FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Tomáš Musil

Neural Language Models with Morphology for Machine Translation

Institute of Formal and Applied Linguistics

Supervisor of the master thesis:	RNDr. Ondřej Bojar, Ph.D.
Study programme:	Computer Science
Study branch:	Artificial Intelligence

Prague 2017

I would like to thank my thesis supervisor, RNDr. Ondřej Bojar, Ph.D., for his advice and his patience. I also owe my thanks to my parents, who supported me during my studies.

Title: Neural Language Models with Morphology for Machine Translation

Author: Tomáš Musil

Institute: Institute of Formal and Applied Linguistics

Supervisor: RNDr. Ondřej Bojar, Ph.D., Institute of Formal and Applied Linguistics

Abstract: Language models play an important role in many natural language processing tasks. In this thesis, we focus on language models built on artificial neural networks. We examine the possibilities of using morphological annotations in these models. We propose a neural network architecture for a language model that explicitly makes use of morphological annotation of the input sentence: instead of word forms it processes lemmata and morphological tags. Both the baseline and the proposed method are evaluated on their own by perplexity, and also in the context of machine translation by the means of automatic translation quality evaluation. While in isolation the proposed model significantly outperforms the baseline, there is no apparent gain in machine translation.

Keywords: language model, neural network, morphology

Contents

Introduction

Machine translation is a fast developing branch of natural language processing. In recent years, we have witnessed a move towards general machine learning methods (mostly recurrent neural networks), which are linguistically uninformed and can be used to translate arbitrary sequences of symbols, not only natural languages. For these methods, the main limitation seems to be the amount of training data that we are able to obtain and process. Data sparsity greatly affects the translation of inflected languages, where we are unlikely to see all forms of rare words in the training data.

One way to fight the data sparsity in inflected languages is to use lemmas and morphological information instead of word forms. That way, we do not have to encounter all the word forms of the given lemma in the training data.

In this thesis, we compare different ways of representing morphological information in the neural network. As the evaluation measure, we use both perplexity of the model and results of its use in a translation task.

In the first chapter, we describe the theory that our work is based on. We cover the basics of artificial neural networks, language models in general and their combination—*neural language models*. In the following chapter, we review the related works. We describe available implementations of neural language models and results of relevant experiments. In the third chapter we discuss the architecture of our model and its implementation. In the fourth and fifth chapter, we evaluate the results of our experiments.

Chapter 1

Neural Language Models

In this chapter, we explain the theory behind neural language models (NLMs), as first described by Bengio et al. [2003]. Because the theory behind artificial neural networks (ANNs) is quite extensive, this is more of a guide to related literature than a comprehensive description. For an exhaustive description of today's state-of-the-art, see for example [Goodfellow et al., 2016].

We begin this chapter with a brief introduction to neural networks and learning by backpropagation. Then we describe the architecture of NLMs and its use in machine translation.

1.1 Neural Networks

ANN (or just neural network (NN)) is a computational model inspired by the central nervous system. In an ANN, each neuron is a small computational unit, that takes some numbers as inputs and produces an output (also called *activation*). We will denote each input by x_i, x will denote the vector of inputs. Each input has a weight (w_i, w being the vector of weights). The output (y) of a neuron with n inputs is given by

$$y = f(\sum_{i=1}^{n} w_i x_i), \tag{1.1}$$

where f is the *activation function.* We can also write this in the vector form: $y = f(x^T w)$.

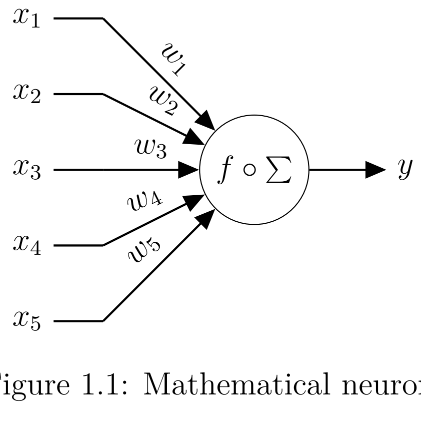

Figure 1.1: Mathematical neuron.

Various activation functions (plotted in Figure 1.2) are being used:

- sigmoid (logistic) function: $f(x) = \frac{1}{1+e^{-x}}$,
- hyperbolic tangent: $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$,
- rectified linear function: $f(x) = max(x, 0)$,
- and other.

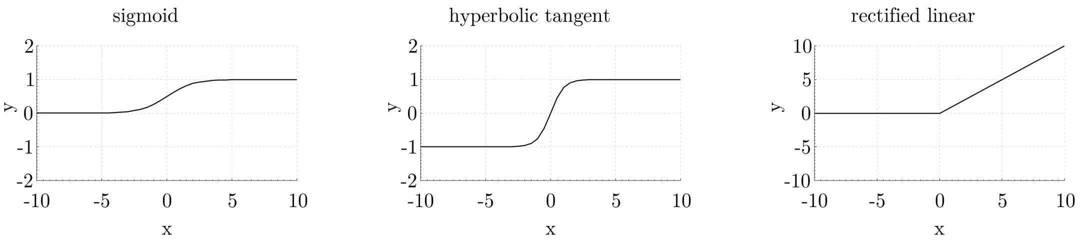

Figure 1.2: Various activation functions.

To form a NN, neurons are connected together—the output of one neuron is taken as the input of another neuron. NNs are usually organized in layers. The first layer is the *input* layer. The activities of the input layer are given to the NN from outside. The last layer is the *output layer*. The activities of the *output layer* are the results of the computation. The layers between the input layer and the output layer are called *hidden layers*. In the simplest case, inputs of each neuron are connected to outputs of all the neurons in the previous layer (except for the input layer, which has no previous layer and gets input from the data).

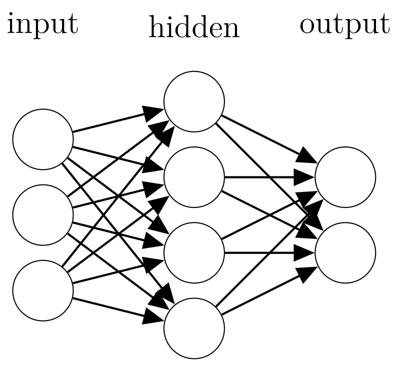

Figure 1.3: Simple feedforward neural network.

The architecture described above may be referred to as feed-forward NN with fully connected layers. There are other, more complicated architectures, notably recurrent neural networks (RNNs), described later in Section 1.1.4.

1.1.1 Backpropagation

Backpropagation is the most commonly used method of training of NNs. The goal is to find the parameters of the model (the weights of inputs for neurons) that minimize the error of the output for the dataset (or rather the expected error for the unseen data). The error function and the activation function of neurons must be differentiable. The main principle of backpropagation consists in propagating derivatives of the error back through the network and adjusting the weights accordingly.

The backpropagation algorithm computes the gradient of the weights of the network, with respect to the error function. If the error function were convex and we computed the gradient over the whole dataset in each step, we could use gradient descent, which (with sufficiently small learning rate) would converge to the optimal weights (with respect to the error function, for the given dataset). However, computing the gradient over the whole dataset is usually too expensive. What we can do instead is to compute the gradient for a single data-point and apply it to the weights before computing the gradient for the next point. This algorithm is called stochastic gradient descent (SGD). It is not guaranteed to converge to the global optimum, it might get stuck in a local one. A compromise between the computationally expensive gradient descent and its stochastic variant is to average the gradient over a batch of training examples. The size of the batch is selected so that it can be computed quickly. The bigger batch size we choose, the slower will the weights converge, but with bigger batch size the gradient descent is less likely to get stuck in a local optimum.

There are also more sophisticated algorithms for gradient-based optimization. We use Adam [Kingma and Ba, 2014], which computes individual adaptive learning rates for different weights from estimates of first and second moments of the gradients. It only requires first-order gradient, it is memory efficient and it can be effectively used without tuning any hyperparameters.

1.1.2 Training

There are some technical terms related to the training process, that we will use later on. We explain them in this section.

During the training we go through the training data repeatedly. Each pass through the data is called an *epoch* (or sometimes an *iteration*).

We periodically measure the performance on a separate dataset, called the *development set* or the *devset* for short (sometimes also called the *validation set*). The main reason to do this is to detect *overfitting*, that is a situation when the model fits the training data well, but fails to generalize to data that it did not see in the training set.

A technique to prevent overfitting is generally called a *regularization* method. Intuitively, most regularization methods make the machine learning task "harder" by introducing additional constraints on the model or by modifying the dataset, to prevent the model from "remembering" the whole dataset and forcing it to generalize. Examples of regularization include constraining the parameters of the model, introducing noise into the data, or stopping the training early, before the model starts overfitting.

If we tune any hyperparameters based on the performance on the development set, we should also report the performance on yet another set of data, that did not influence the learning of the model. This dataset is called a *test set*.

A plot displaying a performance measure with respect to training step (or epoch, or time) is called a *learning curve*.

1.1.3 Softmax Estimation

If we want to use an NN to estimate a probability distribution, we need to normalize the values of the output layer to sum to 1. This is usually achieved by applying the softmax function

$$\sigma(z_j) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \tag{1.2}$$

to the layer. The normalization is computationally expensive, particularly during training.

The usual way to build an n-gram NLM is to build an NN that takes $n-1$ words as input and estimates the probability distribution for the next word. As the output layer needs one neuron for each possible word, the training is getting prohibitively expensive for a large vocabulary. This so-called *softmax problem* is the main problem in building a NLM.

One solution consist in dividing the words into classes and work with the softmax in a hierarchical fashion [Le et al., 2011]. Chen et al. [2015] suggest that this is only efficient for very large vocabularies.

Other solution is to estimate the gradient of the softmax layer from a random sample of its neurons during the training and only compute the whole softmax for evaluation and inference. There are various ways to do that, we present *noise contrastive estimation* and *sampled softmax* below.

Noise Contrastive Estimation

Noise contrastive estimation (NCE), introduced by Gutmann and Hyvärinen [2010], is a method for efficiently estimating the error of the softmax layer, without having to compute the sum over the whole layer. The basic intuition behind this method is to transform the classification problem into a small set of decision problems. We have a true candidate (the class that we encountered in the data) and we sample the classes for a set of false candidates. The task then becomes to distinguish whether each candidate is the true one or a sampled one. The error from this task is then backpropagated to the softmax layer, where the weights and approximate normalization constants are learned. Mikolov et al. [2013b] proposed a simplified variation of NCE called *negative sampling.*

Mnih and Teh [2012] first proposed to use NCE to train NLMs and Mnih and Kavukcuoglu [2013] used it to efficiently learn word embeddings. Baltescu and Blunsom [2014] examine the combination of NCE and class-factored output layer.

Sampled Softmax

Jean et al. [2014] proposed another approach, based on a work by Bengio and Senécal [2008]. It consists in using importance sampling to reduce the size of the softmax computed during the training. To ensure that all the words in the dictionary get their representations updated, the training corpus is partitioned and different subset of the vocabulary is used as a sample for each part of the corpus.

1.1.4 Recurrent Neural Networks

A RNN is type of NN where connections form a directed circle. This means that outputs of some neurons are connected—possibly through other neurons—to their own inputs. The circular part of the network can work as a memory, reading data and feeding them back to itself in the next time step. This allows RNNs to process input sequences of arbitrary length and makes them suitable to work with temporal data, for example in speech recognition. RNNs are also used in neural machine translation.

RNNs operate in discreet time steps. For the learning, the so-called *backpropagation through time* is used. The activation values are computed for several time steps in advance by unfolding the network—computing each time step in a new copy of the network. The gradient is then computed and propagated back through the time steps. This can lead to very long paths between the neurons where the error is computed and the neurons that are being updated by the backpropagation algorithm. Long backpropagation paths often suffer from the vanishing or exploding gradient problem. As the gradient gets multiplied by many weights, it tends either to disappear (if the weights are small) or to grow to infinity. Either way, the network is not learning.

The first RNN architecture to overcome these problems was the long short-term memory (LSTM) method [Hochreiter and Schmidhuber, 1997]. It does not use any activation function in the recurrent component, so the gradient is not repeatedly squashed and does not vanish. LSTM networks are usually implemented in blocks, where each block contains several neural layers, one that works as a memory cell, and two or three that serve as gates that control the flow of information through the block: the input gate controls what information from the input gets to the cell, the forget gate controls what information is kept to the next time step and the output gate controls what information the block outputs. The input and forget gates are in some implementations merged into one.

