Convolutional Encoders in Sequence-to-Sequence Lemmatizers

Jessica Stringham

Master of Science
Artificial Intelligence
School of Informatics
University of Edinburgh
2018

Abstract

Encoder-decoder sequence-to-sequence (seq2seq) neural networks such as Lematus (Bergmanis and Goldwater, 2018) have achieved high accuracy in lemmatization, which is an important preprocessing step in natural language processing. Taking inspiration from other uses of convolutions in NLP tasks, this dissertation explores replacing one component of the network architecture, the bidirectional recurrent encoder, with a convolutional encoder in a new lemmatizer called ConvLematus. This dissertation then explores the effect of using different configurations of the convolutional encoder. The architectures are trained on multiple languages using a new lemmatization dataset derived from CoNLL–SIGMORPHON 2017 (Cotterell et al., 2017). It then compares the lemmatization ability of ConvLematus to Lematus configuration that is identical except for the encoder component. The models are compared on multiple languages and datasets, including using POS tags, context, and a more natural distribution of words from Universal Dependencies (Nivre et al., 2017a). Some of the ConvLematus configurations tested achieve as high of validation accuracy as Lematus, though none clearly outperformed Lematus. However, where ConvLematus excels is lemmatization and training time. The average lemmatization time of ConvLematus is 53% the time Lematus takes, and the average model training time score is 55% that of than Lematus. A comparison of which words ConvLematus and Lematus incorrectly lemmatized shows that both models are modifying words reasonably given they don't have access to a lexical resource (e.g. lemmatizing `shelling` to `shel` is reasonable if a model doesn't know `shell` is a word.

Acknowledgements

I would like to thank my advisor, Sharon Goldwater, who brought us all together, asked insightful questions, and gave useful feedback. I also thank Sameer Bansal, Clara Vania, and Alexander Robertson for the feedback, suggestions, and helping me focus my work.

I thank my fellow lemmatizers, Maria Corkery, Bogomil Gospodinov, Faheem Kirefu, Sahand Shamal, Lauren Watson, and Shijie Yao for reading drafts and discussing work and life over coffee.

I also thank Jennifer Williams, Antreas Antoniou, and Janie Sinclair for sharing about natural language processing, deep learning, and how to do research.

Finally, I thank Albin and the rest of my family for their support and encouragement.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

An interesting development in machine learning and neural network research is seq2seq learning. A seq2seq neural network architecture can be used to learn how to convert an input sequence into an output sequence. These seq2seq learners can be used in a variety of natural language processing (NLP) problems, such as lemmatization (producing a lemma of a provided word) (Bergmanis and Goldwater, 2018), neural machine translation (translating text from one language to another) (Bahdanau et al., 2015), and text-to-speech (converting text into an audio signal) (Van Den Oord et al., 2016). There are many possible neural network architectures that can be used for seq2seq learning. This dissertation focuses on the lemmatization task in order to explore whether a novel seq2seq lemmatization architecture that uses a convolutional component, called ConvLematus, can lemmatize faster and as accurately as a more typical recurrent seq2seq architecture, Lematus (Bergmanis and Goldwater, 2018).

In natural language, the spelling of words is often modified to account for features such as number, tense, and case. For example, in `I am writing`, `writing` is the *inflected form* of the lemma `write` and uses the inflectional *affix* `+ing`. Lemmatization converts a given inflected form into its lemma, where the lemma is the form that would appear in a dictionary. Additional examples of English lemmatization are shown in Table 1.1. Lemmatization has been used as a preprocessing step in NLP tasks, for example, as a way to normalize text (Jurafsky and Martin, 2014).

While this dissertation focuses on a machine learning technique for lemmatization, it is useful to consider alternative ways to build a lemmatizer to explain the challenges of lemmatization. One way to build a lemmatizer is to build a word-

`painted`	→	`paint`	`running`	→	`run`
`changed`	→	`change`	`generations`	→	`generation`
`occurred`	→	`occur`	`treehouses`	→	`treehouse`
`caught`	→	`catch`			

Table 1.1: Examples of English lemmatization of seven words. The inflected form is shown to the left of the arrow and its corresponding lemma is shown to the right of the arrow.

lemma-dictionary that maps each inflected form to its corresponding lemma. For example, the dictionary could include the pairs shown in Table 1.1. However, the word-lemma-dictionary lemmatizer would not be able to lemmatize words that are missing from its dictionary, for example, it may not contain all new words (`podcasting`) or domain-specific words (`andesites`). This problem of missing words can also be called an *out-of-vocabulary* problem. Another problem with a word-lemma-dictionary is that the size of the dictionary can become very large, especially in a morphologically complex language such as Finnish. For example, Finnish has 62 conjugations of the verb *to read*, and all 62 conjugations that verb would need to appear in the dictionary. A second approach to lemmatization which mitigates the out-of-vocabulary problem and reduces the size of the dictionary is to leverage the patterns that appear in lemmatization. For example, a rule to "remove `+ed`" would correctly lemmatize the English words `climbed`, `painted`, and `cooked`. Additional rules would be needed for other inflected forms such as those in Table 1.1. Defining lemmatization rules and exceptions can be time-consuming and error-prone, so a third approach, and the one used in this dissertation, is to use machine learning to automatically learn how to lemmatize a language from a large dataset. Specifically, this machine learning technique for lemmatization trains a model on a dataset containing examples of pairs of lemmas and the corresponding inflected form. The models are given a inflected form and must correctly predict the corresponding lemma (Chrupala, 2006; Chakrabarty et al., 2017; Bergmanis and Goldwater, 2018). This dissertation explores a specific machine learning lemmatizer, a seq2seq neural network that uses a convolutional encoder called ConvLematus. The seq2seq learning approach and ConvLematus will be described next.

Seq2seq learning has successfully been used for lemmatization (Bergmanis and

Goldwater, 2018; Chakrabarty et al., 2017), as well as the related tasks neural machine translation (Sennrich et al., 2016; Lee et al., 2016; Bahdanau et al., 2015; Gehring et al., 2017) and morphological inflection (Cotterell et al., 2017). In the general seq2seq formulation, a deep neural network learns to convert a sequence from the source class into its corresponding sequence from the target class by training on examples of source and target sequence pairs. In the case of lemmatization in this dissertation, the input sequence is made up of the characters of the word, and the output sequence is made up of the characters of the lemma. For example, in

$$\boxed{\texttt{p a i n t i n g} \rightarrow \texttt{p a i n t}}$$

to lemmatize `painting`, the sequence of eight characters is passed into the seq2seq model which should output the sequence of five characters for the lemma `paint`.

A type of architecture frequently used for seq2seq learning, the *encoder-decoder seq2seq neural network*, has performed at state-of-the-art levels on lemmatization (Chakrabarty et al., 2017; Bergmanis and Goldwater, 2018), as well as neural machine translation (Sennrich et al., 2016; Gehring et al., 2017). Most variations of the encoder-decoder architecture use a type of component called recurrent units, with Gated Recurrent Unit (GRU) (Cho et al., 2014a) or Long Short-Term Memory (LSTM) (Hochreiter and Schmidhuber, 1997) being particularly common. There has also been some research into using a component called convolutional layers as an alternative to recurrent units in seq2seq and other neural networks for NLP.

Intuitively, a convolutional layer is computed by sliding a window (where the window width is called the *kernel size*) across the input sequence and computing a linear combination of the window and the convolution's parameters and optionally applying a non-linearity (see section 3.2 for equations and more information). Like convolutional layers, recurrent units also apply non-linearities to a linear combination but does so over the current input and the output of the previous recurrent unit. Recurrent units process input sequentially. An illustration of the difference between convolutional layers and recurrent units is shown in Figure 1.1.

One cited benefit of convolutional layers over recurrent units is that they can process input faster because unlike recurrent units, they do not need to wait for the previous timestep to be processed. As a trade-off, convolutional layers only use a finite *receptive field* (the context around the currently viewed element) when processing each element of the sequence while certain recurrent units can use the full sequence. The difference in the receptive fields of convolutional layers and recurrent units is illustrated in Figure 1.1 and described in detail in future chapters (see subsection 4.2.3).

An extreme example can illustrate the importance of the receptive field. If an English lemmatizer only had a finite receptive field of size 1, then when lemmatizing `defined`, the lemmatizer would see both `d`'s as identical. In other words, the lemmatizer would not have the information that the first `d` is at the beginning of the word and so it could not learn that the first `d` is probably part of the lemma. Similarly, the lemmatizer could not learn that the second `d` follows an `e` and is at the end of the word, and could not learn that therefore it is probably part of an affix that should be deleted. The result may be that the lemmatizer overeagerly deletes `d` in the middle of words, or that it does not remove `ed` at the end of words.

While the convolutional layer's disadvantage of a finite receptive field is important, there is justification for why a convolutional encoder in an encoder-decoder architecture could still reach similar performance to a recurrent encoder without the full receptive field. First, using multiple convolutional layers can create a receptive field that covers the entire input sequence for most words. Second, perhaps when convolutional layers are used as part of an attentional encoder-decoder architecture, the other components can compensate for a small receptive field. For example, both ConvLematus and Lematus use an attention mechanism that takes as input the entire input sequence, which effectively increases the receptive field to the size of the input sequence.

1.1 Contributions

A novel type of neural lemmatizer is introduced, called ConvLematus. ConvLematus is similar to the attentional encoder-decoder seq2seq lemmatizer used in Lematus but with the encoder replaced with convolutional layers.

The remaining contributions can be grouped by two overarching questions:

- *How do different architectures of the convolutional encoder used in* ConvLematus *affect lemmatization accuracy across different languages?* (section 7.2)

  Several variations of ConvLematus are trained for lemmatization using

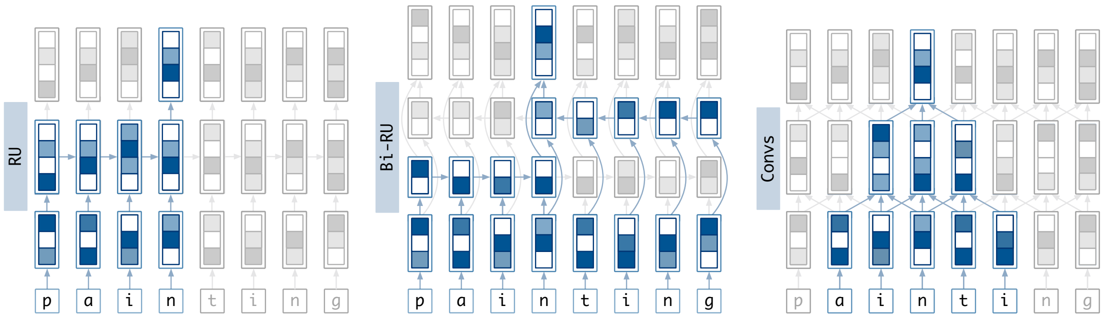


Figure 1.1: Comparison of how data flows through a recurrent unit, bidirectional recurrent unit, and two convolutional layers. Vectors are represented by the rectangles containing colored squares.
In all cases, the sequence of characters (bottom row) is first passed through an embedding layer that results in a sequence of vectors (length-3 vectors in the second-from-bottom row). Regardless of if recurrent units or convolutional layers are used, the output of the components is another sequence of vectors of the same length as the input sequence (represented by the top row of length-4 vectors.)
Arrows represent what data is combined to form the next vector. The highlighted boxes show how data traveled from the input sequence to a given element of the output sequence. **Recurrent unit (left)**, shows the flow of information of a uni-directional recurrent unit (one row of length-2 vectors, representing the hidden state size 2). Uni-directional recurrent units are used in the attentional mechanism and decoder of Lematus and ConvLematus. **Bidirectional recurrent component (center)** shows the flow of information of a bidirectional recurrent unit (two rows of length-2 vectors, again representing a hidden state size 2.) Bidirectional recurrent components are used in the encoder of Lematus. The highlighted vectors show how each output timestep of the sequence is influenced by the entire input sequence. **Convolutional layers (right)**, shows the flow of information through two convolutional layers with `kernel_size` $= 3$ (the first layer is the set of arrows connecting the embedded sequence to the intermediate representation of length-4 vectors, and the second layer is the set of arrows connecting the intermediate representation to the output sequence.) Convolutional layers are used in the encoder of ConvLematus. In this configuration, each output element has a maximum receptive field of (i.e. is influenced by) five characters.

different configurations of the convolutional encoder, focusing on the receptive field size by adjusting depth and kernel size. Models are trained for Arabic, English, Finnish, Latvian, Russian, and Turkish. We use a dataset that is newly re-purposed for lemmatization, which has a higher rate of inflected lemmas.

- *How do* Lematus *and* ConvLematus *differ in lemmatization?* (section 7.3)

  We show that ConvLematus can lemmatize nearly as well as Lematus on several languages, though none can be said to outperform it. We find that ConvLematus has a higher lemmatization and training speed than Lematus. Then we train Lematus and ConvLematus models on three additional datasets. Finally, we begin to explore possible reasons why ConvLematus is not reaching as high of accuracy as Lematus.

Chapter 2

Literature Review

This project introduces ConvLematus, an encoder-decoder seq2seq lemmatizer that uses convolutional layers in its encoder. The background section is divided into two parts. The first part focuses on the lemmatization task and the second part surveys neural network architectures that are similar to ConvLematus. Specifically, section 2.1 introduces the lemmatization task, section 2.2 describes background of the datasets used, and section 2.3 surveys recent machine learning approaches to lemmatization. Next, section 2.4 discusses encoder-decoder seq2seq networks, section 2.5 motivates convolutional layers, section 2.6 surveys neural network architectures used in NLP that use convolutions, and section 2.7 discusses architecture choices within convolutional components.

2.1 Lemmatization Task

Lemmatization falls under the morphology branch of linguistics. Morphology is the study of how words are formed. from morphemes, including lemma (e.g. the version of the word that would appear in the dictionary) and affixes (e.g. `-ed`). One way to motivate lemmatization is by considering its inverse, morphological inflection. In morphological inflection, the lemma and morpho-syntactic tags (e.g. "1st person past-tense verb") are used to produce the grammatical word, the *inflected form*. For example, to form a clause combining the number `three` and the lemma `cat`, the lemma `cat` needs to be pluralized. In this case, pluralization is done by adding the affix `-s`. This gives `three cats`. Lemmatization is converting a given inflected form, which often occurs in text, into its corresponding lemma, which is typically a form which could be looked up in a lexical resource such as a dictionary. Additional examples of English lemmatizations are given in Table 1.1.

2.1.1 Example of Lemmatization as a Component: Search Engine

Lemmatization is useful as a component of NLP programs, such as a search engine. Describing the search engine example in detail can both motivate lemmatization and illustrate the challenges of morphologically complex languages. This example search engine takes a word such as `vacation` and returns web pages about vacations. A bag-of-words approach can be applied by splitting the text of the web pages into tokens (individual words) using a process called *tokenization*. The search engine stores a list of types (unique tokens) for each web page in an index. When `vacation` is searched for, all web pages that contain `vacation` in the search engine's index are returned. Ideally, searches for `vacation` should also return results that contain the word `vacations`. This is solved with text normalization, such as lemmatization. Instead of storing the inflected form `vacations` in its index, the search engine stores its lemma `vacation`. Besides different forms of the word, it is also desirable that searches of `vacation` should return web pages containing `holiday`. Because the search engine uses lemmas, the search engine could look up `vacation` in a lexical resource to find synonyms, which can be searched for in index. In short, by using lemmas, the search engine could return results for `vacation` containing `vacation`, `vacations`, `holiday`, or `holidays`, among others.

The search engine example illustrates how lemmatization reduces *sparsity*, which is also important in other NLP tasks. Instead of the search engine needing duplicate articles that mention `vacations`, `vacation`, and `holiday`, the search engine is able to return the same document for all queries. The sparsity problem and importance of lemmatization becomes more pronounced in morphologically-rich languages. For example, while English has two inflected forms of the noun `vacation`, Finnish has 26 inflected forms of the corresponding `loma`. If text normalization such as lemmatization is used in the search engine, more web pages can be returned for each search. In the same way, reducing sparsity can aid in other tasks such as sentiment analysis by allowing the system to transfer dependencies it learned to related items (e.g. if a sentiment analyzer learns `dislike` is negative, it also knows `disliked` is negative because it is just an inflected form

person/tense	`laskea`	`lukea`
1st/present	`lasken`	<u>`luen`</u>
2nd/present	`lasket`	<u>`luet`</u>
3rd/present	`laskee`	`lukee`
1st/past	`laskin`	<u>`luin`</u>
2nd/past	`laskit`	<u>`luit`</u>
3rd/past	`laski`	`luki`

Table 2.1: Example of *consonant gradation* in Finnish. The table shows a subset of the positive inflections of Finnish's `laskea` (v. *to lower*), with no consonant gradation, and `lukea` (v. *to read*), with `-k` consonant gradation. While there does appear to be a rule for each person/tense inflectional class (e.g. remove `+ea` and add `+ee` for the 3rd person present tense), there's an additional rule of consonant gradation on `lukea` which removes the `k` in some cases (marked in bold and underlined.)

of `dislike`.)

2.1.2 Challenges in Morphologically-Rich Languages

Learning the rules and exceptions of lemmatization provides an interesting test case for machine learning. For example, the English lemmatizations in Table 1.1 show how lemmatizing past-tense words is not necessarily removing the affix `+ed`, and sometimes involves spelling changes or special cases. Even still, English is considered relatively "morphologically simple" compared to other languages. For example, Turkish has a rule called *vowel harmony* which affects non-adjacent sounds (Matthews, 1991) and Finnish, along with its large number of inflections, has a rule called *consonant gradation* illustrated in Table 2.1 that changes the spelling in certain situations (Karlsson, 2015). These morphological rules motivate one of the goals of ConvLematus: *An ideal lemmatizer should be able to learn a variety of morphological rules.* By evaluating a lemmatizer on many languages with different morphological properties, the lemmatizer's ability to learn lemmatization rules will be more apparent.

2.1.3 Lemmatization in this dissertation

More specifically, for the purposes of this dissertation, the lemmatizer will be given an inflected form as input. The lemmatizer returns a single[1] lemma that could be looked up in a lexical resource (i.e. not a stem[2].) The input may or may not include additional context.[3]

2.2 Lemmatization Datasets

As specific decisions such as whether to split compound words or what is considered the stem are deferred to the datasets used, to further cement the lemmatization task description, a brief background on the datasets used is given in the subsections below. The previous use of these datasets also places this dissertation in the context of previous work. For more details on the datasets themselves, see 5.

The neural lemmatizer will be trained on examples of either *word-lemma pairs*, pairs of the inflected form and its lemma, or *word+context-lemma pairs*, pairs of the inflected form as well as some additional context and its lemma. In this dissertation, the additional context is either the part-of-speech (POS) tag (as in the CS-POS dataset) or part of the text that the inflected form appears in (as in the UD10K-20CTX dataset). A total of four datasets are used in this dissertation. They were derived from the two data sources below:

2.2.1 CS and CS-POS

Most experiments in this dissertation are trained on CS (CoNLL–SIGMORPHON). CS and CS-POS are two lemmatization datasets extracted for this dissertation from the word-lemma-POS triplets from the high-resource datasets from CoNLL–SIGMORPHON Shared Task 2017, Task 1 (Cotterell et al., 2017). The Task 1 datasets were originally intended for a morphological inflection task in which

[1] By returning a single lemma, this leaves the some decisions about what is a lemma to the tokenizer (Jurafsky and Martin, 2014). For example, `it's` should probably be tokenized to `it` and `'s`. The compound `treehouse` should either be tokenized to `treenhouse` or `tree` and `house` depending on whether one or two lemmas are required.

[2]A possible stem of both the noun `computer` and the verb `compute` is `comput`. The lemmas should be `computer` and `compute`.

[3]Note that other papers consider context as a distinguishing feature of lemmatization as compared to stemming (Chakrabarty et al., 2017; Jivani et al., 2011). However, this dissertation considers finding the lemma of the inflected form (even with no context) as lemmatization.

models were given the lemma and POS tag and were tasked with creating the inflected form.

