* class of models is:

$$z = Vx + Wy$$

where V, W are matrices, and composition is given by matrix multiplication.

The simplest instance of this class is given by vector addition:

$$z = x + y$$

which we obtain from the general definition by letting V, W be identity matrices.

The authors note several limitations of this approach, among them, insensitivity to word order, due to commutativity of vector addition. Consider for example that

[1] Our *approximative* class is rather broadly defined, and further subdivision is likely necessary.

recursive application of vector addition would derive identical meanings for the phrases "man bites dog" and "dog bites man". The model's only advantage seems to be its conceptual simplicity and absence of parameters requiring optimization. Yet, despite its simplicity, the method performs comparably well on the task.

Adding scalar weights for each vector results in the *weighted additive* model:

$$z = \alpha\, x + \beta\, y$$

Here, left and right input vectors are scaled (uniformly, for each vector) by parameters α, β which are optimized on a development set. For $\alpha \neq \beta$, this introduces some limited form of syntactic awareness, since left and right input are scaled differently during composition.

Blending Problem While adding weights partially counteract the problem due to commutativity described above, the authors observe another fundamental problem, referred to as *blending* of meaning: Summing, or, equivalently, under cosine evaluation, *averaging* components of the input vectors results in a blend of features in the output. The problem is illustrated by the example phrase "brown cow". Its intended meaning refers to something that is both brown and a cow, i.e. the composed meaning is *more specific* than that of its constituents, 'brown' and 'cow', individually. Additive composition however appears to have the opposite effect: the output vector is a sum of the constituents' components, forming a blend of their properties, thus deriving a *less specific*, "in-between" meaning, according to the authors.

3.2.2 Multiplicative Composition

Multiplication of components is suggested as a solution to the *blending* problem above, leading to a general form of *multiplicative* models:[2]

$$z = \mathcal{V}xy$$

where $\mathcal{V}$ is a 3rd-order tensor mapping the two constituent vectors x, y to output vector z. Composition is consequently a bilinear function of the constituents.

The first example of this class is composition by *element-wise product*:

$$z = x \odot y$$

Here, the i-th component of output z only depends on the i-th components of input x, z, in analogy to vector addition above. However, the interaction between components is now multiplicative instead of additive, as before.

Consider then why multiplication of components can be seen as a solution to the blending problem: Additive interaction does not relate input components directly, i.e. the components of input vectors are added (possibly scaled by a constant factor) to the output vector, independently of any other components. Instead, if composition is

[2] *Multiplicative models* can refer to the entire class, or only the element-wise product model.

defined in terms of component multiplication, values interact directly by their products. For example, consider some component with a high numerical value interacting (by multiplication) with another component with a value (near) zero . Despite its high value, the meaning contribution of the former is limited by its interaction with the latter, effectively barring it from significantly influencing the meaning of the composite expression. If additive composition can be seen as feature *blending*, then this type of interaction can be thought of as feature *filtering*.[3]

Tensor-Based Composition While multiplicative interaction might solve the problem of meaning blending, the simple multiplicative model above is still insensitive to word order. To rectify this problem, the *tensor product* model is introduced:

$$z = x \otimes y$$

While the element-wise product before only allowed interactions between components with identical indices, the full tensor product of two d-dimensional vectors x and y, seen as a $d \times d$ matrix, contains the products of all possible component combinations over the two vectors. This model can be related to earlier proposals, such as Smolensky (1990), where the tensor product acts as a universal operator for variable binding.

Note that the tensor product is not commutative, so composition is no longer insensitive to word order. The downside is an exponential increase in dimensionality (from constituents of $\mathbb{R}^d$ to output of $\mathbb{R}^{d^2}$), and a uniform meaning contribution of all component combinations, due to a lack of parameters.

Two more complex multiplicative models are introduced, both defined on the basis of higher-order tensors. The first, *circular convolution*, performs modular addition along the diagonals of the tensor product in matrix form, compressing the tensor product to a vector of the same size as the input. The second, *dilation*, is defined in terms of a 4th-order tensor, and can be seen as a basis-independent variant of the element-wise product model, where the output consists of input vector y scaled in the direction of input vector x by a constant factor.

3.2.3 Combined Models

Combined models, models using both additive and multiplicative composition, are briefly mentioned in Mitchell and Lapata (2008), defined as:

$$z = \alpha\, x + \beta\, y + \gamma\, x \odot y$$

We can see that this class is simply the summation of the previously defined weighted additive and element-wise product models.

The combined class is of additional interest because it shares some characteristics with the RNTN architecture and its tensor-based composition function. While the combined model above and the RNTN tensor differ in complexity, they both combine

[3] Terminology of Grefenstette and Sadrzadeh (2015).

additive and multiplicative elements for composition, and can be seen as belonging to the same class of models therefore.[4]

3.2.4 Experiments

Mitchell and Lapata (2008) and Mitchell and Lapata (2010) evaluate the models by comparing their predictions to human judgment on a (phrasal or sentential) similarity task. For the model prediction, two phrases are processed compositionally, each phrase yielding a single vector. These vectors are then compared by cosine similarity, yielding a prediction of their *semantic similarity*. The same phrase pairs are judged by humans, and model performance is defined as correlation between the model's prediction and the human rating.

In the first set of experiments, multiplicative models generally outperform additive models across word class categories. In particular, the element-wise product model significantly outperforms (almost) all other models. The only exception is the verb+object category, where the dilation model performs approximately equal. The simplest additive model (vector addition) performs badly across categories, but the weighted additive model comes close to the performance of the multiplicative models, taking second or third places across categories, in an evaluation consisting of seven compositional and two non-compositional models.

In a second set of experiments using a LDA topic model, results are less conclusive, with no individual model outperforming all others across classes. Further, the gap between additive and multiplicative models all but disappears. The weighted additive model continues to perform well in this setting, while performance of the element-wise product suffers.

Significance of Results The results of Mitchell and Lapata (2008, 2010) have occasionally been used as evidence against (syntactically) complex composition, due to the relatively strong performance of the simple element-wise models. Such conclusions seem doubtful however, considering that the task of Mitchell and Lapata (2010) used composition input from syntactically shallow bag of word models, and comparably inexpressive versions of the complex composition operations. While the input vectors are possible structured meaningfully, they likely require additional 'restructuring' during composition. The parameter-free tensor product model, while being non-commutative, has no means of imposing structure on the input, or reshaping the structure to fit the task. The parametrized models possess some means to do so: The weighted additive models can balance the left against the right argument, while dilation adjusts to which degree one input vector scales the meaning of the other. Yet neither of them fully reshapes input on a more fine-grained level, in contrast, for example, to a fully parametrized linear or bilinear map. In conclusion, it seems that the

[4] *Combined models* are also called "mixture models" in the literature. Note however that other authors use 'mixture' to refer to the unstructured composition by pointwise operations.

complex composition method were at a disadvantage compared to the simple methods, given that their *effective* syntactic sensitivity was limited by a lack of parameters.

It seems that if an investigation similar to Mitchell and Lapata (2010) would be undertaken today, more powerful versions of the original composition methods would need to be included in the evaluation. More generally, major advances in the field since the original publication seem to require an update to the study. A step in this direction is Milajevs et al. (2014), applying composition methods similar to those of Mitchell and Lapata (2010) to vectors from two different count-based word models and neural word embeddings. The results are somewhat inconclusive, with neither of the two classes of word models generally outperforming the other. It should be noted critically however that the embeddings where used "off the shelf" from word2vec, while the two count-based models were trained by the authors for the study. While they were not trained directly on the task, they contain parameters (e.g. window size) that could allow some degree of (implicit) task fitting, thus casting doubt on the symmetry of the comparison.

3.3 Type-Based Compositional Models

In this section, we discuss the models of the *type-based* or *typed* approach to compositional distributional semantics, following our classification of Section 3.1. Informally, the approach can be seen as a 'translation' of the symbolic semantic theories of Section 2.1.1 into a vector space setting, through the use of tensors. Following the example of formal semantics, the typed approaches analyze expressions under a parallel assignment of syntactic categories and semantic types, the latter determining the order of the tensor representing an expression of a given type. As a result, composition in the typed models structurally matches composition of symbolic theories. However, given the vector space representation, lexical content can be learned from corpus data, using techniques similar to those of Section 2.2.

3.3.1 Background

In the following discussion, we largely identify the typed approach with the research of two groups: the proposals of Baroni and Zamparelli (2010) and Baroni et al. (2014a) on the one hand, and the framework developed in Clark et al. (2008) and Coecke et al. (2010) on the other. While similar in many respects, in cases where we need to distinguish between the two, we refer to the proposals of the first group as the *Trento framework*, and to the proposals of the second group as the *Oxford framework* – using the names of the universities the authors of the earliest articles were affiliated with at the time of publication.

Related Approaches One of the earliest proposals to use the tensor product as a universal operator for meaning combination was made by Smolensky (1990). While not directly analyzing composition, the proposal addressed the related *binding* problem. Structures (expressions) are decomposed into *structural roles* (similar to the categories

Class	Syntax	Old Type	Space	Tensor
		basic		
S	S	t	$\mathbf{S}$	$\mathbb{R}^{S}$
N	N	$e \rightarrow t$	$\mathbf{N}$	$\mathbb{R}^{N}$
		complex		
ITV	S\N	$e \rightarrow t$	$\mathbf{N} \rightarrow \mathbf{S}$	$\mathbb{R}^{S \times N}$
TV	(S\N)/N	$e \rightarrow (e \rightarrow t)$	$\mathbf{N} \rightarrow (\mathbf{N} \rightarrow \mathbf{S})$	$\mathbb{R}^{S \times N \times N}$
ADJ	N/N	$(e \rightarrow t) \rightarrow (e \rightarrow t)$	$\mathbf{N} \rightarrow \mathbf{N}$	$\mathbb{R}^{N \times N}$
CON	(N\N)/N	$(e \rightarrow t) \rightarrow ((e \rightarrow t) \rightarrow (e \rightarrow t))$	$\mathbf{N} \rightarrow (\mathbf{N} \rightarrow \mathbf{N})$	$\mathbb{R}^{N \times N \times N}$

Table 3.1: Syntax-semantics correspondence (symbolic and tensor-based)[5]

of Section 2.1.1), which are instantiated by *fillers*, the possible values of the role variables. Binding between the two is defined as the tensor product of a role and its filler. The arguments in favor of the tensor product here are similar to those we saw in the context of *structured* composition, in Section 3.2. Other proposals include Aerts and Gabora (2005a,b), approaching semantic modeling with an (adjusted) tensor formalism used in physics, or Van de Cruys et al. (2013), who propose a form of latent factor analysis with tensors to model composition.

3.3.2 Typed Tensor Model

This section presents an adapted version of the syntax-semantic interface of Baroni et al. (2014a), integrating some elements of Maillard et al. (2014).

Syntax-Semantics Interface In Section 2.1.1 we saw that compositionality of meaning in formal semantics is given by a parallel assignment of syntactic structure and semantic content, where syntactic analysis of an expression determines the object or function type representing it, which are constructed parallelly through lambda expressions. From Appendix A.0.3 we recall that tensors allow us to represent general functions from vector spaces to vector spaces, in way that resembles the functions of typed lambda calculus.

This seems to suggest that we can translate functions of symbolic semantics to multilinear maps, represented by tensors of appropriate shapes, by associating basic types with basic vector spaces, and complex types with functions over these spaces.[6] *Functional application*, implemented by the reduction operations of lambda calculus in

[5] Adapted version of the syntax-semantics interface of (Baroni et al., 2014a, Section 3.6).

[6] Representation of arbitrary lambda expressions by multilinear maps, $V_1 \times \ldots \times V_n \rightarrow W$, including functions from complex types to complex types, e.g. predicative adjectives of type $((e \rightarrow t) \rightarrow (e \rightarrow t))$, is guaranteed *formally* by the axioms of monoidal categories (Coecke et al., 2010, Section 3.1), or given *practically* by reshaping of tensors before composition by general matrix multiplication or matrix-vector multiplication (Baroni et al., 2014a, Section 3.3).

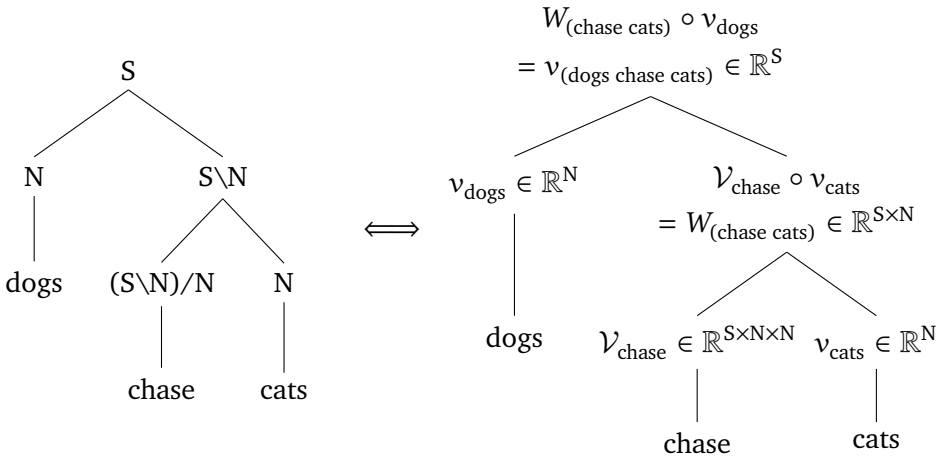

Figure 3.1: Derivation under the syntax-semantics interface of Table 3.1

symbolic theories, corresponds to tensor contraction in the vector space models, or *matrix multiplication* for matricized tensors.

This idea is made formally precise in the category-theoretic framework of Coecke et al. (2010), by establishing a correspondence between syntax, given by a pregroup grammar P (similar to categorial grammars) and semantics, given by a vector (tensor) space **FVect**. Expressions of natural language are then mapped to elements of an object of the category $\mathbf{FVect} \times P$, i.e. expressions are mapped to an object representing both its syntactic structure (essentially, a parse of the expression), and the meaning of the expression, under this parse, expressed in the vector space. It should be easy to see that this mapping resembles the one seen in the symbolic approach, i.e. assigning to each expression a category and a type. Formalizing this intuition is clearly more challenging, and is one of the achievements of Coecke et al. (2010).

Tensor-Based Derivation Table 3.1 is the updated version of Table 2.1, relating language expressions, syntactic categories (based on a simplistic categorial grammar), the previous interpretation of these expressions as functional types, and the new vector space semantics, i.e. the tensor-based interpretation of the current system. Note that the type-based approach establishes a new correspondence between expressions, syntactic categories and meaning, but that there is not necessarily a full correspondence between old and the new meanings. For example, basic types in the symbolic theories were individuals of type e, and truth values of type t. We will consider next the reasons for another choice for the basic categories in a vector space setting, i.e. why common nouns are the *primitives* of the new system, rather than being complex types like in the symbolic approach.

Fig. 3.1 is a derivation of example sentence “dogs chase cats”, analogously to the symbolically defined derivation in Fig. 2.1. As before, the phrase structure rules define one (of possibly several) syntactic derivations of the sentence, while the category-type correspondence of Table 3.1 specifies the interpretation of an expression of a certain

category. The composition of expressions, marked here by enclosing the tensors in square brackets, consists in applying the *function* tensor to its *argument* tensors, where the application process is specified by semantic rules defined in analogy to the syntactic rules. Functional application itself can be implemented in different ways, e.g. by tensor contraction, matrix multiplication, or matrix-vector multiplication, depending on the form the tensors take in a particular model.

3.3.3 Basic Spaces of Models

Similar to the *basic types* of the symbolic approach, certain vector spaces in the typed approach are *basic* (or atomic) spaces, as seen in the syntax-semantics interface of Table 3.1.

Sentence Spaces Note first that, in principle, one can follow the symbolic theories and define a two-dimensional "truth value" space. A suggestion to this effect is worked out in Coecke et al. (2010). However, by doing so, the models lose one of their most interesting properties: the ability to compare sentences on a more fine-grained level than by simple comparison of truth values, i.e. measuring *similarity* of sentences e.g. by spatial distance. As an example, consider that in a binary sentence space, the similarity between sentences "The sun is shining" and "It is rather warm today" is the same as the similarity between "The sun is shining" and "There is no greatest integer" (if the weather statements are true in the model). Since this is usually not a desirable feature, the sentence space is mostly set to be a higher-dimensional space. A proposal to this effect can be seen in Grefenstette et al. (2011). Here, sentences containing *transitive* verbs are represented in space $\mathbf{N}\otimes\mathbf{N}$, for $\mathbf{N}$ the space of nouns – also the space of arguments of intransitive and transitive verbs. Sentences based on *intransitive* verbs are instead represented in the simple noun space $\mathbf{N}$ itself. This decision is based on the reasoning that sentences with transitive verb *relate* basic concepts (features) across the arguments of the transitive verbs, in contrast to simple (unary) *predication* over an argument in intransitive verb sentences. Then, the space for transitive verb sentences should be able to represent this additional structural complexity, i.e. defining it as the tensor product of the individual noun spaces.

Learning Space While the choice of the sentence space is effectively a choice of the model *output space*, determining the second atomic space can be seen as a choice of the model *input space*, i.e. the space in which *learning* takes place primarily. Often, this will be the space of *nouns*, since the word class carries substantial lexical meaning, and is comparably 'simple', being representable by vectors.

Note that learning the content of higher-order objects is in principle, and practice, also possible. For example, the proposal of Grefenstette et al. (2013) extends the linear regression method of Baroni and Zamparelli (2010), which learns the content of vectors like nouns, to a *multi-step regression* method that can learn the word content of higher-order objects like transitive verbs. But even then, noun vectors are the *primary*

objects of learning: in a first step, noun content is derived, which then becomes the input to the next regression step that learns the content of transitive verbs.

3.3.4 Limitations of Typed Approaches

The type-based approach is characterized by a high formal and conceptual clarity that is possibly unique among the compositional model. At the same time, we can think of several problems affecting it.

Dimensionality Problems The objection relates to the *problem of dimensionality* mentioned in Appendix A.0.3, i.e. the intractable nature of higher-order tensors in high-dimensional tasks. Recall that parameter size grows exponentially with the order of the tensor, and that full-rank third-order tensors already are hardly practical in high-dimensional spaces. Note then that the type-based tensors go well beyond third-order: Ditransitive verbs, like 'give', are fourth-order tensors, while some prepositions, like 'with' in "$\boldsymbol{x}$ eats $\boldsymbol{y}$ with a $\boldsymbol{z}$", are fifth-order tensors. Even higher typed expressions exist, and further complications might arise if the typed approach would incorporate *type shifting*. The latter is a standard technique in formal semantics, where in cases of type mismatch, the type of an expression can be lifted (raised), possibly further increasing tensor order.

Related to parameter explosion is the problem of *sparsity of data*. If learning the word content of tensors would proceed without deconstructing expressions based on them, examples of these expressions would be too infrequent for learning to succeed. Further, their meaning would not be derived correctly – for example, "drinking water" and "drinking wine" would not relate through their common verb. If instead learning methods break apart such expressions, as in the multi-step regression of Grefenstette et al. (2013), data sparsity can be avoided, but higher computational cost is incurred by the iterated learning methods.

The approach can therefore only be made to work if *tensor approximation* methods are employed, as seen in Appendix A.0.3. Proposals to this effect exist: Grefenstette et al. (2013) and Polajnar and Clark (2014) use low-rank third-order tensors for transitive verbs. Nonetheless, the dimensionality problem is likely to persist: Third-order tensors can usually be made tractable with dimensionality reduction methods, but tensors of very high order often remain intractable even then.[7] It seems overly optimistic therefore to expect that tensors of order five or above will become practically usable in the near future, given the available dimensionality reduction or approximation techniques.

Conceptual Limitations Another point of criticism of the type-based approach stems from a primarily linguistic concern. The rigorously defined translation of symbolic theories into vector spaces can both be seen as a major advantage, and as a

[7] While some tensors of very high order can be made tractable due to their structural properties, in general, tensors of such order tend to be intractable (Hackbusch and Kühn, 2009).

burden in the task of creating a model of meaning. Leaving the practical concerns of the previous paragraph aside for now, it might well be that the typed approach has an initial advantage over other compositional distributional methods, given that it builds upon years of research in the symbolic field.

At the same time, this also suggests that any limitations applying to the symbolic approach are *inherited* by these models. It seems that the typed approach is bound to whatever progress (or lack thereof) is made in formal semantics, since the autonomous choices of the approach are mainly related to the implementation of an external theory in a new setting.

A related objection stems from *linguistic inquisitiveness*. Delegating the task of compositional analysis proper to another field can be seen as a lost opportunity from a linguistic point of view. Rather than only investigating compositional semantics in symbolic frameworks, any attempt to derive *additional* knowledge, for example by the analysis of semantic vector space models, should be welcomed. The class of approximative models about to be described could be a promising tool to achieve such a goal. In order to achieve it, however, models need to be understood sufficiently, both on the level of their architectures (which generally seems to be true), and on the level of model instances resulting from training (which seems less generally true). This latter process, the analysis of the patterns learned by models during training, could however be an important tool to gain new insights into semantic processing, and could add a new perspective to the field of formal semantics itself.

3.4 Approximative Compositional Models

In the final section of our literature and model overview, we focus on the class of approximative compositional models, understood here as architectures accounting for compositionality not by explicitly specifying compositional rules, as seen in Section 3.3, but by indirect means, generally aiming to *approximate* composition.

Our overview is focused on neural architectures, although other model classes could be included here as well. We motivate this restriction by the following considerations: The models at the core of our analysis are neural networks, and we limit our discussion to the close relatives of these models. Further, although compositional processing is not exclusively the domain of neural networks, recent formal results suggest that functional composition and neural network architectures are intimately related in the context of *deep learning* architectures Mhaskar et al. (2016); Montufar et al. (2014); Pascanu et al. (2013).

3.4.1 Recurrent and Recursive Neural Networks

The following models account for compositionality, and in particular: for arbitrary sentence length, by recursive definitions.

Recursion One way to think of compositional semantic analysis is to describe its goal as the derivation of meaning of complex expressions, and in particular, of full sentences.

Semantic meaning, in full generality, is either *expressed* by complex expressions, or, in word-level models, usually *depends* on the meaning of complex expressions. As noted in Section 2.2, non-compositional models of distributional semantics face the challenge of deriving this general meaning without the ability to deconstruct complex expressions. This suggests as a possible minimal requirement for compositional models the ability to break down expressions into smaller components, such that the model output given a complex expression is defined as a function of these smaller components. In other words, such models should be based on a *recursively* defined derivation process.

Recurrent Networks A relatively simple, but powerful architecture satisfying this requirement is the *recurrent neural network* (RNN), proposed in Elman (1990). It is also known as Elman network, and in some cases referred to as the *simple* RNN to distinguish it within a larger class of related models, based on similar recurrent (recursive) definitions.

The function learned by an RNN model is defined as a recursion over an input sequence of vectors $x_1, \ldots, x_n$. This sequence of input vectors can be chosen quite generally, for example, as vectors representing the *words* of a sentence, or, breaking input down further, as the *characters* of a sentence.

