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

Language models

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.

Acknowledgments

First, I want to thank my supervisor, Prof.Thomas Hofmann, for allowing me to do my thesis in the Data Analytics laboratory. I am also thankful for his early comments that helped steering the project in the right direction.

I would like to express my gratitude to Florian Schmidt and Paulina Grnarova for providing me such an interesting and challenging thesis topic. I further thank them for the insightful discussions that we had all along the project and for helping me proofreading this report.

I thank all my friends that made me think of Zurich as a second home.

Finally, I would like to thank my family and specially my parents. Without your unconditional support I would not be writing this thanking words in the first place.

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 gives a brief overview of the state-of-the-art techniques used for word-level neural language modeling.

Recurrent neural network language models (RNNLM) [1] introduced by Mikolov et al. (2010) have become ubiquitous in modern language models. RNNLMs incorporate the advantages of previous attempts at neural language modeling [2]: Namely projecting words into low dimensional space solving smoothing implicitly and being trained with standard backpropagation techniques. In addition to that, recurrent networks have the ability to learn how to compress arbitrary long histories into low dimensional space. Jozefowicz et al. (2016) reported the performance gains achieved by these architectures, outperforming competing models when trained on large scale datasets [3].

Since then, several improvements have been proposed to improve on vanilla RNNLMs:

Kim et al. (2015) introduced a novel method that only relies on character-level information while still making predictions on the word level [4]. Word characters are processed by a one-dimensional convolutional layer and fed through a highway network in order to obtain the word representations. Among their advantages, character-aware models require fewer parameters by removing the need for trainable word embeddings and improve over their word-level counterparts when applied to morphologically rich languages.

On a different note, using large vocabularies can be sometimes unfeasible as calculating the output probability of a word implies computing the whole distribution due to the normalization necessary for the softmax operation, which makes training prohibitively slow. Mnih & Hinton (2009) employed a hierarchical representation of the vocabulary in the form of a binary tree in which the output probability of a word is directly encoded in the path specified by the hierarchy [5]. Another possible approach relies on Noise Contrastive Estimation (NCE), which introduces a surrogate binary classification task in

which a classifier is trained to discriminate between the true data and samples coming from a noise distribution [6]. In both cases normalization is rendered unnecessary, which makes this techniques appealing for large scale models.

Regarding regularization, Zaremba et al. (2014) were one of the first to successfully apply dropout to RNNs [7]. A random binary mask is only applied to the non-recurrent connections, the cell's inputs and outputs. In [8], Gal & Ghahramani (2016) provided a bayesian interpretation for dropout (known as variational dropout) and use it to argue that the mask should be the same for all time steps and also applied to the recurrent connections. A different approach called zoneout was taken by Krueger et al. (2016) in [9], where randomly selected neurons of the hidden state are not updated (in contrast with being dropped). More recently Merity et al. (2017) introduced another regularization variant where dropout is applied to the hidden-to-hidden recurrent weights rather than directly to the hidden states. In combination with other techniques, this method has achieved state-of-the-art perplexities on several datasets [10].

Following the work in deep RNNs, recurrent highway networks (RHN) [11] introduced by Zilly et al. (2016) extend LSTMs to enable the learning of deep recurrent state transitions. This is achieved by modifying the LSTM cell to allow multiple hidden state updates per time step.

Inan et al. (2016) made a case for sharing the weights between the embeddings and the softmax layer [12], which improves previous results and significantly reduces the amount of trainable parameters. This technique is theoretically motivated and takes advantage of the metric encoded into the space of word embeddings to generate a more informed target distribution. This prevents the model from having to learn a one-to-one correspondence between the input and output.

Similar to other areas of NLP, several memory augmented architectures have been proposed for language modeling. Daniluk et al. (2017) formulated an attention mechanism for RNNLMs [13] and explored to what extent is the learning of long-range dependencies improved. Grave et al. (2016) and Merity et. al (2016) proposed pointer based attention models where a probability distribution is generated over the history of recent words and blended with the distribution over the whole vocabulary. While the former does not need to be trained [14], the latter introduces additional parametrization that allows to calculate the interpolation weight of the two distributions dynamically [15].

A very interesting research direction has been recently proposed by Zoph & Le (2016), where a reinforcement learning agent is used to generate custom RNN cell architectures tailored to the specific task of language modeling [16]. This technique allows to find good neural network architectures automatically without the need of explicit human intervention.

With respect to the evaluation of RNNLMs, Melis et al. (2017) stressed the importance of careful hyperparemeter tuning and arrived at the conclusion that standard LSTM architectures, when properly regularized, outperform more recent models [17]. Paperno et al. (2016) raised their concerns with regards to the genuine language understanding of SOTA models and introduce LAMBADA [18], a dataset specifically designed to put this to the test. As this thesis is focused on LAMBADA, section 4.1 studies its characteristics in detail. Along these lines, Jia & Liang (2017) proposed a method to build adversarial examples for reading comprehension systems and found out that when evaluated on them, models produce very poor results under standard evaluation metrics [19].

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 Network Language Models (FFNNLM)

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 [20].

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 **b**) to this hidden representation to obtain **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.

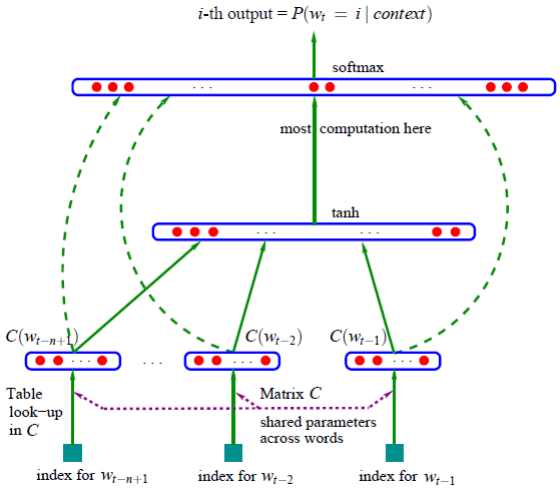


Figure 3.1: Feed-Forward NLM architecture [2].

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 **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** [5] 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. [21] 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 FFNNLM 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 [22]. 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** [23], 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, [21] 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 Network Language Models (RNNLM)

So far all the models that we have seen (n-grams and FFNNLM) 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. [1] 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 FFNNLMs.

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

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) [24] 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 [25] 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 [26], *"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) [27] and Gated Recurrent Units (GRU) [28]. 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 $W_f, W_i, W_o, W_f \in \mathbb{R}^{h \times h}$, $U_f, U_i, U_o, U_f \in \mathbb{R}^{h \times m}$, $\mathbf{f}(t), \mathbf{b}_f, \mathbf{i}(t), \mathbf{b}_i$, $\mathbf{o}(t), \mathbf{b}_o, \mathbf{c}(t), \mathbf{b}_c, \tilde{\mathbf{c}}(t) \in \mathbb{R}^h$, $\odot$ is the element-wise multiplication 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.

