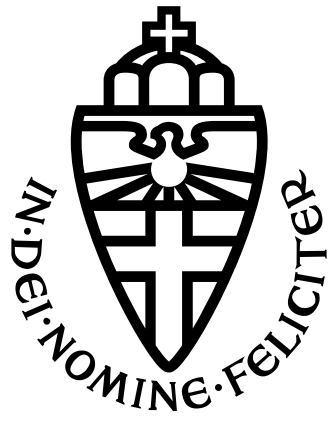
Master Thesis Data Science



Radboud University Nijmegen

On the Influence of Tokenizers in NLP Pipelines

Author:
Hendrik Werner

Supervisor:
Arjen P. De Vries

August, 2022

Abstract

Tokenizers convert between heterogeneous natural text and homogeneous sequences of tokens. They are essential components in the pipelines of established NLP models, yet are typically developed as independent artifacts. This is not ideal, because different pipelines components have to be optimized separately. Great success has been archieved with end-to-end learning, where all components are differentiable, and are optimized in unison.

Token-free models have been experimented with quite early in the history of ML-based NLP, but were quickly abandoned, due to their drawbacks at the time. Recently though, they have had a resurgence, following advancements in model architecture, training techniques, and hardware. It has been shown that token-free models can rival and exceed the performance of token-based models, while offering increased flexibility and robustness, and decreased complexity.

This thesis examines the influence of tokenization in NLP pipelines, by analyzing, reproducing, and quantifying claims from the token-free NLP literature, using the example of NER. We especially focus on gaps in previous work, by compiling and evaluating different representation options, developing algorithms for extracting mentions from those representations, and quantifying and documenting the sequence packing process. Finally, we argue that commonly-used NLP datasets are not necessarily representative of natural language, and suggest a better alternative. All algorithms, code, and results are published.

We conclude that (a) token-free models, like ByT5, offer significant advantages over their tokenizer-based alternatives, like mT5, (b) huge vocabulary sizes (e.g. mT5) become a liability, (c) the Groningen Meaning Bank is a great English-only dataset for NER, (d) sequence packing is extraordinarity effective at improving efficiency, and (e) there are two prime candidates for representing NER problems in a text-to-text framework. We dub these entity list and entity delimiters, respectively. The entity list representation empirically achieves better test scores, but the entity delimiters option is more flexible, and includes more information. mT5-small does not have enough generative capacity to handle the entity delimiters representation.

Contents

1 Introduction

1.1 Natural Language

Natural language can be thought of as being organized in nested hierarchies. At the lowest level, western languages consist of letters forming words, which are composed into sentences. Sentences are further grouped into paragraphs, sections, chapters, etc., which are finally assembled into a text, such as a book.

There are also languages based on different concepts than letters and words, such as Chinese languages like Mandarin. It seems like most of these can be transliterated into a system of words consisting of letters [8, 37, 18], so the same Natural Language Processing (NLP) techniques can be applied.

Words are the smallest constituents carrying meaning in such a system. Consequently, they are often used as the basis for NLP tokenizers. While this is a logical choice, we will discuss later that there are other possibilities which offer advantages over word(piece)-based tokenization.

Tokenization, in the context of NLP, is the process of breaking up highly irregular natural language text into a set of homogeneous constituents called tokens (see section 1.2 and section 2.2).

1.2 Tokenization

Representing text as sequences organized in nested hierarchies (see section 1.1) is not ideal for Machine Learning for several reasons, chief of which their heterogeneity. Words have varying numbers of letters, sentences consist of varying numbers of words, etc. Moreover, the nesting depth and structure can be nearly arbitrary.

Almost all contemporary state-of-the-art NLP models expect their input to be both (a) flat, and (b) homogeneous. Therefore, tokenization is employed to convert natural language to a flat sequence of regular, fixed-size tokens, which are much easier to deal with. Most NLP models in use today require a tokenizer as the first step in their pipeline. [42, 33, 23, 16, 19, 9, 7]

Traditionally, tokenizers were modeled after natural language, by splitting at word boundaries. As discussed in section 1.1, this is a sensible choice, as words are the smallest natural language elements carrying meaning. On the other hand, there are several drawbacks. For example, it

is often not straightforward to determine where word boundaries are, especially in languages without (mandatory) spaces between words. As such, traditional tokenizers require domain expertise, but are still complex, language specific, and error prone. [33]

Tokenizers can be trained unsupervised, and many simple yet powerful tokenization algorithms exist. However, they still possess a number of disadvantages. Namely, they split text into a predetermined finite set of tokens, called their vocabulary, which means they are language specific and can produce Out-Of-Vocabulary (OOV) tokens when encountering novel character sequences. This problem is exacerbated when mixing different alphabets, like many Asian languages do [33].

Tokens are a categorical representation, which is not well-suited for gradient-based Deep Learning (DL). Basically, each token is assigned an arbitrary ID neither capturing syntax nor semantics, so those dimensions are absent from the token space. The simple fact that two tokens might have similar IDs tells you nothing about their meaning or usage in a sentence. Each token has to be treated separately from all others, thwarting generalization and decreasing training efficiency. [2]

> When modeling continuous variables, we obtain generalization more easily [...], because the function to be learned can be expected to have some local smoothness properties. For discrete spaces, the generalization structure is not as obvious: any change of these discrete variables may have a drastic impact on the value of the function to be estimated, and when the number of values that each discrete variable can take is large, most observed objects are almost maximally far from each other in Hamming distance. (Bengio et al., "A Neural Probabilistic Language Model" [2])

Finally, for a lot of traditional tokenizers, encoding is a non-deterministic and ambiguous process, such that the same character sequence can be tokenized in different ways. This is not ideal, because components further down the pipeline have to deal with this ambiguity, complicating and slowing down their training. NLP itself is already a difficult problem, so a non-ambiguous representation is preferable. Otherwise, models will need to spend some of their capacity on disambiguation. They might work well on one encoding of a text, but fail on an equally valid alternate encoding.

See section 1.3 and section 2.2 for more information and ways of addressing some drawbacks of classic tokenization approaches.

1.3 Embeddings

Embeddings are a way of refining tokens to address many of their shortcomings. Instead of assigning arbitrary IDs, the basic idea is to capture both syntactic as well as semantic information, typically by turning tokens into vectors, based on their co-occurence. Words with similar meanings and words used in similar contexts end up close[1] in vector space [27].

This property is useful, because the model has to perform less abstraction itself. It does not need to figure out that "big" and "large" are close semantically; despite their disparate English

[1] by some metric, for example cosine similarity

spellings, and their token representations being arbitrary and independent (i.e. categorical). By the nature of embeddings, both are probably assigned similar vector representations.

Capturing syntactic and semantic information leads to models operating on embeddings generalizing better and being easier to train. Tokens must be treated categorically, so when encountering an unseen token, models do not know how to handle it at all. Unseen embeddings, on the other hand, can be treated based on how similar embeddings are treated, because embeddings for similar tokens are close in vector space. Additionally, learning about an embedding will teach the model about nearby vectors as well, thus increasing training efficiency. [2]

In early Machine Learning (ML) models for NLP, each token was assigned one static embedding. For the reasons outlined above, this is already a huge improvement over tokens. However, this method has trouble dealing with homonymy and polysemy, which require context to disambiguate words and word senses. Thus, contextual embeddings were introduced to choose embeddings during encoding, based on context. Choosing from a fixed set of embeddings for each token allows for the disambiguation of homonyms and polysemes [21].

However, even with word sense disambiguation, there is still context missing from contextual embeddings, because there is a fixed set of embeddings for each token to choose from. For example, "this" refers to some other part of the text. Cross-references like this are not captured by contextual embeddings. Language Model (LM) embeddings tackle this problem, by dynamically creating embeddings for each token on-the-fly, based on the internal representation of some LM. There is no predetermined set of embeddings for each token. Instead, they are created as needed, meaning that much richer information can be captured [30].

1.4 Transfer Learning

For a long time, ML models have been trained from scratch for each new task. This was partly necessitated by the bespoke, task-specific architectures, and partly by a lack of suitably general datasets and training techniques. This mode of operation is inefficient, because abstractions have to be relearned over and over again, and because only information from the downstream task's annotated dataset is used.

For example, the first convolution layers of image recognition networks almost always converge to edge detectors. Analogously, the general meaning of words does not change significantly between texts, so a lot of the same syntactic and semantic information has to be rediscovered when training new embeddings. Also, the quality of embeddings typically improves when training on larger datasets. Generating annotated, task-specific datasets is labor-intensive and costly, which is why they are much smaller than the huge corpora of unstructed, unannotated datasets used for unsupervised learning [20, 34, 35, 31].

With the advent of generic model architectures, which can be reused across different downstream tasks with minimal adaptations, it became possible to apply the technique of Transfer Learning (TL), which aims to transfer knowledge learned from one task to another, related task.

As discussed in section 2.1, TL has been around for quite a while, but could only recently be widely applied, due to the rise of generic model architectures and the availability of huge, generic datasets, combined with unsupervised (pre-)training tasks.

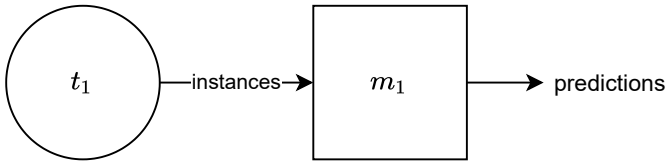
Transfer Learning comes with several advantages, including

- increased data efficiency, as generic information can be shared across datasets and tasks, thus making small datasets viable, and still benefitting large datasets with the additional knowledge,
- improved training efficiency, because models start with generic knowledge, instead of from scratch, which can be used as a stepping stone to start learning the downstream tasks immediately, without needing to rediscover the transferable basics,
- decreased training time, due to increased data and training efficiency, combined with being able to share the generic task-independent knowledge, such that only downstream-specific training has to be repeated for each task, and
- better model performance, resulting from all of the advantages mentioned above.

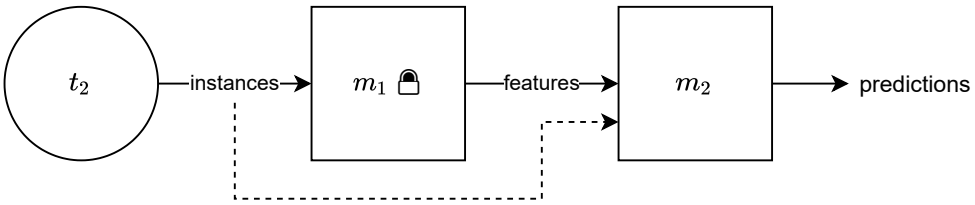
There are two main flavors of TL, called feature extraction and fine-tuning [16].

For the feature extraction approach (see Figure 1.1), a model m_1 is trained on task t_1, then its weights are frozen. In the next step, the output of some (internal) layer of m_1 is used as a feature to train another model m_2 on another task t_2, which is related to the first task t_1.

In order for model m_1 to improve on task t_1, it learns to extract information from its inputs in the early layers, so it can use that information to generate predictions for t_1 in the later layers. Since task t_2 is related to task t_1, there is a good chance that information extracted to solve t_1 will also aid in solving t_2, thus making for good features for m_2.



(a) First Task



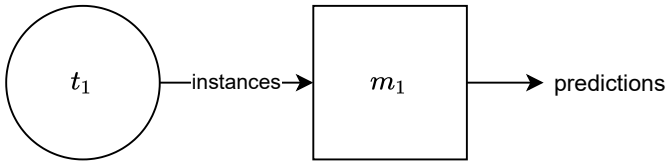
(b) Second Task

Figure 1.1: Transfer Learning – Feature Extraction Approach

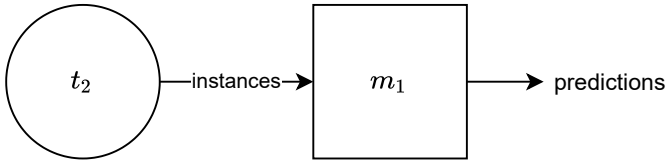
Likewise, fine-tuning (see Figure 1.2) starts by training model m_1 on task t_1, but then the methods diverge. Contrary to the feature extraction approach, m_1's weights are not frozen. Instead, model m_1 is "fine-tuned" on the other task t_2, by optimizing end-to-end for t_2, starting with the weights learned from t_1. There is no second model m_2 using the features extracted by m_1 to solve t_2, but rather m_1 itself is reused with no or only minor modification.

For fine-tuning, the first step is the same as for feature extraction, so model m_1 becomes better at task t_1, by learning to extract useful information from the features it is given. As discussed

earlier, large parts of what the model has learned are applicable to related task t_2 as well, which gives m_1 a running start when training on t_2.



(a) First Task



(b) Second Task

Figure 1.2: Transfer Learning – Fine-Tuning Approach

1.5 Foundation Models

Building on the Transfer Learning concept (section 1.4), foundation models have become popular and prevalent in recent years [16, 31, 41, 40, 28]. Resulting from their effectiveness, they are applied with huge success to a variety of different tasks across varying domains. Workings and (dis-)advantages of foundation models are explained in this section, and contrasted with the classical approach to model architecture and training.

Figures 1.3 and 1.4 juxtapose the classical and foundation model approaches to model training, respectively. Squares (□) indicate Neural Network (NN) models, circles (◯) represent datasets, and rhombi (◊) stand for training steps.

Relative costs of pipeline components are shown by the number of dollar symbols. Steps annotated with $ are relatively cheap, those with $$ are relatively expensive, and $$$ steps are extremely costly. As all measures of cost are highly correlated, it almost does not matter which one is chosen. The idea is just to give an intuition for how different training steps compare.

In the classical approach to model training, outlined in Figure 1.3, each task gets its own bespoke model architecture, and each task-specific model is trained from first principles on a labeled dataset matching that task. As discussed in section 1.4, this mode of operation is vastly inefficient.

Achieving good performance when training from scratch with contemporary DL model architectures and training methods requires massive datasets [32], and is consequently prohibitively expensive. Compounding with the restrictive costs of gathering and labeling these datasets for each task, this constituted a primary bottleneck for ML and Artificial Intelligence (AI) progression.

Training state-of-the-art DL models is extremely costly across almost every measurable axis, including time, energy, money, and environmental impact. [35] Therefore, it is hugely beneficial to share as much training as possible between different models, which is how foundation models

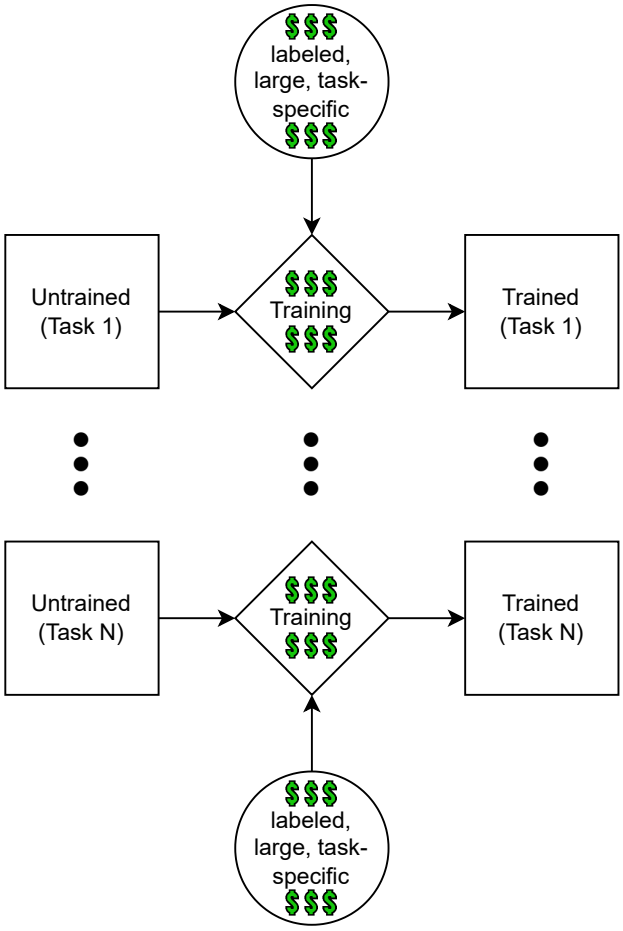


Figure 1.3: Classical Model Training

were conceived.

Instead of having to assemble a large, labeled, task-specific dataset for each new task – an enormously costly undertaking in itself – and repeatedly training models from scratch, foundation models share most of the expensive part of the training process across tasks. This is accomplished through two main ideas: 1. unified model architectures, and 2. splitting training into two parts called pre-training and fine-tuning.

Figure 1.4 depicts the training process of foundation models, which apply a unified model architecture to several different tasks. This allows for the training to be split into two parts, thereby amortizing the costliest steps of generating these models.

Pre-training is responsible for imparting generic knowledge to the model, which is generally useful in some domain, resulting in a so-called "foundation" that can be used to cheaply train specialized models on downstream tasks.