1.2 Neural Language Model

A statistical language model (LM) assigns a probability to a sequence of m words (or other tokens) $P(w_1, \ldots, w_m)$. Most sequences of words are very rare—for example this sentence probably only occurs in this thesis (and with each next word the probability that the exact same sentence occurred elsewhere decreases rapidly). Therefore the task has to be split into reasonable subtasks. We can use the chain rule

$$P\left(\bigcap_{k=1}^{n} w_k\right) = \prod_{k=1}^{n} P\left(w_k \,\middle|\, \bigcap_{j=1}^{k-1} w_j\right) \tag{1.3}$$

to decompose the probability of a sequence of words into a product of probabilities that a word occurred given that its predecessors did. For example, a decomposition of four words sequence would look like this:

$$P(w_4, w_3, w_2, w_1) = P(w_4 \mid w_3, w_2, w_1) \cdot P(w_3 \mid w_2, w_1) \cdot P(w_2 \mid w_1) \cdot P(w_1) \tag{1.4}$$

To further simplify the matter, we can assume that every word depends only on the last $n-1$ words. With this assumption, we can use an n-gram model to estimate the sequence probability. The statistical n-gram model estimates the probability of an n-gram as its relative frequency in the training corpus. Various smoothing techniques are used to assign non-zero probability to unseen n-grams and address the imbalance between frequent and infrequent n-grams [Chen and Goodman, 1996].

1.2.1 Neural Language Model Architecture

In a *neural language model*, the probability distribution is realized by an artificial neural network. The first effective NN architecture for language modelling was introduced by Bengio et al. [2003]. A comprehensive (although already somewhat dated) introduction to NLMs is given by Mikolov [2012]. In this thesis, we focus on n-gram (feedforward) NLMs. Although NLMs based on recurrent neural networks can process sequences of arbitrary length, their training is more complicated, both training and inference are computationally expensive, and RNN language models (as such—not counting end-to-end neural machine translation) are relatively rare in machine translation [Alkhouli et al., 2015].

If we want to use a feedforward NN as a language model, we need to address a few complications. The NN accepts a numeric input of fixed length. A sentence in a natural language is a sequence of tokens (for example words) of arbitrary length.

The problem of the arbitrary length of input can be solved by the n-gram approach. This method uses a *sliding window*, as illustrated in Figure 1.4. We construct a NN to take a sequence of words with one of them missing (typically the last one for LMs intended for use in machine translation, or the middle one if the goal is to extract word representations from the model) and train it to output a probability distribution over all possible words for the empty position. When we use the language model, we move this *window* over the whole sentence and collect the probability for each word given its context. The probability of the sentence is the product of probabilities of all the words in the sentence.

The problem of numerical representation of words is solved like this: the vocabulary is enumerated, each word is given an unique number in range from one to the size of the vocabulary. Then we construct a so-called *one-hot vector* for each word: it is a vector from $\mathbf{R}^v$, where v is the size of the vocabulary, and it is filled with zeroes, except for the position indicated by the position of the word in the dictionary, where there is a one. If by $w_{i,j}$ we denote the j-th position in the vector representing the i-th word, then

$$w_{i,j} = \begin{cases} 1, & \text{if } j = i, \\ 0, & \text{otherwise.} \end{cases} \tag{1.5}$$

Getting and processing the whole vocabulary of a natural language is of course impossible. The NLM will therefore encounter words that are unknown to it. This is usually solved by designating a special word as an *unknown word token* and replacing the words that are not in the model's vocabulary with it. There must also be some unknown words left in the training data, otherwise the model will not learn when to expect an unknown word. Unlike a translation model, a LM only evaluates the sequence, it does not generate sequences, therefore the unknown words are not a serious problem.

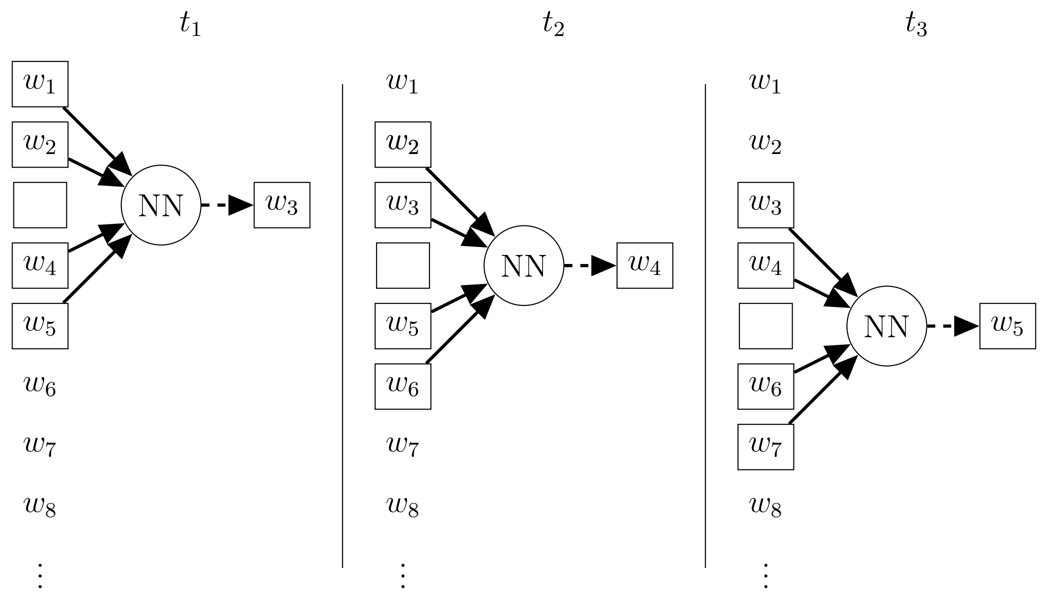

Figure 1.4: The learning of the n-gram neural language model, with the middle word of the n-gram being the predicted one.

1.2.2 Word Embeddings

An interesting feature of NLMs (an other neural methods of text processing) are the so-called *word embeddings*, vectors that represent words in the NLM. As we explained above, each word is represented by a *one-hot vector*. The multiplication of a *one-hot vector* by the matrix of weights of the first layer of the neural network is equal to selecting a single column from the matrix. Therefore, each word from the vocabulary is associated with a vector of real numbers—the column of the weight matrix.

Word embeddings capture both semantic and syntactic information. In Figure 1.5, you can see a t-SNE [Maaten and Hinton, 2008] two dimensional projection of 300 dimensional embeddings for the one thousand most frequent words from a language model trained on the Czech part of the CzEng corpus (see Section 4.1). In the global picture, we see various part-of-speech forming clusters (nouns on the left, verbs in the middle and the top and so on). In the marked areas, we see that semantically similar words tend to have similar embeddings.

Mikolov et al. [2013c] describe even more interesting properties of word embeddings. They demonstrate various relationships between the embeddings. Each relationship (syntactic, for example singular—plural and semantic, for example man—woman, as illustrated in Figure 1.6) is represented by a vector in the embedding space, that is the difference between the two words for which the relationship obtains. For example, in case of singular—plural, $v_{cars} - v_{car}$, $v_{apples} - v_{apple}$ and $v_{kings} - v_{king}$ should all be the same (or at least a fairly similar) vector. Another example would be the equation

$$v_{king} - v_{man} + v_{woman} \approx v_{queen}, \tag{1.6}$$

which means that if we take the embedding for the word *king*, subtract the embedding for *man* and add the embedding for *woman*, the embedding nearest to the resulting vector would represent the word *queen*. In other words: the left side

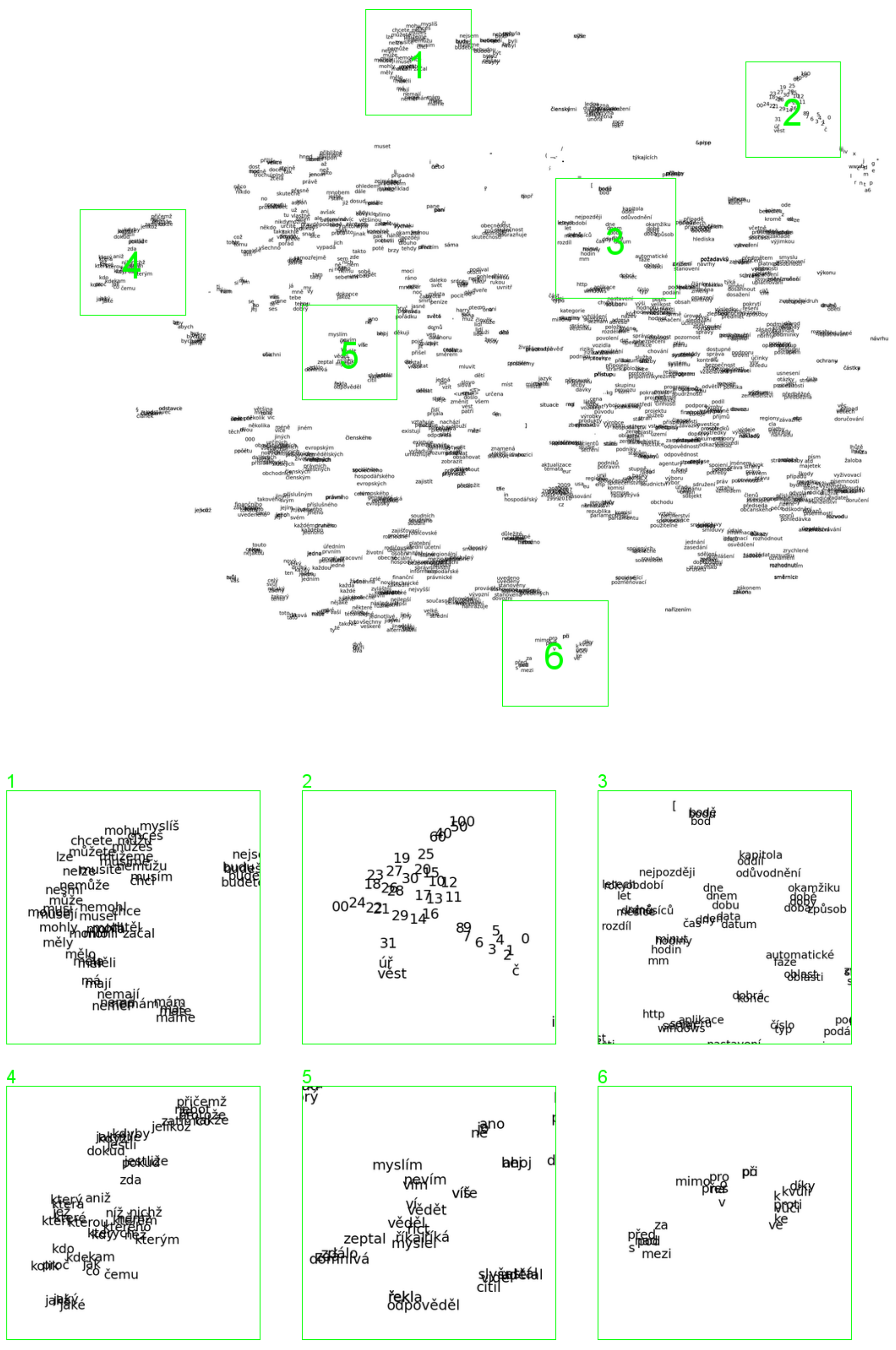

Figure 1.5: One thousand most common words from a language model trained on the Czech part of CzEng corpus. Projected from $\mathbf{R}^{300}$ to $\mathbf{R}^2$ by t-SNE.

of the equation poses the question “the word *man* is to the word *king* as the word *woman* is to which word?” and if we do the vector arithmetic, we find that the answer is the word *queen*.

There are also attempts to model subword units this way, see for example Mikolov et al. [2012], or Kocmi and Bojar [2016].

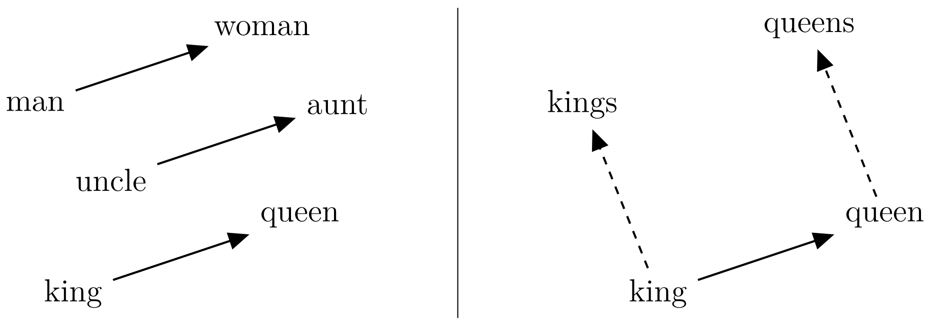

Figure 1.6: Relations between embeddings according to Mikolov et al. [2013c].

