ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Broad Discourse Context for Language Modeling

Master Thesis

Moisés Torres

November 1, 2017

Supervisor:
Prof. Dr. Thomas Hofmann

Co-supervisors:
Florian Schmidt
Paulina Grnarova

Department of Computer Science, ETH Zürich

Abstract

Should be aimed for a person already familiar with the topic!

Discourse understanding summarizes a speaker's ability to perform multi-sentence reasoning. This in turn requires temporal reasoning, resolving co-reference chains, identifying named entities and keeping track of the state of a conversation. In language modeling, the de-facto standard task for text generation systems, deep neural network have become the state of the art. Attempts to transfer similar architectures to a dialogue task directly, have revealed the lack of a rigorous evaluation metric

As a middle ground, newly released datasets try to cast existing NLP problems in a discourse context. For example, collects hard word-prediction tasks to put to the test genuine understanding of language models. To succeed on this task, a language model cannot simply rely on local context, but must be able to keep track of information in the broader discourse. However, the established evaluation methodologies from language modeling still apply.

Previous work like have shown that plain RNNs' memory representation may not be enough to effectively capture long term dependencies. Thus, several approaches have been proposed to tackle this problem: attention , memory networks or latent variable RNNs, among others. In this thesis we implement several baselines of enhanced language models using some of the aforementioned techniques and evaluate them on hard-word prediction tasks like. Based on our empirical findings, we propose and compare extensions to tackle the identified shortcomings.

First paragraph, high level description of the problem. Check project proposal!
- Describe thesis approach, mention lambada maybe
- What focus for the abstact? Still focus on discourse?

Acknowledgments

I would like to thank... Finish acknowledgements

Contents

Chapter 1

Introduction

Here comes the intro...

Finish intro (3 pages)

1.1 Problem Statement and Motivation

motivation...

Finish motivation (1 page)

1.2 Thesis Contributions

contributions...

Finish contributions (1 page)

1.3 Thesis Outline

outline...

Finish outline (1 page)

Chapter 2

Related Work

This chapter will describe the state-of-the-art of...

Finish related work (3 pages)

	Local	**Global**
Domain	Trivial to do	us, Dger
Vocabulary	Socher	hardest

Table 2.1: Model's classification

Chapter 3

Neural Language Models

In this chapter, we introduce the notation used throughout the thesis and give a brief overview of the development of neural language models (NLM). We also review some of the main weaknesses shown by this family of models and how they have been addressed in the literature.

3.1 Notation

Before continuing, we will define the notation used in the thesis:

- Scalars are denoted with lowercase letters, such as x.
- Vectors are denoted with bold lowercase letters, such as $\mathbf{x}$ with $\mathbf{x}_i$ its i-th element, and are always assumed to be column vectors.
- Matrices are denoted with uppercase letters, such as X with X_{ij} its (i, j)-th element, $X_{i,:}$ its i-th row and $X_{:,j}$ its j-th column.

3.2 Background

Prior to introducing the specifics of NLMs, we will formalize the task at hand and introduce some of its core concepts.

3.2.1 Language Modeling

First, we define a **word-based language model** as a model able to compute the probability of a sentence or sequence of words $p(w_1, \ldots, w_n)$. Such models are of great use in tasks where we have to recognize words in noisy or ambiguous input such as speech recognition or machine translation, among others.

If we now decompose the joint probability of a sequence using the chain rule of probability as shown in Equation 3.1, we observe that the function

that needs to be estimated boils down to the conditional probability of a word given the history of previous words. However, taking into account the whole context poses a problem as language is creative and any particular sequence might have occurred few (or no) times before. Many of the models that we will introduce approximate the true conditional distribution by making a Markov assumption as shown in Equation 3.2. This means that the probability of an upcoming word is fully characterized by the previous $n-1$ words. Despite seeming an incorrect premise for a complex source of information such as language, it has been proven to work really well in practice.

$$\begin{aligned} p(w_1, \ldots, w_n) &= p(w_1)p(w_2|w_1)p(w_3|w_1^2)\ldots p(w_n|w_1^{n-1}) \\ &= \prod_{k=1}^{n} p(w_k|w_1^{k-1}) \end{aligned} \tag{3.1}$$

$$p(w_k|w_1^{k-1}) \approx p(w_k|w_{k-1}^{k-n}) \tag{3.2}$$

3.2.2 Evaluation

Following a common practice in machine learning, we use a test set in order to evaluate our models. In the case of language modeling we have a word sequence $W_1^n = \{w_1, \ldots, w_n\}$ and we expect the model to assign it a high probability. Rather than working directly with raw probabilities we define a metric called **perplexity**, which is the geometric average of the inverse of the probability over the test set, as shown in Equation 3.3. Therefore, lower perplexity is better.

$$\begin{aligned} \text{Perplexity}(W_1^n) = p(W_1^n)^{-\frac{1}{n}} &= \sqrt[n]{\frac{1}{p(W_1^n)}} \\ &= \sqrt[n]{\frac{1}{\prod_{k=1}^{n} p(w_k|W_1^{k-1})}} \end{aligned} \tag{3.3}$$

Moreover, we can regard language as an information source and therefore use Information Theory to find a different (and equivalent) interpretation of perplexity. For that we need to introduce the basic concept of **entropy** (Equation 3.4 shows its formulation for discrete variables), which measures the expected uncertainty or "surprise" S of the value of a random variable X. Without going into details, it is easy to see that defining uncertainty as the negative logarithm (the specific base doesn't matter, but traditionally it is assumed to be 2) of the probability of each event matches our intuition (like $S(p) > S(q)$ then $p < q$).

$$H(X) = \mathbb{E}[S(X)] = -\sum_{x \in \mathcal{X}} p(x) \log_2(p(x)) \quad \text{with} \quad S(\cdot) = -\log_2(\cdot) \tag{3.4}$$

A difference when it comes to language is that it involves dealing with sequences W_1^n of discrete random variables. For a given language L we can define the entropy of a variable ranging over all possible sequences of length n. To obtain the entropy-per-word we only need to normalize by n (Equation 3.5).

$$\frac{1}{n} H(W_1^n) = -\frac{1}{n} \sum_{W_1^n \in L} p(W_1^n) \log_2(p(W_1^n)) \tag{3.5}$$

Additionally, in order to calculate the true entropy of a language we would need to consider sequences of infinite length (Equation 3.6). Fortunately, the Shannon-McMillan-Breiman theorem states that if a stochastic source (such as language) is regular in certain ways (stationary and ergodic) we can take a single long enough sequence instead of summing over all possible sequences (* in Equation 3.6).

