Universität Zürich^{UZH}

Masterarbeit

zur Erlangung des akademischen Grades

Master of Science

der Wirtschaftswissenschaftlichen Fakultät der Universität Zürich

A Neural Transducer in PyTorch: Efficient Mini-Batching for Large-Scale Training, Transformer Encoders, and Batched Decoding

Verfasser: Silvan Wehrli

Matrikel-Nr: 15-703-275

Referent: Prof. Dr. Martin Volk

Betreuer: Dr. Simon Clematide, Peter Makarov

Institut für Computerlinguistik, Institut für Informatik

Abgabedatum: 22.07.2022

Abstract

Neural methods have shown great success in string transduction tasks such as grapheme-to-phoneme conversion or morphological inflection. A particularly successful class of models for these tasks are recurrent neural transducers that use an encoder-decoder architecture to predict character-level edit actions. This work builds on such a neural transducer. Despite its ongoing success, this particular approach offers room for improvement: The model uses an outdated software framework, and the implementation is tailored toward the use on a CPU. This leads to low training efficiency preventing the application to large datasets. Moreover, the fully recurrent structure of the model does not reflect recent technological developments, as the non-recurrent transformer architectures have become dominant in many areas of natural language processing.

This thesis addresses these shortcomings by reimplementing the model using the machine learning framework PyTorch, implementing GPU-supported mini-batch training and batched greedy decoding, as well as adding support for transformer-based encoders. Experimental results on standard datasets confirm the successful reimplementation. Top rankings in the SIGMORPHON 2022 Shared Task on Morpheme Segmentation (featuring training sets of up to 750,000 samples) demonstrate the model's ability to scale: GPU training and greedy decoding are up to 250 times, respectively, 10 times faster. While transformer-based encoders rarely outperformed recurrent encoders, the initial experiments in this work lay the foundation for further experimentation.

Zusammenfassung

Neuronale Methoden haben sich bei der Umwandlung (engl. *transduction*) von Zeichenketten (engl. *strings*), wie derjenigen von Graphemen in Phoneme oder der morphologischen Flexion, als sehr erfolgreich erwiesen. Eine besonders erfolgreiche Klasse von Modellen für diese Aufgaben sind rekurrente neuronale Transducer, die eine Kodierer-Dekodierer-Architektur (engl. *encoder-decoder architecture*) zur Vorhersage von konkreten Änderungen (engl. *edit actions*) auf Zeichenebene verwenden. Die vorliegende Arbeit baut auf einem solchen Transducer auf. Trotz seines anhaltenden Erfolgs bietet dieser konkrete Ansatz Potential für Verbesserungen: Das Modell verwendet ein veraltetes Software-Framework, und die Implementierung ist auf den Einsatz auf einem CPU zugeschnitten. Dies führt zu einer geringen Trainingseffizienz, was die Anwendung auf grosse Datensätze verhindert. Ausserdem spiegelt die vollständig rekurrente Struktur des Modells nicht neuere technologische Entwicklungen wieder, da nicht-rekurrente Transformer-Architekturen in Natural Language Processing für viele Aufgaben vorherrschend geworden sind.
Diese Arbeit befasst sich mit diesen Schwächen, indem das Modell in dem Machine-Learning-Framework PyTorch reimplementiert wird, GPU-gestütztes Mini-Batch-Training und Batched Greedy Decoding implementiert werden und das Modell um Support für Transformer-basierte Kodierer (engl. *encoders*) erweitert wird. Experimentelle Ergebnisse mit Standard-Datensätzen bestätigen die erfolgreiche Neuimplementierung. Spitzenplatzierungen im *SIGMORPHON 2022 Shared Task on Morpheme Segmentation* (mit Trainingsdatensätzen von bis zu 750'000 Beispielen) demonstrieren die Skalierbarkeit des Modells: Training und Greedy Decoding auf einem GPU sind bis zu 250 respektive 10 Mal schneller. Während Transformer-basierte Kodierer nur in wenigen Fällen besser abschnitten als rekurrente Kodierer, bilden die experimentellen Ergebnisse dieser Arbeit die Grundlage für weitere Experimente.

Acknowledgement

I would like to thank my supervisors, Simon Clematide and Peter Makarov, for their ongoing support throughout this thesis. Their technical and conceptual expertise in NLP has guided me and helped me to shape my research. The many NLP courses I took with Simon Clematide sparked my interest in NLP. He also motivated me to participate in a shared task for the first time a while ago. I learned a lot! I would also like to point out Peter Makarov's reviews of my code, which helped me to focus on what is relevant. I am happy that we could create competitive submissions to the SIGMORPHON 2022 shared tasks on morpheme segmentation and morphological inflection based on the work of this thesis. I would also like to thank Marie for proofreading the present text and challenging the comprehensibility of my work.

Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

CLI	command-line interface
G2P	grapheme-to-phoneme conversion
IL	imitation learning
IPA	International Phonetic Alphabet
NLP	natural language processing
NMT	neural machine translation
PER	phone error rate
POS	part of speech
seq2seq	sequence-to-sequence
SED	Stochastic Edit Distance
SOTA	state-of-the-art
WER	word error rate
WFST	weighted finite-state transducer

1 Introduction

1.1 Morphological and Phonological String Transduction Tasks

As in many areas of natural language processing (NLP), neural methods have also been successfully applied to morphological and phonological string transduction tasks. Loosely speaking, such tasks deal with transforming single words from or to a specific morphological or phonological form. A typical phonological string transduction task is grapheme-to-phoneme conversion (G2P). In this task, a sequence of characters (input) is mapped to a sequence of transduction symbols (output), representing the input's pronunciation [Black et al., 1998; Jiampojamarn et al., 2007]. Table 1.1 shows an example of this task. The table also shows examples of morphological tasks: Morphological inflection and morpheme segmentation. The former is the task of generating a target word form, given a source word form and the target word form's morpho-syntactic features, e.g., its part of speech (POS) or gender [Cotterell et al., 2016, 2017]. The latter is the task of converting a word into a sequence of morphemes [Creutz and Lagus, 2002]. These tasks typically have applications in other downstream NLP tasks. Morphological inflection, for instance, is used in neural machine translation (NMT) systems, and G2P is an important building block in text-to-speech or speech-to-text systems [Rao et al., 2015; Aharoni and Goldberg, 2017].

Task	**Input**	**Output**
INFL	sue V;PST	sued
G2P	abandonner	a b ɑ̃ d ɔ n e
MS	hierarchyisms	hierarch @@y @@ism @@s

Table 1.1: Examples for morphological inflection (INFL), grapheme-to-phoneme conversion (G2P) and morpheme segmentation (MS).

1.2 Neural Transducers with Edit Actions

One class of models has been particularly successful in these tasks: The neural transducer, which models the output string as a sequence of edit actions based on the input string. Freely speaking, these models transduce a sequence of input characters into a sequence of output characters by traversing the input sequence stepwise. At each step of the process, the model produces a single edit action based on the model's current state. The edit action of the previous step and an aligned character from the input string describe this model state. What edit actions are possible depends on the specific model. Typically, this includes insertions (adding a character), deletions (removing a character from the input sequence), and substitutions (replacing a character from the input sequence). These edit actions serve two purposes: Firstly, they transduce a character in the input sequence into a character in the output sequence. Secondly, they decide on the alignment of the input character used in the modelling of edit actions (altering the model state). Table 1.2 gives an illustrative example of this process. It shows the G2P transduction of the French word *abandonner*. At each step, the model produces an edit action. Depending on the action, the output is extended (in the case of a COPY or INSERT action) and the alignment pointer in the input sequence is moved (in the case of a DELETE or COPY action).[1]

Step	Edit action	Input alignment	Output
1	COPY(a)	**a** b a n d o n n e r	a
2	COPY(b)	a **b** a n d o n n e r	a b
4	DELETE(a)	a b **a** n d o n n e r	a b ã
3	INSERT(ã)	a b a **n** d o n n e r	a b ã
4	DELETE(n)	a b a **n** d o n n e r	a b ã
5	COPY(d)	a b a n **d** o n n e r	a b ã d
6	DELETE(o)	a b a n d **o** n n e r	a b ã d
7	INSERT(ɔ)	a b a n d o **n** n e r	a b ã d ɔ
8	COPY(n)	a b a n d o **n** n e r	a b ã d ɔ n
9	DELETE(n)	a b a n d o n **n** e r	a b ã d ɔ n
10	COPY(e)	a b a n d o n n **e** r	a b ã d ɔ n e
11	DELETE(r)	a b a n d o n n e **r**	a b ã d ɔ n e

Table 1.2: The G2P transduction process for the French word *abandonner*. Bold and underlined **letters** mark the aligned input character of the current transducer step.

Inductive bias The success of these transducers has many reasons. First of all, the way of solving the problem is based on an inductive bias known to work well

[1]A more comprehensive and theoretical introduction is given in Chapter 2.

for string transduction tasks: Transduction is modelled as a monotonic one-to-one mapping between input and output characters [Makarov and Clematide, 2018a; Wu and Cotterell, 2019; Rios et al., 2021]. Table 1.2 demonstrates this. At each step, precisely one input character is aligned, and the characters are consumed strictly from left to right (monotonic).

Data scope What is more, data requirements are comparatively low compared to other NLP tasks such as NMT (hundreds versus millions of training samples). As such, data scalability is typically not an essential requirement for string transduction tasks [Wu et al., 2021]. This is handy as these models typically do not allow for the parallelization of computations needed to speed up training and inference time. These models commonly use recurrent neural structures (such as RNNs or LSTMs), which only allow for limited parallelization by their design. More importantly, efficient training and inference procedures for transition-based systems such as neural transducers may be unequally hard to parallelize [Ding and Koehn, 2019; Noji and Oseki, 2021], effectively decreasing efforts to implement suitable solutions.

Interpretability Lastly, modelling explicit edit actions offers interpretability. The model's output, a sequence of edit actions, resembles a human's problem-solving strategy.

1.3 Motivation

Makarov and Clematide [2018a] present such an approach for a neural transducer. Their recurrent neural-transition based model uses imitation learning to learn explicit edit actions (deletion, copy, and insertion) derived from an expert policy. Variations of this approach have reached top rankings in various shared task submissions. This includes the CoNLL-SIGMORPHON 2018 Shared Task on Universal Morphological Reinflection [Makarov and Clematide, 2018c] as well as the SIGMORPHON 2020 and 2021 Shared Tasks on Multilingual G2P [Makarov and Clematide, 2020; Clematide and Makarov, 2021]. Despite their recent and ongoing success, the approach can be improved on various levels in terms of actuality.

Software actuality Firstly, the software uses DyNet [Neubig et al., 2017]. DyNet is a software toolkit that can be used for building neural architectures. However, the development has effectively stopped.[2] An up-to-date and widely-used software framework is a key requirement for sustainable and reliable software development. Such a framework ensures that the developer community adequately addresses software bugs and that new software architectures are timely implemented. With regard to the ever-changing field of NLP, the latter is especially important.

Model actuality Secondly, the use of recurrent model architecture does not reflect more recent developments in the field of NLP, such as the use of transformers [Vaswani et al., 2017]. Generally, transformers have shown great success in various NLP areas such as language modelling [Devlin et al., 2019] or question answering [Yang et al., 2019]. Wu et al. [2021] show that such architectures can also be applied successfully to morphological tasks. It stands to reason that the here discussed approach may also benefit from such architecture.
Transformers also provide a good example for the previously discussed software actuality. As of now, the DyNet framework does not offer any transformer-based implementations according to its documentation[3], and community-driven projects seem inexistent. Other frameworks such as PyTorch [Paszke et al., 2019] fill this gap: PyTorch provides own implementations[4], and community-driven projects such as Hugging Face[5] make up-to-date and research-related models available as open source.

Data scalability Thirdly, the discussed approach of Makarov and Clematide [2018a] lacks data scalability for the same reasons as previously discussed (recurrent architecture, complex implementation for training and inference procedure). While this may not be a particularly critical issue in morphological tasks, the recent SIGMORPHON 2022 Shared Task on Morpheme Segmentation requires models with a capacity to process higher data volumes [Batsuren et al., 2022]: Training data sets contain up to 750,000 samples. This is in stark contrast to, for instance, the SIGMORPHON 2020 Shared Task on Multilingual G2P with 3,600 training samples per language [Gorman et al., 2020]. Importantly, data scalability may also be a requirement for the successful application of novel neural architectures (transformers), e.g., Wu et al. [2021] state that the success of their transformer-based model largely depends on the batch size.

[2]Evidence of this is the activity in their GitHub repository (https://github.com/clab/dynet). Commits are rather rare and concentrate on documentation and deployment/installation issues.

[3]https://dynet.readthedocs.io/en/latest

[4]e.g. https://pytorch.org/docs/stable/generated/torch.nn.Transformer.html

[5]https://huggingface.co

1.4 Research Questions

This thesis builds on the successful work on neural transducers by Makarov and Clematide [2020] used for morphological and phonological tasks. The focus lies on contributions to overcome the previously discussed challenges (software actuality, architecture actuality, and data scalability) by further developing an existing neural transducer model. The model used by Makarov and Clematide [2020] serves as a starting point. This model was successfully used for the SIGMORPHON 2020 shared task on multilingual G2P and served one year later as a baseline for the 2021 version of the same shared task [Gorman et al., 2020; Ashby et al., 2021]. The model is a variation of Makarov and Clematide [2018a] and is, therefore, an appropriate starting point. Given this model and the identified challenges (Section 1.3), this thesis aims to answer the following research questions.

1.4.1 Can the Model Be Ported from DyNet to PyTorch with Feature Parity?

To tackle this question, the baseline model [Makarov and Clematide, 2020] is rewritten using PyTorch [Paszke et al., 2019]. The idea is to produce a conceptually equivalent model (feature parity) that performs equally. The PyTorch-based reimplementation serves two purposes. It updates the existing model to use an up-to-date framework and builds the basis for additional features developed using PyTorch (as discussed in the next questions).

Choosing PyTorch is motivated by multiple reasons: Firstly, it is widely used in academia. An analysis of *Papers with Code* shows that, as of the end of June 2022, around 60% of papers with implementations use PyTorch.[6] Secondly, the development of the PyTorch framework is backed by a large community with regular software updates.[7] Thirdly, the framework offers a wide range of components for data handling, model architecture, and training/inference optimization facilitating the development and use of machine learning models.[8]

The reimplementation is evaluated using recent shared task datasets on G2P by comparing results with the baseline model.

[6]https://paperswithcode.com/trends

[7]In the first half of 2022, the framework has received two minor updates (1.11 and 1.12) with a wide range of new features and bug fixes (https://github.com/pytorch/pytorch/releases).

[8]Chapter 3 looks at some of these aspects in more detail.

1.4.2 Can the Model Scale to Higher Data Volume (or Is It Technically or Conceptually Bound to Lower Settings)?

Two aspects of the model are targeted to increase data scalability: Firstly, the approach's training routine is adapted to enable mini-batch training, i.e., the simultaneous training of multiple samples (batches). Secondly, greedy decoding used during inference is also leveraged to process batches. Importantly, mini-batch training and batched greedy decoding should provide GPU support to make full use of parallelization.

It is evaluated with respect to the baseline model examing whether the implemented changes lead to performance changes. Additionally, the implementation is competitively challenged as submissions to this year's SIGMORPHON 2022 shared tasks on morpheme segmentation and morphological inflection.

1.4.3 Can More Recent Software Architectures Improve Performance? More Concretely, Can a Transformer Encoder Beat an LSTM Encoder?

This question is addressed by enabling the architecture to use a transformer-based encoder. The choice of transformer is inspired by the wide and successful application of transformers in NLP and the more recent success in morphological tasks.

The implementation for the transformer encoder follows the original transformer architecture of Vaswani et al. [2017], supported by the success of Wu et al. [2021]. Generally, different (transformer-based) encoders are possible, or decoders, for that matter. However, a comprehensive analysis of encoder (and decoder) architectures is out of the scope of this thesis. The work in this thesis aims to provide initial results opening up the field for more experiments in the future. For this purpose, the implementation's software design aims to facilitate adding more encoder architectures.

The evaluation specifically compares the performance to the so far used LSTM-based encoder on shared task datasets.

1.5 Thesis Structure

In the following, I provide a short overview of the following chapters and their relations to each other.

Chapter 2 This chapter serves two purposes: Firstly, it introduces the reader to the related literature. Secondly, it familiarizes the reader with the theoretical background of the neural transducer model used in this thesis. More precisely, Section 2.1 discusses the various approaches used for string transduction tasks. Special attention is given to recent neural approaches and how they relate. Section 2.2 is dedicated to the neural transducer approach used in this thesis.

Chapter 3 Here, the design and implementation of the various adaptations to the baseline model are discussed. The discussion is related to the previously established research questions and is similarly structured. Section 3.1 highlights important aspects of the baseline model's code base, and Section 3.2 covers the documentation of the source code. Section 3.3 addresses the first research question and discusses the reimplementation of the model in PyTorch. Based on the work discussed in Section 3.3, Section 3.4 covers the work for implementing mini-batch training and batched greedy decoding. The implementation of transformer encoders is addressed in Section 3.5. Section 3.6 summarizes other noteworthy aspects of the implementation.

Chapter 4 This chapter evaluates the work as discussed in Chapter 3. Section 4.1 introduces the datasets used in the experiments (Section 4.2 lists the corresponding metrics). The evaluation of the experiments is structured similarly to Chapter 3: Section 4.3 focuses on the PyTorch reimplementation, Section 4.4 on the mini-batch training implementation and Section 4.5 on evaluating transformer-based encoders.

Chapter 5 Finally, this chapter concludes the thesis. It discusses the main results of this thesis and identifies future work.

2 Related Work and Background

2.1 Related Work

Traditionally, weighted finite-state transducers (WFST) were used to solve string transduction tasks such as inflection generation [Dreyer et al., 2008; Cotterell et al., 2014]. In the context of language, such transducers model a mapping between an input and output sequence of characters. They are weighted because each transition carries a weight. These weights are derived from the data with statistical techniques (or set manually) and allow to account for uncertainty in the data [Mohri, 2004]. WFSTs inherently model the alignment between characters in input and output as monotonic. This is a fitting property as many morphological string-to-string tasks mostly consist of monotonic transduction.

Following the general shift from non-neural to neural approaches in NLP, neural approaches also started being applied to morphological tasks. Yao and Zweig [2015] propose a sequence-to-sequence (seq2seq) encoder-decoder model for G2P. They evaluate neural models without and with explicit character alignment. They find that, while neural models without alignment come close to previous (non-neural) best results, explicit character alignment between encoder and decoder is needed to beat said results. In a similar attempt, Faruqui et al. [2016] experiment with seq2seq models for morphological inflection generation. Kann and Schütze [2016b] go in a similar direction. However, they use soft attention between their encoder and decoder, allowing the modelling of soft character alignment. As proposed by Bahdanau et al. [2016], the soft attention mechanism models a weighted representation of all input character embeddings at each decoding step. The decoder then uses this representation to produce the output character. Importantly, these weights are learned during training. The idea is that the model automatically learns alignment from the training data. While this soft attention mechanism resembles alignment, it is not the same conceptually: Alignment defines a vector of hidden variables, and attention defines a learned parametric feature function. In any case, the work of Kann and Schütze [2016b] has identified important gaps in previous work. In fact, their submission [Kann and Schütze, 2016a] ranked best in the SIGMORPHON 2016 Shared

Task on Morphological Reinflection [Cotterell et al., 2016], indicating the importance of some alignment mechanism.
Aharoni and Goldberg [2017] emphasise (monotonic) alignment in their encoder-decoder model, inspired by the inherent alignment between words and their inflected forms. Their approach models alignment as a pointer (to the input string). While less explicit approaches such as Kann and Schütze [2016b] can reach high performance, their seq2seq architecture is comparatively more data-intensive as the model must learn the alignment (expressed by attention) on its own. This makes the approach by Aharoni and Goldberg [2017] more data efficient and leads to comparatively higher performance in low data settings. However, their method relies on gold action sequences computed by an external aligner. During training, the model is then optimized based on these sequences. This setup promotes well-known issues with this kind of training: On the one hand, the model is never exposed to its own mistakes during training (exposure bias). On the other hand, the model is optimized based on token-level loss, while it is evaluated on sequence-level loss (loss-evaluation mismatch). Both phenomena can affect performance negatively [Wiseman and Rush, 2016].
Makarov and Clematide [2018b] build on the idea of Aharoni and Goldberg [2017]. They address the problem of the exposure bias and loss-evaluation mismatch with exploration at training time. More specifically, the model can choose alternative actions (leading to the same, correct output). Additionally, they model the transducer as a transition-based system over edit actions and add an additional copy edit. In a further step, Makarov and Clematide [2018a] eliminate the need for a character aligner. Instead, the approach uses an expert policy and imitation learning (IL), leading to high performance on several benchmarks. Wu et al. [2018] examine the effect of hard (non-monotonic) attention in neural encoder-decoder models, modelling alignment between input and output characters as a latent variable learned by the model. The alignment is restricted such that an output character is only allowed to attend to exactly one input character. They find that hard attention improves performance compared to similar methods with soft attention (such as Kann and Schütze [2016b]). In contrast to methods such as Aharoni and Goldberg [2017], their approach differs in two ways: Firstly, the alignment is non-monotonic. Secondly, the alignment is learned by the model itself and not produced by a separate aligner. Wu and Cotterell [2019] build on the idea of Wu et al. [2018] and experiment with hard monotonic attention. They find that enforcing monotonicity brings further improvements. In fact, they claim state-of-the-art (SOTA) performance for the task of morphological inflection at the time. They argue that jointly training monotonic alignment with the transduction model is advantageous to pipeline approaches such as Aharoni and Goldberg [2017].

All of the previously discussed neural approaches use some form of recurrent architecture (such as LSTMs [Hochreiter and Schmidhuber, 1997]) in their encoder and decoder. Such architectures have been popular not only for string transduction tasks but also for other seq2seq NLP tasks. A prominent example is neural machine translation [Bahdanau et al., 2016]. It is this domain in which the transformer architecture was firstly applied [Vaswani et al., 2017]. Ever since, transformer-based architectures have been applied with great success to manifold tasks such as language modelling [Devlin et al., 2019], question answering [Yang et al., 2019], part-of-speech tagging [Heinzerling and Strube, 2019] or punctuation prediction [Michail et al., 2021]. Consequently, these successes raise the question of whether transformers can similarly benefit string transduction tasks. A quick glance at recent shared tasks results would not necessarily suggest that: Although two out of the four winners of the SIGMOPRHON 2020 shared task on morphological inflection are transformer-based models, other neural architectures perform similarly well [Vylomova et al., 2020]. A similar outcome can be observed in the SIGMORPHON 2021 shared task on G2P [Gorman et al., 2020]. The early tries of Yolchuyeva et al. [2019] point in a similar direction. While they achieve competitive results, their transformer model does not deliver exceeding performance.

Wu et al. [2021] make a similar observation and try to explain this phenomenon. Firstly, models used for morphological tasks are usually trained on comparably small data sets, especially with regard to neural machine translation (from where the transformer originated). Secondly, the benefits of non-recurrent architectures may not be clear. They argue that benefits of the transformer architecture such as the modelling of long-range dependency [Vaswani et al., 2017; Gehring et al., 2017], word disambiguity or training speed are not critical for the success of morphological-related neural models. However, it is also Wu et al. [2021] that claim SOTA performance with their transformer-based approach for morphological inflection, G2P, and transliteration. They argue that a sufficiently large batch size is crucial in outperforming recurrent models. This may lead to the conclusion that the *correct* choice of hyperparameters is crucial for the success of transformers, and the literature evolving around morphological problems has only started to elaborate on these settings. A hint to the possibly large effects of different hyperparameter values for transformers is also given by Popel and Bojar [2018], who evaluate a range of different parameters and parameter values in the context of neural machine translation.

Other noteworthy contributions are made by and Rios et al. [2021] and Dong et al. [2022]. Former experiment with a loss function that biases attention towards monotonicity. They argue that this may bring slight improvements for some setups with transformers. Latter, develop a pre-trained grapheme model, similarly to BERT, but pre-trained on graphemes. They show that their approach can lead to substantial performance improvements for transformer-based models in G2P.

2.2 A Closer Look at the Neural Transducer

This section gives a closer look at the neural transducer model that serves as a starting point for this master thesis to prepare the reader for the following chapters. This model is equivalent to one of the baselines for the SIGMORPHON 2021 Shared Task on Multilingual G2P [Ashby et al., 2021]. Conceptually, this model is very similar to Makarov and Clematide [2018a] with adaptations to the G2P task [Makarov and Clematide, 2020]. The following explanations are largely based on Makarov and Clematide [2018a, 2020].

The model defines a neural seq2seq (encoder-decoder) system that *transduces* an input string into an output string. The transducer performs single-character edits (insertion, deletion, substitution, and copy). Given an input sequence of characters $\mathbf{x} = x_1 \dots x_{|\mathbf{x}|}, x_i \in \Sigma_x$ and an edit action sequence $\mathbf{a} = a_1 \dots a_{|\mathbf{a}|}, a_i \in \Sigma_a$, the transducer models a conditional distribution over the possible edits:

$$p_\theta(\mathbf{a} \mid \mathbf{x}) = \prod_{j=1}^{|\mathbf{a}|} p_\theta(a_j \mid \mathbf{a}_{<j}, \mathbf{x}) \tag{2.1}$$

The characters of the output sequence $\mathbf{y} = y_1 \dots x_{|\mathbf{y}|}, y_i \in \Sigma_y$ are deterministically computed from $\mathbf{x}$ and $\mathbf{a}$.