The CS and CS-POS datasets show how this dissertation relates to previous work. Many neural network approaches were used for the inflection task (Cotterell et al., 2017), including a convolutional encoder seq2seq model (Östling, 2016) which will be discussed section 2.6. In the experiments related to CS-POS, POS tags are included as part of the input. This differs from the lemmatization approach in Müller et al. (2015), which showed that a model learning to predict the tag and the lemma jointly can perform better than combining separate systems. This and other multi-task learning approaches are not explored in this dissertation but are interesting future directions.

2.2.2 ud10k-0ctx and ud10k-20ctx

UD10K-0CTX and UD10K-20CTX are a subset of the annotated words from the Universal Dependencies (UD) v2.1 dataset (Nivre et al., 2017b), which contains annotated text from newswire, Wikipedia articles, Europarl speeches, blog entries, and other sources. Datasets similar to UD10K-20CTX and UD10K-0CTX have been evaluated for lemmatization with LEMATUS(Bergmanis and Goldwater, 2018)[4].

Because many words in natural text are reused, the overall lemmatization accuracy can over-represent certain lemmatizations (e.g. 5.9% of the UD10K-0CTX dataset is `the` $\rightarrow$ `the`). To supplement the overall accuracy, Bergmanis and Goldwater (2018) also measured the accuracy on *unseen* words, where unseen words are the word-lemma pairs where the inflected form does not appear in the training data.

2.3 NLP approaches to Lemmatization

The previous sections described the lemmatization task, the challenges of morphologically complex languages, and the background of datasets used. This section surveys recent machine learning approaches to lemmatization, including models that use seq2seq and convolutional neural networks.

[4]Note that Bergmanis and Goldwater (2018) uses UDv2.0, while this dissertation uses UDv2.1.

One application of convolutional neural networks to lemmatization was described in Kestemont et al. (2016). It is important to note that the task in Kestemont et al. (2016) was *historical text* lemmatization, which has different priorities. To do this, the convolutional neural network uses a list of lemmas as its class labels and classifies based on a word's characters, the full word embedding, and its context. A problem with the lemma-classification approach is that the model cannot correctly lemmatize any word where lemma is the not in the class labels.

Another classification approach that does allow inflected forms with unseen lemmas to be lemmatized is to assign a class and then edit the inflected forms based on the assigned class (Chrupała et al., 2008; Müller et al., 2015). In other words, the class is not the lemma, but the rule to create the lemma from the inflected form. Systems that use this technique first extract *edit scripts* from a dataset that encodes the rules needed to convert an inflected form to its lemma. For example, a rule could encode "delete the suffix `+ing`." The model then classifies the inflected form (e.g. "For `painting`, delete the suffix `+ing`"), applies the edit script that corresponds to the class (e.g. `paint`), and returns the resulting lemma. Using this approach avoids the unseen lemma problem of directly assigning a lemma while still being able to approach lemmatization as a classification problem.

A third approach, and the one used in this dissertation, is to generate the lemma character-by-character using a neural network architecture known as seq2seq (Chakrabarty et al., 2017; Bergmanis and Goldwater, 2018). Seq2seq was introduced in chapter 1 and will be described in detail in subsection 2.4.1. When seq2seq is used for lemmatization, a sequence of characters representing a word is given to the model. The model then returns a second sequence of characters representing the lemma. For a lemmatization to be correct, the model must return the exact characters that make up the lemma. Because the model only needs to know about characters instead of words, it does not have as many out-of-vocabulary issues as approaches such as Kestemont et al. (2016).

2.3.1 Use of Context in Lemmatization

Many lemmatizers take into account the context of the word (Chakrabarty et al., 2017; Bergmanis and Goldwater, 2018; Müller et al., 2015; Chrupała et al., 2008;

Kestemont et al., 2016), which can disambiguate inflected forms. For example, `leaves` can be lemmatized as either as the noun `leaf` or the verb `leave`. Without context, the model must guess which of the two lemmas to use. For example, Bergmanis and Goldwater (2018) trained a seq2seq model using an input sequence that includes characters from the original text surrounding the word. The output sequence remains as the lemma of the target word. For example, instead of lemmatizing the inflected form alone, the input instead consists of

```
<w> t h e <lc> l e a v e s <rc> o f <s> t r e e s </w>  → <w> l e a f </w>
```

where `<lc>` and `<rc>` are elements in the sequence used to delimit the inflected form from the context. In this case, `the` is a strong clue that `leaves` is a noun. Additional information about the challenges context pose to ConvLematus are outlined in subsection 7.3.3. This dissertation focuses on using ConvLematus to learn patterns *within* words, so finding relationships *across* words (as in the case of context) is not explored thoroughly.

2.4 Neural networks for NLP

The task of lemmatization, the datasets, and other NLP approaches to lemmatization have been described. This section moves away from lemmatization and considers neural networks architectures that are related to ConvLematus. This section begins with the seq2seq architectures that ConvLematus shares most aspects with. Then it gives a brief motivation for convolutions and describes ways convolutional components have been used for NLP and in seq2seq models. This survey of architectures is summarized in Figure 2.2. Finally, it discusses a few ways convolutional components have been implemented for NLP tasks.

2.4.1 Seq2seq, Lematus, and Nematus

In seq2seq neural networks, a variable-length input sequence is passed through a network and the network returns a variable-length outputs sequence. The approach usually involves an encoder-decoder architecture, where the input sequence is first encoded by one set of components into a representation, then is decoded by another set of components into the output sequence (Kalchbrenner and Blunsom, 2013; Cho et al., 2014b; Sutskever et al., 2014; Bahdanau et al., 2015). While early approaches encoded the input sequence into a fixed-length representation (Cho et al., 2014b; Sutskever et al., 2014), a popular approach is to encode into the same length as the input sequence and then use a mechanism called attention which can use the entire encoded representation when generating each element of the output sequence (Bahdanau et al., 2015). The model introduced in this dissertation ConvLematus, as well as model used for comparison, Lematus (Bergmanis and Goldwater, 2018), both use the attentional encoder-decoder architecture Nematus (Sennrich et al., 2017), which is based on Bahdanau et al. (2015). Lematus is Nematus directly applied to the lemmatization task. ConvLematus involves modifications to the Nematus architecture to use a convolutional encoder, but otherwise uses the same configuration and task format as Bergmanis and Goldwater (2018).

What follows is a brief overview of shared aspects of Nematus, Lematus, and ConvLematus, which are illustrated in Figure 2.1. A higher-level view of this pair of architectures compared to others is illustrated in Figure 2.2. First, each input character is mapped to a vector (called the *embedding*). The parameters of the embedding are learned as part of training the network. Next, each embedded character is passed through the encoder. The key difference between Lematus/nematus and ConvLematus is the implementation of the encoder. Lematus/nematus's encoder is a two-layer bidirectional GRU (Cho et al., 2014a; Graves and Schmidhuber, 2005). The encoder of ConvLematus is a convolutional neural network. For all architectures, the encoded input sequence is then passed through the attention mechanism, a conditional GRU with attention (Sennrich et al., 2017). The result is then decoded character-by-character into the output sequence using the decoder and beam search described in Sennrich et al. (2017) with the parameters in Bergmanis and Goldwater (2018).

2.5 Motivation for Convolutional Layers

Outside of NLP, convolutional neural networks have performed well in image processing tasks (Krizhevsky et al., 2012; He et al., 2016; Szegedy et al., 2015), speech-generation (Van Den Oord et al., 2016), among others. Convolutional layers can often be used interchangeably with recurrent units on variable-length input (see Figure 1.1 for a visualization of the sequential input of recurrent units compared to the window inputs of convolutional layers). They are also relatively efficient compared to recurrent units. For example, in machine translation,

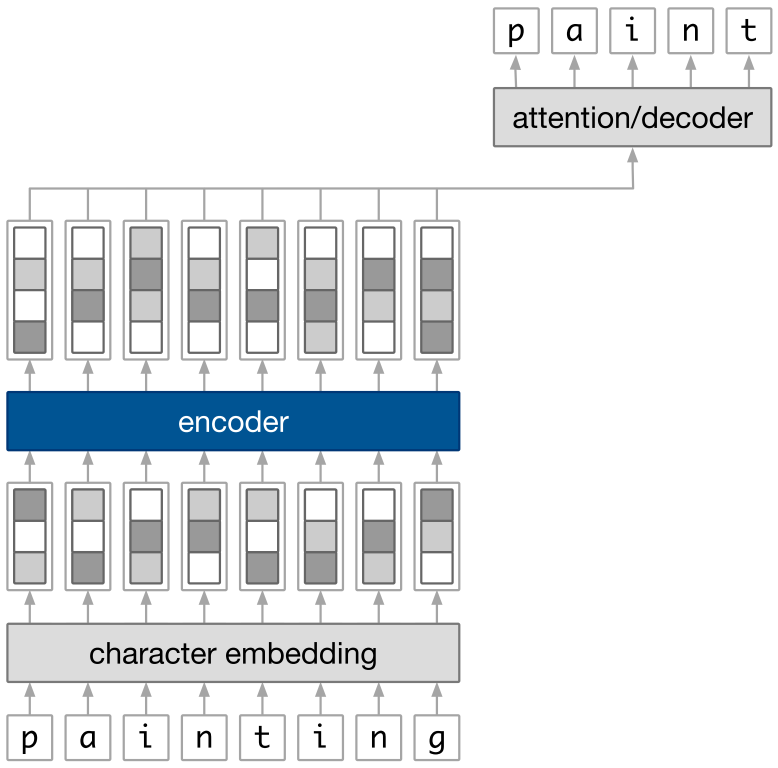


Figure 2.1: A simplified view of the Lematus/ConvLematus architectures applied to lemmatization. Beginning at the bottom of the image, each character of the inflected form `painting` passes through a character embedding which returns the embedded characters (in the image, the length-3 vectors). The embedded characters pass through an encoder, highlighted blue in the image, and the output (in the image, the length-4 vectors) are passed into attention/decoder components which then outputs the lemma `paint`. When Lematus and ConvLematus models are compared, everything about the architecture has been held constant except for the encoder component.

Gehring et al. (2017) used a convolutional seq2seq model to reach a 9x faster translation speed than a recurrent seq2seq architecture. In NLP applications the windows of convolutions can be thought of as extracting an n-gram feature from parts of the input sequence, which is a common technique in NLP (Jurafsky and Martin, 2014). Multiple layers could be thought of as building a hierarchical representation. Convolutional layers have also been used in natural language processing tasks such as machine translation (Gehring et al., 2017; Kalchbrenner et al., 2016; Lee et al., 2016; Costa-Jussà and Fonollosa, 2016), language modeling (Dauphin et al., 2016; Kalchbrenner et al., 2014; Kim et al., 2016; Bradbury et al., 2016), and sentiment analysis (dos Santos and Gatti, 2014). The motivation for using convolutional encoders as in this dissertation is elaborated on in chapter 4.

2.6 Convolutions in Seq2seq and NLP

This section compares ConvLematus to other neural networks in NLP that use convolutional layers. First, many architectures that use convolutions for NLP produce fixed-length output (unlike seq2seq's variable-length output). For this reason, subsection 2.6.1 explains a major difference between the fixed-length output architectures and seq2seq architectures like ConvLematus. The remaining subsections ignore the fixed-length/variable-length output distinction and groups architectures by how they use convolutions. The architectures can be roughly grouped into "convolutional embeddings" which use convolutions to reduce subword units into vector representations of words, "convolutional encoders" which use convolutions to encode input but do not aim to reduce the size of the input, and "convolutional seq2seq" for architectures that use convolutions in decoders as well.

2.6.1 Architectures with fixed-length output

Some architectures are similar to ConvLematus but are used for tasks that take in a variable-length input sequence and produce a *fixed-length* output vector instead of variable-length. Examples of fixed-length-output tasks[5] include

[5]To be extra clear, a task isn't fixed-length or variable-length, but its formulation is. As seen in the case of formulations of the lemmatization task (section 2.6), the "lemma as class label" formulation is fixed-length formulation, the "edit script as class label" may be considered a hybrid formulation (the classifier is fixed-length output, but the full system produces a variable-length sequence), and the "seq2seq" approach is variable-length formulation.

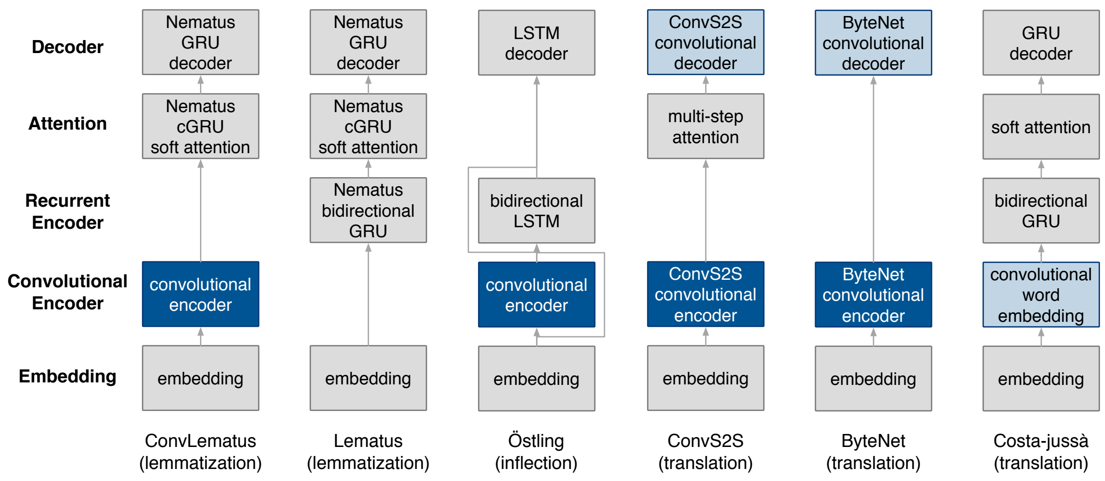


Figure 2.2: A simplified comparison of several seq2seq architectures to highlight some relevant architecture differences from ConvLematus. Models included are ConvLematus (this dissertation), Lematus (Bergmanis and Goldwater, 2018), Östling (Östling, 2016), ConvS2S (Gehring et al., 2017), ByteNet (Kalchbrenner et al., 2016), and Costa-jussà (Costa-Jussà and Fonollosa, 2016). Convolutional encoders highlighted in dark blue, and other convolutional components are highlighted in light blue. Arrows show how data flows, such as Östling's optional bypassing of either the recurrent or convolutional encoder, though some arrows are omitted. Note that with the exception of ConvLematus and Lematus, most systems use different code and likely have additional differences.

language modeling (given a sequence, returns the distribution over next possible words) or classification (given a sequence, return a class). Convolutional layers have been used for the fixed-length output tasks of sentiment analysis (dos Santos and Gatti, 2014) and other text classification (Zhang et al., 2015; Kim, 2014), and language modeling (Kalchbrenner et al., 2014; Kim et al., 2016; Dauphin et al., 2016). One main difference between these architectures and seq2seq models like ConvLematus is that they tend to use a component that could be called *max-pooling over the sequence* to convert the variable-length input sequence into a fixed-length vector. In *max-pooling over the sequence*, for each element in a vector, it selects the max value across all elements of the sequence and returns a fixed-length output of the size of an individual vector. Otherwise, the architectures that produce a fixed-length output make use of convolutional embeddings and encoders similar to ConvLematus and will be discussed further below.

2.6.2 Convolutional embeddings

Convolutions have been used to build vector representations of words from characters or other sub-word parts which are used as input into the rest of the network (Costa-Jussà and Fonollosa, 2016; Kim et al., 2016; Lee et al., 2016; Vania and Lopez, 2017). To draw a distinction, this use of convolutions could be referred to as *convolutional embeddings* as opposed to *convolutional encoders.* While the distinction can be blurry, a convolutional embedding's purpose is to reduce the size of the input by combining character embeddings into a word or other multi-character embedding. A convolutional encoder does not need to reduce dimensions.

Two examples of convolutional embeddings used for seq2seq learning are Costa-jussà (Costa-Jussà and Fonollosa, 2016) and Lee (Lee et al., 2016). Both Costa-jussà and Lee did machine translation by applying convolutional layers with multiple kernel sizes[6] to sequences of characters (Kim et al., 2016). Both also feed the resulting convolutional embedding into a recurrent attentional encoder-decoder network. Where the architectures differ is how they handle words. Costa-jussà first split the input into words, then applied the convolutions, and finally performed *max-pooling over the sequence* to produce a fixed-length representation of each word. Lee differs in that it did not split into words.

[6]One way to view this approach is to compute the convolutional layer output for multiple kernel sizes, and then concatenating the output along the dimension of the vector elements (the filters). See section B.3 for further description of this approach and experiments.

Lee instead applies the convolutions to the entire sequence. Then the input is split into windows of size k, and *max-pooling over the sequence* is applied to each of those windows.

ConvLematus is closely related to these models as it also uses a convolutional component between the embedding and recurrent components (attention and decoder). Figure 2.2 illustrates the relationship between ConvLematus and Costa-jussà (though Lee is interchangeable with Costa-jussà in this high-level view.) ConvLematus is more similar to Lee than Costa-jussà because ConvLematus and Lee do not tokenize the input sequence (in the case of ConvLematus the input sequence is already characters of a word) and the convolutional components produce a variable-length output. However ConvLematus does not use max-pooling while Lee does. There are also differences in the attentional encoder-decoder network implementation, such as ConvLematus's cGRU soft attention inherited from Nematus.

2.6.3 Convolutional encoders

Like convolutional embeddings, architectures that use convolutional encoders use convolutional layers between character or sub-word unit embeddings and additional recurrent components. However, unlike convolutional embeddings, convolutional encoders do not aim to reduce the length of the input sequence. Using this definition, ConvLematus, Östling (Östling, 2016), and C-LSTM (Zhou et al., 2015) use a convolutional encoder. One particularly similar model, Östling, uses a character-level seq2seq model for morphological inflection that adds an optional convolutional encoder between the embedding layer and an LSTM encoder. Unlike the ConvLematus architecture, Östling adds connections that can bi-pass the convolutional component (see Figure 2.2), uses a bidirectional recurrent unit, and does not use cGRU soft attention.

2.6.4 Convolutional seq2seq and others

Convolutional seq2seq architectures aim to replace all recurrent units with convolutions (Gehring et al., 2017; Kalchbrenner et al., 2016; Östling, 2016). The lack of recurrent units and the addition of convolutional decoders is highlighted in Figure 2.2. While the success of these provides motivation for using convolutions, ConvLematus uses a different architecture. In a very different approach, Xingjian et al. (2015) modifies the recurrent unit itself by replacing the dense layers between the timesteps of recurrent units with convolutions. Both convolutional seq2seq and other approaches show other directions convolutions usage in seq2seq models could go.

2.7 Convolutional Components Architecture

The previous section gave an overview of ways the convolutional component can be placed in a neural network's architecture, including the approach used in this dissertation, the convolutional encoder. This section explores literature related to specific implementations of the convolutional encoder.

In this dissertation, a survey of kernel size and number of convolutional layers is carried out in section 7.2. Kernel sizes can be seen as a way to detect features of a certain size (Szegedy et al., 2015), such as morphemes in the case of lemmatization. To do this, some approaches in NLP use a single convolutional layer that concatenates the output of convolutions that use kernel sizes of different length (Kim, 2014; Kim et al., 2016; Lee et al., 2016; Costa-Jussà and Fonollosa, 2016). Other papers use a single kernel size, (dos Santos and Gatti, 2014; Gehring et al., 2017) or if using multiple layers, use a different kernel size depending on layer (Zhang et al., 2015). Testing different depths of convolutional layers was done by (Zhang et al., 2015).

Several features are frequently used with convolutional components, but due to time-constraints were not explored thoroughly in this dissertation. The preliminary results are given in Appendix B. Convolutions with varying-sized kernels (Kim et al., 2016) benefit from using multiple kernel sizes to detect affixes of different sizes. Highway layers (Srivastava et al., 2015) are often used to combine convolutional output filters. Batch normalization (Ioffe and Szegedy, 2015) is used to help train deeper convolutional networks.