Originally, models were typically pre-trained on labeled datasets. In fields like Computer Vision (CV), where models are often pre-trained on datasets like ImageNet [12], this approach is still common. However, it has empirically been shown that the size of the training set critically influences model performance. Simply training on larger datasets commonly improves performance. [32, 20, 34, 26, 36]

Composing unlabeled data is relatively straightforward, while labeling is complex and work-intensive. Accordingly, the largest available datasets are unlabeled. As a result, researchers

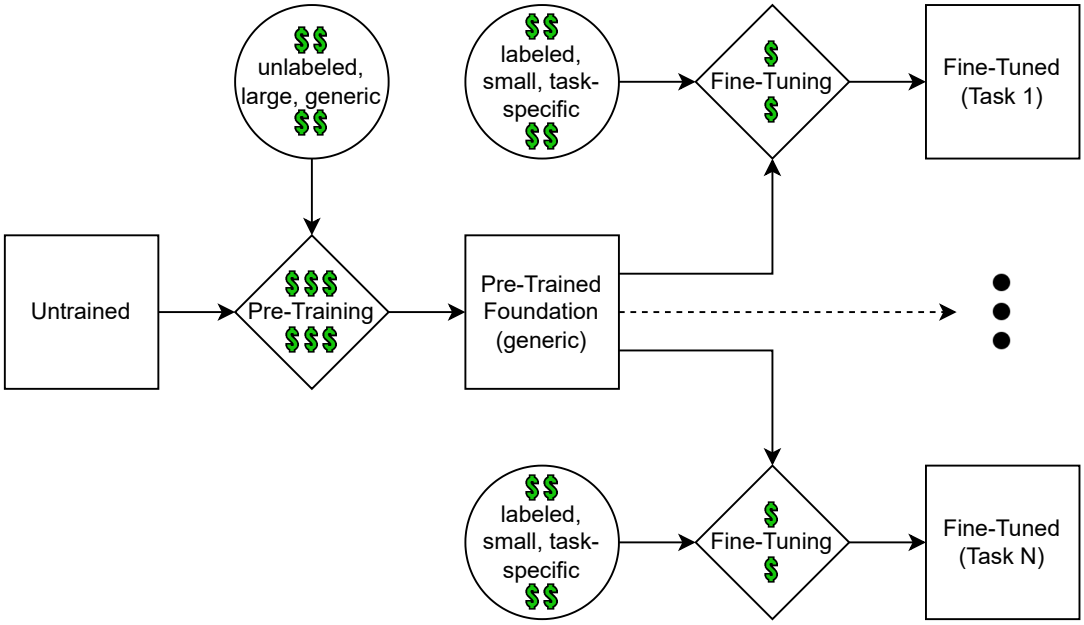


Figure 1.4: Foundation Model Training

have more recently introduced weakly supervised and unsupervised pre-training tasks, which can leverage these colossal, unlabeled, generic datasets. [26]

It is cheaper to compile these datasets compared to the ones required for the classical approach, since (a) they are generic, and must only be related to the domain, not any specific downstream task, and (b) datapoints need not be labeled, which is typically the most expensive aspect of dataset generation by far, because it requires domain experts.

Fine-tuning a foundation model necessitates both smaller datasets and fewer training steps compared to starting from scratch, as in the classical approach. Despite their reduced size, it remains somewhat expensive to assemble the task-specific datasets, as they still have to be labeled by domain experts. Training itself, however, becomes comparatively highly economical.

Many contemporary state-of-the-art NLP ML models employ TL through foundation models, including the original Transformer [38], BERT and its derivates [15, 16, 23, 9, 24, 10, 19], GPT variants [7], as well as T5 and its derivatives [31, 41, 40]. This paradigm has also successfuly been applied in other domains, such as CV.

1.6 Goals and Contributions

This thesis is intended primarily as an investigation of the influence of tokenization in NLP pipelines. We aim to analyze, reproduce, and quantify claims from the token-free NLP literature, with a focus on Named Entity Recognition (NER), by critically examining the mT5 and ByT5 papers. We try to draw generalizable conclusions, by performing our experiments on actually natural language, comprised of full texts. Lastly, we advance the research on NER in the context of a text-to-text framework, which is underanalyzed and insufficiently documented in prior work.

To achieve these goals, we make the following contributions:

- We compile and analyze different representation options for NER tasks, which are left underspecified, underanalyzed, and sparsely — if at all — documented in the existing literature.
- We develop and document mention extraction algorithms for these representation options.

  This aspect is completely neglected in the existing literature, despite being essential for reproducibility. Slight changes in choice and implementation of extraction algorithms can affect measured metrics. Without this information, it is unclear if and how comparable metrics reported in different publications are.
- We detail the citeria for and choice of dataset, and we extensively analyze the chosen dataset statistically.

  Prevalent datasets are not necessarily a good representation of natural language, or not well-suited towards our use case, so we suggest a better option.
- We specify and algorithm for sequence packing, and quantify its results.

  Sequence packing is a common and essential technique for improving efficiency by reducing padding, that is surprisingly almost completely passed over in the literature.
- We determine and document differences between the mT5 and ByT5 models, as proposed originally, and their implementations used in this thesis.
- We train models for different architecture and representation option combinations, and compare their training process and inference performance.

2 Related Work

Contained in this chapter is a review of the literature relevant to this thesis. It includes background information, for example regarding the lineage of current state-of-the-art architectures and models, as well as recent cutting-edge research.

2.1 Transfer Learning

Transfer Learning is the process of transferring knowledge gained on one task to a different but related task. It has become prevalent in several ML branches, including NLP and CV, due to its numerous advantages, most notably improved efficiency and performance. More information can be found in section 1.4.

Bozinovski and Fulgosi explored the concept of TL in the context of ML quite early on, in their 1976 article "The influence of pattern similarity and transfer learning upon training of a base perceptron B2 (original in Croatian: Utjecaj slicnosti likova i transfera ucenja na obucavanje baznog perceptrona B2)", "The influence of pattern similarity and transfer learning upon training of a base perceptron B2 (original in Croatian: Utjecaj slicnosti likova i transfera ucenja na obucavanje baznog perceptrona B2)". They conducted a mathematical and experimental investigation of TL for NNs, and developed models to describe and measures to quantify it.

Being originally published in Croatian, in a niche publication (*Proceedings of the symposium Informatica*), and being ahead of its time in terms of applicability, the article has largely been ignored in the two decades following its publication. However, in recent decades, and especially over the last few years, there has been renewed interest in TL, for the advantages it offers (see section 1.4). [5]

2.2 Tokenization

Tokenization is the process of slitting up arbitrary heterogeneous text into smaller homogeneous subunits. This representation is conducive to contemporary ML approaches, and therefore often employed as the first step in NLP pipelines. For more information, refer to section 1.2.

In their 2012 article "Japanese and Korean voice search", "Japanese and Korean voice search", Schuster and Nakajima identified and addressed a subset of problems with existing tokenizers at the time. They proposed a data-driven, language-agnostic method of building tokenizers (called segmenters in the paper) named WordPieceModel. Using an unsupervised greedy algorithm,

it can generate an arbitrary number of tokens called "word pieces" from complex, infinite vocabularies with mixed alphabets.

WordPieceModel starts with a vocabulary of the most basic building blocks — Unicode characters — and builds a Language Model from them. New tokens are added iteratively through a greedy algorithm, by combining existing ones, such that the LM likelihood of the training data is maximized. The process is repeated until a predetermined vocabulary size is reached, or the LM likelihood converges.

Following this algorithm, an inverted binary tree of token pairs is build, allowing for exceedingly efficient, deterministic, and thus non-ambiguous tokenization, which does not produce OOV tokens. Encoding has linear time complexity with respect to the length of input sentences.

WordPieceModel has successfully been applied across several languages at large scale at Google. Initially being developed for Japanese, it has been applied without modification to other languages — including Korean — under adverse conditions, such as inconsistent use of spaces between words, lots of alternative spellings, and mixed alphabets.

2.3 Embeddings

As discussed in section 1.2, each token is assigned an arbitrary, categorical token ID, which is not conducive to gradient-descent-based Deep Learning, which dominates the modern ML and NLP landscape. Hence, it is advantageous to embed finite, categorical tokens into a continuous vector space, capturing syntactic and semantic aspects, along with context. Embeddings, as discussed in section 1.3, are a method of representing tokens as vectors, typically vectors of real numbers.

Efficient Estimation of Word Representations in Vector Space, *Efficient Estimation of Word Representations in Vector Space* is the article that introduced the word2vec model architecture, which is able to learn word embeddings from large text corpora, based on co-occurence of words. These embeddings capture both syntactic and semantic information. word2vec is based on the fact that similar words occur in similar contexts. At the time of publication, it achieved state-of-the-art performance on word similarity tasks, while also lowering the computational cost, relative to existing methods.

2.4 Transformer Language Models

In 2017, Recurrent Neural Networks (RNNs), Long Short-Term Memorys (LSTMs), and Gated Recurrent Units (GRUs), had been the established as state-of-the-art architectures for NLP. The best-performing models also included an attention mechanism, that lets the decoder selectively attend to different parts on the input, which are relevant to the current output. These models come with the major drawback of operating sequentially, which presents a major bottleneck during both training and inference [38].

Then, Vaswani et al. showed that *Attention Is All You Need*, *Attention Is All You Need*, by introducing the Transformer language model — a sequence-to-sequence model solely based on

the self-attention mechanism, that works by repeatedly replacing vectors with a weighted sum of other vectors. Self-attention is a special case of attention, where each layer only attends to itself.

Transformer models immediately became widely successful, because of two inherent advantages. Firstly, they reach better inference scores than previous models. Secondly, and arguably most importantly, they are much more parallelizable, making them vastly more efficient to train and use.

Building on the Transformer model, Devlin et al. introduced the Bidirectional Encoder Representations from Transformers (BERT) architecture, which — as its name suggests — is a bidirectional Transformer encoder, leaving out the entire decoder stack, and replacing it with a single, fully-connected output layer. The output layer is task-specific, but the rest of the model is generic, and shared across tasks. A generic, task-independent model architecture enables a two-step training regimen, consisting of pre-training and fine-tuning, which is known today as the foundation model paradigm, discussed in section 1.5.

Foundation models have quickly become extremely prevalent and influential since their introduction, because they allow the amortization of the most expensive training steps. It has become customary to release pre-trained model checkpoints, which can be fine-tuned with much smaller datasets on commodity hardware. They serve as a baseline to build upon, and enable research that would not otherwise have been possible, including this thesis.

BERT does not limit self-attention to only the right or left context, like previous models. Instead, it uses bidirectional context, which necessitates new training tasks. With classical language modeling tasks, the model could just peek at the correct answer. To address this, the authors introduced the unsupervised Masked Language Modeling (MLM) and next sentence prediction tasks.

After conducting an extensive survey of the literature on Transformer models, Raffel et al. presented the Text-To-Text Transfer Transformer (T5) architecture and in their 2020 article *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer*. They also composed and published the Colossal Clean Crawled Corpus (C4) dataset, on which they trained their model.

While previous models were still relatively specialized, the big novelty of T5 is to encode every problem in a unified text-to-text framework, dispensing with the different task-specific model specializations of prior architectures, and even the interchangeable output layers from BERT, to achieve a fully task-independent architecture. Interestingly, T5 returns to the original encoder-decoder structure of the Transformer architecture, as proposed by Vaswani et al.

Inputs and targets for many NLP tasks are naturally represented as text, and thus easily fit into a text-to-text paradigm (see section 3.1), which has the advantage of enabling multi-task learning, where a single model is trained concurrently on multiple tasks. This was utilized during pre-training, by mixing supervised with unsupervised tasks, that are discriminated between through the use of task prefixes (see subsection 3.1.1). Besides, the same loss function and training paradigm can be used across all tasks.

Building on the T5 architecture, Xue et al. developed *mT5: A massively multilingual pre-trained text-to-text transformer*, *mT5: A massively multilingual pre-trained text-to-text transformer*. It has a slightly modified architecture, uses a much bigger vocabulary, and is pre-

trained on the newly introduced multilingual C4 (mC4) dataset, covering 101 languages. Pre-training is unsupervised-only, supplanting multi-task learning. To prevent "accidental translation", arising from monolingual fine-tuning, Domain Preserving Training (DPT) is employed, by mixing in 1% unsupervised pre-training examples during fine-tuning, uniformly sampled across all supported languages.

Token-free models have long disregarded, due to their inherent disadvantages. Single bytes or characters rarely carry meaning on their own. RNNs and LSTMs have trouble retaining information across long sequences. Some architectures, including BERT, have a limit on their input and output size. Many techniques scale unfavorably with the length of the input, including self-attention, which has quadratic time complexity with respect to the sequence length [40]. For all of these reasons, tokenizers have long been optimized to produce large tokens, covering word and sub-word units. Sections 1.2 and 2.2 contain more information on the topic.

Recently, Xue et al. showed that mT5 can be operated token-free, and applied to byte-to-byte problems, with minimal modifications. *ByT5: Towards a token-free future with pre-trained byte-to-byte models*, *ByT5: Towards a token-free future with pre-trained byte-to-byte models* describes this modified mT5 architecture, dubbed ByT5, showing that it can rival and exceed the performance of token-based models, while being more flexible and robust at the same time. It is argued that token-free models are conceptually simpler, and reduce pipeline complexity and technical debt, while being empirically powerful. They also allow more holistic model training, as the tokenizer no longer exists as an independent artifact.

ByT5 utilizes the ubiquitous variable length Unicode UTF-8 encoding, which has the goal of being able to encode any human script in existence as a byte sequence. The version described in the paper has a vocabulary of all 256 possible byte values. UTF-8 bytes are shifted by 3, to make room for the padding `<pad>`, end-of-sentence `</s>`, and unknown `<unk>` tokens. ByT5 can encode any possible byte sequence, so the `<unk>` token is not strictly needed, and is only included by convention. Rather than introducing separate sentinel tokens, the last 100 byte IDs are reused.

The ByT5 implementation used in this thesis deviates slightly from the recipe described here. These deviations are documented in section 3.5.

3 Methods

3.1 Problem Representation

The T5 represents every problem in a text-to-text framework. Some tasks lend themselves easily to this representation. Either their inputs and outputs are already naturally represented as free-form text, like for translation and summarization; or their labels can simply be encoded as text, as for sequence classification. Other tasks, including NER, are not as straightforward to frame as text-to-text problems.

NER was not one of the tasks covered in the original T5 paper [31], and there is a lack of conclusive literature on the subject. There are several candidate solutions with different trade-offs, which we will exemplify using the sentence "London visits London.". It is short and simple, yet demonstrates the difficulty of dealing with homonymy, and the importance of context for named entity classification.

Generally, named entity mentions have three attributes: 1. the mention itself (a substring of the input text), 2. its entity type, and 3. its position in the input text. Depending on what the extracted entities are used for, we might be interested in all these properties, or a subset thereof.

Throughout the remainder of this section, we explore different representation options, their advantaged, disadvantages, and trade-offs. Finally, we select Option 1: Entity List and Option 2: Entity Delimiters as the most promising representation options, which are used to conduct experiments.

3.1.1 Task Prefixes

T5, as originally presented in *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer*, made use of multi-task learning, where a single model is trained on multiple tasks simultaneously, and can perform inference on all of them. This was necessitated by the mix of supervised and unsupervised pre-training tasks. To inform the model about the current task, task prefixes were used [31].

Task prefixes are short, textual descriptions of the tasks, that are prepended to the model inputs. For translation tasks, training input-output pairs for T5 consist of texts in two different languages, like English and German, where the input is prefixed with `"translate English to German: "`; while summarization tasks use pairs of a full texts, prefixed with `"summarize: "`, and their summaries.

mT5 and subsequently ByT5 renounced the supervised pre-training tasks in favor of purely unsupervised pre-training on the span corruption task, obsoleting the need for task-prefixes, if the model is only fine-tuned in a single downstream task [41, 40]. This is the case for this thesis, which only pertains to NER. Accordingly, we did not make use of task prefixes.

3.1.2 Option 1: Entity List

output: `[person] London [location] London`

Advantages	Disadvantages
• trivial to process, as long as no entity positions are required • does not depend on alignment between input and output • similar in form (but not function) to existing T5 tasks [31] • short targets, easy to generate	• difficult/impossible to reconstruct entity positions in the input • can be ambiguous, possibly complicating the calculation of useful gradients • output has different size than input

Target labels can be represented simply as a list of entity types and mentions present in the input. This encoding tells us about the type and mention of named entities, and their order. It does not, however, contain information about the position of named entity mentions in the input.

Main advantages of this option include being similar in shape to existing T5 tasks, not depending on alignment between input and output, and being trivial to process, if all one is interested in are the entity mentions and types. While all T5 variants have considerable generative abilities, shorter targets are nonetheless easier to generate, which is another advantage of the entity list representation.

Getting suggested by T5 author Colin Raffel makes the approach more likely to work well, given that one expects him to have built a reasonable intuition for the model and framing tasks as text-to-text problems; while not depending on input-output alignment increases the model's robustness against errors during output generation.

Depending on the input text, it might be difficult to impossible to reconstruct the positions of entity mentions in the input. As an example, take the following output: `[person] London`. It is not possible to differentiate whether the model has correctly extracted and classified the first entity mention while ignoring the second, or whether it ignored the first mention, but then extracted and misclassified the second one.

For certain use cases, this information may not be needed during inference, but such behavior could adversely affect the calculation of gradients during training. The output looks the same in both cases, but the most useful gradients likely differ significantly. In the first case, the handling of person "London" is perfect, and only the behavior for location "London" should be revised, while the model should adapt both in the second case.

3.1.3 Option 2: Entity Delimiters

output: `[S-person] London [E-person] visits [S-location] London [E-location].`

Advantages	Disadvantages
• more versatile than other options (e.g. overlapping mentions) • all information is included in the output • does not depend on alignment between input and output	• versatility leads to ambiguities (e.g. mismatch between opening and closing delimiters) • long targets, harder to generate • calculating entity location in input is more difficult, compared to option 3

Another option is to reproduce the input text, while putting entity delimiters around entity mentions. Entity delimiters are special tokens demarcating the start (`[S-type]`) and end (`[E-type]`) of entity mentions.