At input step x_i of input sequence $\boldsymbol{x}$, the output z_i of an RNN is given by:

$$z_i = f(W^{hh} z_{i-1} + W^{hx} x_i \; + \; b^h) \tag{3.1}$$

where $z_i \in \mathbb{R}^H$, and H is the dimension of the hidden layer, $x_i \in \mathbb{R}^X$, $W^{hh} \in \mathbb{R}^{H \times H}$ is the matrix of hidden-to-hidden connections, $W^{hx} \in \mathbb{R}^{H \times X}$ the matrix of input-to-hidden connections, $b^h \in \mathbb{R}^H$ is a bias vector, and f an element-wise nonlinear function, called the *nonlinearity* of a layer.[8] Frequent choices for this nonlinearity are the sigmoid function, tanh, or a rectified linear unit (ReLU). Note that many models place an additional layer between the hidden layer at step i and model output at i. However, this step can be seen as external to the core of the recursively defined process of Eq. (3.1).

For finite input sequences, the general recursive definition can be replaced by an *unfolded* version of the model instance. Given an input sequence of length n, the network can be seen as a chain of $n + 1$ hidden layers or states, where each state has two outgoing connections (one to the next hidden layer, one being the output at the current step), and two incoming connections (one being the output of the previous hidden layer, one for the input at the current step). The i-th state of this chain is the model representation of the input sequence up to and including input element i, given by summing the linear combination of the i-th input vector, the linear combination of the $(i-1)$-th state or output, and the bias vector, then applying the element-wise nonlinearity f to the resulting vector.

Using simple RNN models, the work of Mikolov (2012) constitutes an early, influential exploration of RNNs applied to language tasks. In general, however, most

[8] Here, and in most of this section, we follow the concise model definitions of Goldberg (2016), sometimes with minor notational adjustments.

notable results produced by RNN architectures in recent years were in fact *extensions* of the architecture, often produced by the highly successful class of LSTM models, which will be discussed later in this section.

Recursive Networks A notable *structural* extension of the simple RNN architecture is the *recursive neural network*, referred to as tRNN in this thesis. The tRNN in the form described here was introduced in Socher et al. (2010), based on the proposals of Pollack (1990) and Goller and Küchler (1996). We will describe this architecture in detail in Chapter 4, and for now will only briefly remark how it differs from the simple RNN. The tRNN can be seen as a generalization of RNNs in terms of input structure, in the following sense: While the input sequence of a simple RNN is unstructured, and composition invariably proceeds in one direction, the tRNN architecture allows for the compositional process to be structured by syntactic analysis of the input. In practice, this syntactic analysis is usually provided externally, by providing the model with a parse tree for a given input sentence.

A variant of the tRNN is the *recursive matrix-vector model* (MV-RNN), introduced in Socher et al. (2012). Composition order is the same as in a tRNN, given by tree-structured inputs. However, the parameters of the MV-RNN composition function are no longer purely global (or *task-specific*), but bound to individual words as well. Each word is assigned both a vector of size $\mathbb{R}^n$ and a matrix of size $\mathbb{R}^{n \times n}$. Composition of two nodes in a tree is performed by applying the matrix of one node to the vector of the other, resulting in two vectors of size $\mathbb{R}^n$. These vectors are concatenated and mapped back to a single vector of size $\mathbb{R}^n$ by a global weight matrix. We can see then that composition in the MV-RNN allows for composition arguments to modify each other, similar to the mechanism of the typed approach of Section 3.3.

The MV-RNN model can be seen as a *hybrid* architecture, combining elements of the approximative approach (e.g. using nonlinearities) and of the type-based approach (e.g. words modifying each other), and in particular resembles the adjective-noun model of Baroni and Zamparelli (2010). Yet, certain key differences suggest the model is not an instance of the typed approach. The MV-RNN treats words *symmetrically* with respect to composition – regardless of its class, each word is represented by a matrix and vector, allowing both words to modify each other. This seems to be in marked contrast to the typed approach, where the representation of a word depended on its syntactic *category*, determining whether words act as functors or arguments.

3.4.2 Memory Mechanisms

A major challenge when training *deep* neural networks – networks consisting of many stacked hidden layers – is the *vanishing gradient problem*. This cause for this problem relates to the training algorithm of networks, which passes information (gradients) down the network as a chain of *products*. Since individual terms of this chain are often small, their product tends to decrease with the length of the chain, to the point of vanishing. As a result, lower layers of a deep network only receive a greatly diminished learning signal, thus negatively affecting learning success.

Long Short-Term Memory The seminal work of Hochreiter and Schmidhuber (1997) presented a solution to the problem, by introducing the *long short-term memory architecture* (LSTM). We only describe the general idea behind the approach here, and refer the reader to Graves (2008) for a technical explanation of the mechanisms.

The LSTM architecture, based on the (simple) RNN model of Eq. (3.1), adds *memory cells* and *(control) gates*, allowing a higher degree of retainment of gradient information across network layers. Memory cells are vectors retaining past gradient information, where access to these cells is controlled by three types of gates (*input*, *output*, and *forget* gates). Intuitively, these gates can be seen as vector space versions of logic gates, interacting with the components of the memory cells by pointwise multiplication with values near 0 or 1, i.e. *soft* boolean values. During training, stored gradient information and the gates interact to *preserve* old and *select* new gradient information that will be passed downwards in the network. This mechanism leads to major improvements in training effectiveness of deep networks, and most major results in recent years by models of the RNN class were produced by LSTMs, or further model extensions of the RNN architecture, with added LSTM gates.

RNN models using LSTM gates generated major results on several language tasks. In an early study of LSTMs and language processing, Gers and Schmidhuber (2001) showed that their model can learn simple context-free and context-sensitive languages, e.g. strings of the form $a^n b^n$. In Graves (2013), LSTM models are used to generate novel sentences after being trained on Wikipedia data, and learn to produce sentences in realistic script (i.e. the model learned *handwriting*).

Replacement for Tree-Structured Networks? While the models we use in our own experiments do not use LSTM gates, they are related by a possible overlap in the function they serve. LSTM models have been suggested as a *replacement* for the explicit syntactic information provided to the tRNN models, due to their ability to store (training) information across the processing of long input sequences, such as sentences. While syntactic information that is given to the tree-structured networks *explicitly*, LSTM models possibly can rely on *implicit* syntactic information through their storage mechanism.

The question whether syntactically guided processing is a necessary feature of distributional models is not fully answered yet. Recently, tRNN models lost much of the popularity they gained in the years before, and most of the currently popular models (such as convolutional networks) do not rely on any explicit syntactic information. At the same time, there are also recent state-of-the-art results that were achieved by models using syntactic information, in combination with other processing mechanisms Kokkinos and Potamianos (2017).

Note also that it is possible to combine the two architectures, i.e. enrich tree-structured networks with a memory mechanism. The proposals of Le and Zuidema (2015) and Tai et al. (2015) extend the tRNN architectures with LSTM gates, aiming to improve training efficiency of deep networks, and ideally, to yield better results in modeling long distance dependencies.

3.4.3 Tensor-Based Architectures

In the following, we introduce the family of multiplicative, or tensor-based architectures. Included in this class is the tRNTN model, which is the focus of our own investigation in the next chapters.

Arity of Functions The final aspect of composition and composition approximation to be covered in detail here relates to the representation of functions of several arguments, i.e. to the *arity* of functions. Correspondingly, we discuss here architectures based on higher-order tensors, or models that define some form of multiplicative interaction between input arguments.

Recall that in Appendix A.0.3, we pointed out the relation between (vector space) functions of multiple arguments and higher-order tensors, by taking the perspective on tensors as *multilinear functions*. Next, in our discussion of the study of Mitchell and Lapata (2010), we saw the recurring theme of *additive* and *multiplicative* composition and argument interaction. The latter type of interaction corresponded to composition by the tensor product or the simpler element-wise product, or was given by higher-order tensors.

Recursive Tensor Network We begin our discussion of tensor-based models with the *recursive neural tensor network*, proposed in Socher et al. (2013b), and abbreviated here as tRNTN. Extending the previously mentioned tRNN architecture, composition in an tRNTN is shaped as well by tree-structured input. In addition, the composition layers of the tRNN contain a full-rank third-order tensor term. The model architecture and training algorithms will be described in detail in Chapter 4, and their performance analyzed in Chapters 5 and 6.

The trained composition function of the tRNTN is defined as:

$$z = f(\mathcal{V}_{\mathsf{m}}(x \otimes y)_{\mathsf{v}} + W[x\,;y] + b) \tag{3.2}$$

Here, $z \in \mathbb{R}^d$ is the layer output for input $x, y \in \mathbb{R}^d$, $b \in \mathbb{R}^d$ the bias, $\mathcal{V}_{\mathsf{m}}$ the matricized third-order tensor $\mathcal{V} \in \mathbb{R}^{d\times(d\times d)}$, $(x \otimes y)_{\mathsf{v}}$ the vectorized tensor product of x, y, $[x\,;y]$ the concatenation of vectors x and y, and f the layer nonlinearity.

Adding a tensor term is motivated in Socher (2014) by the observation that in standard RNN or tRNN models, input vectors only interact *indirectly*, through the parameters of the composition function, which are reshaped during training by signals passed through the layer nonlinearity. Instead, the author argues, *direct* interaction of the input seems preferable, and can be implemented by adding a higher-order tensor term to the layer.

We will next describe two different multiplicative extensions of the (simple) RNN architecture, both of which are closely related to the tRNTN.

Multiplicative Integration Wu et al. (2016) proposes the generally defined *multiplicative integration* (MI) design, as an extension that can be implemented in various network architectures. In the terminology of Wu et al., the sum of the linear state

combinations in a standard RNN, i.e. $W^{hh}z_{i-1} + W^{hx}x_i$ in Eq. (3.1), is the *additive building block* of the architecture. The MI design replaces the additive building block by pointwise multiplication of the terms, resulting in a *simple* version of the MI design.

Adding vector terms to *gate* (scale) each product independently, they derive the *general* MI equation, where the output for x_i is given by:

$$z_i = f(\alpha \odot W^{hh}z_{i-1} \odot W^{hx}x_i + \beta_1 \odot W^{hh}z_{i-1} + \beta_2 \odot W^{hx}x_i + b^h) \tag{3.3}$$

The adjustment can be understood as implementing a mutual *gating* behavior, i.e. the first summand of Eq. (3.3) allows the current input term to enable, disable, or generally, *scale* the impact of the state history term, and vice versa. Note that the original RNN function is *contained* in Eq. (3.3), and that we can recover it by setting α to be the zero vector, and β_1, β_2 one vectors. Analyzing the gradient calculations of MI-adjusted architectures, Wu et al. (2016) conclude that the newly derived composition functions have desirable convergence properties compared to those of the unadjusted models.

The empirical evaluation of Wu et al. (2016) begins by implementing MI-adjusted variants of a simple RNN and an LSTM, resulting in the *MI-RNN* and *MI-LSTM* architectures. These models are tested on several tasks, including semantic relatedness and paraphrase detection. The authors find that the MI models perform well across the board, easily outperforming the baseline models, and coming close to the best reported performance on some tasks.

Multiplicative RNN Sutskever et al. (2011) propose a different multiplicative extension of the basic RNN architecture, introducing the *tensor RNN*, and the related *multiplicative RNN* (mRNN). The tensor RNN is based on the second-order RNN architecture of Pollack (1991), while the definition of the mRNN additionally relates to the proposal of Taylor and Hinton (2009), in which restricted Boltzmann machines were extended by factorized third-order tensors.

The mRNN and the MI-RNN models of the previous passage are closely related. Essentially, the extensions only differ in their choice of the multiplicative operation added to the state transition function. Note that both proposals, aside from adding a multiplicative component, make no changes to the basic RNN architecture, i.e. input/output dimensionality and syntactic structure of composition remain unchanged.

The output of the *tensor RNN* for input x_i is defined as:[9]

$$z_i = f\left(\mathcal{V}^{hh}_{(x_i)}z_{i-1} + W^{hx}x_i + b^h\right) \tag{3.4}$$

where $\mathcal{V}^{hh}_{(x_i)}$ is a full-rank third-order tensor. If the input $x_i \in \mathbb{R}^M$ uses 1-of-M encoding, we can think of $\mathcal{V}^{hh}_{(x_i)}$ as assigning a different weight matrix to the character encoded by x_i, for a total of M characters.

[9] We adjust the notation of (Sutskever et al., 2011, Equation 3) to unify definitions across Eqs. (3.1) to (3.4).

As discussed in Appendix A.0.3 and Section 3.3, full-rank tensors are generally intractable, and Sutskever et al. (2011) factorize tensor $\mathcal{V}^{hh}_{(x_i)}$ by:

$$^{\star}\mathcal{V}^{hh}_{(x_i)} = V^{hf} \operatorname{diag}\left(V^{fx} x_i\right) V^{fh} \tag{3.5}$$

Here, V^{hf}, V^{fh} are input independent, dense matrices, $\operatorname{diag}(V^{fx} x_i)$ is an input dependent, sparse matrix, i.e. the transformation of vector $V^{fx} x_i$ into a diagonal matrix. The value of dimension f determines the parameter size of the model and the expressiveness of the factorization. Replacing $\mathcal{V}^{hh}_{(x_i)}$ by $^{\star}\mathcal{V}^{hh}_{(x_i)}$ in Eq. (3.4) yields the definition of the mRNN.

Context Conjunction Sutskever et al. motivate their mRNN proposal by the following argument, describing how a character-level network processes linguistic structure: Consider a character sequence 'ing', which could be a likely continuation of previously encountered character sequences 'fix' or 'break', but not of a sequence 'multipl', for example. We assume next that the model successfully learned hidden representations for 'fix' and 'break' such that both representations encode information about 'ing' being a frequently encountered continuation. We would then ideally like to represent the *conjunction* of a history, e.g. 'fix' or 'break', with an input character, 'i', such that in case their conjunction is true, a continuation of the sequence by character 'n' becomes more likely.

So far, the argument suggests that *context conjunction* seems to be a useful feature. In order to represent (fully general) logical conjunction – a binary connective, or *two-place function* – the model should be capable of representing multiplicative interaction of two arguments, which is naturally expressed by a *third-order tensor*. We can informally think of this tensor as a bilinear function from a *history* space and *current input* space to a space of *likely continuations*. This function encodes, for example, that the conjunction of some history and some input character will likely be continued by a particular character sequence. This *likely continuation* corresponds to the vector resulting from composition of the hidden-to-hidden matrix *selected* by the current input character, with the input vector itself, $\mathcal{V}^{hh}_{(x_i)} z_{i-1}$. The resulting vector is one of the terms that, in combination, determine the current hidden state, thereby influencing the *next* state and the predictions licensed by it.

Evaluation and Extension Sutskever et al. employ the mRNN architecture as a generative model: After initialization by arbitrary starting values, full prediction sequences are generated by sampling from the model's output distribution, then feeding the result back to the model as input for the next prediction step. Among other datasets, the authors train the model on English Wikipedia articles. While the resulting sentences are nontrivially structured and (mostly) grammatical, they exhibit a clear lack of semantic coherence above the word level, suggesting that the model's representation of sentence meaning is rudimentary.

Irsoy and Cardie (2014) compare the mRNN architecture to *matrix space models* – approximately corresponding to the *typed* model class of Section 3.3. The authors

conclude that the mRNN can be approximately seen as a matrix space model with *parameter sharing*: In matrix space models, a matrix or vector is explicitly assigned to every word. In contrast, mRNN models 'extract' such matrices and vectors from the input words by a tensor, and require fewer parameters than matrix space models in computing a similar function.

In Krause et al. (2016), the mRNN model is *combined* with LSTM gates, and evaluated against regular LSTM models. The authors note that their combined *multiplicative LSTM* (mLSTM) architecture contains two multiplicative components, both from the mRNN and the LSTM mechanisms. However, the authors remark, these components serve complementary purposes: multiplicative argument interaction and gating of gradient histories, and a combination therefore justifiable.

3.4.4 Conclusions

In the preceding two chapters, we reviewed some of the literature relevant for our investigation. We noted there that symbolic models provide a solid foundation of the compositional aspects of natural language semantics, while such a foundation is arguably lacking in the purely distributional models. Further, we saw that higher-order tensors can potentially be used to integrate some of the advantages of the symbolic approach into distributional models – either directly, or "by approximation". Recursive neural networks are one example of a distributional model architecture motivated by compositional insights. An extension of this architecture incorporates higher-order tensors to compose expressions, and seems to satisfy many of our requirements for an *compositional* distributional model. The following chapter will introduce these two models in greater detail then.

Chapter 4

The Recursive Neural Tensor Network

4.1 Feedforward Neural Networks

Artificial neural networks or simply *neural networks* are a class of machine learning models inspired by biological systems of cognition like the human brain. One of the earliest formal models of such systems was the *logical calculus of nervous activity* of McCulloch and Pitts (1943), forming the basis of modern neural network architectures. Closely related is the *perceptron* classifier of Rosenblatt (1958), becoming the first computationally implemented model of neural networks in the late 1950s.

Feedforward Networks Perceptrons form the basis of modern *feedforward neural networks* (FFNN), also known as *multilayer perceptrons* (MLP). Inspired by neural activity of biological brains, the computational units of the model are termed *neurons*. Grouping neurons by *layers*, one can define connections between units of different layers. In a fully connected network, a single neuron of layer n receives the output values of all neurons of layer $n-1$ as its input. The neuron calculates a weighted sum over this input, then passes the result through an *activation* function for the output of the unit. In models consisting of multiple layers, we distinguish between the *input* layer, the *output* layer, and layers in between, called the *hidden* layers of the network.

Fig. 4.1 is a graphical representation of such a layered arrangement of units. We use superscript for layer indices, and subscript to index units within a layer. Each single neuron is represented here by two separate nodes: A node marked by $f(\cdot)$, representing the neuron's calculation of a weighted sum and application of an activation function, followed by an indexed node, denoting the (scalar) output of the unit, used as input for the next layer. Arrows connecting units across layers are associated with a scalar value, specifying the *weight* of a value in the calculation of the weighted sum by the unit receiving the input.

For example, h_2^1 is the output of the second neuron of the first hidden layer, given by passing the weighted sum: $w_{2,1}^1 x_1 + w_{2,2}^1 x_2 + w_{2,3}^1 x_3$ through the activation function

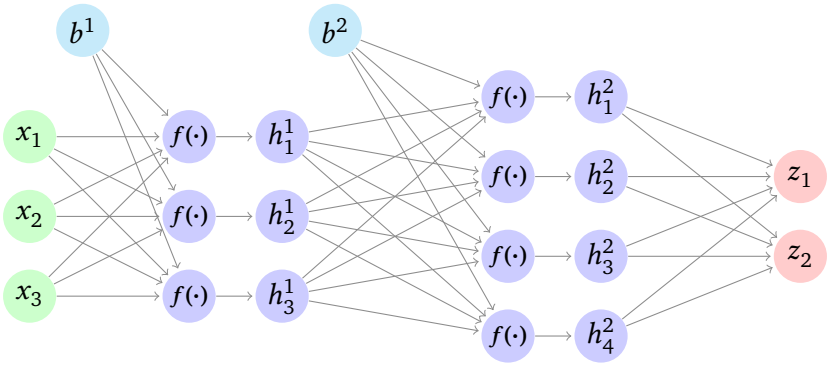

Figure 4.1: Simple feedforward neural network with two hidden layers

f. Here, $w_{2,3}^1$ is the weight assigned to the output of the third unit of the input layer during calculation of the weighted sum of unit h_2^1. Fig. 4.1 also contains two *bias* nodes, b^1 and b^2. These units receive no input, and produce a constant output that is used as additional (weighted) input by all other units of the layer.

Vectorized Notation We can think of the network of neurons in Fig. 4.1 as defining a complex function φ from multiple input values to multiple output values. Doing so suggests a more concise notation of the network, describing φ as a function from vectors to vectors. We can represent each layer as a vector, with components given by the output values of units, and connection weights between a layer of m units and a layer of n as a matrix of shape $n \times m$. Fig. 4.1 can then be compactly represented by the following equations for $\varphi^{\text{nn2}}(x)$, defining a feedforward neural network with two hidden layers:

$$\begin{aligned} \varphi^{\text{nn2}}(x) &= z \\ h^1 &= f(W^1 x + b^1) \\ h^2 &= f(W^2 h^1 + b^2) \\ z &= W^3 h^2 \end{aligned} \tag{4.1}$$

Model input x, bias b^1 and the output of the first hidden layer h^1 are 3-dimensional vectors, h^2 and b^2 are 4-dimensional vectors, and model output z is a 2-dimensional vector. The 3×3 matrix W^1 contains the weights between input layer and first hidden layer, 4×3 matrix W^2 those between first and second hidden layer, and 2×4 matrix W^3 the weights between the second hidden layer and output layer.

Representing unit output as vectors, and connection weights between layers as a matrix, we can equivalently describe the weighted sums of neurons by *matrix multiplication*. For vector h_i representing the output of the i-th layer of a network, and matrix W the weights between the i-th and $(i+1)$-th layer, the transition from layer i to $i+1$ is given by $Wh_i = z$, where we call z the *net input* of layer $i+1$.

As noted previously in Appendix A, matrices corresponds to *linear maps*, and we can say that the parameters of a (fully connected) network layer given by W

compute a linear transformation of layer input x. The set of all adjustable (trainable) parameters of a network will be referred to as θ. Given the network layout of function φ, parameters θ, and input x, we can numerically compute output $z = \varphi(\theta, x)$.

4.2 Activation and Cost Functions

In this section we describe the *nonparametric* functions of neural networks. While the layer transformations defined by matrices are *parametric*, i.e. defined by weights that are adjusted during training, these functions generally do not change.

4.2.1 Activation Functions

Various activation functions can be used in place of f in Eq. (4.1) or other neural network architectures. Activation functions are applied *element-wise*, i.e. a function defined over scalars is applied separately to each component of a vector.

Sigmoid functions are frequently employed as activation functions. One example of a sigmoid is the *logistic function*:

$$f(x) = \frac{1}{1 + e^{-x}}$$

with values bounded in $[0, 1]$, and forming an approximate 'S' shape, i.e. a continuous version of the classical step function used in early network models.

Another common sigmoid activation function is the hyperbolic tangent, or *tanh*:

$$f(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

The function range of tanh is $[-1, 1]$, with a graph of a similar shape to that of the logistic function.

More recently, *rectified linear units* (ReLU) have been employed successfully in neural networks, in particular, in deep network layouts, due to their non-saturating behavior. The simple ReLU function is defined as:

$$f(x) = \max(0, x)$$

In contrast to the 'S' shape of sigmoid functions, the graph of a ReLU function is given by two straight lines connecting at an angle at the origin. Since they consist of linear 'pieces', rectifiers belong to the class of *piecewise linear functions*.

The simple ReLU maps all negative input values to zero, which is not always desirable. The *leaky ReLU* scales down negative values by a constant factor instead:

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0.01x & \text{otherwise} \end{cases}$$

Nonlinearity The activation functions above are all examples of *nonlinear* functions, and the terms *activation function* and *nonlinearity* are often used interchangeably. Even so, network layers could be defined to employ *linear* activation functions, the simplest example being the *identity* function. Doing so would however change which functions such networks can represent: The composition of several linear maps is another linear map, and networks using linear activation functions in all layers would in total compute a linear function. As a consequence, these networks would be incapable of representing nonlinear data, and could not solve problems defined on such nonlinear structure. The classical example of a problem that is not linearly separable is the representation of the XOR function, which was shown to be unsolvable by a single layer perceptron in Minsky and Papert (1969).[1]

Layers with nonlinear activation functions on the other hand are able to *linearize* data that was not linearly separable originally. The composition of such layers, i.e. the function computed by the complete network, φ, is nonlinear as well, and able to represent problems that a linear function cannot. In particular, for a large class of nonlinear activation functions, formal results show that all functions of practical interest to us can be (approximately) expressed by a linear combination of their output. Two of the earliest and most well-known of these results, established independently by Cybenko (1989) and Hornik et al. (1989), are commonly referred to as *universal approximation* theorems for neural networks.