$$\begin{aligned} H(L) &= -\lim_{n \to \infty} \frac{1}{n} \sum_{W_1^n \in L} p(W_1^n) \log_2(p(W_1^n)) \\ &\overset{*}{=} -\lim_{n \to \infty} \frac{1}{n} \log_2(p(W_1^n)) \end{aligned} \tag{3.6}$$

Related to the concept of entropy we have **cross-entropy**, which measures the relative entropy of p with respect to m, p being the true probability distribution and m a model (e.g. an approximation) of p over the same underlying set of events. After applying the Shannon-McMillan-Breiman theorem and assuming that n is large enough, we can see in Equation 3.7 the final formulation of the cross-entropy, which has become the standard loss function when optimizing neural language models.

$$\begin{aligned} H(p, m) &= -\lim_{n \to \infty} \frac{1}{n} \sum_{W_1^n \in L} p(W_1^n) \log_2(m(W_1^n)) \\ &\overset{*}{=} -\lim_{n \to \infty} \frac{1}{n} \log_2(m(W_1^n)) \approx -\frac{1}{n} \log_2(m(W_1^n)) \\ &= -\frac{1}{n} \sum_{k=1}^{n} \log_2(m(w_k | W_1^{k-1})) \end{aligned} \tag{3.7}$$

Finally, we can see in Equation 3.8 how cross-entropy and perplexity are connected. This relation gives raise to a nice interpretation of perplexity as branching factor: entropy measures uncertainty (in bits, if we use $\log_2$) but in

exponentiated form it is measured as the cardinality of a uniform distribution with equivalent uncertainty.

$$\text{Perplexity}(W_1^n) = 2^{H(p,m)} = m(W_1^n)^{-\frac{1}{n}} \tag{3.8}$$

3.3 Feed-Forward Neural Language Models (FFNLM)

Until the appearance of NLMs, the most successful approaches were based on n-grams, which are models that estimate words from a fixed window of previous words and estimate probabilities by counting in a corpus and normalizing. Due to their nature, n-gram estimates intrinsically suffer from sparsity and several methods like smoothing, backoff and interpolation have been proposed to deal with this problem [1].

Along those lines, the first successful attempt of applying neural networks to language modeling [2] remarked the effect of the *curse of dimensionality* when it comes to estimating the joint distribution of many discrete random variables (such as words in a sentence). On the contrary, by using continuous variables we obtain better generalization because the function to be learned can be expected to have some local smoothness properties ("similar" words should get similar probabilities of being predicted). While requiring to be trained (n-grams don't), this approach is able to achieve significantly better results (reductions between 10% and 20% in perplexity with respect to a smoothed trigram model) by jointly learning word representations and a statistical language model.

Similar to n-grams, the model introduced in [2] conditions the probability of a word on the previous $n-1$ words. The main difference lies in the concept of **distributed feature vectors**; words are embedded into a vector-space by assigning them a real vector representation of size m via a look-up operation over the embedding matrix C as shown in Equation 3.9.

$$\begin{aligned} \mathbf{x} &= [C_{w_{t-1},:}; C_{w_{t-2},:}; \cdots; C_{w_{t-n+1},:})] \\ \mathbf{y} &= W\mathbf{x} + U\tanh(H\mathbf{x} + \mathbf{d}) + \mathbf{b} \\ \hat{\mathbf{p}}_i &= \hat{p}(w_t = i | w_{t-1}, \ldots, w_{t-n+1}) = \frac{e^{\mathbf{y}_i}}{\sum_n e^{\mathbf{y}_n}} \end{aligned} \tag{3.9}$$

where [ ] is the concatenation operator, $C \in \mathbb{R}^{|V|\times m}$, $\mathbf{x} \in \mathbb{R}^{1\times(n-1)m}$, $W \in \mathbb{R}^{|V|\times(n-1)m}$, $U \in \mathbb{R}^{|V|\times h}$, $H \in \mathbb{R}^{h\times(n-1)m}$, $\mathbf{d} \in \mathbb{R}^h$ and $\mathbf{y}, \mathbf{b} \in \mathbb{R}^{|V|}$.

The concatenated word representations $\mathbf{x}$ are fed through one (or more) nonlinear hidden layer (weights H and bias $\mathbf{d}$), resulting in a hidden representation of size h. We then apply an affine transformation (weights U and bias $\mathbf{b}$) to this hidden representation to obtain $\mathbf{y}$, an unnormalized probability

distribution over the vocabulary V. Optionally, we can include direct connections from the word features to the output (W). Finally, a softmax operation produces a valid probability distribution over the full vocabulary.

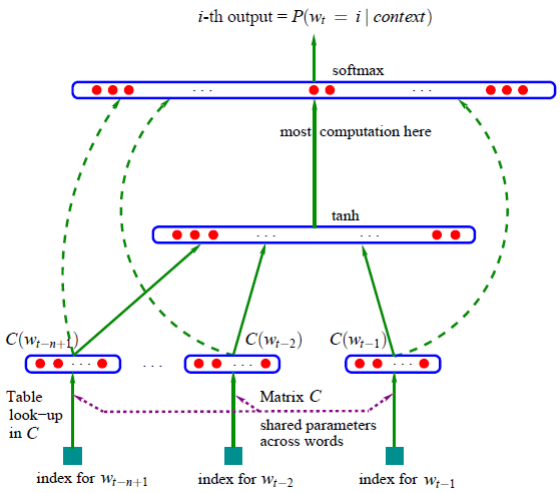

Figure 3.1: Feed-Forward NLM architecture [2]

Having a look at the full model (shown in Figure 3.2) we can observe that most expensive operation is the computation of the unnormalized probability distribution $\mathbf{y}$, which requires a number of dot products that scales linearly with $|V|$ (it is not unusual to work with vocabulary sizes in the order of the tens of thousands).

Several practical solutions have been proposed to avoid it, with **hierarchical softmax** [3] being one of the most popular ones. It uses a hierarchical binary tree representation of the vocabulary with words as its leaves and, for each node, explicitly represents the relative probabilities of its child nodes. These define a random walk that assigns probabilities to words, allowing to reduce the complexity of obtaining the output probability of a single word to around $\log_2(|V|)$.