The encoder consists of a bidirectional LSTM model [Graves and Schmidhuber, 2005]. The encoder produces a representation h_i for every input character x_i

$$h_i = BiLSTM(E(x_i), \dots, E(x_n)), \tag{2.2}$$

where E is a lookup matrix for the embedding of x_i. The transitions (i.e., edit actions) are scored based on the LSTM decoder output. The decoder output s_t of decoder step t is defined as

$$s_t = LSTM(c_{t-1}, [A(a_{t-1}; h_i)]), \tag{2.3}$$

where c_{t-1}, $A(a_{t-1})$, and h_i are the previous decoder state, the embedding of the previous action state, and the embedding of the currently aligned input character, respectively. Which input character x_i is aligned in the current decoder step depends on the previously executed actions. At the beginning of the transduction process, the alignment pointer points to the first element of the input sequence x_i. Consecutively, the alignment is increased by one (moved to the next input character) if a copy, deletion, or substitution edit action is executed and remains unchanged in the case of an insertion. This follows traditional definitions for edit actions (such as Cotterell et al. [2014]). The transition probabilities at time step t are then calculated as:

$$P(a_t = k|\mathbf{a}_{<t}, \mathbf{x}, \Theta) = softmax_k(\mathbf{W} * s_t + \mathbf{b}) \tag{2.4}$$

W and b belong to the model parameters Θ (additionally to the embeddings and LSTM parameters).

The model parameters are optimized using imitation learning (IL), i.e., the model is trained to imitate an expert policy. IL is a training method used in NLP for structured prediction problems [Daumé et al., 2009; Ross et al., 2011; Chang et al., 2015]. Formally, structured prediction refers to problems with exponential output solutions in the input size. IL emphasizes the importance of state space exploration since it addresses the *exposure bias*, as discussed in more detail later on.
The training objective is twofold: The first objective is to minimize the sequence-level loss, expressed as Levenshtein distance [Levenshtein, 1966] between the target sequence and the predicted sequence $\mathbf{y}$. The second ensures that the target sequence is achieved most economically (with a minimum number of actions). The goal is to maximize the likelihood of optimal actions. This is achieved by minimizing the marginal negative log-likelihood of all optimal actions [Riezler et al., 2000]:

$$\mathcal{L}(D, \Theta) = -\sum_{l=1}^{N}\sum_{t=1}^{m} log \sum_{a \in A_t} P(a|\mathbf{a}_{<t}, \mathbf{x}^{(l)}, \Theta), \tag{2.5}$$

where $T = \{(\mathbf{x}^{(l)}, \mathbf{y}^{(l)})\}_{l=1}^{N}$ describes the training data and A_t the set of optimal actions at step t.
The expert policy, i.e., the gold actions the model should learn to perform, is based on the Stochastic Edit Distance (SED) model introduced by Ristad and Yianilos [1998]. Conceptually, it is a probabilistic version of the Levenshtein distance. Importantly, the model parameters are learned from the training data itself (before the actual training of the transducer). During training, the expert policy then uses the trained SED model to query the next best actions, given the partial prediction $\mathbf{y}_{<t}$, input remainder $\mathbf{x}_{\geq i}$ and target sequence. Note that multiple actions may be optimal at some time step t. In this case, all these are optimized with respect to the model parameters Θ, as seen in Equation 2.5. In any case, the actions derived from the SED expert minimize the string distance cost and, thus, minimize the first training objective. Moreover, the training procedure uses exploration at training time (*roll-in* stage). At time step t, the model either executes the optimal action derived from

the SED expert or, alternatively, samples from its own predictions. The idea is to expose the model to its own mistakes during training and, consequently, enable it to recover from them. This decreases the model's exposure to the *exposure bias*, which can influence model performance negatively [Wiseman and Rush, 2016]. Whether the model samples from its own predictions or not is determined by the sampling probability $p_{sampling}$ (depending on the epoch number i), derived from the following schedule:

$$p_{sampling}(i) = 1 - \frac{1}{1 + exp(i)} \tag{2.6}$$

Given the denominator of the second term in Equation 2.6, the second term approaches 0 after around 10 epochs. Thus, after a short time, the model samples entirely from its own predictions.

3 Design / Implementation

This chapter discusses the necessary conceptual and technical changes for the reimplementation of the baseline model in PyTorch (Section 3.3), mini-batch training and batched greedy decoding (Section 3.4), as well as transformer encoders (Section 3.5). Generally, I present important and challenging implementation details to showcase my work. These sections are all similarly structured: The first part discusses the concept for the implementation, followed by the technical implementation in the second part. To prepare the reader for this discussion, Section 3.1 highlights relevant implementation details of the baseline transducer model. Section 3.2 documents the location and documentation style of the source code. Lastly, Section 3.6 summarizes other noteworthy aspects of the implementation.

3.1 The Code Base of the Baseline Model

The starting point for the reimplementation of the model is the code used for one of the baselines for the SIGMORPHON 2021 shared task on multilingual G2P [Ashby et al., 2021] [1]. Conceptually, and as described in detail in Section 2.2, the model is similar to Makarov and Clematide [2020].

The source code consists of multiple Python files that group conceptually close software components. For instance, the code for training the model, the implementation of the neural model itself, and the SED expert are all encapsulated in a different file. However, much of the implementation work done in this thesis revolves around conceptual and technical changes to a single method. The technical understanding of this method is essential in understanding the presented work. The code description in Subsection 3.1.1 is limited to this essential part in order to stay within the scope of this thesis. The observations I make in Subsection 3.1.2 relate to this description.

[1] The code for the baseline model can be found here: https://github.com/peter-makarov/il-reimplementation/tree/feature/sgm2021. In fact, Peter Makarov had already started implementing the model in PyTorch, and the code on the corresponding branch was my actual starting point: https://github.com/peter-makarov/il-reimplementation/commit/aea76439d63820b846452a9074550ddbfe547610.

However, I have added other relevant observations based on analyzing other parts of the code. Subsection 3.1.2 aims to sharpen the reader's view for the necessity of changes (in relation to the different implementations, as discussed in the following).

3.1.1 The transduce Method

The `Transducer` class[2] implements all necessary methods used for training and inference (greedy and beam search decoding). The most important method is the `transduce` method. It is used during training and inference (i.e., when the output is unknown). More concretely, it combines all aspects of the IL training approach, the alignment modeling, and is used for greedy decoding.[3] Figure 3.1 gives a conceptual overview of the `transduce` method. Apart from `beam_search_decode`, this method uses all other methods of the `Transducer` class. In the following, the different steps shown in Figure 3.1 are referenced as *Step x*, and the reader is walked through the implementation of `transduce`.

In the first step, the encoder is executed (Listing 3.1): The input embeddings are retrieved from the embeddings lookup matrix (line 196) and subsequently run through the encoder network (line 197). This corresponds to *Step 1*.

```
196 input_emb = self.input_embedding(encoded_input, is_training)
197 bidirectional_emb = self.bidirectional_encoding(input_emb)[1:]
```

Listing 3.1: The input sequence is encoded.

In a second step, the `transduce` method initializes the transducer state, as shown in Listing 3.2 (*Step 2*). The description of the transducer state follows the theoretical description in Section 2.2: It is described by the alignment pointer used to select the currently aligned input character (`alignment`) and the action history used to access the previously executed action (`action_history`). Additionally, the `output` list stores the transduced output, and the `losses` list stores the loss for each step.

```
alignment = 0
action_history: List[int] = [BEGIN_WORD]
output: List[str] = []
losses: List[dy.Expression] = []
```

Listing 3.2: The initialization of the transducer state.

[2]Implementation path: trans/transducer.py/Transducer (with respect to the repository)

[3]Beam search decoding is implemented by a separate method (`beam_search_decode`). Conceptually, it differs quite a lot from greedy decoding. As this decoding method is not the focus of implementation in this thesis, the discussion is limited to greedy decoding.

In a further step, the `compute_valid_actions` method computes all valid actions for the current time step (*Step 3*). Copy, delete, and substitution actions are valid if the alignment pointer does not (yet) point to the last element of the input sequence. While this is not part of the theoretical description in Section 2.2, it is a logical constraint: These actions would increase the alignment pointer. However, in the case where the length of the input string is smaller then the alignment pointer, the whole input is already consumed.

Then, based on the alignment variable and the encoder output, represented by `bidirectional_emb`, the aligned input embedding is selected with Python list indexing (Listing 3.3, *Step 4*).

```
input_char_embedding = bidirectional_emb[alignment]
```

Listing 3.3: Selecting (aligning) the input embedding.

Consecutively, the selected input embedding is concatenated with the action embedding of the previous time step, which is retrieved from the lookup matrix `act_lookup` (Listing 3.4, *Step 5*).

```
previous_action_embedding = self.act_lookup[action_history[-1]]
decoder_input = dy.concatenate(
    [input_char_embedding, previous_action_embedding])
```

Listing 3.4: Input concatenation before executing the decoder.

The concatenated embeddings `decoder_input` then serve as decoder input. Consecutively, a linear layer and a softmax classifier is used to obtain a probability distribution over all actions (Listing 3.5, *Step 6*).

```
decoder = decoder.add_input(decoder_input)

# classify
decoder_output = decoder.output()
logits = self.pW * decoder_output + self.pb
log_probs = dy.log_softmax(logits, valid_actions)
```

Listing 3.5: The decoder is executed and a probability distribution is obtained.

The next steps depend on whether the model is in training or in inference mode. The code is displayed in Listing 3.6.

In the latter case, the most probable action is selected (lines 225-227, *Step 7b*).

During training, more steps are required as this part implements the IL learning routine. Firstly, the expert is rolled out, i.e., the next best actions are calculated.

This is achieved by calling the `expert_rollout` method, which, in turn, calls the SED model and scores all actions. Note that this step requires accessing the current transducer state (lines 231-233, *Step 7a*).

Based on the optimal actions, the model's output and the valid actions, the loss is calculated using the `log_sum_softmax_loss` method (lines 235-236, *Step 8*).

Then, a prediction respectively action is either sampled from the model's own predictions (*Step 9a*) or as computed by the expert (*Step 9b*) (Lines 14-24).

```
225 if target is None:
226     # argmax decoding
227     action = np.argmax(log_probs_np)
228 else:
229     # training with dynamic oracle
230 
231     # 1. ACTIONS TO MAXIMIZE
232     optim_actions = self.expert_rollout(
233         input_, target, alignment, output)
234 
235     loss = self.log_sum_softmax_loss(
236         optim_actions, logits, valid_actions)
237 
238     # 2. ACTION SPACE EXPLORATION: NEXT ACTION
239     if np.random.rand() <= rollin:
240         # action is picked by sampling
241         action = self.sample(log_probs_np)
242     else:
243         # action is picked from optim_actions
244         # reinforce model beliefs by picking highest probability
245         # action that is consistent with oracle
246         action = optim_actions[
247             int(np.argmax([log_probs_np[a] for a in optim_actions]))
248         ]
```

Listing 3.6: The method selects the most probable action during inference or performs roll-in during training.

Then, the transducer state is updated, as shown in Listing 3.7. The update is performed by extending the state variable lists `action_history`, `output` and `losses` (only during training). The extension of `output` and the alignment pointer update depend on the decoded action type. This procedure is implemented as a Python if-elif-else statement (lines 253-270, *Step 10*). Note the break condition in line 268: This defines the end of the transduction process, in which case the transduced sequence (respectively the loss) is returned (*Step 11*). In all other cases, the transduction process continues. With respect to the Figure 3.1, the process starts over at *Step 3*. This iterative process is implemented with a Python while loop. To be precise, the while loop defines a second stop criterium: The constant `MAX_ACTION_SEQ_LEN` determines the maximum allowed number of actions.

```python
losses.append(loss) # only during training

log_p += log_probs_np[action]
action_history.append(action)
# execute the action to update the transducer state
action = self.vocab.decode_action(action)

if isinstance(action, ConditionalCopy):
    char_ = input_[alignment]
    alignment += 1
    output.append(char_)
elif isinstance(action, ConditionalDel):
    alignment += 1
elif isinstance(action, ConditionalIns):
    output.append(action.new)
elif isinstance(action, ConditionalSub):
    alignment += 1
    output.append(action.new)
elif isinstance(action, EndOfSequence):
    break
else:
    raise ValueError(f"Unknown action: {action}.")
```

Listing 3.7: Updating the transducer state.

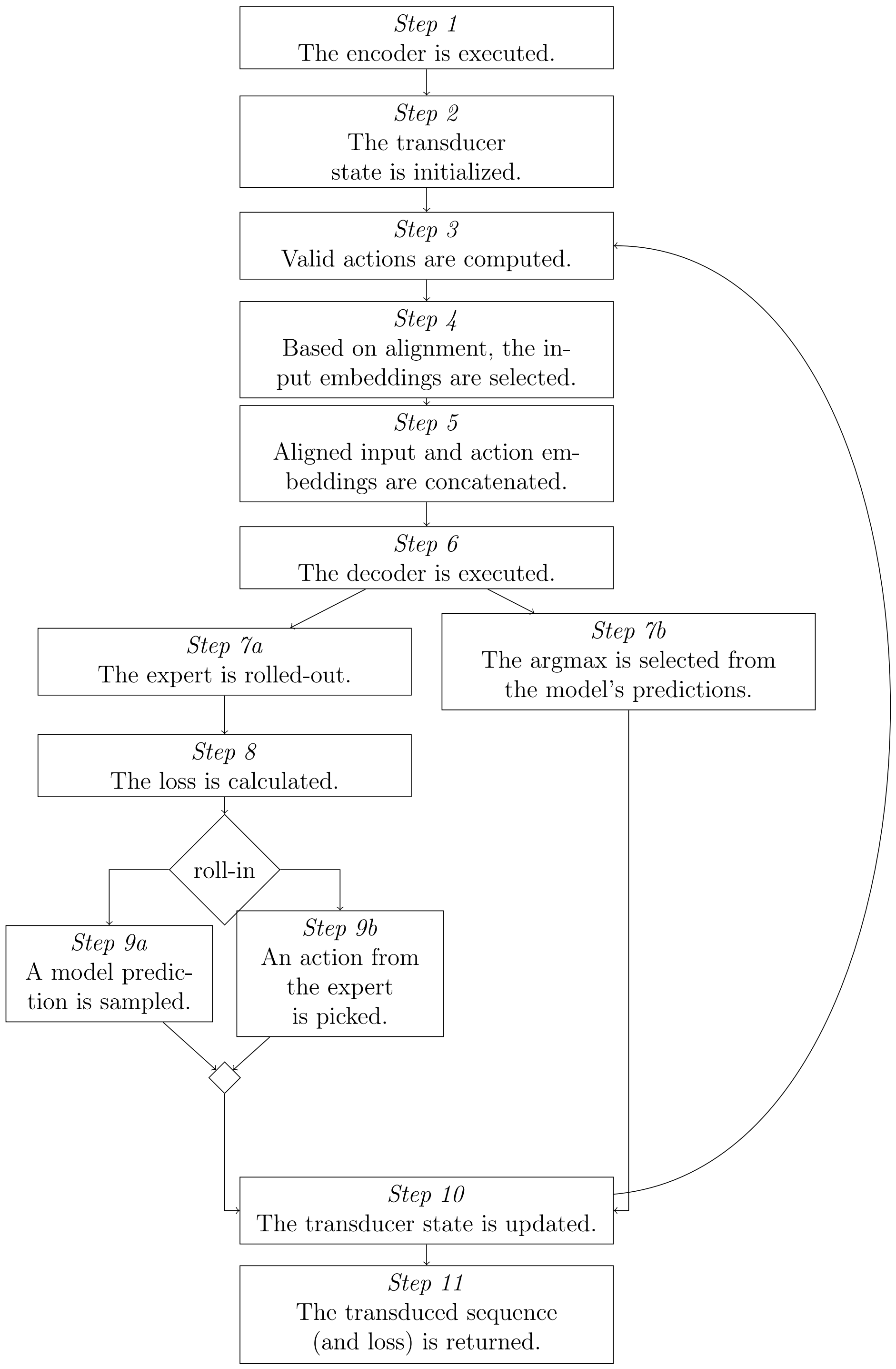
Step 1
The encoder is executed.
Step 2
The transducer state is initialized.
Step 3
Valid actions are computed.
Step 4
Based on alignment, the input embeddings are selected.
Step 5
Aligned input and action embeddings are concatenated.
Step 6
The decoder is executed.
Step 7a
The expert is rolled-out.
Step 7b
The argmax is selected from the model's predictions.
Step 8
The loss is calculated.
roll-in
Step 9a
A model prediction is sampled.
Step 9b
An action from the expert is picked.
Step 10
The transducer state is updated.
Step 11
The transduced sequence (and loss) is returned.

Figure 3.1: The conceptual structure of the `transduce` method used during training and inference.

3.1.2 Observations

Single sample state The code is designed for the processing of one sample at a time, as shown by the declaration of the transducer state (Listing 3.2). The state variables are either expressed as one-dimensional lists or as an integer. Both strucutres are unsuited to hold structured information of multiple samples.

Python data structures The code relies heavily on pure Python data structures (for instance, lists). Machine learning frameworks such as PyTorch typically provide library-specific implementations for data structures. These structures maximize efficiency and offer a wide range of library methods. Additionally, GPU support typically requires the use of these specific data structures. Therefore, an effective and efficient implementation based on such frameworks must carefully consider this.

Python control structures The code mixes DyNet code with Python control structures. The update of the transducer state uses an if-elif-else statement (Listing 3.7), and the step-wise training procedure involves a while loop. In the context of GPU training (in PyTorch), this mixture forces the training on multiple devices, as only a CPU can execute the code for such control structures. This decreases the potential for GPU-induced efficiency gains. An efficient solution must consider efforts to reduce code that only a CPU can execute.

Encoder and decoder components The encoder and decoder architecture is *hard coded*. The parts of the encoder (embedding matrix, forward and backward LSMT) are defined as properties of the `Transducer` object. Similar is true for the decoder. Additionally, the execution of the encoder and decoder requires the access of multiple methods respectively properties (Listings 3.1 and 3.5). What is more, these same steps are also performed in the `beam_search_decode` method. Overall, this makes the architecture less flexible. Changes to the encoder-decoder architecture likely require changes to the three methods leading to increased maintenance and lower readability.

3.2 Preliminaries: Software

3.2.1 Code Repository

All source code that I produced as part of this thesis can be found in my public GitHub repository[4]. Generally, I have used separate branches for different implementations. The concrete branch for each implementation (discussed in Sections 3.3, 3.4 and 3.5) is noted at the beginning of the corresponding section. In the following sections, references to specific code locations are formulated in footnotes as *Implementation path: path_to_location* (with respect to the structure of the corresponding branch).

3.2.2 Documentation

First of all, all functions, respectively, methods, and classes use Docstrings[5] as documentation format. Apart from that, I commented difficult source code parts, for example, whenever specific implementation details are crucial. Generally, the technically savvy reader should be able to understand the source code given these comments.

The necessary software packages (and versions) are defined in the `setup.py` file in the repository of each branch.

What is more, commits have meaningful messages summarizing the work done and contain relatively minor changes. During development, I have consequently used the Git branching model, i.e., features are developed on a dedicated branch (a feature branch) and then merged via Pull Request into the development branch. The idea was to make my development process transparent and understandable.

3.3 A One-to-One Reimplementation in PyTorch

This section addresses the reimplementation of the baseline model in PyTorch in relation to the first research question (Subsection 1.4.1).

[4]https://github.com/slvnwhrl/il-reimplementation

[5]https://peps.python.org/pep-0257

3.3.1 Source Code

The source code for this implementation is accessible here:
https://github.com/slvnwhrl/il-reimplementation/tree/feature/port-to-torch

This tagged version documents the state of code at the end of this thesis:
https://github.com/slvnwhrl/il-reimplementation/releases/tag/1to1_reimp

3.3.2 Concept

Replacing DyNet components The reimplementation itself does not contain any conceptual changes to the model, respectively, code. The reimplementation consists of replacing DyNet's library components with PyTorch's library counterparts.

Data management Additionally, the implementation makes use of PyTorch components for holding data samples (a dataset) as well as accessing these samples during training (a data loader). Within the PyTorch framework, these components facilitate working with data and offer easy-to-use features (for instance, shuffling the training data). More importantly, these components implement methods for handling batches. Using these components is, therefore, also a preparation for the next phase of work in this thesis (mini-batch training and batched greedy decoding).

3.3.3 Implementation

Replacing DyNet components First of all, all DyNet components are replaced. Luckily, DyNet's and PyTorch's library API are similar. This makes the reimplementation rather straightforward. Some concrete code examples are given in the following to illustrate this. For instance, Listing 3.8 shows the initialization of the character embedding lookup matrix in DyNet and PyTorch. Both methods use very similar parameters, and the usage only differs in the right method call. Similarly, the implementation for concatenating embedding vectors only differs slightly, as shown in Listing 3.9. The only difference (apart from the method names) is that the PyTorch-based lookup operation requires a tensor as an input parameter (line 10), while the same DyNet operation takes a simple Python list as input (line 4). Listing 3.10 shows the last example: `losses` is a two-dimensional list. The entries are lists themselves and hold token-level losses. `losses` is then averaged to get the average token-loss per batch (line 2, line 7). The backward pass is performed identically by using the method `backward` on `batch_loss` (line 4 and 9).

```
# DYNET
self.char_lookup = model.add_lookup_parameters(
    (self.number_characters, char_dim))

# PYTORCH
self.char_lookup = torch.nn.Embedding(
    num_embeddings=self.number_characters, embedding_dim=char_dim, ...)
```

Listing 3.8: DyNet and PyTorch: The initialization of the character embedding lookup matrix.

```
# DYNET
decoder_input = dy.concatenate([
    bidirectional_emb[alignment],
    self.act_lookup[action_history[-1]]
])

# PYTORCH
decoder_input = torch.cat([
    bidirectional_emb[alignment],
    self.act_lookup[torch.tensor([action_history[-1]], device=self.device)]
])
```

Listing 3.9: DyNet and PyTorch: Concatenating aligned character and action embedding in the decoder.

```
1 # DYNET
2 batch_loss = -dy.average(losses)
3 train_loss += batch_loss.scalar_value()
4 batch_loss.backward()
5
6 # PYTORCH
7 batch_loss = -torch.mean(torch.stack(losses))
8 train_loss += batch_loss.item()
9 batch_loss.backward()
```

Listing 3.10: DyNet and PyTorch: Performing a backward pass on batch loss.

Data loading The implementation for the baseline model uses a simple Python list to store data samples. Technically, a data sample is represented by a custom Python object `Sample` whose attributes represent the attributes of the data (e.g., the input character sequence). During training, the data list is then traversed and single entries of the list (i.e., a `Sample`) are then passed to the `transduce` method. As previously discussed, this mechanism is replaced by PyTorch provides library components: `torch.utils.data.Dataset`[6] for storing data sampling and

[6]https://pytorch.org/docs/1.10/data.html#torch.utils.data.Dataset

`torch.utils.data.DataLoader`[7] for accessing data.[8]

A note on reproducibility According to the PyTorch documentation[9], reproducibility of experiments is not guaranteed in any case and may depend on factors such as the specific release, platform or used device (CPU or GPU). To approximate reproducibility, the documentation recommends using `torch.manual_seed` to set a seed for generating random numbers on all devices. Additionally, it provides code to make the behaviour of the `torch.utils.data.DataLoader` deterministic. This is important when data is shuffled (e.g., after each training epoch). This implementation follows these recommendations.

3.4 Let's Scale: Batching

This section is related to the second research question (Subsection 1.4.2) by addressing mini-batch training (Subsection 3.4.2) and batched greedy decoding (Subsection 3.4.3).

3.4.1 Source Code

The source code for this implementation can is accessible here:
https://github.com/slvnwhrl/il-reimplementation/tree/development

This tagged version documents the state of code at the end of this thesis:
https://github.com/slvnwhrl/il-reimplementation/releases/tag/mini_batch_imp

3.4.2 Mini-Batch Training (Teacher Forcing)

The basic idea of implementing mini-batch training is to increase training speed by parallelizing as many computations as possible. Based on the discussion in Subsection 3.1, this implies two significant changes two the training regime described in Subsection 3.1.1. Importantly, this implementation simplifies two aspects of the training routine which are motivated and explained in the following.