Other approaches were not explored in this dissertation but provide an interesting avenue of future exploration. Several convolutional architectures for NLP use a very large number of convolutional layers Conneau et al. (2016); Östling (2016); Gehring et al. (2017); Kalchbrenner et al. (2016), possibly inspired by the success in image processing (He et al., 2016). Methods of pooling can also be used to improve performance (Zhang et al., 2015), and different methods such as k-max (Kalchbrenner et al., 2014; Denil et al., 2014) or parameter-free recurrent pooling (Bradbury et al., 2016) show an interesting direction.

Chapter 3

Theory

This chapter outlines the theory of the differences between convolutional and recurrent encoders as well as how the convolutional layer's finite receptive field depends on the convolutional layers' parameters.

For brevity, this chapter will focus on the equations necessary to understand the convolutional encoder, as that is the focus of this dissertation, and treat the other components as black boxes. For more details about the rest of the specific architecture, see Sennrich et al. (2017) or for more general information about neural networks for NLP, see Goldberg (2016).

3.1 Encoder-decoder seq2seq architecture

This section will give a high-level overview of the formulation of seq2seq lemmatization, and highlight what the encoder expects as input and output.

3.1.1 Inputted, predicted, and target sequence

In this dissertation's experiments, lemmatization is framed as a sequence-to-sequence problem in the same way as Lematus. The model is provided an inflected form and optional context characters $X = \big(x_1, ..., x_{\texttt{length}(X)}\big)$, where x_i is a character. The network generates the predicted lemma $Y = \big(y_1, ..., y_{\texttt{length}(Y)}\big)$. The prediction Y is compared to the true lemma $T = \big(t_1, ..., t_{\texttt{length}(T)}\big)$.

The following example gives inflection X of `leaves`, target T of `leave`, and the incorrect prediction Y of `leaf`.

$$X = \texttt{l e a v e s}$$
$$Y = \texttt{l e a f}$$
$$T = \texttt{l e a v e}$$

The rest of this chapter uses *timestep* in place of *character* for units along the length of the word. This is because convolutions are sometimes applied to intermediate representations that don't correspond to characters. For example, in Figure 1.1, an intermediate representation (highlighted vector in the top row) could represent the entire receptive field (highlighted bottom row).

3.1.2 Encoder inputs and outputs

In seq2seq models like LEMATUS, the encoder receives as inputs the embedded characters and passes its output into the attention mechanism. The embedding layer maps each character of the input sequence to a corresponding character embedding vector $\mathbf{c}_t \in \mathbb{R}^{\texttt{embedding\_dims}}$. The character embeddings' values are learned during training. The output of the embedding layer is a $\texttt{length}(X)$-sequence of vectors called the embedded sequence $C = \big(\mathbf{c}_1, ..., \mathbf{c}_{\texttt{length}(X)}\big)$. C is passed into the encoder component. The encoder component of all architectures explored in this dissertation return a sequence of vectors of the same length as the input sequence $\texttt{length}(X)$[1]. The output of the encoder is the encoded sequence $E = \big(\mathbf{e}_1, ..., \mathbf{e}_{\texttt{length}(X)}\big)$. E is passed into an attention mechanism. The attended output $A = \big(\mathbf{a}_1, ..., \mathbf{a}_{\texttt{length}(Y)}\big)$ is computed such that each $\mathbf{a}_t$ uses the entire encoded sequence E. Specifically, CONVLEMATUS and LEMATUS uses a conditional GRU with attention (Sennrich et al., 2017) of the form

$$\mathbf{a}_t = cGRU_{att}(\mathbf{a}_{t-1}, \mathbf{y}_{t-1}, E)$$

The consequence of pairing a convolutional encoder with this attention mechanism is that the receptive field of both together could theoretically be the full input sequence X.

[1]Note that it isn't necessary in general for the encoded output to be the same size as the encoder input. For example, in the case of convolutional embeddings explained in subsection 2.6.2, the output sequence of the encoder layer is shorter than the input sequence.

3.1.2.1 Bidirectional Recurrent Encoder

A bidirectional recurrent encoder consisting of layers of bidirectional recurrent units is used in Lematus. A bidirectional recurrent unit such as the one shown in Figure 1.1 is comprised of two recurrent units. At each timestep, the recurrent unit takes in an element of the input sequence $\mathbf{x}'_t$ and outputs a vector $\mathbf{e}'_t$ and a hidden state vector $\mathbf{h}'_t$ of size `hidden_state_size`. Let the recurrent unit's underlying equations be $\mathbf{RU}$. Each output and hidden state at timestep t is given by combining the output of a recurrent unit moving over the string forward

$$\left(\overrightarrow{\mathbf{e}'_t}, \overrightarrow{\mathbf{h}_t}\right) = \mathbf{RU}(\mathbf{x}_t, \overrightarrow{\mathbf{h}_{t-i}})$$

and one moving backward

$$\left(\overleftarrow{\mathbf{e}'_t}, \overleftarrow{\mathbf{h}_t}\right) = \mathbf{RU}\left(\mathbf{x}_t, \overleftarrow{\mathbf{h}_{t+i}}\right).$$

Specifically, the output at timestep t is the concatenation of the vectors

$$\mathbf{e}_t = \left[\overrightarrow{\mathbf{e}'_t}; \overleftarrow{\mathbf{e}'_t}\right].$$

The resulting vector is $\mathbf{e}_t \in \mathbb{R}^{2\cdot\texttt{hidden\_state\_size}}$. The result of a bidirectional recurrent unit $E = \left(\mathbf{e}_1, ..., \mathbf{e}_{\texttt{length}(X)}\right)$ can be used as input for another bidirectional recurrent unit, which would return another sequence of vectors of the same size (assuming `hidden_state_size` is the same). The encoded sequence E from the last bidirectional recurrent unit is the input into the attention component.

3.1.3 Recurrent and Convolutional Encoders

This dissertation tries to treat the recurrent encoder as interchangeable with a convolutional encoder while holding the rest of the architecture constant. For this reason, the convolutional encoder will take in $C = \left(\mathbf{c}_1, ..., \mathbf{c}_{\texttt{length}(X)}\right)$ like the recurrent encoder. The output of Lematus's bidirectional recurrent encoders are vectors of size $2 \cdot$ `hidden_state_size`. Note that `hidden_state_size` is used in other parts of the Lematus/ConvLematus architecture, such as the decoder recurrent unit's state size. For a convolutional encoder to be interchangeable with a recurrent encoder, the last layer of the convolutional encoder should output `length`(X) vectors of size $2 \cdot$ `hidden_state_size` (see section 3.2).

3.2 Convolutional Encoder in seq2seq

The previous section gave a high-level description of a seq2seq network, including how a convolutional encoder could be used in place of a recurrent encoder while keeping the rest of the network structure identical. This section will explain the internal structure of a convolutional encoder.

A temporal convolution, or 1-dimension convolution, can be thought of as sliding a window of length `kernel_size` along a 1-dimensional input, such as text data. A convolutional layer will compute $Y' = (X' * W)$, where Y' is the output, X' is the input, W are learned parameters, and $*$ is the 1-dimensional convolution operator.[2] The input into a convolutional layer is $X' \in \mathbb{R}^{\texttt{length}(X') \times \texttt{dims}(X')}$. For example, if a convolution is applied after embedding each character, if $C = \left(\mathbf{c}_1, ..., \mathbf{c}_{\texttt{length}(X)}\right)$ and $\mathbf{c}_i$ are column vectors, then

$$X' = \left[\mathbf{x}_1 \cdots \mathbf{x}_{\texttt{length}(X)}\right]^\top = \left[\mathbf{c}_1 \cdots \mathbf{c}_{\texttt{length}(X)}\right]^\top .$$

Temporal convolutions are parameterized on the integers `kernel_size`, the size of the window, and `filter_count`, the number of elements in each output vector.[3] Each convolutional layer learns parameters W including the weights

$$W' \in \mathbb{R}^{\texttt{dims}(X') \times \texttt{kernel\_size} \times \texttt{filter\_count}}$$

and the bias $b \in \mathbb{R}^{\texttt{kernel\_size} \times \texttt{filter\_count}}$. To compute an output at timestep t, $y'_t \in \mathbb{R}^{\texttt{filter\_count}}$, the basic form of convolution gives

$$y'_t = (X' * W)_t = b + \sum_{d=1}^{\texttt{dims}(X')} \sum_{t'=1}^{\texttt{length}(X')} (X'_{window} \otimes W')$$

where $\otimes$ is element-wise multiplication and $X'_{window} \in \mathbb{R}^{\texttt{kernel\_size} \times \texttt{dims}(X')}$ is a subset of consecutive timesteps in X' given by

$$X'_{window} = \left[\mathbf{x}'_{\left(t - \left\lfloor \frac{\texttt{kernel\_size}}{2} \right\rfloor\right)} \quad \cdots \quad \mathbf{x}'_{\left(t + \left\lfloor \frac{\texttt{kernel\_size}-1}{2} \right\rfloor\right)}\right] .$$

The output of the convolutional encoder should be $\texttt{length}(X)$, so X'_{window} is padded, or vectors of zeros are added as needed.[4] The result of a convolutional

[2]Technically the implementation used in the ConvLematus models does cross-correlation.

[3]The terminology "filter" comes from image processing literature.

[4]For example to compute the output at $t = 1$ with a `kernel_size` of size 5, the window is $X'_{window} = \left[\vec{0} \quad \vec{0} \quad \mathbf{x}'_1 \quad \mathbf{x}'_2 \quad \mathbf{x}'_3\right]$.

layer $Y' = (X' * W)$ gives an output $Y' = \left(\mathbf{y}'_1, ..., \mathbf{y}'_{\texttt{length}(X)}\right)$ with the same size as the input sequence. The output of one convolutional layer Y' can be used as input in another convolutional layer $Y'' = (Y' * W)$. The output of the convolutional encoder is $E = \left(\mathbf{y}_1, ..., \mathbf{y}_{\texttt{length}(X)}\right)$, where $\mathbf{y}$ is the output of the final convolutional layer.

3.2.1 Convolutional layers and the receptive field size

Using multiple convolutional layers has the effect of increasing the receptive field (as illustrated in Figure 1.1). If there are n layers with `kernel_sizes` $(k_1, ..., k_i, ..., k_n)$, then the size of the receptive field is given by

$$\texttt{receptive\_field} = 1 + \sum_{i=1}^{n} (k_i - 1).$$

In this dissertation, all convolutional layers use the same kernel size k. In that case, the equation simplifies to

$$\texttt{receptive\_field} = n(k - 1) + 1. \tag{3.1}$$

Chapter 4

ConvLematus

ConvLematus is a seq2seq model that uses a convolutional encoder (i.e. in Figure 2.1, the "Encoder" of ConvLematus uses convolutional layers.) This chapter outlines the goals for ConvLematus for this thesis. It then discusses the differences between convolutional encoders and recurrent encoders to motivate the experiments in the rest of this dissertation. Finally, it lists the detailed set of questions that will be explored in the rest of the paper.

4.1 Principles for ConvLematus

There are many architecture design choices that can be made. To focus the experiments carried out in this dissertation, the following principles were defined.

- *The impact of using a convolutional encoder in place of a recurrent encoder should be apparent.* Specifically, ConvLematus should use a convolutional encoder instead of the bidirectional recurrent encoder and otherwise be identical to Lematus. This choice means that unlike many other convolutional embedding and encoder models (see section 2.6), no bidirectional recurrent unit is used in ConvLematus. This allows us to see whether a convolutional encoder can replace a bidirectional recurrent unit.

- *Simpler models should be preferred.* Using more complex activation functions, highway layers, or connections between convolutional layers could improve accuracy. To simplify the narrative flow, only two parameters are explored in the main part of this dissertation: the kernel sizes and the number of convolutional layers. Initial explorations of complex architectures are described in Appendix B.

- *An ideal lemmatizer should be able to learn a variety of morphological rules.* This translates to accurately lemmatizing unseen inflected forms (inflected forms not in the training set) in a variety of languages. A second motivation for a language-agnostic architecture is that LEMATUS has been shown to perform well on unseen lemmas in a variety of languages (Bergmanis and Goldwater, 2018).

- *(Optional)* A convolutional encoder could lemmatize at state-of-the-art accuracies. However, due to the first two principles, high accuracy is not expected. Instead, we aim to explore how convolutional encoders lemmatizes differently than LEMATUS.

- *(Optional)* A lemmatizer could learn additional language rules to help disambiguate inflected forms. For example, lemmatizers can learn to use part-of-speech tag or context to help lemmatize.

4.2 Convolutional vs Recurrent Encoders

This section motivates why we might use convolutional encoders. It first motivates why recurrent units and convolutional layers work well for seq2seq architectures. Then, because the only difference between CONVLEMATUS and LEMATUS architectures are the encoders, we can use this section to predict what differences we might see between CONVLEMATUS and LEMATUS.

4.2.1 Variable-length input and features

Both recurrent units and convolutional layers work well for variable-length input because their learned parameters are independent of the length of the sequence. For example, convolution's parameters depend on their kernel and filter size (see chapter 3). This independence of learned parameters from the input-length makes both convolutional layers and recurrent units work well as encoders in seq2seq. A second advantage is what convolutional layers and recurrent units learn about input. For example, convolutional layers have *translational invariance*, so with kernel length 3, they can learn to recognize a feature like `-ing` at any point in the sequence (i.e. starting at position 5 or position 18). Recurrent units can also recognize features at any point in the sequence. In these two ways, the size of parameters learned and what is learned, make both convolutional layers and recurrent units useful for seq2seq encoders.

4.2.2 Convolutional layers are faster

One advantage of convolutional layers over recurrent units is that in theory, a convolutional layer can process data faster than a recurrent unit. This is because a convolution layer can process the entire input sequence in parallel. The recurrent unit must process one timestep at a time because it depends on the output from the previous timestep (see subsubsection 3.1.2.1). Indeed, some convolutional seq2seq approaches have reported a magnitude faster prediction (Gehring et al., 2017).

A note, however, is that training speed is not synonymous with the number of parameters. Convolutional encoders don't necessarily result in a smaller number of parameters. Instead, a convolutional encoder model can make predictions faster than a recurrent-encoder model with fewer parameters. The number of parameters may be a problem for space-constrained applications such as models on mobile devices.

4.2.3 Recurrent units can have a larger receptive field size

One important difference between convolutional layers and recurrent units is the receptive field. To describe it in terms of ConvLematus, in the architecture outlined in section 3.1, the convolutional encoder will receive the embedded vectors $C = \big(\mathbf{c}_1, ..., \mathbf{c}_{\texttt{length}(X)}\big)$. The encoder will apply one or more convolutional layer and output the encoded vectors $E = \big(\mathbf{e}_1, ..., \mathbf{e}_{\texttt{length}(X)}\big)$. Each encoded vector $\mathbf{e}_t$ will be influenced by a certain number of embedded vectors $\mathbf{c}$. The maximum of this number is the *receptive field*. An equation for the receptive field of multiple convolutional layers is given in Equation 3.1. A convolutional layer has a fixed-length receptive field which may be smaller than the length of the string. This means that the convolutional encoder cannot learn relationships between input characters and output characters that are farther away than the size of the receptive field.

A recurrent encoder that uses a bidirectional recurrent unit has a receptive field of the length of the input string, as illustrated in Figure 1.1. This is a theoretical advantage over convolutional layers, as every relationship of input and output characters can potentially be learned.

While the convolutional encoder's fixed-sized receptive field is a theoretical disadvantage, it may not matter when used in ConvLematus. First, ConvLematus is a complex architecture. In particular, the attention mechanism and recurrent decoder may compensate for the encoder's limited receptive field. In addition, the use of multiple convolutional layers or larger kernel sizes may be able to increase the fixed-sized receptive field enough for lemmatization. These are now elaborated.

ConvLematus feeds the encoder output into an attention mechanism and recurrent decoders (labeled as "attention/decoder" in Figure 2.1). The attention mechanism takes into account the entire encoded inputs E (see subsection 3.1.2 for the formula). This theoretically means the output of the attention mechanism in ConvLematus has a receptive field of the length of the input word. Besides the attention mechanism, the recurrent units in the conditional GRU attention mechanism and decoder may help too. However, it is important to note that these recurrent units are not bidirectional so can only use earlier encoded/attended vectors to influence later decoded vectors (see "recurrent unit" in Figure 1.1). Given this, the attention mechanism and recurrent units in ConvLematus give it the ability to learn relationships between each element of the output sequence and every element of the input sequence.

The second way that the fixed-size receptive field may not matter is that the receptive field can be increased by using multiple convolutional layers or increasing the kernel size. Since words tend to be short[1], using the formula in Equation 3.1, one could find the number of layers needed increase the convolutional encoder's receptive field to larger than the maximum word length.[2]

[1]Lematus trims words longer than 75 characters.

[2]A theoretical drawback of deeper networks is that each layer must be computed sequentially, which can slow down the network or make it more difficult to train. In the literature, dilated convolutions have been used to efficiently increase the receptive field size to the length of the word without many layers (Van Den Oord et al., 2016; Kalchbrenner et al., 2016). Skip connections have been used to help train deeper networks (He et al., 2016). In the models tested in this dissertation (subsection 7.3.2), the decrease in speed when using more convolutional layers in the convolutional encoder was small compared to the impact of not using a recurrent encoder.

4.2.4 Convolutional layers could represent hierarchy

Another reason to consider convolutions are their intuitive interpretation within morphology. Convolutions can be viewed as similar to learning about n-grams, such as n contiguous characters in a word. n-grams were historically widely used in NLP. In addition, using multiple convolutional layers may introduce a hierarchy. The first layer may detect affixes such as `ar` and `na` in `bilarna`. Later layers may detect combinations of affixes such as `arna`.

4.2.5 Additional difference

Besides the receptive field, there are additional differences between using multiple convolutional layers and using a recurrent unit. For example, the gating mechanism used in recurrent units compared to the ReLU used between convolutional layers may impact the types of relationships learned. While the receptive field difference is the one used to guide the choice of experiments, these other differences may impact the results we see. In other words, if the networks perform the same, we may be able to say “convolutional layer’s limited receptive field is compensated for.” If they aren’t the same, we can’t say “the limited receptive field caused this difference.”

Chapter 5

Datasets

This chapter describes the languages used, the primary dataset (CS) that models were trained on, and the additional datasets (CS-POS, UD10K-0CTX, and UD10K-20CTX) for the ConvLematus vs Lematus set of experiments (subsection 7.3.1).

5.1 Languages

From the list of goals in section 4.1, *an ideal lemmatizer should be able to learn a variety of morphological rules.* To explore whether ConvLematus could learn inflectional rules for a variety of languages, models were trained on six languages which span language groups: Arabic, English, Finnish, Latvian, Russian, and Turkish. These range from having few inflected forms per lemma (English), to having a large number of inflected forms with spelling changes (Finnish, also Arabic, Latvian, and Russian), to adding many affixes without changing spelling (Turkish). A caveat is that the lemmatizer can only learn the rules with examples present in the dataset. Some datasets may be skewed towards certain rules. For example, the CS English dataset only contains English verbs. For this reason, claims in this dissertation that the lemmatizer performs poorly on a certain language in the CS dataset does not necessarily generalize to that language in general. Examples from each dataset are shown in Table 5.1.

5.2 Data input format

For all experiments, the format of the datasets follows Bergmanis and Goldwater (2018). Both ConvLematus and Lematus determine what the elements of the sequence are as the text separated by whitespace. All datasets used in this dissertation represent the inflected form and lemma as a sequence of characters. Spaces are replaced with `<s>`. In addition, `<w>` and `</w>` are added to the beginning and end of the string to match Bergmanis and Goldwater (2018). This means that for the example `defining` → `define`, a line in the source file represents the inflected form as

```
<w> d e f i n i n g </w>
```

and the line in the target file is given by

```
<w> d e f i n e </w>
```

This general format is used for all datasets that follow.

5.3 CS and CS-POS

This section will introduce the CS and CS-POS, two datasets derived from CoNLL–SIGMORPHON 2017 datasets. It will describe the original intention of the source dataset (morphological inflection), a few caveats that result from this, and the motivation for creating these datasets. It will then describe how the individual datasets were generated.