Entity delimiters keep all of the information about entity mentions (mention, location, and position) in one place, thus not depending on alignment between input and output. On the flip side, the insertion of additional tokens results in the targets being longer, requiring models to spend more generative capacity. It also complicates the process of determining original entity positions.

A noteworthy aspect of this approach is that it is possible for entity mentions to overlap. They can either be nested (i.e. fully contained) inside other mentions (Figure 3.1a), or they can partially overlap (Figure 3.1b).

`[S-organization][S-location]London City[E-location] Council[E-organization]`

(a) Fully-Contained

`[S-location]London [S-organization]City[E-location] Council[E-organization]`

(b) Partially-Overlapping

Figure 3.1: Overlapping Entity Mentions

It is unclear how to classify this ability, as, on the one hand, it affords additional possibilities to represent outputs which are not possible with the other techniques. On the other hand though, this additional complexity seems unwarranted, given that typical NER datasets, being encoded with the BIO-encoding, do not contain overlapping mentions.

Two forms of partially-overlapping mentions exist: real and faux. For pairs of real partially-overlapping mentions, each mentions has at least one token not contained in the other mention. Figure 3.1b shows such an example, where "London" is only part of the location mention, while "Council" is only part of the organization mention.

As their name suggests, faux partially-overlapping entity mentions are not actually partially overlapping in terms of tokens. Instead, the overlap is limited to the delimiters themselves. `[S-a] [S-b] t1 [E-a] [E-b]` is an example of faux partial overlap. Both mentions consists of the same single token t_1. Consequently, they do not meet the definition for real partial overlap. Their delimiters, however, make it look like they are partially overlapping, hence the name.

While fully-contained entity mentions are possible and sensible, we are not aware of a situation where having partially-overlapping entity mentions would make sense. The "London City Council" example from Figure 3.1b is contrived, and would not occur like that in natural language texts. Nonetheless, partially-overlapping mentions are supported by both the model and mention extraction algorithm detailed in subsection 3.2.2.

3.1.4 Option 3a: Simple Entity Markers

output: `[person] visits [location].`

Advantages	Disadvantages
• localized, no balancing required (as opposed to entity delimiters) • more concise than aligned entity markers • better suited for text generation, compared to aligned entity markers	• information is split between input and output • brittle, relying on alignment • possibly ambiguous, even when well-formed • clashes with assumptions in ML frameworks

input:	`Lon`	`don`	`vi`	`sits`	`Lon`	`don`	`.`
output:	`[person]`		`vi`	`sits`	`[location]`		`.`

Table 3.1: Example Input-Output Pair – 3a: Simple Entity Markers

For the simple entity markers representation option, each entity mention is replaced with a single entity marker of the correct type. That means that spans of entity mention tokens are replaced with a single marker, as shown in Table 3.1. In the example, `London` is tokenized into tokens `Lon` and `don`, yet the target output only includes one entity marker token per mention.

Normally, this is no problem, and leads to a more concise representation, compared to aligned markers. However, for a sequence of $n_m \geq 2$ consecutive entity mentions, where the number of tokens n_t is larger than than the number of mentions $n_t > n_m$[1], alignment becomes ambiguous.

This ambiguity can best be demonstrated by example. If we take $n_t = 3$ tokens, and $n_m = 2$ mentions, conditions $n_m \geq 2$ and $n_t > n_m$ are fulfilled, which means that the interpretation is dubious. There are two equally valid possibilities. This property makes simple entity markers the only discussed representation option that can be ambiguous, even when the output is well-formed. See Figure 3.2 for a visualization.

In this example, there are only two possibilities. Larger cases with more tokens and/or mentions lead to a combinatorial explosion of possible interpretations. Since consecutive entity mentions

[1]Under the assumption that each mention must consist of at least one token, $n_t \geq n_m$ holds.

input: t1 t2 t3
output: [a] [b]

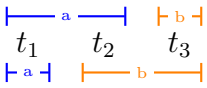


Figure 3.2: Ambiguous Output

are common (see section 4.1), and preliminary training results were not especially promising, it was decided not to pursue this representation option further.

Another disadvantage of this representation is that information is split between input and output, so both are needed to reconstruct entity mentions, which is brittle due to relying on alignment. A single missing or extraneous token can have cascading consequences. Additionally, this clashes with the common assumption in ML frameworks like Tensorflow and PyTorch, that all required information is located in the model's outputs. They are not set up to reconcile information from inputs and outputs, which complicates the implementation of both entity marker representation options.

3.1.5 Option 3b: Aligned Entity Markers

output: `[B-person] [I-person] visits [B-location] [I-location] .`

Advantages	Disadvantages
	• information is split between input and output
• localized, no balancing required	• brittle, relying on alignment
• similar to well-established BIO-encoding	• longer than simple entity markers
• unambiguous	• not a good fit for text-to-text framework
	• clashes with assumptions in ML frameworks

input:	Lon	don	vi	sits	Lon	don	.
output:	[B-person]	[I-person]	vi	sits	[B-location]	[I-location]	.

Table 3.2: Example Input-Output Pair – 3b: Aligned Entity Markers

Aligned entity markers share some of the advantages and disadvantages of the simple entity markers representation from subsection 3.1.4. However, being similar to the often-used BIO-encoding, this option is unambiguous even for malformed outputs, at the cost of being longer.

The main disadvantages of aligned entity markers, as opposed to simple ones, are that targets get longer and contain sequences of repeated tokens. Better generative abilities are required to output longer sequences in general. Also, during text generation, models are penalized for repeating the same token, as that is not something you generally want. Aligned entity markers, however, often require the repetition of continuation tokens (`[I-type]`), and are therefore not well-fitted towards a text-to-text framework.

Ultimately, for these reasons and the ones explained in subsection 3.1.4, we decided not to

further pursue this representation option either.

3.2 Mention Extraction

T5 models produce outputs sequentially, token by token, in a greedy, free-form, generative process, which does not necessarily match the template we want it to. Outputs can have arbitrary length, and are essentially just a sequence of tokens. While we train the model to output certain patterns, adherence to them is not guaranteed. Thus, these models can output arbitrary token sequences, which need to be handled during mention extraction. Just focusing on the happy path is insufficient. Edge-cases need to be accounted for as well.

Consequently, extracting entity mentions from model output is not always straightforward. Lots of edge cases need to be taken into account. Some representation options, like Option 3a: Simple Entity Markers and Option 3b: Aligned Entity Markers, even depend on alignment between input and output.

This section covers algorithms for extracting entity mentions from model outputs, for different representation options. Appendix A contains examples of well-formed and degenerate model outputs, and the corresponding mentions extracted by the algorithms presented here.

I_{start}:	set of start token IDs
$I_{special}$:	set of special token IDs
i_{space}:	space (' ') token ID
I:	sequence of input token IDs
m:	entity mention (sequence of output token IDs)

Table 3.3: Mention Extraction Notation

Throughout this thesis, the symbols defined in Table 3.3 are used. Moreover, the mention finalization function from Listing 3.1 is also referenced repeatedly.

Listing 3.1: Mention Finalization Function

```
function finalize(m):
    if m.length() < 1:
        return

    if m[-1] = i_space:
        m.pop()

    yield m
```

3.2.1 Entity List Extraction

As described in subsection 3.1.2, entity lists are the easiest representation option to extract mentions from, as long as one is not interested in their position in the input text.

Listing 3.2: Mention Extraction Algorithm – Entity List

```
m ← []

∀i ∈ I:
    if i ∈ I_start:
        yield from finalize(m)
        m ← []
    else if i ∈ I_special:
        if m.length() > 0 and m[−1] ≠ i_space:
            m.append(i_space)
    else:
        m.append(i)

yield from finalize(m)
```

The algorithm depicted in Listing 3.2 iterates over all input token IDs $i \in I$.

- When encountering a start token ID $i \in I_{start}$, the current mention m is finalized and output, and a new, empty mention is started.
- For any number of consecutive special token IDs ($i \in I_{special}$), a single space token ID i_{space} is appended to the mention m, unless m is empty.
- All other encountered tokens are appended to mention m.
- Finally, after iterating over all token IDs $i \in I$, the current mention m is finalized and output.

Refer to section A.1 for input/output examples of the entity list mention extraction algorithm.

3.2.2 Entity Delimiters Extraction

Power and versatility usually begets complexity, which is why Option 2: Entity Delimiters (subsection 3.1.3) has the most involved mention extraction algorithm. Entity delimiters can encode sophisticated nesting and partial overlap patters. Moreover, there is more room for malformed token sequences, which do not match the desired template. Possible edge cases – even if rare in practice – need to be accounted for in a robust mention extraction algorithm.

In addition to the general notation from Table 3.3, we introduce supplementary delimiters-specific notation from Table 3.4.

In several places, it is necessary to append the space token ID i_{space} to several mentions $m \in M$. Hence, we introduce the `append_space` abstraction (Listing 3.3) taking care of this.

I_{end}:	set of end token IDs
M:	sequence of entity mentions (m) under construction
s:	sequence of entity types used for constructing mentions

Table 3.4: Supplementary Delimiters Extraction Notation

Listing 3.3: Append Space Function

```
function append_space(M):
    ∀m ∈ M:
        if m.length() > 0 and m[−1] ≠ i_space:
            m.append(i_space)
```

The delimiters mention extraction algorithm makes use of the rightmost index of a value v in a sequence s. Since there is no standard way of getting that index in the implementation language Python, we define helper function `rindex` to find it. Note that the indices are 1-based, matching Python's semantics, where `s[-rindex(s, v)] = v`. Furthermore, we use a vectorized version of the Mention Finalization Function from Listing 3.1.

Listing 3.4: Rightmost Index Function

```
function rindex(s, v):
    for i, x in enumerate(reversed(s)):
        if x = v:
            return i + 1
```

With these prerequisites in place, we can define the delimiters entity mention extraction algorithm itself.

Listing 3.5: Mention Extraction Algorithm – Entity Delimiters

```
s ← []
M ← []

∀i ∈ I:
    if i ∈ I_start:
        append_space(M)
        s.append(i)
        M.append([])
    else if i ∈ I_end:
        latest_start ← rindex(s, i − 1)
        if latest_start ≠ None:
            yield from finalize(M[−latest_start])
            del s[−latest_start]
            del M[−latest_start]
        append_space(M)
    else if i ∈ I_special:
        append_space(M)
```

```
    else:
        ∀m ∈ M:
            m.append(i)
yield from finalize(*M)
```

As shown in Listing 3.5, the algorithm iterates over all input token IDs $i \in I$.

- Upon reading a start token ID $i \in I_{start}$, the space token ID i_{space} is appended to all active mentions $m \in M$. A new empty mention m is started and added to the mentions under construction M. Its entity type, which is determined by i, is appended to s.
- Mention end delimiters $i \in I_{end}$ close the latest corresponding opening delimiter, if it exists. Entity mention m is removed from M, finalized, and output; and its entity type is removed from s. Finally, space token ID i_{space} is added to all remaining active mentions $m \in M$.
- Special token IDs $i \in I_{special}$ are treated as whitespace, by appending i_{space} to all mentions under construction $m \in M$.
- When encountering any other token ID i, it is appended to all currently active mentions $m \in M$.
- After reading the complete input token ID sequence I, the currently active mentions are finalized and output.

Appendix section A.2 contains example input/output pairs for the entity delimiters mention extraction algorithm described above.

3.3 Dataset Choice

The goal of this thesis was to investigate the influence of tokenizers in NLP pipelines, using the example of NER. We did not aim to compare our results to existing models, nor to rival their performance. Indeed, this would probably not have been possible with the hardware and datasets available to us, given that cutting-edge models are typically trained on gigantic datasets, using hundreds of GPUs [31, 7, 41, 40, 13].

Instead, we wanted to dive into the impact of tokenizers in Transformer language models, by reproducing, quantifying, and evaluating claims from prior work. Moreover, we examine different representation options for datasets, and techniques for improving training efficiency. Finally, we propose and implement additional metrics and axes for evaluation of these models.

A common pattern emerged in previous work, where new ideas are first explored in a monolingual (primarily English) context, and then generalized to multilingual setups in follow-up work [15, 24, 10, 41]. We chose to stick to this pattern, and focus on the monolingual case, as to not complicate the experimental setup.

NLP, as its name suggests, is intended to handle natural language. Thus, we found it surprising that many of the pervasive datasets in the NER literature do not actually comprise natural

language. Instead of containing full texts, paragraphs, or at least sentences, they are made up of sentence fragments and loose entity mentions.

It is well-established that best performance is reached by fine-tuning on a dataset as close to the downstream task as possible. Models are evaluated and compared based on their scores on standard benchmark datasets, such as CoNLL-2003 or WikiAnn.

Hence, training on similar data is beneficial in order to claim state-of-the-art performance. Consequently, this is what most authors choose to do. However, results obtained this way are not necessarily representative of the environment those models are indented to be used in. ML models are supposed to generalize to unseen data. Natural language does not look like customary benchmark datasets.

For NER on truly natural language, context can be imperative for finding and disambiguating entity mentions. Groningen Meaning Bank (GMB) provides that context, while many common benchmark datasets do not.

> In the GMB texts are annotated rather than isolated sentences, which provides a means to deal with ambiguities on the sentence level that require discourse context for resolving them. (Bos and Basile, *Groningen Meaning Bank* [3])

We are interested in the real-world performance above benchmarks. Benchmark datasets change every few years. Results obtained on actual natural language are more likely to be generalizable and hold up in practice.

Due to questionable implementation choices, which deviate from the original ByT5 architecture (see section 3.6), it is beneficial for the dataset to consist mainly of characters in the ASCII range. The GMB dataset fits this criterion nicely, whereas WikiAnn and CoNLL-2003 do not.

The established CoNLL-2003 and WikiAnn datasets, which are commonly used for evaluation in the literature, fail some of our requirements, while GMB matches them. For these reasons, we chose to train and evaluate our models on the GMB dataset. Specifically version 2.2.0, which is available in a handy, pre-processed format for NER on Kaggle [39]. We shall refer to this version as the Kaggle NER dataset.

Choosing the GMB-based Kaggle NER dataset is fine, as we explicitly did not aim to achieve state-of-the-art performance. Therefore, we do not need to be able to directly compare our results to those of existing models. It suffices to compare the differences between models and techniques within this work. Care was taken to ensure interal comparability, for example by making the train/test-split independent of the packing-size (see section 3.4), and using representation-independent metrics.

3.4 Sequence Packing

Contemporary tensor accelerators [11, 25] and ML models, including all T5 variants [31, 41, 40], typically require non-ragged input tensors. This means that each batch of input sentences has to be padded to the length of its longest constituent. Considering the distribution of raw and tokenized sentence lengths — depicted in figures 4.1 and 4.6, respectively — this leads to

as much as about 90% of tokens being taken up by padding (see Table 4.2). This reduces the amount of useful information per batch, and consequentially lowers both training and inference efficiency. section 4.2 has more details on the matter.

Padding tokens serve no other purpose than making input batch tensors regularly-shaped. Indeed, we instruct the Transformer models not to attend to them, through the use of attention masks. Padding is therefore essentially "wasted", and we want to reduce it as much as possible.

Sequence packing is the process of packing multiple sentences into fixed-size target sequences. It has the potential of significantly reducing the amount of padding tokens, while shifting the token count distribution towards the high end. Refer to section 4.3 for an evaluation of these effects.

Raffel et al. used sequence packing in the paper that introduced the T5 architecture, where it is only mentioned in passing:

> Whenever possible, we "pack" multiple sequences into each entry of the batch so that our batches contain roughly $2^{16} = 65{,}536$ tokens. (Raffel et al., *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer* [31])

From the quote above, they used it to reduce padding and to "fill up" their training batches. However, neither the reasoning, algorithms, nor effects were further discussed in the T5 paper. The follow-up mT5 [41] and ByT5 [40] papers do not mention it at all.

In this thesis, we aim to rectify this, by justifying the use of sequence packing, describing a suitable algorithm for it, and quantifying its effects for different representation option and tokenizer combinations. For the results, see section 4.3.

3.4.1 Sequence Packing Algorithm

No sequence packing algorithm was specified in earlier work, so we had to choose one ourselves. We went with a naive and greedy algorithm, that is trivial to implement, yet empirically extremely effective, as discussed in section 4.3. Implementation and comparative evaluation of more sophisticated algorithms is left for future work.

Sequence packing corresponds to the Bin Packing problem, which is known to be NP-hard. It has many practical applications, and has therefore been extensively studied. We chose to implement the linear-time and bounded-space Next Fit algorithm, which is known to be near-optimal, despite its simplicity [22].

3.4.2 Packing-Sizes

Packing-sizes were chosen to be as large as possible for each tokenizer and representation combination, such that the models would not crash during training. Memory requirements are much lower during inference, because no gradients have to be computed, so packing-size is limited by the training step. If models can be trained successfully within some memory limit, inference will also work.

Tokenizer	Representation	Packing-Size	% of T5
google/mt5-small	list	224	21.88
	delimiters	224	21.88
google/byt5-small	list	768	75
	delimiters	640	62.5

Table 3.5: Packing-Sizes

Unfortunately, we were limited by the hardware available to us (see section 3.7), which meant that our packing-sizes had to be reduced from the 1024 tokens reference used by the T5 authors.

It is noteworthy that we were able to achieve much larger packing-sizes for the ByT5 models, compared to mT5. Its architecture and/or implementation is more memory-efficient, despite having an almost equal number of parameters.