3.4 Word Vectors

As we stated in the previous section, distributed continuous vectors allow for "clever" smoothing by taking into account syntactic and semantic features that are automatically learnt. [4] picked up on this concept trying to find ways of training these vector representations more efficiently. The paper introduces a family of models known as **word2vec**, whose architecture matches the one from a FFNLM where the nonlinear hidden layer has been removed (ending

up with a simple log bilinear model). The difference between them lies on the objective they optimize for. It is important to note that these objectives are designed as a mean to learn meaningful word embeddings, which are the main focus of the following architectures:

- Continuous Bag-of-Words model (CBOW): given a symmetric window of size k around a specific position i $\{w_{i-k}, \ldots, w_{i-1}, w_{i+1}, \ldots w_{i+k}\}$, we want to predict the word w_i. The term "bag-of-words" comes from the fact that the embeddings of the whole window are summed (instead of concatenated) and thus, order is not preserved.
- Continuous Skip-Gram model: given a specific position i we randomly sample words inside its surrounding window and try to predict them. Therefore, each training example is a 2-tuple consisting of w_i as input and a sampled word from the window as output.

In addition to a simplified architecture and a modified objective, further optimizations for the Skip-Gram model were introduced in a follow-up paper [5]. As we already explained at the end of the previous section, calculating a probability distribution over the full vocabulary is computation-intensive. In order to avoid doing this, the original task is casted into a binary classification problem by making use of a new objective named **negative sampling**. Inspired by noise contrastive estimation (NCE), the objective is to distinguish the target word w_O from draws coming from a noise distribution $p_n(w)$ (e.g. unigram distribution) using logistic regression, where there are k noise samples for each data point (Equation 3.10).

$$\begin{gathered}\mathcal{L}(\theta) = \log(\sigma({\mathbf{v}_{\mathbf{w_O}}}^\top \mathbf{v}_{\mathbf{w_I}})) + \sum_{i=1}^{k} \log(-\sigma({\mathbf{v}_{\mathbf{w_i}}}^\top \mathbf{v}_{\mathbf{w_I}})) \\ \text{with} \quad w_i \sim p_n(w)\end{gathered} \tag{3.10}$$

Another famous family of word vectors is **GloVe** [6], where the objective is a weighted (weighting function $f(\cdot)$) least squares fit of the log-counts (Equation 3.11). Rather than capturing co-occurences one word at a time (like word2vec), GloVe does dimensionality reduction on the whole co-occurrence counts matrix N.

$$\mathcal{L}(\theta, N) = \sum_{i,j:N_{ij}>0} f(N_{ij})(\log(N_{ij}) - ({\mathbf{v}_{\mathbf{w_O}}}^\top \mathbf{v}_{\mathbf{w_I}} + b_O + b_I))^2 \tag{3.11}$$

In general, word vectors have proven to excel at capturing semantic features. For example, [4] analyzes how Skip-Gram vectors achieve very good results on the semantic word relationship task. Furthermore, the authors

qualitatively explore how the resulting vector representations encode semantic information in the affine embedding structure. Hence, a vector operation like *Spain* − *Madrid* + *Switzerland* would lead to the desired answer *Bern*.

In summary, word vectors have become a standard in NLP and are used as input in all sorts of downstream applications such as sentiment analysis.

3.5 Recurrent Neural Language Models (RNLM)

So far all the models that we have seen (n-grams and FFNLM) explicitly use a fixed length context. Recurrent neural networks remove this limitation by introducing recurrent connections that allow information to cycle for an arbitrarily long time (although this is not true in practice). They learn to compress the whole history in low dimensional space (hidden state $\mathbf{h}(t)$) that is sequentially updated by being blended with the current input $\mathbf{x}(t)$ (Equation 3.12 introduces the formulation for Elman networks).

$$\mathbf{h}(t) = \sigma_h(W_h\mathbf{x}(t) + U_h\mathbf{h}(t-1) + \mathbf{b_h}) \tag{3.12}$$

where $W_h \in \mathbb{R}^{h\times m}$, $\mathbf{x}(t) \in \mathbb{R}^m$, $U_h \in \mathbb{R}^{h\times h}, \mathbf{h}(t), \mathbf{h}(t-1), \mathbf{b_h} \in \mathbb{R}^h$ and σ_h is a nonlinear function (e.g. sigmoid).

It is important to note that the network parameters are shared over time. [7] was one of the first to successfully apply RNNs to language modeling. The hidden states $\mathbf{h}(t)$ are fed through a fully connected layer to produce a probability distribution over the vocabulary in a similar way to FFNLMs.

$$\begin{aligned} \mathbf{y}(t) &= W_y\mathbf{h}(t) + \mathbf{b_y} \\ \hat{\mathbf{p}}(t)_i &= \hat{p}(w_t = i|\mathbf{h}(t)) = \frac{e^{\mathbf{y}(t)_i}}{\sum_n e^{\mathbf{y}(t)_n}} \end{aligned} \tag{3.13}$$

where $W_y \in \mathbb{R}^{|V|\times h}$ and $\mathbf{y}(t)$, $\mathbf{b_y}, \hat{\mathbf{p}}(t) \in \mathbb{R}^{|V|}$.

3.5.1 Backpropagation Through Time (BPTT)

Backpropagation is the standard gradient computation technique used for neural networks. However when applied to the recurrent connections of an RNN, the gradients will depend on the previous timesteps (up to $t = 0$). Thus, we call Backpropagation Through Time (BPTT) [8] to the application of backpropagation on an unrolled RNN, which accounts for this dependencies by summing up the gradients for each time step. In Equation 3.14 we see an example of this when calculating the gradient for U_h.

$$\frac{\partial \mathcal{L}(t)}{\partial U_h} = \frac{\partial \mathcal{L}(t)}{\partial \hat{\mathbf{p}}(t)} \frac{\partial \hat{\mathbf{p}}(t)}{\partial \mathbf{h}(t)} \frac{\partial \mathbf{h}(t)}{\partial U_h} = \sum_{k=0}^{t} \frac{\partial \mathcal{L}(t)}{\partial \hat{\mathbf{p}}(t)} \frac{\partial \hat{\mathbf{p}}(t)}{\partial \mathbf{h}(t)} \frac{\partial \mathbf{h}(t)}{\partial \mathbf{h}(k)} \frac{\partial \mathbf{h}(k)}{\partial U_h} \tag{3.14}$$

One of the main problems of BPTT is the high cost of a single parameter update, which makes it impossible to use for large numbers of iterations. **Truncated Backpropagation Through Time** (TBPTT), which is a modified version of BPTT, was introduced in [9] to work around this limitation. When using TBPTT, the sequence is processed one timestep at a time, and every k_1 timesteps, it runs BPTT for k_2 timesteps, so a parameter update can be cheap if k_2 is small. Most implementations assume $k_1 = k_2$.