1.2.3 Evaluation

In this section, we describe metrics that are used to evaluate language models. A better result in these metrics does not guarantee that the model will work better in a real machine translation scenario. We review the metrics for machine translation quality in Section 1.3.3.

Perplexity *Perplexity* is a measure of how well the model can predict the word given its context. Intuitively, it can be interpreted as a measure of “confusion” of the model—high perplexity means that the model is “confused” by the data, low perplexity means that the data correspond with the predictions of the model. We use perplexity if we want to evaluate the model on its own. It is easy to compute during training, because we use cross-entropy as the error function for the backpropagation. Perplexity measures the uncertainty of the model. The lower the perplexity, the better the model is assumed to be. For a probability model, the cross-entropy is defined as

$$H(\tilde{p}, q) = -\sum_x \tilde{p}(x) \log_b q(x), \tag{1.7}$$

where $\tilde{p}$ denotes the empirical distribution, q denotes the distribution given by the model and b is customarily 2 or e. Perplexity is then defined as

$$P(\tilde{p}, q) = b^{H(\tilde{p},q)} = b^{-\sum_x \tilde{p}(x) \log_b q(x)}. \tag{1.8}$$

In the case of language models, the reported perplexity is usually the perplexity per word.

Comparing perplexity is only meaningful for a single task. For a difficult task, the perplexity can reach values over 1000, while on easy tasks, it is common to observe values below 100 [Mikolov et al., 2011].

Accuracy/Word error rate *Accuracy* measures how often would the model select the correct word—assign the largest probability of all possibilities to the one that actually occurred in the data. The *word error rate* is the complement to the accuracy.

1.3 Machine Translation

In this section, we give an overview of machine translation, with focus on the use of NLMs in classical statistical machine translation (SMT) and their relation to neural machine translation (NMT) systems.

1.3.1 Language Models in Statistical Machine Translation

SMT is a wide field with a variety of approaches and a rich history. In this thesis, we will only cover the absolute basics, to understand the role of language models in SMT. For a broader introduction to SMT theory, see Koehn [2009] or Bojar [2012] (in Czech).

The idea behind statistical machine translation comes from information theory. We assume that there is a probability distribution $p(e \mid f)$ that a string e in the target language (for example English) is the translation of the string f in the source language (for example French). The aim of the translator, who is given the string f, is to find the string e that maximizes $p(e \mid f)$. By applying the Bayes' theorem to the distribution, we get

$$p(e \mid f) = \frac{p(f \mid e) \cdot p(e)}{p(f)}. \tag{1.9}$$

Because we want to maximize the probability, we can disregard the denominator and write

$$\tilde{e} = \operatorname*{argmax}_{e \in e^*} p(e \mid f) = \operatorname*{argmax}_{e \in e^*} p(f \mid e) p(e), \tag{1.10}$$

where $\tilde{e}$ is the best translation of f and e^* is the set of all possible strings in the target language. The component $p(e)$ in Equation 1.10 is the language model—the probability of a string in the target language.

The other component is called the translation model. We can keep them separated. The translation model produces a list of n best translation hypotheses and their respective scores. We then use the language model to score the hypotheses, combine the scores from the translation model and the language model and select the best hypothesis according to the final score. This procedure is referred to as *reranking (or rescoring) the n-best list.* Very large or complex language models are often used in the reranking stage [Olteanu et al., 2006].

Another possibility is to use the LM directly in the process of hypothesis generation. The so-called *decoder* is the part of the SMT system that generates the hypotheses. It scores partial hypotheses to guide the search. The most straightforward method is to generate the hypothesis from left to right, using the beam search algorithm to reduce the combinatorial explosion of possible hypotheses.

Factored Language Model

The *factored language model* [Bilmes and Kirchhoff, 2003] was introduced and implemented in SRI language modeling toolkit [Stolcke et al., 2002]. In a factored LM, each word is viewed as a vector of factors. Each factor contains different information about the word, for example morphological information, part-of-speech, or the relationship to other words in the sentence.

1.3.2 Language Models and Neural Machine Translation

The relation between language models and NMT is not so clear. NMT systems include end-to-end translation in one neural network, so there is no clearly separated LM component. In this subsection, we briefly explain how NMT works and then discuss its relation to NLMs.

Neural Machine Translation

The majority of approaches to NMT are based on the sequence-to-sequence NN architecture (see Figure 1.7) first proposed by Sutskever et al. [2014]. There is a RNN, called the *encoder*, that reads the input sentence and encodes it into a state vector. The state vector is then used by another RNN, called the *decoder*, that outputs the translation. Britz et al. [2017] present a large-scale analysis of architecture variations for NMT.

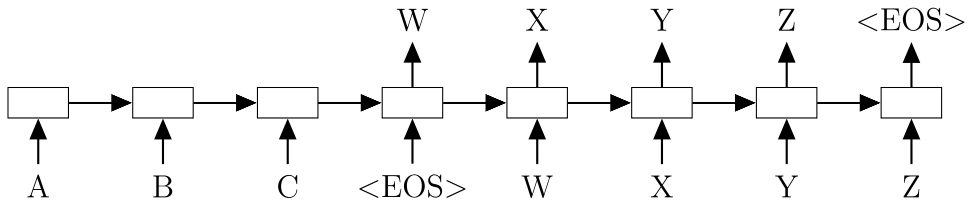

Figure 1.7: Sequence-to-sequence architecture according to Sutskever et al. [2014]. Each rectangle represents one step of a LSTM network computation. The network reads the sequence “ABC” and outputs the sequence “WXYZ”.

Neural Machine Translation and Language Models

There is no separate LM in a NMT system. However, both the encoder and the decoder can be considered a kind of LMs.

The research of NLMs and words representations in particular is relevant to NMT as well, because the methods of representing text are the same for both NLMs and NMT systems.

1.3.3 Evaluation

Machine translation evaluation is itself an area of intensive research.

One way to evaluate translation results is to have people who understands both languages to rate the translation. There are various methodologies to do this.

The other possible approach is automatic translation evaluation, where quality of the translation is measured as similarity to one or more reference translations. A sentence can usually be translated in hundreds thousands of equally valid ways [Bojar et al., 2013a] and for a meaningful evaluation the reference translations should cover as much variants as possible. This makes even automatic machine translation evaluation a costly task, because in ideal setting, the test data should be translated by several human translators. For latest results in automatic evaluation, see Bojar et al. [2016b].

BLEU BLEU (Bilingual Evaluation Understudy, Papineni et al. [2002]) is the most widespread metric for machine translation evaluation. It is based on the *precision* measure. To compute unigram precision, one counts the number of words in the candidate translation which occur in any reference translation and divides it by the length of the candidate translation. The problem with precision is that it gives high score to sentences such as “the the the the the”, which has precision of 5/5 if any reference contains at least one “the”. BLEU uses *modified unigram precision*, that clips the count of each candidate word to the maximal number of occurrences that a word has in any reference translation. In our example, if the reference translation with maximum occurrences of “the” had 2 of them, the modified unigram precision for the candidate sentence would be 2/5. This captures the intuition that a word should not contribute to precision, unless there is a matching word that it can be assigned to in a reference translation. Modified n-gram precision is computed similarly for any n. BLEU is a geometric mean of modified n-gram precisions (usually up to 4-gram, sometimes denoted BLEU4) multiplied by a *brevity penalty* that serves to punish sentences that are too short (without brevity penalty, the sentence “the” would seem perfect translation with BLEU of 1 if any reference contained a definite article).

As defined above, BLEU score is a value between 0 and 1, but in the following chapters, we follow the common convention of reporting BLEU scores multiplied by 100.

METEOR Lavie [2014] describe Meteor, a language specific translation evaluation metric that evaluates hypotheses by aligning them to reference translations and calculating sentence-level similarity scores. It uses four different matchers for the alignment: exact matching, stem matching, synonym matching, and paraphrase matching. Meteor shows higher correlation with human ranking than BLEU.

TER Translation Edit Rate (TER), proposed by Snover et al. [2006], is a translation evaluation measure based on the number of edits needed to transform the hypothesis to match one of the references. Possible edits include the insertion, deletion, and substitution of single words as well as shifts of word sequences. This can be intuitively understood as the amount of work needed to correct the translation.

1.4 Neural Language Models and the Language Itself

The author of this thesis believes that one of the main goals of the research of machine translation (and other natural language processing) should be to get a better understanding of the way language itself works. Our reflections on the relationship between NLMs research and our understanding of the essence of language exceed the scope of this thesis. However, we will present a brief summary and hope to publish a more detailed account elsewhere at a later date.

There is not much literature concerning the relation between NLMs and philosophy of language. The only article we found [Honkela, 2007] is ten years old and mostly concerned with self-organizing maps and Quine's version of semantic holism.

We would like to argue that recent developments in the NLM and word embedding research bring strong arguments for Frege's thoughts on holism. Their subsequent reformulation by Tugendhat [1970] almost reads as instructions to construct a Skip-gram model, an NN architecture proposed by Mikolov et al. [2013a] to compute word embeddings that work well in various semantic tasks.

The relation between NLMs and the theory of meaning of the word as its use in language, as first proposed by Wittgenstein [1953] in his later phrase, should also be considered, as well as connections between word embeddings, methods of computing them and various structuralist theories of language.

Chapter 2

Related Works

In this chapter, we describe various implementations of NLM and work related to factored NLMs.

2.1 Neural Language Models

There are many variants of NLMs. Here we concentrate mainly on those that have an open-source implementation and work with the Moses decoder. We focus on n-gram NLMs, because they are directly comparable with our architecture.

The works below present improvements in machine translation with NLMs. Baltescu and Blunsom [2014], on the other hand, present a pessimistic view, saying that NLMs are only better than traditional n-gram back-off models in memory constrained environments.

2.1.1 Continuous Space Language Model

The first freely available[1] implementation of neural LM was the Continuous Space Language Model toolkit (CSLM) [Schwenk, 2007, 2010, Schwenk et al., 2012, Schwenk, 2013].

The softmax problem is solved by reducing the vocabulary size and falling back on SRILM n-gram model. The neural network is trained on a so-called *short list* of most common words. The current version (v4, June 2015) has support for rescoring HTK lattices, continuous space translation model, and runs on a GPU.

CSLM is used to rescore n-best lists, there is no support for it in the Moses decoder.

2.1.2 Neural Probabilistic Language Model

Another available[2] implementation is the Neural Probabilistic Language Model Toolkit (NPLM) [Vaswani et al., 2013].

The softmax problem is solved by using the NCE method. To achieve speed sufficient for the use of NPLM in a SMT decoder, the softmax layer is not normalized during decoding. Ironically, even though it has "probabilistic" in its name,

[1]`http://www-lium.univ-lemans.fr/cslm/`

[2]`http://nlg.isi.edu/software/nplm/`

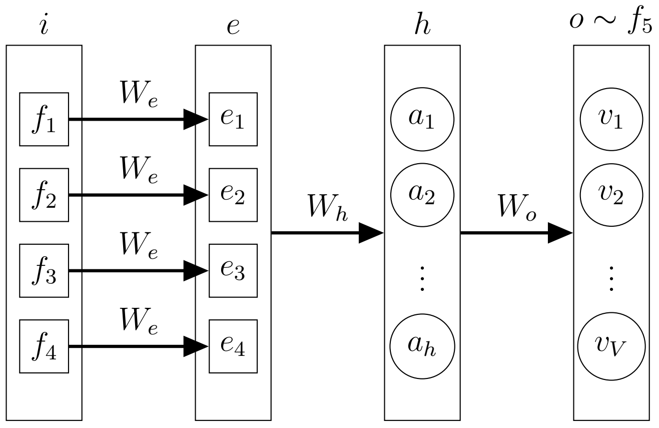

Figure 2.1: Schematic representation of the NPLM NN architecture. There are four layers: the input layer i, the embedding layer e, the hidden layer h, and the output layer o. Each input word is represented by a *one-hot vector* f_k. The one-hot vector is multiplied by the embedding matrix W_e to get the corresponding word embedding. These are then concatenated and multiplied by the weight matrix W_h to get the input activations for the hidden layer. After the activation function is applied, the output of the hidden layer is multiplied by the weight matrix W_o to get the input activations of the output layer. The softmax function is computed over the layer and the result is a probability distribution over the vocabulary for the output word.

it does not produce a probability distribution. It improved translation by up to 0.6 BLEU on French to English and German to English language pairs.

In Figure 2.1, we see the NN architecture used in NPLM. The context words in the input layer i are represented in *one-hot representation*. Each of them is then projected (by a shared weight matrix W_e) to the embedding layer e. The rest is a conventional feed-forward NN, with one hidden layer and softmax output layer.