Softmax A special case of a nonparametric function of a network layer is the *softmax* function. Unlike the nonlinearities of the hidden layers, the purpose of this function is not linearization of data, but to represent model output as a *probability distribution* For vector $x \in \mathbb{R}^k$, the softmax of x is:

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{k} e^{j}}$$

The components of softmax(x) sum to 1, and we can interpret the resulting k-dimensional vector as a discrete probability distribution of k classes. Used in the last layer of a network trained on a classification task, we can then interpret model output as a probabilistic prediction over k possible outcomes.

4.2.2 Cost Functions

In order to train neural network models, i.e. adjust model parameters θ, we need to define a *training objective*, according to which the current values of θ are evaluated. In supervised learning, the training objective generally consists of minimizing a *cost* value, also called *loss* or *error*. This scalar value is given by the *cost function* of a model, which is defined over the output of the model and a target solution.

[1] The SLP is unable to solve the XOR problem because it lacks a *hidden layer*. Both a nonlinear activation function and one or more hidden layers are needed for a solution.

A simple cost function used for binary classification tasks is *hinge loss*:

$$C(z,t) = \max(0, 1 - z \cdot t)$$

Here, z is the one-dimensional model output, i.e. the model prediction which of two classes an input example belongs to, and t is the *target* label, with classes represented as elements of $\{-1, 1\}$. If prediction and target share the same sign (i.e. z and t are both negative or both positive), and $|z| \geq 1$, hinge loss is 0, i.e. the model has learned the intended classification. Otherwise, loss will be greater than 0, and increases linearly with the distance between prediction and target.

The *quadratic cost* function, or *mean squared error* is defined as follows:

$$C(z,t) = 0.5 \left\| z - t \right\|_2^2$$

where $\left\| \cdot \right\|_2^2$ is the square of $\left\| \cdot \right\|_2$, the *Euclidean distance* of z and t, or equivalently, the L^2 *norm* of their difference, given by:

$$\left\| z, t \right\|_2 = \sqrt{(z-t)(z-t)} \tag{4.2}$$

The (categorical) *cross entropy* cost, or *negative log-likelihood*, is used when model output is interpreted as a probability distribution over (discrete) classes:

$$C(z,t) = -\sum_i \log(z_i) t_i$$

where t is a d-dimensional vector with the target distribution over d labels or classes, and z is the prediction of the model over these classes, e.g. the output of a softmax layer, and *log* is the natural logarithm.

The next definition is a special case of general cross entropy, and the cost function employed in the models of Chapters 5 and 6. For models that learn a *hard* classification, i.e. if only their prediction over the target label matters, we can simplify cross entropy for the cost of the *correct label*:

$$C(z,t) = -\log(z_t)$$

where t is the index of the target relation R_t in the output of the softmax layer, i.e. z_t is the probability the model assigns to R_t.

4.3 Recursive Neural Networks

In this section we describe the forward pass of the *recursive neural network*, and the linguistic motivation for the architecture. We also discuss the limitations of the model, which lead to the tensor-based variant defined in the following section.

Sequence Processing The feed-forward architecture of Eq. (4.1), while shown to be powerful both formally and empirically, is ill-equipped to deal with input *sequences* of arbitrary length. In principle, we can construct an FFNN model for sequences of any given length, by matching input layer dimensions and sequence length. However, we are unable to use a *single* model instance to process sequences of different length, i.e. we cannot share parameters across different input sizes. Naively, we could try to share weights by setting input layer dimensionality to the highest expected sequence length, then pad all shorter sequences by 'dummy' vectors. Several problems would arise however – for example, dimensionality of such a model would be unnecessarily large for all but the longest input sequences.

A more principled solution consists in recursively defining composition, as seen in the case of the (simple) RNN architecture of Eq. (3.1). Due to the recursive composition process of the RNN, sequences of arbitrary length can be processed by one model instance, i.e. weights can be shared across examples of different length. Nonetheless, the architecture has sometimes been criticized as being syntactically limited, since composition proceeds in *linear order* over all sequences, ignoring their (assumed) syntactic structure. The recursive neural network is intended to address this limitation by composing words and phrases according to their position in a tree-structured representation of a sentence.

tRNN Architecture In the form presented here, the *recursive neural network*, or tree-structured recurrent neural network (tRNN), was introduced in Socher et al. (2010). The tRNN is closely related to the tree-structured network architecture of Goller and Küchler (1996), which was itself based on the *recursive auto-associative memory* (RAAM) of Pollack (1990). Socher et al. (2010), and the related proposals collected in Socher (2014), streamlined the presentation of the architecture and introduced several new variants of it, used in the construction of language models. As a result of this work, tRNN models gained substantial popularity in the field of natural language processing.[2]

The tRNN composition process structurally resembles the one defined in symbolic models of semantics. Intuitively, it can be understood as follows: In sentence processing tasks, model input is a *parse tree* of a sentence, together with vector representations of its words. At each tree node, a recursive composition function reduces two d-dimensional input vectors to a single d-dimensional output vector, representing the subtree rooted at the current node. Composition proceeds until a sentence is reduced, for the model output, to a single d-dimensional vector.

Tree-Structured Network Graph Formally, the order of composition in a tRNN model corresponds to a *directed acyclic graph* (DAG), specifically: a *directed rooted tree* with fixed branching factor two.[3] Tree nodes which are not parents of any other

[2] Arguably, this popularity has waned again in the more recent past, with tRNNs often being replaced by LSTMs, which themselves were superseded by convolutional networks, which in turn ...

[3] All definitions here are based on binary tree input, but extensions to n-ary trees are straightforward.

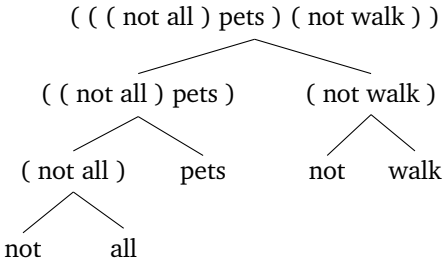

Figure 4.2: Bracketed input sequence and corresponding parse tree

nodes are called *leaf* nodes, and correspond to individual words of the input sequence. Parent nodes result from the composition of its child nodes, and correspond to *inside representations* that encode the entire subtree rooted at the node. We assume that tree structure information is provided externally, so model input is a sentence, its parse tree, and vectors for all words of the sequence. Tree input can be encoded in the form of bracketed expressions – see Fig. 4.2 for an example of such an expression and its corresponding parse tree. Alternatively, trees can be represented as sets of triplets of the form $(p \rightarrow c_1 \;\; c_2)$, where p is the parent node of child nodes c_1, c_2, which can either be leaf nodes or the parent node of another triplet.

Given an input sentence, the network graph for the forward pass of the model is constructed analogously to the parse tree of the input. Computing the activation vectors attached to the nodes of the graph proceeds bottom up, by composition of word vectors (the leaf nodes), previously computed internal node vectors, or a combination of the two. Formally, this bottom-up computation is given by the recursively defined tRNN composition function, defined next.

Composition Function For triplet $(z_{(x,y)} \rightarrow x \;\; y)$ of input $\boldsymbol{s}$, where x, y are word vectors attached to leaf nodes, or activation vectors computed by Eq. (4.3), the composition of x and y for the activation vector of $z_{(x,y)}$ is given by:

$$z_{(x,y)} = f\big(W[x\,;y] \;+\; b\big) \tag{4.3}$$

Here, $[x\,;y] \in \mathbb{R}^{2d}$ is the concatenation of $x, y \in \mathbb{R}^d$, and $W \in \mathbb{R}^{d \times 2d}$ defines the linear transformation of $2d$-dimensional input to d-dimensional output. Since dimensions of child and parent vectors are identical, $z_{(x,y)}$ can be the input for composition by Eq. (4.3) at the parent node of $z_{(x,y)}$, thus allowing for a complete recursive reduction of input $\boldsymbol{s}$ to a single d-dimensional vector.

Applying Eq. (4.3) to the example of Fig. 4.2 yields the root node representation:

$$f\big(W\big[f\big(W\big[f\big(W\big[v_{\text{not}}\,;v_{\text{all}}\big] + b\big)\,;v_{\text{pets}}\big] + b\big)\,; \\ f\big(W\big[v_{\text{not}}\,;v_{\text{walk}}\big] + b\big)\big] + b\big)$$

Model Variants We described the tRNN architecture used in the models of Chapters 5 and 6, but briefly mention here some existing variants of the architecture. Input

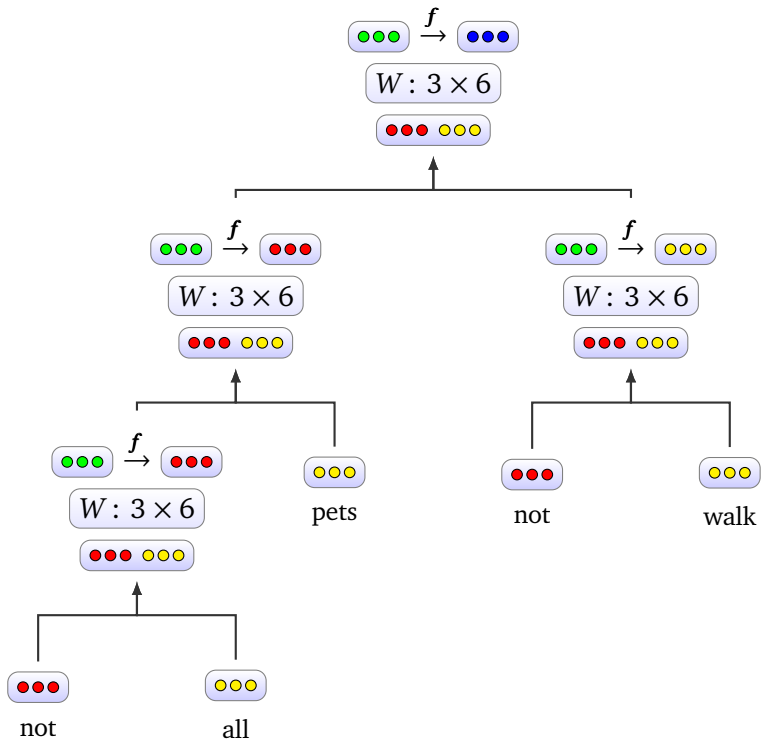

Figure 4.3: **Global tree structure and local composition in tRNN forward pass.** Rectangles represent vectors and colored circles their components; red and yellow denote left and right children of a parent node. Two-colored vectors at the base of a node represent their *concatenation*, used as input for the linear transformation by matrix W. The output of this transformation, in green, is fed into nonlinearity f. The resulting activation vector is colored according to its input position in the next composition step. Model output at the root node is blue.

other than sentences can be processed by tRNN models, including images, split up into image segments (Socher, 2014, Section 3.1). While we defined model output at the root node only, activations of internal nodes can be used as well, e.g. to predict sentiment ratings for phrases below sentence level (Socher, 2014, Section 4.3). Parse trees were assumed to be provided as external input, while a variant of the model can learn to parse input sentences itself (Socher, 2014, Section 3.1).

The model we described uses *unlabeled* tree input, i.e. without syntactic categories of tree nodes. Given labeled tree input, the *syntactically untied* tRNN (Socher, 2014, Section 4.1) defines the composition of nodes x, y with labels X, Y as:

$$z_{(x,\mathrm{X}),(y,\mathrm{Y})} = f\left(W^{(\mathrm{X},\mathrm{Y})}[x\,;y] + b\right)$$

where matrix $W^{(\mathrm{X},\mathrm{Y})}$ is chosen depending on the syntactic labels of the child nodes.

4.4 Recursive Neural Tensor Networks

While the tRNN model of the last section *globally* accounts for linguistic structure, by performing a hierarchical forward computation analogously to a parse tree of the input sentence, *local* composition at the tree nodes remains additive. As discussed previously, this type of function seems inadequate for the representation of composition including function words, such as adjectives, transitive verbs or negation.

The syntactic structure of composition in the tRNN suggests that the composition process is defined by a function of the constituent vectors, in analogy to the functions of formal semantics. In reality, tRNN composition input is a *concatenation* of constituents, which is equivalent to the addition of two *independent* mappings over the arguments. This representation seems to contradict our understanding of functional application in compositional models of semantics.

A possible alternative is the recursive matrix-vector model of Socher et al. (2012), where each word is represented by a matrix and a vector, and composition is defined by symmetric functional application of the constituents. The requirement that words can act as functions on other words is satisfied here, but due to the added word representations, model complexity quickly grows with vocabulary size. A preferable solution seems to be employing higher-order tensors, allowing for an adequate representation of argument structure, while still using a single, global composition function.

Tensor Layer by Slices With the above considerations in mind, the *recursive neural tensor network* (tRNTN) is proposed in Socher et al. (2013a,b). The previous tRNN composition function of Eq. (4.3) is augmented by a third-order tensor term, and composition is given by the sum of a linear map and a bilinear map of layer input. Defined by tensor slices, the tRNTN composition function is then:[4]

$$z_{(x,y)} = f\left(x^\top \mathcal{V}^{(1:d)} y \; + \; W[x\,;y] \; + \; b\right) \tag{4.4}$$

where $x, y \in \mathbb{R}^d$ is the composition input, and $\mathcal{V} \in \mathbb{R}^{d\times d\times d}$ is a full-rank third-order tensor defining a bilinear map from $\mathbb{R}^d$ input spaces to an $\mathbb{R}^d$ output space. The remaining terms are identical to those of Eq. (4.3).

$\mathcal{V}^{(i)}$, the i-th of d slices of $\mathcal{V}$, determines the i-th component of $z_\mathcal{V} \in \mathbb{R}^d$, the output vector of the tensor term, as follows:

$$z_{\mathcal{V},i} = x^\top \mathcal{V}^{(i)} y$$

We can think of the tRNTN layer output z as resulting from three d-dimensional terms: tensor output $z_\mathcal{V}$ defined above, z_W, the linear combination of concatenated input arguments parametrized by W, and the constant term $z_b = b$. Summing, and applying the nonlinearity, yields the layer output, $z = f(z_\mathcal{V} + z_W + z_b)$.

[4] We define the tensor function of Socher et al. (2013a) and Bowman et al. (2015), which differs from the tensor of Socher et al. (2013b), where input consists of concatenated child vectors, i.e. $[x\,;y]$ in place of x and y in Eq. (4.4).

Matricized Tensor Layer Rather than using the definition by slices of Eq. (4.4), we will usually describe the tRNTN in *matricized* form, which allows for a more intuitive exposition of the multiplicative interaction of arguments. We can then equivalently define the tRNTN composition as follows:

$$z_{(x,y)} = f\big(\mathcal{V}_{\mathsf{m}}(x \otimes y)_{\mathsf{v}} \; + \; W[x\,;y] \; + \; b\big) \tag{4.5}$$

where $\mathcal{V}_{\mathsf{m}}$ is the matricization of third-order tensor $\mathcal{V} \in \mathbb{R}^{d\times d\times d}$, and $(x \otimes y)_{\mathsf{v}}$ the vectorization of the tensor (or Kronecker) product of input vectors x and y. All other terms are identical to Eq. (4.4).

A more detailed view of the matricized tensor $\mathcal{V}$, and the vectorized Kronecker product of input vectors x, y is given by:

$$\mathcal{V}_{\mathsf{m}}(x \otimes y)_{\mathsf{v}} = \begin{bmatrix} v^{(1)}_{1,1} & \dots & v^{(1)}_{1,d} & \dots & v^{(1)}_{d,1} & \dots & v^{(1)}_{d,d} \\ \vdots & & & \ddots & & & \vdots \\ v^{(d)}_{1,1} & \dots & v^{(d)}_{1,d} & \dots & v^{(d)}_{d,1} & \dots & v^{(d)}_{d,d} \end{bmatrix} \begin{bmatrix} x_1y_1 \\ \vdots \\ x_1y_d \\ \vdots \\ x_dy_1 \\ \vdots \\ x_dy_d \end{bmatrix}$$

Here, tensor $\mathcal{V}$ is represented by its *mode-3 matricization*, defined in Appendix A.0.3, resulting in the $d \times d^2$ matrix $\mathcal{V}_{\mathsf{m}}$. The Kronecker product of input x, y is given in vectorized form, arranged as a $d^2 \times 1$ column vector. We denote here by superscript indices $v^{(1)} \dots v^{(q)}$ the slicing index of the mode-3 matricization. These indices also allow us to connect the *slice* definition with the *matricized* definition: In the matricized view, the d slices of Eq. (4.4) are vectorized, and vertically stacked row-wise to form matrix $\mathcal{V}_{\mathsf{m}}$. Composition of $\mathcal{V}_{\mathsf{m}}$ with $d^2 \times 1$ column vector $(x \otimes y)_{\mathsf{v}}$ is given by matrix multiplication, and yields a d-dimensional vector, the tensor term of Eq. (4.5), and equivalent to the tensor term of the slice definition.

In the previous definitions we described composition of constituents belonging to $\mathbb{R}^d$ spaces, for a composition output in $\mathbb{R}^d$. We can in principle define composition over arbitrary dimensions, e.g. by some tensor $\mathcal{V} \in \mathbb{R}^{r\times(p\times q)}$.[5] In order to allow for composition of arbitrary constituents, we will generally demand that their dimensions match, thus constraining the mapping to $\mathbb{R}^{r\times(p\times p)}$. For *recursively* defined composition, such as in the tree-structured networks described here, we demand additionally that input and output dimensions match, thus constraining the mapping further to the case of our definitions, $\mathcal{V} \in \mathbb{R}^{p\times(p\times p)}$.[6]

[5] Parentheses and index order here mark, purely for convenience, a mapping from $\mathbb{R}^p \times \mathbb{R}^q$ to $\mathbb{R}^r$.

[6] Note that the models of Chapter 5 also employ the functions of Eqs. (4.3) and (4.5) in a non-recursive composition step, and input and output dimensions differ in this case.

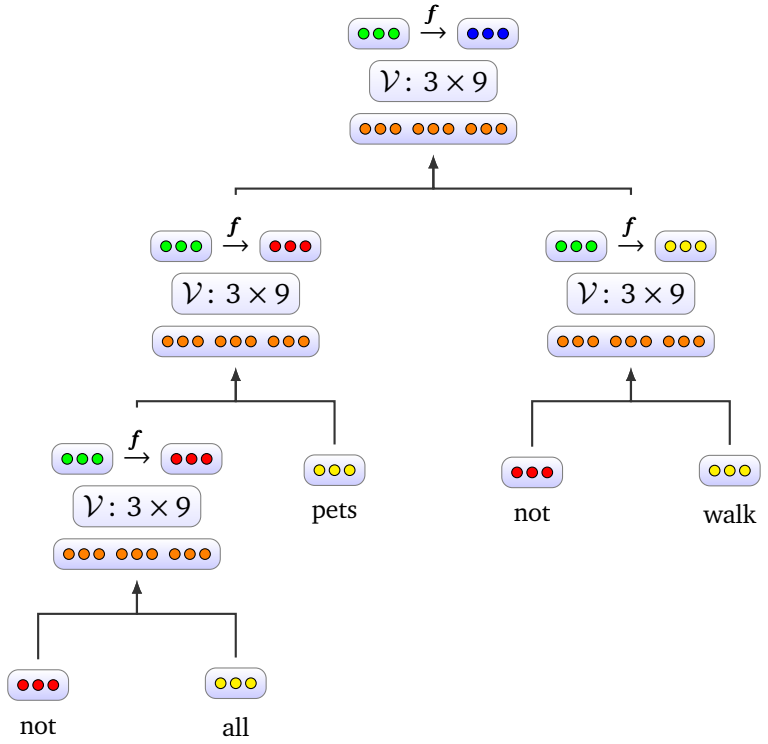

Figure 4.4: **Global structure and tensor composition of tRNTN forward pass.** The syntactic structure of the tRNTN forward pass is identical to the tRNN one, but local composition is multiplicative rather than additive. Node input in orange is given by the Kronecker product of child vectors. The (matricized) tensor $\mathcal{V}$ is applied to this vector by matrix multiplication. The graph only depicts calculations of the tensor term; the full function of Eq. (4.5) corresponds to summing green vectors at each node of Figs. 4.3 and 4.4, and adding the (omitted) bias term.

4.5 Contrasting Additive and Multiplicative Composition

Using the definitions of the previous sections, we can now describe in greater detail how additive and multiplicative composition differ in the two recursive neural network architectures we described here.

Reinterpreting Additive Composition By a basic algebraic equivalence we can split the tRNN $d \times 2d$ transformation matrix W, which is applied to concatenated child vectors, into two separate matrices, such that each matrix half contains the transformation parameters for a single child vector. We can then replace the standard tRNN definition, repeated below as Eq. (4.6), by the equivalent Eq. (4.7), where W^x and W^y are the $d \times d$ matrix halves transforming child vectors x and y separately.

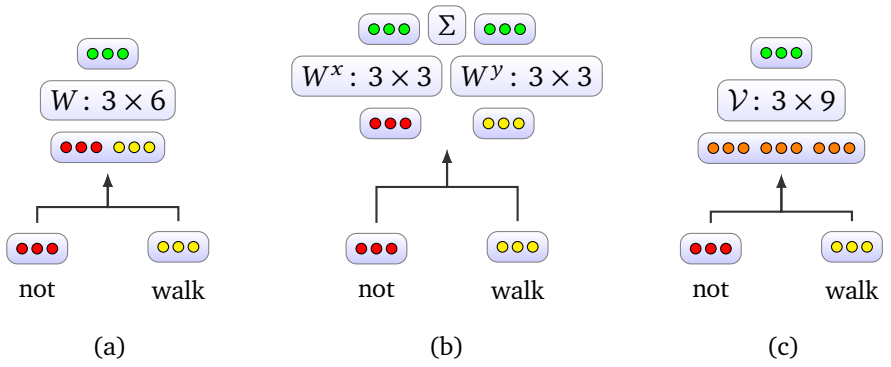

Figure 4.5: **Additive vs. multiplicative composition in tRNN and tRNTN models.** Close-up view of the composition process shown in previous Figs. 4.3 and 4.4. Subfigures 4.5a and 4.5b of this graph equivalently depict tRNN composition. 4.5b visualizes Eq. (4.7), emphasizing that additive composition is a sum of independent linear maps for the left child (red) and right child (yellow). Accordingly, we separate the input for transformation W, split into W^x, W^y, for a more accurate depiction of additive composition than 4.5a. Input for tensor $\mathcal{V}$ in 4.5c is the tensor product of child vectors. The resulting 9-dimensional vector is colored orange to emphasize the direct interaction of left and right child components given by their products.