[7]https://pytorch.org/docs/1.10/data.html#torch.utils.data.DataLoader

[8]The concrete changes are summarized in this commit: https://github.com/slvnwhrl/il-reimplementation/commit/3f4271489463173ea1ee0e16648d42598f033342

[9]https://pytorch.org/docs/1.10/notes/randomness.html

3.4.2.1 Conceptual Simplifications

With respect to the model of Makarov and Clematide [2020] (as discussed in detail in Section 2.2), this implementation differs in two aspects of the training regime:

- *Teacher forcing*: The training routine does not incorporate exploration at training time (roll-in), i.e., the model always executes the expert's action (and never samples from its own predictions). Concerning the theoretical background, this means that the sampling mechanism described by Equation 2.6 is not applied. Therefore, the training routine implements teacher forcing [Williams and Zipser, 1989]. While this simplification possibly exposes the model to the previously discussed exposure bias, it increases the potential for training efficiency: Without exploration at training time, all training targets can be computed prior to the training allowing the unrolling of decoder steps (cf. Subsection 3.4.2.2).
- *Optimization of a single target action*: The loss function in the training routine only considers at maximum one optimal target action. With respect to the definition of the loss function (Equation 2.5), this means that the set of optimal actions A_t is restricted to a single action. Optimizing multiple targets builds on the idea that multiple edit sequences may lead to the same output. However, these edit sequences may have different lengths, which is incompatible with the idea of unrolling decoder steps (cf. Subsection 3.4.2.2).

3.4.2.2 Concept

Unrolling decoder steps The main challenge in achieving parallelization during training comes from the conceptual design of the model: Given some action sequence $\mathbf{a}^{(1)}$, every training step t is dependent on a_{t-1} as well as the alignment pointer. However, this information is only discovered at training step $t-1$ and requires the roll-out of the expert. This makes the process unsuited for parallelization: The implementation is forced to access unparallelizable methods (the roll-out of the expert) in a step-wise manner (implemented with a Python while loop), as discussed in Subsection 3.1. To achieve parallelization of the training process, the training of each input sequence must be *unrolled*. More precisely, the information required for every training step t must be known *prior* to the training such that the training of multiple sequences can be performed simultaneously. Conceptually, this drastically changes the training procedure described in Figure 3.1. Figure 3.2, describing the training procedure for mini-batch training, displays this: In the first step, the expert

is rolled out for every training sample such that the optimal action sequence is known for each training sample. Concretely, rolling out the expert before performing the training steps provides the following information: The action sequence that is optimized during training, the number of decoding steps, the alignment and the permissible actions at every decoding step. This information then allows to batch the training data and consecutively perform mini-batch training.
To sum up, the main conceptual difference between the implementation for the baseline model (Figure 3.1) and the mini-batch training implementation (Figure 3.2) lies in the source of information. The former procedure provides it, while the latter is provided with it.

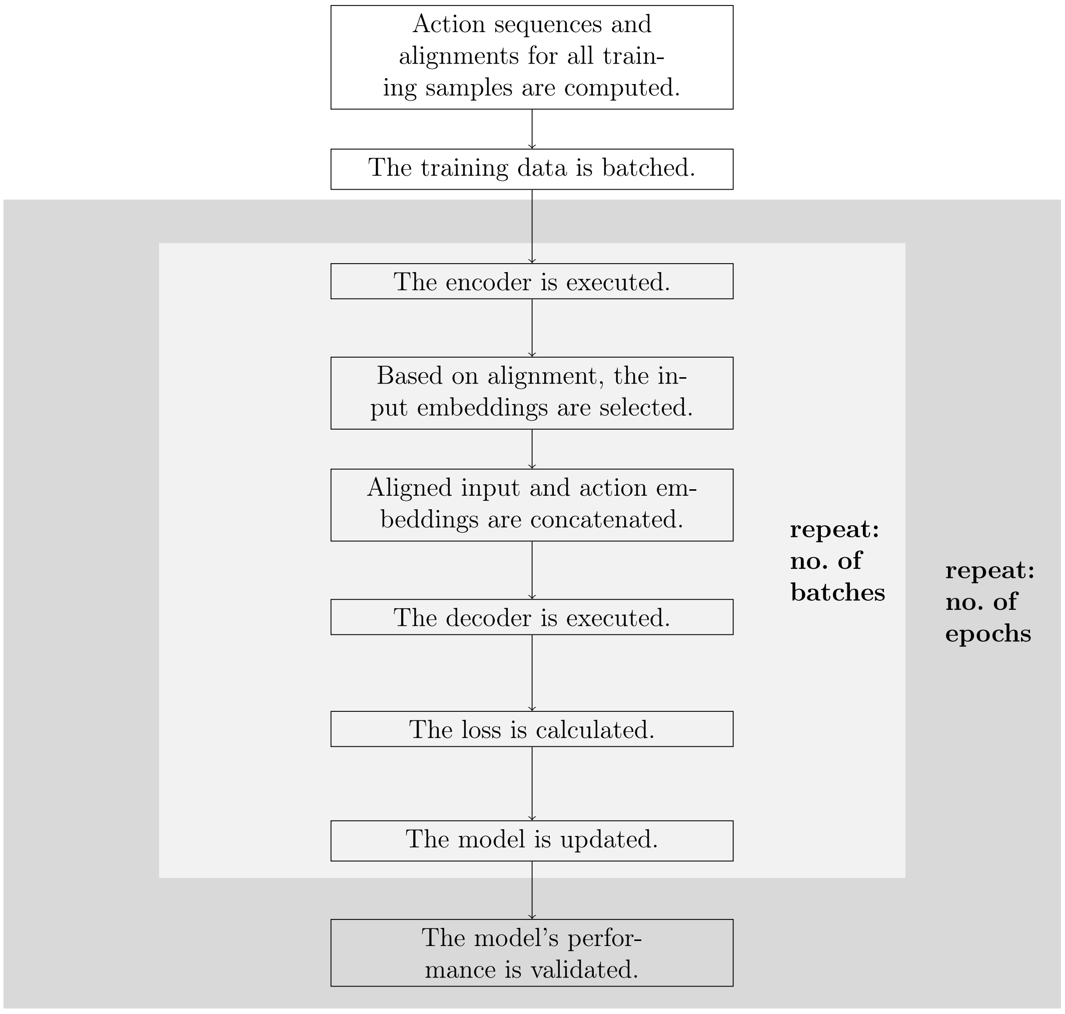
Action sequences and alignments for all training samples are computed.
The training data is batched.
The encoder is executed.
Based on alignment, the input embeddings are selected.
Aligned input and action embeddings are concatenated.
The decoder is executed.
The loss is calculated.
The model is updated.
The model's performance is validated.
repeat: no. of batches
repeat: no. of epochs

Figure 3.2: The training procedure for mini-batch training.

Matrix operations As previously discussed, the implementation for the baseline model primarily uses Python data structures (for example, lists). While it is possible to work with Python data structures and PyTorch (to some extent), PyTorch provides Tensors as a data structure.[10] A Tensor is a multi-dimensional matrix and contains only elements of a single type (e.g., integer or float). Using Tensors offers many advantages: The PyTorch ecosystem is designed to use Tensors. Firstly, many methods rely on Tensor-type input. Not using Tensors makes it harder to profit from the plethora of help offered by the PyTorch library. Secondly, being a multi-dimensional data structure, Tensors are well-suited for processing batches. Lastly, many operations involving numbers can be represented as matrix operations, which Tensors can efficiently perform. This is especially valuable in connection with a GPU: It makes these operations parallelizable and allows performing more computations on a single device.

To summarize, this concept defines a clear design guideline: Use Tensors whenever possible.

3.4.2.3 Implementation

Pre-training expert roll-out As discussed in Subsection 3.4.2.2, the most important building block for unrolling decoder steps is to precompute the relevant information used for optimization during training. To achieve this, I have implemented a dedicated function `precompute_from_expert`[11]. This function rolls out the expert for the whole input string of a training sample. The method produced Tensors for each training sample storing information about the optimal action sequence, the alignment position, and valid actions with respect to every decoder step.

Batching Batches are represented as 3-dimensional padded Tensors combining single training samples. PyTorch's `torch.utils.data.DataLoader` is used to batch single sequences.[12] Generally, neural architecture components in PyTorch are able to receive batched input and this implementation strictly follows PyTorch's operating principle.[13]

Separating training from inference Given that the training and inference procedure are now quite different, training and inference are separated. Additionally, the steps required to perform an encoder and decoder step are encapsulated into

[10]https://pytorch.org/docs/1.10/tensors.html

[11]Implementation path: trans/train.py/precompute_from_expert

[12]Implementation path: trans/utils.py (various classes/methods)

[13]See, for instance, the description of input parameters for an LSTM:
https://pytorch.org/docs/1.10/generated/torch.nn.LSTM.html

separate methods. These methods are independent of whether the model is in training or inference mode and can be used similarly in both situations. This results in three different methods (of the `Transducer` class): `encoder_step` for performing an encoder step, `decoder_step` for performing a decoder step, and `training_step` for performing a training step.[14] Importantly, all methods work with PyTorch Tensor, representing the input as batches. The declaration of `encoder_step` (Listing 3.11) demonstrates this: The input parameter `encoded_input`, representing the encoded input sequence, is of shape `[batch_size x sequence_length]`. The method then performs a batched embedding lookup, exectutes the encoder and returns a Tensor of shape `[sequence_length x batch_size x embedding_dim]`.

```python
def encoder_step(self, encoded_input: torch.tensor, is_training: bool = False)
    -> torch.tensor:
    """Runs the encoder.
    Args:
        encoded_input: Encoded input character codes.
        is_training: Bool indicating whether model is in training or not.
    Returns:
        Encoder output."""
```

Listing 3.11: Declaration of the `encoder_step` method.

An unrolled decoder step Basically, an unrolled decoder step is performed by passing a batched Tensor to the PyTorch decoder component of the model (performed in `decoder_step`). The most difficult part is thereby the preparation of this input: In the first step, a batched selection of input embeddings (as defined by the alignment) must be performed. My solution is presented in Listing 3.12. The `alignment` input tensor is used to select the aligned embeddings from the encoder output. The shape of `alignment` represents a vector. It is a flattened view of an *alignment matrix*: A matrix of shape `[action_sequence_length x batch_size x 1]` storing the alignment position for every decoding step for all sequences in the batch. Basically, the embeddings are selected with a 2-dimensional tensor of size `[action_sequence_length x batch_size]` that is built from the `alignment` vector (lines 384-385 in Listing 3.12). The first dimension of this Tensor corresponds to the position in the input sequence and the second dimension denotes the position in the batch. Importantly, the position in the batch can be directly constructed from the `alignment` vector (given the batch size). The selection process results in a matrix of shape `[(action_sequence_length x batch_size) x embedding_dimension]` (flattened view), which is then reshaped into three dimension. The reshaped Tensor

[14]Implementation paths: trans/transducer.py/Transducer.{encoder_step,decoder_step, training_step}

is then concatenated with the action embeddings of the previous decoder step and used as input for the decoder.

```
input_char = encoder_output[alignment, torch.tensor([i for i in range(batch_size)
    for _ in range(len(alignment)//batch_size)], device=self.device)].unsqueeze(...)
input_char = torch.reshape(input_char,
    (batch_size, len(alignment)//batch_size, -1)).transpose(0, 1)
```

Listing 3.12: Selecting the aligned input embeddings in the `decoder_step` method. `input_char` is abbreviated for `input_char_embedding`.

Training step Performing a training step mainly consists of running `encoder_step` and `decoder_step`. Additionally, the decoder output is run through a linear layer, and the loss is subsequently calculated based on this output. This is, in principle, similar to the implementation of the baseline model, as discussed in Subsection 3.1. However, the loss calculation for batches is slightly different. The reason is that sequences contain paddings: To calculate the per-sequence loss for each sequence in the batch, the padded elements must be neglected when averaging token-level loss, as shown in Listing 3.13.

```
# compute losses
# the loss for each seq in the batch is divided by the nr of non-padding elements
# --> loss per seq = avg. loss per token in seq
true_action_lengths = action_history.size(0) - (action_history == PAD).sum(dim=0)
losses = self.log_sum_softmax_loss(logits, optimal_actions_mask, valid_actions_mask)
losses = -losses.sum(dim=0) / true_action_lengths
```

Listing 3.13: Excluding padded elements when calculating the per-sequence loss in `training_step`.

GPU support To enable GPU support, PyTorch provides a very simple interface: Tensors must simply be moved to the GPU device during initialization. For flexibility, this implementation makes us of a CLI option (`--device`) that allows to define the device on which computations are executed. Any Tensor is then accordingly initialized.

3.4.3 Greedy Decoding

The idea of this implementation is to decrease the time needed for inference. In the context of large data volumes, faster inference becomes increasingly valuable. It helps to speed up training by decreasing the time needed for evaluation (after each epoch).

3.4.3.1 Concept

Simultaneous decoding While the implementation for mini-batch training requires conceptual and technical changes, the implementation for batched greedy decoding is rather technical. The inference process described in Figure 3.1 is hardly changed. In principle, the idea of this implementation is to update the `transduce` method such that the same (i.e., in terms of time) decoding steps of multiple sequences can be performed in parallel. This requires two fundamental changes to the procedure: On the one hand, the transducer state must be able to describe the state of multiple sequences. Temporally, the state for all sequences corresponds to the same decoding step. On the other hand, all steps required to perform one decoder step must be able to process multiple sequences simultaneously. Additionally, the training-related steps in the `transduce` are removed as training and inference are now separated.

Matrix operations Similarly to the concept for mini-batch training, the idea is to perform as much as possible of the process within the PyTorch framework. This has two design consequences for the implementation: Firstly, all numeric data structures should be expressed with Tensors. And secondly, all computations should be either expressed as matrix operations or, if not possible, rearranged to the beginning or end of the decoding process.

3.4.3.2 Implementation

Transducer state initialization Compared to Listing 3.2, the transducer state in the `transduce`[15] method is now described by PyTorch Tensors (Listing 3.14). `alignment` is a vector of size `[batch_size x 1]` representing the alignment position for every sequence in the batch and for the current decoder step. `action_history` representes a matrix of size `[1 x batch_size x number_decoding_steps]`. This matrix holds the action history of all batch sequences up to the current decoder step.

```python
alignment = torch.full((batch_size,), 0, device=self.device)
action_history = torch.tensor([[[BEGIN_WORD]] * batch_size],
                              device=self.device, dtype=torch.int)
```

Listing 3.14: The initialization of the transducer state for multiple sequences.

Importantly, the `output` state variable is removed from initialization. Conceptually, the transduced string sequence is unnecessary for performing a decoder step. The

[15]Implementation path: trans/transducer.py/Transducer.transduce

transduced sequence is produced by decoding the encoded actions (integers) and executing the decoded action (e.g., copying the currently aligned input character). The decoding process involves string data types (the output sequence) and can therefore not be represented with Tensors. Consequently, the action decoding is moved to the end of the inference process, reducing the amount of executed pure Python code while performing a single decoder step. In the implementation of the baseline model, action decoding is performed during the transducer state update (Listing 3.7).

Stop criteria The transduction process is terminated if one of two criteria is matched: Either a predefined maximum action sequence length is reached, or an `End-of-Sequence` action is produced. In the implementation of the baseline model, these criteria must always be full filled for a *single* sequence. The same is true for a specific sequence in the context of batches. However, the point of time when the stop criteria is met may differ for the different sequences within the batch. As a decoding step is always performed for multiple sequences, the termination process can only be terminated once the stop criteria are met for *all* sequences. This restriction is implemented by the function `continue_decoding` (Listing 3.15). The function returns `True` if all action sequences in `action_history` contain the encoded integer for the `End-of-Sequence` action, and `False` otherwise. Technically, this is achieved by comparing the batch size with the number of sequences in the batch that contain an `End-of-Sequence` token. `continue_decoding` is used in the while loop in line 509 in Listing 3.18. The condition of this loop also takes care of the second stop criteria. Every loop iteration corresponds to one decoding step, and the loop is exited as soon as one of both criteria is met.

```
504 # decoding is continued until all sequences
505 # in the batch have "found" an end word
506 def continue_decoding():
507     return torch.any(action_history == END_WORD, dim=2).sum() < batch_size
508
509 while continue_decoding() and action_history.size(2) <= MAX_ACTION_SEQ_LEN:
```

Listing 3.15: A single decoding step in batched greedy decoding.

Updating alignment pointers The transducer update also includes the update of the alignment pointer - this update depends on the type of action and is required in every decoder step. While this update cannot be rearranged, the simultaneous update of all alignment pointers can be represented as a matrix operation. For this purpose, the `Transducer` object is initialized with an additional property `alignment_update` (Listing 3.16). It is a vector of shape `[number_of_actions x 1]`. The vector represents a mapping between encoded actions and how they affect the alignment pointer. More precisely, a vector entry is 1 (if the corresponding action moves the alignment to the next position) or 0 (if the alignment is not moved). The update of the state vector `alignment` can then be represented as a matrix addition (cf. line 526 in Listing 3.18).

```python
# maps action index to alignment update
alignment_update = [0] * self.number_actions
for i, action in enumerate(self.vocab.actions.i2w):
    if isinstance(action,
                  (ConditionalCopy, ConditionalDel, ConditionalSub)):
        alignment_update[i] = 1
self.alignment_update = torch.tensor(alignment_update, device=self.device)
```

Listing 3.16: Initializing the lookup vector used to perform batched alignment pointer updates.

Valid actions lookup Which actions may be executable at the current time step also depends on whether an action is valid (as discussed in Section 3.1). Similarly to the alignment lookup vector (Listing 3.16), this operation is implemented as a precomputed lookup table. The lookup table `valid_actions_lookup` is initialized as a property of the `Transducer` object (Listing 3.17). During training, valid actions can be retrieved efficiently for the whole batch at every decoder step.

```python
# lookup for valid actions (given length of encoder suffix)
self.valid_actions_lookup = torch.stack(
    [self.compute_valid_actions(i)
     for i in range(MAX_INPUT_SEQ_LEN)],
    dim=0).unsqueeze(dim=0)
```

Listing 3.17: Initializing the lookup matrix used for retrieving valid actions for a whole batch.

Updating the transducer state The update of the transducer state is performed by concatenating the matrix `action_history` with the `actions` vector obtained in the previous step (line 3, Listing 3.18). The update of `alignment` vector is shown in line 526 in Listing 3.18. The update is performed using simple matrix addition. The first summand is the matrix itself. The second summand is a vector of equal shape.

```
522 action_history = torch.cat(
523     (action_history, actions.unsqueeze(dim=2)),
524     dim=2
525 )
526 alignment = alignment + self.alignment_update[actions.squeeze(dim=0)]
```

Listing 3.18: Updating the transducer state for a batch.

Action decoding After the batch decoding has stopped, and on the basis of the `action_history` matrix, the transduced output strings are produced for all batch sequences. This step applies two modifications to `action_history` (line 536, Listing 3.19): The first action can be negelected as it represents, in any case, the `Begin-of-Sequence` action. Additionally, each sequence is right trimmed up to first occurrence of an `End-of-Sequence` action. Then, `decode_encoded_output` is used to transduce the output strings.

```
533 # trim action history
534 # --> first element is not considered (begin-of-sequence-token)
535 # --> and only token up to the first end-of-sequence-token (including it)
536 action_history = [seq[1:(seq.index(EndOfSequence()) + 1 if EndOfSequence()
537                                                           in seq else -1)]
538                   for seq in action_history.squeeze(dim=0).tolist()]
539 
540 return Output(action_history, self.decode_encoded_output(input_, action_history),
541               log_p, None)
```

Listing 3.19: Adjusting history and returning the transduced output.

GPU support Similar to the discussion for the GPU support of mini-batch training (Subsection 3.4.2), GPU support in this implementation is achieved by moving any Tensors to the device specified via CLI.

3.5 Transformer Encoder

This section addresses third research question (Subsection 1.4.3) by implementing transformer encoders. This implementation extends the work presented in Section 3.4.

3.5.1 Source Code

The source code for this implementation can is accessible here:
https://github.com/slvnwhrl/il-reimplementation/tree/development

This tagged version documents the state of code at the end of this thesis:
https://github.com/slvnwhrl/il-reimplementation/releases/tag/mini_batch_imp

3.5.2 Concept

Encapsulating encoders In the implementation of the baseline model, the initialization of the encoder is hardcoded in the initialization of the `Transducer` object. The encoder initialization arguments (for example, the number of encoder layers) are passed as keyword arguments to `Transducer.__init__`. Given that the implementation has exclusively used LSTM encoders, this is a suitable approach. For any model, the encoder is always initialized similarly. However, enabling the model to use a different encoder architecture changes this. Different architectures may require different initialization steps, respectively, parameters. Consequently, the initialization procedure of the `Transducer` object becomes more complex: On the one hand, the `Transducer.__init__` method must consider any possible parameter for any encoder type. On the other hand, the same method is also responsible for initializing the correct type of encoder with the correct arguments.

Similar is true for the execution of the encoder. The encoder's forward pass may look very different for a different architecture. For instance, the transformers encoder, as opposed to an LSTM encoder, typically adds a positional encoding to the character embedding before the actual forward pass. Additionally, the encoder is accessed for different purposes represented by different methods (training, greedy decoding, beam search decoding). With the logic of the existing implementation, these differences must be taken care of at multiple execution points. This increases the code's complexity decreasing its understandability and making it more prone to errors.

To ensure the code's maintanability and understandability, any encoder architecture is therefore encapsulated in a separate method such that the initialization of the `Transducer` object is independent of the encoder type. Similarly, executing the encoder should logically be independent of the encoder type, i.e., any encoder architecture should require a single method call.

In a nutshell, encapsulation aims to represent encoders as standalone components with an encoder-independent interface.

Configuring the encoder using the CLI The implementation of the baseline model makes heavy use of the CLI. Any model parameter or hyperparameter can be configured using the CLI. The same should be possible for the choice of encoder type and any corresponding initialization parameter.

Optimizers and Learning rate schedulers Different encoders might require a different training procedure. This includes optimizers and possibly learning rate schedulers. For instance, the transformer was used initially with the Adam optimizer [Kingma and Ba, 2015] and is a popular choice in successful implementations [Vaswani et al., 2017; Wu et al., 2021]. However, the status quo exclusively uses the Adadelta optimizer [Zeiler, 2012]. Therefore, the model is extended to use different optimizers and learning rate schedulers. Specifically, the model should be able to use the Adam optimizer and the inverse square root warmup scheduler as proposed by [Vaswani et al., 2017]. The same criteria as discussed above apply for this part of the implementation. Each optimizer and scheduler should be encapsulated into a separate method, the handling in the code should be as independent as possible of the type of optimizer or scheduler, and these components should be easily configurable via CLI.

Extendability This thesis only aims to extend the model architecture with a transformer encoder. Future work might target different encoder architectures. Therefore, the code should be easily extendable without restructuring large parts of the code.

3.5.3 Implementation

A separate file for components In the first step, all components for the encoders are separated in a dedicated Python file `encoders.py`.[16] Similarly, the components for optimizers and learning rate schedulers are grouped in a Python file `optimizers.py`.[17] The separation reflects the design as separate and interchangable components of the `Transducer` class.

The encoder object Every encoder architecture is represented by a dedicated Python class in `encoders.py`. Consequently, to initialize the transducer with a specific encoder, the corresponding class is instantiated (line 92, Listing 3.20). To further abstract the initialization of an encoder, the `Transducer.__init__` receives an

[16]Implementation path: trans/encoders.py

[17]Implementation path: trans/optimizers.py

`argparse.Namespace` object instead of predefined keywords (line 65, Listing 3.20).[18] This argument is passed down to the `__init__` method of the encoder class, which in turn, uses the argument's attributes as initialization arguments (Listing 3.21).

```
63 class Transducer(torch.nn.Module):
64 def __init__(self, vocab: vocabulary.Vocabularies,
65              expert: optimal_expert.Expert, args: argparse.Namespace):
66     ...
92     self.enc = ENCODER_MAPPING[args.enc_type](args)
```

Listing 3.20: The initialization of `Transducer` and its encoder component.

```
@register_component('lstm', 'encoder')
class LSTMEncoder(torch.nn.LSTM):
    """LSTM-based encoder."""
    def __init__(self, args: argparse.Namespace):
        super().__init__(
            input_size=args.char_dim,
            hidden_size=args.enc_hidden_dim,
            num_layers=args.enc_layers,
            bidirectional=args.enc_bidirectional,
            dropout=args.enc_dropout,
            device=args.device
        )
    ...
```

Listing 3.21: The initialization of the LSTM encoder component.

The optimizer and scheduler object Similar to encoders, optimizers and learning rate schedulers are each represented as a separate Python class in `optimizers.py`.[19] Currently, the following optimizers are implemented: Adadelta (`Adadelta` class), Adam (`Adam` class) and AdamW (`AdamW` class) [Loshchilov and Hutter, 2019]. With respect to learning rate schedulers, two different options are available: Firstly, the `ReduceOnPlateau` class, which represents a scheduler for reducing the learning rate on a plateau (based on some evaluation metric). Secondly, the scheduler described by Vaswani et al. [2017], is implemented in the `WarmupInverseSquareRootSchedule` class.

Dynamically registering components Listings 3.20 and 3.21 also show another feature of the implementation: Components can be dynamically added to the model architecture with the decorator `@register_component` defined in `__init__.py` file.[20] Basically, `register_component` can be used to add named components to the model

[18]Implementation path: trans/transducer.py/Transducer.__init__

[19]Implementation path: trans/optimizers.py

[20]Implementation path: trans/__init__.py/register_component

architecture. The first argument of the function defines the name of the component, the second argument the type (encoder, optimizer or lr_scheduler). Depending on the type, the component is stored in a dedicated lookup dictionary: `ENCODER_MAPPING` for encoder components, `OPTIMIZER_MAPPING` for optimizer components and `SCHEDULER_MAPPING` for learning rate scheduler components. In the code, the component can then be accessed using the respective lookup and its name.