2.1.3 OxLM

Another open-source[3] implementation of a neural LM is OxLM [Baltescu et al., 2014]. It can be used in the Moses decoder. It can be trained in various configurations, with class-factorization and direct features. The softmax problem is solved by NCE, the architecture of the basic configuration is similar to NPLM.

2.1.4 Neural Language Models with Subword Units

Botha and Blunsom [2014] computed embeddings for morphological segments. They then represented words as sums of embeddings for their morphemes. With this approach, they were able to achieve a gain of 1 BLEU point in machine translation from English to Czech compared to translation with back-off n-gram models.

Kim et al. [2016] describe a simple neural language model that relies only on character-level input. The model combines a recurrent neural network, outputs

[3] `http://github.com/pauldb89/oxlm`

from a convolutional neural network and a so-called *highway network*. On languages with rich morphology, they outperform word-level/morpheme-level LSTM baselines in perplexity measure, despite having fewer parameters than them.

A similar architecture is used by Passban et al. [2017], who extend it to handle small subword units as well as characters. They report lower perplexity than Kim et al. [2016] and improved English to Farsi and English to German translation.

2.1.5 Other Neural Language Models

Devlin et al. [2014] used a bilingual NLM to improve translation from Arabic to English. Their model is fundamentally an n-gram NLM, with additional source context. They integrated it into a SMT decoder and produced a gain of 3 BLEU points.

2.2 Factored Neural Language Models and Translation

The oldest paper on factored neural LMs we found is by Alexandrescu and Kirchhoff [2006]. They trained and tested their model on Arabic and Turkish corpora and only compared perplexities.

Niehues et al. [2016] used factored NLM for rescoring of n-best lists. They used a RNN based model with the word surface form, part of speech (POS) tag and word clusters as factors. They used embedding for each factor and then concatenated them. They report gains between 0.3 and 0.7 BLEU points on English–Romanian, English–German, and German–English WMT16 data.

Our model differs in using lemmata instead of surface forms. We hope that this will help us with data sparsity. We also represent POS tags in one-hot encoding instead of embedding them.

Sennrich and Haddow [2016] used linguistic input features in neural MT. They use lemmata, morphological features, POS tags and dependency labels. They train an embedding matrix for each feature. They report gains between 0.6 and 1.5 BLEU points on English–Romanian, English–German, and German–English WMT16 data.

Our model differs in representing features other than lemmata in one-hot encoding.

Chapter 3

Neural Language Models with Morphology

In this chapter, we discuss the motivation for our experiments, the architecture of our models and their implementation.

3.1 Motivation

There are tools capable of automatically tagging Czech text with morphological tags that achieve more than 95 % accuracy. We also have a parallel Czech-English corpus, where the Czech side is annotated with morphological information. We wanted to see if we can improve translation by including this information in the language model. With morphological information included, it makes sense to use lemmata instead of word forms in the LM. Traditional statistic LMs were sometimes trained on lemmata, because n-grams of lemmata are less sparse than n-grams of forms. Take for example these two sentences:

Na **okně seděla kočka**.

(A cat sat at the window.)

Viděl jsem u **okna sedět kočku**.

(I saw a cat sitting at the window.)

A NLM trained on word forms has no way of knowing that the marked words are the same in both sentences. A NLM trained on lemmata and morphological tags is not only explicitly told that these are the same words, but it is also given a description of the syntactic situation that causes the word forms to differ.

Neural LM architecture allows to easily combine embeddings of lemmata with encoded morphological information. We hope that this approach will benefit from denser lemmata n-grams without losing the morphological information that is striped away if we just replaced word forms with lemmata.

To test this hypothesis, we designed a neural LM architecture that uses lemmata and morphological tags instead of words forms. We decided to use a neural n-gram model (as opposed to a recurrent NN model) because existing tools and LMs for machine translation follow the n-gram approach.

```
shaw|Shaw_;S|NNMS1-----A----
se|se_^(zvr._zájmeno/částice)|P7-X4----------
v|v-1|RR--6----------
duchu|duch|NNMS6-----A---1
na|na-1|RR--4----------
sebe|se_^(zvr._zájmeno/částice)|P6-X4----------
zlobil|zlobit_:T|VpYS---XR-AA---
,|,|Z:-------------
že|že-1|J,-------------
nezačal|začít-1|VpYS---XR-NA---
právě|právě-1|Db-------------
tady|tady|Db-------------
.|.|Z:-------------
```

Figure 3.1: An example of annotated Czech sentence ("Shaw cursed himself under his breath for not starting here.").

3.2 Morphology

The morphological annotation that we use was obtained by the Morče tagger [Hajič et al., 2007]. The annotation format is the same as in the Prague Dependency Treebank (PDT) [Hajič et al., 2006]. There is a manual for morphological annotation by Zeman et al. [2005], also available online.[1] For detailed overview of computational morphology of Czech, see Hajič [2004].

The morphology is annotated in form of positional tags. The lemma and the tag together should uniquely identify the word form. Each positional tag is a string of 15 characters. Every position encodes one morphological category (see Table 3.1) with one character (mostly upper case letters or numbers). An example of annotated sentence is displayed in Figure 3.1.

The information about each word is organized in factors, divided by the vertical bar ("`|`"). The first factor contains the lowercased word form, the second factor contains the lemma, and the third factor contains the morphological tag.

To encode the tags for the neural network, we converted them to binary vectors. For each possible value of all morphological categories (in the order in which they appear in the documentation), we reserve one position in the binary vector. The position is filled with 1 if the tag has the corresponding value and with 0 otherwise. The resulting vector has 139 positions, where up to 13 values are 1, the rest is 0. In other words, it is concatenation of a series of one-hot vectors, one for each morphological category in the tag (or a zero vector of the corresponding dimension if the category is not used in that tag).

We also tried shorter representation, where we decided to ignore some of the categories (notably category "SubPOS", which has almost as many possible values as the rest of the tag combined) and ended up with 60-dimensional binary tags. The reasoning behind this experiment was that if there are many possible values, each individual value will be relatively rare in the training data, therefore hard to learn to use properly. Also, general usefulness of very specific linguistic

[1] `http://ufal.mff.cuni.cz/pdt2.0/doc/manuals/en/m-layer/html/index.html`

position	**name**	**description**
1	POS	Part of speech
2	SubPOS	Detailed part of speech
3	Gender	Gender
4	Number	Number
5	Case	Case
6	PossGender	Possessor's gender
7	PossNumber	Possessor's number
8	Person	Person
9	Tense	Tense
10	Grade	Degree of comparison
11	Negation	Negation
12	Voice	Voice
13	Reserve1	Reserve
14	Reserve2	Reserve
15	Var	Variant, style

Table 3.1: Positions in Czech morphological tags.

classification seemed questionable. However, in our experiments, the full 139 position tags worked better.

3.3 Architecture

In this section, we describe the architecture of neural networks for our baseline and proposed model.

Both models are feed-forward n-gram NLMs, trained to predict the last word of an n-gram.

3.3.1 Baseline Model

The baseline architecture is modeled after NPLM. The architecture is the same as in Figure 2.1. The main difference is that we use two hidden layers.

3.3.2 Proposed Model

The architecture of our experimental model (Figure 3.2) is similar. The model predicts word forms (so it is directly comparable to the baseline) but the embeddings are trained on lemmata. We add the information from binary-encoded morphological tags to the embedding layer.

3.4 Implementation

In this section we describe the implementation of our models.

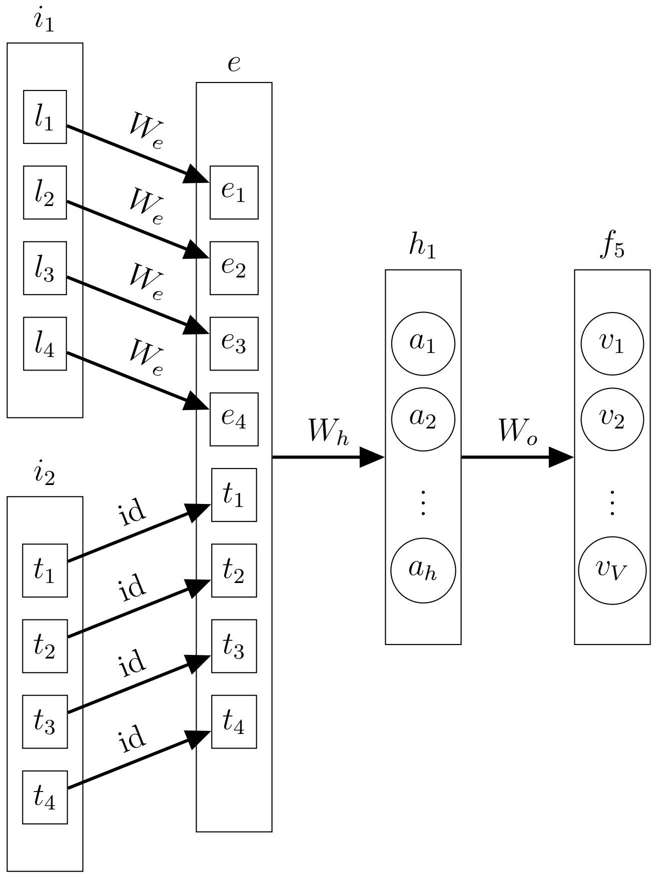

Figure 3.2: Schematic representation of the proposed LM architecture. Compare with Figure 2.1. The input layer is split into two parts. Each l_k denotes one-hot vector representation of the k-th lemma, e_k is the embedding of the lemma and t_k is the corresponding morphological tag, encoded in a binary vector. For simplicity, the second hidden layer is omitted in this schema.

3.4.1 Tools Employed

In this subsection, we describe the tools that we used to implement our models and run our experiments.

TensorFlow

Our NLMs are implemented in TensorFlow[2] [Abadi et al., 2015]. TensorFlow is an open source software library for numerical computation using data flow graphs. It has support for a wide variety of neural network components. Computations can be executed on CPUs or GPUs.

We decided to use TensorFlow because it offers all the needed NN components, including various softmax layer estimation methods. Because the same model can be easily run either on a CPU or a GPU, we can run many smaller experiments on a CPU cluster to broadly search the parameter space, as well as longer experiments on bigger data on a GPU.

Moses

Moses [Koehn et al., 2007] is a state-of-the art statistical machine translation system. It offers two types of translation models: phrase-based and tree-based. Moses features factored translation models [Koehn and Hoang, 2007], which enable the integration linguistic and other information at the word level. It supports several different language model implementations.

We used Moses to test our models in an SMT task.

Eman, ÚFAL SMT Playground

Eman [Bojar and Tamchyna, 2013] is an experiment manager developed at the Institute of Formal and Applied Linguistics, Charles University (ÚFAL MFF UK). It is written in Perl and it comes with ÚFAL SMT Playground,[3] a set of tools for experimenting in SMT.

We used Eman to manage our corpora and run baseline experiments with Moses.

In the Eman workflow, each experiment is composed of steps. A step is a self-contained part of an experiment. It may depend on other steps. It may be used and reused in various experiments. An experiment is a directed acyclic graph of depending steps. A step seed is a recipe to build individual steps.

In the course of writing this thesis, we have contributed the following seeds to the ÚFAL SMT Playground:

nplmbin downloads and compiles the NPLM toolkit,

nplm trains a NPLM language model,

oxlmbin downloads and compiles the OxLM toolkit,

oxlm trains an OxLM language model.

[2]`https://www.tensorflow.org/`

[3]`https://redmine.ms.mff.cuni.cz/ufal-smt/`

3.4.2 morphLM

To carry out our experiments, we wrote a tool called morphLM. The source code is available online[4] under the MIT license. It is written in Python using TensorFlow. It contains code to

- read the data in factored format,
- convert the data to use them in the NN,
- create a computation graph for training the NN,
- run the training,
- compute evaluation metrics,
- and save the model.

The usage of morphLM is described in Attachment A.1.

[4]`https://github.com/tomasmcz/morphlm`

Chapter 4

Experiments with Language Models

This chapter contains experiments with language models evaluated by perplexity. We first introduce the datasets on which we run our experiments. Then we describe experiments with other people's implementations of NLM. Finally, we describe experiments with our implementation of the baseline model and the proposed architecture.

4.1 Datasets

All our datasets are taken from the CzEng 1.0 corpus [Bojar et al., 2012b]. There is also a newer version, CzEng 1.6 [Bojar et al., 2016a]. Table 4.1 lists the sizes and sources (from which part of CzEng they were taken) of our datasets.