For ease of comparison, we repeat the tensor layer definition as (4.8) below:

$$z = f\big(W[x\,;y] \;+\; b\big) \tag{4.6}$$

$$z = f\big(W^x x \;+\; W^y y \;+\; b\big) \tag{4.7}$$

$$z = f\big(\mathcal{V}_{\mathsf{m}}(x \otimes y)_{\mathsf{v}} \;+\; W[x\,;y] \;+\; b\big) \tag{4.8}$$

Comparing Eqs. (4.7) and (4.8) we can naturally contrast additive and multiplicative composition behavior. It follows from Eq. (4.7) that the additive transformation of a tRNN does not contain parameters which scale component *combinations* over child vectors x and y. While composition output is a weighted sum of all components, the parameters of W apply to components individually, in effect, composing them independently. In contrast, the mapping given by $\mathcal{V}_{\mathsf{m}}$ in Eq. (4.8) applies to the product of child vectors x and y, i.e. the tensor parameters define a weighted sum of component *combinations* over the constituents.

Additive and Multiplicative Negation In our example of Fig. 4.5, composition arguments are 'not' and 'walk', i.e. the model has to represent *negation* of an expression. In the additive model, composition is a weighted combination of the features that define 'not', and the features of 'walk'. Informally, we can think of the representation as a (weighted) average of the *separate* meanings of the constituents, apparently contradicting our linguistic intuitions of how negation should be represented. In contrast, composition output in the multiplicative model is given by a weighted sum

of (product) *combinations* of the features of 'not' and 'walk'. Here, components of the negation expression can directly 'interact' by multiplication with the components of the negated expression. Accordingly, the tensor representation appears to be closer to our linguistic conceptions of negation.

4.6 Learning

4.6.1 Learning by Gradient Descent

Gradient descent, or in practice more commonly: *stochastic* gradient descent (SGD) is widely used as the fundamental algorithm in training neural networks. Given a differentiable, convex multivariate function defining the network error on a given task, SGD in fact guarantees convergence to a *global* (error) minimum.

In practice, most tasks and their objective functions do not satisfy the convexity requirement, yet there is mounting evidence that SGD performs well in practice on these problems as well. Dauphin et al. (2014) describe the problem of *saddle points* as constituting a major challenge for high dimensional non-convex optimization, more so than local minima.

Note that adjusting the parameters of function φ in the direction of the gradient constitutes the greatest *increase* of the value of $\varphi(x)$. Since our goal is *minimizing* the output of the network function, i.e. the error, we adjust parameters by taking a step in the direction of the *negative* gradient.

The parameter updates defined by gradient descent can be written in vectorized form as follows:

$$\theta_{n+1} = \theta_n - \eta \nabla_\theta \varphi(\theta, x, t)$$

Where θ_{n+1} is the new parameter set of the network function resulting from updating the current parameters θ_n by the (adjusted) gradient. We write $\nabla_\theta \varphi(\theta, x, t)$ for the gradient of the cost with respect to the parameter set θ, for input x and target t, both assumed to be fixed for the calculation of the gradient.[7]

4.6.2 Backpropagation

An adjusted backpropagation algorithm for neural networks defined on tree-structured input, termed *backpropagation through structure*, was proposed in Goller and Küchler (1996).

So far, the output of the error function told us by how much the model is off from the target solution, we still need to know how to improve the adjustable parameters of the model in order to reduce the error. Gradient descent provides us with the general format to do so, but in an implementation of a non-trivial model, we also require an *efficient* way to perform the necessary calculations. A naive approach

[7] Note that $\varphi(\theta, x, t)$ refers to the network function including the cost calculation. We could instead explicitly separate the cost function and the actual model by writing: $C(\varphi(\theta, x), t)$.

might consist of a layer-wise application of the (multivariate) chain rule, the more efficient way to proceed is via *backpropagation* (BP), or rather, in the case of our model, *backpropagation through structure*.

The backpropagation algorithm is sometimes characterized as "the chain rule plus dynamic programming" in the computer science or machine learning literature. Alternatively, based on viewing BP as a top-down sequence of matrix-vector calculations, the algorithm can be characterized as exploiting associativity of matrix multiplication in order to perform the required computation faster than a naive implementation proceeding in the reverse direction.

Backpropagation Equations The BP algorithm is defined by four fundamental equations: a formula for the delta error of the final output layer; a formula to recursively calculate a lower layer's delta error, given the layer immediately on top, and two formulas allowing calculation of the gradients w.r.t. the trainable parameters, given the delta error of the corresponding layer.

In training the tRNN, two important differences need to be noted between *backpropagation through structure* (the name given to training an unrolled tRNN via backpropagation) and backpropagation as employed in FFNN. First, the requirement to sum up all layers' gradients w.r.t the trainable parameters before performing exactly one update at the end of one parameter update run (i.e. at the end of processing one minibatch of training input). Second, since intermediate layers take two vectors from preceding layers (leaf nodes or intermediate nodes themselves) as input, the delta error has to be split up between the child nodes, so that only that part of the error vector is passed into the branch of the tree that has contributed to the error calculated at the higher level of the tree.

Tensor Gradients One additional complication comes from the update now being necessary as well for the tensor, i.e. calculating the gradient vector for a bilinear map. The crucial difference to the matrix calculation is that now, we derive over two arguments multiplied with each other, instead of a single argument, resulting in the gradient calculation on the tensor component being (in vectorized form):

$$\frac{\partial E}{\partial \mathcal{V}} = \delta^{l+1} x \otimes y$$

4.6.3 Advanced Optimization

Gradient descent, or more commonly, stochastic gradient descent, is the general optimization algorithm used to train most neural network models. Various methods exist that 'optimize' gradient descent-based optimization, for example, algorithms that adjust the step size of parameter updates during training.

Adaptive Learning Rates As noted in Section 4.6.1, successful training requires scaling down gradients by a learning rate before using them to update parameters. The simplest solution is to use a *constant* learning rate, in which case its value becomes

an important hyperparameter of the model. More sophisticated optimization methods exist that adjust a manually set learning rate during training, or completely remove the need for a manually specified rate. These adaptive learning rates can further be allowed to vary between different *parameters*. AdaGrad, and its variant AdaDelta, are algorithm implementing such a parameter-dependent adaptive learning rate, and are the optimization algorithms used in the models of the following chapters. These methods, similar to momentum methods, keep track of historical gradient information, and use this history to scale the current gradient before updating weights. Letting the learning rate vary between weights is motivated by the idea that infrequently updated parameters, or parameters that are only changed by a small amount, should receive a learning rate 'boost' compared to those that are updated more frequently.[8]

AdaGrad The *AdaGrad* ("adaptive gradient") algorithm was introduced by Duchi et al. (2011). It was widely adopted in the following years, often used to train deep neural networks. By now, it has been largely superseded by its more recent variants, such as AdaDelta (defined next) or *Adam* (Kingma and Ba, 2014).

Section 4.6.1 introduced the basic update rule of gradient descent methods, where $\theta_{n+1,i}$, the i-th parameter of θ at step $n+1$ is given by:

$$\theta_{n+1,i} = \theta_{n,i} - \eta\, g_{n,i}$$

where η is the step size or learning rate, and $g_{n,i}$ is used as a shorthand for the i-th element of $\nabla_\theta \varphi(\theta, x, t)$ at step n, i.e. the gradient with respect to the i-th adjustable network parameter.

Based on the regular update rule, the *AdaGrad update* is given by:

$$\theta_{n+1,i} = \theta_{n,i} - \frac{\eta}{\sqrt{G_{n,(i,i)} + \epsilon}}\, g_{n,i}$$

Here, G_n is a diagonal matrix containing the gradient history collected over previous updates, and its entry $G_{n,(i,i)}$ contains the sum over the squared gradients with respect to parameter i up until the current update step n. As before, η is a fixed, initially specified learning rate, which is however scaled here by the gradient history term in the denominator. ϵ is a small constant added for numerical stability in implementations of the algorithm.

In vectorized form, the AdaGrad update rule is defined as:

$$\theta_{n+1} = \theta_n - \frac{\eta}{\sqrt{G_n + \epsilon}} \odot g_n$$

Note that the update rules defined above still need to be adjusted for the number of examples seen during the gradient calculation. Usually, this means scaling the equations above for the hyperparameter of minibatch size.

[8] While the usual motivation of adaptive algorithms like AdaGrad is to ensure that parameters are updated sufficiently despite *input* sparsity, we briefly discuss in chapter Chapter 5 why these methods seem to gain additional importance in the training of *tensor* models.

AdaDelta *AdaDelta*, introduced in Zeiler (2012), is a variant of the AdaGrad algorithm, intended to mitigate the tendency of the original method to indefinitely accumulate historical gradients, eventually suppressing updates entirely. Instead of collecting all previous gradients like AdaGrad, the history of AdaDelta is given by a *moving average* of gradients (or updates), effectively 'forgetting' historical gradients that were encountered in the distant past.

The history of AdaDelta is given by the following *moving average* over gradients:

$$E\left[g^2\right]_n = \rho E\left[g^2\right]_{n-1} + (1-\rho)g_n^2 \tag{4.9}$$

where g_n is the gradient at step n as before, g_n^2 the vector of its squares, and ρ is a manually set constant, determining the *decay* of influence of historical gradients on the current gradient (update). Note that due to recursive multiplication of the decay factor ρ, Eq. (4.9) defines an *exponential moving average*.

Based on the history of squared gradients, the AdaDelta parameter update is:

$$\theta_{n+1} = \theta_n - \frac{\eta}{\sqrt{E\left[g^2\right]_n + \epsilon}}\, g_n \tag{4.10}$$

4.6.4 Regularization

Regularization techniques are used to reduce the *generalization error* of models, i.e. their aim is to keep performance on unseen data – usually measured on a test set – close to the model performance on the training data. Regularization is particularly relevant in complex models, with a large number of trainable parameters, to prevent the model from *overfitting*.

Regularization Methods Various regularization techniques are in use, the simplest one being *early stopping*, i.e. interrupting training once model performance on out-of-sample error begins to degrade. L^1 and L^2 *regularization* both implement a form of *weight decay* – we will describe these two methods in detail in the following paragraph. Another method that gained popularity in recent years is *dropout*, introduced in Hinton et al. (2012), which prevents overspecialization of the learned function by randomly disabling a subset of weights at each iteration. *Batch normalization* (Ioffe and Szegedy, 2015), a widely employed algorithm for normalizing layer input, while not explicitly introduced as a regularization method, has been shown to improve generalization capacity of models, similar in effect of explicit regularization.[9]

Weight Decay *Weight decay*, arguably the most widely employed regularization technique, adds a regularization term to the training objective that 'penalizes' weights which deviate greatly from their prior value. Two types of weight decay are frequently

[9] There is mounting (empirical and formal) evidence that the different methods are strongly linked. For example, Collobert and Bengio (2004) relate the effects of early stopping and L^2 regularization, while Wager et al. (2013) establish a link between L^2 regularization and dropout.

used, L^1 and L^2 regularization. Both inhibit weight specialization by adding a term summing over network parameters θ, thereby progressively increasing the cost of weights with high (absolute) values.

L^1 *regularization* adds the sum of absolute values of θ to the training objective:

$$\lambda \sum_i |\theta_i|$$

while L^2 *regularization* adds the sum of squared values:

$$\frac{\lambda}{2} \sum_i \theta_i^2$$

Here, λ is the *regularization coefficient*, a constant value to scale the impact of the regularization term. Note that λ is frequently multiplied by $\frac{1}{2}$ in L^2 definitions to cancel out the square when taking the derivative. We can equivalently express L^2 regularization in terms of the L^2 norm of Eq. (4.2) as:

$$\frac{\lambda}{2} \|\theta\|_2^2$$

4.7 Conclusions

The architectures we described in this chapter appear to be capable of performing well on compositional tasks, due to the structurally guided processing mechanism they employ (visualized in Sections 4.3 and 4.4). In addition, the tRNTN architecture employs a (full-rank) third-order tensor as part of its composition function, which potentially adds to the expressiveness of the model.

The next step is then to put these models to the test: first, in a replication of the main results of Bowman et al. (2015), then, in an extension of these experiments that aims to evaluate whether the models are able to *generalize* sufficiently.

Part II

Results and Discussion

Chapter 5

Language Inference with Neural Networks

We describe in this chapter the natural language inference task of Bowman et al. (2015), and report the results of our replication of the main experiments of this study.

5.1 Quantified Inference Task

The quantified inference task of Bowman et al. (2015) is aimed to simulate natural language inference, with the goal of learning which type of inference relation holds between two expressions given to the model. The data used to train and test models is generated by an adapted (and reduced) version of the natural logic of MacCartney (2009), previously introduced in Section 2.1.2. The relations that the model should predict are the seven basic relation of the natural logic. Examples of pairs of input sentences are shown in Table 5.1.

The model uses as its architectural core the tRNN and tRNTN of the previous chapter. A pair of sentences, together with a target relation, i.e. the relation that is correct and needs to be learned by the model, is given as input. Each sentence is processed by a separate network based on either the tRNN or tRNTN architecture. This part of the modeling process is performed by what the authors call the *composition* networks. As seen in the previous chapter, the final output of these networks is a single vector, the model representation of the entire sentence. While the sentences are processed separately, the *weights* of the two composition networks are shared, i.e. trained over all inputs.

The two resulting vectors are then fed into a separate *comparison* layer, employing the same function as the composition layer below, but with a distinct set of parameters defining the function of the layer. As the final step, the output of the comparison layer feeds into a softmax classifier layer, which represents the model's belief of which relation holds for the pair, expressed as a probability distribution. The highest valued, i.e. "most likely" relation is selected, by applying the max function to the softmax layer output, resulting in the relation *predicted* by the model for a given pair of input

Symbol	Sentence Pair Example	
$x \equiv y$	`((all warthogs) walk)`	`((all warthogs) walk)`
$x \sqsubset y$	`((all warthogs) walk)`	`((all warthogs) move)`
$x \sqsupset y$	`((all warthogs) move)`	`((all warthogs) swim)`
$x \wedge y$	`((all warthogs) walk)`	`((not_all warthogs) walk)`
$x \mid y$	`((all warthogs) swim)`	`((all warthogs) walk)`
$x \smile y$	`((all warthogs) (not walk))`	`((not_all warthogs) swim)`
$x \# y$	`((all warthogs) (not walk))`	`((not_all warthogs) (not swim))`

Table 5.1: Examples of NatLog basic relations used in training data

sentences. The model layout of the quantified inference task, instantiated by an example from the data, is shown in Fig. 5.1.

Language and Vocabulary The synthetic language that is used to generate the data consists of 20 lexical items: 10 quantifiers, a 'symbol' for negation, 5 nouns and 4 verbs. The quantifiers set contains 'all', 'some', 'most', and two numerical expressions ('two' and 'three'). The remaining 5 items are distinct symbols of their negation. This choice, of representing quantifiers and their negation as distinct, non-compositional symbols will become relevant in our further experiments shown in Section 6.2.

5.2 Model Implementation

We opted for a low-level implementation of the investigated models, by the reasoning that doing so should facilitate a deeper analysis of the model internals. Basing our implementation on one of the established machine learning frameworks like *TensorFlow* or *Theano* presumably would have been easier, but also would have limited our ability to perform a low-level analysis on the individual components of the architecture, in particular, the tensor term.

For example, the complete low-level control over the composition mechanism allowed us to perform an initial investigation whether a computationally tractable,

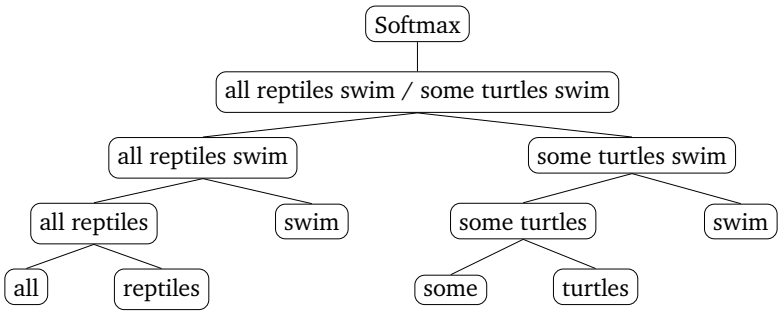

Figure 5.1: Model layout of the Quantified Inference Task

	Reported		Replication	
	Train	Test	Train	Test
25/75 tRNN	99.6	99.2	97.8	96.4
30/90 tRNN	-	-	100	99.6
16/45 tRNTN	-	-	100	99.9
25/75 tRNTN	100	99.7	100	100

Table 5.2: Classification accuracy (%) reported by Bowman et al. (2015) and of our replication (average over 3 training runs using early stopping)

'minimal' tensor layer is expressive enough to duplicate our main results of Chapter 6 for which a full-rank tensor was used.[1,2]

5.3 Replication of Quantified Task Results

The data provided by Bowman et al. (2015) contains different sets, presumably corresponding to different splits of the data into training and test portions. Unfortunately, the authors are not explicit about their motivation behind the split, and further, do not specify which sets where used for their results on the quantified inference task. We could not determine the systematicity of these sets, and did not find significant differences between them in our own experiments. In all of our experiments, we use therefore the strategy of testing models for a single training run across all data sets, to establish that no major performance differences emerge. We then report our results as an average of several training runs on only one of the sets, usually the `f1` set of Bowman et al. (2015), since multiple training runs across all sets were prohibitive due to training time.

The first step of the experimental evaluation of the tensor network consisted in a replication of the results reported by Bowman et al. (2015). Our attempt to replicate the original results succeeded fully, even reaching accuracy scores above the reported results in some cases.

Table 5.2 shows the reported results of Bowman et al. (2015) and our own replication results. We could not replicate the results of the tRNN model with dimension 25/75 (composition network/comparison layer), hindered by the fact that the authors did not provide details of all relevant hyperparameters. Instead, we were able to replicate (and slightly improve) the reported results by increasing model dimensionality and using optimized hyperparameters. In particular, we manually tuned the strength of L2 regularization, since the higher dimensional tRNN tended to overfit otherwise.

[1] We report the initial results of this architectural adjustment in Section 6.5.2.

[2] Example functions of our research code are shown in Appendix B. The full code and data are made available on: `https://github.com/backslashvarphi/`.

As a consequence of having trained a higher-dimensional tRNN than the original study, we could also more closely compare the tensor and matrix network in terms of their total number of parameters. Recall that in the cubic tensor, the number of tensor weight grows much faster than the quadratic matrix layer. Comparing a model using a lower dimensional tensor and a higher dimensional matrix brought us closer to a 'fair' comparison between the models in terms of the total number of trainable parameters. The 16/45 tensor contains around 18,000 trainable weights, in contrast to the 8,500 weights of the 30/90 matrix model. We noted also that further increasing the dimensions of the matrix model did not improve results.

While the results of Bowman et al. (2015) report a small, but noticeable difference between the tRNN and the tRNTN, it seems that we were able to level the difference by our increased dimensional tRNN model. As will be seen in Chapter 6 however, we eventually succeeded in sharpening the contrast between these two models by an adjustment of the data and using a new testing regime.

5.4 Other Experiments

Our main goal was a replication of the maximally complex artificial task ("quantified experiments" in Bowman et al. (2015)), as a prerequisite for the analysis of Chapter 6. However, we also performed some experiments on the natural language data sets of the original study, both to confirm the correctness and efficiency of our implementation, and to determine whether the reported results of Bowman et al. (2015) are aligned with our own findings.

The natural language experiments of Bowman et al. (2015) are based on the SICK data set of Marelli et al. (2014). The set contains around 10,000 natural language sentences, from image and video captioning data. The dataset provides target relations, and parse trees. This data set only uses three target labels, 'neutral', 'entailment', and 'contradiction'.

The model layout we used was simpler than that of Bowman et al. (2015): we did not use pre-trained embeddings, and omitted the additional layers between the vocabulary matrix before and the composition network used in the original model. Despite these simplifications, the tensor model reached a performance of 99%/72% after about 500 epochs, with layer dimensions 16/45 (for a total of around 53k adjustable parameters). While our results stay below the reported values (98%/77%), our experiments seem to confirm that the model is in principle able to learn the task to a satisfactory degree.

Chapter 6

Model Representations and Generalization

The previous chapter gave an account of our preliminary results, in particular, a replication of the experiments on language inference and reasoning with quantifiers and negation by Bowman et al. (2015).[1] An independent confirmation of these findings appears to be a prerequisite for further analysis – however, replication on its own cannot provide much additional insight into the inner workings of the models which produced the results.

Accordingly, in the current chapter we aim to address some of the latent questions that remain after a replication and comparably basic quantitative evaluation of the results: What are the limits of the inference system learned by the model? How are the components of this inference system represented inside the model? Finally, and more broadly, our goal is to assess whether the claim is justified that the model learned to "reason logically" in a general sense.

6.1 A Glimpse into the Black Box

The field of machine learning, and in particular, neural network research, has sometimes been criticized as taking a "black box" approach to scientific modeling. These model often yield impressive practical results, the argument goes, but the solutions emerging from them are inherently opaque, and do not advance our understanding of the modeled phenomena themselves. We suggest to place the analysis of this chapter within the context of this discussion then, going beyond simply reporting of barren performance results, instead attempting to gain a better understanding of these models that seem to defy direct interpretation.

[1] A closely related set of experiments is described in Bowman (2014). For additional commentary by the author on using MacCartney (2009) natural logic, see Sections 3.3 and 5.4.2 of Bowman (2016).

Motivation The authors of Bowman et al. (2015) claim, quite generally, that the results of their experiments indicate that neural network models are able to "learn logical semantics". As discussed in Chapter 2, distributional semantic models evidently perform well on tasks requiring the extraction and representation of *lexical* information, but it is less certain whether they can also perform well on tasks based on *structural* semantic processing. Accordingly, the assertion of Bowman et al. warrants further investigation, since a confirmation of their claim could be a substantial addition to the debate whether logical reasoning, widely considered to require symbolic treatment, can be represented by models without symbolically defined logical mechanisms.[2]

Choice of Experiments We considered two of the experiments of Bowman et al. (2015) as potential candidates for further analysis as outlined above: the *quantified inference task* – the structurally most complex of the experiments using synthetic language data, and the *SICK experiment* – the natural language experiment based on the SICK textual entailment challenge.

The SICK experiment (Bowman et al., 2015, Section 6) seemed appealing, considering that modeling actual language data is the eventual goal of natural language processing. Our replication of the SICK experiment (see Chapter 5) produced results sufficient for further investigation, yet we eventually opted for the synthetic language experiment. For our investigation, we were mainly interested in a clearly delineated model behavior on *logical* aspects of the task, as well as the ability to analyze model internal *representations* of logical components. For both of these key desiderata, the SICK experiment proved to be less than ideal.

First, the 'messiness' – or *complexity* – of natural language data complicated the separation of (structural) logical aspects of reasoning and lexical semantic aspects. Second, since our goal was to analyze and interpret the model internal representations, the higher dimensionality of the models used to run the SICK experiment proved to be problematic. For example, our results on model representations of logical constants (Section 6.4 of this chapter) were enabled by the relatively low dimensionality of models that were able to solve the quantified inference task. In higher-dimensional models, a similar 'direct' interpretation of learned representations presumably would have not been possible, despite employing dimensionality reduction techniques on the model output and internal representations.[3]

Quantified Inference Task Following the reasoning above, we focused our analysis on the inference experiments using artificial language data with quantifiers and negation (Bowman et al., 2015, Section 5). The controlled environment of an artificial

[2] We will eventually arrive at a nuanced conclusion regarding the question. At the risk of spoiling the surprise, it seems that symbolic approaches are perhaps not completely doomed yet.