3.5 Model Choice

Our goal was the evaluation of the influence of tokenization, ergo it makes sense to choose models which differ in the tokenization, but are otherwise as close as possible, especially architecturally, and with respect to the choice of pre-training dataset.

Transformers are currently the dominant architecture across a wide range of NLP tasks. ByT5 was chosen for being the state-of-the-art token-free Transformer architecture, at the time of writing. It is based on mT5, with the explicit intent of architecturally deviating as little as possible from that template. Both models were pre-trained on the mC4 corpus, using essentially the same hyperparemeters and task. Hence, mT5 was chosen as the token-based reference model to compare ByT5 to [31, 41, 40].

3.5.1 Model Implementations

Hugging Face Transformers provides easy-to-use implementations of foundation models, coupled with peripheral artifacts, such as tokenizers and pre-trained model checkpoints. It has been widely adopted in the industry, with a large number of available architectures and checkpoints for different model architectures and variants.

We utilize the mT5 and ByT5 implementations provided by the Transformers library. Following the papers that introduced them, they are available in different sizes, ranging from small (300M parameters) to XXL (13B parameters). Due to hardware limitations, we are only able to work with the small variants. More on that in section 3.7.

While the Transformer library's mT5 implementation stays faithful to the design presented in *mT5: A massively multilingual pre-trained text-to-text transformer*, *mT5: A massively multilingual pre-trained text-to-text transformer*, the ByT5 model deviates from the one described by Xue et al. regarding tokenization. To avoid confusion about which version is referred to, we call the Transformer library implementations of these models by the names assigned to them

within that library, which are `google/mt5-small` and `google/byt5-small`, respectively.

3.6 Deviations from the ByT5 Template

In *ByT5: Towards a token-free future with pre-trained byte-to-byte models*, *ByT5: Towards a token-free future with pre-trained byte-to-byte models*, ByT5 is called token-free, which is consentient with how it is defined in the article (see section 2.4). The authors describe a model with 256 tokens, representing all possible byte values. UTF-8 bytes are shifted to make room for 3 special tokens, but apart from that, raw Unicode is used.

`google/byt5-small`, however, is not really token-free, opting for 384 tokens instead of only 256. In addition to the 256 byte values, these encompass the 3 special tokens padding `<pad>`, end-of-sentence `</s>`, and unknown `<unk>`, as well as 125 sentinel tokens `<extra_id_0>` to `<extra_id_124>`. Given than 128 tokens are reserved for special and sentinel tokens, shifting the UTF-8 bytes would not have been necessary, yet this aspect was kept anyway.

This implementation leads to a cleaner separation between byte values and special or sentinel tokens, at the cost of having to devote more parameters to the embedding and language modeling layers of the model. ByT5-small dedicates 0.3% of its parameters to these layers, while `google/byt5-small` increases this to 0.4%.

Tokenization works as follows: First, the special and sentinel tokens are encoded. `<pad>` becomes 0, `</s>` becomes 1, and `<unk>` becomes 2. Sentinel tokens `<extra_id_0>` through `<extra_id_124>` map to token IDs 259 to 383. This leaves the 256 token IDs 3 to 258, which could be mapped to the raw UTF-8 bytes of the remaining text.

However, for some reason, this is not what the implementers chose to do. Instead, the text left after handling special tokens and sentinel tokens is UTF-8 encoded once, resulting in a sequence of bytes, each of which is UTF-8 encoded for a second time, before being shifted by 3. Doing this leaves the first 128 byte values unchanged, but the remaining 128 bytes are expanded to 2 bytes by this step.

It is unclear to us why this choice was made. Given that the increased sequence length is the main disadvantage of byte-level tokenization [40], it seems like a pessimization. The hugging face emoji, for example, which has the 3-byte UTF-8 representation `f09fa497`, is expanded to the 8-token sequence `[198, 179, 197, 162, 197, 167, 197, 154]` by this process. The simple approach would instead result in the 4-token sequence `[243, 162, 167, 154]`. `google/byt5-small`'s tokenization process appears to be more complex and less optimal, for no discernible reason.

Due to using an English-only dataset, which consists almost exclusively of characters in the ASCII range, represented as 0 to 127, this problem is orthogonal to this thesis, and does not affect our results.

3.7 Model Training

Achieving state-of-the-art performance was not a goal, so we stuck to the proven hyperparameters from the mT5 and Byt5 papers where possible. Table 3.6 contains a summary of these reference values, and the ones used by us.

	Reference		**This Thesis**	
learning rate:		0.001		0.001
dropout rate:		10%		10%
optimizer:		AdaFactor		Adam
DPT:		yes		no
	mT5	**ByT5**	`google/mt5-small`	`google/byt5-small`
batch size:	256 to 1024	1024	1	1
sequence length:	1024	1024	224	640 / 768

Table 3.6: Fine-Tuning Hyperparameters

We were severely memory-limited by only having access to a single Nvidia RTX 3080 with 10GB VRAM. Transformers are very memory-intensive, and it is common for them to be trained on clusters of dozens to hundreds of specialized GPUs or TPUs [14, 13]. We had to drastically cut the batch sizes, and also decrease the sequence lengths, to be able to barely run the small model variants `google/mt5-small` and `google/byt5-small`.

DPT was introduced to combat accidental translation, caused by monolingual fine-tuning of multilingual models (see section 2.4). We limited our investigations to the monolingual case, so we did not use it.

T5, mT5, and ByT5 were all optimized with AdaFactor, which is essentially a memory-optimized version of Adam. AdaFactor is not implemented in TensorFlow, so it was replaced with Adam.

All models were fine-tuned on the Kaggle NER dataset covered in section 3.3. It was split into 90% training set, and 10% test set. 10% of the training set was used for validation, resulting in a 81/9/10% train/validation/test split. At the end of each epoch, models were checkpointed and evaluated on the validation set. The checkpoint with the highest validation F_1-score was retained and evaluated on the test set.

Model training results are summarized in section 4.4.

4 Results

4.1 Dataset Statistics

A single sentence was filtered from the Kaggle NER dataset, for being by far longer than any other sentence at 104 words, and causing problems down the line. Apart from that, no changes were made to the dataset as available on Kaggle [39].

Table 4.1 summarizes statistics about the raw dataset, before being tokenized or sequence-packed. Figures 4.1 through 4.1 visualize sentence length, the number of mentions per sentence, and the number of consecutive mentions per sentence, respectively.

Metric	**Value**	
sentences		47,958
words		1,048,471
mentions		111,881
	Mean	**Median**
words per sentence	21.8623	21
mentions per sentence	2.3329	2
mentions per word	0.1067	0
consecutive mentions		5,581
sentences with consecutive mentions		4,893
	Mean	**Median**
consecutive mentions per sentence	0.1164	0
B tags		111,881
I tags		48,766
O tags		887,824
mean mention length		1.4359 words

Table 4.1: Raw Kaggle NER Dataset Statistics

4.2 Tokenization

Table 4.2 and Figure 4.5 show that mT5 takes much fewer tokens than ByT5 to represent the same text. This is expected, as ByT5 has more fine-grained tokens, corresponding to character

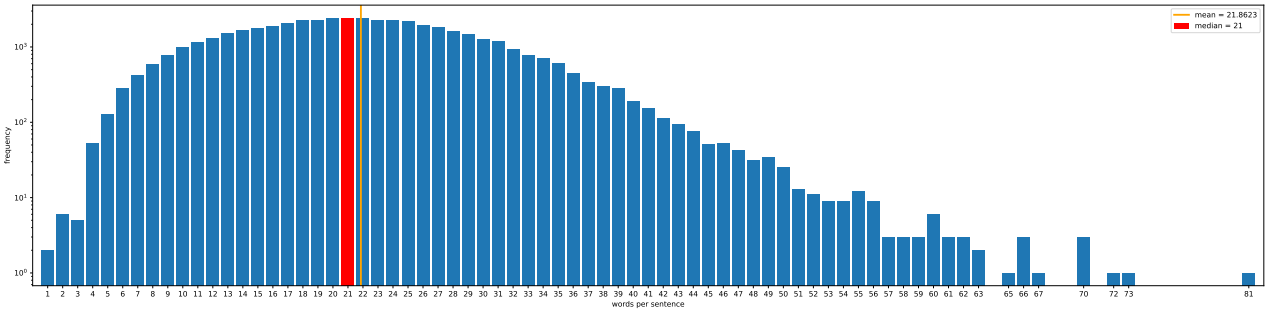


Figure 4.1: Words per Sentence

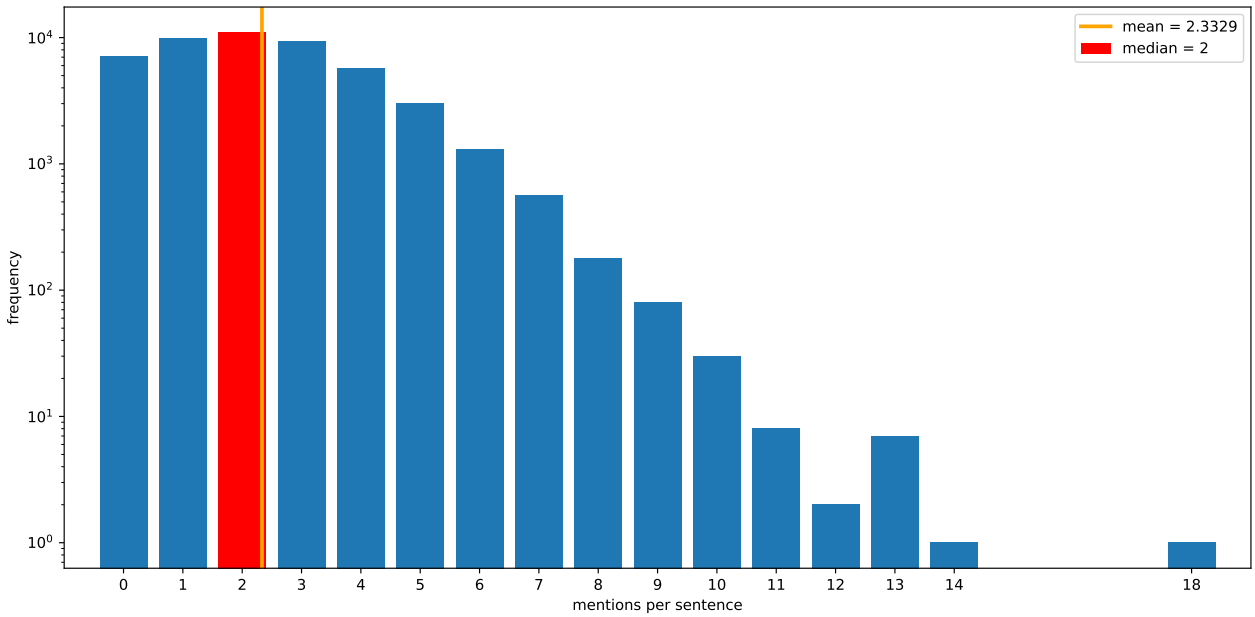


Figure 4.2: Mentions per Sentence

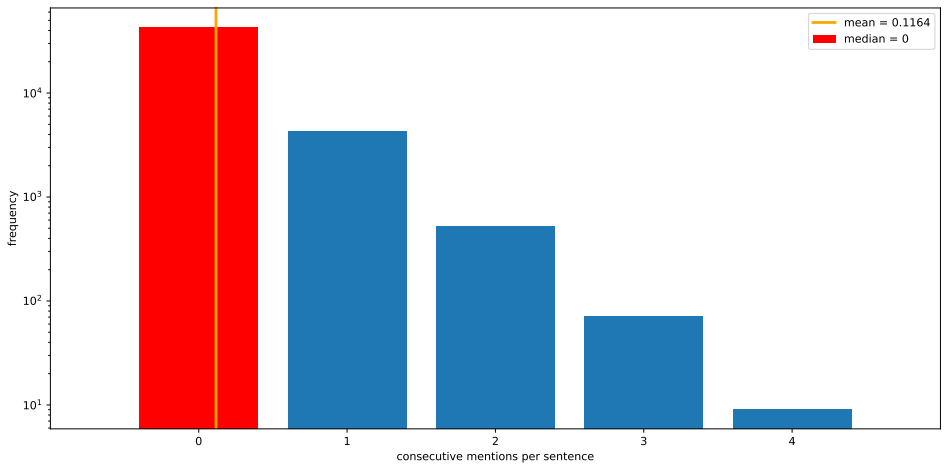


Figure 4.3: Consecutive Mentions per Sentence

and sub-character units, compared to the word and sub-word tokens of mT5.

There is a trade-off with regards to the number of tokens needed to represent some text. On one hand, mT5 can look at more text, given the same number of tokens. On the other hand, ByT5 has a higher budget to spend on encoding a fixed amount of text.

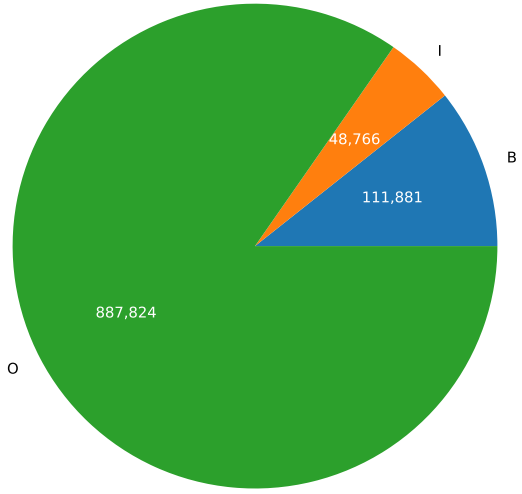


Figure 4.4: Tag Distribution

Tokenizer:	`google/mt5-small`		`google/byt5-small`	
Inputs				
non-padding tokens:	1,642,533		6,053,345	
padding tokens:	4,256,301		15,048,175	
padding portion:	72.15%		71.31%	
unique tokens:	22,566		105	
coverage of the vocabulary:	9.02%		27.34%	
mean tokens per word:	1.5666		5.7735	
Targets				
Representation:	**list**	**delimiters**	**list**	**delimiters**
non-padding tokens:	417,759	1,861,349	1,184,162	6,072,653
padding tokens:	2,171,973	4,612,981	8,359,480	15,076,825
padding portion:	83.87%	71.25%	87.59%	71.29%
unique tokens:	12,942	23,866	101	121
coverage of the vocabulary:	5.17%	9.54%	26.30%	31.51%

Table 4.2: Tokenized Kaggle NER Dataset Statistics

Due to having miniscule vocabulary size compared to mT5, ByT5 has better coverage of its vocabulary. mT5-small spends 85% of its parameter budget on the embedding and language modeling layers. A large part of those parameters is wasted, if their corresponding tokens are never used. Having more examples per token also eases the calculation of gradients, and can lead to more stable and faster training, ultimately leading to better inference performance, as seen in section 4.4.

4.3 Sequence Packing

Tokenizer:	`google/mt5-small`		`google/byt5-small`	
Representation:	**list**	**delimiters**	**list**	**delimiters**
Packing-Size:	**224**	**224**	**768**	**640**
packed sequences:	8,004	9,183	8,691	10,680
mean sentences per sequence:	5.9918	5.2225	5.5181	4.4904
Inputs				
non-padding tokens:	1,642,533	1,642,533	6,053,345	6,053,345
padding tokens:	150,363	414,459	621,343	781,855
padding portion:	8.39%	20.15%	9.31%	11.44%
Targets				
non-padding tokens:	417,759	1,861,349	1,184,162	6,072,653
padding tokens:	630,765	195,643	1,822,924	762,547
padding portion:	60.16%	9.51%	60.62%	11.16%

Table 4.3: Sequence-Packed Kaggle NER Dataset Statistics

Sequence packing (see section 3.4) is extraordinarily successful at improving training efficiency, due to a shift from sentences consisting mostly of padding tokens to sequences with comparatively little padding. This means more data can be fitted into a single training batch, speeding up the training process without decreasing the quality of the resulting model.

Tokenizer	**Representation**	**% Padding Reduction**		
		Inputs	**Targets**	**Overall**
`google/mt5-small`	list	96.47	70.96	87.85
	delimiters	90.26	95.76	93.12
`google/byt5-small`	list	95.87	78.19	89.56
	delimiters	94.80	94.94	94.87

Table 4.4: Padding Reduction from Sequence Packing

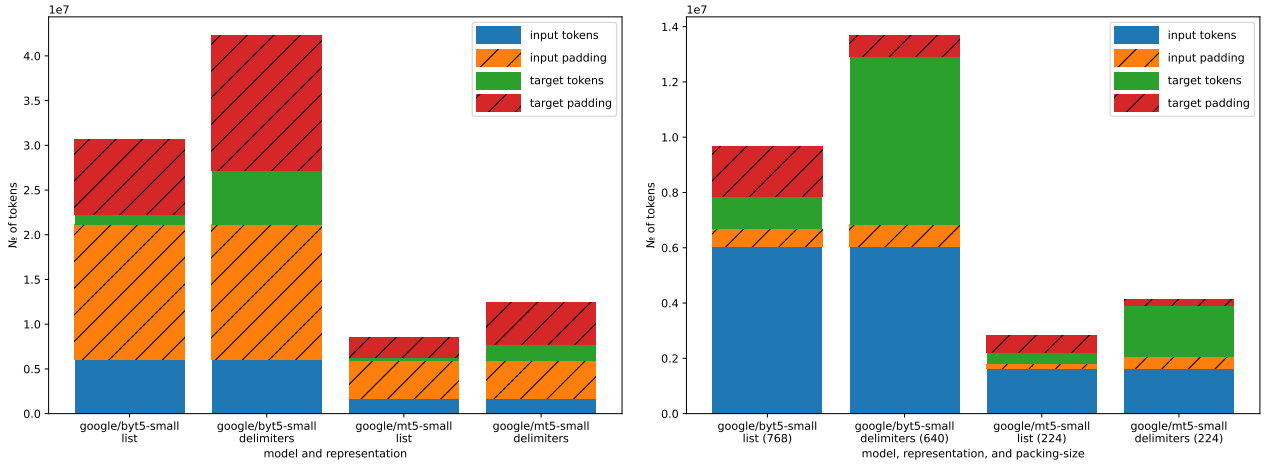
Unpacked, tokenized sentences consist of 71.30% – 76.38% padding, which is reduced by 87.85% – 94.87% (see Table 4.4) to 11.30% – 27.49% padding for packed sequences. This results in token counts per sequence being roughly 360% – 600% higher than the corresponding token counts per sentence, as summarized in Table 4.5.