Dynamic configuration via CLI The encoder type and configuration are expressed as CLI options. Former is derived by `ENCODER_MAPPING`. A registered encoder component is made available via the CLI option `--enc-type` using its name as CLI value. To add encoder-specific CLI options, the static method `add_args` is added to the corresponding encoder class, as exemplified in Listing 3.22. The options defined in this method are only added to the CLI options if the corresponding encoder is chosen.

```python
@staticmethod
def add_args(parser: argparse.ArgumentParser) -> None:
    parser.add_argument("--enc-hidden-dim", type=int, default=200,
                        help="Encoder LSTM state dimension.")
    parser.add_argument("--enc-layers", type=int, default=1,
                        help="Number of encoder LSTM layers.")
    ...
```

Listing 3.22: The `add_args` method of the `LSTMEncoder` class.

The transformer encoder Generally, all implementations for encodesr, optimizers and learning rate schedulers are based on PyTorch implementations. An exception is the implementation for the transformer encoder: This implementation conceptually follows Vaswani et al. [2017] and is inspired by the implementation used by Wu et al. [2021][21]. More precisely, the implementation uses the PyTorch components `torch.nn.TransformerEncoderLayer`[22] (a single transformer layer) and `torch.nn.TransformerEncoder`[23] (a stack of a defined number of transformer layers), but relies on the implementation from Wu et al. [2021] for the positional encoding.[24]

[21]Wu et al. [2021] provide their open-source implementation here: https://github.com/shijie-wu/neural-transducer/tree/master/src

[22]https://pytorch.org/docs/1.10/generated/torch.nn.TransformerEncoderLayer.html

[23]https://pytorch.org/docs/1.10/generated/torch.nn.TransformerEncoder.html

[24]Initially, I have also experimented with positional encodings used in PyTorch tutorials. However, I have found that these implementations do not work well, which has caused me quite some time to realize.

3.6 Miscellaneous

3.6.1 Beam Search Decoding

The implementation for beam search decoding is not optimized to use batches. While this would be an interesting effort, the implementation is unequally more challenging than batched greedy decoding (Subsection 3.4.3). The reason is that in beam search decoding, multiple states for a single sequence are explored (of which the best performing is then selected), leading to more complex control flows than in greedy decoding. Efforts to implement batched greedy decoding would go beyond the scope of this thesis. Instead, the implementation is updated such that it is compatible with any other code changes (e.g., by ensuring GPU support).[25]

3.6.2 Roll-in

As discussed in Subsection 3.4.2.1, the mini-batch training implementation does not feature roll-in. My experiments (cf. Chapter 4) have led me to believe that, in some cases, using roll-in could notably improve performance. This has caused me to think about the possibilities of implementing roll-in in mini-batch training without drastically sacrificing efficiency (in terms of training speed). Therefore, I have drafted an implementation for roll-in. Given the scope of this thesis, I was, unfortunately, unable to finish the implementation or systematically experiment with it. The basic idea of the implementation would be to roll out the expert for an increasing number of training samples during training (at the beginning of each epoch). The model would then be increasingly exposed to its own predictions. The roll-in feature is developed on a separate branch *feature/roll-in*[26] and an open Pull Request[27] discusses some implementation details.

3.6.3 Additional Input Features

Specifically for the submission to the SIGMORPHON–UniMorph 2022 Shared Task on Typologically Diverse and Acquisition-Inspired Morphological Inflection Generation (discussed in Subsection 4.1.4 and 4.4.4), Peter Makarov implemented the possibility to add additional input features (e.g., morpho-syntactic features). Input

[25]Implementation path: trans/transducer.py/Transducer.beam_search_decode

[26]https://github.com/slvnwhrl/il-reimplementation/tree/feature/roll-in

[27]https://github.com/slvnwhrl/il-reimplementation/pull/12

features are represented as embeddings. Before the execution of the decoder, these feature embeddings are concatenated with the input characters embeddings serving as decoder input.[28]

3.6.4 Grid Search

To facilitate experimenting, I have implemented a Python-based CLI for grid search.[29] A JSON file can be used to specify (hyper)parameters and the program then automatically trains the model for any parameter combinations. The README file in my GitHub repository provides a short manual.[30]

[28]This commit gives an overview of the necessary changes: https://github.com/slvnwhrl/il-reimplementation/commit/c21376a1cf373c8785ad450c9fa5f4b4eb2da5fa

[29]Implementation path: trans/grid_search.py

[30]https://github.com/slvnwhrl/il-reimplementation#grid-search

4 Experiments

This chapter is dedicated to the evaluation of the implementations described in Chapter 3. The naming conventions defined in Table 4.1 are used to facilitate the discussion throughout this chapter, referencing the different software implementations.

Name	**Description**
baseline model	The original DyNet-based implementation that served as a starting point in this thesis (discussed in Section 2.2 and Subsection 3.1).
one-to-one reimplementation	The reimplementation of the baseline model in PyTorch with feature parity (discussed in Section 3.3).
mini-batch TF implementation	The implementation with efficient mini-batch training using teacher forcing. This implementation includes transformer-based encoders (discussed in Sections 3.4 and 3.5).

Table 4.1: Naming conventions for the different software implementations discussed in Chapter 4.

Section 4.1 gives an overview and a short description of all datasets used for the experiments. Section 4.2 describes the quantitative metrics that are used for the evaluation. Section 4.3 is dedicated to the analysis of the one-to-one reimplementation. Section 4.4 analyses the mini-batch TF implementation. Here, the analysis focuses on LSTM-based encoders, comparing the baseline model and one-to-one reimplementation. Lastly, Section 4.5 focuses on transformer-based encoders.

4.1 Datasets

The following subsections discuss the datasets on which the experiments are based. The choice for the datasets from the SIGMORPHON 2020 and 2021 Shared Tasks on Multilingual G2P (Subsections 4.1.1 and 4.1.2) is motivated by the fact that the baseline model was first used in the 2020 shared task and then, due to its success, served as a baseline in the 2021 shared task. Thus, the results of the baseline model on these datasets present a competitive benchmark.

The deadline for submissions to the SIGMOPRHON 2020 Shared Task on Morpheme Segmentation (Subsection 4.1.3) fell within the time span of this thesis. It matched my implementation schedule for the mini-batch TF implementation rather well. The considerably large dataset of this shared task presented a suitable possibility to challenge this implementation.
The SIGMORPHON–UniMorph 2022 Shared Task on Typologically Diverse and Acquisition-Inspired Morphological Inflection Generation (Subsection 4.1.4) had a similar deadline and presented the possibility of applying the mini-batch TF implementation in the context of a different morphological task and data scope.

4.1.1 SIGMORPHON 2020 Shared Task on Multilingual G2P (SIGMORPHON2020-G2P)

The dataset used in the SIGMORPHON 2020 Shared Task on Multilingual G2P (SIGMORPHON2020-G2P) features data for 15 different languages [Gorman et al., 2020]. Table 4.1.1 shows the languages and a training example for each language. The training dataset size is the same for all languages with $3,600$ examples per language. The task is evaluated using the WER and PER, where WER serves as the primary evaluation criterion.

Language	**Training data sample**	
	grapheme	phoneme
Adyghe	анэл	aː n a l
Armenian	լումա	l u m ɑ
Bulgarian	закон	z ə k ɔ n
Dutch	reuze	r øː z ə
French	peinture	p ɛ̃ t y ʁ
Greek (Modern)	λιμένος	l i m e n o s
Hungarian	hazánk	h ɒ z aː ŋ k
Icelandic	saltfiskur	s a l̥ t f ɪ s k ʏ r
Japanese (Hiragana)	がったい	ɡ a̠ t̚t a̠ i
Korean	근무하다	k ɯː n m u ɦ a̠ d a̠
Lithuanian	davusios	d aː ʋ ʊ sʲ o s
Romanian	absorbi	a b s o r b i
Vietnamese	ba	ʔ ɓ aː ˧˧

Table 4.2: Languages with a training data sample (*grapheme*: input, *phoneme*: output) for the SIGMORPHON2020-G2P dataset. Note that Georgian and Hindi are not shown.

4.1.2 SIGMORPHON 2021 Shared Task on Multilingual G2P (SIGMORPHON2021-G2P)

The dataset used in the SIGMORPHON 2021 Shared Task on Multilingual G2P (SIMORPHONG2021-G2P) is the continuation of the same task in 2020 (Subsection 4.1.1) [Ashby et al., 2021]. While the 2021 iteration features similar languages as the 2020 iteration, it emphasises resource availability. Three different data settings are created with a different number of available training data samples: *Low* (800 samples), *medium* (8,000 samples), and *high* (41,000 samples).[1] Armenian (Eastern), Bulgarian, Dutch, French, Georgian, Serbo-Croatian (Latin), Hungarian, Japanese (Hiragana), Korean and Vietnamese (Hanoi) are available for the low setting; Adyghe, Greek, Icelandic, Italian, Khmer, Latvian, Maltese (Latin), Roman, Slovenian, Welsh (Southwest) for the medium setting; English for the high setting.[2] The evaluation is performed based on the WER.

4.1.3 SIGMORPHON 2022 Shared Task on Morpheme Segmentation (SIGMORPHON2022-MS)

The SIGMORPHON 2022 Shared Task on Morpheme Segmentation (SIGMORPHON2022-MS, Batsuren et al. [2022]) consists of two parts: Part 1 is a word-level problem (i.e., inputs are single words), Part 2 is modelled as a sentence-level problem (i.e., inputs are whole sentences). Table 4.3 shows the languages and examples for both parts. Compared to the datasets for SIGMORPHON2020-G2P (Subsection 4.1.1) and SIGMORPHON2021-G2P (Subsection 4.1.2), the training dataset sizes are larger by many orders of magnitude: For part 1, the training data for each language contains on average around 43,300 samples (Mongolian with 15,171 samples being the smallest dataset and Hungarian with 742,239 samples the biggest dataset). For part 2, both Mongolian and Czech contain 1000 sentences (13,237 respecitvely 15,157 tokens). The English dataset is with 11,005 sentences (169,117 tokens) considerably larger.

[1]I have restricted my experiments to the low and medium settings following the submission of Clematide and Makarov [2021].

[2]The reader is referred to Table 4.1.1 for examples of the task.

	Language	Training data sample	
		unsegmented	segmented
part 1	Czech	autor	aut @@or
	English	enleaguing	en @@league @@ing
	French	âneries	âne @@erie @@s
	Hungarian	alakúig	áll @@k @@ú @@ig
	Spanish	tesas	teso @@ar @@as
	Italian	saltato	saltare @@ato
	Latin	īnferentīs	īnfere @@nt @@īs
	Russian	засасывав	за @@соснуть @@ать @@ывать @@л @@в
	Mongolian	ёсолгоотой	ёс @@лгоо @@той
part 2	Czech	Černá bankovní středa	Čern @@á bank @@ovn @@í střed @@a
	English	yeah , " things " ...	yeah , " thing @@s " ...
	Mongolian	Түүнээс хойш өссөн .	Түүнээс хойш өсөх @@сөн .

Table 4.3: Languages with a training data sample (*unsegmented*: input, *segmented*: output) for the SIGMORPHON2022-MS dataset.

In both parts, a system must be able to split the input into morphemes. The difference between the parts lies in the necessity of context: Part 1 effectively models a one-to-one mapping (each input word corresponds to exactly one segmentation). In contrast, the segmentation of a word may vary depending on the context (e.g., for homonyms). An example for Mongolian is shown in Table 4.4: The word эмээ is a homonym and either means *grandmother* (upper sentence) or *medicine* (lower sentence) depending on the context. It is only segmentable in the latter case leading to multiple theoretical possibilities.

Гэрт	**эмээ**	хоол	хийв	.
Гэр @@т	**эмээ**	хоол	хийх @@в	.
Grandmother cooked at home.				
Би	өдөр	**эмээ**	уусан	.
Би	өдөр	**эм @@ээ**	уух @@сан	.
Today I took my medicine.				

Table 4.4: An example in Mongolian demonstrating the segmentation ambiguity found in part 2 of the SIGMORPHON2022-MS data.

For both parts, the primary evaluation metric is a per-language F_1-score.[3] Precision and recall are calculated based on the number of *morphemes*, i.e., the measurement considers partially correct predictions.

[3]The task description [Batsuren et al., 2022] formulates additional evaluation metrics (precision, recall, and an edit distance). Following Wehrli et al. [2022], the analysis in this thesis is restricted to the primary metric.

4.1.4 SIGMORPHON–UniMorph 2022 Shared Task on Typologically Diverse and Acquisition-Inspired Morphological Inflection Generation (SIGMORPHON2022-INFL)

The SIGMORPHON–UniMorph 2022 Shared Task on Generalization and Typologically Diverse Morphological Inflection (SIGMORPHON2022-INFL) asks to predict an inflected word form given its lemma and a set of morphosyntactic features specified according to the UniMorph standard [Kodner et al., 2022]. Part 1 consists of 32 languages with **small** training sets (mostly 700 items, but for 4 languages only 70 to 240 items) and 21 **large** training sets (exactly 7,000 items).[4] Part 2 has an ablation-style setup for Arabic, English, and German: Each language has a dataset for each increment of 100, ranging from 100 to 600 (German) or 1,000 training samples (Arabic, English). Both tasks target the generalization capabilities of morphology learning systems by separately examining their test set performance on unseen lemmas and feature specifications. The evaluation is performed using accuracy.[5]

4.2 Metrics

4.2.1 Accuracy

Accuracy is a quantitative metric measuring the relative fraction of correctly predicted words for a given set of words [Géron, 2019]. It is expressed as a percentage. A higher percentage equals higher performance.

4.2.2 Precision, Recall, and F$_1$-Score

Precision, recall, and F$_1$-score are defined as

$$Precision = \frac{T_p}{T_p + F_p}, \tag{4.1}$$

$$Recall = \frac{T_p}{T_p + F_n}, \tag{4.2}$$

[4]Given the large number of languages, the reader is referred to Kodner et al. [2022] for a detailed overview. A single training sample consists of a lemma (input), the inflected word (target), and the additional morphosyntactic features. To give an example (Englisch): `learn learned V;PST`.

[5]This description is taken from Wehrli et al. [2022].

$$F_1\text{-}score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}, \tag{4.3}$$

where T_p describes the true positives, F_p the false positives an F_n the false negatives [Géron, 2019]. The higher precision and recall (respectively F_1-score), the better.

4.2.3 Word Error Rate (WER)

The word error rate (WER) is a quantitative measure and expresses the ratio of words for which the model's prediction does not match the gold reference [Gorman et al., 2020]. It is expressed as a percentage. The lower the WER, the better.

4.2.4 Phone Error Rate (PER)

The phone error rate (PER) measures the normalized distance between a prediction and the gold standard [Gorman et al., 2020]. The normalized distance is expressed as the number of edits (insertions, deletions, and substitutions) needed to transform the prediction into the gold standard. It is defined as:

$$PER := 100 \cdot \frac{\sum_i^n edits(p, r)}{\sum_i^n |r|}, \tag{4.4}$$

where p is the predicted sequence, r the gold sequence and $edits(p, r)$ references the Levenshtein distance between p and r. It is expressed in percentage. Similar to the WER, a smaller PER is considered better. Unlike WER, PER operates on the character-level allowing for partial correctness of predictions.

4.3 DyNet or Pytorch: Does It Make a Difference?

Goal This section compares the performance of the (DyNet-based) baseline model with its (PyTorch-based) one-to-one reimplementation. The goal of these experiments is to reproduce the results of the baseline model to validate feature parity performance-wise.

Experimental setup The model parameters and hyperparameters are based on the configuration of the baseline model used in SIGMORPHON2021-G2P [Ashby et al., 2021] if not stated otherwise.[6] All results are based on own training runs and

[6]A summary is given here: For all languages, input data is NFD-normalized. The SED expert is trained for 10 epochs. All models are trained for a maximum of 60 epochs (with a patience of

represent an average of 10 runs with fixed seeds. The presented results in this section are produced using greedy decoding.

4.3.1 SIGMORPHON2020-G2P

Results Table 4.5 compares the development and test set results for the baseline model (*DyNet*) with the one-to-one reimplementation (*PyTorch*). Per language, the average absolute difference measured in WER is 0.61% and 0.62% on the development, respectively, test set. For nine out of 15 languages, this difference is smaller than absolute 0.5% on the test set. The one-to-one reimplementation performs slightly better on both sets (absolute 0.47% and 0.56% lower WER). However, this must be seen in the context of the small set sizes: Both, the development and test set contain 450 samples. On average for the test set, the one-to-one reimplementation makes around 2.5 correct prediction more than the baseline model. Thus, the WER performance difference may be attributed to random variations (that depend on the choice of seeds), which may be less significant for bigger datasets.[7]

	dev							**test**						
	DyNet			PyTorch				DyNet			PyTorch			
Language	WER	SD	PER	WER	SD	PER	$\lvert\Delta\rvert$	WER	SD	PER	WER	SD	PER	$\lvert\Delta\rvert$
Adyghe	**25.73**	0.49	6.71	**24.82**	0.99	**6.38**	0.91	**29.89**	1.62	7.15	**28.51**	1.49	**6.72**	1.38
Armenian	16.53	0.86	**3.42**	**15.82**	0.58	**3.22**	0.71	**15.53**	1.57	3.41	**14.84**	0.99	**3.34**	0.69
Bulgarian	**34.16**	2.51	7.11	**31.51**	0.92	**6.86**	2.64	29.56	1.86	5.91	**28.67**	1.55	**5.60**	0.89
Dutch	21.04	0.57	**4.19**	**20.33**	0.50	**3.99**	0.71	20.24	0.64	3.76	**20.13**	0.97	**3.63**	0.11
French	16.47	0.82	**2.95**	**15.73**	0.85	**2.87**	0.74	**18.33**	1.29	3.28	**18.09**	1.25	**3.18**	0.25
Georgian	**10.40**	0.70	2.62	**9.78**	0.53	**2.49**	0.62	**8.93**	1.09	1.98	**7.49**	0.57	**1.68**	1.45
Greek (Modern)	27.33	1.64	4.90	**25.98**	1.32	**4.78**	1.36	29.29	1.68	4.99	**28.76**	2.02	**4.97**	0.53
Hindi	**15.67**	0.67	**2.80**	15.80	0.47	2.83	0.13	20.04	1.36	3.29	**17.82**	1.22	**2.95**	2.22
Hungarian	**6.16**	0.67	1.66	6.38	0.33	**1.60**	0.22	**7.58**	1.08	**1.78**	**7.25**	0.49	1.80	0.33
Icelandic	**2.98**	0.32	**0.62**	3.11	0.21	**0.62**	0.13	4.82	0.38	**1.21**	**4.78**	0.42	**1.09**	0.04
Japanese (Hiragana)	10.00	0.99	**2.12**	**9.96**	0.64	2.16	0.04	**10.69**	0.84	**2.30**	11.11	0.84	2.34	0.42
Korean	**8.15**	0.42	**2.49**	**8.15**	0.53	2.52	0.00	7.56	0.44	**1.97**	**7.40**	0.85	**1.91**	0.16
Lithuanian	20.82	0.85	3.74	**20.58**	1.08	**3.66**	0.24	28.09	1.41	4.92	**27.71**	1.41	**4.80**	0.38
Romanian	11.71	0.90	2.94	**11.58**	0.44	**2.91**	0.13	12.58	1.03	2.88	**12.14**	0.57	**2.77**	0.44
Vietnamese	**1.51**	0.23	**0.32**	2.00	0.23	0.49	0.49	**1.54**	0.13	**0.46**	1.58	0.20	0.52	0.04
AVG	15.24	0.84	3.24	**14.77**	0.64	**3.16**	0.61	16.31	1.09	3.29	**15.75**	0.99	**3.15**	0.62

Table 4.5: Results for the SIGMORPHON2020-G2P dataset comparing the baseline model (*DyNet*) with the one-to-one reimplementation (*PyTorch*). *SD* means standard deviation. $|\Delta|$ measures the absolute WER difference between *DyNet* and *PyTorch*.

Error similarity To get a better understanding of how similar the performance of the baseline model and one-to-one reimplementation is, Table 4.6 shows the five most

12 epochs) and are optimized using the Adadelta optimizer. Character and action embeddings have a dimension of 100. Both encoder and decoder are 1-layer LSTMs with a hidden dimension of 200. Gradient accumulation uses a batch size of five.

[7]Table A.1 reports results with beam search decoding.

frequent errors for the least and most similarly performing languages (Bulgarian and Korean) in terms of the WER. For both languages, the set of errors is identical, and the error frequencies are near-identical.

Bulgarian	DyNet	ɾ/r/25	ɔ/o/21	ɛ/◌̺/15	a/ə/15	ə/a/13
	PyTorch	ɾ/r/23	ɔ/o/19	ə/a/17	a/ə/15	ɛ/◌̺/14
Korean	DyNet	ɛ/ː/63	ː/ɛ/23	əː/ʌ̹/16	ʌ̹/əː/10	k͈/g/6
	PyTorch	ɛ/ː/63	ː/ɛ/24	əː/ʌ̹/21	ʌ̹/əː/11	k͈/g/6

Table 4.6: The five most common errors for Bulgarian and Korean for the baseline model (*DyNet*) and one-to-one reimplementation (*PyTorch*) in the test set of SIGMORPHON2020-G2P.

Conclusion The results strongly suggest that the PyTorch-based one-to-one reimplementation and DyNet-based baseline model perform very similarly: As measured by WER and PER and in terms of error similarity.

4.3.2 SIGMORPHON2021-G2P

Experimental setups Table 4.7 shows results for two experimental setups: Firstly, it compares results for the baseline model (*DyNet*) with the one-to-one reimplementation (*PyTorch*) for the baseline setup (*baseline*) similar to the experiments on the SIGMORPHON2020-G2P dataset (Subsection 4.3.1). Secondly, it shows experiments with a stacked LSTM encoder (2-layers) and different dropout probabilites (*2-layer LSTM encoder (PyTorch)*). These experiments test whether the additional complexity (i.e., the additional LSTM-layer) could be a cheap effort to increase the performance.

Results With respect to the baseline setup, the results for the SIGMORPHON21-G2P dataset resemble those for the SIGMORPHON2020-G2P dataset: The one-to-one reimplementation performs slightly better in both the low and medium data setting. For the former, the differences of the averages over all languages amount to absolute 0.94% and 0.45% on the development and test set. For the latter, the differences are absolute 0.34% and 0.24%.[8]

Influence of the test set size Again, these results should be put into perspective by the size of these datasets. The low setting of the SIGMORPHON2021-G2P

[8]Table A.4 reports results with beam search decoding.

dataset features 100 samples for the development and test set. Thus, the absolute differences between the two implementations are, on average, less than one prediction for both datasets. Figure 4.1 gives a more holistic picture and compares the average of absolute differences between both implementations in relation to development and test dataset size: The higher the sample size, the lower the average differences.

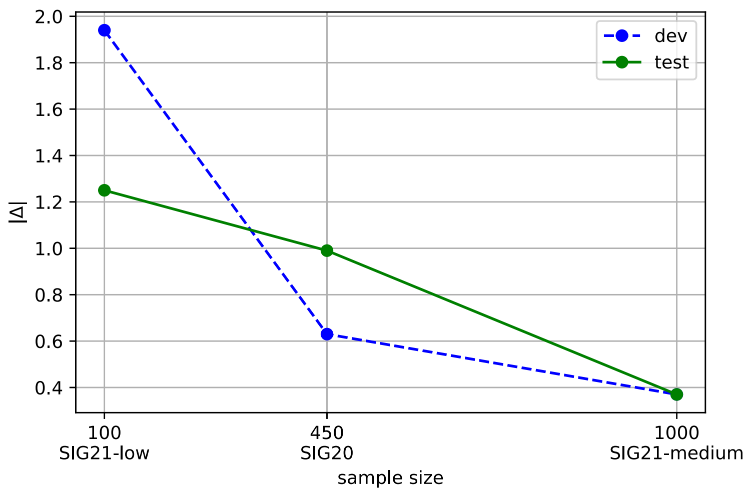

Figure 4.1: Average of absolute differences over all languages (|Δ|) between the baseline model and the one-to-one reimplementation for the SIGMORPHON2020-G2P (*SIG20*) and SIGMORPHON2021-G2P (*SIG21*) datasets.

Stacked LSTMs The additional model complexity (i.e., the additional model parameters of the second layer) does not improve performance in the low data setting. On average, over all languages, all stacked LSTM setups perform worse. However, with higher data settings additional model complexity seems to improve performance. While, on average, the performance gains of stacking LSTM encoders are small, one of the best models for eight out of 10 languages is a stacked LSTM-based model. Thus, stacking LSTM layers may be a cheap way to boost performance for specific languages. For instance, the performance improvement over the 1-layer PyTorch baseline model on the test set is absolute 0.80% for Bulgarian and absolute 0.92% for Serbo-Croatian.