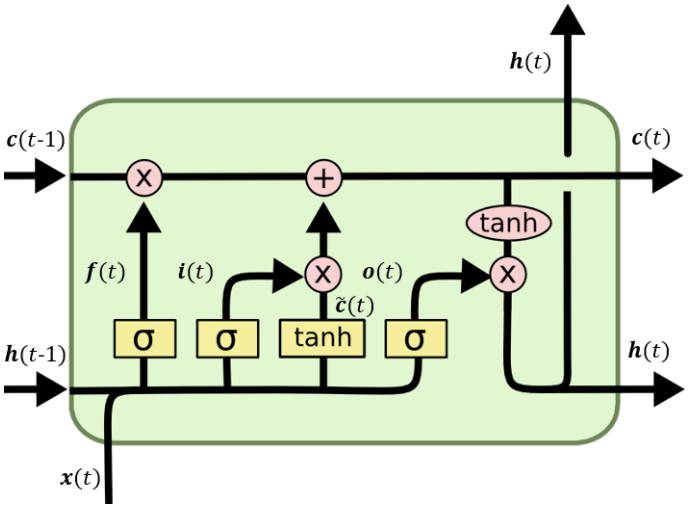


Figure 3.2: LSTM cell architecture [29].

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 [3].

Chapter 4

Rare Word Prediction

In this chapter we present the LAMBADA dataset, which will be used throughout this thesis to evaluate our different models. We also discuss the rare word problem and examine several RNN regularization techniques.

4.1 The LAMBADA dataset

This dataset was first introduced in [18] as a challenging test set specifically designed to probe the genuine language understanding of state-of-the-art NLP models. In the authors' words, *"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"*. Below we can find an example extracted from the dataset:

> **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

As illustrated in Figure 4.1, the dataset consists of narrative passages formed by a *context paragraph* (with an average length of 4.6 sentences) and a *target sentence*. The objective is to predict the last word of the target sentence (known as the *target word*). In this way, LAMBADA casts the complex task of

evaluating language understanding into the simple and general word prediction framework of language modeling.

4.1.1 Construction Process

LAMBADA was built using a large initial dataset, BookCorpus , which was then distilled into a difficult subset. The original dataset features 5325 unpublished novels (after duplicate removal) and 465 million words [30]. Novels were then randomly divided into equally-sized training and development+test partitions. Models tackling LAMBADA are intended to be trained on raw text from the training partition, which encompasses 2662 novels and more than 200 million words.

In order to obtain the LAMBADA passages, an automated filtering step was first applied to the development+test partition. Specifically, passages from the initial candidate set were discarded if the target word was given a probability ≥ 0.00175 by any of the four different standard language models (both neural and count based) that were used in this stage of the process.

To make it into the final dataset, the remaining passages were then evaluated by human subjects in a three-step process:

1. A human evaluator had to guess the target word correctly based on the whole passage (comprising the context and the target sentence).
2. A second human evaluator had to also guess the target word correctly in the same conditions.
3. Finally, ten human evaluators had to fail at guessing the target word having access to only the target sentence and 3 allowed attempts.

Due to the specifics of this process, the passages that finally were selected have the property of not being guessable by just relying on local context and require broader understanding, probing the long range capabilities of language models. The final development and test sets that constitute LAMBADA consist of 4869 and 5153 passages, respectively. Additionally, a control set containing 5000 unfiltered passages was also constructed to allow for comparisons between standard language modeling scenarios and LAMBADA.

4.1.2 Dataset Analysis

The authors theorize that the inspection of the LAMBADA data suggests that, in order for the target word to be predictable in a broad context only (which is the case of LAMBADA by design), it must be strongly cued in the broader discourse. Figure 4.2 compares the proportion of passages in the LAMBADA and control sets that include the target word in the context:

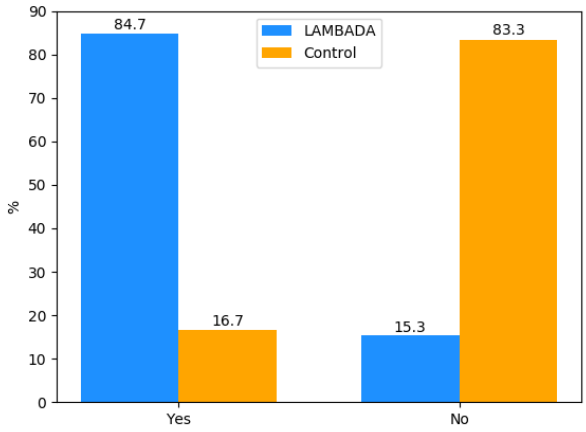


Figure 4.2: Target word appears in passage.

Indeed, in 84.7 % of the LAMBADA items the target word (or its lemma, extracted with `nltk`'s WordNetLemmatizer) is mentioned in the context in the context, compared to the 15.3 % for the control set. Furthermore, we can study the mention distance (understood as the number of words between the target word and its mention) distribution for LAMBADA. As illustrated in Figure 4.3, 73 % of the mentions are at a distance of more than 30 words (LAMBADA passages feature an average length of 75 words). This is specially important as many of the publicly available RNNLMs implementations are trained with TBPTT using k_2 somewhere between 20 and 35.

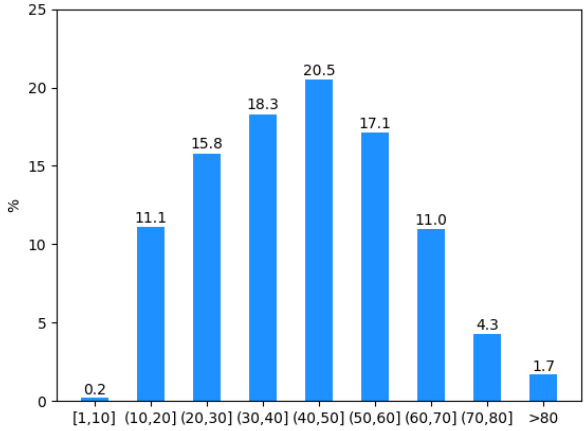


Figure 4.3: Target word appears in passage.

Moreover, it is also interesting to study the part of speech (PoS) tag distribution of the target words (also extracted with `nltk`'s default PoS tagger). As shown in Figure 4.4, proper nouns lead (48.1%) followed by common nouns (37%) and, far-off, verbs (7.7%). By comparing with control's PoS distribution, it is clear that proper nouns are over-represented in LAMBADA while the rest of the categories are downplayed.

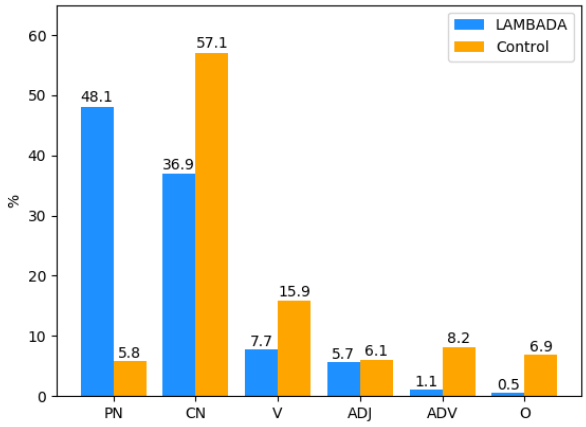


Figure 4.4: Target words' PoS distribution (PN=proper noun, CN=common noun, V=verb, ADJ=adjective, ADV=adverb, O=other).