Figure 4.5 shows the token type fractions for different tokenizer, representation option, and packing-size combinations. It highlights padding making up the majority of the tokenized sentences before packing, as mentioned above. Sequence packing does not affect the number of non-padding tokens, while the number of padding tokens is materially reduced, resulting in (a) a large overall decrease of tokens required to represent the datasets, and (b) an enormous reduction of the portion taken up by padding tokens.

Sequence packing shifts token count distributions from the low end to the high end, as depicted in Figure 4.6. Regardless of tokenizer, dataset representation, and packing-size, resulting

Tokenizer	Representation	Packing-Size	Median Tokens per Sentence	Sequence	% Increase
`google/mt5-small`	list	224	18.0	125.5	597.22
	delimiters	224	36.0	191.0	430.56
`google/byt5-small`	list	768	58.0	343.0	491.38
	delimiters	640	125.0	576.0	360.80

Table 4.5: Token Count Increase from Sequence Packing



(a) Tokenized Sentences

(b) Packed Sequences

Figure 4.5: Token Type Fractions

sequences are much "fuller", containing many more useful tokens and less padding, compared to the sentences they were derived from.

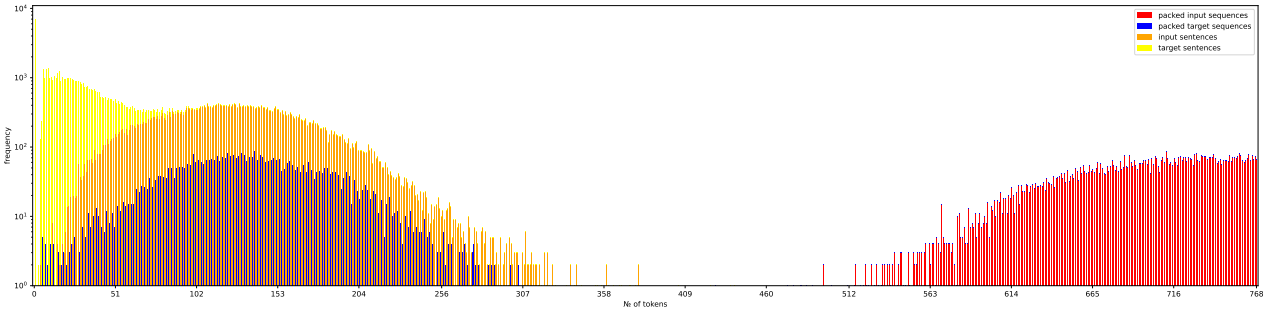
Due to the time and space complexity of Transformer models, it is not always beneficial to increase the sequence length.

> The self-attention mechanism at the core of the ubiquitous Transformer architecture has a quadratic time and space complexity in the sequence length, so [longer] sequences can result in a significantly higher computational cost. (Xue et al., *ByT5: Towards a token-free future with pre-trained byte-to-byte models* [40])

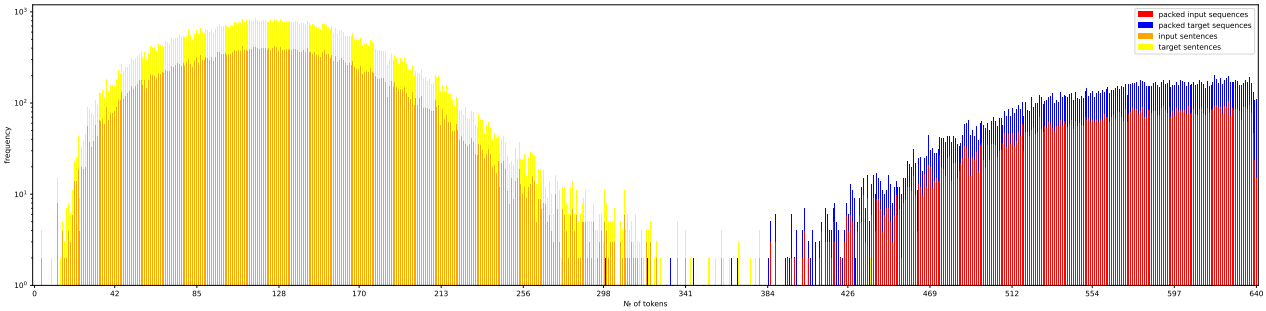
Nonetheless, we were so severely memory-bottlenecked that sequence packing resulted in a training time reduction equivalent to the mean number of sentences per sequence. In other words, packing 5 sentences into a sequence equates to an 80% reduction in training time. With the packing-sizes we could reach within our memory budget, we were unable to saturate the GPU, so scaling was linear.

4.4 Model Training

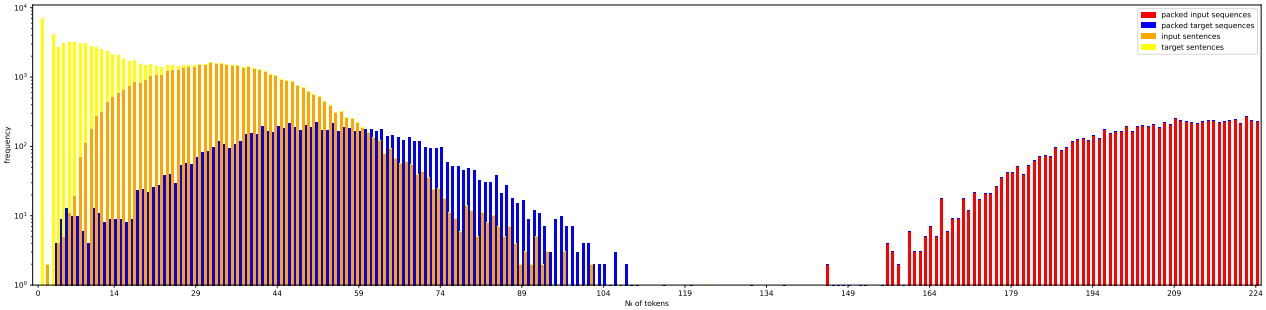
Figures 4.8 through 4.11 show the results of successful training runs of different model and problem representation combinations. All training plots share the same legend, depicted in



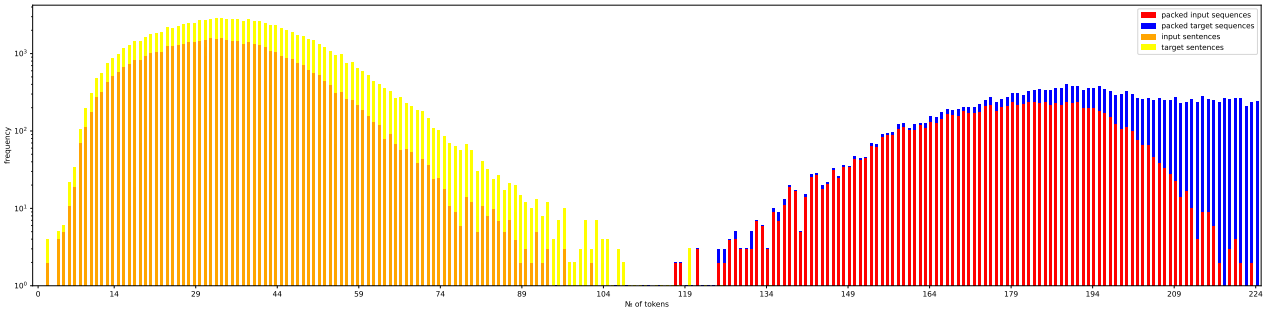
(a) `google/byt5-small`, list representation, packed to 768 tokens



(b) `google/byt5-small`, delimiters representation, packed to 640 tokens



(c) `google/mt5-small`, list representation, packed to 224 tokens



(d) `google/mt5-small`, delimiters representation, packed to 224 tokens

Figure 4.6: Token Count Distributions

Figure 4.7.

ByT5's training plots look like one would expect from successful training runs. Loss quickly decreases, until it reaches the neighborhood of its asymptote. The metrics mirror this behavior, by quickly growing until an equilibrium is reached. Training is stable, exhibiting low volatility.

mT5 shows much more erratic behavior during training. It takes some time to ramp up, occasionally not improving for several epochs. If and when it finally does improve, it does not

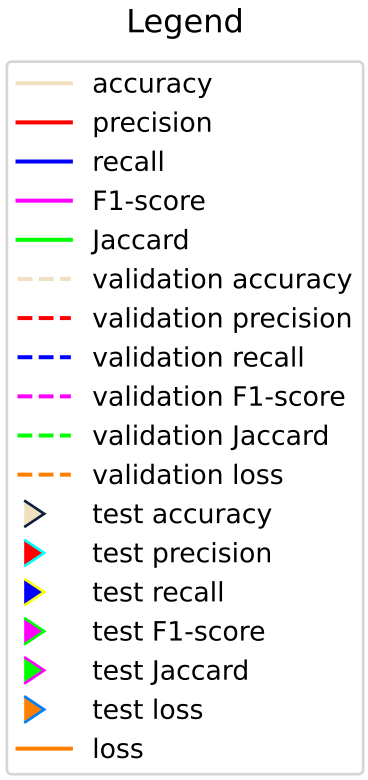


Figure 4.7: Legend for Training Plots

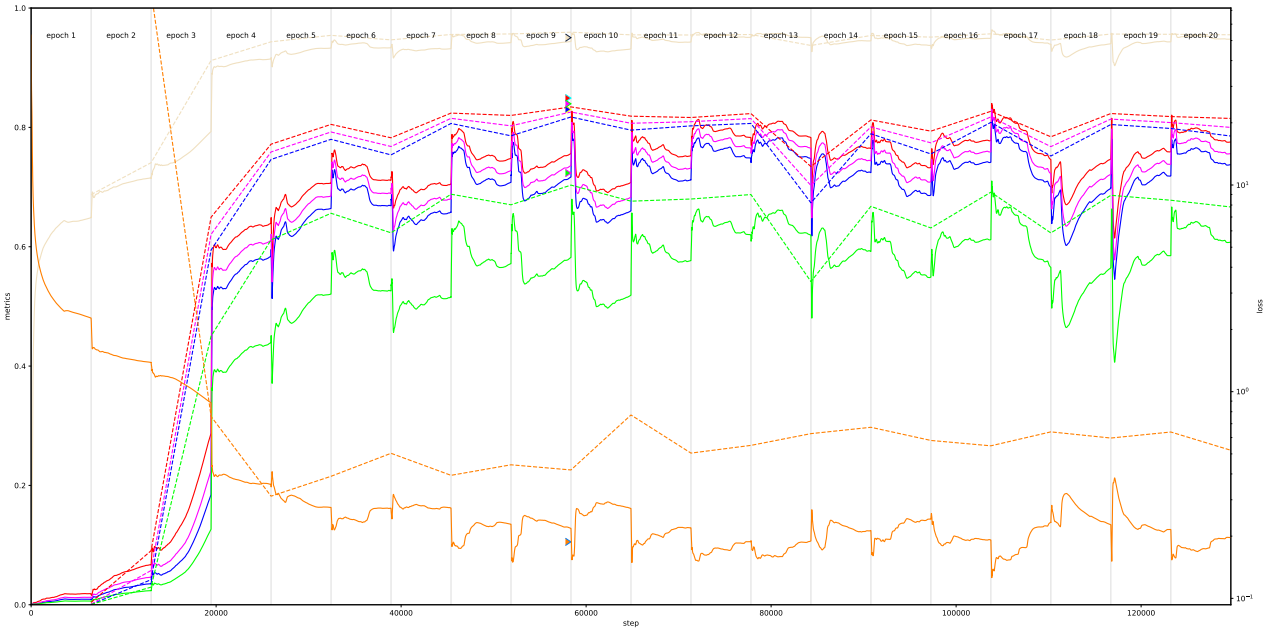


Figure 4.8: `google/mt5-small` Training on the List Representation

stay close to its asymptote, even after reaching it. Instead, loss and metrics vary unpredictably. Sometimes, the models do not improve at all, or suffer through catastrophic collapses. Most of the time, the models do not recover from them, though rarely they do. Plots of failed training runs are included in [reference failed mT5 training plots].

We do not have such a collection of failed training runs to show for ByT5, for the simple reason that it is much easier to train. We did not experience failed runs like for mT5.

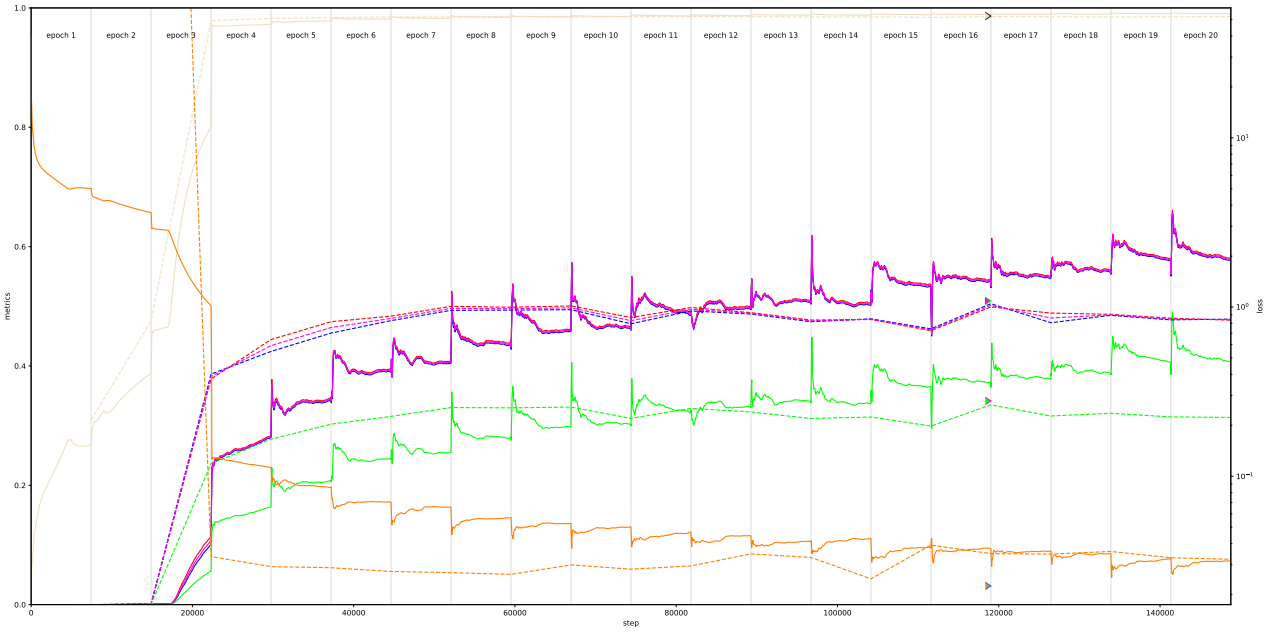


Figure 4.9: `google/mt5-small` Training on the Delimiters Representation

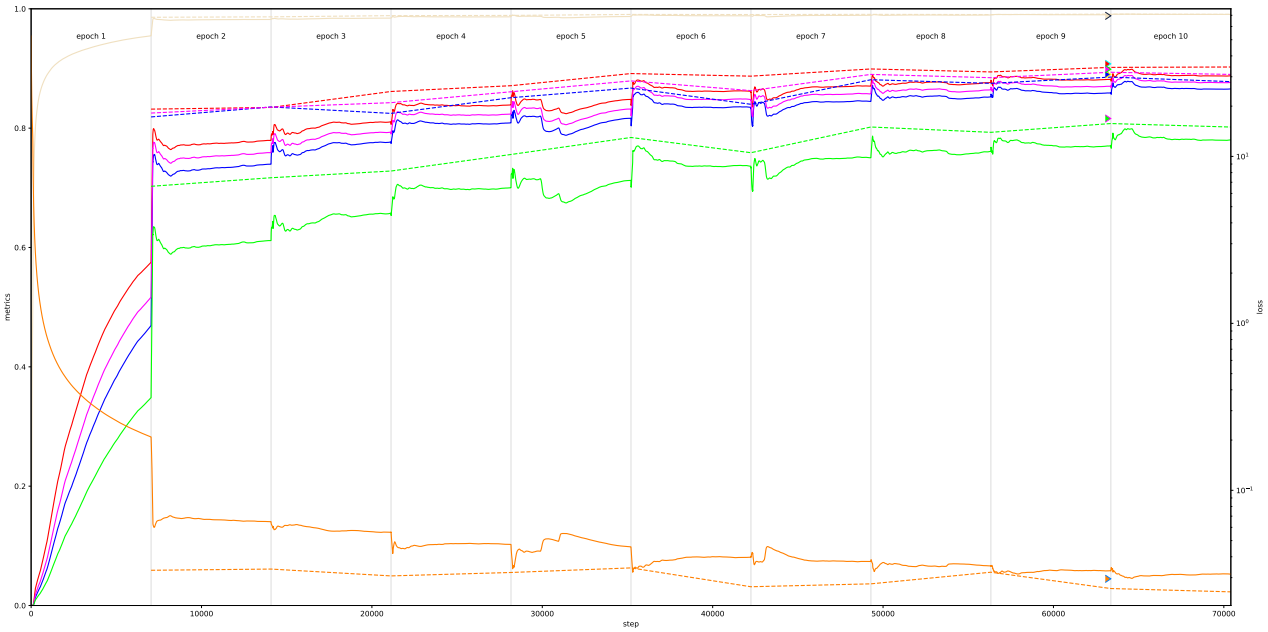


Figure 4.10: `google/byt5-small` Training on the List Representation

Evaluation results for the resulting models are summarized in Table 4.6. All model and representation combination reach good performance, with the exception of the mT5 model on the delimiters representation. For that combination, we were never able to achieve F_1-scores significantly above 50%.