[3] In principle, dimensionality reduction allows visualization of even higher dimensional data. However, the results of Section 6.4 rely on a particular reduction method (PCA), and a small number of resulting (informative) dimensions. Other techniques, such as t-SNE, could visualize more complex data, but would not preserve the global organization of model representations that we were interested in.

language allowed us to cleanly distinguish different components, for example, to analyze particular logical constants in isolation. Further, the comparably low model dimensionality allowed us to extract representations from the trained networks that (almost) directly correspond to the internal representations – permitting an undistorted view into the "black box" of our models.

Despite the synthetic nature of the language data, we believe the quantified inference task is of considerable interest, since it contains a wide range of language phenomena encountered in actual language data. The data consists of recursively constructed sentences, includes quantification and negation, and requires accounting for interaction between lexical and structural semantic information, as well as non-trivial syntax/semantics interplay. We also describe in the following section how the original task was adjusted to make the learning process more challenging, by adding a 'missing' aspect of compositionality to the data.

Nonetheless, since we are analyzing a synthetic data task, applicability of our results to larger natural language tasks cannot be taken for granted, and further evidence or formal arguments in favor of their generality is required. We will discuss these limitations, and conversely, the extent to which our results seem applicable to language task in general in the results discussion of Section 6.5.

Analysis Steps We outlined above – in general terms – the goals of our analysis: assessing to which degree the investigated models learned a general form of reasoning, and to gain an understanding of how the models solved the reasoning tasks 'internally'. Another (related) objective is to find evidence whether the more complex, theoretically well-justified model outperforms the simpler model in practice. In other words, we want to know whether the three-way (tensor) model has any significant advantage over the two-way (matrix) model, to justify its added complexity.

Our first step, intended to heighten the performance distinction between the two models, consists of a subtle adjustment of the logical language used to generate the task data. This adjustment, mainly affecting model *training*, increases the task difficulty, while potentially also rendering the data structurally more informative. Next, we employ a different set of sentences for *testing* the trained models on unseen data. The new test set, motivated by logical considerations, replaces the original random test cases, and should provide a better measure of the models' generalization capacity.

The previous steps constitute the *quantitative* part of our analysis, which will be followed by the *qualitative* analysis of learned representations extracted from the trained models. Here, we focus mainly on representations of logical constants of the task language, i.e. the classically interpretable quantifiers and the negation operator.

Finally, we will discuss both quantitative and qualitative results in tandem, and suggest a causal link between the two.

Original	CompQuant	Examples
`not_all`	`not all`	`(((not all) warthogs) walk)`
`not_most`	`not most`	`(((not most) mammals) (not move))`
`no`	`not some`	`(((not some) turtles) growl)`
`lt_two`	`lt two`	`(((lt two) reptiles) (not walk))`
`lt_three`	`lt three`	`(((lt three) (not pets)) (not swim))`

Table 6.1: Original and adjusted compositional quantifiers (*CompQuant*)[4]

6.2 Increasing the Task Difficulty

As noted in Chapter 5, model performance on the *quantified inference* task was close to perfect, with both tRNN and tRNTN models achieving nearly 100% prediction accuracy on training and test data. Designing models that perform well is evidently the purpose of applying machine learning methods to natural language tasks. Yet, instances of *extremely* high model performance can also invite additional scrutiny, both of the methodology and the investigated task.

One concern that can be voiced is that the task was potentially too simple, only allowing limited conclusions to be drawn from the results achieved on it. A related concern regarding task difficulty arises from the specific goals of our analysis. Since both model architectures perform close to the maximum, we are unable to determine whether a substantial capacity difference exists between the simpler tRNN model and the more complex tRNTN model. A higher task difficulty is likely to decrease the performance of both models, yet ideally, might not affect both models equally, thereby exposing a potential performance difference.

Finally, the reported performance raises doubts whether the *holdout method* was implemented with sufficient consideration for the properties of the particular task. Specifically, the possibility needs to be considered that the original train/test data split is unsuitable as a test for model generalization on a logical reasoning task. This last concern will be the topic of Section 6.3, while we address here the previous issue by increasing the general difficulty of the task.

Incomplete Compositionality There are several ways to reach the objective of increasing the difficulty of the quantified inference task of Bowman et al. (2015). For example, we could increase the vocabulary size of the language used to generate the task data, by adding lexical items to an existing word class (nouns and verbs) or a new one. However, since our main interest lies in the logical aspects of the task, we instead focused on the logical expressions of the task language rather than the content words. Taking a closer look at the data, we were able to identify the potential for a selective difficulty adjustment among these expressions.

[4] Verbatim data excerpts, highlighting the effect of a single input character on the implicit task definition. Original data: negation and quantifiers connected → non-analytical quantification step. Adjusted data: separate symbols → compositional quantification step.

We noticed that the data of Bowman et al. (2015) includes a connecting character between quantifiers and 'not', effectively representing quantified expressions and their negation non-analytically (see Table 6.1 and corresponding footnote). For example, 'all' and 'not all' are distinct symbols, not further decomposable by the model. As a result, no systematic relation is established by the data between quantified expressions and their negation, deviating from an otherwise compositional task.

Since the experimental section of Bowman et al. (2015) makes no mention of this peculiarity, we can only speculate as to why the data is structured in this way. Possibly, the authors wanted to avoid the 'ambiguity' that would result from associating a single symbol with the negation of both lexical terms ("not mammal") and logical expressions ("not some"). If so, introducing a second negation symbol, i.e. distinct "lexical negation" and "logical negation", could perhaps be motivated as a concession to the demands of a data-driven approach. Even so, from a semantic perspective, these 'non-compositional' quantifiers appear to be both unwarranted and undesirable, and should be replaced by their compositional counterparts.

Our difficulty adjustment consists then in extending task compositionality to the quantification step of sentence construction, removing the (non-analytical) negated quantifier expressions, and replacing them by a combination of 'not' and the appropriate quantifier. A similar replacement is made for expressions of the form "(exactly) three" and "less than three".

CompQuant Data We use the `f1` data set of Bowman et al. (2015) for this data adjustment, and for the further experiments of this section. As before in Section 5.1, our experiments use a single data set (of five) since no discernible differences could be found between single model runs on these sets, and no motivation of the splits is offered or could be derived by us. For brevity, we will from now refer to training or test data that we adjusted for quantifier compositionality as *CompQuant* data.

The original data contained 5 quantifiers (*all*, *most*, *some*, *three*, *two*), in addition to their negated forms, for a total of 10 language elements. Removing the negated forms, adding a symbol for "less than", and reusing the negation symbol for nouns and verbs for quantified expressions, our adjustment decreases the number of symbols by 4 in total. Despite this reduction of the vocabulary size, the overall difficulty of the task increases. For one, the composition process now requires an additional processing step for negated (and "less than…') sentences . While this increase of complexity might seem minor, recall that task complexity is comparably constrained due to short sentences being generated by the language. In terms of (internal) network nodes required for the compositional processing of one sentence, our adjustment increases the maximum number from 4 to 5, i.e. a noticeable increase in comparison to the original task complexity.

Further, replacing the separate symbols arguably makes it harder for models to merely *memorize* the distinct cases – as opposed to finding a *general* solution. Previously, distinct symbols (represented by distinct vectors of the model vocabulary) were assigned to each type of quantified expressions, making it easier for models to

	Replication		CompQuant	
	Train	Test	Train	Test
30/90 tRNN	100	99.6	98.3	97.7
25/75 tRNTN	100	100	100	99.9

Table 6.2: Replication results (original data) and CompQuant results (original data with substituted quantifiers of Table 6.1).[5]

fit the different cases individually by adjusting the weights of the associated vectors during training. Memorization is still not excluded, but seems less likely now, since negated (and "less than") expressions now globally share the respective symbol. While potentially also improving the generality of model solutions, at face value, the more systematic data representation should be harder to fit on a *per case* basis, and therefore, pose a greater challenge for models that attempt to solve the task unsystematically.

Initial Results As a first application of the higher-difficulty task language, we train tRNN and tRNTN models on the adjusted training set, using the hyperparameters of the best models of Chapter 5, then evaluate their performance on an adjusted version of the original test set. The results of this experiment are reported in Table 6.2. We note first that the performance gap between the tRNN and tRNTN observed in our replication results persists under the new task setup. Further, performance of the two-way tRNN noticeably suffers (-1.9 percentage points), while the three-way tRNTN performance is mostly stable (-0.1 points). We conclude then that these initial results confirm our assumptions about an increased task difficulty, expressing themselves mainly in a performance degradation of the simpler model.

Nonetheless, the difference between the two models is still comparably small, and the continued performance of the tensor model near the ceiling confirms our doubts about the suitability of the original test data. Consequently, we do not want to claim that these results already establish a significant (qualitative) difference between the two architectures, i.e. answer our question whether the tensor model is better suited for a logical reasoning task. In the following section however, when applying the same data adjustment to a new set of test sentences, we observe a much clearer performance difference between the two models.

6.3 Evaluating Generalization by De Morgan's Laws

We want to test the model on sentences that are not simply 'unseen', but that additionally correspond to an actual generalization of the cases encountered during the training phase. Any definition of test cases that constitute such a proper generalization

[5] Average accuracy over 3 training runs with early stopping. Models are trained with the optimized hyperparameters of Section 5.1.

should be seen then in relation to the task the model is supposed to learn. In the context of this analysis, we need to determine then which examples of *inferences* in the natural logic of MacCartney (2009) can be considered as meaningful generalizations of the training examples.

Improved Generalization Test In a related investigation of the results of Bowman et al. (2015), Veldhoen and Zuidema (2017) use pairs of *equivalent* sentences to probe the models for their capacity to *generalize*. In particular, sentences that (under a classical interpretation) are equivalent by the duality of the existential and universal quantifier – also called the *De Morgan laws for quantifiers* – seem to pose a fitting test of generalization due to a crucial limitation of MacCartney's natural logic.

As mentioned in Section 2.1.2, MacCartney's natural logic is unable to derive syntactically the equivalence of sentences due to the De Morgan laws for predicate logic. The general form of the duality as it would be stated in a system of predicate logic is repeated below:

$$\forall x P(x) \equiv \neg[\exists \neg P(x)] \tag{6.1}$$

$$\exists x P(x) \equiv \neg[\forall \neg P(x)] \tag{6.2}$$

The inability of MacCartney's natural logic to derive the validity of these equivalences consequently influences the data used to train and test the models of Bowman et al. (2015), since the system generating this data is based on the logic. As a result, the models of Bowman et al. are not presented with any meaningful examples of equivalent sentences. While the training data contains sentences that are labeled *equivalent*, the only instances encountered during training are trivial equivalences, given by pairs of *identical* sentences.

By the natural logic definition of equivalence (see Table 6.3) the intended interpretation is given by extensional equality of two sets representing propositions.[6] Clearly, plain identity does not capture the intended general definition of equivalence of the system – yet, these are the only cases encountered by the models during training. Using equivalent sentences as test cases can therefore be seen as probing models for actual instances of generalization.

Before testing the models on these sentences however, we believe it is necessary to first derive which model prediction we can reasonably expect. Essentially, we will argue that a prediction of equivalence itself cannot be expected, but a weaker prediction that is nonetheless informative seems to be possible. The aspects taken into account here are the characteristics of the underlying logic, and of the machine learning approach, in which a general case has to be learned from individual examples.

[6] Technically, MacCartney's relations only seem to be defined explicitly for sets of individuals, i.e. for predicates of type ⟨e,t⟩. MacCartney (2009) remarks however that corresponding definitions for sentences (propositions) could be based on sets of models or possible worlds.

Symbol	Name	Shorthand	Interpretation
$\equiv$	equivalence	EQ	$X = Y$
$\sqsubset$	forward entailment (*excl.*)	FOR	$X \subset Y$
$\sqsupset$	reverse entailment (*excl.*)	REV	$X \supset Y$
$\wedge$	negation	NEG	$X \cap Y = \emptyset \wedge X \cup Y = \mathcal{U}$
$\mid$	alternation	ALT	$X \cap Y = \emptyset \wedge X \cup Y \neq \mathcal{U}$
$\smile$	cover	COV	$X \cap Y \neq \emptyset \wedge X \cup Y = \mathcal{U}$
#	independence	INDY	*all other cases*

Table 6.3: NatLog basic relations and their interpretation

6.3.1 Expected Model Predictions

Two questions need to be answered to determine what predictions the models we trained should ideally arrive at: *(i)* Do the De Morgan equivalences hold in the logical system implicitly defined by the training data?, *(ii)* If so, what is the maximally informative prediction the models could derive for these sentences?

De Morgan Laws and Natural Logic To answer the first question, recall that we cannot simply query MacCartney's natural logic itself, due to its inability to derive equivalences based on the De Morgan laws.[7] Therefore, any argument in favor or against has to proceed by external reasoning. However, our question is greatly simplified since the data of Bowman et al. was generated by a system that is only (loosely) based on MacCartney's natural logic, not by a full implementation of it. In particular, the data generation system entirely dispenses with the more complex elements of MacCartney's monotonicity apparatus, in particular, the rules for *compositional entailment*. Instead, the system of Bowman et al. only makes use of the static elements of the natural logic, i.e. simple inferences over content words and quantifiers.

The remaining fragment of the logic used to generate the task data however clearly seems to license a classical interpretation. In particular, *all* and *some* reduce to classical universal and existential quantification, and accordingly, we expect the De Morgan equivalences to hold within the data logic, and thus, in the logic (ideally) learned by the models. A final piece of evidence in support of the above claim stems from the preliminary results of an experiments in which the original data is reconstructed in a classical first-order language. Here, the equivalences can be derived by an automated theorem prover, suggesting that the task data itself licenses the De Morgan equivalences (Mathijs Mul, *personal communication*).

Equivalence and Entailment The question remains then which prediction we can expect of the models for the equivalent sentences presented after the training completed. Since the models encountered the *equivalence* relation only as the target label

[7] For a detailed argument why MacCartney's edit sequences (the system's basic proof calculus) cannot derive such equivalences, see (MacCartney, 2009, Section 6.5.4).

for *identical* sentences, we cannot expect any systematic relation between equivalent sentences and the equivalence label itself. Specifically, the equivalence label was *only* assigned to identical sentences, and identical sentences *only* appeared under this label. Hence, even a hypothetical model able to perfectly learn from data would be unable to connect these isolated instances of the equivalence *label* to any (potentially learned) *semantic* notion of equivalence.

Although we cannot expect the models to predict the equivalence of equivalent sentences by its intended symbol, we can nonetheless pursue our generalization test as originally intended. Under the classical definition, (logical) equivalence and entailment are directly related: Sentences ϕ and ψ are (logically) equivalent if and only if ϕ entails ψ and ψ entails ϕ. We can therefore expect the prediction that equivalent expressions stand in an *entailment* relation – assuming the models succeeded to learn a sufficiently general form of inference.

Exclusive Entailment and 'Definition by Data' A complication arises however due to the definition of entailment as *exclusive* or *strict* entailment in MacCartney's natural logic. We can see by the definitions of Table 6.3 that the forward entailment relation holds between (predicates) X and Y in case X is a proper subset of Y; similarly, for the definition of reverse entailment.[8] Entailment is defined in such a way, because MacCartney (2009) aims build an inference system in which the basic relations are *mutually exclusive*, to allow maximally informative reasoning over given sentence pairs. Consequently, the 'containment relations' of the system (equivalence, forward and reverse entailment) are mutually exclusive as well – it might seem therefore that our predicted predictions will not be derivable by our models after all.

At this point, we need to take a look at the actual training data, and which interpretation is licensed by it. Consider the following training example:

```
((all turtles) walk) ⊐ ((all reptiles) walk)
```

Note first that the data contains nouns that are intended to form a subclass of another noun, such as 'turtles' and 'reptiles', or 'warthogs' and 'mammals'. These classes and subclasses are used to learn (and test) lexical inferences under the basic relations of Table 6.3. The example above seems to conform then to an exclusive interpretation of entailment, or at least, does not contradict such an interpretation. Worlds in which all reptiles walk form a subset of worlds in which all turtles walk, as expressed by the reverse entailment label interpreted as a subset relation. Until now, without a symbolic definition of the target relation, the model might still be 'agnostic' regarding the question whether the subset relation that emerges during training is strict or not.

However, the data *only* contains examples of entailment that are based on sets of entities where one is a proper subset of the other (like "turtles/reptiles"), and thus, all examples point towards an exclusive interpretation. Without any counterexample (i.e. an example of inclusive entailment), and considering that the model is designed to fit

[8] As noted previously, an extension of the predicate definitions to the sentence case needs to be assumed, e.g. by a possible world interpretation.

the given data, it is most likely that an *exclusive* entailment relation will eventually emerge during training.[9]

As it turns out, one of the above assumptions is unwarranted however, as we noticed when analyzing the data in detail. Perhaps by an oversight of the authors, predicate 'turtles' happens to be the *only* subclass of predicate 'reptiles' in the actual data; similarly, for the other nouns. Hence, regardless of the (noun) names assigned to the input words, which suggest a strict subset relation via world knowledge, the 'subset' class of entities (turtles) is not guaranteed to be a *strict* subset of the larger class (reptiles). Extending the "fitting the given data" remark of the previous paragraph, we can now add "… and *only* the given data". In other words, it is unlikely that model training will fit weights to a strict entailment interpretation, since no corresponding data points exist.[10]

In consequence, while entailment might not be derivable from equivalence in natural logic due to the symbolic definition of an *exclusive* entailment relation, in the "definition by data" on which our models rely, an *inclusive* definition of entailment should emerge. It is easy to see then that the equivalent sentences we will use as a new test case should in fact license a prediction of entailment.

6.3.2 Prediction Results

In the previous sections, we argued why equivalent sentences seem to constitute a proper test of model generalization. We further explained why the maximally informative prediction we expect from models that learned to generalize should be *entailment*. After answering these preparatory questions, we can finally put the models to our new test.

Test Data The new test set consists of 160 distinct sentences. Nouns and verbs of the task language instantiate x and y in the following equivalences derived by the De Morgan laws:

$$\text{all } x\ y \equiv \text{no } x \text{ not } y \tag{6.3}$$

$$\text{all } x \text{ not } y \equiv \text{no } x\ y \tag{6.4}$$

$$\text{some } x\ y \equiv \text{not all } x \text{ not } y \tag{6.5}$$

$$\text{some } x \text{ not } y \equiv \text{not all } x\ y \tag{6.6}$$

The full set of 160 sentences results from repeating the inference of Eqs. (6.3) to (6.6) in reversed order. This duplication is motivated by the order sensitivity of input, and intended to fully test model predictions over both forward and reverse entailment.

---

[9] We should say that the model is likely to be fitted *mostly* to an exclusive interpretation, since some interpretation 'looseness' is enforced by the L^2 regularization term.

[10] While random moves of the training process towards fitting this more specific interpretation, and thus, less general parameter configuration should be countered by – once more – the impact of regularization.

	Softmax Output							Prediction (max)
	INDY	FOR	EQ	REV	NEG	COV	ALT	FOR+REV
30/90 tRNN	0	7	0	17	0	33	44	28
25/75 tRNTN	0	21	0	76	0	1	2	100

Table 6.4: Equivalence test, aggregate results (%). Original data.[11]

The following example shows a single test pattern instantiated by lexical items:

`((all pets) (not growl))` ≡ `((no pets) growl)`

Note that in the equations and the example above, we continue to use the original (non-compositional) quantifiers of the original data of Bowman et al. (2015). We will however test on both types of data, and accordingly adjust the test cases for models trained on the CompQuant data (e.g. "no x y" → "not some x y").

Original Data Results Table 6.4 reports the results of the new equivalence test on models that were trained on the original data of Bowman et al. (2015). The first seven columns (INDY ... ALT) report the output values of the softmax layer for each relation, as an average over all tested sentences. These columns can be interpreted as a probability distribution over the seven basic relations, for the 160 test cases in aggregate. The rightmost column (FOR+REV) reports the model predictions of forward *or* reverse entailment for a given test sentence, as the average over all 160 cases.[12] In other words, a model prediction of either of the two entailment relations is counted as a correct classification here.[13]

We can see in the final column that the *matrix model* (tRNN) classified equivalent sentences pairs correctly (in the sense of Section 6.3.1), by a prediction of either forward or reverse entailment, in 28% of the cases (44 of 160). Therefore, the model performs almost exactly at the random baseline.[14] By the same criterion, the *tensor model* (tRNTN) correctly predicts entailment in 100% of the cases (160 of 160).

Columns 1 to 7 show that the tRNN models assign most of the 'mass' of softmax probabilities to *alternation* (ALT) and *cover* (COV). Neither of these two relations follows from equivalence. Alternation, the natural logic relation to describe "non-exhaustive negation", even amounts to a prediction in clear contradiction to the intended interpretation. The matrix model, despite classifying by entailment only at

[11] Results report average of 3 models from Chapter 5, trained on `f1` data set. Consistency across models will be discussed later in the section. Basic relations are abbreviated here by the shorthand of Table 6.3.

[12] Note that columns FOR and REV do not exactly sum to the value of column FOR+REV, since the former two directly report averages of softmax probabilities, while the latter reports the average of actual model predictions, i.e. the softmax column/relation selected by max.

[13] Motivation for this correctness criterion is provided later in the section.

[14] Prediction of either of 2 entailments counts as success, of 7 relations in total, for a baseline of ~29%.

	Softmax Output							Prediction (max)
	INDY	FOR	EQ	REV	NEG	COV	ALT	FOR+REV
30/90 tRNN	0	1	0	4	0	46	49	0
25/75 tRNTN	0	10	0	76	0	7	7	96

Table 6.5: Equivalence test, aggregate results (%). CompQuant data.

a random baseline, assigns softmax 'weight' above baseline to the two entailment relations (though still far less than the tensor model). However, the cover and alternation probabilities outweigh those of the entailments by far, and the latter are therefore rarely selected as the most likely relations in the final classification step.

In contrast, judging by the (aggregate) assignment of probabilities, the tRNTN correctly interprets equivalence pairs as forms of entailment, with nearly all of the mass centered around forward and reverse entailment. Reverse entailment outweighs forward entailment by more than 3:1, despite input pairs being balanced with respect to entailment direction. This observation will be discussed further later in the section.

Finally, note that neither the matrix nor the tensor model assigns any significant weight to the equivalence label itself. This outcome should not be surprising, as we argued before, since the suggested interpretation of the equivalence label by the training data effectively was *input sequence identity*. Identity, however, is a highly isolated, easily learned type of input, and therefore unlikely to be mistaken for any other input configuration, including sentences in an entailment relation.[15] Accordingly, the "identity equivalence" relation learned by the models is given nearly zero prediction weight by the models when presented with cases of actual equivalence – correctly so, it should be noted, given how the relation was defined by the data.

CompQuant Data Results Table 6.5 reports our results of evaluating the models trained on CompQuant data by the equivalence test set. The experimental setup is identical to the previous one, except for an adjustment of the test set, replacing the quantifiers to match those of the training data.

The combined entailment prediction average of the tRNN drops, from the previous baseline performance of 28% to 0%. Of the 160 sentences we tested, the matrix model did not predict any type of entailment for any of the sentences. The aggregate softmax mass shifted from a small, but somewhat relevant concentration around reverse entailment in the previous experiment to the cover relation in the current experiment.