Conclusion The results support the finding of the previous section: The baseline model and one-to-one reimplementation behave very similarly. Possible performance differences likely depend on the test set size. What is more, in larger datasets stacking LSTM encoders can improve performance for some languages.

		dev								**test**							
		baseline					2-layer LSTM encoder (PyTorch)			baseline					2-layer LSTM encoder (PyTorch)		
		DyNet		PyTorch			dp = 0.00	dp = 0.10	dp = 0.25	DyNet		PyTorch			dp = 0.00	dp = 0.10	dp = 0.25
	Language	WER	SD	WER	SD	$\|\Delta\|$	WER	WER	WER	WER	SD	WER	SD	$\|\Delta\|$	WER	WER	WER
	Adyghe	**25.50**	0.01	26.40	0.02	0.90	27.00	25.90	25.70	**26.50**	0.02	26.90	0.03	0.40	27.30	**26.50**	28.10
	Greek	7.60	0.01	**6.80**	0.01	0.80	6.90	6.80	6.90	**24.40**	0.01	**24.40**	0.02	0.00	25.90	25.90	24.90
	Icelandic	**16.10**	0.01	16.20	0.02	0.10	17.40	18.00	16.80	16.90	0.04	15.50	0.01	1.40	16.20	16.50	**15.10**
	Italian	25.80	0.02	**20.30**	0.01	5.50	23.50	22.70	22.40	**22.00**	0.02	23.50	0.02	1.50	25.50	25.20	25.00
	Khmer	**40.30**	0.02	41.70	0.02	1.40	41.90	42.10	40.70	39.10	0.03	39.60	0.03	0.50	**38.70**	39.30	38.80
low	Latvian	44.10	0.02	**42.20**	0.02	1.90	44.60	44.30	43.40	57.20	0.03	58.40	0.03	1.20	56.40	**53.80**	54.60
	Maltese (Latin)	21.90	0.02	**16.70**	0.01	5.20	17.60	17.50	18.30	20.40	0.02	20.00	0.03	0.40	**18.50**	20.00	20.90
	Romanian	10.60	0.01	11.50	0.01	0.90	11.50	**11.40**	**11.40**	**11.20**	0.01	11.60	0.01	0.40	12.30	12.20	12.80
	Slovenian	**47.50**	0.03	49.20	0.02	1.70	47.10	47.80	47.70	56.50	0.04	**50.10**	0.04	6.40	52.30	53.70	54.60
	Welsh (Southwest)	**18.70**	0.02	17.70	0.01	1.00	19.20	18.90	**18.70**	14.60	0.01	14.30	0.02	0.30	14.30	14.80	**14.20**
	AVG	25.81	0.02	**24.87**	0.02	1.94	25.67	25.54	25.20	28.88	0.02	**28.43**	0.02	1.25	28.74	28.79	28.90
	Armenian (Eastern)	5.44	0.00	**5.13**	0.00	0.31	5.30	5.28	5.16	7.56	0.00	7.03	0.01	0.53	**6.89**	6.94	7.04
	Bulgarian	12.45	0.02	10.78	0.01	1.67	10.87	**9.74**	9.97	20.16	0.01	19.72	0.03	0.44	19.65	20.59	**18.92**
	Dutch	12.79	0.00	12.93	0.01	0.14	12.48	**12.33**	12.54	17.00	0.01	17.42	0.01	0.42	17.36	**16.96**	17.26
	French	8.94	0.01	8.65	0.00	0.29	8.35	8.35	**8.13**	9.40	0.01	9.26	0.01	0.14	9.04	9.16	**8.98**
	Geogian	**0.00**	0.00	**0.00**	0.00	0.00	**0.00**	**0.00**	**0.00**	0.01	0.00	**0.00**	0.00	0.01	0.01	**0.00**	0.03
medium	Serbo-Croatian (Latin)	39.19	0.01	39.09	0.01	0.10	39.02	38.64	**38.20**	39.24	0.02	39.25	0.01	0.01	**38.33**	38.33	39.21
	Hungarian	1.72	0.00	1.37	0.00	0.35	1.48	**1.47**	1.52	2.39	0.00	1.72	0.00	0.67	**1.62**	1.66	1.63
	Japanese (Hiragana)	7.03	0.00	7.09	0.00	0.06	7.03	**6.93**	7.01	6.68	0.00	6.90	0.00	0.22	6.84	**6.58**	6.59
	Korean	20.71	0.01	19.93	0.01	0.78	**19.50**	19.92	20.04	19.41	0.01	**18.29**	0.01	1.12	18.47	18.49	18.81
	Vietnamese (Hanoi)	1.38	0.00	1.34	0.00	0.04	1.29	1.32	**1.24**	2.40	0.00	2.28	0.00	0.12	**2.22**	2.25	2.32
	AVG	10.97	0.01	10.63	0.00	0.37	10.53	10.40	**10.38**	12.43	0.01	12.19	0.01	0.37	**12.04**	12.10	12.08

Table 4.7: Results for the SIGMORPHON2021-G2P dataset comparing the baseline model (*DyNet*) with the PyTorch one-to-one reimplementation (*PyTorch*) for the baseline parameter configuration. *SD* means standard deviation. $|\Delta|$ measures the absolute WER difference between *DyNet* and *PyTorch*. *dp* means dropout probability.

4.4 LSTMs and Mini-Batch Training

Goal This section discusses experiments with the mini-batch TF implementation.

Subsections 4.4.1 and 4.4.2 present results for the SIGMORPHON2020-G2P and SIGMORPHON2021-G2P dataset, respectively. Results are compared to those of the one-to-one reimplementation (discussed in the previous section). Here, the idea is to examine whether the conceptual simplifications of the training routine in the mini-batch TF implementation lead to substantial performance changes.

Subsections 4.4.3 and 4.4.4 describe the submissions of Silvan Wehrli, Simon Clematide and Peter Makarov to SIGMORPHON2022-MS and SIGMOPRHON2022-INFL. In these subsections, submission details are summarized, and the submission results are presented. The discussion is based on Wehrli et al. [2022], and I refer the interested reader there for all details.[9]

Lastly, Subsection 4.4.5 gives an impression of the speed improvements for mini-batch training and batched greedy decoding.

Experimental setup If not stated otherwise, model parameters and hyperparameters follow the setup defined in Section 4.3. Note, however, that the mini-batch TF implementation uses real batches of size five, while the one-to-one reimplementation uses gradient accumulation (with batches of size five). All results are based on own training runs and represent an average of 10 runs with fixed seeds. Results are produced using greedy decoding.

4.4.1 SIGMORPHON2020-G2P

Results Table 4.8 compares the SIGMORPHON2020-G2P dataset results for the one-to-one reimplementation with the mini-batch TF implementation. Apart from Japanese and Vietnamese, the performance seems rather similar. Not considering these two languages, the WER average over all languages amounts to 16.26% and 16.34% on the development set and 17.48% and 17.80% for the one-to-one reimplementation, respectively, mini-batch TF implementation. However, when all languages are considered, the one-to-one reimplementation performs clearly better, both in terms of WER and PER: The WER difference on the test set amounts to almost absolute 5%, and the PER is around 50% smaller.

[9]The submission for SIGMORPHON2022-MS was largely based on my own work. Silvan Wehrli. However, Simon Clematide and Peter Makarov provided essential help during experimentation and writing. Simon Clematide largely shaped the submission for SIGMORPHON2022-INFL.

	dev							test						
	1:1			TF				1:1			TF			
Language	WER	SD	PER	WER	SD	PER	|Δ|	WER	SD	PER	WER	SD	PER	|Δ|
Adyghe	24.82	0.99	6.38	**24.22**	0.55	**6.22**	0.60	**28.51**	1.49	6.72	28.87	1.22	6.87	0.36
Armenian	**15.82**	0.58	**3.22**	17.33	6.34	3.49	1.51	**14.84**	0.99	3.34	17.07	7.01	3.74	2.22
Bulgarian	31.51	0.92	**6.86**	**30.69**	1.90	7.24	0.82	28.67	1.55	5.60	**28.51**	2.73	5.74	0.15
French	**9.78**	0.53	**2.49**	10.53	0.64	2.65	0.75	**7.49**	0.57	1.68	7.87	0.74	1.84	0.38
Georgian	25.98	1.32	**4.78**	**24.09**	0.85	4.86	1.89	28.76	2.02	4.97	**26.20**	1.70	4.98	2.56
Greek (Modern)	15.80	0.47	2.83	**15.47**	0.82	**2.80**	0.33	**17.82**	1.22	2.95	18.69	0.84	3.08	0.87
Hindi	**6.38**	0.33	**1.60**	7.16	0.63	1.76	0.78	**7.25**	0.49	1.80	8.09	1.09	1.92	0.84
Hungarian	**3.11**	0.21	**0.62**	4.25	1.65	0.90	1.14	**4.78**	0.42	1.09	5.91	1.92	1.38	1.13
Icelandic	9.96	0.64	2.16	**9.67**	0.54	**2.06**	0.29	**11.11**	0.84	2.34	11.74	0.78	2.55	0.62
Japanese (Hiragana)	**8.15**	0.53	**2.52**	21.91	18.75	8.02	13.76	**7.40**	0.85	1.91	21.16	18.57	10.65	13.76
Korean	**20.58**	1.08	3.66	**20.58**	0.67	**3.62**	0.00	27.71	1.41	4.80	**27.49**	0.86	4.79	0.22
Lithuanian	20.33	0.50	3.99	**20.22**	0.67	**3.88**	0.11	20.13	0.97	3.63	**19.16**	1.11	3.69	0.98
Romanian	11.58	0.44	2.91	**11.55**	0.57	**2.88**	0.02	**12.14**	0.57	2.77	12.18	1.42	2.81	0.04
Vietnamese	**2.00**	0.23	**0.49**	60.71	34.84	43.66	58.71	**1.58**	0.20	0.52	49.76	36.34	45.57	48.18
AVG	**14.77**	0.64	**3.16**	19.68	4.72	6.47	5.45	**15.75**	0.99	**3.15**	20.15	5.23	6.87	4.92

Table 4.8: Results for the SIGMORPHON2020-G2P dataset comparing the one-to-one reimplementation (*1:1*) with the mini-batch TF implementation (*TF*). *SD* means standard deviation and is in relation to the WER score. $|\Delta|$ measures the absolute WER difference between *1:1* and *TF*.

Model variance To investigate the huge performance difference for Japanese and Vietnamese, Figure 4.2 shows the model performance variance (measured in WER) on the test set. The one-to-one reimplementation clearly shows very little variance. While this is visually and analytically not the case for the mini-batch TF implementation, the visual analysis offers another insight: At least some of the models perform on a similar level as the one-to-one reimplementation. The models' training logs suggest that this is not a learning problem, i.e., for all models (independent of the WER performance), the training loss decreases steadily during training.[10] I suspect that this might be a consequence of the model not being exposed to its own mistakes during training.[11]

Insertion errors Test set outputs suggest that these failing models do not recognize the correct end of the transduction and instead produce seemingly random insertions. Table 4.9 shows such output for Japanese and Vietnamese with samples from two low-performing models. The models fail to stop the transduction at the right point and produce repeating characters.

Conclusion The mini-batch TF implementation performs similarly for many languages while, theoretically, allowing to scale efficiently. For some languages, however,

[10]Table A.2 in the appendix shows the complete training logs for the worst and best performing models for Japanese and Vietnamese.

[11]Simon Clematide and Peter Makarov reported that they experienced similar behaviour in early implementations of the baseline model without exploration at training time.

specific models may behave unreliably (depending on the seed initialization). Not using exploration at training time may explain this unreliability.

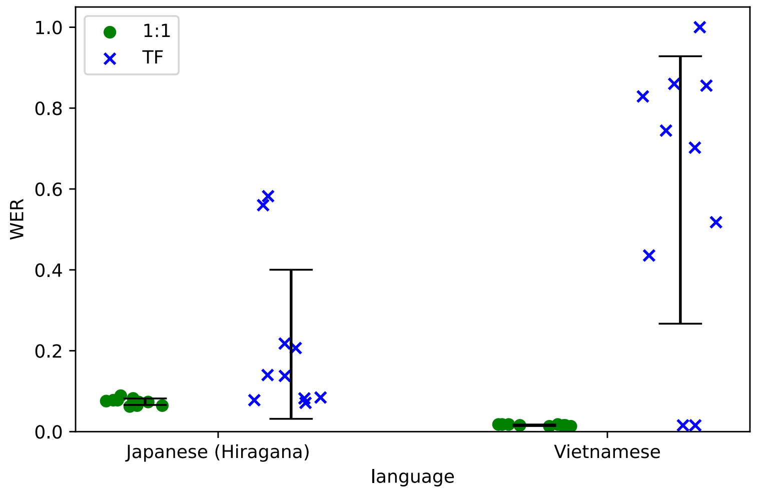

Figure 4.2: Model variance for Japanese and Vietnamese test set scores for the one-to-one reimplementation (*1:1*) and mini-batch TF implementation (*TF*).

Japanese	gold	a̠ t a̠ e̞ ɾ ɯ̟ᵝ
	samples	a̠ t a̠ e̞ ɾ ɯ̟ᵝ kʲ i e̞ kʲ i e̞ kʲ i e̞ kʲ i e̞
		a̠ t a̠ e̞ ɾ ɯ̟ᵝ ɯ̟ᵝ ɯ̟ᵝ ɯ̟ᵝ
Vietnamese	gold	ʔ ɓo m ˧˧
	samples	ʔ ɓo m ˧˧ ˧˧ ˧˧ ˧˧ ˧˧ ˧˧ ˧˧ ˧˧ ˧˧ ˧˧ ˧˧
		ʔ ɓo m ˧˧ ʔ ˧˧ ʔ ˧˧ ʔ ˧˧ ʔ ˧˧ ʔ ˧˧ ʔ ˧˧ ʔ ˧˧ ʔ ˧˧ ʔ ˧˧ ʔ ˧˧

Table 4.9: Samples for a typical test output of failing models for Japanese and Vietnamese in the SIGMORPHON2020-G2P dataset.

4.4.2 SIGMORPHON2021-G2P

Results The results in Table 4.10 present a rather similar picture compared to the results for the SIGMORPHON2020-G2P dataset (Table 4.8). For many languages, the performance is similar - with a few extreme exceptions. The model variance of Khmer (in the low setting), Japanese, and Vietnamese (in the medium setting) is substantial. Some models suffer from the previously discussed problem of over-insertion (Table 4.9).

		dev					**test**				
		1:1		TF			1:1		TF		
	Language	WER	SD	WER	SD	|Δ|	WER	SD	WER	SD	|Δ|
low	Adyghe	26.40	0.02	**25.50**	1.27	0.90	**26.70**	0.03	28.80	2.78	2.10
	Greek	**6.80**	0.01	7.10	1.45	0.30	**24.00**	0.02	25.60	2.80	1.60
	Icelandic	**16.20**	0.02	19.30	1.16	3.10	**15.60**	0.01	16.50	1.84	0.90
	Italian	**20.30**	0.01	20.40	2.01	0.10	**23.50**	0.02	26.30	3.20	2.80
	Khmer	**41.70**	0.02	51.60	17.12	9.90	**38.90**	0.04	53.70	18.54	14.80
	Latvian	**42.20**	0.02	44.90	1.73	2.70	57.10	0.03	**52.30**	4.24	4.80
	Maltese (Latin)	**16.70**	0.01	19.90	2.38	3.20	19.90	0.03	**17.10**	1.97	2.80
	Romanian	11.50	0.01	**10.50**	0.85	1.00	**11.80**	0.01	13.30	2.00	1.50
	Slovenian	**49.20**	0.02	50.20	1.93	1.00	**50.00**	0.04	53.50	3.06	3.50
	Welsh (Southwest)	**17.70**	0.01	19.30	1.49	1.60	**14.10**	0.02	14.30	1.70	0.20
	AVG	**24.87**	0.02	26.87	3.14	2.38	**28.16**	0.02	30.14	4.21	3.50
medium	Armenian (Eastern)	**5.13**	0.00	5.44	0.31	0.31	7.08	0.01	**6.75**	0.65	0.33
	Bulgarian	10.78	0.01	**10.07**	0.66	0.71	19.61	0.03	**18.72**	2.04	0.89
	Dutch	**12.93**	0.01	13.49	1.53	0.56	**17.32**	0.01	17.86	1.61	0.54
	French	**8.65**	0.00	8.94	0.57	0.29	**9.08**	0.01	9.39	0.56	0.31
	Geogian	**0.00**	0.00	**0.00**	0.00	0.00	**0.00**	0.00	**0.00**	0.00	0.00
	Serbo-Croatian (Latin)	39.09	0.01	**38.11**	0.83	0.98	39.12	0.01	**38.28**	0.79	0.84
	Hungarian	**1.37**	0.00	1.69	0.22	0.32	**1.67**	0.00	1.92	0.20	0.25
	Japanese (Hiragana)	**7.09**	0.00	12.45	8.48	5.36	**6.44**	0.00	11.99	8.56	5.55
	Korean	**19.93**	0.01	20.10	0.71	0.17	**18.18**	0.01	19.01	0.90	0.83
	Vietnamese (Hanoi)	**1.34**	0.00	47.72	35.30	46.38	**2.25**	0.00	47.54	34.80	45.29
	AVG	**10.63**	0.00	15.80	4.86	5.51	**12.08**	0.01	17.15	5.01	5.48

Table 4.10: Results for the SIGMORPHON2021-G2P dataset comparing the one-to-one reimplementation (*1:1*) with the mini-batch TF implementation (*TF*). *SD* means standard deviation. |Δ| measures the absolute WER difference between *1:1* and *TF*.

The influence of batch size Additionally, I experimented with different batch sizes. Figure 4.3 summarizes these experimental results. It shows a clear picture: The performance is inversely proportional to the batch size.[12]

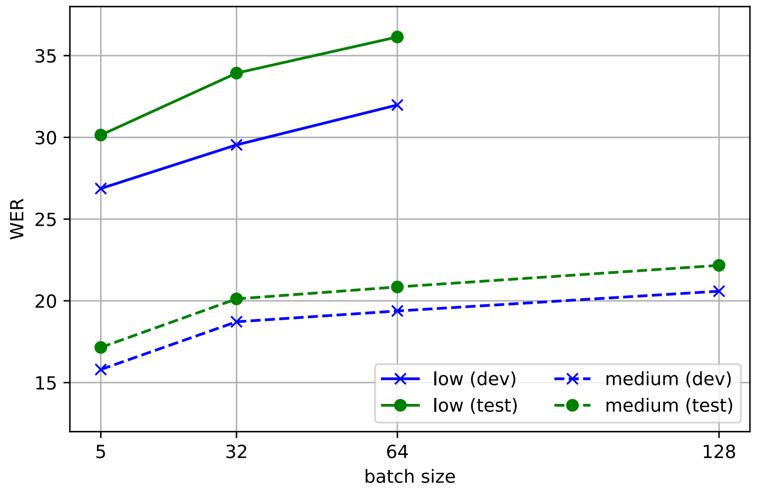

Figure 4.3: The influence of batch size on the SIGMORPHON2021-G2P dataset. Results represent the average over all languages.

[12]Detailed results are given in Table A.3 in the appendix.

Conclusion The experimental results for the SIGMORPHON2021-G2P dataset confirm the findings from the SIGMOPRHON2020-G2P dataset (Subsection 4.4.2): Generally, the mini-batch TF implementation performs similarly to the one-to-one reimplementation (which uses roll-in) but shows a high model performance variance for some languages.

4.4.3 SIGMORPHON2022-MS

4.4.3.1 Submission Details

Data preprocessing Besides NFD normalization as a preprocessing step, we substitute the multi-character morpheme delimiter (" @@") by a single character unseen in the data to decrease the length of the output.

Sentence-level segmentation We simplify part 2 of the shared task by reducing it to a word-level problem. Concretely, we split the input sentences into single word tokens and train the model on these word tokens, similarly to part 1. The single word predictions are then simply concatenated to form the original sentence. Since this completely neglects the context of the words, we have also experimented with POS tags as additional input features. We use TreeTagger [Schmid, 1999] to obtain the features.[13]

Model parameters We use a 2-layer stacked LSTM as the encoder and experimented with encoder dropout. We found the Adam optimizer [Kingma and Ba, 2015] to work well, as well as the scheduler that reduces the learning rate whenever a development set metric plateaus. We settled on a batch size of 32 for all models, which offers a good trade-off between model performance and training speed. Other hyperparameters (e.g. various embedding dimensions) are similar to the previous work [Makarov and Clematide, 2020].

Ensembling All our submissions are majority-vote ensembles. For part 1, we submit a 5-strong ensemble, **CLUZH**, composed of three models without encoder dropout and two models with encoder dropout of 0.1.[14]

[13]The parameter files are available at https://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/.

[14]Due to a mistake, the predictions by the models with dropout 0.1 were included twice, and a prepared model with dropout 0.25 was not used at all. However, the F_1 macro-average over all the languages for the intended ensemble on the development set is only 0.08 points higher.

For part 2, we submit three ensembles. All individual models have an encoder dropout probability of 0.25 and vary only in their use of features: **CLUZH-1** with three models without POS features, **CLUZH-2** with three models with POS tag features, and **CLUZH-3** with combines all the models from **CLUZH-1** and **CLUZH-2**.

4.4.3.2 Results and Discussion

Table 4.11 and Table 4.12 show our results for parts 1 and 2, respectively. Based on the macro-average F_1-score over all languages, our submission for part 1 ranks third out of 7 full submissions. For part 2, our submission CLUZH-3 was declared the winner out of 10 full submissions.[15]

	dropout = 0.0 (avg. of 3 models)		**dropout = 0.1** (1 model)		**dropout = 0.25** (1 model)		**ensemble** (5 models)		**best other**
Language	**dev**	**test**	**dev**	**test**	**dev**	**test**	**dev**	**test**	**test**
Czech	92.96	93.31	93.35	93.60	93.32	93.49	94.07	93.81	**93.88**
English	90.33	90.33	91.01	90.86	90.91	90.68	92.65	92.70	**93.63**
French	93.22	93.02	93.95	93.85	93.72	93.48	94.94	94.80	**95.73**
Hungarian	99.40	98.28	99.15	98.09	99.63	98.57	99.61	98.54	**98.72**
Spanish	97.79	97.78	98.57	98.61	98.53	98.56	98.71	98.74	**99.04**
Italian	95.54	95.54	96.15	96.19	96.02	96.11	96.93	96.93	**97.47**
Latin	99.20	99.20	99.30	99.26	99.30	99.23	99.40	99.37	**99.38**
Russian	97.52	97.54	96.38	96.43	96.65	96.54	98.58	98.62	**99.35**
Mongolian	98.21	97.73	98.47	97.80	98.47	97.90	98.53	98.12	**98.51**
AVG	96.02	95.86	96.26	96.08	96.28	96.06	97.05	96.85	**97.30**

Table 4.11: F_1-scores for SIGMORPHON2022-MS part 1.

	without features				**with POS tags**				**combined**		
	average (3 models)		ensemble (3 models)		average (3 models)		ensemble (3 models)		ensemble (6 models)		**best other**
Language	**dev**	**test**	**dev**	**test**	**dev**	**test**	**dev**	**test**	**dev**	**test**	**test**
Czech	94.06	90.90	94.54	91.35	94.15	91.15	94.45	91.76	94.72	**91.99**	91.76
English	98.12	89.27	98.31	89.47	98.18	89.29	98.38	89.47	98.41	89.54	**96.31**
Mongolian	85.95	81.57	87.06	82.22	86.24	81.84	87.26	82.55	87.62	**82.88**	82.59
AVG	92.71	87.25	93.30	87.68	92.86	87.43	93.36	87.93	93.58	88.14	**90.22**

Table 4.12: F_1-scores for SIGMORPHON2022-MS part 2. All models are trained with a dropout probability of 0.25.

Dropout The results for part 1 suggest that encoder dropout can help improve model performance. For some languages, the performance can improve by as much

[15]Our submission performed the best on two out of three languages (Czech and Mongolian). As it was beaten by another submission based on the macro F_1 average, two submissions were declared winners.

as 1% F_1-score absolute.

Ensembling Ensembling brings a clear improvement over single-best results. On average, the improvement is +0.55% on the development set and +0.53% on the test set (compared to the best single model result).