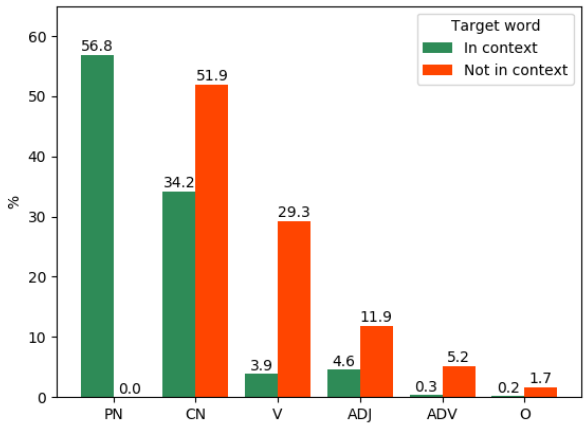


Figure 4.5: LAMBADA target words' PoS distribution.

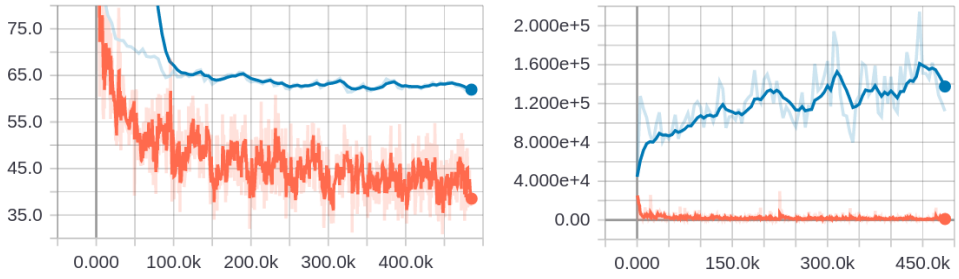
The authors argue that the reason for this bias is the fact that proper nouns are favored by the specific setup. Namely, in many cases the context clearly demands a referential expression and the constraint of predicting a single word excludes other possibilities such as noun phrases with articles. This hypothesis seems to be confirmed by Figure 4.5, which shows the PoS distribution for LAMBADA split depending on whether the target word appears in the context. It can be seen that explicit mention in the preceding discourse context is critical for proper nouns (when the target word doesn't previously appear in the passage, none of them are proper nouns), while the other categories can often be guessed without having been explicitly introduced.

To sum up, we can conclude that LAMBADA does indeed put to the test language models' capabilities to exploit **long-range dependencies**. Besides, it is also important to note the bias towards proper nouns present in the dataset. Proper nouns are used to refer to unique entities (e.g. the name of a specific character in a book) and due to their very nature, they are **rare words** (when compared to the rest of the training corpus). We will describe this phenomena in detail in the next section.

4.2 The Rare Word Problem

As we saw in chapter 3, a common approach followed by recent neural language models is to use a softmax output layer where each of the dimensions corresponds to a word in a predefined vocabulary.

This approach has a fundamental problem, known as the *rare word problem*. It is caused by some of the words in the vocabulary occurring much less frequently in the training set and thus, difficulting the learning of good representations for them and resulting in poor generalization performance.



(a) All words

(b) Only names

Figure 4.6: Perplexity traces on LAMBADA (training trace is in orange and development, in blue).

Figure 4.6 illustrates a real example of this problem. Figure 4.6a shows traces of average word perplexity over training batches and the development set for a vanilla LSTM language model trained on the LAMBADA training partition (more details in chapter 6). If we now focus on just rare words (in this specific example, character names), we can see in Figure 4.6b that the model is clearly overfitting on this subset of words. Intrinsically, rare words constitute a small fraction of the corpus and hence, their effect is averaged out when taking into account all words.

These results suggest that in general existing neural language models remain fundamentally lacking, failing to handle rare words.

4.3 RNN Regularization

To end this chapter, we will describe in detail several recent efforts for effectively regularizing and avoiding overfitting in recurrent neural networks:

4.3.1 Dropout

Neural networks are very expressive models that can learn very complicated input-output mappings which may in turn lead to overfitting. Ensemble methods such as model averaging (where we average the outputs of several estimators that have been trained independently) are known to nearly always improve their single base estimators and have a regularization effect as the variance of the resulting model is reduced. However, the idea of averaging the outputs of many separately trained neural networks is prohibitively expensive.

Dropout is an extremely effective and simple regularization technique introduced in [31]. It is implemented by only keeping a neuron active with some probability p (a hyperparameter), or setting it to zero otherwise. If we consider a network with n units, the application of dropout can generate 2^n different possible "reduced" networks (all of them sharing the same weights). In words of the authors, *"training a neural network with dropout provides a way of approximately training a collection of* 2^n *thinned networks with extensive weight sharing, where actually each thinned network gets trained very rarely, if at all".*

At test time, it is not feasible to explicitly average the predictions from exponentially many models. However, a very simple approximate averaging method is then used: no dropout is applied while testing and all weights are scaled by p (same probability as used during training). This has the goal of ensuring that for any unit, the expected output (under the dropout distribution used in training time) is the same as the actual output at test time. Given a unit's output y, its expected value is $\mathbb{E}[y] = py + (1-p)0 = py$.

In practice, we employ a variant known as **"inverted dropout"** where kept units are also scaled by $1/p$. This has the very appealing property of not requiring any additional operations during test time.

One of the first successful attempts at applying dropout for regularizing RNNs was described in [7]. The authors claim that dropout should only be applied to non-recurrent connections (namely, input and output units) by sampling a different dropout mask at each time step. Therefore it is ensured that information is only corrupted $L+1$ times (being L the number of layers of the RNN), as exemplified in Figure 4.7. In this way, the method can benefit from dropout regularization without sacrificing the valuable memorization ability of RNNs.

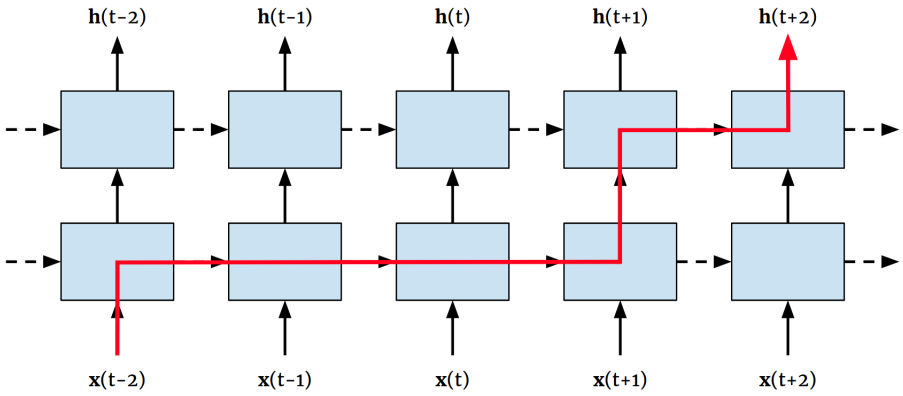


Figure 4.7: Example of information flow in a multilayer RNN with dropout (only applied at connections denoted with a solid arrow).

4.3.2 Variational Dropout

Following previous results at the intersection of Bayesian research and deep learning, [8] proposes a new dropout variant theoretically grounded on the results arisen from the application of approximate variational inference to probabilistic Bayesian RNNs.