The erratic training behavior of mT5, and its bad performance on the delimiters representation, might be explained by `google/mt5-small` dedicating 85% of its parameter budget on the embedding and language modeling layers, leaving only 15% for the encoder and decoder stacks. This is somewhat amortized for larger variants, but even mT5-XXL still has 16% of its parameters tied up in dealing with its enormous vocabulary. Most of the tokens are never used by the Kaggle NER dataset, and this capacity is therefore wasted.

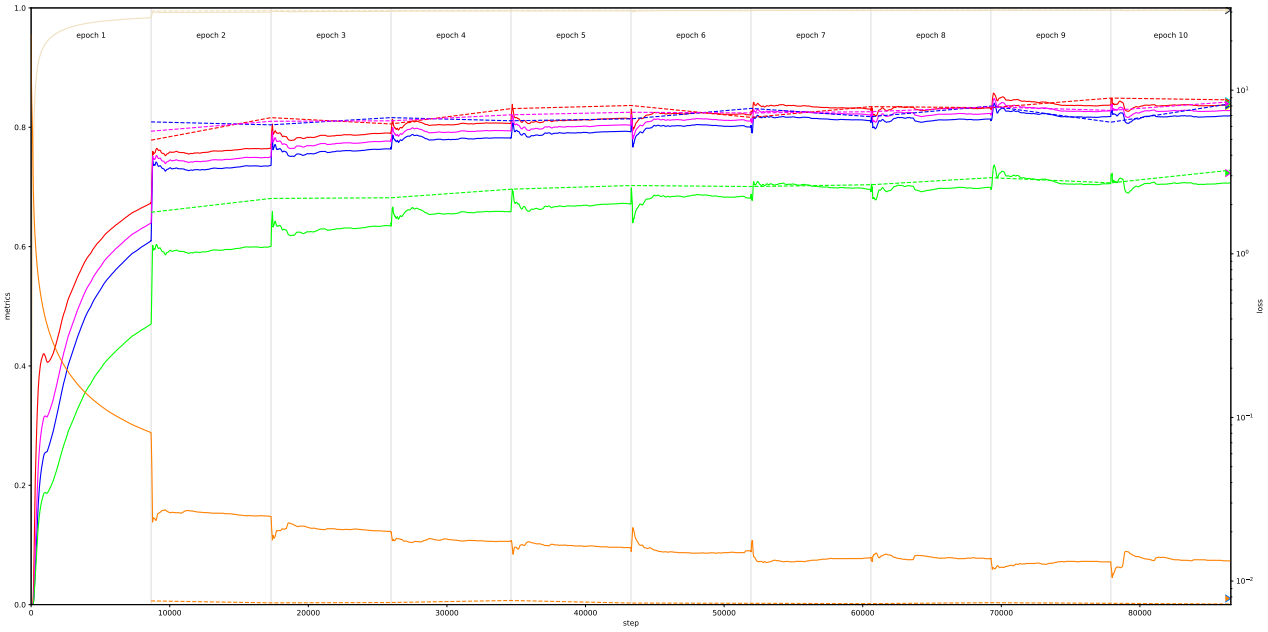


Figure 4.11: `google/byt5-small` Training on the Delimiters Representation

Model	Representation	Test Set				
		Accuracy	**Precision**	**Recall**	**F_1-score**	**Jaccard Index**
`google/mt5-small`	list	0.9503	0.8491	0.8302	0.8396	0.7235
	delimiters	0.9864	0.5089	0.5092	0.5091	0.3414
`google/byt5-small`	list	0.9882	0.9077	0.8900	0.8988	0.8161
	delimiters	0.9963	0.8440	0.8348	0.8394	0.7232

Table 4.6: Evaluation Results for Best-Performing Models

ByT5 and mT5 also differ in their d_{ff}, d_{model}, `num_layers`, and `relative_attention_num_buckets` parameters. One or several of these could also contribute to the difference in performance. A deeper analysis is left for future work.

5 Conclusions

In this chapter, we summarize the conclusions drawn from the results obtained in this thesis. We close with ideas for future work.

When it comes to representation of NER problems, there were many options, but two clear favorites: the entity list and entity delimiters representation options, described in sections 3.1.2 and 3.1.3, respectively. Among these two, it is hard to pick a clear winner. Each option has its unique advantages and drawbacks.

Entity lists are a straightforward and simple representation option. It is empirically easier for models to learn than the delimiters representation, at the cost of missing positional information about entity mentions. If that information is not required, this option is a good choice, because we measured better inference results. Both models achieved satisfactory performance on the list representation, with ByT5 even reaching an F_1-score of approximately 90%, the highest result we attained.

The delimiters representation is more varsatile and flexible, but is more difficult for models to learn and generate. We achieved lower inference scores with this option, but it also contains more information than the list representation. `google/mt5-small` was not able to achieve better than roughly 50% F_1-score with the delimiters representation.

For the NER task on the Kaggle dataset, precision seems generally easier to achieve than recall. Across different models and representation options, precision is consistently higher than recall.

We confirmed ByT5's impressive generative abilities, claimed by its authors. Despite having fewer decoder layers than mT5, it was able to show superior generative capacity on the delimiters representation option. This is especially impressive considering that ByT5 has to operate on the (sub-)character level, whereas mT5 can generate (sub-)word units.

mT5, due to its huge vocabulary, spends a large fraction of its parameter budget on its embedding and language modeling layers, which could have otherwise been used to improve its generative capabilities. ByT5, having a very small vocabulary, spends a negligible amount of parameters on it, leaving more for other aspects.

Hugging Face Transformer's ByT5 implementation `google/byt5-small` is not really token-free, contrary to the template it was built on. This is because of different trade-offs made by its implementers. It is unclear whether these trade-offs are worthwhile.

`google/byt5-small` uses a strange double UTF-8 encoding scheme, which duplicates the number of tokens required to represent bytes outside of the ASCII range. We do not see an advantage to this decision, which seems to us like a regression when compared to the obvious choice

specified in *ByT5: Towards a token-free future with pre-trained byte-to-byte models*. Due to our choice of dataset, this specific drawback does not affect us.

Mention extraction algorithms influence the calculation of evaluation metrics. To guarantee reproducibility and comparability between different publications, they should be documented and published, which is what we did in this work.

Sequence packing is a great method for increasing efficiency, without diminishing model performance. It has been strangely neglected in the literature, in spite of its ability to drastically reduce training time. We rectified the lack of information on the subject, though there still remains research to be done (see below).

5.1 Future Work

We thoroughly motivated our choice of dataset, and are quite confident in it. Still, it should be confirmed that our findings generalize to other datasets as well.

Transformer language models show emergent behavior at larger sizes. There is even evidence to suggest that small and large transformers behave so differently that one should not generalize between them [14, 13]. As such, it remains to be confirmed that our results hold for larger model variants.

mT5 and ByT5 are designed to be multilingual, and both are pre-trained the the mC4 dataset, comprising 101 languages. To keep things simple, we limited ourselves to a monolingual, English-only dataset, following predecents set in the literature. Future work could expand the analysis presented here to multilingual environments.

Considering that this thesis is mainly about the influence of tokenization, we already dove quite deep into sequence packing, simply because it was such a large gap in prior art, yet essential for efficiency. However, there are still many aspects that could use even deeper examination, including

- a quantitative analysis of the effects of sequence packing on model performance,
- an assessment of errors made by models trained with and without sequence packing, or with different sequence packing parameters,
- the determination of the saturation point at which training efficiency is no longer improved by larger packing-sizes, and
- an investigation of the interactions between batch size and sequence length.

Bibliography

[1] University of Antwerpen. *CoNLL-2003*. 2003. URL: https://www.cnts.ua.ac.be/conll2003/ (visited on 2022-08-15).

[2] Yoshua Bengio et al. "A Neural Probabilistic Language Model". In: *Journal of Machine Learning Research* 3 (2003-03). Ed. by Jaz Kandola et al., pp. 1137–1155. ISSN: 1532-4435. DOI: 10.5555/944919.944966. URL: https://dl.acm.org/doi/10.5555/944919.944966.

[3] Johan Bos and Valerio Basile. *Groningen Meaning Bank*. 2022. URL: https://gmb.let.rug.nl/ (visited on 2022-08-15).

[4] Johan Bos et al. "The Groningen Meaning Bank". In: *Handbook of Linguistic Annotation*. Ed. by Nancy Ide and James Pustejovsky. Vol. 2. Springer, 2017, pp. 463–496.

[5] Stevo Bozinovski. "Reminder of the First Paper on Transfer Learning in Neural Networks, 1976". In: *Informatica* 44.3 (2019-06), pp. 291–302. DOI: 10.31449/inf.v44i3.2828.

[6] Stevo Bozinovski and Ante Fulgosi. "The influence of pattern similarity and transfer learning upon training of a base perceptron B2 (original in Croatian: Utjecaj slicnosti likova i transfera ucenja na obucavanje baznog perceptrona B2)". In: *Proceedings of the symposium Informatica*. Bled, Slovenia, 1976.

[7] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020-07. arXiv: 2005.14165 [cs.CL].

[8] Yuen Ren Chao. *A Project for General Chinese*. Beijing: Shāngwù Yìnshūguǎn, 1983. URL: https://openlibrary.org/books/OL24456574M/A_Project_for_General_Chinese.

[9] Kevin Clark et al. *ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators*. 2020-03. arXiv: 2003.10555 [cs.CL].

[10] Alexis Conneau et al. "Unsupervised Cross-lingual Representation Learning at Scale". In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, 2020-07, pp. 8440–8451. DOI: 10.18653/v1/2020.acl-main.747. URL: https://aclanthology.org/2020.acl-main.747.

[11] Nvidia Corporation. *Tensor Cores: Versatility for HPC & AI*. 2022. URL: https://www.nvidia.com/en-us/data-center/tensor-cores/ (visited on 2022-08-18).

[12] Jia Deng et al. "ImageNet: A Large-Scale Hierarchical Image Database". In: *Conference on Computer Vision and Pattern Recognition*. Miami, Florida, USA: IEEE, 2009-06, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848. URL: https://www.image-net.org/.

[13] Tim Dettmers. *LLM.int8() and Emergent Features*. 2022-08. URL: https://timdettmers.com/2022/08/17/llm-int8-and-emergent-features/ (visited on 2022-08-20).

[14] Tim Dettmers et al. *LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale*. 2022-08. DOI: 10.48550/ARXIV.2208.07339. arXiv: 2208.07339 [cs.LG].

[15] Jacob Devlin. *Multilingual BERT*. 2018. URL: https://github.com/google-research/bert/blob/master/multilingual.md (visited on 2022-08-15).

[16] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019-05. arXiv: 1810.04805 [cs.CL].

[17] Antske Fokkens, Vivek Srikumar, and Jens Lemmens. *CoNLL*. 2022. URL: https://conll.org/ (visited on 2022-08-15).

[18] Margalit Fox. "Zhou Youguang, Who Made Writing Chinese as Simple as ABC, Dies at 111". In: *The New York Times* (2017-01). URL: https://www.nytimes.com/2017/01/14/world/asia/zhou-youguang-who-made-writing-chinese-as-simple-as-abc-dies-at-111.html (visited on 2021-11-12).

[19] Pengcheng He et al. *DeBERTa: Decoding-enhanced BERT with Disentangled Attention*. 2021-10. arXiv: 2006.03654 [cs.CL].

[20] Joel Hestness et al. *Deep Learning Scaling is Predictable, Empirically*. 2017-12. DOI: 10.48550/ARXIV.1712.00409. arXiv: 1712.00409 [cs.LG].

[21] Eric Huang et al. "Improving Word Representations via Global Context and Multiple Word Prototypes". In: *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics*. Vol. 1. Jeju Island, Korea: Association for Computational Linguistics, 2012-07, pp. 873–882. URL: https://aclanthology.org/P12-1092.

[22] David Stifler Johnson. "Near-Optimal Bin Packing Algorithms". PhD thesis. Massachusetts Institute of Technology, 1973-06.

[23] Zhenzhong Lan et al. *ALBERT: A Lite BERT for Self-supervised Learning of Language Representations*. 2020-02. arXiv: 1909.11942 [cs.CL].

[24] Yinhan Liu et al. *Multilingual Denoising Pre-training for Neural Machine Translation*. 2020-01. DOI: 10.48550/ARXIV.2001.08210. arXiv: 2001.08210 [cs.CL].

[25] Google LLC. *Train and run machine learning models faster — Cloud TPU*. 2022. URL: https://cloud.google.com/tpu (visited on 2022-08-18).

[26] Dhruv Mahajan et al. *Exploring the Limits of Weakly Supervised Pretraining*. 2018-05. DOI: 10.48550/ARXIV.1805.00932. arXiv: 1805.00932 [cs.CV].

[27] Tomas Mikolov et al. *Efficient Estimation of Word Representations in Vector Space*. 2013-09. DOI: 10.48550/ARXIV.1301.3781. arXiv: 1301.3781 [cs.CL].

[28] Mike Murphy. *What are foundation models?* 2022-05. URL: https://research.ibm.com/blog/what-are-foundation-models (visited on 2022-06-01).

[29] Xiaoman Pan et al. "Cross-lingual Name Tagging and Linking for 282 Languages". In: *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*. Vol. 1. Vancouver, Canada: Association for Computational Linguistics, 2017-07, pp. 1946–1958. DOI: 10.18653/v1/P17-1178. URL: https://aclanthology.org/P17-1178.

[30] Matthew E. Peters et al. *Deep contextualized word representations*. 2018-03. DOI: 10.48550/ARXIV.1802.05365. arXiv: https://arxiv.org/abs/1802.05365 [cs.CL].

[31] Colin Raffel et al. *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer*. 2020-07. arXiv: 1910.10683 [cs.LG].

[32] Lance A. Ramshaw and Mitchell P. Marcus. *Text Chunking using Transformation-Based Learning*. 1995-05. arXiv: cmp-lg/9505040 [cmp-lg].

[33] Mike Schuster and Kaisuke Nakajima. "Japanese and Korean voice search". In: *International Conference on Acoustics, Speech and Signal Processing*. ICASSP 2012. Kyoto, Japan: IEEE, 2012-03, pp. 5149–5152. DOI: 10.1109/ICASSP.2012.6289079.

[34] Noam Shazeer et al. *Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer*. 2017-01. DOI: 10.48550/ARXIV.1701.06538. arXiv: 1701.06538 [cs.LG].

[35] Emma Strubell, Ananya Ganesh, and Andrew McCallum. *Energy and Policy Considerations for Deep Learning in NLP*. 2019-06. DOI: 10.48550/ARXIV.1906.02243. arXiv: 1906.02243 [cs.CL].

[36] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. *Sequence to Sequence Learning with Neural Networks*. 2014-12. DOI: 10.48550/ARXIV.1409.3215. arXiv: 1409.3215 [cs.CL].

[37] Unknown. "Pinyin celebrates 50th birthday". In: *Xinhua News Agency* (2008-02). URL: http://www.china.org.cn/english/news/242463.htm (visited on 2021-11-12).

[38] Ashish Vaswani et al. *Attention Is All You Need*. 2017-12. DOI: 10.48550/ARXIV.1706.03762. arXiv: 1706.03762 [cs.CL].

[39] Abhinav Walia. *Annotated Corpus for Named Entity Recognition*. 2017. URL: https://www.kaggle.com/datasets/abhinavwalia95/entity-annotated-corpus (visited on 2022-08-15).

[40] Linting Xue et al. *ByT5: Towards a token-free future with pre-trained byte-to-byte models*. 2021-05. arXiv: 2105.13626 [cs.CL].

[41] Linting Xue et al. *mT5: A massively multilingual pre-trained text-to-text transformer*. 2021-03. DOI: 10.48550/ARXIV.2010.11934. arXiv: 2010.11934 [cs.CL].

[42] Peilin Yang, Hui Fang, and Jimmy Lin. "Anserini: Enabling the Use of Lucene for Information Retrieval Research". In: *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*. SIGIR '17. Shinjuku, Tokyo, Japan: Association for Computing Machinery, 2017-08, pp. 1253–1256. ISBN: 9781450350228. DOI: 10.1145/3077136.3080721.