The CS and CS-POS datasets were created for this dissertation using the high resource datasets for each used language from CoNLL–SIGMORPHON 2017 shared task 1[1] (Cotterell et al., 2017). The dataset consists of mostly distinct inflected form, and was intended to have few inflected forms for each lemma. An example from the original dataset are the lemma, inflected form, and morphological tags:

```
pearl  pearling  V;V.PTCP;PRS
```

The original task for the dataset was to convert the lemma (`pearl`) and its tags (`V;V.PTCP;PRS`) into its inflected form (`pearling`).

[1]`https://github.com/sigmorphon/conll2017/`

CS

inflected forms	lemmas
`<w> r e c a r b o n i z e d </w>`	`<w> r e c a r b o n i z e </w>`
`<w> s t e v v o n e d </w>`	`<w> s t e v v o n </w>`
`<w> o u t e a t e n </w>`	`<w> o u t e a t </w>`

CS-POS

inflected forms	lemmas
`<w> 0 <pos> r e c a r b o n i z e d </w>`	`<w> r e c a r b o n i z e </w>`
`<w> 0 <pos> o u t e a t e n </w>`	`<w> o u t e a t </w>`
*`<w> 1 <pos> k u t r i s s a </w>`	`<w> k u t r i </w>`

UD10K-0CTX

inflected forms	lemmas
`<w> C o l u m b i a </w>`	`<w> C o l u m b i a </w>`
`<w> e v e n </w>`	`<w> e v e n </w>`
`<w> w e r e </w>`	`<w> b e </w>`

UD10K-20CTX

inflected form
`<w> a n d <s> d e m o n s t r a t i o n s <s>` `<lc> w e r e <rc>` `<s> p r o v o k e d <s> b y <s> t h e <s> U S </w>`
lemma
`<w> b e </w>`

Table 5.1: Examples from each dataset used. Examples from each of the four English datasets (except for * which is from the Finnish dataset) showing the words and format. Note that the UD10K-20CTX inflected form is a single example broken across multiple lines for formatting reasons.

5.3.1 Caveats

There are some caveats about this dataset due to the original task being to convert a lemma into its inflected form. One caveat is that the inflected form may contain multiple words. For example, one pair from Finnish is the multi-word `et liene nylkenyt` → `nylke`. This is different than the original definition of lemmatization given in subsection 2.1.3. In addition, some inflected forms may be difficult to lemmatize correctly without the POS tag: one example from the English validation set that is unseen in the training set is `sproing` → `sproing`, and without additional information, `sproing` → `spro` is also a reasonable lemmatization.

5.3.2 Motivation

The reason these new datasets were created because the alternative, the UD10K datasets, contain a large percentage of duplicate and un-inflected entries. A goal is to evaluate CONVLEMATUS model's ability to learn spelling-changes, and the duplicates and string-copying in UD10K suggest there are fewer spelling-change rules to learn. The CS and CS-POS datasets contain a larger number distinct inflected forms compared to natural text. For example, Table 5.2 shows that the nearly all inflected forms in the CS training set are distinct, unlike the UD10K-0CTX training set has more than 40% duplicated inflected forms in all languages. That said, experiments are also ran on UD10K for comparing LEMATUS and CONVLEMATUS (see section 5.4).

5.3.3 Generating the datasets

We use the same splits as the original CoNLL–SIGMORPHON dataset. The training set contains 10k word-lemma pairs, the validation set contains 1k word-lemma pairs, and the test set contains ≈ 1k word-lemma pairs.

5.3.4 Generating CS

The CS dataset was created from the CoNLL–SIGMORPHON dataset using the inflected form as the source sequence and lemma as the target sequence. Examples of the input and output of this dataset are shown in the CS section of Table 5.1.

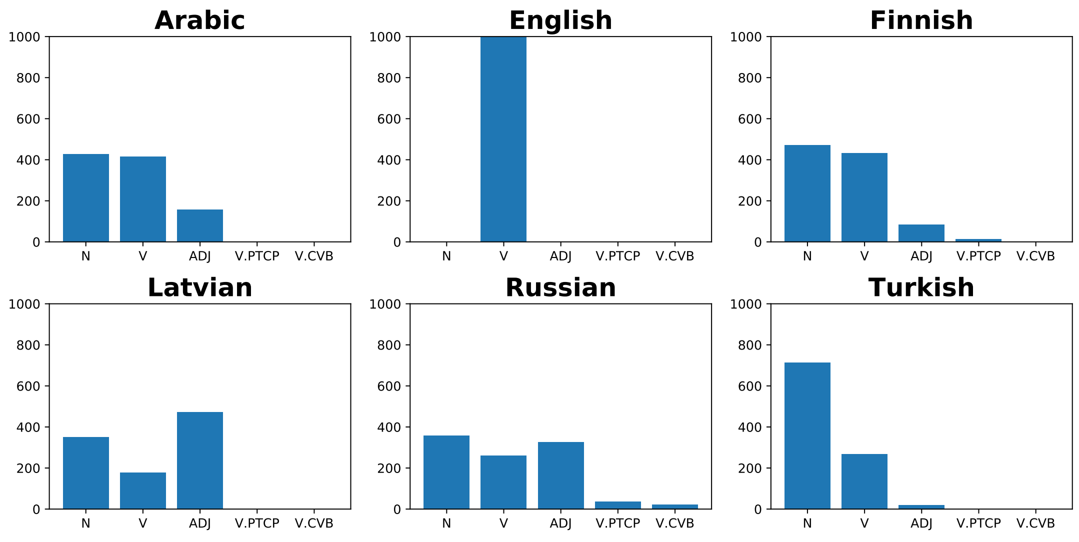


Figure 5.1: Distributions of part-of-speech tags in the CS-POS dataset. The tags are nouns (N), verbs (V), adjectives (ADJ), past-tense verbs (V.PCTP)

5.3.5 Generating CS-POS

The CS-POS dataset is identical to CS, but also adds parts-of-speech (POS) tags from the original dataset to the input sequence. A sample facilitates explaining the format: If an example from SIGMORPHON has a tag of `V.PTCP;ACT;PRS`, the first component is extracted as `V.PTCP`[2]. All distinct tags are mapped to a single digit per language, so `V.PTCP` may map to `2`. This value is prepended to the inflected form with the delimiter `<pos>`. A complete example is shown in the CS-POS section of Table 5.1. The distributions of POS-tags used by language are shown in Figure 5.1. Notably, the English dataset only contains the tag `V`.

5.4 ud10k-0ctx and ud10k-20ctx

UD10K-0CTX and UD10K-20CTX are Universal Dependencies (UD) v2.1-based datasets.[3] The process for creating UD10K-0CTX and UD10K-20CTX follow Bergmanis and Goldwater (2018)'s 10k datasets. While the paper contains the full description, a sketch of the process follows.

[2]The POS tag depends on the annotation of the CoNLL dataset. For example, an example in English might use `V;V.PTCP;PRS`, so the CS-POS uses the tag `V`. Finnish might use `V.PTCP;ACT;PRS`, which would produce the tag `V.PTCP`.

[3]`http://universaldependencies.org`

In general, the datasets are based on the UDv2.1 training dataset (Nivre et al., 2017b) (note that Bergmanis and Goldwater (2018) uses UDv2.0), which contains data for the languages in section 5.1 from annotated newswire, Wikipedia articles, Europarl speeches, blog entries, and other such sources. The words are given in the context of their complete sentence. For example, a sentence may be

```
The battles and demonstrations were provoked by the US assault on Fallujah.
```

and this is followed by annotations of each word, such as

```
4  demonstrations   demonstration   NOUN  ...
5  were             be              AUX   ...
6  provoked         provoke         VERB  ...
```

which has the second column as the inflected form and the third column as the lemma.

For both UD10K-0CTX and UD10K-20CTX, the training set is based on the first 10k words[4] from the corresponding language's UDv2.1 training set and the validation set are the first 3k examples from the UDv2.1 validation set. We did not use the testing set for these datasets. The specific UD datasets used are Arabic (PADT), English (EWT), Finnish (TDT), Latvian (LVTB), Russian (GSD), and Turkish (IMST).

5.4.1 Generating ud10k-0ctx

The inflected form and lemma are extracted from each line and the transformation described in section 5.2 is applied. See the UD10K-0CTX section of Table 5.1 for an example.

5.4.2 Generating ud10k-20ctx

The UD10K-20CTX is the same as UD10K-0CTX, but the source sequence also includes the up-to-20-character context from the original sentence on both the left and right of the word. For example, using the original sentence

The battles $\underline{\text{and demonstrations }\textbf{were}\text{ provoked by the US}}$ assault on Fallujah.

the 20-character context of `were` are underlined. This will appear as a source sequence of the UD10K-20CTX section of Table 5.1. These are delimited from the inflected form by the elements `<lc>` (left context) and `<rc>` (right context).

[4]Some "words" that are numbers and symbols are skipped.

5.5 Comparison of datasets

5.5.1 CS for exploring convolutional encoders

The first question explored is *How do different architectures of the convolutional encoder used in* ConvLematus *affect lemmatization accuracy across different languages?* Specifically:

- *Does the size of the receptive field contribute to the network's accuracy?* (subsection 7.2.1)
- *Does convolutional layer depth matter, or do the extra layers just increase the receptive field?* (subsection 7.2.2)
- *Which* ConvLematus *architecture performs the best across languages?* (subsection 7.2.3)

As these questions are explored, one important goal in chapter 4 was *An ideal lemmatizer should be able to learn a variety of morphological rules.* For this reason, the CS dataset is primarily used. To justify this choice, there are a few approximate measures of the presence of morphological rules, including the number of distinct inflected forms, the percentage of string-copying, and the percentage of unseen words.

First, because of the original dataset's purpose, the CS dataset has a much higher number of distinct inflected form, which probably correlates to more examples of rules. The other dataset explored, the ud10k-0ctx and ud10k-20ctx datasets, use words at the frequency they appear in text. This leads to, for example, 5.9% of the training dataset in English is the inflected form `the`. The number of distinct inflected forms is shown in Table 5.2. The ud10k datasets tend to contain many duplicates, which suggests there are fewer rules present. Second, the CS dataset has a lower rate of string-copying[5] than ud10k for each language. The values are also shown in Table 5.2. For example, to use the example from above again, those 5.9% occurrences of `the` lemmatize as `the` $\rightarrow$ `the`. Third, CS has a higher percentage of unseen words, which is also shown in Table 5.3. For unseen words, the lemmatizer must learn how to apply general rules instead of memorizing the inflected form-lemma pair. For each language, almost the entire

[5]Note that string-copying isn't a perfect measure of rules as string-copying can contain interesting rules such as `string` $\rightarrow$ `string`, not `string` $\rightarrow$ `str`.

CS

Languages	Training Set (10k)		Validation Set (1k)	
	Distinct Words	Distinct Lemmas	Distinct Words	Distinct Lemmas
Arabic	9809	3181	998	809
English	9841	8377	998	985
Finnish	10000	8668	1000	984
Latvian	9503	5293	996	920
Russian	9935	8186	998	980
Turkish	9901	2934	998	852

UD10K-0CTX

Languages	Training Set (10k)		Validation Set (3k)	
	Distinct Words	Distinct Lemmas	Distinct Words	Distinct Lemmas
Arabic	3356	2508	1263	1013
English	2800	2326	1196	1024
Finnish	5267	2963	1998	1261
Latvian	4936	3194	1692	1221
Russian	5958	4236	2113	1742
Turkish	5816	2638	2089	1285

Table 5.2: Summary of dataset sources used. Summary of distinct inflected forms ("Words") and lemmas ("Lemmas") in the training and development set for the CS dataset and UD10K-0CTX datasets. The numbers in the top row (e.g. 10k) refer to the total number of tokens of each class.

CS validation sets are unseen, while UD10K has only ≈24% (English) to ≈55% (Russian) unseen.

5.6 Rule-based split of a CS dataset

In order to compare specific errors for the question *Do* CONVLEMATUS *and* LEMATUS *make different types of mistakes?* the English test dataset of CS was partitioned by rules such as "remove `-s`" and "remove `-ing` and add `+e`". A manually-coded classifier takes in the inflected form and its lemma and returns a subset of possible rules that could convert the inflected form into the lemma.

CS

Languages	Copying (train)	Copying (valid)	Unseen
Arabic	3.1%	2.4%	94.6%
English	20.3%	19.0%	95.6%
Finnish	4.0%	5.0%	99.9%
Latvian	6.6%	7.2%	90.5%
Russian	9.1%	8.9%	98.6%
Turkish	1.5%	1.7%	98.1%

UD10K-0CTX

Languages	Copying (train)	Copying (valid)	Unseen
Arabic	7.3%	4.6%	27.6%
English	76.1%	74.6%	24.1%
Finnish	36.7%	34.0%	52.5%
Latvian	37.1%	39.3%	41.7%
Russian	3.0%	2.7%	53.7%
Turkish	39.0%	39.5%	45.5%

Table 5.3: Amount of String-Copying and Unseen per dataset. "Copying" is the number of lemmas (not distinct) that are identical to their inflected form for the training and validation set. Unseen are the number of inflected forms from the validation set that are not in the training set. CS tends to have much fewer examples of copying and more unseen examples than UD10K-0CTX.

Rules		Explanation	Example	Train	Test
-	-	no change	`malinvest` → `malinvest`	2031	180
`ed`	-	remove `-ed`	`forayed` → `foray`	1532	184
	`+e`	remove `-d`	`stodged` → `stodge`	1656	168
	`double`	remove `-ed` and duplicate letter	`hugged` → `hug`	276	20
`ing`	-	remove `-ing`	`pearling` → `pearl`	949	101
	`+e`	remove `-ing` and add `-e`	`psychoanalysing` → `psychoanalyse`	832	84
	`double`	remove `-ing` and duplicate letter	`sonogramming` → `sonogram`	166	16
`s`	-	remove `-s`	`DIYs` → `DIY`	1842	185
	`-e`	remove `-es`	`topinches` → `topinch`	103	17
`other`	`other`	other	`wrongtook` → `wrongtake`	454	43

Table 5.4: Summary of RULE-BASED SPLIT. The sets of rules used for subsection 7.3.1 as explained in section 5.6. Also, the number of examples that match those rules from the training and testing set of CS.

(See GitHub[6] for code). This can be used to measure the number of mistakes that were due to applying the wrong rule, or due to scrambling the output. The rules and counts for the training and test dataset from CS are shown in Table 5.4.

5.6.1 CS models are used for speed comparison

Because of the number of models trained for the convolutional exploration question, the CS dataset is also used for the architecture comparison question *Does using convolutional encoders improve speed?*

5.6.2 ud10k-0ctx and ud10k-20ctx for natural distributions

UD10K-0CTX and UD10K-20CTX are used to explore *Does the* CONVLEMATUS *performance change relative to* LEMATUS *when introduced to a more natural distribution of words?* UD10K-0CTX and UD10K-20CTX are extracted from more

[6]`https://gist.github.com/jessstringham/149959520824f4ff9f686a9646ca0c20`

realistic text. Training using UD10K-0CTX and UD10K-20CTX can test a architecture's ability to selectively lemmatize (not overeagerly lemmatizing any word that looks like an inflected form), while still lemmatizing when necessary.

5.6.3 CS-POS and ud10k-20ctx for long dependencies

CS-POS and UD10K-20CTX are used for the question *Can* CONVLEMATUS *handle large distances between input sequence features and where they modify the output sequence?* These "long-distance" dependencies may also occur in the CS dataset, but results are likely to be more apparent in CS-POS and UD10K-20CTX. In both of these, additional information about the inflected form is added to the beginning and/or end of the sequence.

In the case of CS-POS, the information is the part-of-speech tag that is added to the beginning of the inflected form. The lemmatization of many[7] languages involve modifying the suffixes of words (Matthews, 1991). If the POS tag is useful, the model must learn to use information from the tag at location 1 to modify the suffix which may be several letters away. To use an example, in

```
<w> 0 <pos> r e c a r b o n i z e d </w>
```

the POS tag `0` is 12 characters away from the suffix `-ed`. Most CONVLEMATUS configurations explored in this dissertation use a receptive field smaller than that. For those configurations to lemmatize this word using the part-of-speech tag, they need to use the attention and decoder components (see subsection 4.2.3 for a discussion of how convolutions could make up for a small receptive field.)

UD10K-20CTX adds 20 characters of the surrounding text to the inflected form. Now rather than just learning the morphology within words, models trained with this dataset is able to use information from surrounding words too. For example, in

```
<w> t h e <s> i l l u m i n a t e d <lc> l e a v e s <rc> ...
```

that `the` appears before `leaves` makes `leaves` extremely likely to be a noun and should lemmatize as `leaf`. However, `the` is 17 letters away from the letters `-ves`, so the attention, so the convolutional encoder alone cannot learn that.

[7]At least Indo-European languages, which of the languages tested in this dissertation include English, Latvian, and Russian.

Chapter 6

Methods

This chapter describes how the model is trained and evaluated, including the default configurations of the ConvLematus and Lematus models.

6.1 Default Model Configurations

The base code used is the TensorFlow implementation of Nematus.[1][2] The configuration used for all experiments is shown in Table 6.1. When possible, the same configuration used by Bergmanis and Goldwater (2018) was used. Otherwise, the default configuration for the TensorFlow-implementation of Nematus was used.

For the purposes of this dissertation, the Nematus code-base was modified to be able to add convolutional layers (with one kernel size, or concatenating multiple kernel sizes), highway layers, batch normalization, activation functions, and skip connections (unused in these experiments). Initialization for convolutional layer parameters was Glorot Uniform (Glorot and Bengio, 2010).[3]

For all convolutional encoder experiments used in this dissertation, the following configuration was used. Padding is used in all layers (padding is described in section 3.2). If multiple layers are used, each layer uses the same kernel size.

[1]`https://github.com/EdinburghNLP/nematus/`

[2]Note that Bergmanis and Goldwater (2018) used an earlier Theano-based implementation among other reasons, all Lematus models were retrained so they would be comparable to ConvLematus. This was also necessary because Bergmanis and Goldwater (2018) did not train on the CONLL-derived datasets and were trained on UDv2.0 instead of UDv2.1. In addition, a different model-selection criterion was used (lowest validation loss instead of highest validation accuracy.)

[3] For experiments in Appendix B: highway layers also use Glorot Uniform, except for the bias of the gating parameter W_d which is initialized around -2 as recommended in Srivastava et al. (2015) and used in Lee et al. (2016).

Option	Value
Char embedding	300
Hidden state size	100
Encoder output size	200*
Decoder layers	2
Dropout: hidden	0.2
Dropout: embedding	0.2
Other dropout	0
Mini-batch size	50
Optimizer	Adam**
Learning rate	0.0001
Patience	10
Compute	GPU
Max word length	75
Beam Width	12

Table 6.1: Default configuration used for all experiments.
*This value is not explicitly set, but depends on the hidden state size or convolutional layer configuration. For explanation, see subsection 3.1.3.
**Note that Adam (Kingma and Ba, 2014) was *not* the optimizer used in the original LEMATUS paper (Bergmanis and Goldwater, 2018).

For activations, ReLU is used after every layer except the last as it performed best on average in experiments outlined in section A.2. Batch normalization is *not* used.[4]

As subsection 3.1.3 outlined, for a convolutional encoder to be interchangeable with a recurrent encoder of a Lematus model, the last layer of the convolutional component should have `filter_count` = 2 · `hidden_state_size`, where `hidden_state_size` is the hidden state used in corresponding Lematus model's recurrent units. Like Bergmanis and Goldwater (2018), the hidden state for all models is 100. Consequently, all convolutional encoders have a final convolutional layer with `filter_count` = 200. The intermediate convolutional layers also use `filter_count` = 200.

Models were trained for each of the languages in section 5.1. A different random seed was used for each model. In most cases, a single model was trained per language per configuration. The model is optimized by minimizing the cross-entropy loss on the validation set. Training is early stopped, and the values of parameters corresponding to the lowest validation loss were used. (Using this model from training runs has implications on reported accuracies. See section A.1 for an analysis and commentary.).