3.5.2 Exploding and Vanishing Gradient

The number of derivatives required for a single weight update is directly proportional to the number of steps of our input sequences. We can see this in Equation 3.15 by observing that the term $\frac{\partial \mathbf{h}(t)}{\partial \mathbf{h}(k)}$ is a chain-rule itself. Citing [10], *"a product of $t-k$ real numbers can shrink to zero or explode to infinity, so does this product of matrices"* and therefore, this can cause gradients to vanish or explode.

$$\frac{\partial \mathbf{h}(t)}{\partial \mathbf{h}(k)} = \prod_{t \geq i > k} \frac{\partial \mathbf{h}(i)}{\partial \mathbf{h}(i-1)} \tag{3.15}$$

For exploding gradients, clipping the norm of the gradients to a predefined threshold has proven to be an effective remedy for this problem. Moreover, vanishing gradients translate into gradient contributions from "far away" steps becoming zero, and thus hindering the learning of long-range dependencies. The most popular solution for this problem has been the introduction of new cell architectures explicitly designed to deal with vanishing gradients such as Long Short-Term Memory (LSTM) [11] and Gated Recurrent Units (GRU) [12]. We will focus on the LSTM as the GRU cell is just a simplified version of the former. The main differences to a vanilla RNN are the introduction of a cell state $\mathbf{c}(t)$ (that acts as internal memory) and the gates $\mathbf{f}(t)$, $\mathbf{i}(t)$ and $\mathbf{o}(t)$:

$$\begin{aligned}
\mathbf{f}(t) &= \sigma_g(W_f \mathbf{h}(t-1) + U_f \mathbf{x}(t) + \mathbf{b}_f) \\
\mathbf{i}(t) &= \sigma_g(W_i \mathbf{h}(t-1) + U_i \mathbf{x}(t) + \mathbf{b}_i) \\
\mathbf{o}(t) &= \sigma_g(W_o \mathbf{h}(t-1) + U_o \mathbf{x}(t) + \mathbf{b}_o) \\
\tilde{\mathbf{c}}(t) &= \sigma_c(W_c \mathbf{h}(t-1) + U_c \mathbf{x}(t) + \mathbf{b}_c) \\
\mathbf{c}(t) &= \mathbf{f}(t) \odot \mathbf{c}(t-1) + \mathbf{i}(t) \odot \tilde{\mathbf{c}}(t) \\
\mathbf{h}(t) &= \mathbf{o}(t) \odot \sigma_h(\mathbf{c}(t))
\end{aligned} \tag{3.16}$$

where $W_f, W_i, W_o, W_f \in \mathbb{R}^{h \times h}$, $U_f, U_i, U_o, U_f \in \mathbb{R}^{h \times m}$, $\mathbf{f}(t), \mathbf{b}_f, \mathbf{i}(t), \mathbf{b}_i$, $\mathbf{o}(t), \mathbf{b}_o, \mathbf{c}(t), \mathbf{b}_c, \tilde{\mathbf{c}}(t) \in \mathbb{R}^h$ and $\sigma_g, \sigma_c, \sigma_h$ are non linear functions.

Gates are a way to optionally let information through and are learnt in such a way that the cell can remember long-range dependencies. Each gate is calculated as an affine transformation of the current input $\mathbf{x}(t)$ and the previous hidden state $\mathbf{h}(t-1)$ followed by a non linearity, as shown in Equation 3.16. These gates modify the cell state through pointwise multiplications; specifically, the new cell state is formed by a combination of the previous cell state weighted by the forget gate $\mathbf{f}(t)$ and the "newly proposed" state $\tilde{\mathbf{c}}(t)$ weighted by the input gate $\mathbf{i}(t)$. Therefore, gates provide an explicit way for the cell to decide what information should be forgotten and what is worth remembering for the next steps.

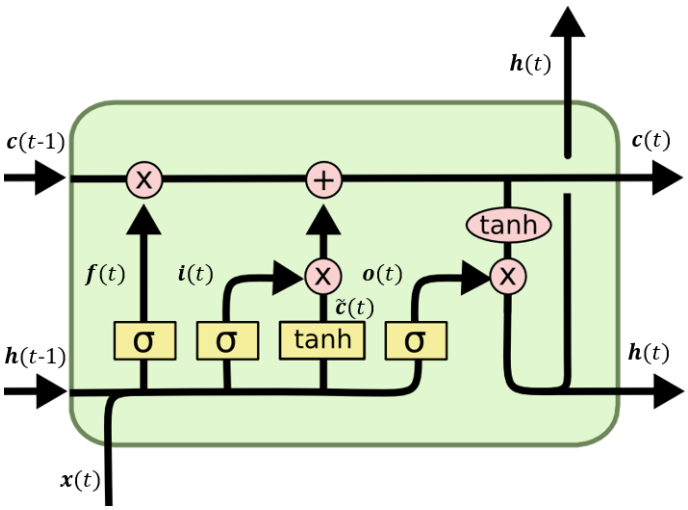

Figure 3.2: LSTM cell architecture [13]

To sum up, recurrent neural network architectures using LSTM cells have become ubiquitous in a wide range of NLP tasks and are a key element for state-of-the-art language models [14].

Chapter 4

Rare Word Prediction

This chapter introduces

Finish this

4.1 Problem Description

Talk about the well known problem of static language models and lack of adaptation

[15] describes the rare word problem with the vocab softmax and so on. Rare words won't generalize

4.2 The LAMBADA dataset

This dataset was first introduced in [16] as a challenging test set specifically designed to probe the genuine language understanding of state-of-the-art NLP models, since in words of the authors *"models' effectiveness at picking statistical generalizations from large corpora can lead to the illusion that they are reaching a deeper degree of understanding than they really are"*. And it does so by casting language understanding in the classic word prediction framework of language modeling. An example of the dataset can be seen in Figure 4.1.

Context: *"Why?" "I would have thought you'd find him rather dry," she said. "I don't know about that," said Gabriel. "He was a great craftsman," said Heather. "That he was," said Flannery.*

Target sentence: *"And Polish, to boot," said* _____ .

Target word: *Gabriel*

Figure 4.1: Example of a LAMBADA passage

4.2.1 Dataset Construction

LAMBADA was built using the Book Corpus dataset [17], which features 5325 unpublished novels and 465 million words. It was divided

4.2.2 Analysis

Chapter 5

Extended Language Models

This chapter describes several recently proposed NLM architectures designed to tackle the rare word prediction problem and highlights the main similarities and differences among them. We also introduce our proposed model called "Sentinel Mixture".

- Do you agree with the order of the sections (our model going last)?
- Any other insights regarding our model?