The tensor model performance is largely unchanged, although its performance (measured as prediction of entailments) slightly drops under the new data regime. Entailment predictions were made on all sentences in the previous experiment, while

[15] 'Isolated', because identical pairs do not license further relational inferences. "Easily learned", because trained independently, binary classification of input as identical/non-identical would be a trivial task for the networks – for example, by fitting weights to perform subtraction, with a classification threshold of 0.

they now occur in 96% of all cases (153 of 160). The 'misclassification' cases of the tensor are spread across the inference patterns of Eqs. (6.3) to (6.6), and we cannot find a systematic pattern in their distribution. We believe therefore that the most likely explanation for the performance decline is due a general increase in task difficulty, approximately in line with our initial results using the CompQuant data in Section 6.2. One noticeable change however is the shift of 'mass' from forward entailment to cover and alternation, when comparing the tensor model softmax output of the previous experiment with the current one. We will discuss a possible explanation of this shift that seems to take place in both types of models in the next subsection.

While the performance of the tRNTN slightly suffered, the performance of the tRNN declined drastically. We mentioned previously that the CompQuant data would allow us to exhibit a clear performance difference between the matrix and the tensor models. Considering the gap in their ability to predict the intended entailment relation – 'never', for the matrix model, 'almost always', for the tensor model – we believe this claim is justified. By the previous reasoning about equivalent sentences as a plausible test of generalization, this would further imply then a major difference in *generalization* performance on the task between these two architectures.

6.3.3 Results in Detail

In the following, we will address some of the open questions related to our results of the generalization experiments.

Differences by Inference Pattern It seems that a detailed analysis of prediction differences depending on the inference pattern could be interesting from a logical point of view. However, the results provide little room for such a distinction by logical forms. The tensor model correctly predicted entailment for nearly every test case, and thereby, across all inference patterns. The few misclassifications of the model in the second experiment are spread across inference patterns without any apparent systematicity. The matrix model on the other hand yielded zero entailment predictions in the second experiment, and we cannot distinguish non-occurrences by case.

The single exception is the result of the matrix model in the first experiment. Here, the tRNN predicted entailments in aggregate near a random baseline. However, most of these were made on a single inference pattern, the equivalence of Eq. (6.4), for sentences of the form: all x not $y \equiv$ no x y. We currently cannot say with certainty whether the effect has a systematic reason, or is a random artifact of the training process. Considering however that these entailment predictions completely vanished in the tRNN when using the higher-difficult data, we tend to favor the latter explanation, i.e. an effect without a systematic cause.

Results Variability Our results here use three models (i.e. training runs) per model architecture. The predictions were highly consistent, in fact, nearly identical across runs. While puzzling at first perhaps, we believe the most likely reason it is a stabilization effect of the final prediction steps: a 'squashing' of values by the softmax

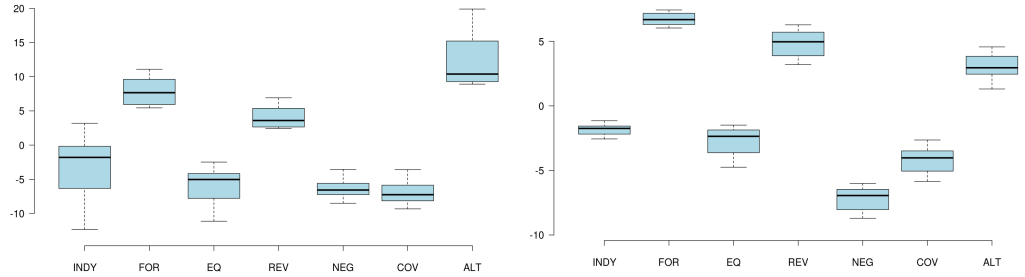

Figure 6.1: Softmax layer input. Pattern: no x $y \equiv$ all x not y.
tRNN (left), tRNTN (right).[16]

layer activation function, followed by application of the max function. Minor (and perhaps even moderate) differences are leveled by these steps, and models that are 'internally' different can appear indistinguishable on the outside.

To be sure then that our results are not merely a side effect of some leveling, we also inspected the (weighted) softmax input, i.e. the state of the model before those steps apply. Fig. 6.1 shows the input to the softmax layer of one model instance, over all test sentences based on the inference pattern of Eq. (6.4). We can see then that, in fact, the contrast between the two model types seems less stark than in the prediction outcomes. For example, both architectures assign most of the prediction weight to the same three relations here (FOR, REV, ALT). However, even at this incomplete state of the prediction process, some clear differences emerge. The median values of the two 'correct' labels (FOR, REV) are above the median value of the 'wrong' label (ALT) in the tensor network, while the opposite is the case for the matrix model. Further, the minimum softmax input values of the correct labels are roughly the same as the maximum values of the wrong label in the tRNTN, while the reversed case holds in the tRNN. In other words, the 'raw' model predictions match our previous results based on the final model predictions, just in a less sharply contrasted form.

Data Imbalance We mentioned that we observe a 'shift' of predictive weight, from entailment to cover, when comparing the first and the second experiment. We believe this effect is likely linked to the distribution of relation examples in the data. The relations independence, cover and alternation make up about 70% of the training examples (INDY: 30%, COV: 21%, ALT: 19%), while the other four relations are encountered substantially less frequently during training. This underlying 'imbalance' of relation examples could then explain two observations: the tendency of both models to assign significant predictive weight to cover and alternation, and the above weight 'shift'. Both could plausibly be seen as expressions of a *prior* given by the training data. For example, upon increasing task difficulty, the tRNN predictions shift back to the prior that was learned during training.

[16] Box plots of 1st/3rd quartile, median, and minimum/maximum of values.

We note however that this explanation is incomplete: while the data prior might account for the predictive weight of cover and alternation, it seems to be contradicted by the observation that independence is assigned only little weight in the tests, while being the most frequent relation encountered during training. As a possible hypothesis, we can perhaps suggest that independence is somewhat easier to 'isolate' for the models. While other relations can overlap to a degree in their 'meaning' (if learned only incompletely), the one relation expressing "absence of any informative relation" seems to stand out somewhat.

Direction of Entailment In Section 6.3.1 we gave an account of the reasons why a prediction of entailment is the most informative answer we can expect from the models, given the limitations of the training data. In fact, a prediction of entailment in both directions for a single equivalence test case would amount to a prediction of equivalence in everything but the name. However, in our experiments, we accepted the less informative answer of a single entailment prediction.

The weaker correctness criterion is justified, because the definition of model prediction, and the respective training objective of our models did not permit the stronger prediction. Note that model predictions are defined here as the *single* most probable relation for a pair of sentences, and cost was given by negative log-likelihood of the correct label. By this cost function, only model output over the target relation directly influences cost, while the softmax distribution over the remaining labels does not enter the calculation. Therefore, models were not trained for two relation predictions over one sentence, and the weaker prediction is the best possible outcome for these models.

An extension of the models to predict multiple relations would be an interesting direction for future work. This can be achieved by replacing the current cost by general negative log-likelihood, by which models learn a probability distribution over multiple classes.[17] Doing so would allow us to verify if the models learned the full 'meaning' of equivalence, i.e. whether a distribution emerges in which forward and reverse entailment are the two most likely relations given a pair of equivalent sentences.[18]

6.4 Visualization of Model Representations

The equivalence tests of the previous section seem to show a significant difference in generalization performance between the simpler two-way tRNN models and the more complex three-way tRNTN models. While interesting as an observation, we would ideally like to understand *how* this difference comes about. For this reason, we now turn to the inspection of the model internal representation, by using dimensionality reduction techniques on parts of the model.

[17] Both cost functions were defined in Section 4.2.

[18] We would however have to replace the natural logic relations then, or at least, adjust them for our purposes, since they are originally defined to be mutually exclusive.

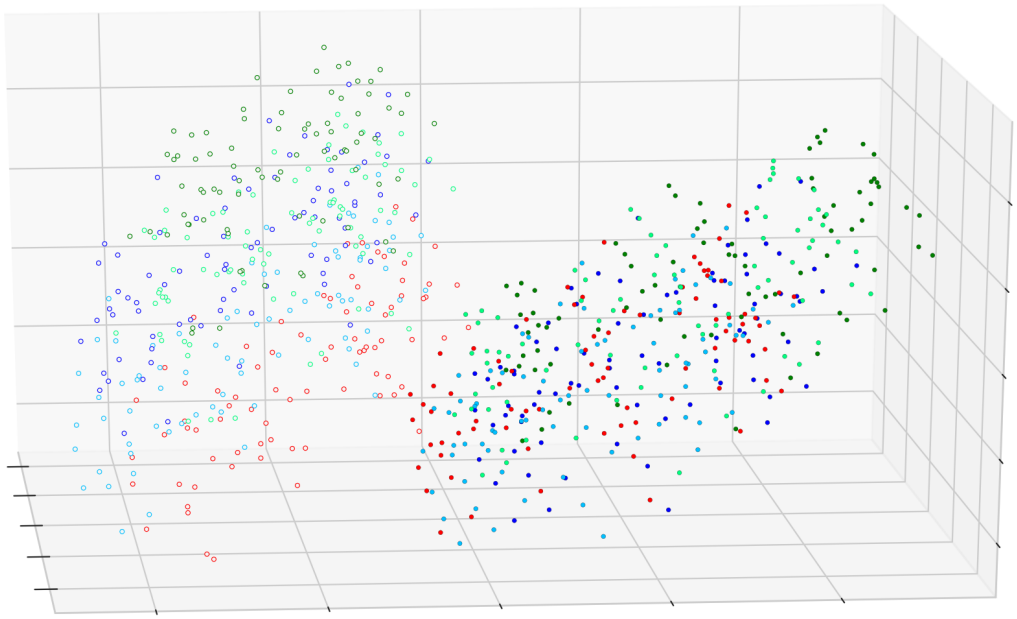

Figure 6.2: Tensor model.
Hyperplane separation of quantifiers and their negation

Visualization Method We tested several different techniques to inspect the model representation of the processed sentences. The best results were achieved by using *principal component analysis* (PCA), which takes a set of correlated values and turns them into a set of linearly uncorrelated values by an orthogonal transformation. Intuitively, PCA can be thought of as a rotation of the d-dimensional coordinate system in which our data lives such that in the new coordinate system, the first dimension (principal component 1) explains most of the data variance, the second dimension explains most of the variance *independent of the first component*, and so on.

Visualization by PCA was preferable over other methods, because we intended to explore the model internals without major distortions or loss of crucial information. The advantage of a PCA analysis is that the *global* structure of our model representations is preserved, in particular, how these models organize *sentences* they learned to classify.[19] Other popular visualization (and dimensionality reduction) techniques, such as t-SNE, were less suitable for this task. While t-SNE produces precise local 'clusters' of related observations, the global arrangement of these clusters in the visualization coordinate system is not representative of the global organization of the data points these clusters contain within the *original* coordinate system of our models.

Quantifier Negation Fig. 6.2 is one of the clearest depictions of a systematic representation of quantifier negation in the tensor model that we were able to extract.

[19] By 'global' we mean data points that are far apart in the representation space constructed by the model during training.

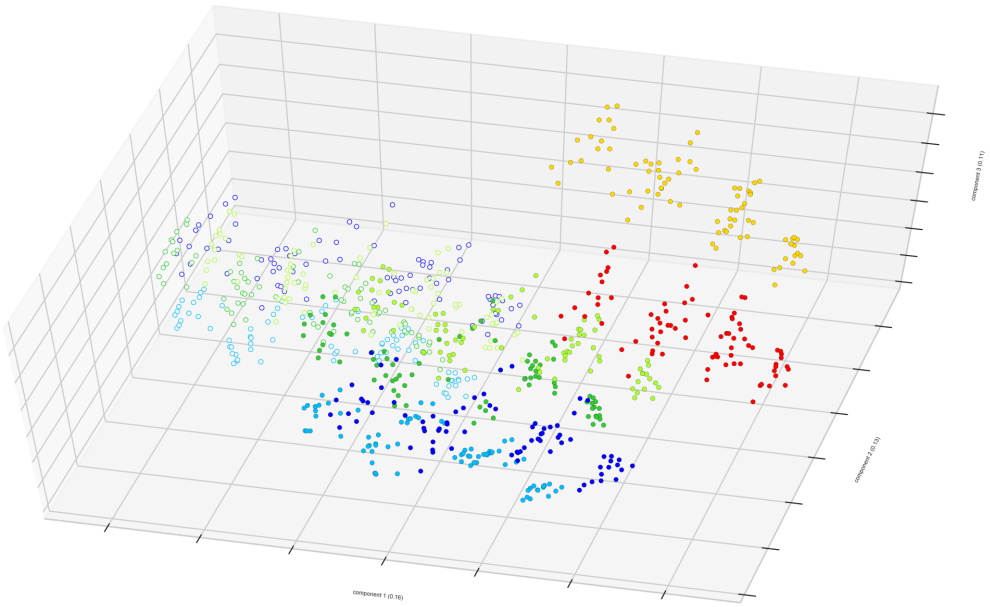

Figure 6.3: Matrix model.
Organization inside quantifiers along nouns

Plotted are components 1/2/4 of a single run of the tensor model on the CompQuant data. Coloring of the data points is chosen manually to mark the classes that underly the observed spatial separation. Different colors mark the different quantifiers, with solid circles for the non-negated quantifiers and transparent circles for their negation. We see that, without exception, non-negated quantifiers are on one side of a separating hyperplane, while their negations are on the other side.

In contrast, Fig. 6.3 depicts a similar plot for the matrix model. Across model runs, we found no evidence that the matrix model found a "negation hyperplane" similar to that of the tensor model. Instead, organization of Fig. 6.3 appears to proceed along different *nouns*, i.e. each quantifier cluster contains an organization of all nouns encountered within the scope of the quantifier. This observation appears to conform to the observation of Veldhoen and Zuidema (2017), who observe that the matrix model only learns many "local approximations" or clusters. While there are clear signs of systematicity *inside* these clusters, they find no evidence of a global organization in the matrix models.

Noun Negation Fig. 6.4 shows another plot of the tensor representations, using a different component configuration, and coloring by nouns (e.g. red for 'mammal'). As before, solid vs. transparent mark non-negation and negation. In contrast to the case of quantifiers, here, both negated and non-negated forms appear on the same side of a separating line. However, we also observe that a diagonal displacement separates the negated and non-negated nouns on each side, and it is therefore possibly that we simply lack one additional principal component to depict another 'perfect' separation, as in the previous case of quantifiers.

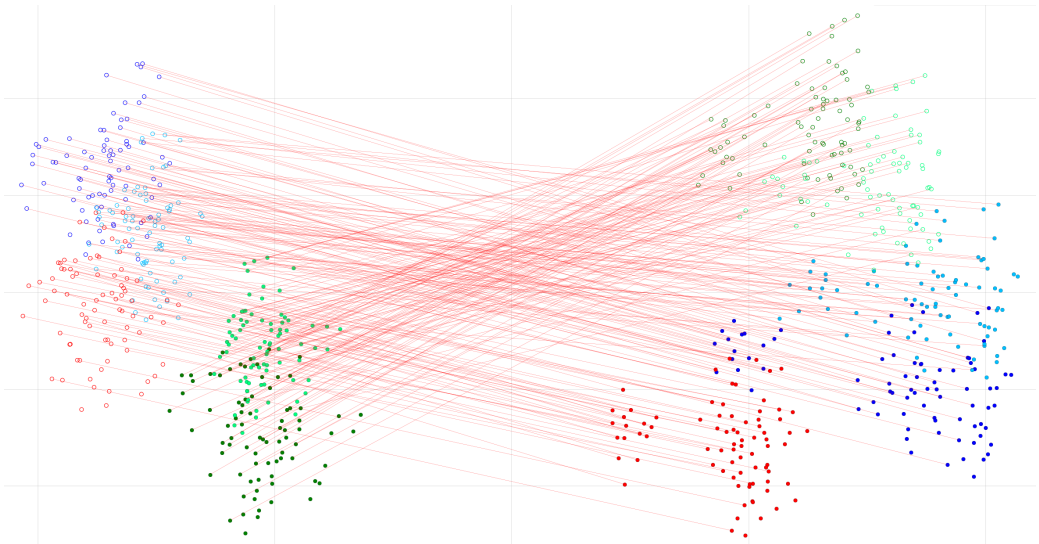

Figure 6.4: Tensor model. Noun negation, all.

Verb Negation In Fig. 6.5 depicts the organization of verbs and their negation in the tensor model, showing the separation of verb 'growl' and its negation. 'Growl' (solid red) and "not growl" (hollow red) are separated by a horizontal line of separation. Our results indicate that verb negation is the least 'general' representation in the tensor model: while each verb and its negation are clearly separated, we could not find evidence that the tensor model found a *general* representation that would capture the negation of *all* verbs.

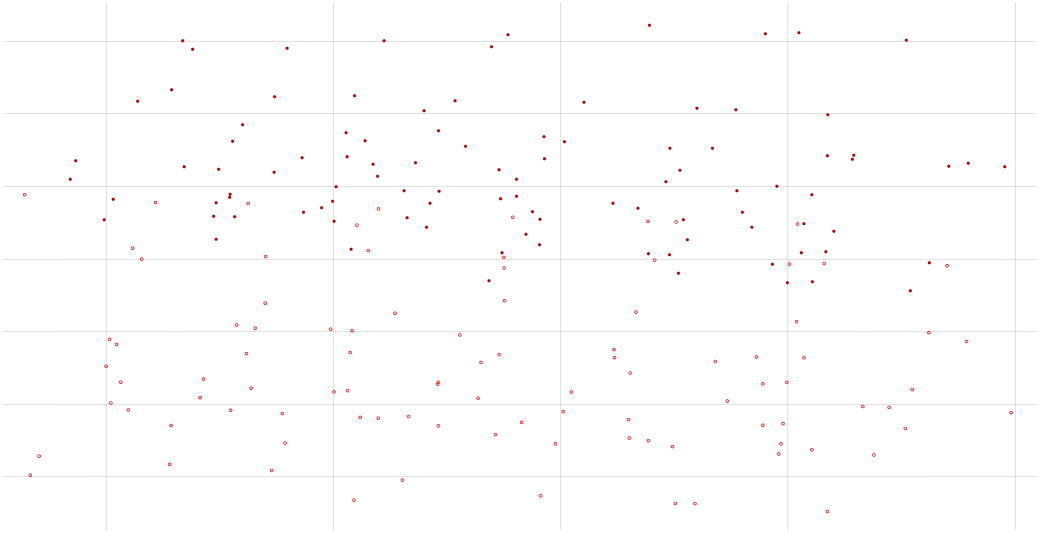

Figure 6.5: Tensor model. Negation of verb 'growl'.

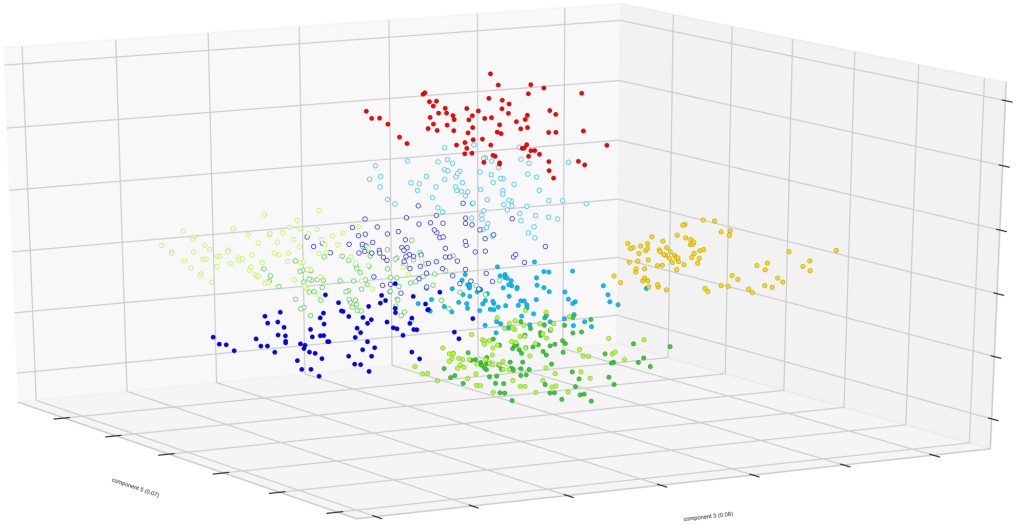

Figure 6.6: Tensor model.
Quantifiers ordered by numerosity (view 1)

Quantifier Ordering Figs. 6.6 and 6.7 depict the organization of quantifiers in the tensor model, colored here to highlight that the model appears to order quantifiers based on their approximate *numerosity*. Consider here that the (logical) model learned by the (network) model is likely to be interpreted as finite. We could therefore consider an ordering of the quantifiers along some range of the finitely many values they could take in the model. We see then that 'all' (blue), 'three' (dark green), 'two' (light green) and 'no' (red) are arranged approximately corresponding to their implicit numerosity, including their respective negations. Note however that 'some' (yellow) appears to be fall outside of this ordering, for reasons that remain unclear.

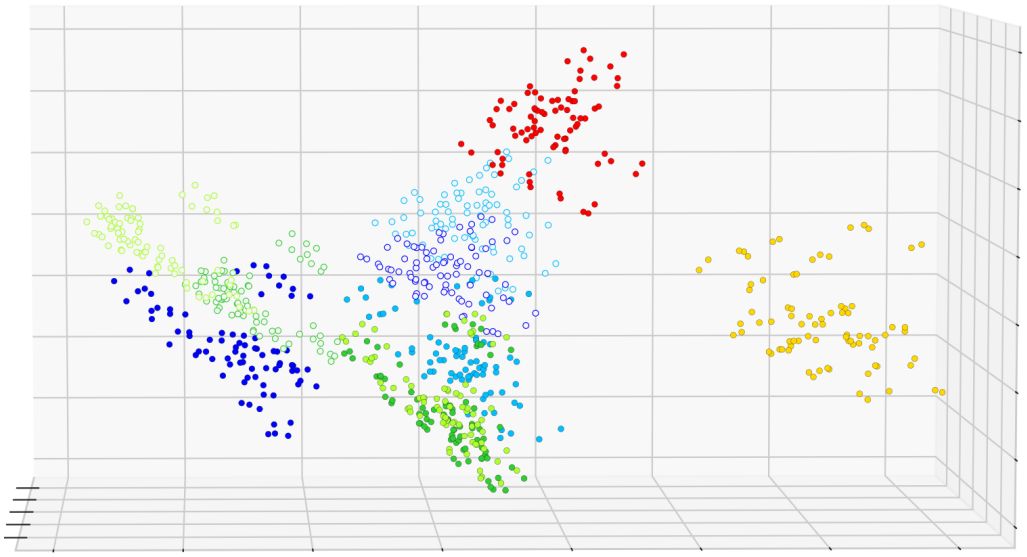

Figure 6.7: Tensor model.
Quantifiers ordered by numerosity (view 2)

6.5 Discussion

In this section, we suggest a link between the previous results, and conclude by a brief discussion of possible extensions of the models and experiments.

6.5.1 Connecting the Results

Given the extracted model representations of the previous section, we can now suggest an explanation for the prediction results of Section 6.3. We suggest that the relation between the different results can be summarized as follows:

(*i*) In order to correctly derive the De Morgan equivalences, a sufficiently general representation of negation is required.