A Mention Extraction

This chapter contains additional information regarding section 3.2: Mention Extraction.

A.1 List Extraction Examples

If all mentions are empty, the number of start tokens does not matter, since empty mentions are not output. Mentions containing only whitespace and special tokens are considered empty.

[S-a]

(a) Single Empty Mention

[S-a] [S-b]

(b) Multiple Empty Mentions

[S-a] [E-a]

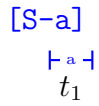
(c) Single Empty Mention with Special Token

[S-a] [E-a] [S-b]

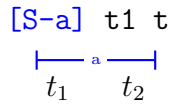
(d) Multiple Empty Mentions with Special Tokens

Figure A.1: Empty Mentions

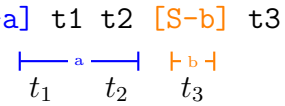
Well-formed outputs contain a sequence of mentions, consisting of start tokens followed by an arbitrary number of tokens making up the mention.



(a) Single-Token Mention



(b) Multi-Token Mention



(c) Multiple Mixed Mentions

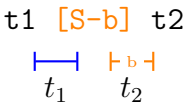
Figure A.2: Well-Formed Outputs

For the first mention in a mention sequence, the start token can be left out. However, in that case, it is not possible to determine the entity type, only the mention itself. n mentions must be separated by at least $n - 1$ start tokens.

Consecutive special tokens are essentially treated like whitespace. They are ignored at the start and end of mentions. When occuring inside of a mention, they are treated like a single space token.

(a) Single Mention

(b) Multiple Mentions, One Empty

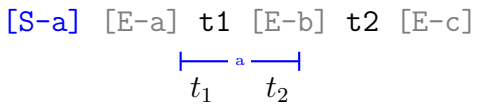


(c) Multiple Mentions

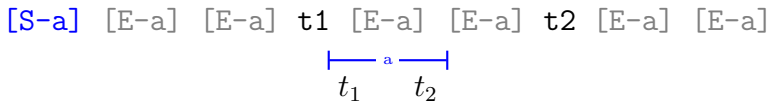
Figure A.3: Missing Start Tokens

[PAD] [EOS] [UNK] [E-a] [E-b] [E-c]...

(a) No Mention, Solely Special Tokens



(b) Single Special Tokens



(c) Spans of Special Tokens

Figure A.4: Special Tokens

A.2 Delimiters Extraction Examples

Option 2: Entity Delimiters being the most flexible and versatile of the representation options means that it also presents the most edge cases and ambiguities to handle, and therefore has the most complex extraction algorithm (see subsection 3.2.2).

Empty mentions can be demarcated by a start and an end token, like they are supposed to. However, both the start and end tokens could be missing.

[S-a] [E-a]

(a) Normal

[S-a]

(b) Start Only

[E-a]

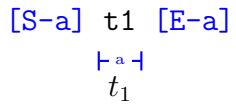
(c) End Only

Figure A.5: Empty Mentions

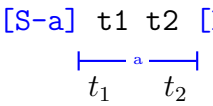
Well-formed outputs contain start and end delimiters surrounding every mention. Unlike Option 1: Entity List, the outputs do not only contain the mentions themselves, but also surrounding context. Essentially, the inputs are annotated with additional tokens.

Entity delimiters allow encoding nested and partially-overlapping mentions, as discussed in subsection 3.1.3. Nested mentions are handled as one would expect. For simplicity, only two levels of nesting are shown here, although the algorithm supporty arbitrary nesting depth.

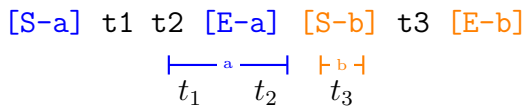
As mentioned above, delimiters normally occur in pairs surrounding mentions. Due to the free-form generative output process, there can be an imbalance between delimiters, such that



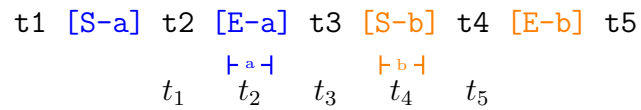
(a) Single-Token Mention



(b) Multi-Token Mention

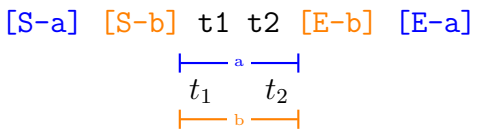


(c) Multiple Mixed Mentions

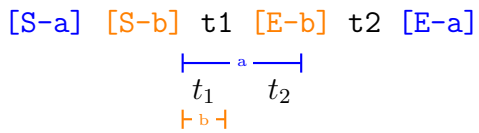


(d) Including Context

Figure A.6: Well-Formed Outputs

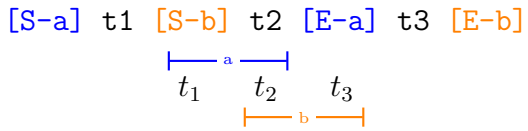


(a) Same Length

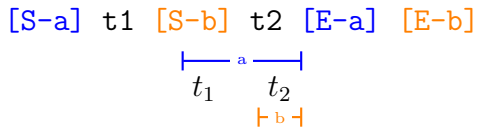


(b) Different Lengths

Figure A.7: Nested Mentions



(a) Real Overlap

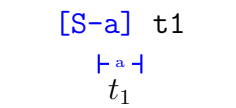


(b) Faux Overlap

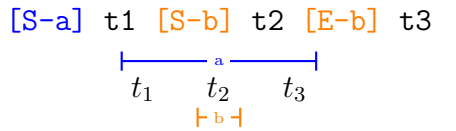
Figure A.8: Partially-Overlapping Mentions

they do not form pairs.

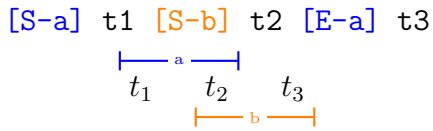
All imbalanced start tokens are closed when encountering the end of the token sequence.



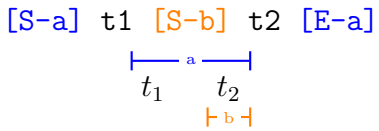
(a) Simple



(b) Nested



(c) Partial Overlap



(d) Faux Partial Overlap

Figure A.9: Imbalanced Start Delimiters

Unmatched end tokens, on the other hand, are simply treated as whitespace.

(a) Simple

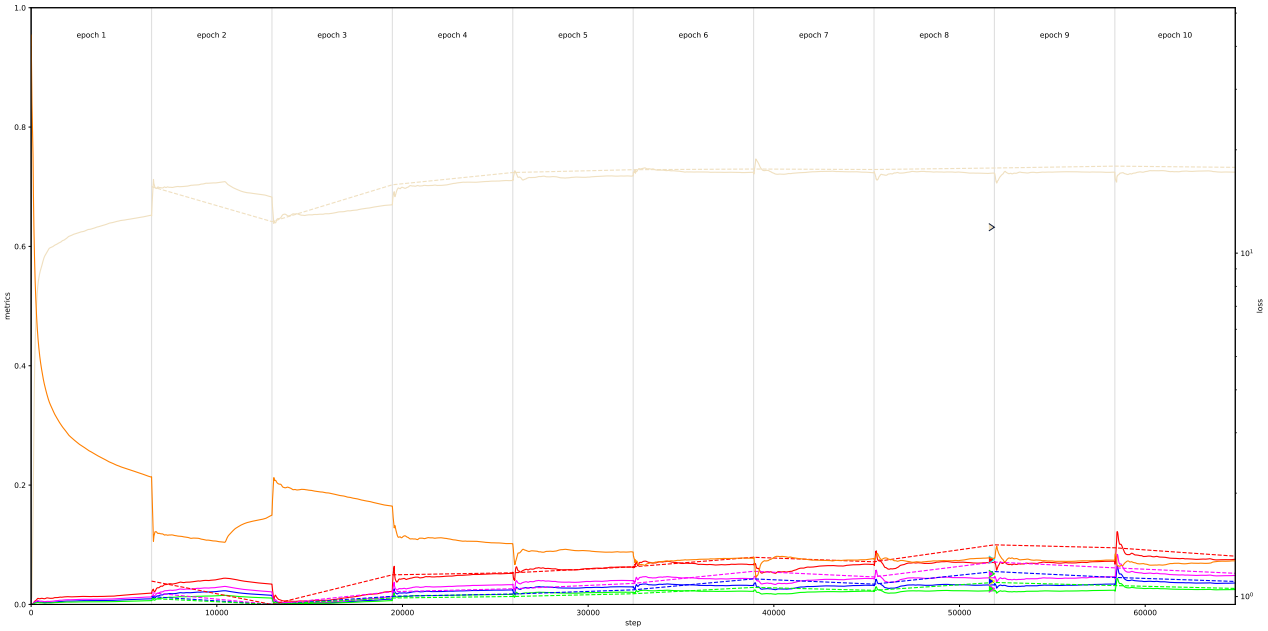
(b) Overlapping

Figure A.10: Imbalanced End Delimiters

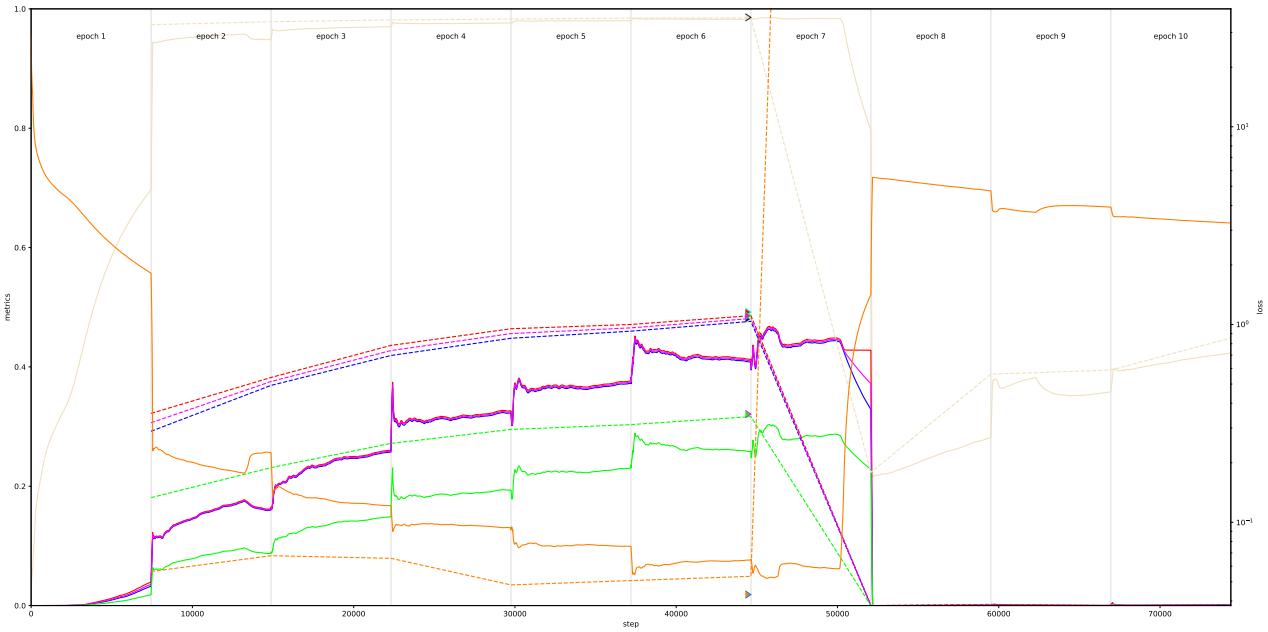
B Failed mT5 Training Runs

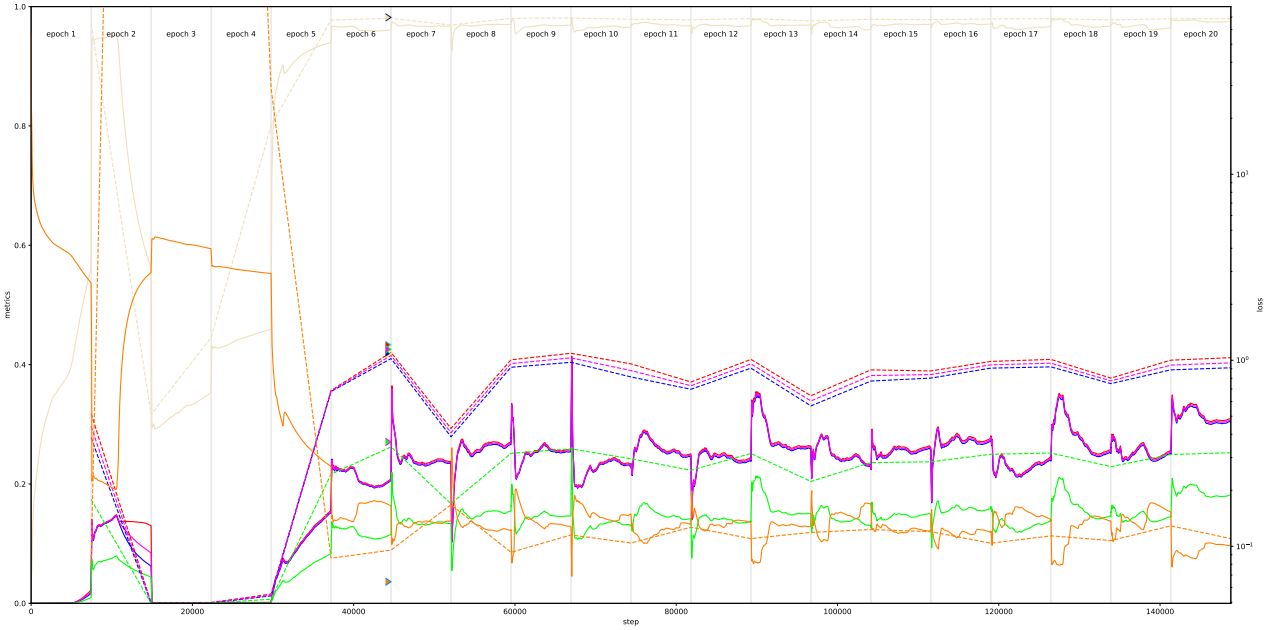
This chapter contains plots of failed mT5 training runs. See Figure 4.7 for the legend.