6.2 Reported Metrics

6.2.1 Accuracy

The reported accuracies are on the validation set for the same language. Accuracy is given by the ratio of exact matches of prediction Y and target T to the total number of words.

$$\text{accuracy} = \frac{\text{exact matchs}}{\text{total words}}. \tag{6.1}$$

6.2.2 Speed

Three metrics are used to approximate speed. The devices used for training and translation were GeForce GTX 1060 6GB. Each model was trained using a single

[4] After the experiments in chapter 7 were ran, batch normalization (Ioffe and Szegedy, 2015) was also tested on some deeper networks in section B.1. While these models had improved training speed, they did not change the accuracy enough to change the conclusions. So the experiments in chapter 7 were not rerun using batch normalization.

GPU.

6.2.2.1 Training speed per epoch

As an approximate measure of speed, we can compare the average number of inflected forms processed per second during training time. Note that this only measures the network computing the loss, which differs from *Translation time*, which measures the network doing beam search. As mentioned in section 6.1, information about the number of inflected forms processed per second is stored every training epoch. For each model, the average of averages is computed. As each epoch is the same size, this gives the average inflected forms per second over the entire training session.

6.2.2.2 Training time score

If training speed is a concern, the *Training speed per epoch* should be weighted by the number of epochs the model took to train. To do that, the following equation is used, where 10000 is the number of examples in the training set

$$\texttt{training\_speed\_score} = \texttt{number\_of\_epochs} \cdot \frac{10000}{\texttt{average\_sentence\_per\_second}}$$

The “training time score” is just an approximation of the time it takes to train, as there is overhead in, for example, computing the validation loss. Hence this metric is called a “score” instead of a “time.”

6.2.2.3 Translation time

The final metric for speed is *Translation time*, or the time it takes to translate the validation set. This would be the most relevant if one wanted to use the lemmatizer as part of a pipeline and just needed to lemmatize a dataset. The time metric is extracted by reading the timestamp when the script that performs translation was called and comparing it to the timestamp when the accuracy was outputted.

Additional meta-data is stored for each model trained, including the number of parameters and the average number of inflected forms processed per second in the training epoch.

6.3 Code

The configuration of each experiment is stored in a YAML file. Experiment runs can be loaded into a Pandas DataFrame. Code can be found on GitHub[5], including modifications to the Nematus code[6].

[5]http://github.com/jessstringham/dissertation

[6]https://github.com/jessstringham/dissertation/blob/master/nematus/patches/nematus_add_conv

Chapter 7

Results

This section outlines experiments and results that use the methods described in chapter 6 to explore two main groups of questions:

- *How do different architectures of the convolutional encoder used in* ConvLematus *affect lemmatization accuracy across different languages?* (section 7.2)
- *How do* Lematus *and* ConvLematus *differ in lemmatization?* (section 7.3)

The first group of experiments, section 7.2, measure the effects of different types of convolutional encoders in ConvLematus models trained on the CS dataset (described in section 5.3). Then a single ConvLematus configuration is chosen for the second part, section 7.3. In the second part, models using the Lematus and the chosen ConvLematus architectures are trained on three additional datasets: CS-POS, ud10k-0ctx, and ud10k-20ctx (see chapter 5).

Throughout this section, the shorthand `n3k4` means a ConvLematus model with 3 consecutive convolutional layers each with `kernel_size` = 4.

7.1 Lematus

In section 7.2, Lematus validation accuracy is used as a measure of how difficult each language is. In section 7.3, ConvLematus is compared directly to Lematus. For each language, Lematus models were trained on CS using three different seeds. The results are shown in Table 7.1.

language	min	med	max	average
Arabic	85.9%	88.1%	88.1%	87.1%
English	93.7%	93.7%	94.6%	93.8%
Finnish	83.1%	83.5%	85.6%	83.4%
Latvian	81.1%	82.3%	83.5%	82.4%
Russian	83.9%	84.1%	84.7%	84.2%
Turkish	93.0%	93.2%	93.4%	93.0%

Table 7.1: Lematus validation accuracy on CS. Results are ordered within each language by validation accuracy to illustrate the range of accuracies. In later parts of this chapter, when a spread is needed, the range between the min and max are used. When a single number is needed, the averaged accuracy of the three models is used.

7.2 ConvLematus Convolutional Encoder Architectures

This set of experiments explore the first question: *How do different architectures of the convolutional encoder used in* ConvLematus *affect lemmatization accuracy across different languages?* Specifically

- *Does the size of the receptive field contribute to the network's accuracy?* (subsection 7.2.1)
- *Does convolutional layer depth matter, or do the extra layers just increase the receptive field?* (subsection 7.2.2)
- *Which* ConvLematus *architecture performs the best across languages?* (subsection 7.2.3)

These experiments will involve modifying the kernel size and depth of the convolutional encoder and measuring the trained model's accuracy on CS.

7.2.1 ConvLematus with a single convolutional layer

One issue with using a convolutional encoder is that it has a fixed receptive field, as described in subsection 4.2.3. To summarize, the convolutional encoder of ConvLematus can only learn relationships within a fixed-length window

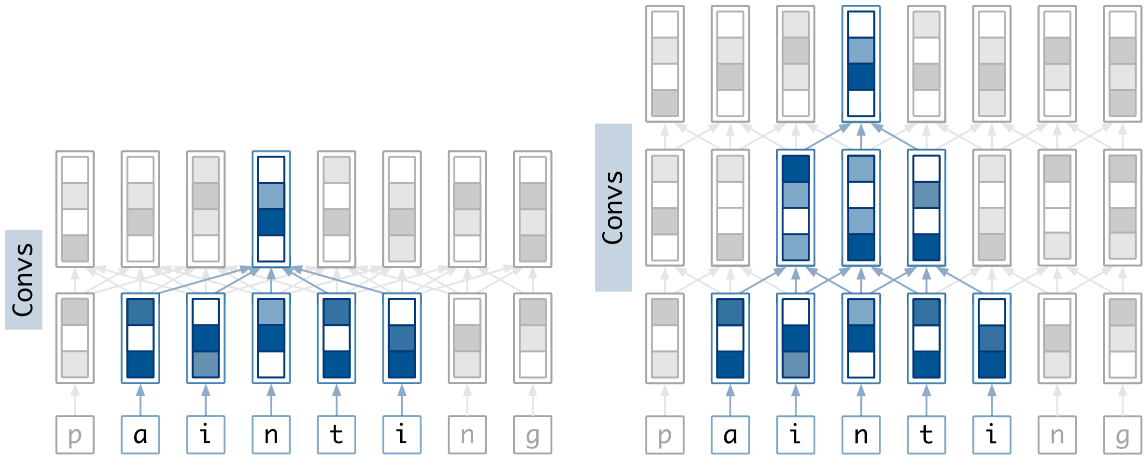


Figure 7.1: Two convolutional different encoders with the same receptive field. Both images use the same scheme laid out in Figure 2.1. **A single convolutional layer with `kernel_size` = 5 (left)**, notated as `n1k5`. **Two convolutional layers with `kernel_size` = 3 (right)**, notated as `n2k3`. Both have a receptive field of size 5.

of characters covered by the convolutional encoder's receptive field. The other components of the seq2seq architecture, such as the attentional mechanism and the decoder may be able to expand the receptive field to cover the full input sequence.

This set of experiments measures how the receptive field size relates to lemmatization accuracy. A very simple configuration of CONVLEMATUS is tested: the convolutional encoder is a single convolutional layer with a fixed `kernel_size` (illustrated in the left diagram in Figure 7.1). The size of the receptive field is varied by changing the size of the kernel. Everything else about the network remains the same. For each language, 13 models are trained with `kernel_size` between 1 and 13. The accuracies of these models are summarized in Figure 7.2. A subset of the single-layer results is also displayed in Figure 7.4.

Based on the accuracies of the single-convolutional-layer encoders, a few observations can be made.

Tiny kernel sizes result in low validation accuracy. All `n1k1` models perform poorly. In `n1k1`, the convolutional encoder just computes a linear combination of each individual embedding without combining with vectors from adjacent characters. The low accuracy could mean that the attention

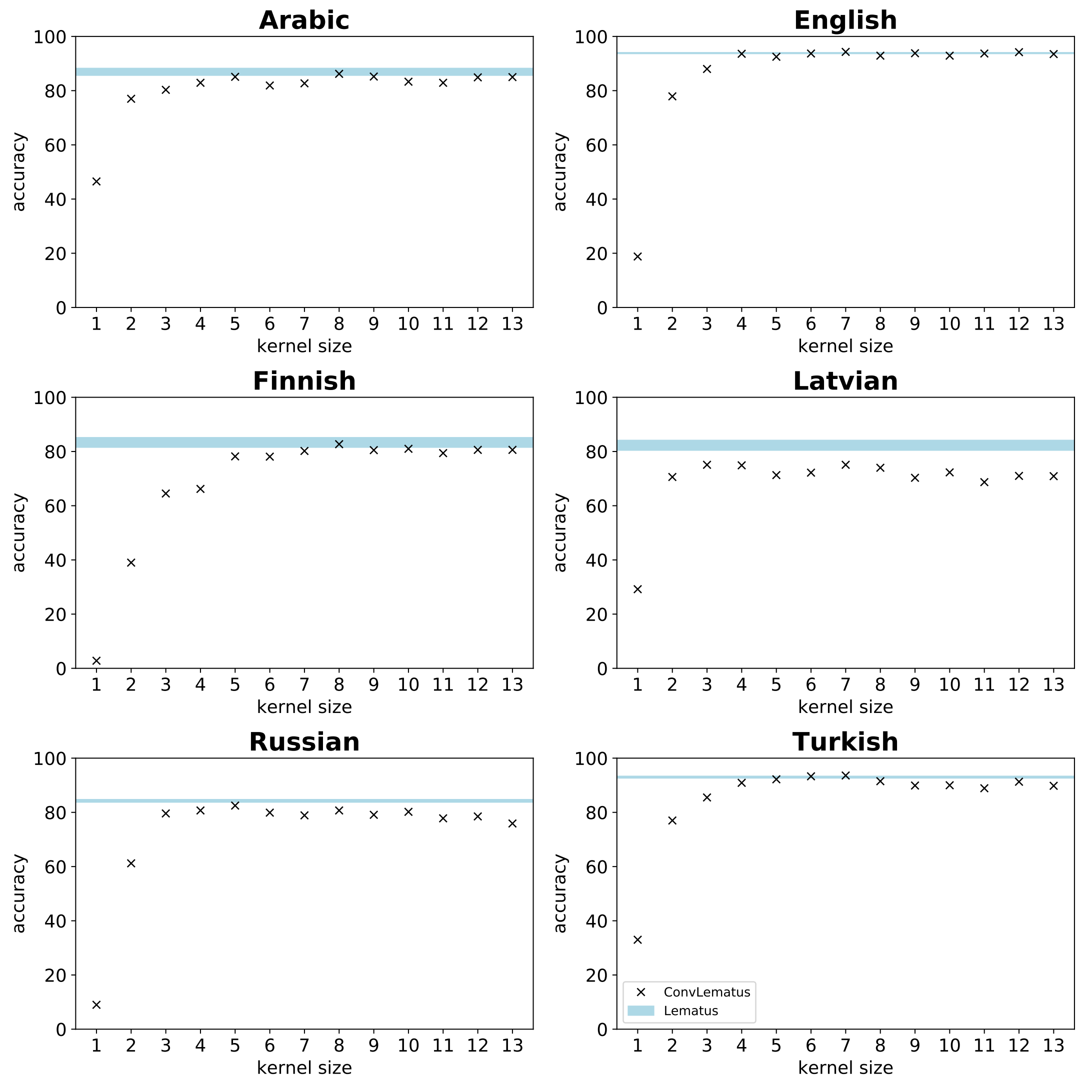


Figure 7.2: Single layer validation accuracies on CS. Validation accuracy on the CS dataset vs `kernel_size` for models with a convolutional encoder with a single convolutional layer and no activation. Each dot represents the accuracy of a model trained on the given language. All models have a convolutional encoder which consists of a single layer with the kernel size noted on the x-axis. The blue horizontal line shows the range of performance of the Lematus models (see Table 7.1). The bottom boundary of the shaded area represents the lowest accuracy Lematus model, and the top is the highest accuracy Lematus model.

and decoder components do not completely compensate for a poor encoder. However, an alternative explanation is that the low accuracy is because the hyperparameters were not optimized for very small kernel sizes. The improvement with larger kernel sizes (`kernel_size` = 2, 3, 4, ...) shows that a larger kernel size in the convolutional encoder does contribute to the validation accuracy.

Single-layer convolutional encoders rarely beat the Lematus model No Turkish, Latvian, or Russian convolutional models outperform the Lematus. The Arabic and Turkish models that outperform Lematus do not outperform by much: it can probably be attributed to differences due to random seeds. That said, it is notable that even a very simple ConvLematus can perform within 2% of Lematus in most languages.

Larger kernel sizes only improves accuracy to a point. Interestingly, the validation accuracies of ConvLematus suggest they are reaching an upper-bound, which is at or below the Lematus performance. Again, as hyperparameters were not tuned, we cannot rule out that the poor performance is due to hyperparameters. However, it does suggest that the settings of the attention and the decoder are restricting improvements in accuracy for both Lematus and ConvLematus. Another potential reason is some inflected forms in the validation set is not possible to learn without luck.

The smallest useful kernel size differs by language. Using an arbitrary cutoff of 10% below Lematus performance, Arabic and Latvian models reach the cutoff using kernel size 2 or greater, Russian and Turkish require kernel size 3, and Finnish doesn't reach that level of performance until kernel size 5.[1] It's possible that this is due to specific morphological rules in the language that cannot be detected by smaller window sizes. This could be interesting to further explore.

7.2.1.1 Attention by kernel sizes

As described in subsection 4.2.3, we'd expect smaller kernel sizes to push more work onto the attention mechanism and decoder. We can visualize how the atten-

[1]Since there is an element of chance in each individual model's validation accuracy, it's possible a model had a bad initialization and the exact cutoff could be slightly different than what is shown.

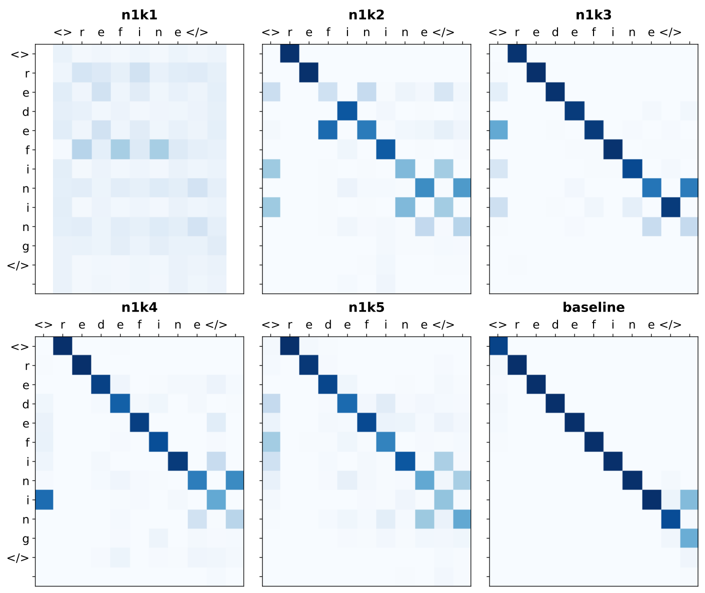


Figure 7.3: Attention plots for different English ConvLematus lemmatizing the word `redefining`. `n1k1` and `n1k2` both incorrectly lemmatize the word (apparently overeager to delete the `ed` in the middle of the word). Most of the ConvLematus are shifted to the right compared to Lematus.

tion mechanism is weighting each element of the encoded sequence for different models on the same word. Figure 7.3 shows attention plots (generated using a similar method as the alignment plots in Bahdanau et al. (2015)) for several single-layer ConvLematus models and a Lematus model. In this example, `n1k1` and `n1k2` models both produce the incorrect lemma. The mistakes themselves are interesting: `n1k1` deletes the `ed` in the middle of the word (which would be perfect if it was at the end of the word). Oddly, `n1k2` deletes the `ed` and then repeats part of the inflected form. Looking at the attention plot, it does seem like `n1k1` and `n1k2` have more complicated attention plots than the other plots. In this case, it does not enough to make up for the convolutional encoder.

7.2.2 Receptive Field vs Depth

In subsection 7.2.1, we showed that increasing the receptive field using a single convolutional layer influences the lemmatization accuracy up until a point. Subsection 4.2.4 suggested an intuition for why multiple convolutional layers could perform better because they could represent the hierarchy reflected in morphology theory. An alternative hypothesis is that multiple convolutional layers only perform better because they create a larger receptive field. For this section, the question of interest is: *Does convolutional layer depth matter, or do the extra layers just increase the receptive field?*

In this set of experiments, ConvLematus models with 2, 3, 4, and 5 convolutional layers are trained on the languages. These models and subsection 7.2.1 are grouped by their receptive field size and depth, where the receptive field is computed using Equation 3.1. For example, the two configurations illustrated in Figure 7.1 (`n2k3` and `n1k5`) would appear in the same column (receptive field = 5). The results are shown in figure 7.4. Next, deeper models are compared to their counterparts with the same receptive field but only one layer in Figure 7.5.

Based on these experiments, the following observations can be made.

Receptive field alone does not account for accuracy. For example, the experiments on the receptive field of size 3 for Finnish, Russian, and Turkish all improve when two layers of kernel size 2 are used rather than a single layer of kernel size 3. In Latvian, Russian, and Turkish, single-layer networks with large receptive field sizes (9+) tend to perform poorer than deeper networks with the same receptive field size.

There are trends, but some differ across languages. While accuracy is a noisy measure, there are still visible trends. For example, the smallest receptive field size plotted, 3, does not produce the highest performing model in any language. However, different languages have different trends, which may reflect differences in the language morphology or at least the CS dataset. In Finnish, deep networks with a large receptive field tended to perform better (the bottom right). In Latvian and Russian, the best performing models had a smaller receptive field size, and were deep but not too deep.

Deep networks do slightly better. Deep networks do slightly better more often than not, but shallow networks perform well too. Figure Figure 7.5 attempts to quantify this difference. The figure shows that networks with more than one layer do have a slight advantage over single-layer networks. Deeper networks didn't provide a clear advantage. For Arabic, Finnish, and Turkish, the best-performing model was one a single convolutional layer.

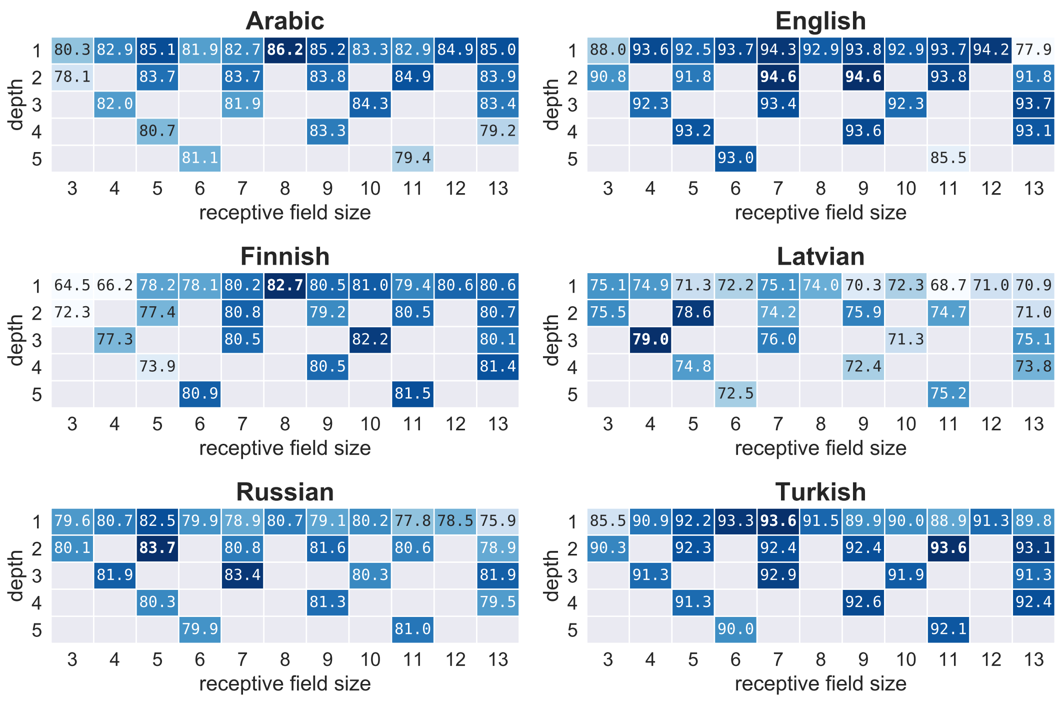


Figure 7.4: Effect of convolutional encoder's receptive field and depth on lemmatization accuracy. Effect of convolutional encoder's receptive field and depth on lemmatization accuracy. Each number is the validation accuracy percentage of a single model trained on the given language. The model uses a convolutional encoder with the specified number layers. Its kernel size in combination with the number of layers gives the receptive field size. (The exact kernel size used can be extracted using Equation 3.1.) Blank squares represent invalid configurations. The squares are shaded based on accuracy. The minimum cutoff for the shading (nearly white) is 10% less than the best-performing model per language (bold text and dark-blue).