(*ii*) Linear separation in the tensor model by a single axis (nouns, verbs) or a single hyperplane (quantifiers) constitutes a more general representation than local clusters (matrix).

(*iii*) Higher expressiveness of the tensor allows representing negation by a single general operation, which could explain the observed linear organization of negated expressions.

The following paragraphs elaborate on these three key points.

Derivation of Equivalences A correct (approximative) derivation of the equivalences following from the De Morgan law seems to require a correct, and sufficiently general representation of negation by the models. A model that failed to learn a general representation for the negation of verbs, nouns and quantifiers cannot systematically relate expressions and their negation, and therefore, is unlikely to be unable to generalize well enough to predict an informative relation for the unseen cases of equivalent sentences. An insufficiently general representation of negation then leads to the prediction behavior we observed for the matrix model in our equivalence tests, and fits the merely local organization we found in the PCA analysis of the matrix model.

Systematic Representation of Negation We saw in the previous section that the tensor model found linearly separated all word classes, i.e. nouns, verbs and quantifiers. While the matrix model is able to linearize the data as well, it appears to do so by gathering distinct input patterns inside local, separated clusters. This does however not correspond to a general, model-wide representation of negation, in contrast to the global separation of negated expressions in the tRNTN.

Negation by Slices A plausible explanation *why* the tensor model was able to learn general representation of negation, resulting in a global separation of components and their negation, is offered by the possibility of encoding the negation operation as an individual *slice* of the higher-order tensor.

In general composition of a third-order tensor with a single word (vector) results in a single tensor slice, encoding the linear transformation associated with the word. Consider then the negation symbol ('not'), when multiplied with the trained, can 'select' a single matrix as well. The linear transformation associated with this matrix could then be trained by the model perform negation – as a global negation operation applying to all input cases.

An example of an analytically derived negation matrix can be seen in (Coecke et al., 2010; Grefenstette, 2013). Here negation is a represented by a *swap matrix*, which corresponds to a 90° rotated identity matrix . It is unlikely that the global, task-trained composition function employed in the models of this section learns the exact high-dimensional analog of such a swap matrix, due to the noise inherent in the data. However, the possibility of *representing* a similar negation mapping is formally guaranteed in a tRNTN model, by association of a tensor slice with a word vector for 'not'.

Generalization to Novel Data A model that is able to employ a unique, globally applied negation operation can plausibly be said to have learned a more 'general' form of negation than model that relies on local cluster of negated and non-negated expressions. If the tensor did in fact learn such a negation matrix, it could serve as an explanation for the global linear separation we observed in the tensor, but not the matrix model. A global, and thereby, general separation of negation however appears to be a necessary condition find an informative predictions on unseen sentences (i.e. that have not been 'memorized' before), which are systematically related to another unseen sentence, in a way that requires negation to be applied to the components of the expression.

6.5.2 Limitations and Future Work

An important open question at the conclusion of our analysis is whether the results of this chapter will extend to a larger task. Given that the results of this chapter are derived in the context of a synthetic language, it is not certain how generally applicable they are. Ultimately, to answer this question we should repeat our experiments on large-scale natural language data. Doing so however requires employing a computationally more efficient model, given that full-rank tensors are intractable for higher-dimensional tasks.

Pointwise Multiplicative Layer To tackle this problem, we adjusted the tRNTN architecture to make it computationally tractable. We replace the full-rank tensor of the tRNTN by a rank-one tensor, resulting in an architecture based on what we suggest to call a *pointwise multiplicative layer* (PML). This rank-one replacement can be achieved by replacing the tensor product of Eq. (4.5) by the Hadamard (or pointwise) product, resulting in:

$$z = f(V(x \odot y) + W[x\,;y] + b)$$

Note that V here is a simple matrix, unlike the higher-order tensor $\mathcal{V}$ before. We essentially enrich the normal linear layer of a tRNN by a linear combination of (pointwise) multiplicative interactions between co-indexed components.

We trained the new architecture on the tasks of the previous sections, and observed that the new architecture was able to converge to an optimal solution much faster than any other model of our experiments. Our results are likely to be related to the reported behavior of the MI-RNN architecture of Wu et al. (2016), as discussed in Section 3.4.3. This similarity in convergence behavior is unsurprising, since the model of Wu et al. is closely related to our PML model – essentially, the MI-RNN is the recurrent network analog of our tree-structured model.

While the convergence behavior of the PML appears to be extremely promising, another experiment we performed hinted at a crucial limitation. When using the equivalence test set to evaluate the PML model, we discovered that its performance on our generalization test based on entailment predictions was similar to the results of the (matrix-based) tRNN.

It seems then that the step from a full-rank tensor to the other end of complexity, a rank-one tensor, was perhaps too large, and we lost the generalization capacity of the full-rank tensor in the process. As a next step, it might be worthwhile then to employ the mRNN architecture of Sutskever et al. (2011), since the expressiveness of its tensor can be reduced gradually by a parameter. Ideally, a range of tests using different expressiveness parameters on the equivalence test set would then reveal at which point of the tensor reduction the generalization capacity of the full-rank tensor gives way to the less general behavior of the matrix model.

Chapter 7

Conclusion

At the beginning of our investigation, we resolved to describe insights from symbolic theories of semantics that could, perhaps even *should* be incorporated into distributional models of language. To this end, Chapter 2 not only provided the technical background for our later analysis, but also offered suggestions which aspects of symbolic semantics appear to be relevant for distributional models. Similarly, Chapter 3 investigated which of the established methods and architectures used in machine learning models can be seen as (partially) implementing some of the aspects identified in the previous chapter. We arrived at the conclusion that higher-order tensors seem to be a key element in creating distributional models of semantics that are able to sufficiently account for *compositionality*.

We then put our hypothesis to the test in Chapter 6, where we found a significant difference in the generalization capacity between a conventional matrix-based neural network model and a model employing a higher-order tensor. These differences emerged when models were evaluated in their ability to predict the semantic relations between a type of sentences that they had never encountered during training, as a true test case of *generalization*. We finally suggested that these differences in the prediction results likely result from a more general representation of negation learned by the tensor model. As a technical reason, we suggested that the increased expressiveness of the tensor-based composition function allowed the tensor model to represent negation as a single, global operation over the entire data, thus avoiding the locally organized, insufficiently general representation learned by the less expressive matrix-based models.

Part III

Appendices

Appendix A

Linear Algebra and Tensor Primer

In this section we will introduce some basic concepts of linear algebra. We neither aim for completeness, nor for full mathematical generality of definitions, taking instead an applied perspective along the conventions of the machine learning literature. We refer interested readers to Strang (2009) for a thorough introduction to linear algebra, and to Hackbusch (2012) for a mathematical textbook on tensor calculus, or Kolda and Bader (2009) for a shorter, application-oriented introduction to tensors.

A.0.1 Scalars, Vectors and Vector Spaces

We begin by introducing vectors, vector spaces, and operations over them, as well as providing some basic intuitions for these concepts.

Vectors Nearly all (non-symbolic) semantic models we will discuss are so-called *vector space models*. A d-dimensional *vector space* over the field of real numbers, $\mathbb{R}^d$, is the collection of vectors of d dimensions, the elements of the space, together with operations (vector) addition and (scalar) multiplication. A vector x can be thought of as an array of d (real) numbers, called *scalars*. If seen as an array, a single index is needed to refer to each element of a vector.

Alternatively, vectors are sometimes described in the form of 'arrows' inside some coordinate system, indicating the *direction* (angle) and *magnitude* (length) of a vector, leading to a *point* (coordinates) in the space. In principle, this requires declaring the *basis* of our space, a set of basis vectors by which all other elements of the space can be uniquely expressed, as a linear combination of these basis vectors. In many applications however, mentioning the basis is omitted, with the assumption that models use the standard (natural) basis of Euclidean spaces. Given a basis (either implicit or explicit), the d numerical values of a vector are often referred to as the *components* of that vector.

We generally write α, β for scalars, and lowercase Latin letters for vectors, e.g. $x \in \mathbb{R}^d$, for d-dimensional vector x. We refer to the i-th component of vector x by writing $x_{(i)}$ or x_i. In some cases, we will however use the same notation to index vectors themselves, for example, when saying that vector z_i is the output of a function

given input x_i, which itself is the i-th element of a *sequence* of vectors $\boldsymbol{x}$. When explicitly listing the components of a vector, we will write them as an array of scalar values or variables, enclosed by square brackets. To mark that a word, e.g. 'car', is represented by some vector, we write v_{car}.

Vector Operations *Vector addition* of two vectors $x, y \in \mathbb{R}^d$, $x + y$, is a vector of the same dimension as the input, given by:

$$\begin{bmatrix} x_1 \\ \vdots \\ x_d \end{bmatrix} + \begin{bmatrix} y_1 \\ \vdots \\ y_d \end{bmatrix} = \begin{bmatrix} x_1 + y_1 \\ \vdots \\ x_d + y_d \end{bmatrix}$$

We call this type of function *element-wise* or *pointwise*, i.e. applying an operation to the components of the two arguments with matching indices. Correspondingly, *vector subtraction*, $x - y$, is defined as the pointwise subtraction of y from x.

Scalar multiplication of a scalar $\alpha \in \mathbb{R}$ with vector $x \in \mathbb{R}^d$, written: αx, is a vector of the same dimension as x, given by:

$$\alpha \begin{bmatrix} x_1 \\ \vdots \\ x_d \end{bmatrix} = \begin{bmatrix} \alpha x_1 \\ \vdots \\ \alpha x_d \end{bmatrix}$$

It will often useful to measure the *distance* between vectors, or, related, their *similarity*. One frequently encountered metric is *cosine similarity*, defined as the cosine of the angle between two vectors:

$$\frac{\sum_i x_i y_i}{\sqrt{\sum_i x_i x_i}\sqrt{\sum_i y_i y_i}}$$

A.0.2 Matrices and Linear Maps

Next, we introduce matrices, matrix operations and linear maps, and relate them to machine learning models.

Matrices As noted above, we can equate vectors with single index arrays. We now consider an array of two indices, a *matrix*, which can be viewed as a table of values. We will use uppercase letters to refer to matrices, e.g. W. We refer to the components of a matrix by two indices, $W_{i,j}$ or $W_{(i,j)}$, usually using the first index to refer to the *rows* of the matrix, i.e. the horizontally arranged elements at an index, and the second index to refer to its *columns*, i.e. the vertically arranged elements at an index.[1]

We will refer to a matrix W with m rows and n columns as an $m \times n$ matrix, or say that matrix W is of *shape* (m, n). We note now that a vector can be described as a

[1] This order of indexing is a matter of convention and sometimes differs.

matrix with one of its two indices fixed, i.e. a d-dimensional vector can be expressed as a $d \times 1$ matrix, a *column vector*, or as a $1 \times d$ matrix, a *row vector*. We list the elements of a matrix in the form an array with two indices, i.e. a *table* of values, enclosed by square brackets, writing $w_{i,j}$ for the (scalar) component of W at (i, j).

Matrix Operations The *transpose* of $m \times n$ matrix W, written as W^T, yields a $n \times m$ matrix, given by: $(W^\mathsf{T})_{i,j} = W_{j,i}$. The operation corresponds to a reflection of the elements of the original matrix across its *main diagonal* – the latter being the sequence of all entries of a matrix $U_{x,y}$ such that $x = y$.

The vector operations described in the previous paragraph can be easily extended to operations on matrices of identical shapes, as the pointwise addition, subtraction of the elements of the matrix. Similarly, scalar multiplication multiplies all matrix entries by a scalar value. We can also define a pointwise multiplication operation on matrices, the Hadamard or element-wise product. The *Hadamard product* of matrices V, W of identical shape is then:

$$(V \odot W)_{i,j} = V_{i,j} W_{i,j}$$

We now introduce *matrix multiplication*, the central operation of many models we will discuss here. For $m \times n$ matrix V and $n \times p$ matrix W, we write $U = VW$ for the matrix multiplication of V and W, where $m \times p$ matrix U is given by:

$$U_{i,j} = \sum_{1 \le k \le n} V_{i,k} \, W_{k,j}$$

By this definition, each entry of the matrix product VW is the sum of entries across a row of V multiplied by entries across a column of W.

A related operation is the *dot product* or *inner product*. The *dot product* of column vectors $v, w \in \mathbb{R}^d$, written $v \cdot w$, is equivalent to the matrix multiplication of $1 \times d$ row vector v and $d \times 1$ column vector w, or: $v \cdot w = v^\mathsf{T} w$. For vectors v, w, we will often write vw (without operator) for their dot product, $v \cdot w$.

Linear Maps A *linear map* (or linear transformation) is a function f between vector spaces V, W such that for any vectors x, y and scalar α, it holds:

$$f(x + y) = f(x) + f(y)$$
$$f(\alpha x) = \alpha f(x)$$

In other words, a linear map is a function preserving addition and (scalar) multiplication, as defined above. In finite-dimensional spaces, such as those of (real world) models of machine learning, and under the choice of a basis, matrices and linear maps correspond to each other: every linear map can be represented by a finite matrix, and every matrix describes a linear map. In particular, if W is a $n \times m$ matrix, then the function given by matrix multiplication of W with an m-dimensional vector x, Wx, defines a linear function from space $\mathbb{R}^m$ to space $\mathbb{R}^n$, commonly written as $W \in \mathbb{R}^{m \times n}$.

We write $V \circ W$ for the *composition* of two linear maps V and W, which is a linear map as well. In finite-dimensional cases, the composition of linear maps is given by matrix multiplication.

Network Layers As an example of the concepts introduced so far, we now briefly consider the layers of a (feedforward) neural network. Fully connected layers of these networks correspond to linear maps – ignoring momentarily the bias vector and nonlinear activation function usually contained as well.

Consider the n-th layer of a neural network, where the layer input is a vector x of d_{in} dimensions, $x \in \mathbb{R}^{d_{\text{in}}}$, and the layer output is vector $y \in \mathbb{R}^{d_{\text{out}}}$. If we assume that the trainable parameters of layer n form a $d_{\text{out}} \times d_{\text{in}}$ matrix W^n, then the scalar components of this matrix (also called its *weights*) parametrize a linear map from $\mathbb{R}^{d_{\text{in}}}$ to $\mathbb{R}^{d_{\text{out}}}$, given by $W^n x$. Expressing $x \in \mathbb{R}^{d_{\text{in}}}$ in the form of a $d_{\text{in}} \times 1$ matrix, the composition of W^n with vector x proceeds by matrix multiplication. The resulting vector, $W^n x = y \in \mathbb{R}^{d_{\text{out}}}$, is the linear transformation of input x by W^n, the trainable parameters of layer n.

Most network layers also contain a constant (but trainable) *bias vector*, added to the output vector y by vector addition. A linear transformation 'shifted' in such a way by a constant vector is called an *affine transformation*. Finally, most networks pass the resulting vector element-wise through a nonlinear function, called the *nonlinearity* of the layer or network.

The trainable weights of a single neural network layer are one example of a linear map. Other examples are certain elementary vector transformations, such as *rotations*, *reflections*, or *projections*.

A.0.3 Tensors and Multilinear Maps

Next, we introduce higher-order tensors, showing three different but equivalent ways to define them. Afterwards, we explain the dimensionality problem related to higher-order tensors, and briefly sketch possible solutions to it.

Multidimensional Arrays So far we encountered arrays of up to two indices, i.e. matrices. Generalizing, one can consider arrays with n indices. In general, we refer to a 'multidimensional' array (in the sense of arrays with *multiple indices*) using n indices as an *n-th order tensor*, or tensor of order or rank n.[2] Under this view, a scalar is a tensor of order zero, a vector a first-order tensor, and a matrix a second-order tensor. Tensors of order three or higher are usually referred to as *higher-order* tensors. We will use calligraphic uppercase letters to denote higher-order tensors, e.g. $\mathcal{V}, \mathcal{W}$.

Third-Order Tensor In analogy to the view of a matrix as a table of values with two indices, a *third-order tensor* can be seen as a 'cube' of entries, resulting from adding a third index. As we will see, third-order tensors also happen to be the most relevant type of higher-order tensors in the context of the models we discuss here. Extending the row/column terminology of matrices, the entries of a tensor along one of its indices are referred to as *fibers*. Fixing all but two indices of a higher-order tensor results in a "two-dimensional" segment of a tensor called a (tensor) *slice*. There are

[2] We avoid the term *rank* since it carries another meaning in the context of factorization.

three distinct ways to slice a third-order tensor: horizontally, laterally and frontally, each representing a different way to divide the three-index array (tensor) into its two-index components (matrices). This separation of a third-order tensor into its matrix components will be relevant in the semantic interpretation of operations represented by a (trained) tensor network layer.

Tensor Product We now introduce the *tensor product* of vectors. In the context of our analysis, the tensor product is realized by the *Kronecker product*.[3] The Kronecker product relates to the tensor product as follows: With respect to a choice of basis, the Kronecker product is equivalent to the tensor product of vectors, represented in matrix form. Note that the two terms are frequently used interchangeably in the machine learning literature – likely due to the aforementioned practical equivalence in this setting.

For $m \times n$ matrix V and $p \times q$ matrix W, their *Kronecker product*, $V \otimes W$ is the $mp \times nq$ matrix given by:

$$V \otimes W = \begin{bmatrix} v_{1,1}W & \dots & v_{1,n}W \\ \vdots & \ddots & \vdots \\ v_{m,1}W & \dots & v_{m,n}W \end{bmatrix}$$

In the context of the discussed models, we will mostly encounter the Kronecker product of two vectors, represented in column form. In that case, the Kronecker product corresponds to the outer product, vw^T, for column vectors v, w.

Vectorization For reasons that will become apparent later, it will often be convenient to change the 'shape' of arbitrary matrices and represent them as vectors. We are able to do so by the operation of *vectorization*. For $m \times n$ matrix W, vectorization of vec(W) is the $mn \times 1$ vector given by the components:

$$\left(w_{1,1}, \dots w_{m,1}, \dots, w_{1,n}, \dots, w_{m,n}\right)$$

arranged as a single column, i.e. the columns of W are "stacked on top of each other".

Similarly, we can represent higher-order tensors as matrices, by *matricization* (or flattening), which can be seen as a generalization of the vectorization operation. We omit the (notationally cumbersome) general definition of mode-n matricization of n-th order tensor $\mathcal{V}$, denoted by $V_{(n)}$. Instead, we describe the special case of mode-3 matricization of a third-order tensor – the only case needed for our purposes.[4]

The mode-3 matricization of tensor $\mathcal{V} \in \mathbb{R}^{p \times q \times r}$ is the $r \times pq$ matrix:

$$\begin{bmatrix} v^{(1)}_{1,1} & \dots & v^{(1)}_{1,p} & \dots & v^{(1)}_{q,1} & \dots & v^{(1)}_{q,p} \\ \vdots & & & \ddots & & & \vdots \\ v^{(r)}_{1,1} & \dots & v^{(r)}_{1,p} & \dots & v^{(r)}_{q,1} & \dots & v^{(r)}_{q,p} \end{bmatrix}$$

[3] The tensor product over vectors can be defined in terms of the more abstract (algebraic) tensor product over vector spaces (Hackbusch, 2012)

[4] For the general definition, see (Kolda and Bader, 2009, Section 2.5)

We denote here by superscript indices $v^{(1)} \dots v^{(r)}$ the *slicing* index of the mode-3 matricization, running over r in $\mathcal{V} \in \mathbb{R}^{p \times q \times r}$. Note that the ordering of the columns of the matricized tensor is a matter of convention and only needs to be consistent within one system.

Multilinear Maps So far, we took two different perspectives on tensors: as multidimensional arrays, and as objects constructed by the tensor (or Kronecker) product. We will sketch here one additional perspective, that of tensors as multilinear map. Viewing tensors under this definition has the advantage that it will become apparent why higher-order tensors are an important representational component in compositional distributional models.

Intuitively, a multilinear map is a function combining elements of two or more vectors spaces, mapping them to an element of one vector space, and preserving linearity of the individual arguments. Given vector spaces $V_1, \dots, V_n, W$, we say that a function $f : V_1 \times \dots \times V_n \to W$ is a *multilinear map* if for each variable v_i, ranging over V_i in $V_1 \times \dots \times V_n$, when holding all variables other than the one indexed by i constant, $f(v_1, \dots, v_n)$ is a linear map. In other words, for each vector argument of f considered separately, f behaves like a linear map – we say that f is *linear in each of its arguments*.

By universality of the tensor product, multilinear maps and tensors are equivalent in the following sense: every multilinear map $f : V_1 \times \dots \times V_n \to W$ can be expressed by a unique linear map $g : V_1 \otimes \dots \otimes V_n \to W$. This transformation of a multilinear mapping of vectors to a linear mapping over tensor products of the vectors is sometimes referred to as the *linearization* of multilinear maps by tensors.[5] Summarizing, we note that the three perspectives, of tensors as multidimensional arrays, as multilinear maps, and as functions over spaces constructed by the tensor product, are equivalent ways to describe tensor $\mathcal{V}$:

$$\mathcal{V} \in V_1 \otimes \dots \otimes V_n \otimes W \tag{A.1}$$

$$\mathcal{V}_f : V_1 \times \dots \times V_n \to W \qquad \mathcal{V}_f \text{ is a multilinear map} \tag{A.2}$$

$$\mathcal{V}_g : V_1 \otimes \dots \otimes V_n \to W \qquad \mathcal{V}_g \text{ is a linear map} \tag{A.3}$$

Note also the relation between the order of a tensor and the number of arguments (variables) of the corresponding multilinear map, i.e. a tensor $\mathcal{V}$ of order $n + 1$ is equivalent to a multilinear map of n arguments.

Bilinear Maps and Interaction In the context of this thesis we will mostly encounter third-order (cubic) tensors, corresponding to multilinear functions of two arguments, called *bilinear* maps. Instantiating the general case of Eqs. (A.1) to (A.3) by a third-

[5] (Hackbusch, 2012, Proposition 3.22)

order tensor and field $\mathbb{R}$ yields:[6]

$$\mathcal{V} \in \mathbb{R}^{p\times q\times r} \tag{A.4}$$

$$\mathcal{V}_f : \mathbb{R}^p \times \mathbb{R}^q \to \mathbb{R}^r \qquad \mathcal{V}_f \text{ is a bilinear map} \tag{A.5}$$

$$\mathcal{V}_g : \mathbb{R}^p \otimes \mathbb{R}^q \to \mathbb{R}^r \qquad \mathcal{V}_g \text{ is a linear map} \tag{A.6}$$

Eq. (A.5), the view of a third-order tensor as a bilinear function is perhaps the most transparent way to see that $\mathcal{V}$ is a function of *two* vector space arguments (variables), in contrast to linear maps (matrices), which describe functions of a *single* vector space argument. Taking a compositional perspective on language (modeling), finding adequate representations of functions of more than one argument appears to be a necessity. Consequently, the bilinearity of tensor $\mathcal{V}$ constitutes the formal justification for the frequent appeal for tensors in the compositional literature for functional representations.

The representation of a tensor by Eq. (A.6) allows for a natural understanding of *multiplicative interaction* between composition arguments.[7] The domain of $\mathcal{V}_g$ is given by the tensor product of the input spaces. By definition of the tensor (or Kronecker) product, the input of function $\mathcal{V}_g$ consists therefore of products of all component combinations of some vector pair $(x \in \mathbb{R}^p, y \in \mathbb{R}^q)$. It is evident then that *multiplicative interaction* is representable by a bilinear tensor, but not a linear map $h\colon \mathbb{R}^m \to \mathbb{R}^n$, where the domain is a single space, and input is not given by products over several vector arguments.