Gains from POS tags The results for part 2 suggest that treating a sentence-level problem as word-level may be a simple yet powerful strategy for morpheme segmentation. The success of this strategy depends on the language and the data. The more segmentation ambiguity a language has, the more important the context is. Mongolian has the highest segmentation ambiguity (Table 4.13). Around 1/5 of the tokens in the training data have at least two possible segmentations, whereas Czech and English exhibit little to no ambiguity. This may partially explain why the performance on the Mongolian data is the lowest. This also explains why using POS tags as additional features bring the biggest improvement for Mongolian: +0.29% and +0.27% on the development and test sets, based on the average of individual models. Using POS tags improves the prediction of ambiguous segmentation by an absolute 1.1% and 0.6% on the development and test sets for Mongolian (Table 4.14). When looking at the whole dataset, using POS features increases the relative number of correct predictions by 0.11% (development set) and 0.06% (test set) compared to not using the features. Using POS tags brings slight improvements and helps mitigate the loss of context.

	train		**dev**	
Language	1	≥2	1	≥2
Czech	100%	0%	100%	0%
English	99.58%	0.42%	99.75%	0.25%
Mongolian	77.91%	22.09%	90.00%	10.00%

Table 4.13: Segmentation ambiguity in SIGMORPHON2022-MS part 2: Relative frequency of unambiguous (1) vs ambiguous (≥ 2) word tokens.

dev			**test**		
ambiguous		all	ambiguous		all
NF	POS	Δ	NF	POS	Δ
63.0%	64.1%	+0.11%	59.5%	60.1%	+0.06%

Table 4.14: Impact of POS features on Mongolian, SIGMORPHON2022-MS part 2. *ambiguous* shows the average percentage of correctly predicted ambiguous segmentations for Mongolian. *NF* denotes models without features, *POS* denotes models using POS tags. *all* shows the absolute improvement for POS compared to NF, in relation to the whole dataset.

Token–type ratio Another reason for the lower performance of Mongolian might lie in the high variance in the data: The Mongolian training dataset contains around 40% unique tokens (Table 4.15). This is around four times more than in the English dataset. This makes the learning problem much harder, which is further exacerbated by the relatively small size of the data (compared to English).

	train		**dev**	
Language	total	unique	total	unique
Czech	15,157	5,126	7,545	3,217
English	169,117	17,249	21,444	4,849
Mongolian	13,237	5,293	6,632	3,216

Table 4.15: Word counts in SIGMORPHON2022-MS part 2: The total number of word forms and the number of unique words.

Conclusion Our competitive submission to this shared task demonstrates the approach's ability, i.e., the effectiveness of the mini-batch training implementation, to scale successfully to large datasets. Given the low ambiguity of the dataset in part 2 (apart from Mongolian), reducing sentence-level morpheme segmentation to a word-level problem presents a viable strategy. Conditioning on POS tags brings further improvements. We leave it to future work to explore more powerful representations of context.

4.4.4 SIGMORPHON2022-INFL

4.4.4.1 Submission Details

Data preprocessing For both parts, we apply NFD normalization to the input and split the UniMorph features at ";" by default. For languages that showed lower performance compared to the neural or non-neural baseline on the development set in part 1, we also computed models without NFD normalization and chose the best based on their development set performance. For Korean, we observed some Latin transliteration noise in the train/development set targets, which we removed before training. For Lamaholot (slp), we observed a very low accuracy (5%) on the development set compared to the neural baseline's 20% performance. By splitting UniMorph features at "+" as well as ";",[16] we achieved better generalization for this low-resource language (only 240 training examples available).

[16]For instance, `V;ARGAC2P+ARGNO2P;SBJV` would be split into 4 separate features.

Hyperparameters For small datasets in both parts: batch size one, a patience of 30 epochs, one-layer encoder and decoder with hidden size 200, character and action embeddings of size 100, feature embeddings of size 50, the AdamW optimizer Loshchilov and Hutter [2019] with a learning rate of 0.0005 (half of the default value), the reduce-learning-rate-on-plateau scheduler with factor 0.75, and beam decoding with beam width four. For a few languages whose development set performance was lower than that of the baselines, we computed models without NFD normalization and used those in case of improved accuracy.[17]

For large datasets in part 1, we made the following changes from the above: batch size 32, a patience of 20 epochs, action embeddings of size 200, a two-layer encoder with a hidden size of 1,000, a one-layer decoder with a hidden size of 2,000. In case of the development set performance was below that of any of the official baselines, we used some alternative hyperparameters:[18] no NFD normalization, batch size 16, a one-layer encoder with a hidden size of 2,000, a one-layer decoder with a hidden size of 4,000, and the Adadelta optimizer [Zeiler, 2012] with the default learning rate. Hyperparameters were not chosen using a systematic grid search or experimentation.

Convergence For the small datasets in part 1 with default hyperparameters and NFD normalization, we observe large differences in the number of epochs to convergence (mean 27.3, SD 22.8). For some languages, e.g. Chukchi (ckt), Ket (ket), and Ludian (lud), we see the best results on the first epoch, which typically means the model has just learned to copy the input to the output. For other languages, much larger or highly varying numbers of epochs to convergence are observed: Slovak (15-93), Karelian (13-88), Mongolian, Khalkha (19-61), and Korean (12-143).

For the large datasets in part 1 (7,000 training examples) with default hyperparameters and NFD normalization, we observe a mean of 17.3 epochs to convergence (SD 16.0). For Ludian, even in the large setting, the first epoch with copying gave the best results. In contrast, Georgian could generally profit from more epochs (mean 36.8, SD 17.9).

Ensembling Our submission for part 1 is a 5-strong majority-voting ensemble, and it is a 10-strong ensemble for part 2.

[17]Arabic, Gothic, Hungarian, and Old Norse.

[18]Arabic, Assamese, Evenki, Hungarian, Kazakh, Mongolian, Khalkha, and Old Norse.

4.4.4.2 Results and Discussion

The part 1 test set results are shown in Table 4.16. Given the large number of languages, we discuss the average accuracy on small and large training sets. An important goal for this shared task was to assess a system's performance on test data subsets defined by whether both the lemma and the feature specification were seen in the training data (+L +F in the Table), whether only the lemma (+L, -F), or only the feature specification (-L, +F) were seen, or whether neither of them (-L -F) appeared in the training data.

		seen status (± **L**emma/**F**eatures)			
System	Overall	+L +F	+L –F	-L +F	-L –F
Small dataset setting					
CLUZH	56.87	77.31	31.27	**77.97**	43.26
Best	**74.76**	**81.64**	72.91	**77.97**	70.87
Δ	-17.89	-4.33	-41.64	0.00	-27.62
Large dataset setting					
CLUZH	**67.85**	**90.99**	41.43	**87.17**	**60.30**
Best	62.39	89.57	**42.17**	85.31	55.56
Δ	5.46	1.43	-0.74	1.86	4.74

Table 4.16: Test results (accuracy macro-averaged over languages) for INFL part 1 split by training dataset size: large (7,000 training examples) vs small (up to 700 examples). Δ shows the difference between our submission and the best competitor covering the full set of languages.

Small datasets On the small datasets, our system only excels on the -L +F subset, meaning it is strong in modeling the behaviour of features. In the small dataset setting, the best competitor system, UBC, has an extremely strong performance in case the lemma is known (+L). It would be interesting to know what kind of information or data augmentation UBC uses: The neural baseline, which utilizes data augmentation, has a much lower performance (24.9%) than our submission. Overall, our submission with a 5-strong ensemble achieves the second-best result of the submissions covering all languages.

Large datasets In the large dataset setting, our submission shows the best performance overall. On the subset with seen lemmas and unseen features (+L -F), the neural baseline is the only system with slightly better results. This indicates that our system's modeling of lemmas is not yet optimal. The information flow in our architecture maybe dominated by the features (they are fed into the decoder at every action prediction step) and the aligned input character, and it may not have the best representation of the input lemma as a whole.

Trajectories The test set results for part 2 are shown in Figure 4.4. Our 10-strong ensemble was the clear overall winner in this low-resource track. It beats the best competing approaches by a substantial margin on the per-language average: Arabic 59.6% accuracy (best competitor OSU 57.5%), German 76.7% (non-neural baseline 74.8%), English 85.7% (OSU 81.5%).

Individual model performance varies, and the majority-vote ensembling improved the scores by 1.4% absolute on average on the test set. Interestingly, the difference between the average model performance and the ensemble performance does not get smaller with larger training sets.

The correlation between the increasing number of training examples and the improving test set performance is almost perfect for the average performance. Ensembles are slightly less stable.

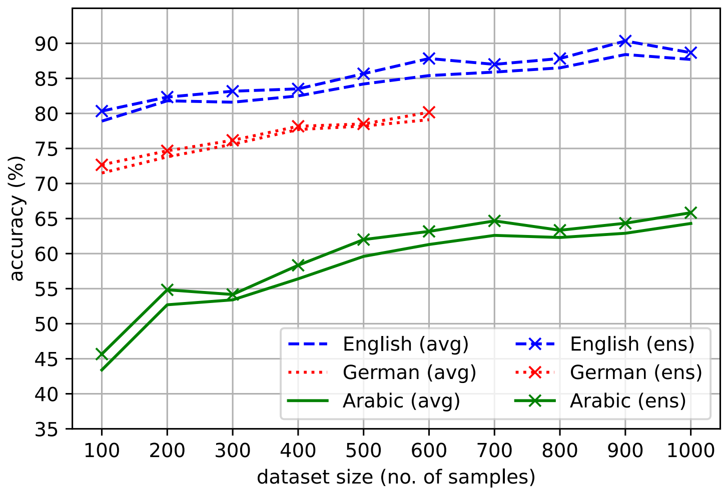

Figure 4.4: Test accuracy results for SIGMORPHON22-INFL part 2. avg=average, ens=10-strong ensemble.

Conclusion The submission presents strong results across data regimes. We note problems with capturing unseen lemmas, which may define future work.

4.4.5 Training and Inference Time

As part of the system description paper of the submission to the SIGMORPHON 2022 Shared Task on Morpheme Segmentation [Wehrli et al., 2022], I have evaluated the speed improvement of the mini-batch training TF implementation compared to the baseline model.[19] The results, shown in Table 4.17, report a clear speed improvement: For a batch size of 32, training is around three times faster on a CPU and close to 100 times faster on a GPU. For a batch size of 512, training is faster by a factor of over 250 on a GPU.

Table 4.17 also reports speed improvements for greedy decoding when using a GPU (compared to a CPU). Depending on the batch size, GPU-supported speeds up the inference process by a factor of over 10.[20]

	training			**greedy decoding**	
	BL	**TF**		**TF**	
Batch size	GA	CPU	GPU	CPU	GPU
1	27.49	18.96	**5.02**	**6.49**	10.00
32	23.58	7.48	**0.25**	2.92	**0.73**
64	23.89	7.46	**0.16**	2.84	**0.47**
128	24.69	7.88	**0.13**	2.88	**0.33**
256	27.14	8.21	**0.12**	3.01	**0.26**
512	31.11	8.51	**0.12**	3.26	**0.23**

Table 4.17: Mini-batch training and greedy decoding speed for the mini-batch TF implementation (*TF*) vs the baseline model (*BL*) . The *BL* models are trained on CPU using gradient accumulation (*GA*). All numbers are given in seconds and per 1,000 samples.

4.5 Transformer Encoder

Goal This section presents experiments with transformer-based encoders. Subsection 4.1.1 looks at experiments on the SIGMORPHON2020-G2P dataset and Subsection 4.5.2 at experiments on part 1 of the SIGMORPHON2022-MS dataset. The goal of these experiments is to evaluate whether transformer-based encoders can improve performance compared to LSTM-based encoders.

Experimental setup I have conducted many experiments with transformer-based encoders to examine the effects of different model parameters. These parameters

[19]For this purpose, I have trained models for the Armenian dataset of SIGMORPHON2020-G2P with different batch sizes. The training times represent averages of 20 epochs on the training set. The greedy decoding times are averages of 20 runs on the development set using a well-trained model. Model hyperparameters are identical to those of Makarov and Clematide [2020].

[20]Note that the precomputation of gold action sequences for the training data takes around 12 seconds per 1000 samples. However, this procedure is only required once per dataset as the precomputed output can be reused for any training run. In any case, the gains shown in Table 4.17 easily offset the additionally required time.

include the number of encoder layers, the number of heads, or hyperparameters such as the optimizer, learning rate scheduler, or batch size. However, the results for transformer-based encoders in this section all use the same setup (apart from the batch size). My experiments indicate that this setup works relatively well (compared to other parameter configurations), and I present these results in the following.[21] It is, in fact, similar to what Wu et al. [2021] suggest: Encoders consist of four layers with four heads, and models are optimized using the Adam optimizer (with a learning rate of 0.001). The feed-forward layer uses a hidden size of 1024. Additionally, the model uses an inverse square root scheduler with 4000 warmup steps (warmup scheduler, [Vaswani et al., 2017]). However, for some experiments, I use a scheduler that reduces the learning rate on a plateau (plateau scheduler)[22]. The models use a character embedding dimension of 256. The training is continued for a sufficiently large number of epochs such that models can fully converge.[23] If not stated otherwise, results follow these parameter configurations. All other model parameters follow the configuration of the baseline system for SIGMORPHON2020-G2P [Ashby et al., 2021], similar to Section 4.3. Results represent an average of five runs with fixed seeds and are produced using greedy decoding.

4.5.1 SIGMORPHON2021-G2P

Results With respect to the results in Table 4.18, transformer-based encoders perform clearly worse than the LSTM baseline[24]. In the low setting, no LSTM-based model is outperformed. In the medium setting, solely French performs better on the development and test set. Overall, the performance gap is smaller in the medium setting. Previously discussed results for LSTM-based encoders with mini-batch training have shown that performance variance may be large for some languages. This can also be observed here (e.g., for Italian, Maltese, and Vietnamese).

Dropout The results suggest that, as the dataset size decreases, a higher dropout value is more beneficial. Wu et al. [2021] report similar for their morphological reinflection results. However, my experiments indicate that this behaviour might

[21]Important additional experimental results are given in Appendix A.2.

[22]If not stated otherwise, the learning rate is reduced after 10 epochs by a factor of 0.5 of no improvement on the development set.

[23]My impression is that models with a transformer encoder need noticeably more time to converge compared to LSTM-only models. During experiments, I have chosen a high number of epochs (up to a few hundred) with a patience for early stopping of up to 100 epochs to rule out lower performance as an insufficient convergence.

[24]Results for the one-to-one reimplementation in Table 4.7 serve as baseline.

		dev					**test**				
			Transformer					Transformer			
	Language	LSTM baseline	dp = 0.1	dp = 0.3	SD	|Δ|	LSTM baseline	dp = 0.1	dp = 0.3	SD	|Δ|
	Adyghe	**26.40**	31.40	29.00	1.58	2.60	**26.70**	36.00	32.20	2.39	5.50
	Greek	**6.80**	14.60	12.00	1.22	5.20	**24.00**	33.60	30.80	1.79	6.80
	Icelandic	**16.20**	26.60	26.00	2.74	9.80	**15.60**	32.20	31.80	2.05	16.20
	Italian	**20.30**	40.40	46.80	7.26	26.50	**23.50**	42.20	47.00	6.48	23.50
	Khmer	**41.70**	45.80	43.00	1.22	1.30	**38.90**	51.00	47.00	2.24	8.10
low	Latvian	**42.20**	48.80	48.00	1.87	5.80	**57.10**	60.40	54.60	4.77	2.50
	Maltese (Latin)	**16.70**	27.20	27.00	3.32	10.30	**19.90**	33.80	33.60	3.51	13.70
	Romanian	**11.50**	13.60	12.40	1.14	0.90	**11.80**	23.60	21.80	2.77	10.00
	Slovenian	**49.20**	55.60	55.00	2.12	5.80	**50.00**	66.00	65.80	6.30	15.80
	Welsh (Southwest)	**17.70**	26.60	26.60	1.52	8.90	**14.10**	31.20	28.00	4.69	13.90
	AVG	**24.87**	33.06	32.58	2.40	7.71	**28.16**	41.00	39.26	3.70	11.60
	Armenian (Eastern)	**5.13**	5.16	5.64	0.15	0.03	**7.08**	7.40	8.16	0.23	0.32
	Bulgarian	**10.78**	11.86	13.82	0.67	1.08	**19.61**	23.24	24.36	4.40	3.63
	Dutch	12.93	12.88	**12.74**	0.78	0.05	**17.32**	18.84	18.72	0.47	1.52
	French	**8.65**	8.74	8.68	0.27	0.09	9.08	8.92	**8.78**	0.53	0.16
	Geogian	**0.00**	**0.00**	**0.00**	0.00	0.00	**0.00**	0.10	0.18	0.04	0.10
medium	Serbo-Croatian (Latin)	**39.09**	39.88	40.08	0.72	0.79	**39.12**	41.08	42.16	0.71	1.96
	Hungarian	**1.37**	1.70	1.72	0.13	0.33	**1.67**	2.08	2.08	0.22	0.41
	Japanese (Hiragana)	**7.09**	7.40	10.30	6.55	0.31	**6.44**	7.88	10.46	6.12	1.44
	Korean	19.93	19.62	**18.76**	0.13	0.31	**18.18**	19.04	18.64	0.81	0.86
	Vietnamese (Hanoi)	**1.34**	45.02	48.52	29.27	43.68	**2.25**	43.90	48.50	28.79	41.65
	AVG	**10.63**	15.23	16.03	3.87	4.67	**12.08**	17.25	18.20	4.23	5.21

Table 4.18: WER results for the SIGMORPHON2021-G2P dataset comparing transformer-based encoders with an LSTM baseline. *dp* means dropout probability. *SD* means standard deviation and is in relation to results with a dropout probability of 0.3 for *low* and 0.1 for *medium*. All models use a batch size of 128.

depend on specific hyperparameters: I do not observe this behaviour for a different number of heads and encoder layers (Table A.7). When using a different learning rate scheduler, results show ambiguity in the development and test set (Table A.8).

Batch size The results in Table 4.18 are based on a batch size of 128. The choice of this batch size is motivated by my experiments. Generally, a too low batch size seems to deteriorate performance (Table A.5). This is in stark contrast to what can be observed with LSTM-based encoders (Figure 4.3). However, my experiments also suggest that the benefit of a larger batch size might depend on the training dataset size. Table A.7 reports a clearly lower WER for the overall average in the low setting for a batch size of 32 (30.60% and 37.17% for the development and test set with a dropout probability of 0.3). The same table shows worse results for the medium setting (15.39% and 17.68% for the development and test set with a dropout probability of 0.1). However, compared to a batch size of 32, increasing the batch size to 64 results in a similar performance on the development set (15.35%) and test set (17.69%), as reported in Table A.9.

Optimizer and scheduler Generally, the choice of the optimizer and learning rate scheduler seems to affect the model performance substantially. My experiments suggest that the combination of Adam with a warmup scheduler works well (Tables A.6 and A.9).

Conclusion Overall, LSTM-based encoders seem superior in performance. For some languages, transformer-based encoders can outperform a high-performance LSTM baseline. What is more, the hyperparameter choice seems to have a significant impact on transformer performance.

4.5.2 SIGMORPHON2022-MS

Results Overall, the results in Table 4.19 show a clear performance advantage of the LSTM baseline[25] over the transformer-based model. While transformer-based models perform slightly better for some languages (Czech, Latin, and Mongolian on the development set and Latin on the test set), the LSTM baseline offsets this by much higher performance on all the other languages. Partial results for models with a dropout probability of 0.1 suggest that a lower dropout value might improve generalizability.[26]

	dev				**test**			
		Transformer				Transformer		
Language	LSTM baseline	dp = 0.1	dp = 0.3	SD	LSTM baseline	dp = 0.1	dp = 0.3	SD
Czech	92.96	**93.14**	93.00	0.25	**93.31**	93.26	92.96	0.32
English	**90.33**	85.11	79.08	14.59	**90.33**	85.04	79.12	14.54
French	**93.22**	-	86.45	4.30	**93.02**	-	86.30	4.25
Hungarian	**99.40**	-	91.93	11.18	**98.28**	-	91.23	10.70
Spanish	**97.79**	-	93.49	6.60	**97.78**	-	93.50	6.63
Italian	**95.54**	-	90.54	4.11	**95.54**	-	90.61	4.10
Latin	99.20	-	**99.31**	0.02	99.20	-	**99.28**	0.01
Russian	**97.52**	-	92.27	5.16	**97.54**	-	92.22	5.23
Mongolian	98.21	**98.39**	98.35	0.03	**97.73**	97.56	97.54	0.15
AVG	**96.02**	-	91.60	5.14	**95.86**	-	91.42	5.10

Table 4.19: F_1-score results for part 1 of SIGMORPHON2022-MS comparing transformer-based models with an LSTM baseline. *dp* means dropout probability. *SD* means standard deviation and is in relation to results with a dropout probability of 0.3. All models use a batch size of 256.

[25] Results for a dropout probability of 0.0 from Table 4.11 serve as baseline.

[26] Due to the size of the dataset, and given the scope of my thesis, I was unfortunately unable to perform large-scale experiments for all languages.

Model variance The high model performance variance for most languages is especially conspicuous. For instance, the standard variance for the English models with a dropout probability is around 15%. Low-performing models (in terms of WER) show similar behaviour to some LSTM models observed on the SIGMORPHON2020-G2P dataset (Subsection 4.4.1): The training loss decreases somewhat steadily, but the evaluation metric does not improve respectively decreases. Interestingly, I cannot report similar behaviour for LSTM encoders in the context of this data (Subsection 4.4.3).

Hyperparameters I have conducted a small hyperparameter study for Czech to better understand the influence of the optimizer and batch size. Table 4.20 compares two transformer-based encoder setups. One uses Adadelta and no scheduler (*Adadelta*), and one uses the Adam optimizer with the plateau scheduler (*Adam w/ plateau*).[27] In this context, I can report similar observations as for the SIGMORPHON2021-G2P dataset (Susbection 4.5.1). The choice of hyperparameters seems to influence the model performance significantly, and the benefit of single hyperparameters (e.g., larger batch size) seems to depend on other hyperparameter configurations (e.g., optimizer).

Adadelta			**Adam w/ plateau**	
64	256	512	64	256
92.25	91.78	91.05	88.17	92.31

Table 4.20: F_1-score results for Czech on the SIGMORPHON22-MS development dataset (part 1) for a transformer-based encoder comparing the F_1-score for different batch sizes for two different optimizer setups. Averages represent results of three models.

Conclusion The results tie in with the findings of the previous section (Subsection 4.5.1). However, transformer-based models show a much larger model variance than fully LSTM-based models. This leaves the question of whether transformer-based models might be more prone to the exposure bias.

[27]The learning rate is reduced after 10 epochs by a factor of 0.1 of no improvement on the development set.

5 Conclusion

Section 5.1 summarizes the motivation of this work. Section 5.2 emphasises the main results in relation to the research questions (Section 1.4). Finally, Section 5.3 outlines future work.

5.1 Thesis Overview

Neural transducers that use an encoder-decoder architecture to predict character-level edit actions have shown great success in many string transduction tasks. The model of Makarov and Clematide [2018a] presents such a model. It uses imitation learning to learn explicit edit actions derived from an expert policy. Variations of this approach have led to various successful shared task submissions: to the CoNLL-SIGMORPHON 2018 Shared Task on Universal Morphological Reinflection [Makarov and Clematide, 2018c], the SIGMORPHON 2020 Shared Task on Multilingual G2P [Makarov and Clematide, 2020] as well as the SIGMORPHON 2021 Shared Task on Multilingual G2P [Clematide and Makarov, 2021].
Despite the ongoing success, the approach may be improved on different levels: Firstly, the model uses the machine learning framework DyNet [Neubig et al., 2017]. DyNet's development has de facto stopped, and highly community-driven frameworks such as PyTorch [Paszke et al., 2019] have surpassed DyNet. Secondly, the model exclusively uses recurrent neural structures. The rise of the non-recurrent transformer architectures [Vaswani et al., 2017] and the recent success of such architectures in string transduction tasks [Wu et al., 2021] beg the question of whether such architecture can improve the general approach of [Makarov and Clematide, 2018a]. Lastly, the model's architecture is tailored for CPU training, limiting its application to data settings with at most a couple of thousand training samples. While datasets for string transduction tasks typically do not go beyond this scope (e.g., [Ashby et al., 2021]), the recent SIGMORPHON 2022 Shared Task on Morpheme Segmentation required models to train on hundreds of thousands of training samples [Batsuren et al., 2022]. Motivation for large-scale training also comes from recent research: Wu et al. [2021] report that a large batch size is critical for the successful

application of transformers in string transduction tasks.

This thesis addresses these shortcomings by improving the concrete neural transducer implementation of Makarov and Clematide [2020]. It does so by reimplementing the model in PyTorch (Research Question 1, Subsection 1.4.1), implementing GPU-supported mini-batch training and batched greedy decoding (Research Question 2, Subsection 1.4.2), as well as implementing transformer-based encoders (Research Question 3, Subsection 1.4.3). I have performed and evaluated experiments on different shared task datasets for string transduction tasks to examine the success of these implementations.

5.2 Main Results

Can the Model Be Ported from DyNet to PyTorch with Feature Parity?

The reimplementation of the baseline model [Makarov and Clematide, 2020] in PyTorch has proven successful. *Yes*, the model can be ported from DyNet to PyTorch with feature parity to answer the first research question (Subsection 1.4.1). The experiments on the dataset for the SIGMORPHON 2020 Shared Task on Multilingual G2P (Subsection 4.3.1) suggest very similar results, effectively replicating DyNet-based results of the model by Makarov and Clematide [2020]. While the Pytorch reimplementation shows a slight performance advantage overall, I attribute these differences to the small test set size. The evaluation of experiments on the dataset for the SIGMORPHON 2021 Shared Task On Multilingual G2P (Susbection 4.3.1) shows that with increasing test set size, the performance differences become smaller. These experiments also show that stacking LSTMs can be a simple way to improve performance. This improvement, however, seems to depend on the amount of training data (improved performance in the larger data setting).