The data in CzEng are tokenized with TrTok [Maršík and Bojar, 2012]. For simplicity, we only work with lowercased word forms.

For the LM, each sentence is padded with *sentence-start* tokens ("<s>"). For n-gram model, we add $n-1$ start tokens, so the model starts its prediction on the first word of the sentence. Each sentence is ended by one *sentence-end* token ("</s>"). Words that are not in the vocabulary are replaced with *unknown-word* tokens ("<unk>").

corpus	**part**	**sentences**	**tokens**
c-news	train	197 k	4 218 k
	heldout	–	500
	dev	2 k	47 k
	test	2 k	41 k
c-fiction	train	4 248 k	56 432 k
	dev	43 k	571 k
	test	43 k	573 k

Table 4.1: Datasets and their sizes. The heldout set is a subset of the training set.

NLMs are trained with limited vocabulary. The training data contain only finite amount of different words. For the NLM to learn to estimate the probability of an unknown word, it needs to encounter unknown words even in the training data, so the vocabulary has to be made even smaller. The tokens that are not present in the vocabulary are called the out of vocabulary (OOV) tokens. In comparing language models, they need to have the same vocabulary (imagine a NLM with vocabulary size 0: all the words are unknown, that is exactly what the model predicts, the perplexity is 0). Unless noted otherwise, following experiments are done with a vocabulary of 50 000 words. For the c-news dataset, approximately 3.5 % of the tokens are OOV with this vocabulary.

4.2 Baseline Experiments

In this section, we describe our experiments with existing NLM implementations.

4.2.1 NPLM

In Table 4.2, we see various configurations of NPLM run on the c-news dataset. The reported perplexity is the smallest perplexity achieved on the development set during training for 10 epochs. The parameters were selected to achieve a reasonable time of training and inference, so we did not test long n-grams with high-dimensional embeddings.

The best configuration we found was a 7-gram model with embeddings of size 200, hidden layer of size 200, and output embedding (the second hidden layer) of size 100.

In experiments with NPLM, we used a smaller development set. NPLM separates the development data from the training set during the data preparation step. We will refer to this dataset as the *heldout* set. The first experiments with our models were run on the heldout set to be comparable with this experiment.

4.2.2 OxLM

We also trained a few different configurations of OxLM. For unknown reasons, it did not report perplexity, so we only state the value of the training objective. The results are listed in Table 4.3, n-gram order is 5, the size of the vocabulary is 50 000 words.

4.3 Neural Language Model on Word Forms

For a baseline, we trained a model with the architecture similar to NPLM (see Figure 2.1) trained on word forms. We will refer to this model as *formLM*.

We will refer to our proposed model (described in Chapter 3) as *morphLM*.

In this section, we discuss the results of experiments with *formLM*, our own baseline model, trained on word forms.

Unless specified otherwise, all presented time measurements were performed on Intel$^{®}$ Core^{TM} i5-4210Y CPU and the time is reported for 200 batches of 1000 examples each.

order	**hidden**	**input emb.**	**output emb.**	**noise**	**perplexity**
7	200	200	200	100	195.32
7	300	300	300	50	210.26
7	200	200	200	50	212.21
7	100	200	200	50	220.29
7	200	200	200	25	224.42
7	100	200	200	25	227.28
7	100	100	100	100	227.55
11	50	50	50	25	228.19
7	100	100	100	50	228.22
7	50	300	300	50	229.55
7	50	200	200	50	232.41
7	0	150	150	25	236.01
7	50	100	100	50	237.59
11	50	50	50	25	239.20
5	0	150	150	25	246.96
5	50	100	100	25	248.34
7	50	50	50	25	260.09
5	100	50	50	25	261.48
5	50	50	50	50	263.49
5	50	50	50	25	269.69

Table 4.2: NPLM trained on the c-news dataset, evaluated on the heldout set.

noise	**clusters**	**objective**
10	0	1.74
10	387	4.79
25	387	6.25
25	0	6.29
25	0	6.29
25	0	6.29
0	0	6.29

Table 4.3: Results for OxLM trained on the c-news dataset.

All parameters that are not specified in the experiment description are set to their default values (listed in Attachment A.1).

We use a simple version of early stopping introduced by Prechelt [1998]. We measure the perplexity of the development set after each epoch and if it is higher than the last one, we take the model from the last epoch as the final model.

Unless noted otherwise, we report perplexity on the development set in the following experiments.

4.3.1 Embedding Size

In Table 4.4, we see the results of training the model with different embedding sizes for a 5-gram model. The perplexity in this table is reported on the heldout dataset. The performance of our baseline model is similar to the performance of NPLM.

In Table 4.5, the same experiment is repeated with full development set. For the c-news dataset, the ideal embedding size seems to be around 300.

In Table 4.6, we see the results of the same experiment repeated on the c-fiction training dataset. The difference between embedding size from 200 up to 1000 was not significant in this experiment. This may be because the processing of the larger dataset is computationally expensive and we did not have the resources to repeat the training as many times as with the smaller dataset. Or it may be a consequence of the bigger dataset acting as regularization, enabling the model to use bigger embeddings without overfitting.

4.3.2 N-gram Order

Table 4.7 shows perplexity of models of different n-gram orders on the c-news dataset. The best results were obtained with n-gram order of 7. It seems that models with higher order n-grams are slightly overfitting.

4.3.3 Computation Time

Training of neural networks can take a lot of time. When we run the default configuration of *formLM* on the Intel® Core™ i5-4210Y CPU, it takes around 34 seconds per 200 batches (with 1 000 datapoints each). On the computer cluster at ÚFAL MFF UK, it takes between 20 and 40 seconds per 200 batches, depending on which cluster node is the job assigned to.

Training runs faster on hardware that is constructed to perform fast matrix multiplication. TensorFlow supports computations on GPU. On Nvidia GeForce GTX 1080, the same configuration of *morphLM* takes approximately 2.7 seconds per 200 batches, which is around 10 times faster than on a CPU.

4.3.4 Final Model

Based on the experiments described above, we selected parameters for the *formLM* model that was compared against the *morphLM* architecture:

word embedding size: 300,

emb. size	**perplexity**	σ
50	214.34	7.50
100	203.13	10.03
150	203.07	5.95
200	199.38	6.59
250	201.94	3.58
300	203.11	7.78
400	203.10	5.02
500	200.80	5.23
600	206.59	7.88
750	211.69	6.64
1000	222.91	9.94
1250	223.28	8.15
1500	233.04	7.23
2000	238.11	11.60

Table 4.4: *FormLM* results on the c-news dataset, evaluated on the heldout set. Perplexity is averaged over 10 runs for each configuration.

emb. size	**perplexity**	σ
50	474.30	51.31
100	421.07	28.60
200	409.28	41.60
300	355.49	11.77
500	388.23	16.93
1000	384.17	9.92
2000	429.79	11.93

Table 4.5: *FormLM* results on the c-news dataset.

emb. size	**perplexity**
50	230.44
100	219.73
200	213.47
300	214.22
500	214.32
750	214.08
1000	211.89
1500	218.28
2000	216.75

Table 4.6: *FormLM* results on the c-fiction dataset.

first hidden layer size: 500,

second hidden layer size: 500,

n**-gram order:** 7,

softmax estimation: sampled softmax,

sampling noise: 500,

maximum number of epochs: 15.

We trained 5 models and selected the one with the lowest perplexity on the development set (246.81). The perplexity of this model on the test dataset was 273.26.

4.4 Neural Language Model with Morphology

In this section we discuss the experiments with *morphLM*, our proposed NLM architecture.

4.4.1 Embedding Size

The results of training a 5-gram model on the c-news dataset with different embedding sizes are in Table 4.8. These are the numbers evaluated on the same dataset as in the NPLM and first *formLM* experiments—the heldout set. Figure 4.1 shows the comparison of *formLM* and *morphLM*.

Table 4.9 shows the results of the same experiment with full development set. The ideal size of the embeddings seems to be around 300. The *morphLM* model has an overall lower perplexity than the *formLM* model.

The results of the same experiment on the c-fiction dataset are shown in Table 4.10. We see that even on the bigger dataset, *morphLM* has lower perplexity than *formLM*.

4.4.2 Computation Time

When we run the default configuration of *morphLM* on the Intel® Core™ i5-4210Y CPU, the training takes around 54 seconds per 200 batches (with 1 000 datapoints each). Compared to *formLM*, this is roughly 60 % more time. On the Nvidia Tesla K40c GPU the training takes around 36 seconds per 200 batches.

4.4.3 Hidden Layers Size

Table 4.11 shows how the size of the hidden layers affects the perplexity and training time. The perplexity decreases with growing hidden layers, but the number of variables (and time needed for training) grows quadratically. The size of the second hidden layer has a slightly larger effect. This is to be expected, because increasing the size of the second layer adds more variables to the model than increasing the size of the first layer. Setting the size of both hidden layers to 500 seems like a good trade-off.

order	**perplexity**	σ
5	360.06	5.59
7	352.62	5.56
9	363.39	6.97
11	373.91	11.41

Table 4.7: Perplexity of *formLM* trained on the c-news dataset for different n-grams orders. Evaluated on the c-news development set.

emb. size	**perplexity**	σ
50	180.30	7.62
100	174.52	5.32
150	168.97	5.41
200	167.48	4.92
250	169.39	4.99
300	166.37	6.04
400	167.47	3.91
500	167.61	5.50
600	170.48	4.20
750	167.32	5.59
1000	170.36	4.47
1250	166.54	5.01
1500	172.63	6.68
2000	174.40	8.06

Table 4.8: *MorphLM* trained on the c-news dataset, evaluated on the heldout set for comparison with NPLM. Averaged over 10 runs.

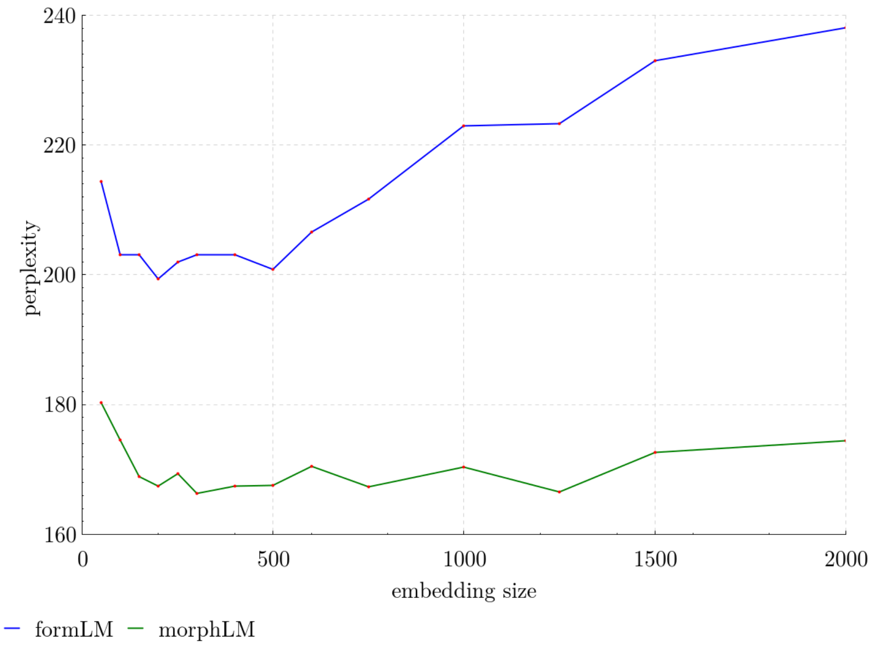

Figure 4.1: Perplexity as a function of embedding size.

emb. size	**perplexity**	σ
50	302.30	25.22
100	290.08	26.22
200	262.62	13.89
300	249.20	6.83
500	274.80	33.05
1000	253.80	11.60
2000	274.86	10.86

Table 4.9: *MorphLM* trained on the c-news dataset. Evaluated on the c-news development set.

emb. size	**perplexity**
50	193.73
100	186.96
200	182.99
300	184.39
500	187.17
750	188.10
1000	183.82
2000	189.08

Table 4.10: *MorphLM* trained on the c-fiction dataset. Evaluated on the c-fiction development set.

4.4.4 Different Morphological Tags

Table 4.12 shows the difference between using the short, 60-bit representation of the morphological tags and the full, 139-bit representation. Models trained with the full representation achieve a lower perplexity.

When we run the configuration with full tags on the Intel® Core™ i5-4210Y CPU, the training takes around 86 seconds per 200 batches (with 1 000 datapoints each). Compared to the default configuration with shorter tags, this is roughly 60 % more time.