Given a training dataset consisting of the sequence of inputs $\mathbf{X} = \{\mathbf{x}(1), \cdots, \mathbf{x}(N)\}$ and outputs $\mathbf{Y} = \{\mathbf{y}(1), \cdots, \mathbf{y}(N)\}$, Bayesian parametric regression aims to infer the parameters W of a function $\mathbf{y} = \mathbf{f}^W(\mathbf{x})$ (in our derivation, $\mathbf{f}^W(\cdot)$ is a neural network and W are its weight matrices). Following the Bayesian methodology, we place a prior distribution $p(W)$ over the weights (often a standard multivariate Gaussian) and define a likelihood distribution $p(\mathbf{y}|\mathbf{x}, W)$ (usually assumed to be a softmax likelihood) and a posterior distribution $p(W|\mathbf{X}, \mathbf{Y})$ given the observed dataset.

For a new input point $\mathbf{x}^*$, an output can then be predicted as shown in

Equation 4.1. However this is not feasible in practice as the posterior is not tractable in general.

$$\begin{aligned} p(W|\mathbf{X},\mathbf{Y}) &= \frac{p(\mathbf{Y}|\mathbf{X},W)p(W)}{p(\mathbf{Y})} \\ p(\mathbf{y}^*|\mathbf{x}^*,\mathbf{X},\mathbf{Y}) &= \int p(\mathbf{y}^*|\mathbf{x}^*,W)p(W|\mathbf{X},\mathbf{Y})dW \end{aligned} \tag{4.1}$$

To work around this problem, variational inference can be used to approximate the posterior with a variational distribution $q(W)$ and then minimize the Kullback–Leibler (KL) divergence between the approximating distribution and the full posterior (Equation 4.2).

$$\begin{aligned} \mathcal{L} = \mathrm{KL}(q(W)||p(W|\mathbf{X},\mathbf{Y})) &\propto -\int q(W)\log(p(\mathbf{Y}|\mathbf{X},W))dW + \mathrm{KL}(q(W)||p(W)) \\ &= -\sum_{i=1}^{N}\int q(W)\log(p(\mathbf{y}(i)|\mathbf{f}^W(\mathbf{x}(i))))dW + \mathrm{KL}(q(W)||p(W)) \end{aligned} \tag{4.2}$$

Going now to the specifics of RNNs, we can further define each input point $\mathbf{x}$ to be a sequence $[\mathbf{x}(1),\cdots,\mathbf{x}(T)]$ of length T, the model output as ${\mathbf{f}_\mathbf{y}}^W(\cdot)$ and the recurrent cell function as ${\mathbf{f}_\mathbf{h}}^W(\cdot)$ (both of them with their respective weight matrices). Then the integral from Equation 4.2 can be rewritten as:

$$\begin{aligned} &\int q(W)\log(p(\mathbf{y}|{\mathbf{f}_\mathbf{y}}^W(\mathbf{h}(T))))dW \\ &= \int q(W)\log(p(\mathbf{y}|{\mathbf{f}_\mathbf{y}}^W({\mathbf{f}_\mathbf{h}}^W(\mathbf{x}(T),\mathbf{h}(T-1)))))dW \\ &= \int q(W)\log(p(\mathbf{y}|{\mathbf{f}_\mathbf{y}}^W({\mathbf{f}_\mathbf{h}}^W(\mathbf{x}(T),{\mathbf{f}_\mathbf{h}}^W(\ldots{\mathbf{f}_\mathbf{h}}^W(\mathbf{x}(1),\mathbf{h}(0)\ldots)))))dW \\ &\overset{*}{\approx} \log(p(\mathbf{y}|{\mathbf{f}_\mathbf{y}}^{\hat{W}}({\mathbf{f}_\mathbf{h}}^{\hat{W}}(\mathbf{x}(T),{\mathbf{f}_\mathbf{h}}^{\hat{W}}(\ldots{\mathbf{f}_\mathbf{h}}^{\hat{W}}(\mathbf{x}(1),\mathbf{h}(0)\ldots)))))\,\text{with}\,\hat{W}\sim q(W) \end{aligned} \tag{4.3}$$

where $*$ means that the integral is approximated with Monte Carlo (MC) integration with a single sample.

Then the minimization objective can be formulated as:

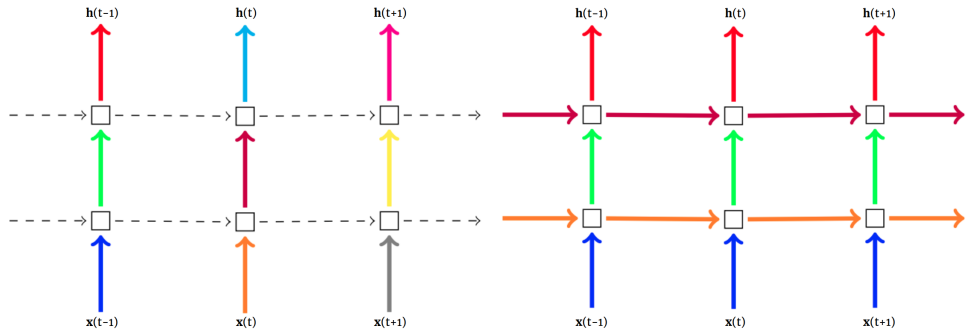
$$\begin{aligned} \mathcal{L} \approx -\sum_{i=1}^{N}\log(p(\mathbf{y}(i)|{\mathbf{f}_\mathbf{y}}^{\hat{W}}({\mathbf{f}_\mathbf{h}}^{\hat{W}_i}(\mathbf{x}(i,T),{\mathbf{f}_\mathbf{h}}^{\hat{W}_i}(\ldots{\mathbf{f}_\mathbf{h}}^{\hat{W}_i}(\mathbf{x}(i,1),\mathbf{h}(0)\ldots))))) \\ +\mathrm{KL}(q(W)||p(W)) \end{aligned} \tag{4.4}$$

It is important to note in Equation 4.4 that new weights are sampled for each point in the dataset. However, the weights are kept the same for all the time steps $t < T$ of every input sequence. Finally, the approximating distribution $q(W)$ is defined to factorize over the different weight matrices and their rows $\mathbf{w_k}$ as:

$$q(\mathbf{w_k}) = p\mathcal{N}(\mathbf{m_k}, \sigma^2\mathbf{I}) + (1-p)\mathcal{N}(\mathbf{0}, \sigma^2\mathbf{I}) \tag{4.5}$$

where $\mathbf{m_k}$ is a vector of variational parameters.

First we can see in Equation 4.5 that sampling from $q(W)$ is identical to applying dropout on the weight matrices. Second, implementing the described approximate inference procedure is equivalent to performing dropout in RNNs with the same network units dropped at each time step, randomly dropping inputs, outputs, and recurrent connections. Figure 4.8 graphically illustrates the main differences of this method with respect to the one introduced in the previous section.



(a) "Naive" (b) Variational

Figure 4.8: Comparison of dropout techniques (only applied at connections denoted with a solid arrow, with different colors meaning different masks) [8].