Can the Model Scale to Higher Data Volume?

Based on the successful reimplementation in PyTorch, I have implemented GPU-supported mini-batch training and batched greedy decoding in PyTorch. To optimize efficiency in mini-batch training, the implementation uses *teacher forcing* [Williams and Zipser, 1989]. In this respect, the implementation diverges from the baseline model of Makarov and Clematide [2020] which uses exploration at training time (*roll-*

in). Not introducing the model to its own mistakes during training (i.e., through *roll-in*), theoretically exposes the model to the *exposure bias*, which can influence performance negatively [Wiseman and Rush, 2016]. The proposed implementation for mini-batch training might therefore present a tradeoff between speed and generalizability. In fact, experiments on the datasets of the SIGMORPHON 2020 and 2021 Shared Task On Multilingual G2P (Subsections 4.4.1 and 4.4.2) show that for a few languages model performance variance can be vast, suggesting, that in these cases, training speed comes at the cost of generalizability. Depending on the initialization, some of the models for these languages produce outputs with repeating characters (insertion errors). This is likely a consequence of not implementing exploration at training time. For any other languages, the performance is comparable to the PyTorch reimplementation with feature parity. Interestingly, we have not observed this behaviour in our competitive submission to the SIGMORPHON–UniMorph 2022 Shared Task on Typologically Diverse and Acquisition-Inspired Morphological Inflection Generation, which offers training datasets of comparable sizes (Subsection 4.4.4, Wehrli et al. [2022]). To fully leverage and evaluate the benefit of mini-batch training and batched greedy decoding, I have participated in the SIGMORPHON 2022 Shared Task on Morpheme Segmentation. The submission has turned out to be highly competitive. In part 1 of the shared task, the submission ranked second out of all teams and was declared winner in part 2 (Subsection 4.4.3, Wehrli et al. [2022]). Again, and in the context of this submission, I cannot report noticeably high model variance (insertion errors). What is more, an evaluation in the context of this submission shows that training is up to 250 times faster on a GPU depending on the batch size (compared to the baseline model of Makarov and Clematide [2020]). I can also report substantial speed improvements for greedy decoding used during inference (by a factor of over 10, depending on the batch size).
To come back to the second research question (Subsection 1.4.2): *Yes*, the model, as originally defined by Makarov and Clematide [2020], can be successfully scaled to higher data volumes. In a few cases, however, the proposed implementation for mini-batch training might lead to inconsistent model behaviour, which cannot be observed for the PyTorch reimplementation with feature parity.

Can a Transformer Encoder Beat an LSTM Encoder?

To answer the third research question right away (Subsection 1.4.3): *No*, given my experiments, transformer-based encoders cannot beat LSTM-based encoders. However, I think it is important to qualify this result and not consider it a final judgment. The experiments on the datasets for the SIGMORPHON 2021 Shared Task on Multilingual G2P (Subsection 4.5.1) and SIGMORPHON 2022 Shared Task on Morpheme Segmentation (Subsection 4.5.2) show that, generally, LSTM-based models perform better. For some languages, however, I can report better performance for transformer-based encoders. Nevertheless, these experiments also show that performance can vary significantly depending on the choices of (hyper)parameters. In this thesis, I have investigated the effect of some of these parameters (e.g., batch size, optimizer, or the type of learning rate scheduler), showing that parameters should be carefully set when working with transformers. For instance, batched training seems to be favourable and an important hyperparameter when working with transformer architectures (as reported by Wu et al. [2021]). As the scope of this thesis is limited, more investigation should be put into different parameters. As I have largely focused on hyperparameters, the effect of model parameters (such as the decoder) remains unexamined. Therefore, more experimentation is needed to better judge the potential benefits of transformer encoders.

5.3 Outlook

Overall, this thesis has provided several improvements for the model of Makarov and Clematide [2020]. I identify two interesting research areas for future work based on my results.

Performance variance As previously discussed, the proposed mini-batch training routine leads to high model performance variance in some cases. Future work should put effort into investigating this behaviour. Should the assumption be sustained that exploration at training time could decrease performance variance, effort should be directed towards implementing this feature in the context of efficient mini-batch training. My implementation draft for *roll-in*, as discussed in Section 3.6.2, offers a starting point.

Experiments with transformers Future work should continue to explore the application of transformer-based architectures in neural transducers. The experimental results for this thesis' transformer-based encoders serve as a baseline. More concretely, future experiments should investigate the influence of model parameters such as the character embedding dimension as well as the general structure of the LSTM decoder (number of layers, dropout, and the size of hidden layers). Furthermore, it is an open question whether an LSTM decoder and a transformer encoder are a fitting combination. Future experiments could address this question by experimenting with different decoder architectures in combination with a transformer encoder. Lastly, Dong et al. [2022] experiment with encoder pretraining, inspired by BERT [Devlin et al., 2019] and adapted to the G2P. Their results are promising and offer motivation to try similar pretraining methods.

References

R. Aharoni and Y. Goldberg. Morphological Inflection Generation with Hard Monotonic Attention. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2017. URL `https://aclanthology.org/P17-1183`.

L. F. Ashby, T. M. Bartley, S. Clematide, L. Del Signore, C. Gibson, K. Gorman, Y. Lee-Sikka, P. Makarov, A. Malanoski, S. Miller, O. Ortiz, R. Raff, A. Sengupta, B. Seo, Y. Spektor, and W. Yan. Results of the Second SIGMORPHON Shared Task on Multilingual Grapheme-to-Phoneme Conversion. In *Proceedings of the 18th SIGMORPHON Workshop on Computational Research in Phonetics, Phonology, and Morphology*, 2021. URL `https://aclanthology.org/2021.sigmorphon-1.13`.

D. Bahdanau, K. Cho, and Y. Bengio. Neural Machine Translation by Jointly Learning to Align and Translate. 2016. URL `http://arxiv.org/abs/1409.0473`.

K. Batsuren, G. Bella, A. Arora, V. Martinovic, K. Gorman, Z. Žabokrtský, A. Ganbold, Dohnalová, M. Ševčíková, K. Pelegrinová, F. Giunchiglia, R. Cotterell, and E. Vylomova. The SIGMORPHON 2022 Shared Task on Morpheme Segmentation. In *Proceedings of the 19th SIGMORPHON Workshop on Computational Research in Phonetics, Phonology, and Morphology*, July 2022. URL `https://aclanthology.org/2022.sigmorphon-1.11`.

A. W. Black, K. Lenzo, and V. Pagel. Issues in Building General Letter to Sound Rules. In *Third ESCA/COCOSDA Workshop on Speech Synthesis*, 1998. URL `https://www.isca-speech.org/archive_open/ssw3/ssw3_077.html`.

K.-W. Chang, A. Krishnamurthy, A. Agarwal, I. I. I. Hal Daumé, and J. Langford. Learning to Search Better than Your Teacher. In *Proceedings of the 32nd International Conference on Machine Learning*, 2015. URL `https://proceedings.mlr.press/v37/changb15.html`.

S. Clematide and P. Makarov. CLUZH at SIGMORPHON 2021 Shared Task on Multilingual Grapheme-to-Phoneme Conversion: Variations on a Baseline. In *Proceedings of the 18th SIGMORPHON Workshop on Computational Research in Phonetics, Phonology, and Morphology*, 2021. URL `https://aclanthology.org/2021.sigmorphon-1.17`.

R. Cotterell, N. Peng, and J. Eisner. Stochastic Contextual Edit Distance and Probabilistic FSTs. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, 2014. URL `http://aclweb.org/anthology/P14-2102`.

R. Cotterell, C. Kirov, J. Sylak-Glassman, D. Yarowsky, J. Eisner, and M. Hulden. The SIGMORPHON 2016 Shared TaskMorphological Reinflection. In *Proceedings of the 14th SIGMORPHON Workshop on Computational Research in Phonetics, Phonology, and Morphology*, 2016. URL `http://aclweb.org/anthology/W16-2002`.

R. Cotterell, C. Kirov, J. Sylak-Glassman, G. Walther, E. Vylomova, P. Xia, M. Faruqui, S. Kübler, D. Yarowsky, J. Eisner, and M. Hulden. CoNLL-SIGMORPHON 2017 Shared Task: Universal Morphological Reinflection in 52 Languages. In *Proceedings of the CoNLL SIGMORPHON 2017 Shared Task: Universal Morphological Reinflection*, 2017. URL `https://aclanthology.org/K17-2001`.

M. Creutz and K. Lagus. Unsupervised Discovery of Morphemes. In *Proceedings of the ACL-02 Workshop on Morphological and Phonological Learning*, 2002. URL `https://aclanthology.org/W02-0603`.

H. Daumé, J. Langford, and D. Marcu. Search-Based Structured Prediction. *Machine Learning*, 2009. URL `https://doi.org/10.1007/s10994-009-5106-x`.

J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, 2019. URL `https://aclanthology.org/N19-1423`.

S. Ding and P. Koehn. Parallelizable Stack Long Short-Term Memory. In *Proceedings of the Third Workshop on Structured Prediction for NLP*, 2019. URL `https://aclanthology.org/W19-1501`.

L. Dong, Z.-Q. Guo, C.-H. Tan, Y.-J. Hu, Y. Jiang, and Z.-H. Ling. Neural Grapheme-To-Phoneme Conversion with Pre-Trained Grapheme Models. In *ICASSP 2022 - 2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2022. URL `https://ieeexplore.ieee.org/document/9746447`.

M. Dreyer, J. R. Smith, and J. Eisner. Latent-Variable Modeling of String Transductions with Finite-State Methods. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing - EMNLP '08*, 2008. URL `http://portal.acm.org/citation.cfm?doid=1613715.1613856`.

M. Faruqui, Y. Tsvetkov, G. Neubig, and C. Dyer. Morphological Inflection Generation Using Character Sequence to Sequence Learning. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2016. URL `http://aclweb.org/anthology/N16-1077`.

J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. N. Dauphin. Convolutional Sequence to Sequence Learning. In *Proceedings of the 34th International Conference on Machine Learning*, 2017. URL `https://proceedings.mlr.press/v70/gehring17a.html`.

K. Gorman, L. F. Ashby, A. Goyzueta, A. McCarthy, S. Wu, and D. You. The SIGMORPHON 2020 Shared Task on Multilingual Grapheme-to-Phoneme Conversion. In *Proceedings of the 17th SIGMORPHON Workshop on Computational Research in Phonetics, Phonology, and Morphology*, 2020. URL `https://www.aclweb.org/anthology/2020.sigmorphon-1.2`.

A. Graves and J. Schmidhuber. Framewise Phoneme Classification with Bidirectional LSTM and Other Neural Network Architectures. *Neural Networks*, 2005. URL `https://www.sciencedirect.com/science/article/pii/S0893608005001206`.

A. Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 2nd Edition [Book]*. O'Reilly Media, Inc., 2nd edition, 2019.

B. Heinzerling and M. Strube. Sequence Tagging with Contextual and Non-Contextual Subword Representations: A Multilingual Evaluation. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019. URL `https://www.aclweb.org/anthology/P19-1027`.

S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. *Neural Computation*, 1997.

S. Jiampojamarn, G. Kondrak, and T. Sherif. Applying Many-to-Many Alignments and Hidden Markov Models to Letter-to-Phoneme Conversion. In *Human Language Technologies 2007: The Conference of the North American Chapter of the Association for Computational Linguistics; Proceedings of the Main Conference*, 2007. URL `https://aclanthology.org/N07-1047`.

K. Kann and H. Schütze. MED: The LMU System for the SIGMORPHON 2016 Shared Task on Morphological Reinflection. In *Proceedings of the 14th SIGMORPHON Workshop on Computational Research in Phonetics, Phonology, and Morphology*, 2016a. URL `https://aclanthology.org/W16-2010`.

K. Kann and H. Schütze. Single-Model Encoder-Decoder with Explicit Morphological Representation for Reinflection. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, 2016b. URL `http://aclweb.org/anthology/P16-2090`.

D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL `http://arxiv.org/abs/1412.6980`.

J. Kodner, S. Khalifa, K. Batsuren, H. Dolatian, R. Cotterell, F. Akkus, A. Anastasopoulos, T. Andrushko, A. Arora, N. Atanalov, G. Bella, E. Budianskaya, Y. Ghanggo Ate, O. Goldman, D. Guriel, S. Guriel, S. Guriel-Agiashvili, W. Kieraś, A. Krizhanovsky, N. Krizhanovsky, I. Marchenko, M. Markowska, P. Mashkovtseva, M. Nepomniashchaya, D. Rodionova, K. Scheifer, A. Sorova, A. Yemelina, J. Young, and E. Vylomova. SIGMORPHON–UniMorph 2022 Shared Task 0: Generalization and Typologically Diverse Morphological Inflection. In *Proceedings of the 19th SIGMORPHON Workshop on Computational Research in Phonetics, Phonology, and Morphology*, 2022. URL `https://aclanthology.org/2022.sigmorphon-1.19`.

V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. *Soviet physics doklady*, 1966.

I. Loshchilov and F. Hutter. Decoupled Weight Decay Regularization. In *International Conference on Learning Representations*, 2019. URL `https://openreview.net/forum?id=Bkg6RiCqY7`.

P. Makarov and S. Clematide. Imitation Learning for Neural Morphological String Transduction. In *Proceedings of the 2018 Conference on Empirical Methods in*

Natural Language Processing, 2018a. URL https://aclanthology.org/D18-1314.

P. Makarov and S. Clematide. Neural Transition-based String Transduction for Limited-Resource Setting in Morphology. In *Proceedings of the 27th International Conference on Computational Linguistics*, pages 83–93, 2018b. URL https://aclanthology.org/C18-1008.

P. Makarov and S. Clematide. UZH at CoNLL–SIGMORPHON 2018 Shared Task on Universal Morphological Reinflection. In *Proceedings of the CoNLL–SIGMORPHON 2018 Shared Task: Universal Morphological Reinflection*, 2018c. URL https://aclanthology.org/K18-3008.

P. Makarov and S. Clematide. CLUZH at SIGMORPHON 2020 Shared Task on Multilingual Grapheme-to-Phoneme Conversion. In *Proceedings of the 17th SIGMORPHON Workshop on Computational Research in Phonetics, Phonology, and Morphology*, 2020. URL https://aclanthology.org/2020.sigmorphon-1.19.

A. Michail, S. Wehrli, and T. Buckova. UZH OnPoint at Swisstext-2021: Sentence End and Punctuation Prediction in NLG Text Through Ensembling of Different Transformers. In *Proceedings of the Swiss Text Analytics Conference 2021*, 2021. URL http://ceur-ws.org/Vol-2957/sepp_paper2.pdf.

M. Mohri. Weighted Finite-State Transducer Algorithms. An Overview. In *Formal Languages and Applications*, Studies in Fuzziness and Soft Computing, pages 551–563, 2004. URL https://doi.org/10.1007/978-3-540-39886-8_29.

G. Neubig, C. Dyer, Y. Goldberg, A. Matthews, W. Ammar, A. Anastasopoulos, M. Ballesteros, D. Chiang, D. Clothiaux, T. Cohn, K. Duh, M. Faruqui, C. Gan, D. Garrette, Y. Ji, L. Kong, A. Kuncoro, G. Kumar, C. Malaviya, P. Michel, Y. Oda, M. Richardson, N. Saphra, S. Swayamdipta, and P. Yin. DyNet: The Dynamic Neural Network Toolkit. 2017. URL http://arxiv.org/abs/1701.03980.

H. Noji and Y. Oseki. Effective Batching for Recurrent Neural Network Grammars. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, 2021. URL https://aclanthology.org/2021.findings-acl.380.

A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and

S. Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

M. Popel and O. Bojar. Training Tips for the Transformer Model. *The Prague Bulletin of Mathematical Linguistics*, 2018. URL http://arxiv.org/abs/1804.00247.

K. Rao, F. Peng, H. Sak, and F. Beaufays. Grapheme-to-Phoneme Conversion Using Long Short-Term Memory Recurrent Neural Networks. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2015. URL https://ieeexplore.ieee.org/document/7178767.

S. Riezler, J. Kuhn, D. Prescher, and M. Johnson. Lexicalized Stochastic Modeling of Constraint-Based Grammars Using Log-Linear Measures and EM Training. In *Proceedings of the 38th Annual Meeting on Association for Computational Linguistics - ACL '00*, 2000. URL http://portal.acm.org/citation.cfm?doid=1075218.1075279.

A. Rios, C. Amrhein, N. Aepli, and R. Sennrich. On Biasing Transformer Attention Towards Monotonicity. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2021. URL https://aclanthology.org/2021.naacl-main.354.

E. Ristad and P. Yianilos. Learning String-Edit Distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1998.

S. Ross, G. Gordon, and D. Bagnell. A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, 2011. URL https://proceedings.mlr.press/v15/ross11a.html.

H. Schmid. Improvements in Part-of-Speech Tagging with an Application to German. In *Natural Language Processing Using Very Large Corpora*, 1999. URL https://doi.org/10.1007/978-94-017-2390-9_2.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Kaiser, and I. Polosukhin. Attention Is All You Need. In *Advances in Neural Information Processing Systems*, 2017. URL https://papers.nips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

E. Vylomova, J. White, E. Salesky, S. J. Mielke, S. Wu, E. M. Ponti, R. Hall Maudslay, R. Zmigrod, J. Valvoda, S. Toldova, F. Tyers, E. Klyachko, I. Yegorov, N. Krizhanovsky, P. Czarnowska, I. Nikkarinen, A. Krizhanovsky, T. Pimentel, L. Torroba Hennigen, C. Kirov, G. Nicolai, A. Williams, A. Anastasopoulos, H. Cruz, E. Chodroff, R. Cotterell, M. Silfverberg, and M. Hulden. SIGMORPHON 2020 Shared Task 0: Typologically Diverse Morphological Inflection. In *Proceedings of the 17th SIGMORPHON Workshop on Computational Research in Phonetics, Phonology, and Morphology*, 2020. URL `https://www.aclweb.org/anthology/2020.sigmorphon-1.1`.

S. Wehrli, S. Clematide, and P. Makarov. CLUZH at SIGMORPHON 2022 Shared Tasks on Morpheme Segmentation and Inflection Generation. In *19th SIGMORPHON Workshop on Computational Research in Phonetics, Phonology, and Morphology*, 2022. URL `https://aclanthology.org/2022.sigmorphon-1.21`.

R. J. Williams and D. Zipser. A Learning Algorithm for Continually Running Fully Recurrent Neural Networks. *Neural Computation*, 1989.

S. Wiseman and A. M. Rush. Sequence-to-Sequence Learning as Beam-Search Optimization. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, 2016. URL `https://aclanthology.org/D16-1137`.

S. Wu and R. Cotterell. Exact Hard Monotonic Attention for Character-Level Transduction. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019. URL `https://www.aclweb.org/anthology/P19-1148`.

S. Wu, P. Shapiro, and R. Cotterell. Hard Non-Monotonic Attention for Character-Level Transduction. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, 2018. URL `https://aclanthology.org/D18-1473`.

S. Wu, R. Cotterell, and M. Hulden. Applying the Transformer to Character-Level Transduction. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, 2021. URL `https://aclanthology.org/2021.eacl-main.163`.

Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. R. Salakhutdinov, and Q. V. Le. XLNet: Generalized Autoregressive Pretraining for Language Understanding. In *Advances in Neural Information Processing Systems*, 2019. URL `https://proceedings.neurips.cc/paper/2019/hash/dc6a7e655d7e5840e66733e9ee67cc69-Abstract.html`.

K. Yao and G. Zweig. Sequence-to-Sequence Neural Net Models for Grapheme-to-Phoneme Conversion. In *Interspeech 2015*, 2015. URL `https://www.isca-speech.org/archive/interspeech_2015/yao15_interspeech.html`.

S. Yolchuyeva, G. Németh, and B. Gyires-Tóth. Transformer Based Grapheme-to-Phoneme Conversion. In *Interspeech 2019*, 2019. URL `http://arxiv.org/abs/2004.06338`.

M. D. Zeiler. ADADELTA: An Adaptive Learning Rate Method, 2012. URL `https://arxiv.org/abs/1212.5701`.

A Results

A.1 Experiments with LSTM Encoders

A.1.1 SIGMORPHON2020-G2P

	dev							**test**						
	DyNet			PyTorch				DyNet			PyTorch			
Language	WER	SD	PER	WER	SD	PER	|Δ|	WER	SD	PER	WER	SD	PER	|Δ|
Adyghe	**25.53**	0.57	6.56	**25.18**	1.13	**6.37**	0.36	**29.56**	1.36	7.03	**28.16**	1.63	**6.57**	1.40
Armenian	16.73	1.05	**3.42**	**15.62**	0.59	**3.17**	1.11	**15.49**	1.63	3.39	**14.71**	0.92	**3.29**	0.78
Bulgarian	**34.13**	2.59	7.08	**31.42**	0.91	**6.83**	2.71	29.49	1.95	5.88	**28.42**	1.50	**5.55**	1.07
Dutch	21.02	0.52	**4.14**	**20.22**	0.50	**3.95**	0.80	20.18	0.70	3.73	**20.02**	0.86	**3.60**	0.16
French	16.38	0.88	**2.91**	**15.85**	0.94	**2.87**	0.53	**18.13**	1.16	3.21	**17.67**	1.09	**3.09**	0.47
Georgian	**10.40**	0.53	2.61	**9.87**	0.61	**2.50**	0.53	**8.93**	1.13	1.96	**7.56**	0.58	**1.70**	1.38
Greek (Modern)	27.33	1.64	4.90	**25.98**	1.34	**4.80**	1.36	29.29	1.68	4.99	**28.58**	2.05	**4.98**	0.71
Hindi	**15.78**	0.85	**2.80**	**15.71**	0.44	2.82	0.07	20.02	1.23	3.29	**17.76**	1.15	**2.92**	2.26
Hungarian	**6.13**	0.70	1.64	6.29	0.42	**1.57**	0.15	**7.31**	0.95	**1.70**	**7.09**	0.35	1.76	0.22
Icelandic	**3.05**	0.36	**0.61**	3.22	0.22	**0.65**	0.18	4.87	0.37	**1.19**	**4.78**	0.42	**1.06**	0.09
Japanese (Hiragana)	10.11	1.06	**2.12**	**9.96**	0.68	2.14	0.15	**10.56**	0.83	**2.25**	11.15	0.88	2.31	0.60
Korean	**7.89**	0.45	**2.39**	**7.67**	0.57	**2.34**	0.22	**7.07**	0.63	**1.87**	**11.58**	0.44	**2.91**	4.51
Lithuanian	20.96	0.89	3.74	**20.44**	1.00	**3.59**	0.51	28.09	1.46	4.86	**27.40**	1.46	**4.70**	0.69
Romanian	**11.71**	0.93	2.94	**11.58**	0.46	**2.92**	0.13	12.49	1.05	2.88	**12.07**	0.59	**2.75**	0.42
Vietnamese	**1.42**	0.30	**0.29**	2.00	0.18	0.48	0.58	**1.47**	0.19	**0.42**	1.51	0.14	0.45	0.04
AVG	15.24	0.89	3.21	**14.73**	0.67	**3.13**	0.63	16.20	1.09	3.24	**15.90**	0.94	**3.18**	0.99

Table A.1: Results for the SIGMORPHON2020-G2P dataset comparing the baseline model (*DyNet*) with the one-to-one PyTorch reimplementation (*PyTorch*). WER results represent an average of 10 runs and are produced using beam search decoding with a beam width of four. *SD* means standard deviation. |Δ| measures the absolute WER difference between *DyNet* and *PyTorch*.