4.4.5 Training Method

We tested two training methods, Adam and SGD. The training setup was *morphLM* with embedding size of 200 and 50 noise samples for each data sample. The resulting learning curves for 5 runs of each method are plotted in Figure 4.2. We can see that Adam gets its best results around epoch 7, then it is rather prone to overfitting. SGD reaches the best results around epoch 15 and doesn't overfit that much. Average perplexity is shown in Table 4.13.

On the Intel® Core™ i5-4210Y CPU, it takes around 38 seconds to train 200 batches (1 000 datapoints each) using SGD. Using Adam, the same amount of training takes around 50 seconds, so it is around 25 % slower. Because it achieves the best results in approximately half number of epochs, it would be faster to train the model with Adam for a lower number of epochs.

4.4.6 Sampled Softmax versus NCE

Table 4.14 shows the comparison between the sampled softmax method and NCE. The experiment was performed on the c-news dataset, with 10 runs for each method. Sampled softmax outperformed NCE in perplexity. In some experiments, NCE suffered from arithmetic underflow.

In our experiments, there was no significant difference in training time between sampled softmax and NCE.

4.4.7 Number of Noise Samples

Table 4.15 shows the relation between number of noise samples and perplexity for the sampled softmax method. The more noise the better, although at around 500 samples the effect starts to fade out.

4.4.8 Number of Epochs

In Figure 4.3, we see learning curves for 5 runs each of *morphLM* and *formLM* on the c-news dataset. After epoch 20, both models start to overfit. The *morphLM* model is overfitting less, even though we used larger embeddings for the lemmata (300) than for the forms (200) in *formLM*, so *morphLM* has more parameters. This is probably because the word forms are sparser, making them easier to overfit to.

h_1 **size**	h_2 **size**	**perplexity**	σ	**time (s)**
200	150	305.54	9.56	
200	200	301.40	8.98	60.00
200	300	283.30	6.59	
200	500	278.85	8.30	
200	750	265.41	3.67	
300	150	293.41	7.97	
300	200	285.85	9.72	
300	300	283.33	4.46	67.52
300	500	268.26	2.14	
300	750	261.94	4.60	
500	150	292.24	6.66	
500	200	281.14	7.47	
500	300	272.34	6.28	
500	500	262.07	5.09	86.06
500	750	258.85	4.88	
750	150	280.88	5.84	85.21
750	200	275.55	4.50	
750	300	263.49	4.06	
750	500	253.49	3.41	
750	750	250.57	4.18	113.47

Table 4.11: Effect of the hidden layers size.

tags	**perplexity**	σ
full	274.90	6.12
short	285.30	9.83

Table 4.12: Effect of different encoding of morphological tags.

method	**perplexity**	σ
adam	309.27	2.45
SGD	246.59	3.18

Table 4.13: Effect of different training methods.

method	**perplexity**	σ
NCE	323.48	12.71
sampled softmax	288.47	9.89

Table 4.14: Sampled softmax versus NCE.

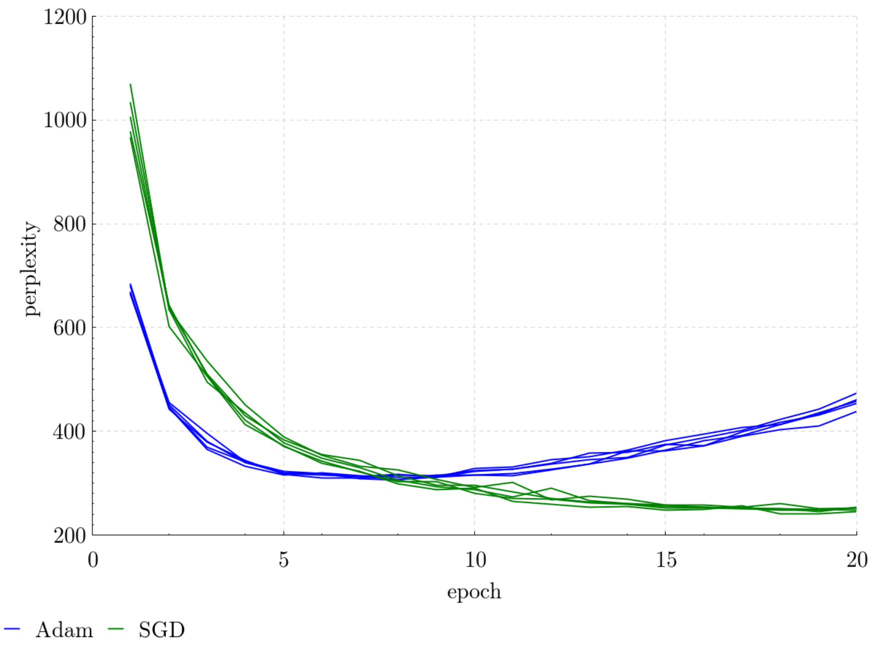

Figure 4.2: Learning curves for Adam and SGD.

noise	perplexity	σ	time (s)
10	440.30	12.27	54.23
50	280.80	4.42	55.54
100	247.77	2.07	57.75
200	226.26	5.06	59.72
300	218.84	9.21	62.61
500	203.84	3.53	68.08
750	197.51	2.55	75.49
1000	204.64	4.59	82.68
1500	194.86	3.21	96.57

Table 4.15: Perplexity and time with respect to sampling noise size.

4.4.9 N-gram Order

Table 4.16 shows the results for different n-gram orders on the c-news dataset, averaged over 5 runs. It seems that 7-grams are the best for this dataset, however the difference between orders 7 and 9 is not as big as in case of *formLM*. This may signal that *morphLM* is less prone to overfitting.

4.4.10 Final Model

Based on the experiments described above, we selected parameters for the *morphLM* model that was compared against the *formLM* architecture:

lemma embedding size: 300,

first hidden layer size: 500,

second hidden layer size: 500,

n**-gram order:** 7,

softmax estimation: sampled softmax,

sampling noise: 500,

morphological tags encoding: full (139 bits),

maximum number of epochs: 15.

We trained 5 models and selected the one with the lowest perplexity on the development set (193.66). The perplexity of this model on the test dataset was 206.46, significantly lower than test perplexity for *formLM* (273.26). The comparison of average perplexities for all models is in Table 4.17, their learning curves are plotted in Figure 4.4.

In the experiments that we performed to select the best parameters, the variance was relatively high, so we cannot be sure that we really selected the best parameters. We also did not have the resources to test the whole parameter space—we always tested one parameter at the time, not accounting for possible interaction between changes of different parameters. The parameters were selected to minimize the expected perplexity of the model, given the results of the experiments that we were able to perform.

From the final experiment, it is clear, that *morphLM* achieves lower perplexity than *formLM*.

Our implementation also reports accuracy on the development set. The accuracy was growing during the training, achieving the maximum of 22 % for *formLM* and 24 % for *morphLM*. The top-10 accuracy (probability that the correct word was within the first ten highest-scoring possibilities) achieved the maximum of 46 % for *formLM* and 49 % for *morphLM*.

In this final setup, *morphLM* training was on average around two times slower than *formLM* training.

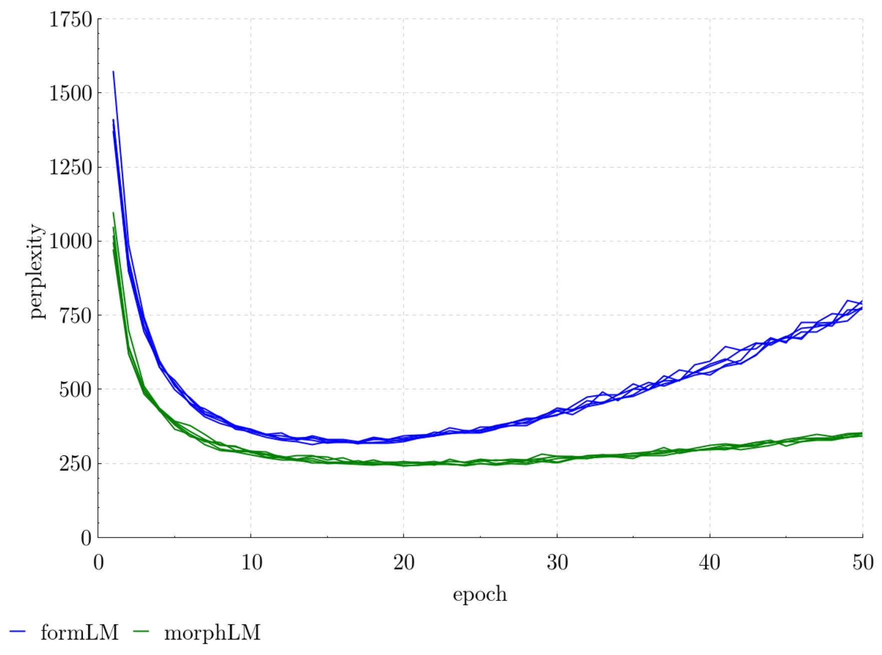

Figure 4.3: Learning curves for *morphLM* and *formLM*.

order	**perplexity**	σ
5	286.12	5.85
7	282.01	7.51
9	282.61	4.94
11	289.21	5.16

Table 4.16: Perplexity for different n-grams orders.

model	**dev**	σ	**test**	σ
morphlm	161.93	1.28	272.55	8.45
formlm	211.91	2.47	354.97	21.56

Table 4.17: Perplexity for final models.

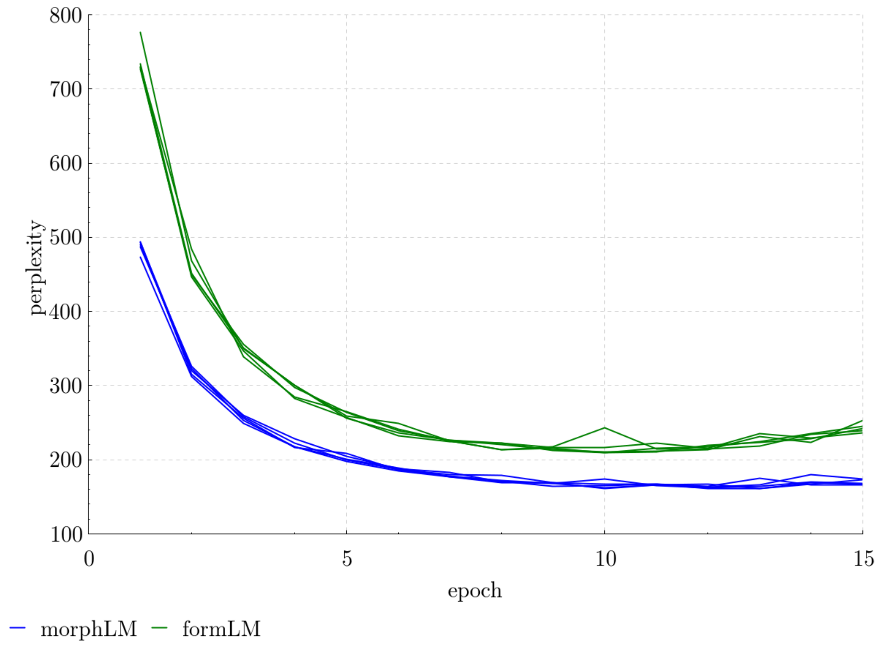

Figure 4.4: Learning curves for the final models.

Chapter 5

Experiments with Machine Translation

In this chapter, we describe our experiments with statistical machine translation.

5.1 Datasets

For training the SMT systems, we used the c-news dataset, described in section 4.1.

For testing the systems, we used the c-news test set and also the WMT2013 test set [Bojar et al., 2013b] with multiple reference translations (see Table 5.1). We will refer to this dataset as the *bigref* dataset.

English	**Czech**	**# Refs**	**# Snts**
newstest2011	official + 3 more from German	4	3003
newstest2011	2 post-edits of a system similar to Bojar et al. [2012a]	2	1997
newstest2009	official	1	2525
newstest2008	official	1	2051
newstest2007	official	1	2007
Total		4	11583

Table 5.1: The contents of the *bigref* dataset. Table reproduced from [Bojar et al., 2013b].

5.2 Baseline Experiments

We trained Moses on the c-news dataset with various language models to compare our models with. In the end, we selected three configurations for further testing. All configurations contain a basic back-off LM, estimated with KenLM Toolkit [Heafield et al., 2013]. It is inexpensive to compute and use and it is standard practice to use the NLM alongside a statistical n-gram model. In these experiments, the NLM was used directly in the Moses decoder. The results, averaged over 5 runs, are shown in Table 5.2. Both models were 7-gram models, with configuration selected on the basis of experiments described in the previous chapter. OxLM took much more time to train and to translate with, but it seems to help, although the gain of 0.3 BLEU points is not significant. NPLM was faster, but did not improve the translation.