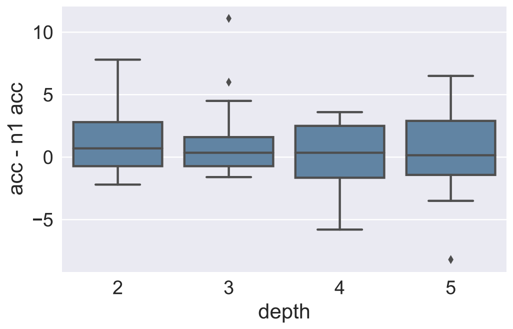


Figure 7.5: Difference between deeper ConvLematus models and `n1` models with the same receptive field size. Each accuracy in Figure 7.4 with depth 2 or more has the accuracy from the model with the same receptive field but depth 1 subtracted from it (e.g., `n3k3` has `n1k7`'s accuracy subtracted from it). Then boxplots are drawn for each depth. If increasing depth while holding the size of the receptive field constant did not improve the model, the values would be near 0. This shows that deeper networks perform better than the `n1` models more often than not.

7.2.3 ConvLematus model choice

The above experiments allowed us to explore the question: *Which* ConvLematus *architecture performs the best across languages?* In order to compare to Lematus, a specific ConvLematus configuration will be chosen from those in section 7.2. Since a goal will be to perform well relative to Lematus on all languages, the average of the corresponding language Lematus accuracy was subtracted from each model's validation accuracy. The percent difference was then averaged across languages for a given configuration. Both `n2k6` and `n3k3` tied for the best relative accuracy with -2.64%. The tie was broken by choosing the model that performed best on its worst language according to the percentage difference, which was `n3k3`. The ConvLematus configuration `n3k3` will be used for the experiments in section 7.3.

7.3 ConvLematus vs Lematus

This section explores *How do* Lematus *and* ConvLematus *differ in lemmatization?* Specifically, the set of questions are:

inflected form	lemma	`n1k2`	`n1k7`	`n3k3`	LEMATUS
beautified	beautify	beautify	beautify	beautify	beautify
frazzling	frazzle	frazzle	frazzle	frazzle	frazzle
unshelled	unshell	unshell	unshell	*unshel	*unshel
cross-breeds	cross-breed	*creeds-bross	cross-breed	cross-breed	cross-breed
unbenumbed	unbenumb	*unbenbenb	unbenumb	unbenumb	unbenumb
strikesthrough	strikethrough	*stroughrike	*strikesthrough	*strikesthroogh	*strikesthrough

Table 7.2: Examples of English CONVLEMATUS lemmatization. Examples chosen of words that all models can correctly lemmatize, and mistakes (annotated with *) some models made.

- *Do* CONVLEMATUS *and* LEMATUS *make different types of mistakes?*
- *Does using convolutional encoders improve speed?*
- *Does the* CONVLEMATUS *performance change relative to* LEMATUS *when introduced to a more natural distribution of words?*
- *Can* CONVLEMATUS *handle large distances between input sequence features and where they modify the output sequence?*
- *Can* CONVLEMATUS *learn rules other than morphology?*

If not otherwise stated, the CONVLEMATUS used is `n3k3`, which was chosen in subsection 7.2.3.

7.3.1 Comparison of mistakes

This section aims to answer the question: *Do* CONVLEMATUS *and* LEMATUS *make different types of mistakes?* To begin to explore this, examples of English lemmatizations and mistakes are shown in Table 7.2.

7.3.1.1 Applying the wrong rule

Looking at the errors that CONVLEMATUS `n3k3` and LEMATUS get in Table 7.2, examples like `unshelled` $\rightarrow$ `unshel` appear to show the model applying the wrong "rule" ("remove the `ed` and delete the duplicate letter" instead of "remove the `ed`"). This differs from the mistakes CONVLEMATUS `n1k2` model makes, which more often involves garbled output such as `unbenumbed` → `unbenbenb`.[2] This section attempts to dive into what kinds of rules the architectures learn and whether the mistakes can be attributed to wrongly applying the rules. Specifically, it asks *Are the errors of* ConvLematus *due to applying the wrong rule?* We also can compare it to what kinds of errors Lematus makes.

Overall, `n1k2` correctly lemmatized 79.9% of the inflected forms, `n3k3` 93.6%, and Lematus 94.7% of the test set. The datasplit used is specified in section 5.6. Results are shown in Figure 7.6.

(Keep in mind, "delete `-ed` add `-e`" is only a convenient way to label the modification of the word. This analysis is not meant to imply the model is learning that specific mechanism. The following sections will use "the model applied the rule" as shorthand for "The model's lemmatization corresponds to the label...")

A second point of clarification is about the "other/other" and "-/-" rules. If the "True Rule" is "other/other", this means it can't be lemmatized using the rules encoded (e.g. `caught` → `catch` or `vegges` → `veg`). If the "Predicted Rule" is "other/other", it means the lemma cannot be reached using the rules coded (e.g. maybe `caught` → `catch`, but also maybe `unbenumbed` → `unbenbenb`). However, if both are "other/other", it may mean the model incorrectly lemmatized it. In all other rules, the diagonal means the network correctly lemmatized it. Next, a "True Rule" of "-/-" represents cases where the word shouldn't change, and a "Predicted Rule" of "-/-" represents cases where the model returned the inflected form as the lemma.

The tiny kernel model, `n1k2`, tends to apply "other" rules to inflected form The darker column on the right side of the `n1k2` chart above "other/other" indicates that rather than applying known rules, the model uses some other rule. (While it's possible the rules are reasonable but fall outside of the ones coded, based on the examples in Table 7.2, it seems more likely that it's more likely that `n1k2` is garbling the word.)

Many of `n3k3` errors are due to applying the wrong rule. This is evident by not having many "other/other" predicted rules when there is another rule to apply. For example, there are two words that Lematus and `n3k3`

[2]That said, garbled output isn't limited to ConvLematus or `n1k2`. For the inflected form `reverse-pickpocket`, `n3k3` lemmatized it as `reverse-picket` and Lematus lemmatized it as `reverse-picktock`.

should have removed `-es` ("s/-e") but instead only removed `-s` ("s/-"). In particular, errors on `-ing` tend to be other `-ing` rules, and the same for `-s` and `-ed` errors. This is also the case for Lematus.

7.3.1.2 Differences in lemmatization

So far, this section has viewed specific examples of errors, and a way of visualizing errors in English. As a final approach, we can quantify how many differences there are between the models. One problem with aggregated accuracies such as those in Figure 7.2 and Figure 7.4 is that when the same accuracy is reported for two models, it's difficult to tell if they're lemmatizing the same words correctly. Figure 7.7 compares ConvLematus models to a Lematus trained on the same language, and groups words by which models correctly lemmatize them. This can begin to tell us if two models with identical accuracies are lemmatizing in the same way.

From this, we see that some ConvLematus models do correctly lemmatize words that the Lematus model incorrectly lemmatized. In Arabic, Finnish, and Latvian, some ConvLematus models correctly lemmatize at least 50 words that Lematus incorrectly lemmatizes. The difference in types of words incorrectly lemmatized means that using an ensemble of Lematus and ConvLematus could have better performance than either individual model. section C.1 shows a way to check if the dark regions (one or the other is correct, or the "xor-count") is more than you'd expect due to the randomness in the model. With the exception of Turkish `k5-k8`, all ConvLematus showed in Figure 7.7 had a higher "xor-count" a pair of Lematus models with identical configurations but different seeds.

7.3.2 Training Speed

Previous sections showed that ConvLematus models reached around the same performance as Lematus for almost all languages, but few experiments surpassed Lematus models' performance. However, the convolutional models did train noticeably faster. The three metrics used to measure the network's speed are outlined in subsection 6.2.2. The comparison of the models trained for subsection 7.2.1 and subsection 7.2.2 are shown in subsection 7.2.2.

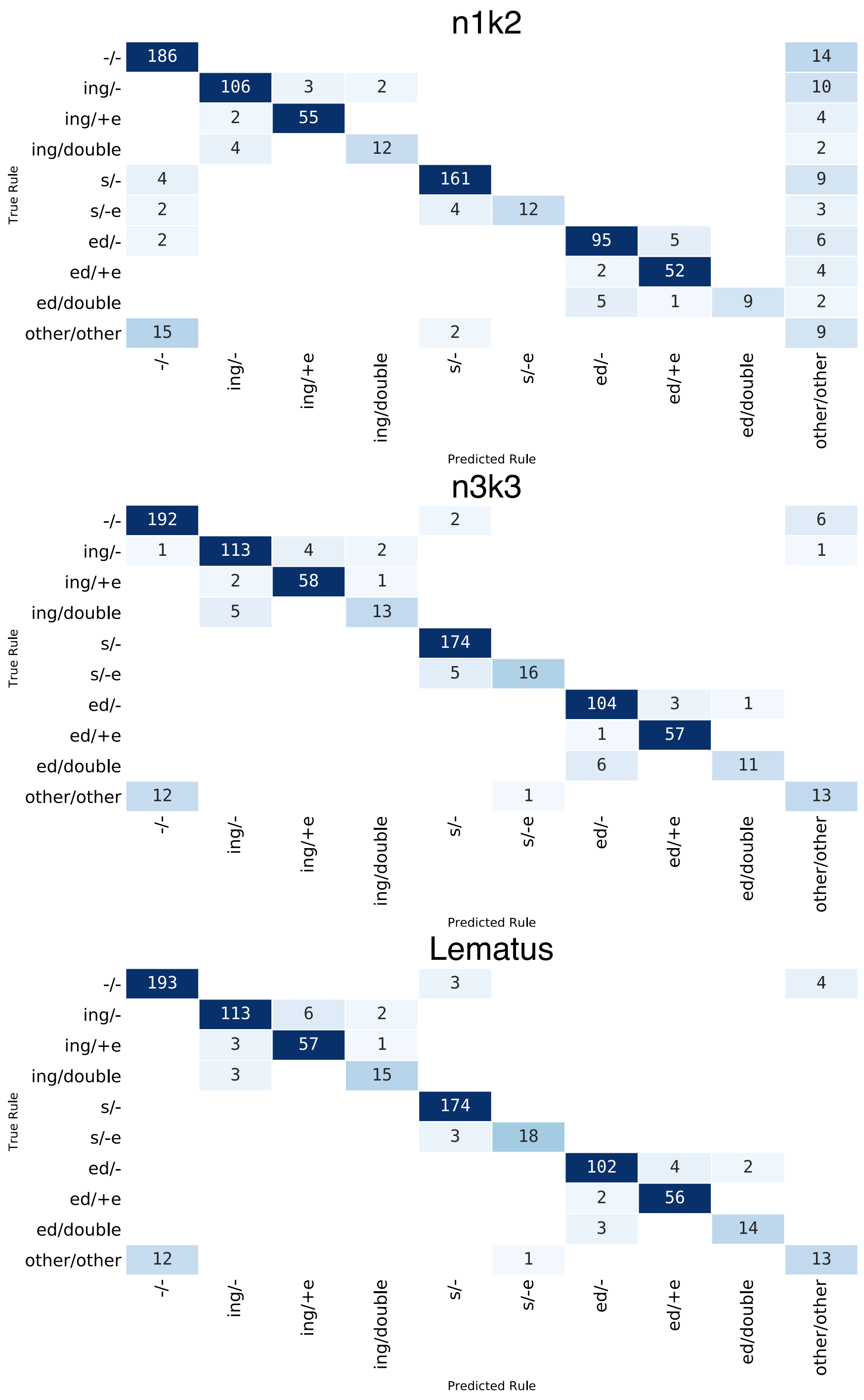


Figure 7.6: An overview of rules applied by three models. See Table 5.4 for a description of rules. If a model used the correct rule on every inflected form in the dataset, the non-diagonal values would be 0. The x-axis represents the rule that the model “applied”. The y-axis represents the rule needed to reach the target inflected form.

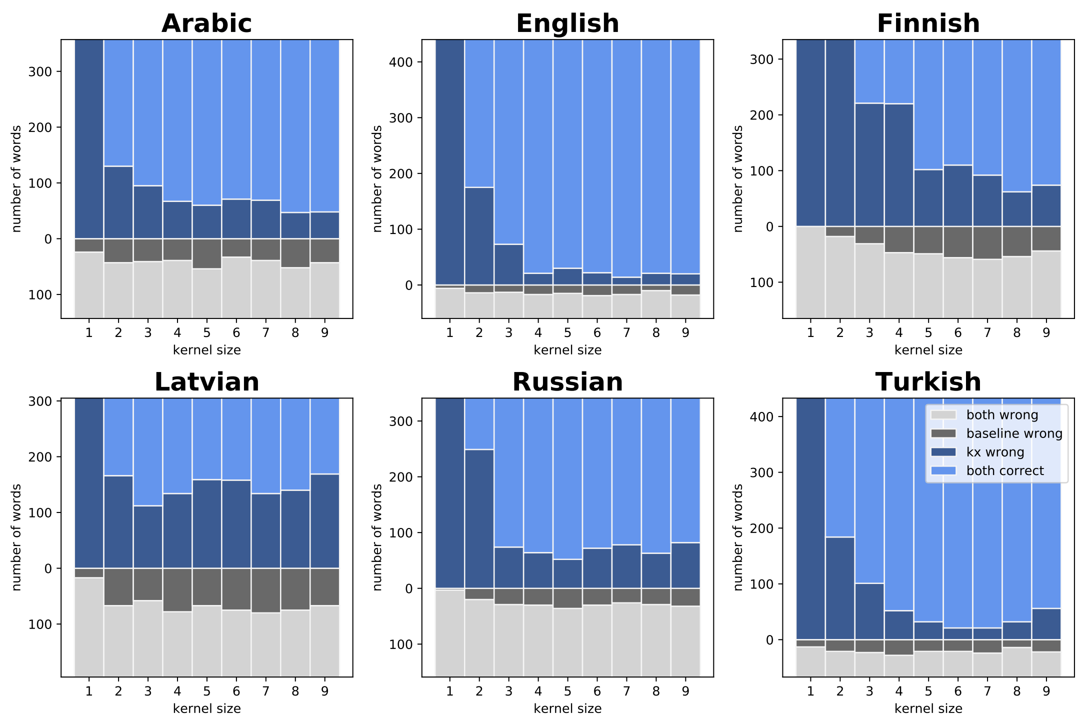


Figure 7.7: Each vertical bar compares a LEMATUS model to a single-layer CONV-LEMATUS with the specified kernel size. In each bar, there is a light and dark blue region above 0, which together represent the number words that the LEMATUS model correctly lemmatizes. The light and dark gray region below 0 together represent the number of words that the LEMATUS model incorrectly lemmatized. The two light regions represent areas where the models are in agreement (both lemmatized correctly or incorrectly), and the dark regions represent areas where the models do not agree. For visualization purposes, the image is cropped to exclude 500 of the words LEMATUS correctly lemmatized, first removing words that both models correctly lemmatized.

As an example, the `kernel_size` = 6 bar in the Turkish chart is a comparison of a Turkish LEMATUS model to a CONVLEMATUS `n1k6` model. From the top down, the light blue bar represents the number of words both models correctly lemmatized (add 500 to account for cropping). The dark blue region represents the number of words LEMATUS correctly lemmatized but `n1k6` incorrectly lemmatized. The dark gray region represents the number of words LEMATUS incorrectly lemmatized but `n1k6` correctly lemmatized, and the light gray region represents the number of words both models incorrectly lemmatized.

In the case of Turkish `k6`, both models lemmatized 21 words incorrectly. However, the chart highlights that the models lemmatized 21 *different words* incorrectly. In aggregated reports of accuracy such as Figure 7.2, in cases where the dark regions are the same size would correspond to both models achieving the same accuracy. The more differences in which words are lemmatized incorrectly, the larger the dark regions.

Training speed per epoch is the best metric to look to remove many sources of noise. This metric compares the speed that the loss can be computed on the validation set. A higher value means that the group of models are faster. In this view, ConvLematus clearly outperforms Lematus by being able to compute the loss of more inflected forms per second. *Training time score* is more interesting if we care about the time it takes to train the model, which is more practical than *Training speed per epoch*. This metric weights the *Training speed per epoch* by the number of epochs the model takes to early stop. In this case, smaller numbers indicate faster training. ConvLematus tends to usually outperforms Lematus, but not by as large of margin as in *Training speed per epoch*.[3] *Translation time* is the clocktime of running the translation script. Again, smaller numbers are better. *Translation time* is the most useful if one cares about the time it takes to run the lemmatizer. For the most part, ConvLematus outperforms Lematus.

7.3.3 Non-morphological rules and long dependencies

This section begins to explore the questions

- *Can* ConvLematus *handle large distances between input sequence features and where they modify the output sequence?*
- *Can* ConvLematus *learn rules other than morphology?*

For both questions, the same datasets are used. As described in subsection 5.6.3, the CS-POS and ud10k-20ctx datasets both introduce information outside of the word. For example, CS-POS adds the part-of-speech to the beginning of the sequence. As a result, this may add more examples of where there is a large distance between features in the input sequence and what they should influence in the output sequence. If these distances are larger than the convolutional encoder's receptive field, then the convolutional encoder cannot learn the relationship and the attention and decoder mechanisms have to compensate.

Another aspect is the ability of models to learn other types of rules than modifying spelling based on a word. In the case of CS-POS, it may be learning how to use the POS tag to choose the correct suffix. In the case of ud10k-20ctx, it may be detecting relationships between words and knowing how they

[3]The models that achieved the lowest *Training time score* were the batch normalized models from section B.1. These 10 models had a median of 2.0k speed by drastically reducing the number of epochs required, but decreased word/sec/epoch to a median of 295.