	Japanese						**Vietnamese**					
	train loss		train acc.		dev acc.		train loss		train acc.		dev acc.	
Epoch	B	W	B	W	B	W	B	W	B	W	B	W
1	0.6457	0.6361	0.6556	0.0000	0.6622	0.0000	0.5978	0.5817	0.7389	0.1056	0.7733	0.1044
2	0.0778	0.0697	0.9389	0.0000	0.9178	0.0022	0.0835	0.0860	0.8556	0.1056	0.8489	0.0644
3	0.0370	0.0301	0.9556	0.0000	0.9489	0.0022	0.0526	0.0548	0.8944	0.1389	0.9089	0.0778
4	0.0246	0.0186	0.9778	0.0000	0.9533	0.0022	0.0387	0.0416	0.8833	0.4056	0.8489	0.4067
5	0.0181	0.0126	0.9722	0.0000	0.9533	0.0022	0.0298	0.0324	0.9056	0.0556	0.9022	0.0222
6	0.0149	0.0093	0.9778	0.0000	0.9644	0.0022	0.0247	0.0258	0.8944	0.1611	0.8667	0.1067
7	0.0121	0.0071	0.9833	0.0000	0.9733	0.0022	0.0201	0.0212	0.9333	0.3222	0.9067	0.2800
8	0.0100	0.0058	0.9833	0.0000	0.9689	0.0022	0.0164	0.0182	0.9389	0.1111	0.8756	0.0400
9	0.0090	0.0047	0.9833	0.0000	0.9689	0.0022	0.0137	0.0150	0.9500	0.0889	0.9133	0.0267
10	0.0075	0.0039	0.9944	0.0000	0.9733	0.0022	0.0120	0.0129	0.9556	0.2444	0.9111	0.1844
11	0.0064	0.0034	0.9944	0.0111	0.9711	0.0022	0.0106	0.0113	0.9833	0.1611	0.9067	0.1111
12	0.0056	0.0028	0.9944	0.0111	0.9711	0.0022	0.0088	0.0095	0.9222	0.0333	0.8333	0.0022
13	0.0052	0.0024	0.9833	0.0111	0.9733	0.0022	0.0078	0.0081	0.9722	0.1278	0.9022	0.0756
14	0.0047	0.0021	0.9944	0.0111	0.9756	0.0022	0.0063	0.0066	0.9778	0.0833	0.8933	0.0200
15	0.0039	-	0.9944	-	0.9778	-	0.0054	0.0056	0.9833	0.0667	0.9178	0.0089
16	0.0031	-	0.9944	-	0.9756	-	0.0049	0.0050	0.9889	0.0722	0.9067	0.0067
17	0.0028	-	0.9944	-	0.9756	-	0.0043	-	0.9944	-	0.8911	-
18	0.0025	-	0.9944	-	0.9778	-	0.0032	-	0.9722	-	0.8867	-
19	0.0022	-	0.9944	-	0.9778	-	0.0026	-	0.9889	-	0.9044	-
21	0.0018	-	0.9944	-	0.9756	-	0.0021	-	0.9889	-	0.8956	-
22	0.0017	-	0.9944	-	0.9733	-	0.0016	-	0.9556	-	0.8867	-
23	0.0013	-	0.9944	-	0.9778	-	0.0014	-	0.9833	-	0.9022	-
24	0.0016	-	0.9944	-	0.9756	-	0.0012	-	0.9833	-	0.8978	-
25	0.0011	-	1.0000	-	0.9756	-	0.0011	-	0.9778	-	0.8933	-
26	0.0008	-	1.0000	-	0.9778	-	0.0012	-	0.9667	-	0.8911	-
27	0.0006	-	1.0000	-	0.9778	-	0.0007	-	0.9833	-	0.8956	-
28	0.0006	-	1.0000	-	0.9778	-	0.0006	-	0.9833	-	0.8933	-

Table A.2: The training logs for the best (B) and worst (W) performing models for Japanese and Vietnamese on the SIGMOPRHON2020-G2P dataset.

A.1.2 SIGMORPHON2021-G2P

	dev					**test**				
	5		32	64	128	5		32	64	128
Language	WER	SD	WER	WER	WER	WER	SD	WER	WER	WER
Adyghe	**25.50**	1.27	25.70	26.70	-	**28.80**	2.78	31.00	31.20	-
Greek	**7.10**	1.45	9.00	9.70	-	**25.60**	2.80	28.00	30.20	-
Icelandic	**19.30**	1.16	22.60	26.10	-	**16.50**	1.84	20.80	22.60	-
Italian	**20.40**	2.01	25.90	32.60	-	**26.30**	3.20	31.80	33.50	-
Khmer	**51.60**	17.12	53.90	55.00	-	**53.70**	18.54	56.30	56.90	-
Latvian	**44.90**	1.73	48.40	50.60	-	**52.30**	4.24	56.70	59.50	-
Maltese (Latin)	**19.90**	2.38	22.90	26.80	-	**17.10**	1.97	22.80	30.50	-
Romanian	10.50	0.85	10.10	**9.80**	-	**13.30**	2.00	14.60	13.60	-
Slovenian	**50.20**	1.93	55.50	59.80	-	**53.50**	3.06	57.90	63.30	-
Welsh (Southwest)	**19.30**	1.49	21.40	22.70	-	**14.30**	1.70	19.40	20.10	-
AVG	**26.87**	3.14	29.54	31.98	-	**30.14**	4.21	33.93	36.14	-
Armenian (Eastern)	**5.44**	0.31	5.82	5.79	6.60	**6.75**	0.65	7.46	7.63	8.84
Bulgarian	**10.07**	0.66	12.46	13.74	16.78	**18.72**	2.04	20.18	21.51	24.26
Dutch	13.49	1.53	13.57	**14.83**	15.63	**17.86**	1.61	18.87	21.01	21.96
French	**8.94**	0.57	9.24	9.63	10.47	**9.39**	0.56	9.60	9.92	10.43
Geogian	**0.00**	0.00	**0.00**	0.01	0.01	**0.00**	0.00	0.03	0.03	0.06
Serbo-Croatian (Latin)	**38.11**	0.83	40.79	43.19	46.07	**38.28**	0.79	42.04	44.31	47.53
Hungarian	**1.69**	0.22	2.15	2.49	2.57	**1.92**	0.20	2.24	2.74	2.83
Japanese (Hiragana)	**12.45**	8.48	21.10	21.60	22.08	**11.99**	8.56	20.70	21.69	22.65
Korean	**20.10**	0.71	20.29	20.50	21.01	19.01	0.90	18.95	**18.53**	18.93
Vietnamese (Hanoi)	**47.72**	35.30	61.74	62.05	64.70	**47.54**	34.80	61.15	61.16	64.23
AVG	**15.80**	4.86	18.72	19.38	20.59	**17.15**	5.01	20.12	20.85	22.17

Table A.3: Results for the SIGMORPHON2021-G2P dataset comparing different batch sizes for the mini-batch TF implementation (*TF*). WER results represent an average of 10 runs and are produced using greedy decoding. *SD* reports the standard deviation for a batch size of five.

		dev								**test**							
		baseline					2-layer LSTM encoder (PyTorch)			baseline					2-layer LSTM encoder (PyTorch)		
		DyNet		PyTorch			dp = 0.00	dp = 0.10	dp = 0.25	DyNet		PyTorch			dp = 0.00	dp = 0.10	dp = 0.25
	Language	WER	SD	WER	SD	$\|\Delta\|$	WER	WER	WER	WER	SD	WER	SD	$\|\Delta\|$	WER	WER	WER
	Adyghe	**25.30**	0.01	26.90	0.02	1.60	28.00	25.80	26.00	**25.80**	0.02	26.70	0.03	0.90	27.30	26.20	27.00
	Greek	7.60	0.02	**7.10**	0.01	0.50	7.10	7.10	**6.90**	**23.80**	0.01	24.00	0.02	0.20	25.40	25.70	24.40
	Icelandic	**16.50**	0.02	16.80	0.03	0.30	17.80	18.10	17.10	16.90	0.03	15.60	0.01	1.30	15.90	16.40	**14.90**
	Italian	25.90	0.02	**20.60**	0.02	5.30	23.90	22.80	22.80	**21.50**	0.02	23.50	0.02	2.00	25.30	24.50	24.80
	Khmer	**40.10**	0.01	41.80	0.02	1.70	41.40	41.50	40.60	**38.00**	0.03	38.90	0.04	0.90	39.20	38.70	38.20
low	Latvian	43.70	0.02	42.20	0.01	1.50	44.50	44.60	**43.50**	56.60	0.03	57.10	0.03	0.50	56.30	**53.80**	54.70
	Maltese (Latin)	21.80	0.02	**16.40**	0.02	5.40	17.40	17.70	18.10	20.20	0.02	19.90	0.03	0.30	**18.50**	19.70	20.30
	Romanian	**11.20**	0.01	11.70	0.01	0.50	11.60	11.90	11.90	**10.60**	0.02	11.80	0.01	1.20	12.80	12.70	12.70
	Slovenian	47.40	0.03	49.20	0.02	1.80	**47.10**	47.60	48.20	55.90	0.04	**50.00**	0.04	5.90	52.30	53.10	54.50
	Welsh (Southwest)	18.60	0.02	**17.70**	0.01	0.90	19.30	19.30	18.90	14.60	0.02	14.10	0.02	0.50	14.30	14.50	**14.00**
	AVG	25.81	0.02	**25.04**	0.02	1.95	25.81	25.64	25.40	28.39	0.02	**28.16**	0.02	1.37	28.73	28.53	28.55
	Armenian (Eastern)	5.44	0.00	**5.15**	0.00	0.29	5.28	5.27	**5.13**	7.55	0.00	7.08	0.01	0.47	**6.87**	6.89	6.92
	Bulgarian	12.42	0.02	10.63	0.01	1.79	10.84	**9.70**	9.83	20.02	0.01	19.61	0.03	0.41	19.56	20.38	**18.89**
	Dutch	12.76	0.00	12.94	0.01	0.18	12.45	**12.25**	12.47	16.91	0.01	17.32	0.01	0.41	17.17	**16.83**	17.04
	French	8.84	0.00	8.57	0.00	0.27	**8.25**	8.30	8.06	9.33	0.01	9.08	0.01	0.25	8.99	8.98	**8.83**
	Geogian	0.02	0.00	**0.00**	0.00	0.02	**0.00**	**0.00**	**0.00**	0.01	0.00	**0.00**	0.00	0.01	**0.00**	**0.00**	0.03
medium	Serbo-Croatian (Latin)	39.03	0.01	39.06	0.01	0.03	38.90	38.64	**38.06**	39.15	0.02	39.12	0.01	0.03	**38.16**	38.32	39.00
	Hungarian	1.79	0.00	**1.39**	0.00	0.40	1.48	1.46	1.54	2.40	0.00	1.67	0.00	0.73	**1.58**	1.65	1.63
	Japanese (Hiragana)	6.90	0.00	6.94	0.00	0.04	6.93	**6.73**	6.81	6.32	0.00	6.44	0.00	0.12	6.40	6.24	**6.11**
	Korean	20.72	0.01	19.92	0.01	0.80	**19.51**	19.90	19.99	19.19	0.01	**18.18**	0.01	1.01	18.36	18.41	18.62
	Vietnamese (Hanoi)	1.42	0.00	1.35	0.00	0.07	1.32	1.33	**1.26**	2.38	0.00	2.25	0.00	0.13	**2.18**	2.19	2.31
	AVG	10.93	0.01	10.60	0.00	0.39	10.50	10.36	**10.32**	12.33	0.01	12.08	0.01	0.36	**11.93**	11.99	11.94

Table A.4: Results for the SIGMORPHON2021-G2P dataset comparing the original DyNet implementation (*DyNet*) with the PyTorch reimplementation (*PyTorch*). Results represent an average of 10 runs and are produced using beam search decoding with a width of four. *SD* means standard deviation. $|\Delta|$ measures the absolute WER difference between *DyNet* and *PyTorch*. *dp* means dropout probability.

A.2 Experiments with Transformer Encoders

A.2.1 SIGMORPHON2021-G2P

	dev						**test**					
Language	16	32	64	128	256	512	16	32	64	128	256	512
Armenian (Eastern)	5.4	5.2	**5.1**	5.6	5.6	5.8	8	7.1	**6.7**	7.3	8	8.3
Bulgarian	18	16	13	11.8	**10.7**	12.2	29.5	26	**24.3**	27.9	24.2	25.6
Serbo-Croatian (Latin)	49.6	43.2	**38.1**	40.5	38.9	42.3	50.9	43.8	41.1	41	**40.1**	43.7

Table A.5: WER results for the SIGMOPRHON2021-G2P dataset for Armenian, Bulgarian and Serbo-Croatian. Results are based on a single model (fixed seed) and show results for different batch sizes. All models are optimized using Adam and the warmup scheduler.

	dev			**test**		
Language	Adadelta	Adam (plateau)	Adam (warmup)	Adadelta	Adam (plateau)	Adam (warmup)
Armenian (Eastern)	7.10	**5**	5.6	8.30	**6.50**	7.30
Bulgarian	16.00	**11.8**	**11.8**	31.20	**23.9**	27.9
Serbo-Croatian (Latin)	45.70	41.9	**40.5**	45.30	41.4	**41**

Table A.6: WER results for the SIGMOPRHON2021-G2P dataset for Armenian, Bulgarian and Serbo-Croatian. Results are based on a single model (fixed seed) and show results for different optimizers and learning rate schedulers. All models use a batch size of 128.

		dev						**test**					
		4 heads, 4 layers			8 heads, 2 layers			4 heads, 4 layers			8 heads, 2 layers		
	Language	dp = 0.1	dp = 0.3	SD	dp = 0.1	dp = 0.3	SD	dp = 0.1	dp = 0.3	SD	dp = 0.1	dp = 0.3	SD
	Adyghe	**29.67**	30.00	1.00	30.67	30.00	1.00	35.67	**30.00**	2.65	33.33	32.33	0.58
	Greek	14.67	11.67	1.53	**11.33**	13.00	2.65	34.67	**30.33**	3.79	33.67	35.33	3.21
	Icelandic	24.33	**24.00**	1.00	25.00	27.33	1.15	30.33	30.67	3.06	**29.67**	36.00	4.36
	Italian	38.00	34.67	2.89	**33.67**	34.67	1.15	**42.00**	43.33	0.58	43.67	45.67	2.08
	Khmer	45.33	**43.67**	0.58	45.33	44.00	1.00	53.00	**46.00**	3.61	47.00	47.67	0.58
low	Latvian	48.00	**46.33**	0.58	49.00	49.67	1.15	59.00	**55.67**	5.86	58.67	**55.67**	0.58
	Maltese (Latin)	**23.67**	24.67	0.58	26.33	28.00	1.00	32.33	**29.67**	2.08	35.33	34.67	1.15
	Romanian	13.33	**12.67**	0.58	**12.67**	13.67	0.58	21.33	20.33	3.06	16.33	**18.00**	2.65
	Slovenian	**52.67**	53.67	2.08	54.33	53.33	1.53	64.67	**59.33**	4.16	63.67	65.33	3.21
	Welsh (Southwest)	**24.33**	24.67	1.53	26.00	27.67	3.21	29.33	**26.00**	1.00	31.00	27.00	2.65
	AVG	31.40	**30.60**	1.23	31.43	32.13	1.44	40.23	**37.13**	2.98	39.23	39.77	2.10
	Armenian (Eastern)	5.57	5.33	0.15	**5.27**	5.83	0.15	8.17	**7.17**	0.40	7.70	7.30	0.26
	Bulgarian	**12.87**	13.23	1.21	13.17	13.27	0.49	**23.60**	24.23	1.19	25.87	25.03	1.80
	Dutch	13.00	**11.80**	0.36	12.33	13.73	0.96	**18.57**	17.87	1.01	18.07	19.73	0.76
	French	9.17	**8.40**	0.46	8.60	9.27	0.12	9.90	**8.13**	0.67	8.77	9.53	0.76
	Geogian	**0.00**	**0.00**	0.00	**0.00**	**0.00**	0.00	0.07	0.23	0.06	**0.03**	0.10	0.17
medium	Serbo-Croatian (Latin)	39.43	**39.23**	1.07	41.13	45.70	1.93	41.87	42.90	0.20	**41.83**	47.77	1.80
	Hungarian	**1.70**	1.83	0.25	1.60	2.03	0.06	2.33	2.10	0.20	**1.93**	2.43	0.15
	Japanese (Hiragana)	8.67	17.20	14.98	15.57	**8.13**	0.68	8.90	17.97	16.22	14.87	**7.97**	0.47
	Korean	19.47	**18.63**	0.49	19.23	**19.20**	0.36	19.00	**18.30**	1.18	19.63	19.00	0.30
	Vietnamese (Hanoi)	44.03	52.93	17.92	**38.90**	39.87	34.74	**44.37**	53.27	19.12	39.83	42.77	31.89
	AVG	**15.39**	16.86	3.69	15.58	15.70	3.95	**17.68**	19.22	4.03	17.85	18.16	3.84

Table A.7: WER results for the SIGMORPHON2021-G2P dataset for two different transformer configurations. All results represent an average of three models. All models use a batch size of 32 and are optimized using Adam and the warmup scheduler. *dp* means dropout probability. *SD* means standard deviation and is related to the results with a dropout probability of 0.3.

		dev			test		
	Language	dp = 0.1	dp = 0.3	SD	dp = 0.1	dp = 0.3	SD
low	Adyghe	31.20	**30.60**	1.14	34.80	**33.60**	3.13
	Greek	**18.20**	20.20	1.30	**36.80**	38.60	0.55
	Icelandic	33.20	**33.00**	2.24	**35.20**	35.40	2.19
	Italian	**47.60**	50.60	0.89	**48.40**	49.20	2.17
	Khmer	**47.80**	48.40	2.07	53.00	**49.40**	1.52
	Latvian	51.60	**51.40**	2.51	**57.80**	58.20	2.28
	Maltese (Latin)	29.80	**29.00**	1.58	38.20	**37.20**	3.03
	Romanian	21.00	**20.40**	1.52	28.80	**24.40**	2.07
	Slovenian	**59.20**	59.40	1.82	67.00	**64.00**	5.29
	Welsh (Southwest)	**31.80**	33.20	2.59	34.00	**32.00**	1.87
	AVG	**37.14**	37.62	1.77	43.40	**42.20**	2.41
medium	Armenian (Eastern)	**5.30**	5.92	0.44	**6.86**	7.68	0.58
	Bulgarian	**15.98**	17.18	1.13	**27.46**	28.88	2.23
	Dutch	**13.24**	14.80	0.58	**18.46**	20.58	0.73
	French	**8.82**	10.12	0.28	**8.64**	9.74	0.33
	Geogian	**0.00**	**0.00**	0.00	**0.00**	**0.00**	0.00
	Serbo-Croatian (Latin)	**47.10**	50.76	1.07	**47.40**	52.18	1.32
	Hungarian	**1.74**	2.34	0.09	**2.02**	2.44	0.60
	Japanese (Hiragana)	**7.58**	8.08	0.97	**7.94**	**7.94**	0.56
	Korean	18.98	**18.88**	0.53	**18.46**	18.58	0.59
	Vietnamese (Hanoi)	**60.52**	66.78	24.59	**60.18**	66.76	23.81
	AVG	**17.93**	19.49	2.97	**19.74**	21.48	3.07

Table A.8: WER results for the SIGMORPHON2021-G2P dataset for models with a transformer-based encoder. All results represent an average of three models. *dp* means dropout probability. *SD* means standard deviation and is related to the results with a dropout probability of 0.3. Models are optimized using Adam and the plateau scheduler. All models use a batch size of 128.

	dev						test					
	warmup			plateau			warmup			plateau		
Language	dp = 0.1	dp = 0.3	SD	dp = 0.1	dp = 0.3	SD	dp = 0.1	dp = 0.3	SD	dp = 0.1	dp = 0.3	SD
Armenian (Eastern)	**5.28**	5.76	0.21	5.30	5.92	0.44	7.72	7.86	0.58	**6.86**	7.68	0.58
Bulgarian	**14.68**	15.24	1.11	15.98	17.18	1.13	**26.52**	26.62	2.38	27.46	28.88	2.23
Dutch	13.62	13.96	2.84	**13.24**	14.80	0.58	**18.40**	20.42	4.34	18.46	20.58	0.73
French	**8.56**	8.64	0.18	8.82	10.12	0.28	8.92	8.92	0.54	**8.64**	9.74	0.33
Geogian	**0.00**	**0.00**	0.00	**0.00**	**0.00**	0.00	0.06	0.12	0.13	**0.00**	**0.00**	0.00
Serbo-Croatian (Latin)	**39.52**	41.70	0.89	47.10	50.76	1.07	**42.14**	43.80	0.65	47.40	52.18	1.32
Hungarian	**1.60**	2.08	0.19	1.74	2.34	0.09	2.06	2.38	0.33	**2.02**	2.44	0.60
Japanese (Hiragana)	7.60	7.68	0.41	**7.58**	8.08	0.97	**7.36**	7.86	0.29	7.94	7.94	0.56
Korean	19.32	19.38	0.40	18.98	**18.88**	0.53	20.20	19.04	0.80	**18.46**	18.58	0.59
Vietnamese (Hanoi)	**43.34**	51.56	29.68	60.52	66.78	24.59	**43.48**	51.46	28.05	60.18	66.76	23.81
AVG	**15.35**	16.60	3.59	17.93	19.49	2.97	**17.69**	18.85	3.81	19.74	21.48	3.07

Table A.9: WER results for the medium setting of the SIGMORPHON2021-G2P dataset for models with a transformer-based encoder. All results represent an average of 3 models. *dp* means dropout probability. *SD* means standard deviation and is related to the results with a dropout probability of 0.3. Models are optimized using Adam and use the plateau scheduler (*plateau*) or the warmup scheduler (*warmup*). All models use a batch size of 64.

Encoder parameters				dev			test		
layers	heads	dimension feedforward	dropout	arm_e	bul	hbs_latn	arm_e	bul	hbs_latn
2	2	256	0.1	6.50	17.90	47.70	8.10	23.00	49.90
2	2	512	0.1	5.80	16.90	47.50	7.40	25.00	46.90
2	2	1024	0.1	6.60	18.30	45.70	7.70	27.10	48.40
2	2	256	0.3	6.60	18.30	50.30	7.90	28.30	51.20
2	2	512	0.3	5.50	20.90	51.80	7.90	28.20	53.00
2	2	1024	0.3	6.50	20.50	50.50	7.30	25.60	50.80
2	4	256	0.1	7.20	18.90	44.60	9.80	27.80	47.50
2	4	512	0.1	6.40	18.60	45.50	8.10	20.30	46.20
2	4	1024	0.1	6.70	15.50	44.30	8.30	27.70	45.40
2	4	256	0.3	7.20	17.70	49.10	8.40	27.00	51.20
2	4	512	0.3	6.80	21.50	49.40	8.10	25.30	50.60
2	4	1024	0.3	6.80	20.50	47.70	8.20	29.10	49.90
2	8	256	0.1	6.90	17.60	44.10	9.40	25.20	46.10
2	8	512	0.1	6.70	17.00	44.40	8.50	25.80	45.30
2	8	1024	0.1	7.00	19.70	45.70	8.30	28.20	47.00
2	8	256	0.3	8.30	21.80	50.10	10.10	30.50	51.40
2	8	512	0.3	8.20	19.10	48.40	9.80	25.90	51.70
2	8	1024	0.3	7.90	21.60	48.30	8.00	31.70	47.60
4	2	256	0.1	5.80	16.60	46.80	7.30	24.80	49.40
4	2	512	0.1	6.50	15.50	46.70	7.70	21.60	47.10
4	2	1024	0.1	6.50	14.60	47.40	7.70	22.40	48.10
4	2	256	0.3	6.70	24.20	50.10	8.20	31.60	51.40
4	2	512	0.3	6.00	21.40	49.00	7.90	30.40	51.30
4	2	1024	0.3	6.20	16.10	51.30	7.90	27.30	55.00
4	4	256	0.1	6.90	16.40	45.70	7.70	26.80	46.70
4	4	512	0.1	6.50	15.80	44.50	8.40	22.80	47.30
4	4	1024	0.1	7.10	16.00	45.70	8.30	31.20	45.30
4	4	256	0.3	7.90	23.90	48.90	8.80	32.30	50.70
4	4	512	0.3	7.10	26.20	50.00	8.90	33.80	50.20
4	4	1024	0.3	6.70	22.30	48.50	7.40	31.10	51.20
4	8	256	0.1	7.10	16.00	45.70	8.30	31.20	45.30
4	8	512	0.1	7.80	17.20	44.30	8.40	27.20	47.30
4	8	1024	0.1	7.40	19.20	45.00	9.40	30.10	45.50
4	8	256	0.3	8.60	20.60	50.60	9.80	29.80	50.00
4	8	512	0.3	8.30	22.50	48.00	10.70	33.70	49.40
4	8	1024	0.3	9.40	20.80	48.40	11.20	30.50	50.60
8	2	256	0.1	6.50	19.80	45.30	8.50	28.10	43.80
8	2	512	0.1	5.80	19.00	45.40	8.40	29.30	46.10
8	2	1024	0.1	6.10	16.80	45.90	8.20	25.60	46.70
8	2	256	0.3	7.10	20.70	49.40	7.90	31.40	50.40
8	2	512	0.3	6.20	24.20	49.80	8.00	29.80	49.60
8	2	1024	0.3	6.60	25.40	51.50	8.10	34.50	51.40
8	4	256	0.1	6.10	16.40	44.50	8.00	26.10	46.90
8	4	512	0.1	7.80	17.20	45.10	9.20	28.20	45.90
8	4	1024	0.1	6.80	16.60	46.20	8.70	26.10	48.80
8	4	256	0.3	8.50	21.80	49.40	8.70	33.90	49.80
8	4	512	0.3	7.50	19.70	49.90	9.50	25.70	51.30
8	4	1024	0.3	7.40	23.30	49.50	8.90	34.20	52.80
8	8	256	0.1	7.10	13.10	47.20	8.30	27.40	45.60
8	8	512	0.1	8.30	17.40	46.00	9.00	28.10	47.70
8	8	1024	0.1	7.90	21.00	47.60	9.20	31.00	49.50
8	8	256	0.3	9.40	22.40	48.80	11.60	30.40	51.20
8	8	512	0.3	9.00	22.10	48.50	10.60	28.50	50.20
8	8	1024	0.3	9.30	27.10	49.90	9.90	31.10	49.70

Table A.10: WER results for the SIGMOPRHON2021-G2P dataset for Armenian (*arm_e*), Bulgarian (*bul*) and Serbo-Croatian (*hbs_latn*). The results are based on a single model (fixed seed) and show different transformer encoder configurations. All models are optimized using Adadelta.