5.1 Attention Models

Although the concept of attention that we are going to introduce is not directly concerned with rare word prediction, it will help us lay the foundations for the subsequent models as they take inspiration from it.

5.1.1 Original Formulation

Attention models were firstly developed in the field of neural machine translation (NMT) in the seminal paper [18]. At that time, most of the proposed NMT models belonged to a family known as "encoder-decoders" [19]. These models consist of two parts: first an encoder (a deep bidirectional LSTM network) compresses the input sentence (specifically the embeddings for each word) $\{\mathbf{x}(1), \mathbf{x}(2), \cdots, \mathbf{x}(n_x)\}$ into fixed-length vector representation $\mathbf{c}$. This is done by sequentially processing the sentence with a recurrent network and taking the last hidden state $\mathbf{h}(n_x)$ as $\mathbf{c}$ (Equation 5.1).

$$\begin{aligned} \mathbf{h}(t) &= \text{LSTM}(\mathbf{x}(t), \mathbf{h}(t-1)) \\ \mathbf{c} &= \mathbf{h}(n_x) \end{aligned} \tag{5.1}$$

Then a decoder (also a deep LSTM network) outputs a translation sentence $o = \{o(1), o(2), \cdots, o(n_o)\}$ using $\mathbf{c}$ as its first input. As shown in Equation 5.2, in a similar way as the RNLM models that we saw in section 3.5,

the probability of outputting a word at step t is obtained as the vocabulary softmax of an affine transformation of the hidden state $\mathbf{s}(t)$.

$$\begin{aligned} \mathbf{s}(t) &= \text{LSTM}(\mathbf{o}(t-1), \mathbf{s}(t-1)) \\ p(o) &= \prod_{t=1}^{n_o} p(o(t)|\{o(1), \cdots, o(t-1)\}, \mathbf{c}) \\ p(o(t)|\{\mathbf{o}(1), \cdots, \mathbf{o}(t-1)\}, \mathbf{c}) &= \text{softmax}(W_o \mathbf{s}(t) + \mathbf{b}_o) \end{aligned} \tag{5.2}$$

A potential problem for this architecture is the fact that all the necessary information from an input sentence has to be encoded into the vector $\mathbf{c}$. Hence, this may make it difficult for the model to deal with long sentences. Instead the decoder is allowed to access the full sequence of the encoder's hidden states $\{\mathbf{h}(1), \mathbf{h}(2), \cdots, \mathbf{h}(n_i)\}$, as shown in Figure 5.1.

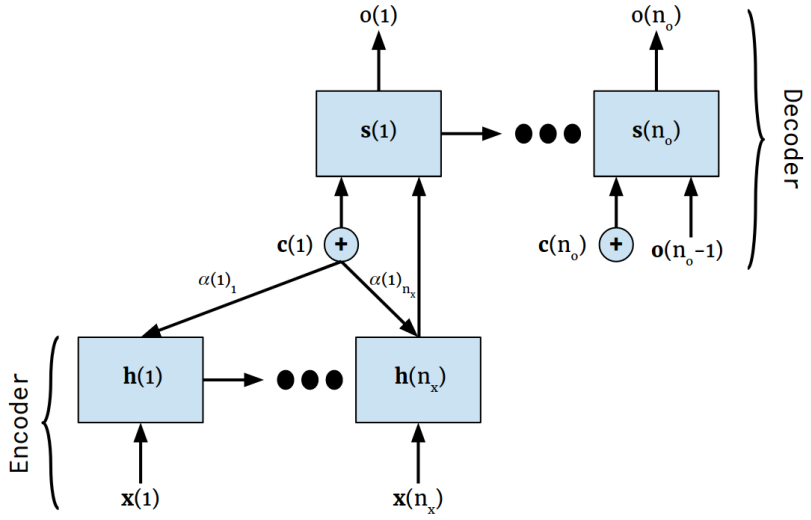

Figure 5.1: Attention NMT model.

In this new architecture the decoder is provided with an additional input, the context vector $\mathbf{c}(t)$. For each step t of the decoder, it is calculated in the following way:

$$\begin{aligned} \mathbf{e}(t)_k = e_{tk} &= \mathbf{v_a}^\top \tanh(W_a \mathbf{s}(t-1) + U_a \mathbf{h}(k)), \ \forall k \in 1, \cdots, n_i \\ \boldsymbol{\alpha}(t)_k = \alpha_{tk} &= \frac{e^{e_{tk}}}{\sum_j e^{e_{tj}}}, \ \forall k \in 1, \cdots, n_i \\ \mathbf{c}(t) &= \sum_{k=1}^{n_i} \alpha_{tk} \mathbf{h}(k) \end{aligned} \tag{5.3}$$

where $\mathbf{e}(t), \boldsymbol{\alpha}(t) \in \mathbb{R}^{n_i}$, $W_a \in \mathbb{R}^{h \times h}$, $U_a \in \mathbb{R}^{h \times 2h}$, $\mathbf{h}(t) \in \mathbb{R}^{2h}$ and $\mathbf{v_a}, \mathbf{s}(t), \mathbf{c}(t) \in \mathbb{R}^h$.

First, a vector of attention scores $\mathbf{e}(t)$ is calculated in order to see how much each input word should be "attended" with respect to the current decoder step. These are obtained with a single-layer multilayer perceptron. Next, the scores are normalized via a softmax operation producing the attention weights (can also be interpreted as probabilities) $\boldsymbol{\alpha}(t)$. Finally, the context vector $\mathbf{c}(t)$ is obtained as a convex combination of all the encoder's hidden states weighted by $\boldsymbol{\alpha}(t)$.

5.1.2 Classification

Since its introduction, attention has become very popular in NLP and has been applied successfully in other areas such as computer vision [20], originating the appearance of multiple variants. Thus, we introduce one possible classification (inspired by [21]) of such models attending to three different aspects:

- Depending on the attention's span:
  - **Global:** all inputs are taken into account.
  - **Local:** only a subset of all inputs is used.
- Depending on whether all inputs are allowed to contribute to the context:
  - **Soft:** context is built as a deterministic weighted combination of the inputs. This variant is smooth and differentiable so learning can be done by using standard backpropagation.
  - **Hard:** context is chosen as a single input that is selected stochastically. This has the drawback of requiring learning techniques such as Monte Carlo sampling or reinforcement learning.
- Depending on whether the content of inputs is used when computing the attention scores:
  - **Location-based:** the attention score for each input is obtained without taking into account their contents (e.g. $\mathbf{h}(t)$ in the setting described in the previous section).
  - **Content-based:** the input's contents are involved when calculating how relevant is each of them. This can be done in multiple ways and some examples are: dot product $\mathbf{s}(t)^\top \mathbf{h}(t)$, general $\mathbf{s}(t)^\top W \mathbf{h}(t)$ or "concatenation" $W\mathbf{s}(t) + U\mathbf{h}(t)$.