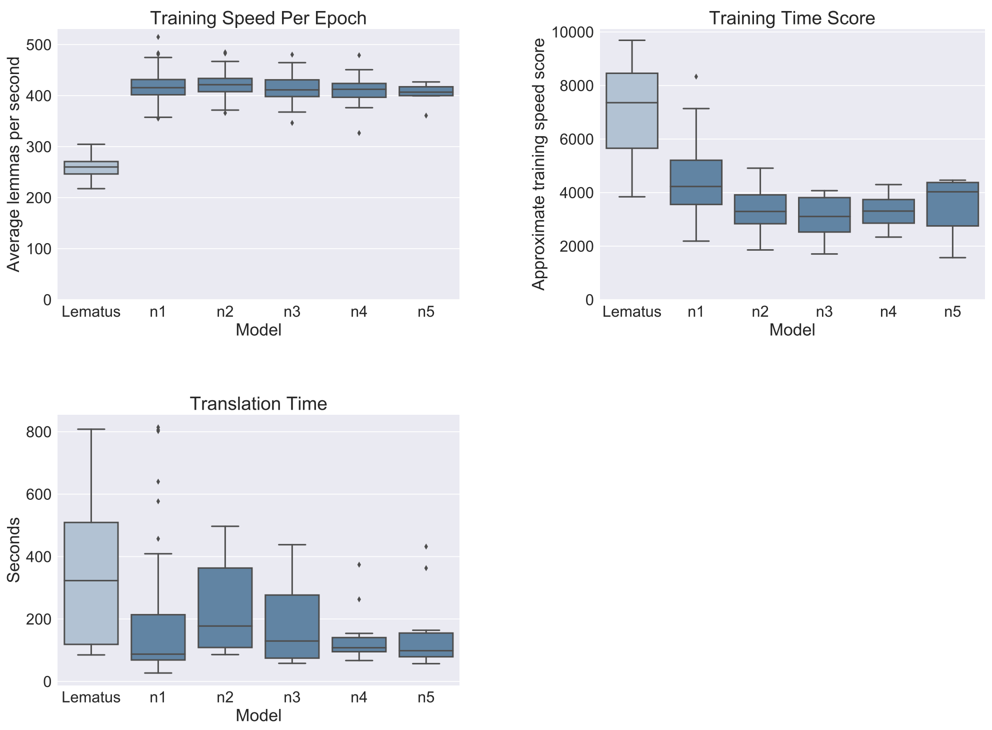


Figure 7.8: Comparison of LEMATUS and CONVLEMATUS on training and translation speed. A higher value in *Training Speed Per Epoch* means that training the model for an epoch is faster. The light blue box plot shows the average sentences per second of the 18 LEMATUS models. The remaining blue box plots represent the average sentences per second for CONVLEMATUS with one (n1) to five (n5) convolutional layers.

affect lemmatization (e.g. `the leaves` means that `leaves` should lemmatize to `leaf` instead of `leave`).

Because CS-POS is identical to CS except for POS tags, and ud10k-20ctx is identical to ud10k-0ctx except for context, a initial way to test for the use of this information is to measure whether the accuracies increase when POS/-context is added. The validation accuracies of CS-POS and ud10k are shown in Table 7.3 and Table 7.4. Both ConvLematus and Lematus achieved higher accuracies on ud10k-20ctx than ud10k-0ctx on every language. Nearly all accuracies on CS-POS were higher than CS. Surprisingly, even CS-POS English improved by a small amount, even though all entries in English were the same POS and the tags did not add useful information (distributions of POS tags are shown in Figure 5.1).

Next, we can verify whether the trained models are using the POS information at all. This investigation will focus on Russian because it has a variety of tags (compared to English). First, Figure 7.9 shows the attention plots for an arbitrarily chosen word. The plot appears to show ConvLematus not using the POS information for that word as much as Lematus does. Whether ConvLematus uses the POS tags can be further investigated by shuffling the POS tags of the validation set input sequences. Specifically, the POS tags are collected from the validation training set (so if there are 300 noun tags in the original dataset, there are noun tags attached to a random set of 300 words in the shuffled dataset.) Then we can lemmatize them using the already-trained models on them and compute the accuracy of the output. If ConvLematus was actually learning to ignore the tags, the accuracy should remain the same. The results of the shuffled-POS test for Russian is shown in Table 7.5. The table shows that both models experience a large drop in accuracy, suggesting that counter to the attention plot's suggestion, both models depend on the correct POS tag. Interestingly, ConvLematus perform better than Lematus did on the shuffled CS-POS. This may suggest that Lematus depends on the correct POS more. Perhaps ConvLematus had a more difficult time learning relationships about the POS tag.

	CS		CS-POS	
Language	Lematus	`n3k3`	Lematus*	`n3k3`
Arabic	**87.1%**	81.9%	**91.6%**	90.3%
English	93.4%	93.4%	**93.7%**	93.6%
Finnish	**83.4%**	80.5%	**85.2%**	84.4%
Latvian	**82.4%**	76.0%	**87.5%**	81.5%
Russian	**84.2%**	83.4%	**86.7%**	83.3%
Turkish	**93.0%**	92.9%	92.1%	**94.1%**

Table 7.3: Experiments on CS datasets for Lematus and ConvLematus. *The Lematus dataset is the average of three runs, shown in Table 7.1.

	ud10k-0ctx		ud10k-20ctx	
Language	Lematus	`n3k3`	Lematus	`n3k3`
Arabic	**74.4%**	43.6%	**76.8%**	72.6%
English	91.3%	**91.8%**	**92.6%**	92.5%
Finnish	**72.8%**	70.0%	**75.0%**	74.0%
Latvian	**75.9%**	73.2%	**80.2%**	77.4%
Russian	**84.9%**	81.3%	**86.7%**	83.3%
Turkish	83.5%	**84.9%**	84.7%	**85.1%**

Table 7.4: Experiments on the ud10k dataset for Lematus and ConvLematus

Tag	Count	Lematus	ConvLematus
N	357	33.3% (-43.7%)	39.8% (-32.5%)
V	259	50.2% (-38.6%)	59.8% (-26.6%)
ADJ	327	48.0% (-46.8%)	52.9% (-38.8%)
V.PTCP	35	45.7% (-42.9%)	60.0% (-31.4%)
V.CVB	22	45.5% (-50.0%)	36.4% (-40.9%)
All	1000	43.2% (-43.5%)	49.9% (-33.2%)

Table 7.5: Validation accuracy for Russian CS-POS before and after shuffling POS tags, split by tag. The percentage indicates the accuracy on the shuffled dataset, and the parenthetical indicates the change from the unshuffled dataset. Both ConvLematus and Lematus classify less accurately when using shuffled POS tags, suggesting both models are using the POS tag. The shuffled tags do impact Lematus more.

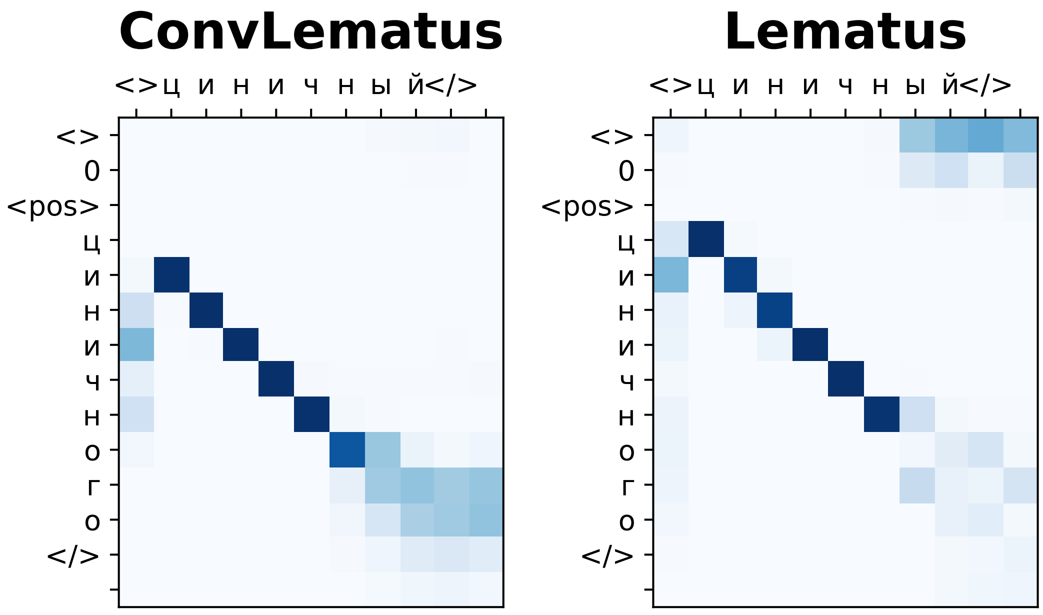


Figure 7.9: Attention plots for two Russian words using the CS-POS-trained models. The `<w> 0 <pos>` signifies the part-of-speech tag. In the plot for LEMATUS (right), a dark area in the upper-right corner suggests that LEMATUS may be using the POS information. The same area is highlighted much more subtly on the plot for CONVLEMATUS (left), suggesting that CONVLEMATUS may not use the POS tag in this case. Both models accurately lemmatize the word. Note that this may not be typical. Another observed plot of CONVLEMATUS does have a dark area in the upper-right corner (see Figure C.1).

	Unseen		Seen	
Language	Lematus	`n3k3`	Lematus	`n3k3`
Arabic	**37.4%**	21.0%	**88.6%**	52.3%
English	**77.8%**	76.3%	95.3%	**96.8%**
Finnish	**53.0%**	49.6%	**94.5%**	92.6%
Latvian	**55.6%**	52.6%	**90.5%**	88.0%
Russian	**75.5%**	69.5%	**95.8%**	95.0%
Turkish	70.1%	**72.3%**	94.8%	**95.5%**

Table 7.6: Unseen/seen accuracy the ud10k-0ctx dataset for Lematus and ConvLematus

7.4 Ability to lemmatize a natural word distribution

This section begins to explore the question *Does the* ConvLematus *performance change relative to* Lematus *when introduced to a more natural distribution of words?* As section 5.4 described, the ud10k-0ctx uses a word distribution more similar to natural text. The overall accuracy of models trained on this dataset is shown in Table 7.4. It is also useful to split this accuracy by words that are seen and unseen in the training set, which is shown in Table 7.6. While not directly comparable, it is interesting that even though CS was completely unseen words, both ConvLematus and Lematus achieved a much higher accuracy on that dataset than on the unseen words in ud10k-0ctx. It may be that needing to learn both when to lemmatize and how to lemmatize is difficult. It could also be that the ud10k datasets lack variety in lemmatizations. Also interestingly, the Turkish ConvLematus performs better on both ud10k-0ctx and ud10k-20ctx.

It is also important to note that these results are a few percentage points lower than those reported in Bergmanis and Goldwater (2018). This is likely a reflection of the model selection (section A.1), which uses the lowest validation loss instead of accuracy.

Language	ConvLematus `n3k3`	Lematus
Arabic	81.3%	86.0%
English	93.7%	94.7%
Finnish	82.5%	84.6%
Latvian	77.2%	83.9%
Russian	81.5%	83.9%
Turkish	91.5%	93.1%

Table 7.7: Test set results from CS.

7.5 Test Set Results

Finally, the two models used can be run on the CS test set. The results are shown in Table 7.7. As before, ConvLematus trails Lematus by a few percentage points.

Chapter 8

Conclusions

In this dissertation, we designed and implemented ConvLematus, a new lemmatizer based on the encoder-decoder seq2seq lemmatizer Lematus that uses a convolutional encoder. The motivation was that convolutions are frequently used for handling morphology in NLP and convolutions tend to be faster. While convolutional encoders have a disadvantage compared to recurrent encoders because of their limited receptive field, we hypothesized that the rest of the encoder-decoder architecture could compensate for it.

Some ConvLematus models achieved within a few percentage points of state-of-the-art lemmatizer while lemmatizing much faster. There are also many other convolutional encoder architectures that can be explored that may improve accuracies as well.

8.1 Questions

The first set of experiments compared different convolutional encoders in ConvLematus to explore the question: *How do different architectures of the convolutional encoder used in* ConvLematus *affect lemmatization accuracy across different languages?*

- *Does the size of the receptive field contribute to the network's accuracy?*

  Yes. If the receptive field is too small (receptive field size of 1 or 2), the ConvLematus models trained did not lemmatize very well. Though it varies by language, the models with greater than receptive field size 5 tend to have similar performance without much additional improvement.

- *Does convolutional layer depth matter, or do the extra layers just increase the receptive field?*

  A little. More often than not, models using deeper convolutional encoders performed better than their corresponding single-layer convolutional encoder model with the same receptive field size. Some languages had interesting sets of configurations that worked better. For example, a configuration that performed very well on Russian (highest accuracy) and Latvian (second highest) was two layers deep with a receptive field size of only 5, while deeper, shallower, and larger receptive field sizes performed worse.

- *Which* ConvLematus *architecture performs the best across languages?*

  This wasn't a clear winner as different languages had different types of configurations that worked best. The configuration that performed the least bad and that was used for the next experiments was `n3k3`

The second set of experiments compared ConvLematus to Lematus to answer the question *How do* Lematus *and* ConvLematus *differ in lemmatization?*

- *Do* ConvLematus *and* Lematus *make different types of mistakes?*

  The majority of Lematus mistakes were applying a reasonable rule, but the wrong rule. Most of the mistakes made by larger ConvLematus models, like `n3k3`, was also due to applying the wrong rule. Compared to Lematus, `n3k3` may have produced more odd lemmas (`strikesthroogh`) and `n1k2` produced a very large number of typos (`stroughrike`).

- *Does using convolutional encoders improve speed?*

  Yes! The average lemmatization time of ConvLematus is 53% the time Lematus takes, and average model training time score 55% that of Lematus.

- *Does the* ConvLematus *performance change relative to* Lematus *when introduced to a more natural distribution of words?*

  The `n3k3` models also trail Lematus in the ud10k-0ctx datasets. Its accuracy is several percentage points lower than Lematus for unseen inflected form across languages.

- *Can* ConvLematus *handle large distances between input sequence features and where they modify the output sequence?* and *Can* ConvLematus *learn rules other than morphology?*

  As no experiments disentangle these two questions, their exploration summary is presented together.

  `n3k3` is able to learn to use long-distance dependencies such as those in CS-POS. There is some evidence that suggests ConvLematus is more reluctant than Lematus to depend on the long-distance dependencies: when the POS tag was shuffled, the accuracy of Lematus models decreased more than the ConvLematus models.

8.2 Critique

If given more time, there are ways to improve this study:

- Rather than using the model that achieved the lowest validation loss, one should use the model that achieved the highest validation accuracy. Initial analysis of this is showed in section A.1, which shows an Arabic model that out-performs Bergmanis and Goldwater (2018) on ud10k-20ctx by increasing patience.

- Compare differences when using different sized datasets. Maybe Lematus or ConvLematus can learn better in low-resource settings.

- Rather than training multiple models for each configuration, in section 7.2 we hoped that by testing multiple very similar models, we could reveal a trend. This could still result in incorrect conclusions: perhaps the jump in performance between Finnish `kernel_size` = 4 and `kernel_size` = 5 had to do with unlucky initialization parameters.

- Some model hyperparameters, such as learning rate or early stopping patience, were not tuned per architecture. For example, `n1k1` might perform better given a higher learning rate.

- The timing metrics may be slightly unfair: Lematus was not built for speed.[1]

[1]Not that the convolutional encoder implementation was built for speed!

8.3 Future work

There are a few questions that would be interesting to explore.

- This dissertation approached the difference between models thinking about the receptive field difference between Lematus and ConvLematus. There are other differences such as Lematus's gating mechanisms.

  With the receptive field experiments, how do the number of parameters relate to accuracy? Deeper networks use a different number of parameters than shallower networks with the same receptive field.

- Why did Latvian perform so poorly? Why did small changes (adding POS tags, or a second layer) result in such large increases in accuracies?

- From the results, there almost appears to be an upper-bound on performance that Lematus achieves and ConvLematus are not able to improve upon. There are several reasons: a) it isn't really the upper-bound, and the convolutional encoders tested here were too simplistic, b) the attention and decoder configurations create a bottleneck c) the dataset is difficult to perform better on.

- Modifications of ConvLematus architecture should be explored. For example, what if a bidirectional encoder is added between the convolutional encoder and the attention mechanism? Does that completely improve performance? Additional variations of the convolutional encoder used in ConvLematus should be explored. In Appendix B, an initial exploration of batch normalization, highway layers, and convolutional layers with multiple kernel sizes are shown. Deeper networks could be explored. In Figure 7.8, it appears that using deeper convolutional encoders did not decrease performance by much relative to the cost of the recurrent encoder in Lematus. For this reason, very deep networks could be used in the encoder possibly without harming the performance boost (He et al., 2016). There are a huge number of other configurations that can be tested. For example, with deeper networks, the gated linear unit activation function (Dauphin et al., 2016) uses convolutions to set gating similar to the mechanism in GRUs. Dilated convolutions can increase the receptive field using fewer layers (Kalchbrenner et al., 2016). Even simpler configurations, like using different kernel sizes on different levels can be explored.

- The charts shown in Figure 7.6 are a useful way to summarize the kinds of mistakes lemmatizers made. Creating the rule-classifier by hand (as described in section 5.6) was time-consuming and required knowledge of the language, hence analysis was limited to English. However, automatic approaches for extracting “rules” from labeled datasets for lemmatization already exist (for example, the “edit scripts” from Chrupała et al. (2008)). These classifiers can be run on both the labeled datasets and the predicted lemmas to extract the “rules” applied in either case. Then the rules can be compared.

- Other types of attention could also be tested. For example, because the alignment between lemmas and their inflection is usually monotonic, an approach called hard monotonic attention was used to improve inflection accuracy (Aharoni and Goldberg, 2016). An approach like this could be transferred to constrain the attention and may produce better lemmatization.

Appendix A

Hyperparameter Tuning

A.1 Choosing a model from a training run

When choosing a model from a training run, we used the suboptimal approach of using the model that achieved the lowest validation loss and then measuring its validation accuracy. This increased experiment iteration speed at the cost that the best validation accuracy model was likely not reported and more noise was introduced.

For example, the reported UD10K results in subsection 7.3.3 are lower than those reported by Bergmanis and Goldwater (2018). While it may be due to the change in dataset (UDv2.1 vs UDv2.0) or the change of NEMATUS version (TensorFlow vs Theano), it's most likely due to the method of choosing a model from a training run. A set of experiments was carried out on a subset of languages that chose the model based on validation accuracy, as well as using `patience=20`. This is illustrated for a single model in Figure A.1, which shows how choosing a model can impact accuracy. For example, an easy way to beat the accuracy in Arabic UD10K-20CTX reported in Bergmanis and Goldwater (2018) is to increase the patience to 20. In a test that used patience = 20, an average of 3 runs of Arabic UD10K-20CTX achieved 79.5%, of 2 runs Finnish UD10K-20CTX achieved 76.9%, and 2 runs of Turkish achieved 88.0%. It's likely that the model selection process is what leads to the discrepancy between the numbers reported in subsection 7.3.3 and those in Bergmanis and Goldwater (2018).

Over experiments used in this dissertation, this can be demonstrated by comparing the accuracy of the final trained model (10 epochs after the model with the best validation loss). In the first 164 experiment runs, the accuracy of the

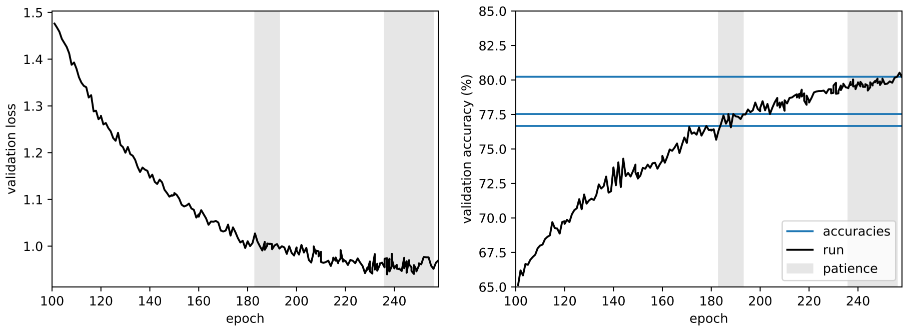


Figure A.1: Comparison of accuracy based on how the model is chosen. This shows training a Lematus model training on the Arabic ud10k-20ctx dataset. (Its result was not used in other sections of this dissertation.) The left image shows the validation loss as the black line. The first gray area on the left shows `patience=10`: the left boundary represents the epoch with the lowest validation accuracy and the right boundary is the 10 epochs that follow. The second gray area is the same but with `patience=20`. The right image shows the same areas highlighted, but the underlying black line is the validation accuracy evaluated at each epoch. The horizontal blue bars represent the three ways of choosing accuracy. The lowest blue bar is using the accuracy of the model with the lowest validation loss. This is the approach taken in this thesis. The middle bar represents the model with the best accuracy using `patience=10`. This is the approach used in Bergmanis and Goldwater (2018). In this case, the bar is higher because accuracy continued to increase after validation loss stopped. Finally, the top bar represents the model with the best accuracy using `patience=20`. This is because after validation loss was not decreasing for 10 epochs (triggering patience=10 to early stop), it started decreasing again between 10 and 20 epochs.

language	relu	tanh	last relu	last tanh	relu/tanh
Arabic	-5.3%	-5.9%	-6.3%	-3.2%	-2.6%
Finnish	-6.6%	-8.1%	-6.4%	-8.7%	-10.3%
Latvian	-6.3%	-8.3%	-5.8%	-6.5%	-7.7%
Russian	-2.7%	-3.0%	-3.9%	-3.5%	-2.3%
Turkish	-1.3%	-1.9%	-2.3%	-2.3%	-2.0%
Average	-4.4%	5.4%	-5.0%	-4.9%	-5.7%

Table A.1: Data used to choose activation configurations. This table aggregates the validation accuracy of 75 models trained to evaluate activation functions. Each column is a different configuration: "relu" means ReLU after every convolutional layer except the last, "last relu" is the same as "relu" but with a ReLU after the last layer, and "relu/tanh" is ReLU after every layer but the last, and a tanh after the last. Cells contain the average of three two-layer convolutional configuration (`n2k2`, `n2k3`, and `n2k4`), from which the average Lematus performance is subtracted (values are from Table 7.1). Less negative values mean models using the activation configuration performed better on average. While underlying data show different trends, the "relu" configuration performed the best by average according to this calculation.

final model was higher than the validation loss model by an average of 0.71% 99 times with a max improvement of 2.8%, while the validation loss model was higher than the final model by an average of 0.65% and a max improvement of 3.5%.