4.3.3 Zoneout

An alternative regularization method that only applies to the recurrent connections of an RNN was introduced in [9]. Rather than setting a unit's activation to 0 with probability $1-p$ as in dropout, zoneout replaces the unit's activation with its value from the previous time step. We can see in Equation 4.6 what is the formulation of zoneout for the LSTM cell:

$$\begin{aligned}\mathbf{c}(t) &= \mathbf{d_c}(t) \odot \mathbf{c}(t-1) + (1 - \mathbf{d_c}(t)) \odot (\mathbf{f}(t) \odot \mathbf{c}(t-1) + \mathbf{i}(t) \odot \tilde{\mathbf{c}}(t)) \\ \mathbf{h}(t) &= \mathbf{d_h}(t) \odot \mathbf{h}(t-1) \\ &+ (1 - \mathbf{d_h}(t)) \odot (\mathbf{o}(t) \odot \tanh(\mathbf{f}(t) \odot \mathbf{c}(t-1) + \mathbf{i}(t) \odot \tilde{\mathbf{c}}(t)))\end{aligned} \tag{4.6}$$

where $\mathbf{d_c}(t)$ and $\mathbf{d_h}(t)$ are Bernoulli random vectors.

It as argued by the authors that compared with dropout, zoneout is appealing because it preserves information flow forwards and backwards through the network. This in turn seems to help with the vanishing gradient problem, as shown by their experimental results.

4.3.4 Weight Dropout

The recent work described in [10] opts for applying DropConnect [32] on the recurrent connections of a recurrent cell. For an LSTM, it is implemented by applying on the recurrent weight matrices (U_f, U_i, U_o, U_f from Equation 3.16) a Bernoulli mask sampled before each forward-backward pass. It is similar to variational dropout in the sense that the same mask is reused over multiple time steps. However by working directly on the weight matrices, "weight dropout" doesn't need to modify the formulation of the cell and is amenable to be used with highly optimized black box implementations.

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".

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 language 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 [33]. At that time, most of the proposed NMT models belonged to a family known as "encoder-decoders" [34]. 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 a 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 initial hidden state. As shown in Equation 5.2, in a similar way as the RNNLM 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, an attention layer is added to give the decoder access to the full sequence of the encoder's hidden states $\{\mathbf{h}(1), \mathbf{h}(2), \cdots, \mathbf{h}(n_x)\}$, as shown in Figure 5.1.

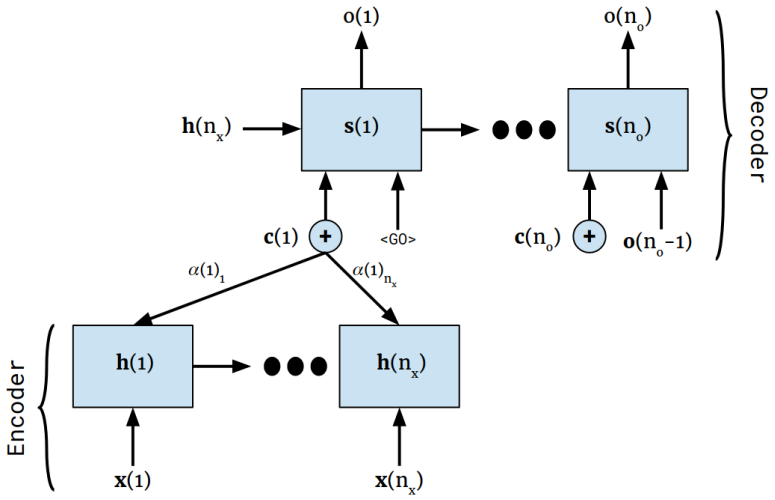


Figure 5.1: Attention NMT model.

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:

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

where $\mathbf{e}(t), \boldsymbol{\alpha}(t) \in \mathbb{R}^{n_x}$, $W_a \in \mathbb{R}^{h \times h}$, $U_a \in \mathbb{R}^{h \times 2h}$, $\mathbf{h}(t) \in \mathbb{R}^{2h}$ 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 Types of Attention

Since its introduction, attention has been applied successfully in other areas such as computer vision [35] and adapted for many other NLP tasks. In this section we introduce one possible classification (inspired by [36]) 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 RNNLMs that we saw in 3.5 was presented in [13]. The first modification 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 [33], as we can see in Equation 5.4.

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

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

5.2 Neural Continuous Cache

One of the first attempts at the LAMBADA dataset was introduced in [14]. The model takes inspiration from traditional cache models [37] and adapts this simple technique in order to be used on top of any recurrent language model. To accomplish that, a cache-like memory is introduced to store key-value pairs $(\mathbf{h}(i), x(i+1))$ as illustrated in Figure 5.2.

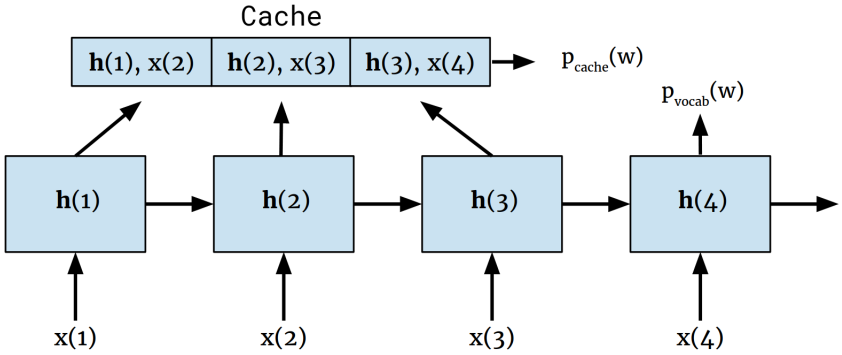


Figure 5.2: Neural Continuous Cache Model.

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 linear interpolation 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 RNNLMs to adapt to the recent history of words. By design, 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 [15], this model is characterized by the introduction of a pointer component (inspired by pointer networks [38]) 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.

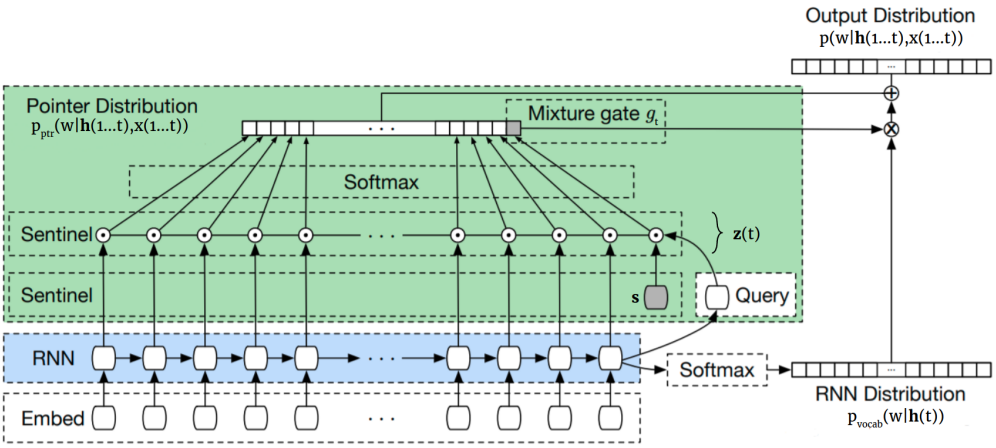


Figure 5.3: Pointer Sentinel Mixture Model architecture [15].