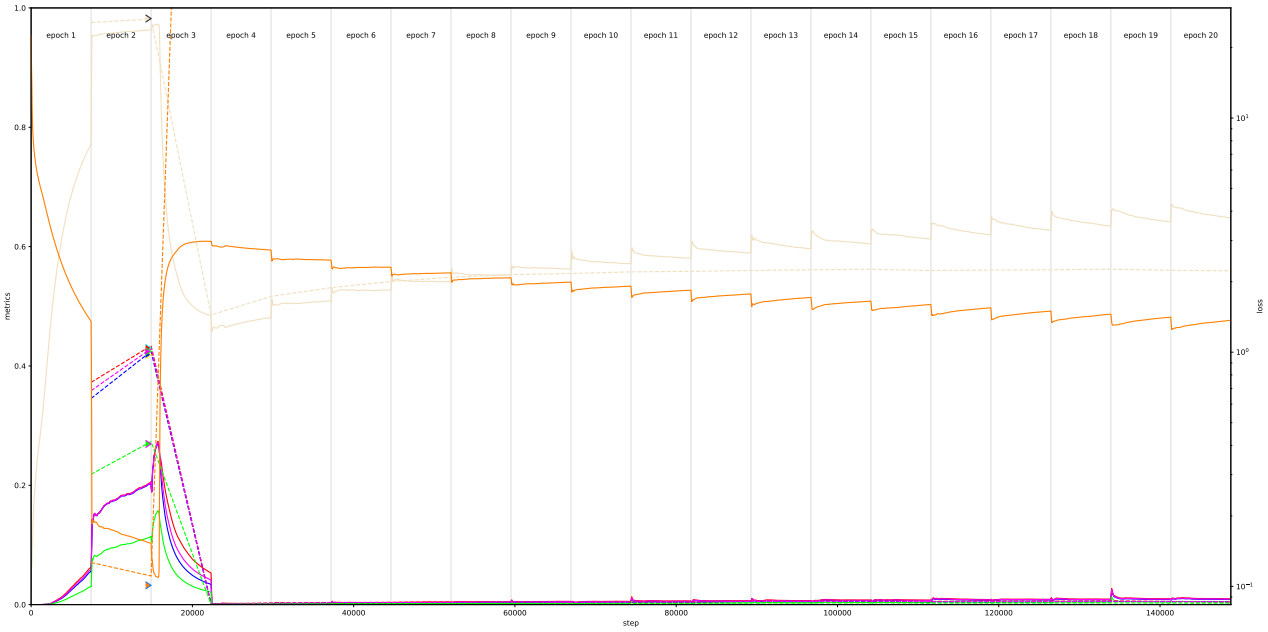
B.1 List Representation

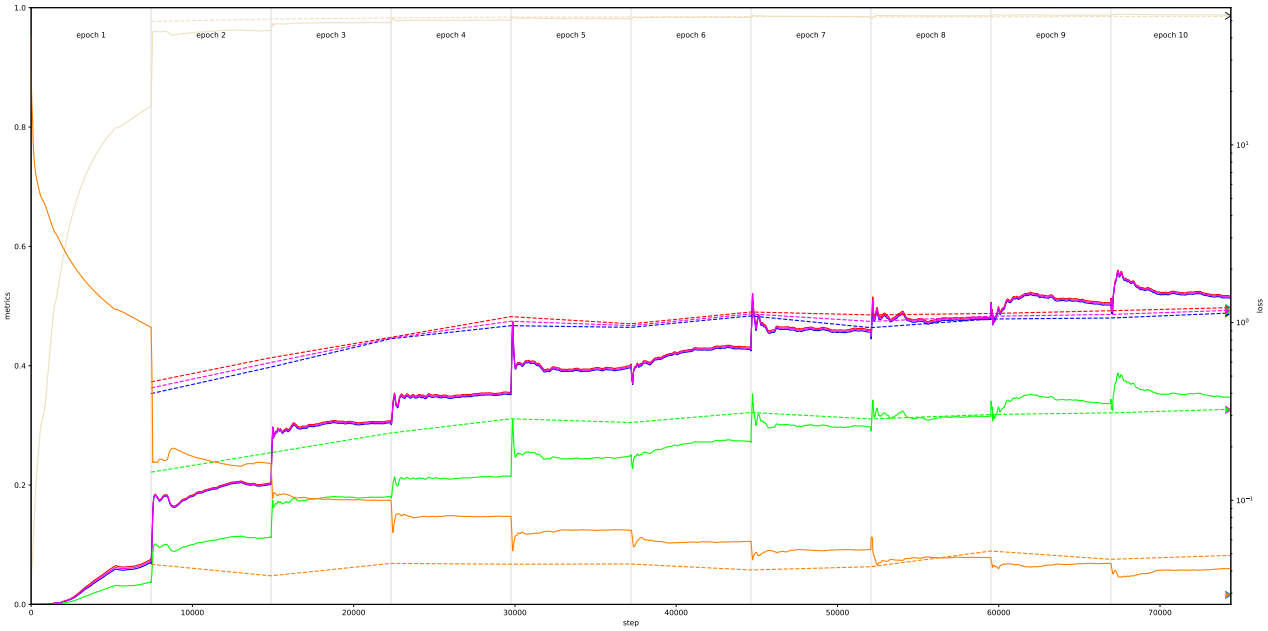


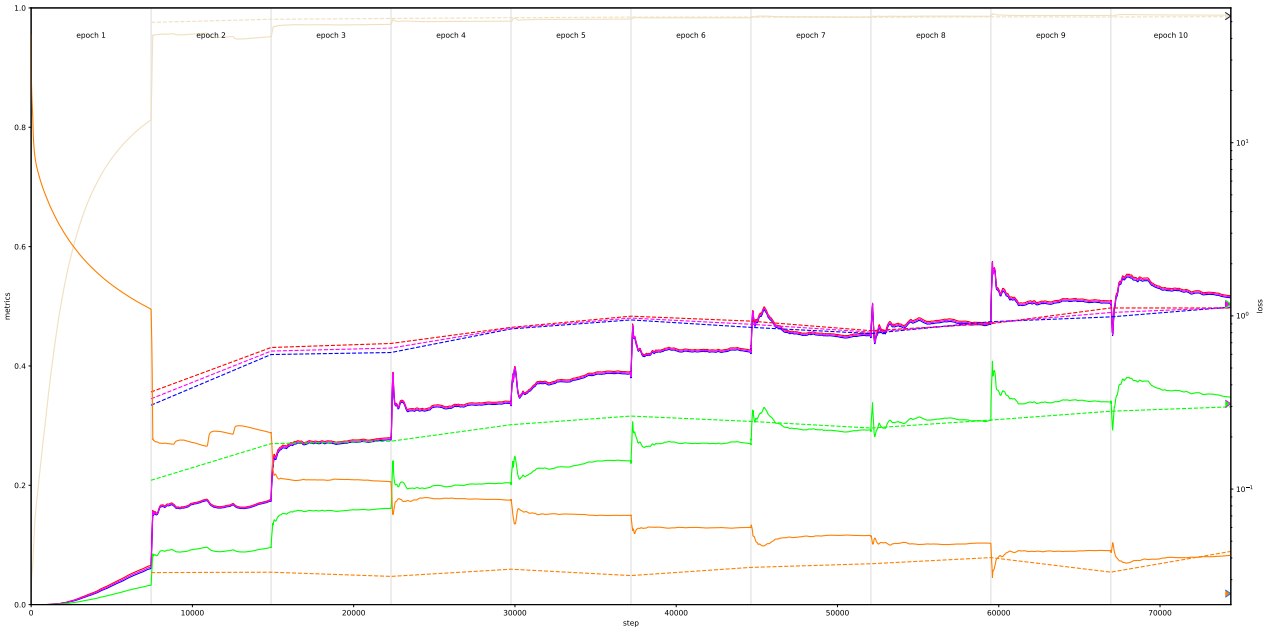
B.2 Delimiters Representation

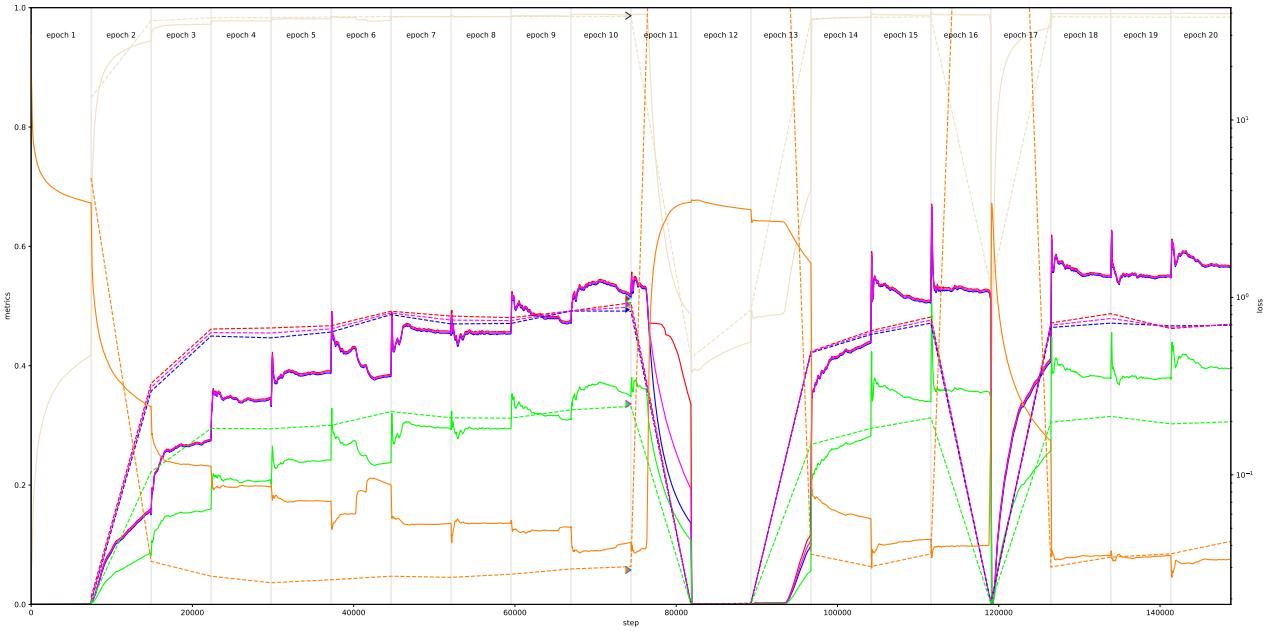


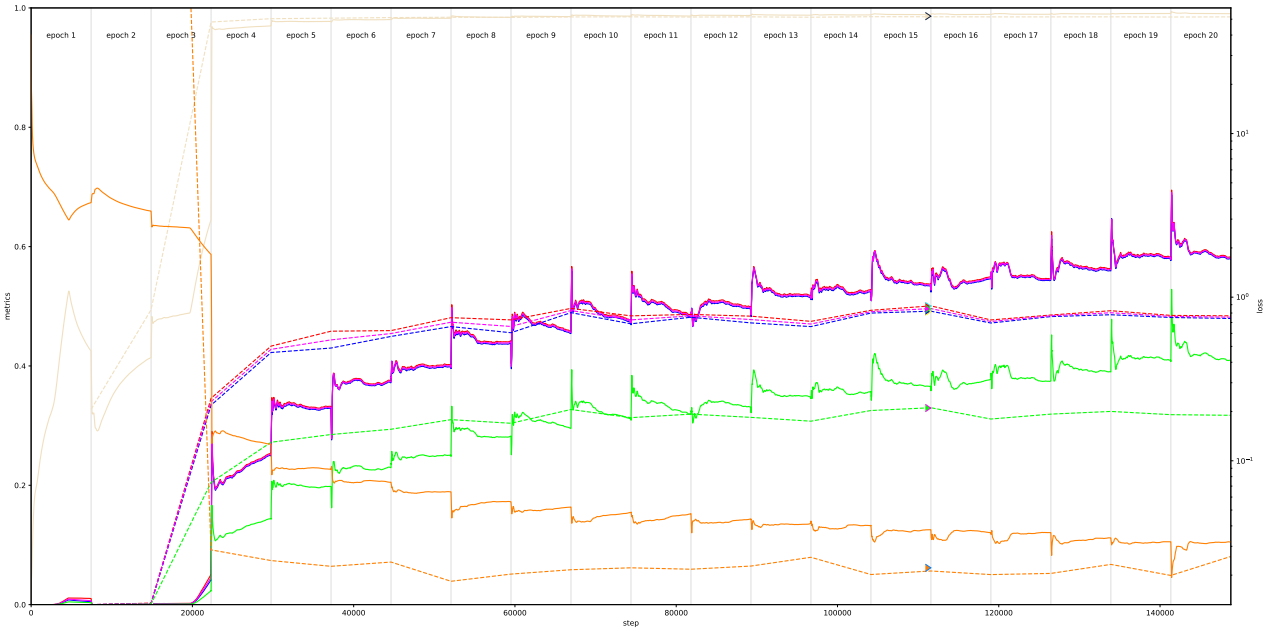



epoch 1
epoch 2
epoch 3
epoch 4
epoch 5
epoch 6
epoch 7
epoch 8
epoch 9
epoch 10
epoch 11
epoch 12
epoch 13
epoch 14
epoch 15
epoch 16
epoch 17
epoch 18
epoch 19
epoch 20
metrics
loss
step


epoch 1
epoch 2
epoch 3
epoch 4
epoch 5
epoch 6
epoch 7
epoch 8
epoch 9
epoch 10
metrics
loss
step


epoch 1
epoch 2
epoch 3
epoch 4
epoch 5
epoch 6
epoch 7
epoch 8
epoch 9
epoch 10
metrics
loss
step


epoch 1
epoch 2
epoch 3
epoch 4
epoch 5
epoch 6
epoch 7
epoch 8
epoch 9
epoch 10
epoch 11
epoch 12
epoch 13
epoch 14
epoch 15
epoch 16
epoch 17
epoch 18
epoch 19
epoch 20
metrics
loss
step


epoch 1
epoch 2
epoch 3
epoch 4
epoch 5
epoch 6
epoch 7
epoch 8
epoch 9
epoch 10
epoch 11
epoch 12
epoch 13
epoch 14
epoch 15
epoch 16
epoch 17
epoch 18
epoch 19
epoch 20
metrics
loss
step
0.0
0.2
0.4
0.6
0.8
1.0
0
20000
40000
60000
80000
100000
120000
140000
10^{1}
10^{0}
10^{-1}

C Glossary

BIO-Encoding BIO (aka. IOB) is a popular tagging scheme for chunking tasks. It uses the tag set $\{B, I, O\}$, which is where it gets its name from.

B denotes that a token is the start of a relevant chunk, I denotes that a token is inside of a relevant chunk, and tokens outside of relevant chunks are tagged with O.

Version 1, introduced in 1995 by Ramshaw and Marcus [32], merely uses B tags for disambiguating consecutive chunks. It was soon supplanted by version 2, which always tags the first token of each chunk with a B, regardless of whether that resolves ambiguity.

Version 2 simplifies the scheme, and improves consistency, by removing unnecessary interdependencies between chunks. The first token of any relevant chunk is always tagged B, irrespective of what precedes it. As a consequence, chunks can be handled independently. Additionally, this consistency also aids in spotting labeling errors. If the model outputs OI, that is always a mistake.

BIO2 does not have drawbacks compared to version 1, which is why it is almost exclusively used today. It is rare to still see version 1, so it is customary to refer to BIO2 simply as "BIO". In this thesis, we exclusively use BIO2.

Example: Let r_k denote relevant chunks, and i_k irrelevant chunks. The following table shows an input sequence and its corresponding encoding in BIO and BIO2, respectively.

input:	c_1	c_2	i_1	c_3	c_3
BIO:	I	B	O	I	I
BIO2:	B	B	O	B	I

15, 17

CoNLL-2003 CoNLL-2003, depending on the context, can refer to either (a) the 2003 Conference on Computational Natural Language Learning [1], or (b) the NER dataset used for the shared task during that conference.

The Conference on Computational Natural Language Learning [17] is a yearly conference on computational linguistics (i.e. NLP). Its organizers regularly propose so-called "shared tasks" to their attendees, who can accept the challenge, submit their results, and compete with each other.

In 2003, the conference was held in Edmonton, Canada at May 31st and June 1st 2003. The shared task during that year was multilingual NER, for which a dataset was provided. Today, that dataset is what is usually referred to by the "CoNLL-2003" moniker, which is sometimes further abbreviated to just "CoNLL". It is still regularly used as a benchmark dataset in the NER literature. 22

Foundation Model An approach to model design and training, using a generic base model trained on a large, unlabeled training set as the foundation. On that foundation, task-specific models are fine-tuned on small, labeled datasets, without adapting the model architecture for each downstream task.

Details of the approach are discussed in section 1.5. 5–7, 11, 24

Groningen Meaning Bank "The Groningen Meaning Bank" [4] is a collection of public domain English texts with semantic and syntactic annotations, developed by Bos et al. at the University of Groningen. Annotations were bootstrapped with NLP tools, and then refined through a combination of domain experts and crowd-sourcing.

Downloads for different versions of the dataset are available from the project's website [3]. 22, 53, I

Homograph A word which is spelled like another word, but has a different meaning.

Example: "lead" (noun) and "lead" (verb) are homographs. 51

Homonym A word which is either a homograph or homophone, or both. 3, 51

Homonymy Existence of homonyms.

Example: "row" (noun, e.g. in a table) and "row" (verb, e.g. a boat) are homonyms, as are "stalk" (pursuing stealthily) and "stalk" (part of a plant). 3, 13

Homophone A word which is pronounced like another word, but has a different meaning.

Example: "sea" and "see" are homophones. 51

Named Entity Disambiguation Synonym of Named Entity Linking (NEL). 51, 53

Named Entity Linking Named Entity Linking, also known as Named Entity Disambiguation (NED), is a follow-up task to NER. It consists of assigning a unique identity to each entity mention.

Example: We extracted the ambiguous entity mention "Olaf Scholz". It is uniquely identified, by assigning it the identity `wikipedia.org/wiki/Olaf_Scholz`. 51, 53

Named Entity Recognition Lying at the intersection of NLP and Information Extraction, Named Entity Recognition is the task of extracting and classifying mentions of named entities from unstructured text.

It is a precursor to other NLP tasks, such as NEL.

Example: The sentence "Hendrik visited Athens in 2021." contains three named entities: 1. person "Hendrik", 2. location "Athens", and 3. time "2021". 7, 51, 53

Polyseme A word with multiple related word senses.

Example: "man" has multiple word senses, including "the entire human species" and "a human male". 3, 51

Polysemy Existence of polysemes.

Example: The sentence "I bank at the local bank." uses two different senses of the polyseme "bank". 3

Text-To-Text Transfer Transformer Novel Transformer architecture, framing every problem in a text-to-text framework. Created at Google Research, based on a review and evaluation of the Transformer-based NLP literature. 11, 53

WikiAnn WikiAnn [29] is a framework for automatically extracting cross-lingual NER datasets from Wikipedia pages, and linking them to existing, external knowledge bases (see NEL). It was developed in 2017 by Pan et al., in order to leverage the high-quality, crowd-sourced semantic information inherent in Wikipedia pages.

Somewhat confusingly, the dataset extracted and published by the authors, using the WikiAnn framework, is also referred to by the same name. 22

Word Sense In linguistics, a word sense is one of the meanings of a word.

Example: "a financial establishment" is one of the word senses of the word "bank", while "the land sloping down to a river or lake" is another. 3, 51

D Acronyms

AI Artificial Intelligence 5

BERT Bidirectional Encoder Representations from Transformers 11, 12

C4 Colossal Clean Crawled Corpus 11, 12, 53

CV Computer Vision 6, 7, 9

DL Deep Learning 2, 5, 10

DPT Domain Preserving Training 12, 26

GMB Groningen Meaning Bank 22, I

GRU Gated Recurrent Unit 10

LM Language Model 3, 10

LSTM Long Short-Term Memory 10, 12

mC4 multilingual C4 12, 24, 37

ML Machine Learning 1, 3, 5, 7, 9, 10, 22, I

MLM Masked Language Modeling 11

NED Named Entity Disambiguation 51

NEL Named Entity Linking 51, 52

NER Named Entity Recognition 7, 8, 13, 14, 21, 22, 26, 27, 29, 30, 34, 36, 50–52, I

NLP Natural Language Processing 1–3, 7, 9–11, 21, 24, 50–52, I

NN Neural Network 5, 9

OOV Out-Of-Vocabulary 2, 10

RNN Recurrent Neural Network 10, 12

T5 Text-To-Text Transfer Transformer 7, 11, 13, 14, 18, 23, 24, 26

TL Transfer Learning 3–5, 7, 9