A.2 Activation Configurations

Five activation configurations were tested: ReLU or tanh after every but the last convolution layer, ReLU or tanh after every layer, and ReLU on after every layer but the last and use tanh on the output of the last layer. Activation configurations were tested on 2 layer models of kernel sizes 2, 3, and 4, for all languages except English. The experiment results are shown in Table A.1. After normalizing language by the performance of Lematus (to approximately correct for more difficult datasets), the best average across languages was -3.8%% for ReLU. The others were worse than -4.2%.

Appendix B

Preliminary explorations

B.1 Batch Normalization

Towards the end of this analysis, a set of experiments on two configurations of ConvLematus using batch normalization was carried out. In these experiments, Nematus's "sort maxibatches" (which creates minibatches based on input sequence length for performance) was turned off so that the distribution within each batch was not biased by sentence length. The validation accuracy is shown in Table B.1. Overall, there were mixed results when adding batch normalization. Batch normalization improved deeper networks (`n5k3`) over no batch normalization, though batch normalized `n5k3` almost always had lower accuracies than un-batch-normalized `n4k3` models. The one exception was that the best-performing ConvLematus configuration for Latvian over all experiments in this thesis was using with batch-normalization `n4k3`. The other consideration is speed. The two measures of speed from subsection 7.3.2 were computed for the models with batch normalization: *Training speed per epoch* and *Training time score* (*Translation time* was not computed). The two metrics are shown in Table B.1. The results show that while each individual epoch takes longer, the number of epochs needed to converge is much lower (on average, $\approx$ 65 epochs with batch normalization and instead of $\approx$ 143 without.) Because the results on average decreased validation accuracy and did not shift most conclusions (the batch normalized deeper networks did not clearly start outperforming the shallower ones), the experiments in Figure 7.4 were not rerun to include batch normalization.

	`n4k3`		`n5k3`	
Language	Yes	No	Yes	No
Arabic	80.4%	**83.3%**	**80.7%**	79.4%
English	91.7%	**93.6%**	**94.1%**	85.5%
Finnish	77.1%	**80.5%**	76.8%	**81.5%**
Latvian	**80.4%**	72.4%	**79.4%**	75.2%
Russian	77.1%	**81.3%**	71.9%	**81.0%**
Turkish	91.8%	**92.6%**	**93.4%**	92.1%

Table B.1: Effect of adding batch normalization to ConvLematus on validation accuracy. For each architecture and language, the table shows two models that are identical except for the use of batch normalization between convolutional layers.

Language	No	Yes
Words/sec	**412**	295
Training speed score	3.8k	**2.0k**

Table B.2: Effect of adding batch normalization to ConvLematus on speed. The metrics are described in detail in subsection 7.3.2. Note that this table uses the median of the 10 models shown in Table B.1 in order to be comparable to the box-plots in Figure 7.8. Bold indicates more ideal for the row.

B.2 Highway Layers

As an ablation study of KIM (Kim et al., 2016), this set of experiments uses a single convolutional layer and passes the output into a single highway layer (Srivastava et al., 2015). In the case of our experiments, the convolutional layers used ReLU after the convolutional layer and a tanh in the highway layer. Note that this is accidentally the opposite of the KIM set-up (KIM is tanh after convolutional layers and a ReLU in the highway layer).

B.2.1 Theory

This section extends the theory in chapter 3. Highway layers (Srivastava et al., 2015) apply a gated dense layer over the filters of each timesteps' vector. Highway layers learn two sets of weights. The first, $W_h \in \mathbb{R}^{\texttt{filter\_count} \times \texttt{filter\_count}}$ is a transform. The second W_d (of the same size) is used as part of a gate that determines whether to return the original or transformed layer.

$$Y' = \sigma(W_d X')W_h X' + (1 - \sigma(W_d X'))X'$$

where X' is the input and σ is the logistic sigmoid function.

In a way, a highway layer is like adding another convolutional layer with a `kernel_size` = 1 convolution, but with an additional gating mechanism. If $\sigma(W_d X) = 1$, these are the same as a convolution with a kernel of length 1. If $\sigma(W_d X) = 0$, these are the same as when no layer is present. If we ignore the gating mechanism, theoretically we'd expect convolutions with a highway (`n1kx`+hw) to perform better than a single convolutional layer (`n1kx`) (because it can do everything `n1kx` can do, plus combine filters), but worse than two layers of kernels with the same size (`n2kx`) (because we're ignoring the gating mechanism, and the second layer should be able to learn the same relationships the highway can, as well as expanding the receptive field).

B.2.2 Results

The results are shown in Table B.3. Slightly more often than not, the `n1kx`+hw networks outperformed the `n1kx` networks. In 4 of the 18 language-kernel pairs, the `n1kx`+hw also outperformed the corresponding `n2kx`, though in 2 language-kernel pairs, the `n1kx` outperformed `n2kx`. There are a few other considerations:

	`n1k3`		`n1k4`		`n1k5`	
	no HW	HW	no HW	HW	no HW	HW
Arabic	**82.9%**	82.6%	***85.1%**	82.6%	81.9%	**82.5%**
English	**93.6%**	92.5%	92.5%	**93.2%**	93.7%	***94.0%**
Finnish	66.2%	**79.7%**	78.2%	**79.1%**	**78.1%**	77.6%
Latvian	74.9%	***77.4%**	71.3%	***77.3%**	72.2%	**76.4%**
Russian	80.7%	***82.2%**	***82.5%**	81.6%	79.9%	**80.2%**
Turkish	90.9%	**91.3%**	**92.2%**	92.0%	**93.3%**	92.7%

Table B.3: Effect of adding highway layers to convolutional encoders. *Indicates outperforming two layers of the same kernel size (data for two-layer networks are in Figure 7.4).

the highway layers only adds about 120k parameters, while the second convolutional layer adds on average 200k parameters. For speed (see subsection 7.3.2), the implementations used of all three (`n1kx`, `n1kx`+hw, `n2kx`) processed on average within 6 words per second of each other (with `n1kx`+hw best at 424, `n1kx` worse at 418).

B.3 Convolutional layers of multiple kernel sizes

As an ablation study of Kim (Kim et al., 2016), ConvLematus models were trained using a similar convolutional layer.

B.3.1 Theory

Continuing from the convolutional layer theory outlined in section 3.2, Kim concatenated the output of convolutional layers with different kernel sizes. For example $\mathbf{y}_t'^{(1)} = (X' * W^{(1)})_t$ could use $W^{(1)}$ with `kernel_size` $= 1$, $\mathbf{y}_t'^{(2)}$ `kernel_size` $=$ 2, and $\mathbf{y}_t'^{(3)}$ `kernel_size` $= 3$. The convolutional layer's output at timestep t is given by concatenating along the filters so that the layer returns another vector

$$\mathbf{y}_t' = \left[\mathbf{y}_t'^{(1)} \quad \mathbf{y}_t'^{(2)} \quad \mathbf{y}_t'^{(3)}\right]. \tag{B.1}$$

For example, if the three kernel sizes are used with filters of sizes f_1, f_2, f_3, then the output $\mathbf{y}_t' \in \mathbb{R}^{f_1+f_2+f_3}$.

Language	`n1k6`	Kim
Arabic	81.9%	**82.6%**
Finnish	78.1%	**79.9%**
Latvian	72.2%	**78.1%**
Russian	79.9%	**82.3%**
Turkish	**93.3%**	91.5%

Table B.4: Comparison of Kim filter to `n1k6`.

B.3.2 Experiment

For this first experiment, to fit within the 200-filter-limit, the number of filters for each kernel size used by Kim was shifted down. The new filter count for kernel sizes [1, 2, 3, 4, 5, 6] is [15, 15, 20, 40, 50, 60].

The results are shown in Table B.4, along with the results from `n1k6` model. In theory, the `n1k6` model should be able to learn everything that the Kim model can by zeroing out parameters. The results from Kim can also be compared to those in Figure 7.2.

B.3.3 Results

All languages except Turkish performed better than `n1k6`. Latvian achieved an impressive 5.9% improvement, though it still has a lower accuracy than the Lematus model. This improvement may be because it is easier for the model to learn smaller features (similar to how the Latvian `n2k3` performed much better than most other Latvian models.)

Appendix C

Additional Tables and Figures

C.1 Measuring "xor-count" of Lematus models

This section is supplemental for subsubsection 7.3.1.2. It provides numbers for the baseline of "xor-count", a measure of the difference between a pair of models. As in subsubsection 7.3.1.2, the "xor-count" is the number of words only one of a pair of models correctly lemmatized. Three LEMATUS models were trained using different seeds. The values in Table C.1 show the range of the number of differences in incorrectly lemmatized words between different LEMATUS models.

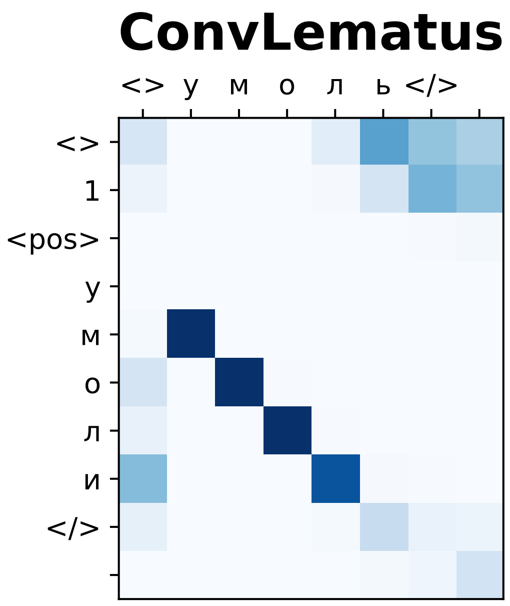
Let us define xor-count of a pair of models as the number of words that only one model lemmatized correctly (i.e., the number represented by the combination of the dark blue and dark gray boxes in Figure 7.7). As evident by Table 7.1, identical models trained with distinct random seeds can produce different lemmatizations. If we want to suggest that the CONVLEMATUS and LEMATUS models are producing different lemmatizations due to their architecture, we should show that xor-count is higher than what we would get for two models that use identical architectures but different seeds. First, we can compute the xor-count for pairs of LEMATUS models trained with different seeds. The values in Table C.1 serve as a baseline for the xor-count of two models that actually share an architecture. We can compare the baseline xor-count to the xor-count between pairs of CONVLEMATUS and LEMATUS models shown in Figure 7.7. Given a baseline of xor-count, we can count the number of models that have a xor-count greater than the maximum observed in a baseline. This is given in subsubsection 7.3.1.2.

Arabic	English	Finnish	Latvian	Russian	Turkish
77 - 86	24 - 27	72 - 89	86 - 114	51 - 64	44 - 56

Table C.1: Range of differences between baseline models. For each language, a "xor-count" of each of the three possible pairs of baselines is computed (i.e. given Lematus models 1, 2, and 3, three pairs were (1, 2), (2, 3), and (3, 1)). The range across the three "xor-counts" of the number of differences is shown in the table. The values in the above table are comparable to the number of words that both dark regions in Figure 7.7 together represent. (Note that this number does not take into account differences in words that both models lemmatized incorrectly.)

C.2 Additional Attention plots

Figure C.1 shows another example of the CS-POS, using the same model as Figure 7.9.



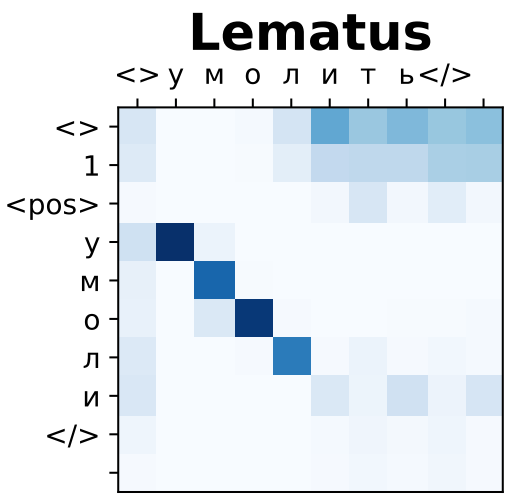


Figure C.1: Attention plots for a second Russian word from CS-POS Like Figure 7.9 but for a different inflected form. Also, ConvLematus incorrectly lemmatizes.

Bibliography

Aharoni, R. and Goldberg, Y. (2016). Sequence to sequence transduction with hard monotonic attention.

Bahdanau, D., Cho, K., and Bengio, Y. (2015). Neural machine translation by jointly learning to align and translate. *ICLR'2015 arXiv:1409.0473.*

Bergmanis, T. and Goldwater, S. (2018). Context sensitive neural lemmatization with lematus.

Bradbury, J., Merity, S., Xiong, C., and Socher, R. (2016). Quasi-recurrent neural networks. *arXiv preprint arXiv:1611.01576.*

Chakrabarty, A., Pandit, O. A., and Garain, U. (2017). Context sensitive lemmatization using two successive bidirectional gated recurrent networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 1481–1491.

Cho, K., Van Merriënboer, B., Bahdanau, D., and Bengio, Y. (2014a). On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259.*

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014b). Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078.*

Chrupala, G. (2006). Simple data-driven context-sensitive lemmatization. *Procesamiento del Lenguaje Natural*, (37).

Chrupała, G., Dinu, G., and Van Genabith, J. (2008). Learning morphology with morfette.

Conneau, A., Schwenk, H., Barrault, L., and Lecun, Y. (2016). Very deep convolutional networks for text classification. *arXiv preprint arXiv:1606.01781*.

Costa-Jussà, M. R. and Fonollosa, J. A. R. (2016). Character-based neural machine translation. *CoRR*, abs/1603.00810.

Cotterell, R., Kirov, C., Sylak-Glassman, J., Walther, G., Vylomova, E., Xia, P., Faruqui, M., Kübler, S., Yarowsky, D., Eisner, J., and Hulden, M. (2017). The CoNLL-SIGMORPHON 2017 shared task: Universal morphological reinflection in 52 languages.

Dauphin, Y. N., Fan, A., Auli, M., and Grangier, D. (2016). Language modeling with gated convolutional networks. *arXiv preprint arXiv:1612.08083*.

Denil, M., Demiraj, A., Kalchbrenner, N., Blunsom, P., and de Freitas, N. (2014). Modelling, visualising and summarising documents with a single convolutional neural network. *arXiv preprint arXiv:1406.3830*.

dos Santos, C. and Gatti, M. (2014). Deep convolutional neural networks for sentiment analysis of short texts. In *Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers*, pages 69–78.

Gehring, J., Auli, M., Grangier, D., Yarats, D., and Dauphin, Y. N. (2017). Convolutional sequence to sequence learning. *arXiv preprint arXiv:1705.03122*.

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256.

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

Graves, A. and Schmidhuber, J. (2005). Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural Networks*, 18(5-6):602–610.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

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

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167.*

Jivani, A. G. et al. (2011). A comparative study of stemming algorithms. *Int. J. Comp. Tech. Appl*, 2(6):1930–1938.

Jurafsky, D. and Martin, J. H. (2014). *Speech and language processing*, volume 3. Pearson London:.

Kalchbrenner, N. and Blunsom, P. (2013). Recurrent continuous translation models. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1700–1709.

Kalchbrenner, N., Espeholt, L., Simonyan, K., Oord, A. v. d., Graves, A., and Kavukcuoglu, K. (2016). Neural machine translation in linear time. *arXiv preprint arXiv:1610.10099.*

Kalchbrenner, N., Grefenstette, E., and Blunsom, P. (2014). A convolutional neural network for modelling sentences. *arXiv preprint arXiv:1404.2188.*

Karlsson, F. (2015). *Finnish: An essential grammar.* Routledge, 3 edition.

Kestemont, M., De Pauw, G., van Nie, R., and Daelemans, W. (2016). Lemmatization for variation-rich languages using deep learning. *Digital Scholarship in the Humanities*, 32(4):797–815.

Kim, Y. (2014). Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882.*

Kim, Y., Jernite, Y., Sontag, D., and Rush, A. M. (2016). Character-aware neural language models. In *AAAI*, pages 2741–2749.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980.*

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.

Lee, J., Cho, K., and Hofmann, T. (2016). Fully character-level neural machine translation without explicit segmentation. *arXiv preprint arXiv:1610.03017*.

Matthews, P. H. (1991). *Morphology (Cambridge Textbooks in Linguistics)*. Cambridge University Press, 2 edition.

Müller, T., Cotterell, R., Fraser, A., and Schütze, H. (2015). Joint lemmatization and morphological tagging with lemming. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 2268–2274.

Nivre, J., Agić,
v Z., Ahrenberg, L., Antonsen, L., Aranzabe, M. J., Asahara, M., Ateyah, L., et al. (2017a). Universal dependencies 2.1. LINDAT/CLARIN digital library at the Institute of Formal and Applied Linguistics (
'UFAL), Faculty of Mathematics and Physics, Charles University.

Nivre, J., Zeljko Agic, L. A., et al. (2017b). Universal dependencies 2.0 conll 2017 shared task development and test data. lindat/clarin digital library at the institute of formal and applied linguistics, charles university.

Östling, R. (2016). Morphological reinflection with convolutional neural networks. In *Proceedings of the 14th SIGMORPHON Workshop on Computational Research in Phonetics, Phonology, and Morphology*, pages 23–26.

Sennrich, R., Firat, O., Cho, K., Birch, A., Haddow, B., Hitschler, J., Junczys-Dowmunt, M., Läubli, S., Miceli Barone, A. V., Mokry, J., and Nadejde, M. (2017). Nematus: a toolkit for neural machine translation. In *Proceedings of the Software Demonstrations of the 15th Conference of the European Chapter of the Association for Computational Linguistics*, pages 65–68, Valencia, Spain. Association for Computational Linguistics.

Sennrich, R., Haddow, B., and Birch, A. (2016). Edinburgh neural machine translation systems for wmt 16. *arXiv preprint arXiv:1606.02891*.

Srivastava, R. K., Greff, K., and Schmidhuber, J. (2015). Training very deep networks. In *Advances in neural information processing systems*, pages 2377–2385.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2015). Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9.

Van Den Oord, A., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A., and Kavukcuoglu, K. (2016). Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*.

Vania, C. and Lopez, A. (2017). From characters to words to in between: Do we capture morphology? *arXiv preprint arXiv:1704.08352*.

Xingjian, S., Chen, Z., Wang, H., Yeung, D.-Y., Wong, W.-K., and Woo, W.-c. (2015). Convolutional lstm network: A machine learning approach for precipitation nowcasting. In *Advances in neural information processing systems*, pages 802–810.

Zhang, X., Zhao, J., and LeCun, Y. (2015). Character-level convolutional networks for text classification. In *Advances in neural information processing systems*, pages 649–657.

Zhou, C., Sun, C., Liu, Z., and Lau, F. (2015). A c-lstm neural network for text classification. *arXiv preprint arXiv:1511.08630*.