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.

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

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. Therefore the authors chose to avoid this 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 of $\mathbf{m}(t)$ (the window of previous word IDs) that are 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 RNNLMs. An intuitive explanation for the second part is that it 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 Our approach: Softmax Mixture Model (SMM)

To conclude this chapter we introduce our proposed model called the "Softmax Mixture". Inspired by the models detailed 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 [39], we define a binary variable z_t that indicates which distribution (out of two) produced the output word. 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 architecture we want to fulfill two objectives: being able to distinguish the correct source of each word (by means of the switching network) and fitting two differentiated output distributions that match the characteristics of each source.

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

Finally, in addition to the cross-entropy loss term used to optimize the RNNLM 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

This chapter summarizes the experiments carried out for our proposed model, the Softmax Mixture, and several pointer based models. We also discuss the results obtained when applying these models on the LAMBADA dataset.

All the experiments described in this chapter have been implemented using Tensorflow v1.4 [40]. The code is publicly available at https://github.com/moisestg/rare-lm.

6.1 Softmax Mixture Model

6.1.1 Experimental Setup

The original LAMBADA dataset contains lowercased text. As will be described in the upcoming experiments section, part of our preprocessing requires PoS tagging the text. Although there are caseless PoS taggers available, in order to maximize the accuracy of this step we opted to use the capitalized version of the dataset[1].

Additionally, we delimited each novel in the corpus with a special a tag and used that to randomize the order in which the novels are fed to the model for each epoch.

Furthermore, following the recommendations from [15] we modified our base code resulting in a much faster implementation: During the training phase it is only necessary to compute the softmax probability of the target word (rather than the whole vocabulary) in order to compute the cross-entropy loss, as the observed distribution is a one-hot encoding of the correct output.

Finally, Table 6.1 summarizes the specific training configuration used in the following set of experiments:

[1] Available at http://clic.cimec.unitn.it/lambada/train-novels-capitalized.txt.gz

Training data	1/3 of all the training novels
Development data	100K contiguous sentences from the remaining 2/3
Vocabulary size	44636
Word embeddings	150-d word2vec (pretrained on the training data)
Optimizer	ADAM with learning rate of 0.001
TBPPT	$k_1 = 1$ and $k_2 = 20$
Batch size	64
Maximum global L2 norm	8
Training epochs	10

Table 6.1: Training configuration for section 6.1.

6.1.2 Regularization Experiments

As described in section 5.4, the Softmax Mixture model has the ability to encode two differentiated output distributions that are dynamically blended by the switching network. In order to train the switching component in a supervised fashion, we need to tag each word in the training set with its corresponding source.

Motivated by our initial analysis of LAMBADA, we decided to tag peoples' names as they are a very specific subset of rare words that a vanilla softmax output layer fails to handle. By using a predefined list[2], we also made sure to reduce the amount of false positives (at the expense of missing some rare or fictitious names).

In our first batch of experiments we started exploring the space of hyperparameters by finding an optimal value for λ, the weight of the loss term in charge of the training supervision of the switching component. Figure 6.1 shows the evolution of the development perplexity during training for the explored range of values. We can see that values of λ up to 100 produce a similar behavior on the overall perplexity of the model. Higher values start to noticeably hinder the learning process as the sigmoid loss of the switching component starts to overcome the cross-entropy loss.

[2] Available at https://deron.meranda.us/data/.

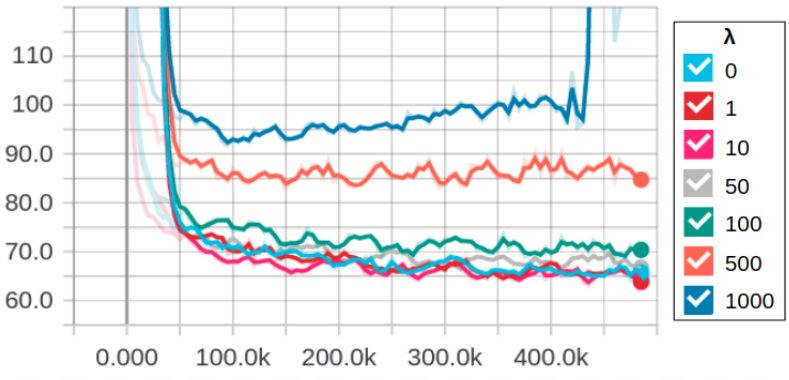


Figure 6.1: Development perplexity traces for different values of lambda.

As already discussed in section 4.2, the behavior of rare words is masked when studying the overall performance of a model. A set of preliminary experiments where we studied the performance of names in isolation revealed the existence of two differentiated regimes. In particular, Figure 4.6 shows the training traces for baseline language model consisting of a single-layered LSTM with 512 hidden units (following the recommendations from [18]), where a clear overfitting trend is observed for names.

These discoveries led us to run a collection of experiments summarized in Table 6.2, where we studied the behavior of the baseline and one specific configuration of our model ($\lambda = 100$) when applying different regularization techniques on them:

		All words		**Names**	
		Train	**Dev**	**Train**	**Dev**
No regularization	**Baseline**	42.5	61.5	1K	130K
	Mixture (λ= 100)	55	69	4K	120K
Input dropout	**Baseline**	52.5	63	2K	140K
	Mixture (λ= 100)	65	72	8K	120K
Hidden state dropout	**Baseline**	48	62.5	1K	120K
	Mixture (λ= 100)	62.5	73	6K	120K
Output dropout	**Baseline**	62.5	65	5K	150K
	Mixture (λ= 100)	80	73	20K	80K
L2 regularization (weight= 0.01)	**Baseline**	100	125	10K	400K
	Mixture (λ= 100)	107.5	119	8K	120K

Table 6.2: Perplexity for different regularization techniques.

Specifically we tested variational dropout applied separately on the input,

outputs and hidden state units[3] using $p = 0.5$. We also tried including an L2 regularization term applied on the weights of the softmax layers. The results indicate that none of the approaches is able to improve on the unregularized baseline in terms of overall perplexity (61.5 on the development set). However if we focus our attention on names, we can see that our model in combination with output dropout reduces the perplexity of the baseline in 40%.

Figure 6.2 shows some additional experiments that focus on the effect of output dropout on our model, featuring different combinations of λ and p and splitting the results depending on whether a word was tagged as a name. In general we observe that for words that are not names, the models benefit from lighter (higher keep probability) dropout while for names, all configurations exhibit a very similar behavior.

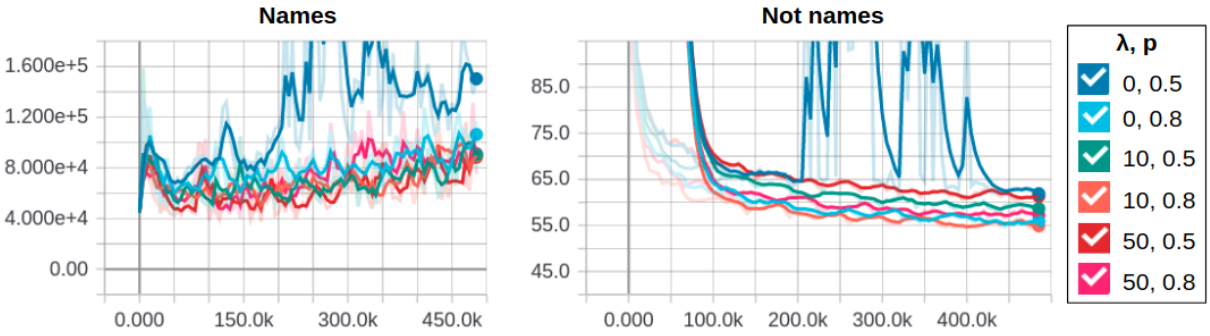


Figure 6.2: Development perplexity traces for different combinations of lambda and output keep probability.