5.3 Experiments with the Proposed Architecture

We first tried to use our models in the Moses decoder. We wrote a plugin for Moses to run TensorFlow language models. Unfortunately, TensorFlow turned out to be too slow to be usable in the decoder.

5.3.1 N-best List Reranking

We used our models to rerank *n*-best lists. We produced a 500-best list for the development set using Moses with a 5-gram KenLM. The hypotheses were lemmatized and tagged with MorphoDiTa [Straková et al., 2014]. We added the score from our models as a new feature to the 500-best list and trained the reranking weights with k-best MIRA [Cherry and Foster, 2012].

Because the last stage of Moses training—MERT [Och, 2003]—and the training of the reranking weights are both random processes, we repeated each of them five times and processed the results with MultEval [Clark et al., 2011]. In the following tables, the baseline is just the translation of the test set with the same model that produced the 500-best list for the development set, *formLM* and *morphLM* are 100-best lists of the test set translations reranked by our models.

We tested our final *formLM* and *morphLM* models (described in the previous section) on the c-news test set. We also tested two models trained with a larger (100 000) vocabulary. The results are summarized in Table 5.3. We also tested our final models on the *bigref* dataset (Table 5.4) with multiple references. With multiple references, there is lower risk of the situation where a system actually achieves better translation, but it is not evaluated as such, because it deviated from the reference too much.

Reranking *n*-best lists improves translation for both *formLM* and *morphLM*, although not significantly. *MorphLM* did not perform better than *formLM* in these tests. Models with larger vocabulary performed slightly better, but the difference is not significant.

We also tested our models on a single MERT run, that produced much better translations, than the rest. The results are in Table 5.5. The effect of the reranking was stronger here and *morphLM* performed slightly better than *formLM*. We think that reranking works better on better baseline, because the LMs are only trained to distinguish good sentences from noise, not bad sentences from worse sentences. However, we were not able to replicate this result and the difference between *formLM* and *morphLM* probably is not significant here.

LMs	**dev**	**test**
KLM + OxLM	**26.9**	**19.9**
KLM + NPLM	26.3	19.3
KLM	26.2	19.6

Table 5.2: Translation results for Moses with various configurations of NPLM and OxLM. BLEU for the c-news development and test datasets. Test results are averaged over 5 MERT runs.

Metric	**System**	**Avg**	$\overline{s}_{\textbf{sel}}$	$s_{\textbf{Test}}$	p**-value**
BLEU $\uparrow$	baseline	19.6	0.8	0.1	-
	formLM	19.9	0.8	0.0	0.00
	morphLM	19.8	0.8	0.0	0.00
	*formLM*100	**20.0**	0.8	0.0	0.00
	*morphLM*100	**20.0**	0.8	0.1	0.00
METEOR $\uparrow$	baseline	23.1	0.4	0.0	-
	formLM	23.2	0.4	0.0	0.00
	morphLM	23.2	0.4	0.0	0.00
	*formLM*100	**23.3**	0.4	0.0	0.00
	*morphLM*100	23.2	0.4	0.0	0.00
TER $\downarrow$	baseline	67.1	0.7	0.2	-
	formLM	66.8	0.7	0.1	0.00
	morphLM	66.9	0.7	0.1	0.00
	*formLM*100	**66.6**	0.7	0.1	0.00
	*morphLM*100	66.7	0.7	0.1	0.00
Length	baseline	102.2	0.4	0.2	-
	formLM	102.4	0.4	0.1	0.00
	morphLM	102.4	0.4	0.0	0.01
	*formLM*100	102.3	0.4	0.0	0.24
	*morphLM*100	102.3	0.4	0.2	0.32

Table 5.3: Reranking with final *formLM* and *morphLM* models. Measured on the c-news test set. *FormLM*100 and *morphLM*100 are models with 100 000 words vocabulary. Metric scores for all systems: jBLEU V0.1.1 (an exact reimplementation of NIST's mteval-v13.pl without tokenization); Meteor V1.4 cz on rank task with all default modules NOT ignoring punctuation; Translation Error Rate (TER) V0.8.0; Hypothesis length over reference length as a percent. *P*-values are relative to baseline and indicate whether a difference of this magnitude (between the baseline and the system on that line) is likely to be generated again by some random process (a randomized optimizer). Metric scores are averages over multiple runs. s_{sel} indicates the variance due to test set selection and has nothing to do with optimizer instability.

Metric	**System**	**Avg**	$\overline{s}_{\textbf{sel}}$	$s_{\textbf{Test}}$	p**-value**
	baseline	29.0	0.4	0.1	-
BLEU $\uparrow$	*formLM*	**29.2**	0.4	0.1	0.00
	morphLM	**29.2**	0.4	0.0	0.00
	baseline	25.7	0.2	0.0	-
METEOR $\uparrow$	*formLM*	**25.9**	0.2	0.0	0.00
	morphLM	**25.9**	0.2	0.0	0.00
	baseline	**61.3**	0.3	0.2	-
TER $\downarrow$	*formLM*	61.4	0.3	0.2	0.01
	morphLM	**61.3**	0.3	0.1	0.46
	baseline	105.2	0.1	0.2	-
Length	*formLM*	105.6	0.1	0.3	0.00
	morphLM	105.6	0.1	0.2	0.00

Table 5.4: Reranking with final *formLM* and *morphLM* models. Measured on the *bigref* test set. For description of the metrics, see Table 5.3.

LMs	**dev-BLEU**	**test-BLEU**
morphLM	26.62	**24.86**
formLM	**26.72**	24.76
baseline	26.24	24.14

Table 5.5: Reranking results for our models on the single best baseline MERT run.

Conclusion

We have examined various architectures of neural language models, both in theory and empirically. We proposed a new NLM architecture, that works with explicit morphological information, to take advantage of annotation tools and annotated datasets.

We were able to train language models with a significantly lower perplexity with the proposed architecture. However, it did not bring significant improvements to statistical machine translation with n-best list rescoring, compared to baseline NLM architecture.

Our implementation is too slow to be used in a SMT decoder. It might be significantly faster, if it was reimplemented in C++, but our results suggest that this is not worth the effort.

Our results suggest that there is a potential to improve machine translation by including morphological information into language models. Recent developments in machine translation seem to favor neural machine translation with various subword approaches. An NMT system that combines a subword encoding with explicit morphological information might be a more promising way of including morphological annotation into machine translation.

Future Work

While this thesis explored possibilities of using neural language models with morphology in statistical machine translation, many opportunities for extending the scope of this thesis still remain. This section presents some of these directions.

Different tasks NLMs with morphology may be used in task other than machine translation, for example spelling correction. For inflected languages, typing errors may result in word forms that exist, but are not correct in the given context. Dictionary-based spellcheckers have no way to correct errors of this type. A LM that works with morphology might be a good tool for this task. Richter et al. [2012] used language models on factors in Korektor,[1] a statistical spellchecker and (occasional) grammar checker. Our proposed model may be effective in a similar tool. Other possible applications include speech recognition, text summarization, and other tasks where LMs are used.

Embeddings analysis In this thesis, we worked with various embeddings. It may be interesting to compare how for example word form and lemmata embeddings perform in various semantic tasks (e.g. the word similarity task from Mikolov et al. [2013a]), similarly as e.g. Hill et al. [2017] compare embeddings from LM-like tasks and NMT. Our hypothesis is that lemmata embeddings would work better than word form embeddings and that lemmata embeddings trained with an architecture that processes the morphological information would work better than embeddings trained on lemmata alone.

Using morphology in NMT Because an NMT system may be regarded as two connected recurrent language models, our approach to morphology can be used in NMT as well. Possible variants include:

- using lemmata and morphological tags in a single encoder,
- using an encoder for lemmata and another one for morphological tags in a multi-encoder architecture (similar to the architecture used for multimodal translation by Libovický et al. [2016]),
- using an encoder for morphological tags in combination with an sub-word units encoder.

[1]`http://ufal.mff.cuni.cz/korektor`

Usage of morphological tags It may be possible to find interesting information about the morphological tags usage. This would consist in training the proposed architecture with various subsets of morphological tags, testing them on various tasks and determining which parts of the tag are most useful for which task.

More languages Similar experiments may be carried out for other inflected languages than Czech, for example other Slavic languages, Baltic languages, German or Sanskrit. We expect that our proposed architecture would be better for languages that are highly inflected.

Bibliography

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL `http://tensorflow.org/`. Software available from tensorflow.org.

Andrei Alexandrescu and Katrin Kirchhoff. Factored neural language models. In *Proceedings of the Human Language Technology Conference of the NAACL, Companion Volume: Short Papers*, pages 1–4. Association for Computational Linguistics, 2006.

Tamer Alkhouli, Felix Rietig, and Hermann Ney. Investigations on phrase-based decoding with recurrent neural network language and translation models. 2015.

Paul Baltescu and Phil Blunsom. Pragmatic neural language modelling in machine translation. *CoRR*, abs/1412.7119, 2014. URL `http://arxiv.org/abs/1412.7119`.

Paul Baltescu, Phil Blunsom, and Hieu Hoang. OxLM: A neural language modelling framework for machine translation. *The Prague Bulletin of Mathematical Linguistics*, 102(1):81–92, 2014.

Yoshua Bengio and Jean-Sébastien Senécal. Adaptive importance sampling to accelerate training of a neural probabilistic language model. *IEEE Transactions on Neural Networks*, 19(4):713–722, 2008.

Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of Machine Learning Research*, 2003.

Jeff A Bilmes and Katrin Kirchhoff. Factored language models and generalized parallel backoff. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology: companion volume of the Proceedings of HLT-NAACL 2003–short papers-Volume 2*, pages 4–6. Association for Computational Linguistics, 2003.

Ondřej Bojar and Aleš Tamchyna. The design of eman, an experiment manager. *The Prague Bulletin of Mathematical Linguistics*, 99:39–56, 2013.

Ondrej Bojar, Bushra Jawaid, and Amir Kamran. Probes in a taxonomy of factored phrase-based models. In *WMT@NAACL-HLT*, 2012a.

Ondřej Bojar, Zdeněk Žabokrtský, Ondřej Dušek, Petra Galuščáková, Martin Majliš, David Mareček, Jiří Maršík, Michal Novák, Martin Popel, and Aleš Tamchyna. The Joy of Parallelism with CzEng 1.0. In *Proceedings of LREC2012*, Istanbul, Turkey, May 2012b. ELRA, European Language Resources Association. In print.

Ondřej Bojar, Matouš Macháček, Aleš Tamchyna, and Daniel Zeman. Scratching the surface of possible translations. In *International Conference on Text, Speech and Dialogue*, pages 465–474. Springer, 2013a.

Ondrej Bojar, Rudolf Rosa, and Ales Tamchyna. Chimera - three heads for english-to-czech translation. In *WMT@ACL*, 2013b.

Ondřej Bojar, Ondřej Dušek, Tom Kocmi, Jindřich Libovický, Michal Novák, Martin Popel, Roman Sudarikov, and Dušan Variš. CzEng 1.6: Enlarged Czech-English Parallel Corpus with Processing Tools Dockered. In Petr Sojka, Aleš Horák, Ivan Kopeček, and Karel Pala, editors, *Text, Speech, and Dialogue: 19th International Conference, TSD 2016*, number 9924 in Lecture Notes in Computer Science, pages 231–238, Cham / Heidelberg / New York / Dordrecht / London, 2016a. Masaryk University, Springer International Publishing. ISBN 978-3-319-45509-9.

Ondřej Bojar, Yvette Graham, Amir Kamran, and Miloš Stanojević. Results of the wmt16 metrics shared task. In *Proceedings of the First Conference on Machine Translation*, pages 199–231, Berlin, Germany, August 2016b. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/W/W16/W16-2302`.

Ondřej Bojar. *Čeština a strojový překlad: Strojový překlad našincům, našinci strojovému překladu.* ÚFAL, 2012.

Jan Botha and Phil Blunsom. Compositional morphology for word representations and language modelling. In *International Conference on Machine Learning*, pages 1899–1907, 2014.

Denny Britz, Anna Goldie, Minh-Thang Luong, and Quoc V. Le. Massive exploration of neural machine translation architectures. *CoRR*, abs/1703.03906, 2017. URL `http://arxiv.org/abs/1703.03906`.

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.

Welin Chen, David Grangier, and Michael Auli. Strategies for training large vocabulary neural language models. *arXiv preprint arXiv:1512.04906*, 2015.

Colin Cherry and George Foster. Batch tuning strategies for statistical machine translation. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 427–436. Association for Computational Linguistics, 2012.