You Call That a Tensor? Considering tensors as arbitrarily shaped multidimensional arrays, the meaning of 'tensor' followed in this introduction, is also the prevalent view in most of the machine learning literature.[8] The reader should be aware however that this meaning of the term is different from the way it is commonly understood in most of pure mathematics and physics. In mathematical treatments tensors are often defined by the tensor product of vector spaces, and the noun 'tensor' usually refers to elements of the tensor product of a space and its *dual space*, i.e. a type (p, q) tensor is an element of $(V \otimes \ldots \otimes V)_{|p|} \otimes (V^* \otimes \ldots \otimes V^*)_{|q|}$. In contrast, 'tensor' in machine learning commonly refers to elements of *any* tensor product of spaces, i.e. a tensor here is, for example, an element of $V \otimes W \otimes W$, where V, W are arbitrary vector spaces over the same field, not necessarily of the same dimension. The latter definition is equivalent to the perspective of tensors as 'multidimensional' arrays of arbitrary spatial dimensions at each index.

---

[6] Indexing order in expressions like Eq. (A.4) varies in the literature. The same tensor can, for example, also be written as $\mathcal{V} \in \mathbb{R}^{r\times(p\times q)}$.

[7] The linguistic relevance of multiplicative interaction is a recurring theme of this thesis, and the concept is introduced in Chapter 3. In Chapter 4 we discuss and visualize the differences between additive and multiplicative interaction, on the basis of composition in recursive networks.

[8] (Kolda and Bader, 2009, Introduction).

Problem of Dimensionality While relevant, a technically detailed introduction to tensor approximation methods is beyond the scope of this introduction. We will however briefly mention some related terminology, and offer the basic intuitions behind some concepts.

The tensors considered so far are referred to as *full rank* tensors in the context of machine learning models, meaning that all components of a tensor are fully specified numerically, i.e. stored, applied and updated when running the model employing the tensor. One major problem in using higher-order tensors in machine learning is the large number of parameters required to specify a tensor in this way. It is easy to see the problem: the number of components (parameters) of a matrix, a second-order tensor, is approximately quadratic in its dimensions, the number of parameters of a third-order tensor is roughly cubic in its dimensions, and so on. In other words, the size of tensors grows exponentially with tensor order. Even for moderately large dimensions at each index of a tensor, this entails that using full-rank tensors quickly becomes prohibitively expensive in terms of required storage, as well as making them computationally intractable at runtime.

Dimensionality Reduction Due to the reason outlined above, there is substantial interest in finding ways to make tensors more tractable, generally in the sense of applying *dimensionality reduction* techniques. Many of these approaches take the form of *low-rank approximations* of tensors. To provide the basic intuition behind such methods, consider the following example: Say we could, more or less accurately, represent some third-order tensor $\mathcal{V}$ by the tensor product of three vectors u, v, w, i.e.$\mathcal{V} \simeq u \otimes v \otimes w$. It is easy to see that purely in terms of storage requirements, the number of components needed to store vectors u, v, w, $u + v + w$, favorably compares to the parameters of $\mathcal{V} \in \mathbb{R}^{u \times v \times w}$. This example is of course a major simplification of the actual problem, since the result of approximation depends on whether a matrix or tensor *can* in principle be approximated by a less complex combination of products than the combination expressed by the full tensor (or matrix) itself. The degree to which such an approximation is possible can be thought of as the intuition behind the *rank* of a matrix or tensor. In the example above, we would call the tensor, being fully representable by a single outer product of three vectors, a *rank one* tensor – the opposite of a full-rank tensor in terms of 'complexity' required to fully represent it.

Tucker Decomposition Concluding, we will briefly sketch an actual decomposition method used to reduce the complexity of higher-order tensors: the *Tucker decomposition*. A third-order tensor $\mathcal{V}$ is decomposed into a *core tensor* $\mathcal{W}$ and three (orthogonal) *factor matrices*, A, B, C, where the number of rows of the i-th factor matrix matches the dimension of the i-th index of $\mathcal{V}$. $\mathcal{V}$ is then approximated as a sum of products over these factors, weighted by the components of $\mathcal{W}$. If the dimensions of $\mathcal{W}$ are identical to those of $\mathcal{V}$, the approximation is exact. Depending on the rank of $\mathcal{V}$, a good approximation can be achieved however by a core tensor of lower dimensions, thus reducing storage and computation demands if the decomposition is used as a proxy for $\mathcal{V}$ in applications.

We will briefly come back to factorized tensor models in Chapter 3, contrasting them with the (full-rank) RNTN model which formed the basis of our own analysis. For a serious introduction to tensor decomposition, and an overview of relevant methods, see for example Kolda and Bader (2009).

Appendix B

Code Samples

Tensor Efficiency The following code shows the core functions for forward and backward passes of the tensor model. Optimization of these functions turned out to be crucial, given that the tensor term dwarfs all other components in complexity. Compared to our initial implementation of these routines, our final, heavily optimized version provided a major speedup, allowing us to train a larger number of models, and ultimately, enabling a deeper investigation of the tensor architecture.

```
self.dW_cpr += delta_cpr.reshape((delta_cpr.shape[0], 1))\
       @l_r_stack.reshape((l_r_stack.shape[0], 1)).T
self.dV_cpr += delta_cpr.reshape((delta_cpr.shape[0], 1))\
       @l_r_prod.reshape((l_r_prod.shape[0], 1)).T

self.dW_cpr += np.outer(delta_cpr, l_r_stack)

id_x_r = np.kron(id_word_dim, r_act)
l_x_id = np.kron(l_act, id_word_dim)

id_x_r = np.outer(id_word_dim, r_act)\
       .reshape((r_act.size, id_word_dim.size))
l_x_id = np.outer(l_act, id_word_dim)\
       .reshape((id_word_dim.size, l_act.size))
l_x_id = np.swapaxes(l_x_id, 0, 1)

id_r_l_stack = np.vstack([id_x_r, l_x_id])
Jac_V_cpr = theta.V_cpr@id_r_l_stack.T

Jac_V_cpr = np.hstack([theta.V_cpr@id_x_r.T,\
       theta.V_cpr@l_x_id.T])

self.dW_cps += np.outer(delta, l_r_stack)
self.dV_cps += np.outer(delta, l_r_prod)

self.dW_cps += delta.reshape((delta.shape[0], 1))\
```

```
        @l_r_stack.reshape((l_r_stack.shape[0], 1)).T

id_x_r = np.kron(id_word_dim, r_act)
l_x_id = np.kron(l_act, id_word_dim)

id_x_r = np.outer(id_word_dim, r_act)\
        .reshape((r_act.size, id_word_dim.size))
l_x_id = np.outer(l_act, id_word_dim)\
        .reshape((id_word_dim.size, l_act.size))
l_x_id = np.swapaxes(l_x_id, 0, 1)

id_r_l_stack = np.vstack([id_x_r, l_x_id])
Jac_V_cps = theta.V_cps@id_r_l_stack.T

Jac_V_cps = np.hstack([theta.V_cps@id_x_r.T, theta.V_cps@l_x_id.T])
```

Regularization Term Correctness The following code blocks shows two different ways of implementing the regularization term of the training routines. The first block is the way this terms is usually implemented, while the second block follows the formally *correct* way, suggested by Bengio (2012).

```
# Unscaled L2
self.dW_voc = self.dW_voc/mb_size + l2*theta.W_voc
self.db_cps = self.db_cps/mb_size
self.dW_cps = self.dW_cps/mb_size + l2*theta.W_cps
if tensor_settings in {1, 3}:
    self.dV_cps = self.dV_cps/mb_size + l2*theta.V_cps
self.db_cpr = self.db_cpr/mb_size
self.dW_cpr = self.dW_cpr/mb_size + l2*theta.W_cpr
if tensor_settings in {2, 3}:
    self.dV_cpr = self.dV_cpr/mb_size + l2*theta.V_cpr
self.dW_sm = self.dW_sm/mb_size + l2*theta.W_sm
self.db_sm = self.db_sm/mb_size

# L2 term scaled by data size
len_data = len(self.train_data.tree_data)
self.dW_voc = self.dW_voc/mb_size + (l2*theta.W_voc)/len_data
self.db_cps = self.db_cps/mb_size
self.dW_cps = self.dW_cps/mb_size + (l2*theta.W_cps)/len_data
if tensor_settings in {1, 3}:
    self.dV_cps = self.dV_cps/mb_size + (l2*theta.V_cps)/len_data
self.db_cpr = self.db_cpr/mb_size
self.dW_cpr = self.dW_cpr/mb_size + (l2*theta.W_cpr)/len_data
if tensor_settings in {2, 3}:
    self.dV_cpr = self.dV_cpr/mb_size + (l2*theta.V_cpr)/len_data
self.dW_sm = self.dW_sm/mb_size + (l2*theta.W_sm)/len_data
self.db_sm = self.db_sm/mb_size
```

Bibliography

Aerts, Diederik and Liane Gabora. A theory of concepts and their combinations I: The structure of the sets of contexts and properties. *Kybernetes*, 34(1/2):167–191. (2005a)

Aerts, Diederik and Liane Gabora. A theory of concepts and their combinations II: A Hilbert space representation. *Kybernetes*, 34(1/2):192–221. (2005b)

Arora, Sanjeev, Yuanzhi Li, Yingyu Liang, Tengyu Ma, and Andrej Risteski. Randwalk: A latent variable model approach to word embeddings. *arXiv preprint arXiv:1502.03520*. (2015)

Baroni, Marco and Roberto Zamparelli. Nouns are vectors, adjectives are matrices: Representing adjective-noun constructions in semantic space. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pp. 1183–1193. Association for Computational Linguistics. (2010)

Baroni, Marco, Raffaela Bernardi, and Roberto Zamparelli. Frege in space: A program of compositional distributional semantics. *LiLT (Linguistic Issues in Language Technology)*, 9. (2014a)

Baroni, Marco, Georgiana Dinu, and Germán Kruszewski. Don't count, predict! A systematic comparison of context-counting vs. context-predicting semantic vectors. In *ACL (1)*, pp. 238–247. (2014b)

Bengio, Yoshua. Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*, pp. 437–478. Springer. (2012)

Bengio, Yoshua, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of machine learning research*, 3(Feb):1137–1155. (2003)

Bojanowski, Piotr, Edouard Grave, Armand Joulin, and Tomas Mikolov. Enriching word vectors with subword information. *arXiv preprint arXiv:1607.04606*. (2016)

Bowman, Samuel R. Can recursive neural tensor networks learn logical reasoning? arXiv preprint arXiv:1312.6192. Unpublished manuscript. (2014)

Bowman, Samuel R. *Modeling natural language semantics in learned representations*. PhD thesis, Stanford University. (2016)

Bowman, Samuel R, Christopher Potts, and Christopher D Manning. Recursive neural networks can learn logical semantics. In *Proceedings of the 3rd Workshop on Continuous Vector Space Models and their Compositionality*. (2015)

Clark, Stephen, Bob Coecke, and Mehrnoosh Sadrzadeh. Towards a Compositional Distributional Model of Meaning. (2008)

Coecke, Bob, Mehrnoosh Sadrzadeh, and Stephen Clark. Mathematical foundations for a compositional distributional model of meaning. *arXiv preprint arXiv:1003.4394*. (2010)

Collobert, Ronan and Samy Bengio. Links between perceptrons, MLPs and SVMs. In *Proceedings of the twenty-first international conference on Machine learning*, p. 23. ACM. (2004)

Cybenko, George. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS)*, 2(4):303–314. (1989)

Dauphin, Yann N, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Advances in neural information processing systems*, pp. 2933–2941. (2014)

Duchi, John, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul): 2121–2159. (2011)

Elman, Jeffrey L. Finding structure in time. *Cognitive science*, 14(2):179–211. (1990)

Firth, John R. A synopsis of linguistic theory, 1930-1955. (1957)

Frege, Gottlob. Über Sinn und Bedeutung. *Zeitschrift fur Philosophie und philosophische Kritik*. (1892)

Gamut, L T F. *Logic, language, and meaning. Volume II. Intensional logic and logical grammar*. University of Chicago Press, Chicago. (1991)

Gers, Felix A and Jürgen Schmidhuber. LSTM recurrent networks learn simple context-free and context-sensitive languages. *IEEE Transactions on Neural Networks*, 12(6): 1333–1340. (2001)

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

Goller, Christoph and Andreas Küchler. Learning task-dependent distributed representations by backpropagation through structure. In *Neural Networks, 1996., IEEE International Conference on*, Vol. 1, pp. 347–352. IEEE. (1996)

Graves, Alex. *Supervised sequence labelling with recurrent neural networks*. PhD thesis, Technical University Munich. (2008)

Graves, Alex. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*. (2013)

Grefenstette, Edward. Towards a formal distributional semantics: Simulating logical calculi with tensors. *arXiv preprint arXiv:1304.5823*. (2013)

Grefenstette, Edward and Mehrnoosh Sadrzadeh. Concrete models and empirical evaluations for the categorical compositional distributional model of meaning. *Computational Linguistics*. (2015)

Grefenstette, Edward, Mehrnoosh Sadrzadeh, Stephen Clark, Bob Coecke, and Stephen Pulman. Concrete sentence spaces for compositional distributional models of meaning. In *Proceedings of the 9th International Conference on Computational Semantics (IWCS11)*, pp. 125–134. (2011)

Grefenstette, Edward, Georgiana Dinu, Yao-Zhong Zhang, Mehrnoosh Sadrzadeh, and Marco Baroni. Multi-step regression learning for compositional distributional semantics. *arXiv preprint arXiv:1301.6939*. (2013)

Hackbusch, Wolfgang. *Tensor spaces and numerical tensor calculus*, Vol. 42. Springer Science & Business Media. (2012)

Hackbusch, Wolfgang and Stefan Kühn. A new scheme for the tensor representation. *Journal of Fourier analysis and applications*, 15(5):706–722. (2009)

Heim, Irene and Angelika Kratzer. *Semantics in generative grammar*, Vol. 13. Blackwell Oxford. (1998)

Hinton, Geoffrey E. Learning distributed representations of concepts. In *Proceedings of the eighth annual conference of the cognitive science society*, Vol. 1, p. 12. Amherst, MA. (1986)

Hinton, Geoffrey E, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*. (2012)

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

Hornik, Kurt, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366. (1989)

Ioffe, Sergey and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pp. 448–456. (2015)

Irsoy, Ozan and Claire Cardie. Modeling compositionality with multiplicative recurrent neural networks. *arXiv preprint arXiv:1412.6577*. (2014)

Kingma, Diederik and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*. (2014)

Kokkinos, Filippos and Alexandros Potamianos. Structural Attention Neural Networks for improved sentiment analysis. *arXiv preprint arXiv:1701.01811*. (2017)

Kolda, Tamara G and Brett W Bader. Tensor decompositions and applications. *SIAM review*, 51(3):455–500. (2009)

Krause, Ben, Liang Lu, Iain Murray, and Steve Renals. Multiplicative LSTM for sequence modelling. *arXiv preprint arXiv:1609.07959*. (2016)

Kruszewski, Germán, Denis Paperno, Raffaella Bernardi, and Marco Baroni. There is no logical negation here, but there are alternatives: Modeling conversational negation with distributional semantics. *Computational Linguistics*. (2017)

Lakoff, George. Linguistics and natural logic. *Synthese*, 22(1):151–271. (1970)

Le, Phong and Willem Zuidema. Compositional distributional semantics with long short term memory. *arXiv preprint arXiv:1503.02510*. (2015)

Levy, Omer and Yoav Goldberg. Neural word embedding as implicit matrix factorization. In *Advances in neural information processing systems*, pp. 2177–2185. (2014)

Levy, Omer, Yoav Goldberg, and Ido Dagan. Improving distributional similarity with lessons learned from word embeddings. *Transactions of the Association for Computational Linguistics*, 3:211–225. (2015)

Lund, Kevin and Curt Burgess. Producing high-dimensional semantic spaces from lexical co-occurrence. *Behavior Research Methods, Instruments, & Computers*, 28(2): 203–208. (1996)

MacCartney, Bill. *Natural language inference*. PhD thesis, Stanford University. (2009)

MacCartney, Bill and Christopher D Manning. An extended model of natural logic. In *Proceedings of the eighth international conference on computational semantics*, pp. 140–156. Association for Computational Linguistics. (2009)

Maillard, Jean, Stephen Clark, and Edward Grefenstette. A type-driven tensor-based semantics for CCG. In *Proceedings of the EACL 2014 Type Theory and Natural Language Semantics Workshop*, pp. 46–54. (2014)

Marelli, Marco, Stefano Menini, Marco Baroni, Luisa Bentivogli, Raffaella Bernardi, and Roberto Zamparelli. A SICK cure for the evaluation of compositional distributional semantic models. In *LREC*, pp. 216–223. (2014)

McCulloch, Warren S and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133. (1943)

Mhaskar, Hrushikesh, Qianli Liao, and Tomaso Poggio. Learning functions: When is deep better than shallow. *arXiv preprint arXiv:1603.00988*. (2016)

Mikolov, Tomas. *Statistical Language Models Based on Neural Networks*. PhD thesis, Brno University of Technology. (2012)

Mikolov, Tomas, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*. (2013a)

Mikolov, Tomas, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pp. 3111–3119. (2013b)

Mikolov, Tomas, Wen-tau Yih, and Geoffrey Zweig. Linguistic Regularities in Continuous Space Word Representations. In *Hlt-naacl*, Vol. 13, pp. 746–751. (2013c)

Milajevs, Dmitrijs, Dimitri Kartsaklis, Mehrnoosh Sadrzadeh, and Matthew Purver. Evaluating neural word representations in tensor-based compositional settings. *arXiv preprint arXiv:1408.6179*. (2014)

Minsky, Marvin and Seymour Papert. Perceptrons. (1969)

Mitchell, Jeff and Mirella Lapata. Vector-based Models of Semantic Composition. In *ACL*, pp. 236–244. (2008)

Mitchell, Jeff and Mirella Lapata. Composition in distributional models of semantics. *Cognitive science*, 34(8):1388–1429. (2010)

Mohammad, Saif M, Bonnie J Dorr, Graeme Hirst, and Peter D Turney. Computing lexical contrast. *Computational Linguistics*, 39(3):555–590. (2013)

Montague, Richard. English as a formal language. (1970a)

Montague, Richard. Universal grammar. *Theoria*, 36(3):373–398. (1970b)

Montague, Richard. The proper treatment of quantification in ordinary English. In *Approaches to natural language*, pp. 221–242. Springer. (1973)

Montufar, Guido F, Razvan Pascanu, Kyunghyun Cho, and Yoshua Bengio. On the number of linear regions of deep neural networks. In *Advances in neural information processing systems*, pp. 2924–2932. (2014)

Partee, Barbara H. Montague Grammar. In Smelser, Neil J and Paul B Baltes, editors, *International encyclopedia of the social & behavioral sciences*, Vol. 11. Elsevier Amsterdam. (2001)

Partee, Barbara H. *Montague grammar*. Elsevier. (2014)

Pascanu, Razvan, Guido Montufar, and Yoshua Bengio. On the number of response regions of deep feed forward networks with piece-wise linear activations. In *International Conference on Learning Representations 2014 (ICLR 2014), Banff, Alberta, Canada*. (2013)

Polajnar, Tamara and Stephen Clark. Reducing dimensions of tensors in type-driven distributional semantics. (2014)

Pollack, Jordan B. Recursive distributed representations. *Artificial Intelligence*, 46(1): 77–105. (1990)

Pollack, Jordan B. The induction of dynamical recognizers. *Machine learning*, 7(2): 227–252. (1991)

Rosenblatt, Frank. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386. (1958)

Smilkov, Daniel, Nikhil Thorat, Charles Nicholson, Emily Reif, Fernanda B Viégas, and Martin Wattenberg. Embedding Projector: Interactive Visualization and Interpretation of Embeddings. *arXiv preprint arXiv:1611.05469*. (2016)

Smolensky, Paul. Tensor product variable binding and the representation of symbolic structures in connectionist systems. *Artificial intelligence*, 46(1-2):159–216. (1990)

Socher, Richard. *Recursive deep learning for natural language processing and computer vision*. PhD thesis, Stanford University. (2014)

Socher, Richard, Christopher D Manning, and Andrew Y Ng. Learning continuous phrase representations and syntactic parsing with recursive neural networks. In *Proceedings of the NIPS-2010 Deep Learning and Unsupervised Feature Learning Workshop*, pp. 1–9. (2010)

Socher, Richard, Brody Huval, Christopher D Manning, and Andrew Y Ng. Semantic compositionality through recursive matrix-vector spaces. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, pp. 1201–1211. Association for Computational Linguistics. (2012)

Socher, Richard, Danqi Chen, Christopher D Manning, and Andrew Ng. Reasoning with neural tensor networks for knowledge base completion. In *Advances in neural information processing systems*, pp. 926–934. (2013a)

Socher, Richard, Alex Perelygin, Jean Y Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the conference on empirical methods in natural language processing (EMNLP)*, Vol. 1631, p. 1642. (2013b)

Strang, Gilbert. *Introduction to Linear Algebra*. Wellesley-Cambridge Press, Fourth edition. (2009)

Sutskever, Ilya, James Martens, and Geoffrey E Hinton. Generating text with recurrent neural networks. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pp. 1017–1024. (2011)

Tai, Kai Sheng, Richard Socher, and Christopher D Manning. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*. (2015)

Taylor, Graham W and Geoffrey E Hinton. Factored conditional restricted Boltzmann machines for modeling motion style. In *Proceedings of the 26th annual international conference on machine learning*, pp. 1025–1032. ACM. (2009)

Turney, Peter D and Patrick Pantel. From frequency to meaning: Vector space models of semantics. *Journal of artificial intelligence research*, 37:141–188. (2010)

Valencia, Víctor Manuel Sánchez. *Studies on natural logic and categorial grammar*. PhD thesis, University of Amsterdam. (1991)

van Benthem, Johan. Language in action. *Journal of Philosophical Logic*, 20(3): 225–263. (1991)

Van de Cruys, Tim, Thierry Poibeau, and Anna Korhonen. A tensor-based factorization model of semantic compositionality. In *Conference of the North American Chapter of the Association of Computational Linguistics (HTL-NAACL)*, pp. 1142–1151. (2013)

Veldhoen, Sara and Willem Zuidema. Can neural networks learn logical reasoning? In *Proceedings of the Conference on Logic and Machine Learning in Natural Language Processing (LAML) (forthcoming)*, Gothenburg, Sweden. (2017)

Wager, Stefan, Sida Wang, and Percy S Liang. Dropout training as adaptive regularization. In *Advances in neural information processing systems*, pp. 351–359. (2013)

Wittgenstein, Ludwig. *Philosophical Investigations*. Wiley-Blackwell. (1953)

Wu, Yuhuai, Saizheng Zhang, Ying Zhang, Yoshua Bengio, and Ruslan R Salakhutdinov. On multiplicative integration with recurrent neural networks. In *Advances In Neural Information Processing Systems*, pp. 2856–2864. (2016)

Zeiler, Matthew D. ADADELTA: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*. (2012)