		Names	Not names
$\boldsymbol{\lambda = 10}$	$\mathbf{p = 0.5}$	62K	59
	$\mathbf{p = 0.8}$	65K	54
	$\mathbf{p_1} = 0.8, \mathbf{p_2} = 0.625$	61K	54
$\boldsymbol{\lambda = 50}$	$\mathbf{p = 0.5}$	60K	61
	$\mathbf{p = 0.8}$	61K	57
	$\mathbf{p_1} = 0.8, \mathbf{p_2} = 0.625$	60K	57

Table 6.3: Development perplexity comparison between regular and splitted dropout.

As already discussed, all previous experiments have shown differentiated

[3] Tensorflow's `DropoutWrapper` implementation of variational dropout for the hidden state is only functional from v1.4 onwards.

dynamics for names and not names. Inspired by this, we decided to try a modified version of variational dropout, called *splitted dropout*, to be used in combination with our softmax mixture model. The basic idea is being able to apply different regularization strengths for the output distributions of names and not names.

The procedure is summarized in Equation 6.1: We first sample two dropout masks $\mathbf{m_1}, \mathbf{m_2}$ (with keep probabilities p_1 and p_2, respectively) of the same size of the hidden states. The mask that will be use for not names is equal to $\mathbf{m_1}$. To obtain the mask for names, we calculate the element-wise product of $\mathbf{m_1}$ and $\mathbf{m_2}$. With this approach we ensure that the zeros from the not names' mask are preserved, maximizing the overlap of the dropped units in both masks. We then apply both masks to the sequence of hidden states. The sequence with lighter dropout (keep probability of p_1) is then fed to the softmax layer encoding the distribution of not name words while the sequence with heavier dropout (keep probability of $p_1 p_2$) is fed to the layer names' softmax layer.

$$\begin{gathered} \mathbf{m_1} \sim \text{Dropout}(p_1) \\ \mathbf{m_2} \sim \text{Dropout}(p_2) \\ \mathbf{m}_{\text{not names}} = \mathbf{m_1},\ \mathbf{m}_{\text{names}} = \mathbf{m_1} \odot \mathbf{m_2}) \\ p(m_{\text{names}} \neq 0) = p(m_1 \neq 0 \wedge m_2 \neq 0) = p_1 p_2 \end{gathered} \tag{6.1}$$

where $\mathbf{m_1}, \mathbf{m_2} \in \mathbb{R}^h$.

Table 6.3 compares the results obtained with regular variational dropout and splitted dropout. Selecting $p_1 = 0.8$ and $p_2 = 0.625$ is equivalent to applying dropouts with keep probabilities of 0.8 and 0.5 for not names and names, respectively. We can observe that the proposed variant allows for stronger dropout of the names distribution without impacting the overall performance.

6.1.3 Switch Component Experiments

We have just seen that none of the tested configurations for the softmax mixture managed to improve on the baseline in terms of overall perplexity. In this section, we detail a series of experiments that were carried out to determine the performance of the switching component and check if this may be cause for the results obtained so far.

As a first step, we decided to study the performance of the classifier in isolation. To do that we used a confusion matrix, one of the standard evaluation tools for classifier evaluation. Table 6.4 shows the results obtained by the classifier on the development set:

		Prediction	
		Name	**Not name**
Ground Truth	**Name**	20859	20292
	Not name	123220	1874029

Table 6.4: Confusion matrix for the name classifier (threshold=0.5).

First of all, we can observe the strong imbalance present in our dataset, with names constituting only 2% of all words. This prevented us from using biased metrics when applied to highly-skewed data such as accuracy (93% in this example). Instead we defined names to be our positive class and used precision, recall and F_1 score as our metrics.

$$\begin{aligned} P &= \frac{TP}{TP+FP} = \frac{20859}{20859+123220} = 0.14 \\ R &= \frac{TP}{TP+FN} = \frac{20859}{20859+20292} = 0.51 \\ F_1 &= 2\frac{P \cdot R}{P+R} = 0.23 \end{aligned} \tag{6.2}$$

The results from Equation 6.2 clearly show that the performance of the classifier for names is very poor. In an attempt to mitigate this imbalance problem, several techniques were applied. Namely, undersampling of the majority class (by reducing the ratio of not names/names from its initial value of 50 to 1) and introducing a class weight for names in the loss function so errors made on names are more costly. However none of our efforts rendered any noticeable performance gains.

Qualitative inspection of the results seemed to suggest that independently from the imbalance in the data, predicting if an upcoming word is a name given the previous ones is inherently a hard problem. The classifier manages to predict names confidently only when they are strongly cued in the previous context. Examples of this would be the end of a character's dialogue line followed by the word "said" (which are quite common in literary texts such as LAMBADA) or after personal titles such as "Mr.". However, more general situations are much harder as common nouns and personal pronouns can be usually used in the same locations as names.

Based on our previous findings, we hypothesized that the switching component was the limiting factor for our softmax mixture model. In order to validate this, we ran an additional experiment where the output of the switching component $p(z_t = 1|\mathbf{h}(t))$ was substituted with the ground truth:

	Names	**All words**
Ground truth	5K	57
Ground truth and splitted dropout $\mathbf{p_1} = 0.8, \mathbf{p_2} = 0.625$	4K	55

Table 6.5: SMM perplexity with ground truth switching.

The results in Table 6.5 show that with this configuration, our model manages to improve on the baseline in overall perplexity (4.5 perplexity points lower down to 57), mainly due to the great reduction of perplexity for names. Additionally, applying the splitted dropout technique described earlier provides further performance gains.

As we detailed in section 5.4, our model aims to learn two differentiated output distributions for names and not names. Table 6.6 shows the top ten predictions made for several words tagged as names[4]. These examples show that our model successfully learns better suited output distributions for names, shifting the probability mass towards names.

Lily	Rachel	England	English
Dean	Caroline	London	English
Xavier	Hunter	America	French
David	Elizabeth	England	Christian
Lily	Emma	Santa	Irish
Noah	David	Paris	German
Al	Julia	France	Roman
Alex	Trevor	Little	Marine
Hunter	Zoe	North	Fae
Eric	Dan	Paul	Dean
Derek	Kim	English	Sunday

Table 6.6: Top 10 predictions for words tagged as names.

Additionally, we also studied how "different" were the average output distributions for names and not names. For this we used the Jensen-Shannon divergence, a metric for measuring the similarity between probability distributions. While experiments from subsection 6.1.2 were producing average divergence values of 0.21, experiments from Table 6.5 reported average diver-

[4]"England" and "English" are very popular english last names

gences of 0.69 (very close to the its upper bound value for our setting, $\ln(2)$). These results seem to confirm that our model is successfully learning two differentiated output distributions.