Using this classification, the model introduced in the previous section would be classified as *soft content-based attention*. In fact, the models that we introduce in the upcoming sections will fall into the same category.

5.1.3 Application to Language Modeling

An adaptation of the attention mechanism for RNLMs that we saw in 3.5 was presented in [22]. The first adaptation is that due to the extensive length of the texts used in language modeling, for practical reasons the attention span is limited to a window of the previous L hidden states, $Y(t)$. The second is that both the context vector $\mathbf{c}(t)$ and the current hidden state $\mathbf{h}(t)$ are blended into an "augmented" hidden state $\mathbf{h}^*(t)$. The remaining aspects of the model are left in the same way as in [18], as we can see in Equation 5.4.

$$\begin{aligned} Y(t) &= [\mathbf{h}(t-L); \cdots ; \mathbf{h}(t-1)] \\ \mathbf{e}(t) &= \mathbf{v_a}^\top \tanh(W_a Y(t) + U_a \mathbf{h}(t) \mathbf{1}^\top) \\ \boldsymbol{\alpha}(t) &= \text{softmax}(\mathbf{e}(t)) \\ \mathbf{c}(t) &= Y(t) \boldsymbol{\alpha}(t)^\top \\ \mathbf{h}^*(t) &= \tanh(W_c \mathbf{c}(t) + W_h \mathbf{h}(t)) \end{aligned} \tag{5.4}$$

where $Y(t) \in \mathbb{R}^{h \times L}$, $W_a, U_a, W_c, W_h \in \mathbb{R}^{h \times h}$, $\mathbf{v_a}, \mathbf{h}(t), \mathbf{s}(t), \mathbf{c}(t), \mathbf{h}^*(t) \in \mathbb{R}^h$ and $\mathbf{e}(t), \mathbf{1}, \boldsymbol{\alpha}(t) \in \mathbb{R}^L$.

5.2 Neural Continuous Cache

One of the first attempts at the LAMBADA dataset was introduced in [23]. The model takes inspiration from traditional cache models [24] and adapts this simple technique in order to be used on top of any RNLM. To accomplish that, a cache-like memory is introduced to store pairs $(\mathbf{h}(i), x(i+1))$ as illustrated in Figure 5.2.

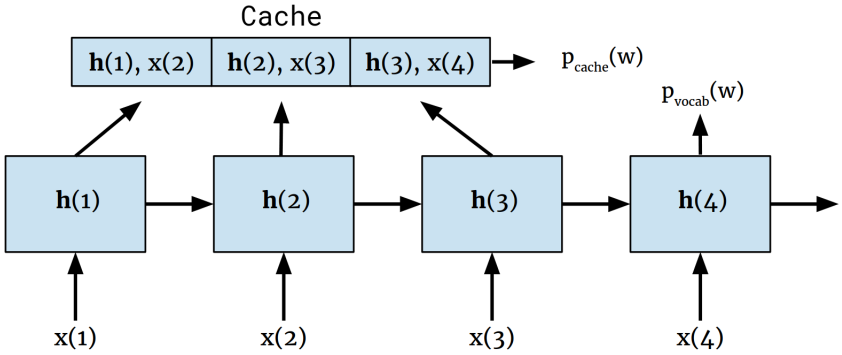

Figure 5.2: Neural Continuous Cache Model.

Each hidden representation $\mathbf{h}(i)$ acts as the key of each cache entry, which

contains the word $x(i+1)$ (the correct output for timestep i). Furthermore, the hidden representations are exploited to define a probability distribution over the words in the cache; at timestep t, the probability for each word in the cache is expressed as:

$$p_{\text{cache}}(w|\mathbf{h}(1\dots t)) = \sum_{i=1}^{t-1} \mathbb{1}_{w=x(i+1)} e^{\theta \mathbf{h}(t)^\top \mathbf{h}(i)} \tag{5.5}$$

where $\mathbf{h}(1\dots t)$ represents the sequence of hidden states for times 1 to t and $\theta \in \mathbb{R}$ controls the sharpness of the distribution ($\theta = 0$ would lead to a uniform distribution over all the words in the cache).

It can be observed that this formulation resembles the one from the attention models that we have introduced in the previous sections. The "score" of each word in the cache is obtained as the dot product $\mathbf{h}(t)^\top \mathbf{h}(i)$ (it is worth noting that this scoring mechanism doesn't require any parametrization). The distribution $p_{\text{cache}}(w)$ is then obtained by normalizing these scores with a softmax. To obtain the final output probability of the model, we blend the cache distribution with our usual vocabulary distribution (probabilities for all the words contained in the vocabulary) by means of linear interpolation:

$$p(w|\mathbf{h}(1\dots t)) = (1-\lambda) p_{\text{vocab}}(w|\mathbf{h}(t)) + \lambda p_{\text{cache}}(w|\mathbf{h}(1\dots t)) \tag{5.6}$$

where λ is a fixed hyperparameter that controls the weight of each distribution and is chosen by optimizing performance on a validation set.

In summary, the neural continuous cache constitutes a simple yet powerful technique for RNLMs to adapt to the recent history of words. It doesn't require to be trained and thus, it can be applied on top of any pretrained model using big cache sizes.

5.3 Pointer Sentinel Mixture Model (PSMM)

Presented in [25], this model is characterized by the introduction of a pointer component (inspired by pointer networks [26]) in addition to the usual vocabulary softmax. As we will see, this is equivalent to the neural continuous cache detailed in section 5.2. However the main difference lies in the fact that the interpolation weight is dynamically calculated, allowing the model to learn when to use each component. Similar to pointer networks, the pointer component keeps track of the previous inputs. However rather than directly outputting one of the previous inputs (as pointer networks do), the pointer component produces a probability distribution over them as illustrated in Figure 5.3.