Jonathan H Clark, Chris Dyer, Alon Lavie, and Noah A Smith. Better hypothesis testing for statistical machine translation: Controlling for optimizer instability. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies: short papers-Volume 2*, pages 176–181. Association for Computational Linguistics, 2011.

Jacob Devlin, Rabih Zbib, Zhongqiang Huang, Thomas Lamar, Richard Schwartz, and John Makhoul. Fast and robust neural network joint models for statistical machine translation. In *52nd Annual Meeting of the Association for Computational Linguistics, Baltimore, MD, USA, June*, 2014.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

Michael Gutmann and Aapo Hyvärinen. Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 297–304, 2010.

Jan Hajič. *Disambiguation of Rich Inflection (Computational Morphology of Czech)*. Karolinum, Charles University Press, Prague, Czech Republic, 2004.

Jan Hajič, Jarmila Panevová, Eva Hajičová, Petr Sgall, Petr Pajas, Jan Štepánek, Jiří Havelka, Marie Mikulová, Zdeněk Žabokrtský, and Magda Ševcíková-Razímová. Prague Dependency Treebank 2.0. *CD-ROM, Linguistic Data Consortium, LDC Catalog No.: LDC2006T01, Philadelphia*, 98, 2006.

Jan Hajič, Jan Votrubec, Pavel Krbec, Pavel Květoň, et al. The best of two worlds: Cooperation of statistical and rule-based taggers for czech. In *Proceedings of the Workshop on Balto-Slavonic Natural Language Processing: Information Extraction and Enabling Technologies*, pages 67–74. Association for Computational Linguistics, 2007.

Kenneth Heafield, Ivan Pouzyrevsky, Jonathan H. Clark, and Philipp Koehn. Scalable modified Kneser-Ney language model estimation. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics*, pages 690–696, Sofia, Bulgaria, August 2013. URL `https://kheafield.com/papers/edinburgh/estimate_paper.pdf`.

Felix Hill, Kyunghyun Cho, Sébastien Jean, and Yoshua Bengio. The representational geometry of word meanings acquired by neural machine translation models. *Machine Translation*, 31(1):3–18, Jun 2017. ISSN 1573-0573. doi: 10.1007/s10590-017-9194-2. URL `https://doi.org/10.1007/s10590-017-9194-2`.

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

Timo Honkela. Philosophical aspects of neural, probabilistic and fuzzy modeling of language use and translation. In *Neural Networks, 2007. IJCNN 2007. International Joint Conference on*, pages 2881–2886. IEEE, 2007.

Sébastien Jean, Kyunghyun Cho, Roland Memisevic, and Yoshua Bengio. On using very large target vocabulary for neural machine translation. *CoRR*, abs/1412.2007, 2014. URL `http://arxiv.org/abs/1412.2007`.

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

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

Tom Kocmi and Ondřej Bojar. *SubGram: Extending Skip-Gram Word Representation with Substrings*, pages 182–189. Springer International Publishing, 2016. ISBN 978-3-319-45510-5. URL `http://dx.doi.org/10.1007/978-3-319-45510-5_21`.

Philipp Koehn. *Statistical machine translation.* Cambridge University Press, 2009.

Philipp Koehn and Hieu Hoang. Factored translation models. In *EMNLP-CoNLL*, pages 868–876, 2007.

Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, et al. Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th annual meeting of the ACL on interactive poster and demonstration sessions*, pages 177–180. Association for Computational Linguistics, 2007.

Michael Denkowski Alon Lavie. Meteor universal: Language specific translation evaluation for any target language. *ACL 2014*, page 376, 2014.

Hai-Son Le, Ilya Oparin, Alexandre Allauzen, Jean-Luc Gauvain, and François Yvon. Structured output layer neural network language model. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pages 5524–5527. IEEE, 2011.

Jindřich Libovický, Jindřich Helcl, Marek Tlustý, Pavel Pecina, and Ondřej Bojar. CUNI system for WMT16 automatic post-editing and multimodal translation tasks. *arXiv preprint arXiv:1606.07481*, 2016.

Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(Nov):2579–2605, 2008.

Jiří Maršík and Ondřej Bojar. TrTok: a fast and trainable tokenizer for natural languages. *The Prague Bulletin of Mathematical Linguistics*, 98:75–85, 2012.

Tomáš Mikolov. *Statistical language models based on neural networks.* PhD thesis, Brno University of Technology, 2012.

Tomáš Mikolov, Anoop Deoras, Stefan Kombrink, Lukáš Burget, and Jan Černocký. Empirical evaluation and combination of advanced language modeling techniques. In *INTERSPEECH*, pages 605–608, 2011.

Tomáš Mikolov, Ilya Sutskever, Anoop Deoras, Hai-Son Le, Stefan Kombrink, and Jan Černocky. Subword language modeling with neural networks. *preprint (http://www.fit.vutbr.cz/imikolov/rnnlm/char.pdf)*, 2012.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013a. URL `http://arxiv.org/abs/1301.3781`.

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, 2013b.

Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. Linguistic regularities in continuous space word representations. In *HLT-NAACL*, pages 746–751. Citeseer, 2013c.

Andriy Mnih and Koray Kavukcuoglu. Learning word embeddings efficiently with noise-contrastive estimation. In *Advances in Neural Information Processing Systems*, pages 2265–2273, 2013.

Andriy Mnih and Yee Whye Teh. A fast and simple algorithm for training neural probabilistic language models. *arXiv preprint arXiv:1206.6426*, 2012.

Jan Niehues, Thanh-Le Ha, Eunah Cho, and Alex Waibel. Using factored word representation in neural network language models. In *Proceedings of the First Conference on Machine Translation*, pages 74–82, Berlin, Germany, August 2016. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/W/W16/W16-2208`.

Franz Josef Och. Minimum error rate training in statistical machine translation. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics-Volume 1*, pages 160–167. Association for Computational Linguistics, 2003.

Marian Olteanu, Pasin Suriyentrakorn, and Dan Moldovan. Language models and reranking for machine translation. In *Proceedings of the Workshop on Statistical Machine Translation*, pages 150–153. Association for Computational Linguistics, 2006.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics, 2002.

Peyman Passban, Qun Liu, and Andy Way. Providing morphological information for SMT using neural networks. *The Prague Bulletin of Mathematical Linguistics*, 108(1):271–282, 2017.

Lutz Prechelt. Early stopping-but when? *Neural Networks: Tricks of the trade*, pages 553–553, 1998.

Michal Richter, Pavel Straňák, and Alexandr Rosen. Korektor–a system for contextual spell-checking and diacritics completion. In Martin Kay and Christian Boitet, editors, *Proceedings of the 24th International Conference on Computational Linguistics (Coling 2012)*, pages 1–12, Mumbai, India, 2012. IIT Bombay, Coling 2012 Organizing Committee.

Holger Schwenk. Continuous space language models. *Computer Speech & Language*, 21(3):492–518, 2007.

Holger Schwenk. Continuous-space language models for statistical machine translation. *The Prague Bulletin of Mathematical Linguistics*, 93:137–146, 2010.

Holger Schwenk. CSLM – a modular open-source continuous space language modeling toolkit. In *INTERSPEECH*, pages 1198–1202, 2013.

Holger Schwenk, Anthony Rousseau, and Mohammed Attik. Large, pruned or continuous space language models on a GPU for statistical machine translation. In *Proceedings of the NAACL-HLT 2012 Workshop: Will We Ever Really Replace the N-gram Model? On the Future of Language Modeling for HLT*, pages 11–19. Association for Computational Linguistics, 2012.

Rico Sennrich and Barry Haddow. Linguistic input features improve neural machine translation. In *Proceedings of the First Conference on Machine Translation*, pages 83–91, Berlin, Germany, August 2016. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/W/W16/W16-2209`.

Matthew Snover, Bonnie Dorr, Richard Schwartz, Linnea Micciulla, and John Makhoul. A study of translation edit rate with targeted human annotation. In *Proceedings of association for machine translation in the Americas*, volume 200, 2006.

Andreas Stolcke et al. SRILM-an extensible language modeling toolkit. In *Interspeech*, 2002.

Jana Straková, Milan Straka, and Jan Hajič. Open-Source Tools for Morphology, Lemmatization, POS Tagging and Named Entity Recognition. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 13–18, Baltimore, Maryland, June 2014. Association for Computational Linguistics. URL `http://www.aclweb.org/anthology/P/P14/P14-5003.pdf`.

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.

Ernst Tugendhat. The meaning of 'Bedeutung' in Frege. *Analysis*, 30(6):177–189, 1970.

Ashish Vaswani, Yinggong Zhao, Victoria Fossum, and David Chiang. Decoding with large-scale neural language models improves translation. In *EMNLP*, pages 1387–1392. Citeseer, 2013.

Ludwig Wittgenstein. *Philosophical investigations.* Basil Blackwell, 1953.

Dan Zeman, Jiří Hana, Hana Hanová, Jan Hajič, Barbora Hladká, and Emil Jeřábek. A Manual for Morphological Annotation, 2nd edition. Technical Report 27, ÚFAL MFF UK, Prague, Czech Republic, 2005.

List of Abbreviations

ANN artificial neural network

LM language model

LSTM long short-term memory

NCE noise contrastive estimation

NLM neural language model

NMT neural machine translation

NN neural network

OOV out of vocabulary

PDT Prague Dependency Treebank

POS part of speech

RNN recurrent neural network

SGD stochastic gradient descent

SMT statistical machine translation

TER Translation Edit Rate

ÚFAL MFF UK Institute of Formal and Applied Linguistics, Charles University

Appendix A

Attachments

A.1 morphLM User Manual

This is the user manual for morphLM, a neural language model implementation that works with Czech morphological information.

A.1.1 Download

morphLM is available on GitHub: `https://github.com/tomasmcz/morphlm`.

A.1.2 Installation

All you need to run morphLM is to have Python and TensorFlow (version 1.1 or newer) installed.

A.1.3 Training the Model

The training is executed by running the `morphlm.py` script. To get the list of all possible options, run `./morphlm.py -h`. The most important options are described below.

Mandatory Arguments

`--data-train DATA_TRAIN` Training data.

`--data-dev DATA_DEV` Development data.

Data files should be in the following format: one sentence per line, words separated by spaces, each word containing the word form, lemma and morphological tag, separated by vertical bars ("`|`").

`--voc-size VOC_SIZE` Vocabulary size.

`--vocab-forms VOCAB_FORMS` Vocabulary for forms.

`--vocab-lemmata VOCAB_LEMMATA` Vocabulary for lemmata.

Vocabulary files should contain one word per line, starting with special tokens (`<unk>`, `<s>`, `</s>`) and then sorted by frequency in the training data in decreasing order.

Optional arguments

`--order ORDER` N-gram order. The default value is 5.

`--emb-size EMB_SIZE` Embedding size. The default value is 200.

`--h1-size H1_SIZE` Size of the first hidden layer. The default value is 750.

`--h2-size H2_SIZE` Size of the second hidden layer. The default value is 150.

`--max-iter MAX_ITER` Maximum number of iterations. The default value is 10.

`--batch-size BATCH_SIZE` Batch size. The default value is 1000.

`--keep-models KEEP_MODELS` Number of saved models to keep. The default value is 3.

`--sampled-softmax` Use sampled softmax. The default value is True.

`--use-nce` Use NCE. The default value is False.

`--adam` Use Adam instead of SGD. The default value is False.

`--morph` Use lemmata and morphological tags. This is the default.

`--nomorph` Use word forms.

`--noise NOISE` Number of noise samples. The default value is 10.

`--prefix PREFIX` Path prefix for storing the experiment data. The default value is ".".

`--experiment EXPERIMENT` Experiment label. The default value is "test".

`--l1-c L1_C` L_1 regularization coefficient. The default value is 0.

`--l2-c L2_C` L_2 regularization coefficient. The default value is 0.

`--dropout DROPOUT` Dropout coefficient. The default value is 1, meaning that no dropout is used.

`--full-tags` Use the longer, 139 bit reprezentation of morphological tags. The default is the shorter, 60 bit representation.

A.1.4 Running the Model

To run a trained model, use the `morphlm-run.py` script. The mandatory arguments are:

`--data DATA_FILE` Data in the format described above.

`--prefix PREFIX` **and** `--experiment EXPERIMENT` Path and experiment label to find your model data.

`--vocab-forms VOCAB_FORMS` Vocabulary for forms.

`--vocab-lemmata VOCAB_LEMMATA` Vocabulary for lemmata.

The following arguments have to be set to the same values as in the training of the model: `--order`, `--morph`, and `--full-tags`.

For each sentence in the data, the script prints a line with the sum of the logarithms of the inferred probabilities and the number of tokens in that sentence.