Finally, we wanted to test the behavior of our model when rather than making a distinction between names and not names (a hard task based on our previous results), we differentiate between nouns and not nouns (we considered as nouns words marked as common or proper nouns by the default `Stanford PoS tagger`[5]). Initial results seem to indicate that while the switch component is able to classify words more confidently, the divergence drops back to 0.21 and overall perplexity is in the same range as name models. A possible explanation for this could be modeling the output distribution of common and proper nouns together might be dominated by common nouns as they are much more frequent. However, further experiments would be required to check this.

6.2 Pointer Based Models

6.2.1 Experimental Setup

The experiments described in this section are evaluated on the LAMBADA task so we follow the training methodology described in [18]. Namely, we train on the full lowercased training set and use a vocabulary size of 93215. Additionally, we incremented the dimension of our word embeddings to 200. For training the pointer sentinel mixture model we used TBPTT with $k_2 = 100$ as in [15].

An important detail when training the PSMM is the initialization of the sentinel vector $\mathbf{s}$. If initialized with too large values, the last element of $\mathbf{z}$, corresponding to the logit of the gate, will be much bigger than the rest and applying the softmax will lead to $g_t = 1$. If this is the case, the loss will be equal to the regular cross-entropy loss (as shown in Equation 6.3) and the pointer component won't be trained.

$$\begin{aligned} \mathcal{L}(\theta) &= -\log(g_t\, p_{\text{vocab}}(w = y(t)|\mathbf{h}(1\dots t)) + \sum_{k \in I(y(t),\, x(t))} \mathbf{a}(t)_k) \\ -\log(g_t + \sum_{i \in I(y(t),\, x(t))} \mathbf{a}(t)_i) &\overset{g_t=1}{=} -\log(p_{\text{vocab}}(w = y(t)|\mathbf{h}(1\dots t)) \end{aligned} \tag{6.3}$$

6.2.2 Experiments

To finish this chapter, we describe the experiments carried out with the PSMM and describe its performance when applied to the LAMBADA task. Table 6.7

[5] `NN,NNS` and `NNP,NNPS`, respectively

summarizes the obtained results for the baseline described in subsection 6.1.2, the neural continuous cache model introduced in section 5.2 and the pointer sentinel mixture model also introduced in section 5.3.

	Control	**LAMBADA**
LSTM-512 [18]	149	5357
Continuous Cache [14]	129	138
PSMM (length=100)	114	154
PSMM (length=200)	116	172

Table 6.7: Perplexity on LAMBADA.

We can observe that the PSMM with a pointer window of 100 words improves upon the continuous cache on the control set and gets worse but comparable results on LAMBADA. It is important to note that the parameters of the continuous cache model (the interpolation weight and the sharpness parameter for the cache distribution) were manually tuned to maximize the performance on each set. On the contrary, the PSMM learnt all its parameters automatically.

This makes the PSMM specially appealing as it does not make any assumptions on the text that is going to be evaluated on. Furthermore, by calculating the mixture gate g_t for each word it allows to inspect the dynamics of the model in a more intuitive way.

For example, Figure 6.3 illustrates a LAMBADA passage where the model is very confident that the next word is contained in the window (low g_t, probably due to the word "miss") and successfully shifts the probability mass to the previous occurrence of the target word.

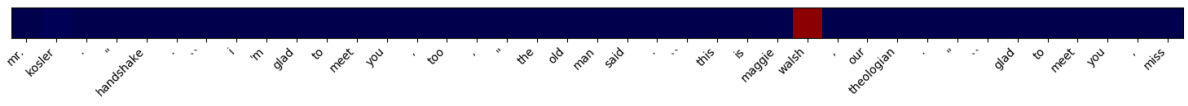


Figure 6.3: Predicting "walsh" ($g_t = 0.07$).

In Figure 6.4 we see that although the target word is a common name, the model successfully identifies the repeated pattern "out of the fort" and assigns the highest probability to the two previous occurrences.

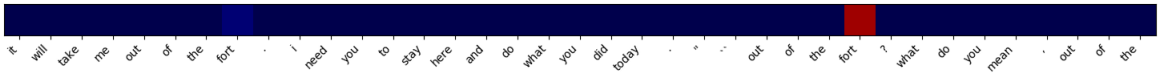


Figure 6.4: Predicting “fort” ($g_t = 0.28$).

Finally, Figure 6.5 shows that the model is also able to identify situations where the pointer is not likely to help.

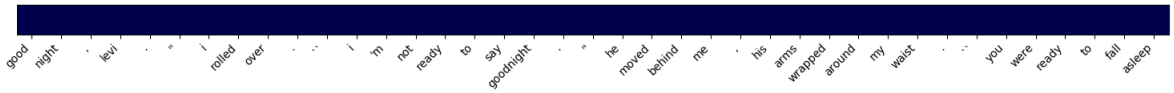


Figure 6.5: Predicting “earlier” ($g_t = 0.99$).

Chapter 7

Conclusion

7.1 Achieved Results

7.2 Future Work

Extend neural language models to handle OOV words ?

Bibliography

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

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

[4] Yoon Kim, Yacine Jernite, David Sontag, and Alexander M Rush. Character-aware neural language models. In *AAAI*, pages 2741–2749, 2016.

[5] Andriy Mnih and Geoffrey E Hinton. A scalable hierarchical distributed language model. In *Advances in neural information processing systems*, pages 1081–1088, 2009.

[6] Andriy Mnih and Yee Whye Teh. A fast and simple algorithm for training neural probabilistic language models. *ICML*, 2012.

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

[8] 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.

[9] David Krueger, Tegan Maharaj, János Kramár, Mohammad Pezeshki, Nicolas Ballas, Nan Rosemary Ke, Anirudh Goyal, Yoshua Bengio, Hugo

Larochelle, Aaron Courville, et al. Zoneout: Regularizing rnns by randomly preserving hidden activations. *arXiv preprint arXiv:1606.01305*, 2016.

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

[11] Julian Georg Zilly, Rupesh Kumar Srivastava, Jan Koutník, and Jürgen Schmidhuber. Recurrent highway networks. *arXiv preprint arXiv:1607.03474*, 2016.

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

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

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

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

[16] Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *ICLR*, 2017.

[17] Gábor Melis, Chris Dyer, and Phil Blunsom. On the state of the art of evaluation in neural language models. *arXiv preprint arXiv:1707.05589*, 2017.

[18] 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.

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

[20] 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.

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

[22] 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.

[23] 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.

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

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

[26] 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.

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

[28] 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.

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

[30] 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.

[31] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

[32] Li Wan, Matthew Zeiler, Sixin Zhang, Yann L Cun, and Rob Fergus. Regularization of neural networks using dropconnect. In *Proceedings of the 30th international conference on machine learning (ICML-13)*, pages 1058–1066, 2013.

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

[34] 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.

[35] 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.

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

[37] 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.

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

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

[40] Google Brain. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from www.tensorflow.org.

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

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

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

ETH

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):

BROAD DISCOURSE CONTEXT FOR LANGUAGE MODELING

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

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

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)**
ZÜRICH, 1-11-2017	Moisés

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