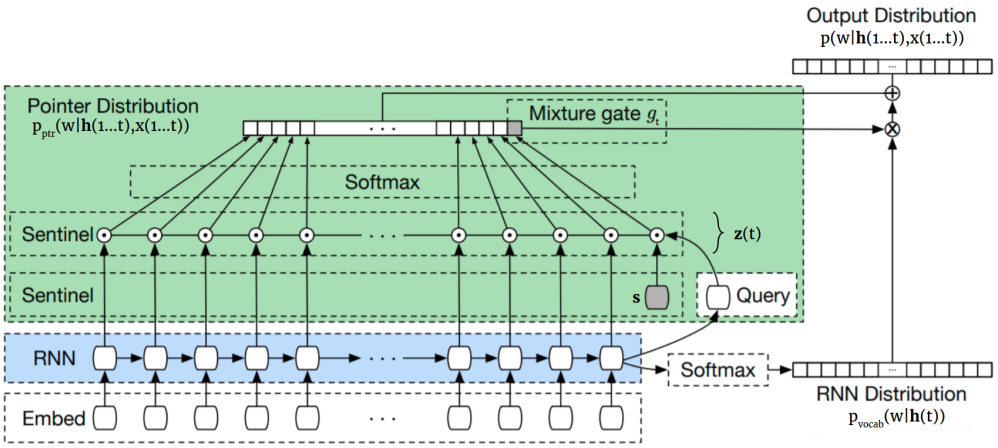

Figure 5.3: Pointer Sentinel Mixture Model architecture [25].

Given an input x that consists of the sequence $\{w_1, \cdots, w_t\}$ of input word IDs up to t and $y(t)$ the ID of the correct word that has to be predicted w_{t+1}, in theory the pointer component would take into account the whole input. In practice due to memory limitations only a window of the L most recent words is for the pointer to match against. Thus both the hidden states and the respective word IDs inside the window need to be stored, as shown in Equation 5.7.

$$\begin{aligned} M(t) &= \Big[\mathbf{h}(t); \quad \cdots ; \quad \mathbf{h}(t-L+1)\Big] \\ \mathbf{m}(t) &= [w_t; \cdots ; w_{t-L+1}] \end{aligned} \tag{5.7}$$

where $M(t) \in \mathbb{R}^{H \times L}$ and $\mathbf{m}(t) \in \mathbb{R}^{L}$.

In order to obtain the attention scores $\mathbf{z}(t)$ over the window, the easiest way is to compute the dot product between the current hidden state $\mathbf{h}(t)$ and all the hidden states in the window (like in section 5.2). However the window also contains $\mathbf{h}(t)$ as this word may be repeated (w_{t+1} is equal to w_t), and as the dot product of a vector with itself results in its magnitude squared, the attention scores would be biased towards the last word. This is avoided by projecting $\mathbf{h}(t)$ into a query vector $\mathbf{q}(t)$ (Equation 5.8).

$$\begin{aligned} \mathbf{q}(t) &= \tanh(W\mathbf{h}(t) + \mathbf{b}) \\ \mathbf{z}(t) &= \mathbf{q}(t)^{\top} M(t) \end{aligned} \tag{5.8}$$

where $\mathbf{q}(t), \mathbf{b} \in \mathbb{R}^{H}$, $W \in \mathbb{R}^{H \times H}$ and $\mathbf{z}(t) \in \mathbb{R}^{L}$.

As mentioned at the beginning of this section, the weight for interpolating both components is calculated dynamically at each timestep. To do this, a

trainable sentinel vector $\mathbf{s}$ is also multiplied with the query vector to obtain the score for using the vocabulary component rather than the pointer. Once we normalize with the softmax, we obtain the gate g_t that will weight the two components:

$$\begin{aligned} \mathbf{a}(t) &= \text{softmax}([\mathbf{z}(t); \mathbf{q}(t)^\top \mathbf{s}]) \\ g_t &= \mathbf{a}(t)_{L+1} \end{aligned} \tag{5.9}$$

where $\mathbf{a}(t) \in \mathbb{R}^{L+1}$, $\mathbf{s} \in \mathbb{R}^h$ and $g_t \in \mathbb{R}$.

It can be seen in Equation 5.9 that the values $\mathbf{a}(t)_{1:L}$ correspond to the pointer probabilities where the probability mass assigned to g_t has been subtracted from the pointer. Hence, the final normalized pointer probabilities are:

$$\begin{aligned} p_{\text{ptr}}(w|\mathbf{h}(1 \dots t)) &= \frac{1}{1 - g_t} \mathbf{a}(t)_{1:L} \\ p_{\text{ptr}}(w = i|\mathbf{h}(1 \dots t)) &= \frac{1}{1 - g_t} \sum_{k \in I(i,\, \mathbf{m}(t))} \mathbf{a}(t)_k \ \in \ \mathbb{R} \end{aligned} \tag{5.10}$$

where $p_{\text{ptr}}(w|\mathbf{h}(1 \dots t)) \in \mathbb{R}^L$, $p_{\text{ptr}}(w = i|\mathbf{h}(1 \dots t)) \in \mathbb{R}$ and $I(i,\ \mathbf{m}(t))$ results in all positions where $\mathbf{m}(t)$ (the window of previous word IDs) is equal to i.

Once the pointer probabilities and the gate have been calculated, the output distribution is obtained as:

$$\begin{aligned} p(w = i|\mathbf{h}(1 \dots t)) &= g_t \, p_{\text{vocab}}(w = i|\mathbf{h}(t)) + (1 - g_t) \, p_{\text{ptr}}(w = i|\mathbf{h}(1 \dots t)) \\ &= g_t \, p_{\text{vocab}}(w = i|\mathbf{h}(t)) + \sum_{k \in I(i,\, \mathbf{m}(t))} \mathbf{a}(t)_k \end{aligned} \tag{5.11}$$

The loss function used to optimize the model (Equation 5.12) is made up of two parts: the first one is the traditional cross-entropy loss used in RNLMs. The second part supervises the pointer component and makes sure that if any probability mass is taken from the vocabulary softmax and given to the pointer, it was the right thing to do (i.e. the correct output $y(t)$ appears one or more times in the window and the pointer assigns its probability to these positions).

$$\begin{aligned}\mathcal{L}(\theta) &= -\log(p(w = y(t)|\mathbf{h}(1\dots t)) - \log(g_t + \sum_{i \in I(y(t),\, x(t))} \mathbf{a}(t)_i) \\ &= -\log(g_t\, p_{\text{vocab}}(w = y(t)|\mathbf{h}(1\dots t)) + \sum_{k \in I(y(t),\, x(t))} \mathbf{a}(t)_k) \\ &\quad -\log(g_t + \sum_{i \in I(y(t),\, x(t))} \mathbf{a}(t)_i)\end{aligned} \tag{5.12}$$

5.4 Softmax Mixture Model (SMM)

To conclude this chapter we introduce our proposed model called "Softmax Mixture". Inspired by the models introduced in the previous sections, we also use two components that are blended with a dynamically calculated weight. In our case both components are of the same type, softmax distributions over the whole vocabulary (as in section 3.5).

Similar to [15], we define a binary variable z_t that indicates whether the output word comes from one of the two distributions. Then, a switching network that takes as input the current hidden state $\mathbf{h}(t)$ outputs a scalar probability $p(z_t = 1|\mathbf{h}(t))$:

$$p(z_t = 1|\mathbf{h}(t)) = \sigma(f(\mathbf{h}(t); \theta)) \tag{5.13}$$

where $\sigma(\cdot)$ is the sigmoid function and $f(\cdot)$ is a multilayer perceptron.

This probability is then used as the interpolation weight that controls the blending of the two distributions. With this we hope that the model learns two differentiated vocabulary distributions and when to use them (based on the hidden representation $\mathbf{h}(t)$).

$$\begin{aligned}p(w|\mathbf{h}(t)) &= p(z_t = 1|\mathbf{h}(t))p(w|\mathbf{h}(t), z_t = 1) \\ &+(1 - p(z_t = 1|\mathbf{h}(t)))p(w|\mathbf{h}(t), z_t = 0)\end{aligned} \tag{5.14}$$

Finally, in addition to the cross-entropy loss term used to optimize the RNLM we introduce the sigmoid cross-entropy loss of the switching network to train it in a supervised fashion, as shown in Equation 5.15.

$$\begin{aligned}\mathcal{L}(\theta) &= -\log(p(w|\mathbf{h}(t))) \\ -\lambda(z_t \log(p(z_t = 1|\mathbf{h}(t))) &+ (1 - z_t) \log(p(z_t = 0|\mathbf{h}(t)))\end{aligned} \tag{5.15}$$

where λ is a hyperparameter that controls how much weight is given to the switch network loss.

Chapter 6

Experiments and Results

experiments...

Finish experiments and results

Maybe mention implementation detail of only calculating $p(y(t))$ when training?

When showing the PSMM mention the initialization of $mathbfs$ and the issue with the gradient.

		All		Names	
		Train	Dev	Train	Dev
Basic	Baseline	42.5	61.5	1000	130K
	Mixture ($\lambda = 100$)	55	69	4000	120K
Input dropout	Baseline	52.5	63	2000	140K
	Mixture ($\lambda = 100$)	65	72	8000	120K
State dropout	Baseline	48	62.5	1000	120K
	Mixture ($\lambda = 100$)	62.5	73	6000	120K
Output dropout	Baseline	62.5	65	5000	150K
	Mixture ($\lambda = 100$)	80	73	20K	80K
L2 regularization ($\beta = 0.01$)	Baseline	100	125	10K	400K
	Mixture ($\lambda = 100$)	107.5	119	8000	120K

Table 6.1: Perplexity

Chapter 7

Conclusion

brief remarks...

Finish conclusion (1/2 page)

7.1 Achieved Results

results...

Finish achieve results (1/2 page)

7.2 Future Work

future work...

Finish future work (1/2 page)

Bibliography

[1] Stanley F Chen and Joshua Goodman. An empirical study of smoothing techniques for language modeling. In *Proceedings of the 34th annual meeting on Association for Computational Linguistics*, pages 310–318. Association for Computational Linguistics, 1996.

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

[3] Frederic Morin and Yoshua Bengio. Hierarchical probabilistic neural network language model. In *Aistats*, volume 5, pages 246–252, 2005.

[4] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *ICLR Workshop*, 2013.

[5] Tomas Mikolov, 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*, pages 3111–3119, 2013.

[6] Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[7] Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *Interspeech*, volume 2, page 3, 2010.

[8] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[9] Ilya Sutskever. Training recurrent neural networks. *University of Toronto, Toronto, Ont., Canada*, 2013.

[10] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning*, pages 1310–1318, 2013.

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

[12] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *EMNLP*, 2014.

[13] Christopher Olah. Understanding lstm networks. http://colah.github.io/posts/2015-08-Understanding-LSTMs/. Accessed: 2017-10-17.

[14] Rafal Jozefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. Exploring the limits of language modeling. *arXiv preprint arXiv:1602.02410*, 2016.

[15] Caglar Gulcehre, Sungjin Ahn, Ramesh Nallapati, Bowen Zhou, and Yoshua Bengio. Pointing the unknown words. *In Proceedings of ACL*, 2016.

[16] Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Quan Ngoc Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernández. The lambada dataset: Word prediction requiring a broad discourse context. *In Proceedings of ACL*, 2016.

[17] Yukun Zhu, Ryan Kiros, Rich Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *Proceedings of the IEEE international conference on computer vision*, pages 19–27, 2015.

[18] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *ICLR*, 2015.

[19] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[20] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International Conference on Machine Learning*, pages 2048–2057, 2015.

[21] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *EMNLP*, 2015.

[22] Michał Daniluk, Tim Rocktäschel, Johannes Welbl, and Sebastian Riedel. Frustratingly short attention spans in neural language modeling. *ICLR*, 2017.

[23] Edouard Grave, Armand Joulin, and Nicolas Usunier. Improving neural language models with a continuous cache. *arXiv preprint arXiv:1612.04426*, 2016.

[24] Roland Kuhn and Renato De Mori. A cache-based natural language model for speech recognition. *IEEE transactions on pattern analysis and machine intelligence*, 12(6):570–583, 1990.

[25] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *ICLR*, 2017.

[26] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems*, pages 2692–2700, 2015.

[27] Robin Jia and Percy Liang. Adversarial examples for evaluating reading comprehension systems. *In Proceedings of EMNLP*, 2017.

[28] Zichao Yang, Phil Blunsom, Chris Dyer, and Wang Ling. Reference-aware language models. *arXiv preprint arXiv:1611.01628*, 2016.

[29] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.

[30] Yarin Gal and Zoubin Ghahramani. A theoretically grounded application of dropout in recurrent neural networks. In *Advances in neural information processing systems*, pages 1019–1027, 2016.

[31] Stanislau Semeniuta, Aliaksei Severyn, and Erhardt Barth. Recurrent dropout without memory loss. *arXiv preprint arXiv:1603.05118*, 2016.

[32] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. Regularizing and optimizing lstm language models. *arXiv preprint arXiv:1708.02182*, 2017.

[33] Hakan Inan, Khashayar Khosravi, and Richard Socher. Tying word vectors and word classifiers: A loss framework for language modeling. *ICLR*, 2017.

[34] Daniel Jurafsky and James H. Martin. *Speech and Language Processing (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2009.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.