



Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Master's Thesis

**Few-Shot Learning Methods for Semi-Automated
Annotations of Large Text Corpora**

in the Language Technology (LT) group

by

Jannis Meißner

Matriculation number: 7330085

Field of study: Computer Science

submitted July 17, 2023

Supervisors: Florian Schneider, Tim Fischer

First Reviewer: Prof. Dr. Chris Biemann

Second Reviewer: Dr. Irina Nikishina

Abstract

Text corpora are large collections of written or spoken language and have become widely available since the beginning of the era of digital information processing. Large text corpora find extensive use in both scientific research and the business world, providing valuable insights and enabling advancements in various domains. While they are historically rooted in linguistics, nowadays text corpora play a key role in various kinds of fields, especially since collecting and processing large amounts of text data is easy to do with today's computational resources. In most fields, the true potential of large text corpora is only fully realized when an interpretative value in the form of annotations is added to them. However, annotating text data by hand is often infeasible in terms of time and/or in terms of money. This thesis, therefore, explores the possibilities regarding a semi-automatic annotation system, where the user only annotates a few examples of the corpus by hand in order for the system to learn the annotation semantics of the specific labels and extend the annotations automatically to the whole corpus. In order to address this, as a first step, extensive research in the field of few-shot named entity recognition has been conducted to determine which state-of-the-art models are best suited for such a system. Subsequently, experiments on five state-of-the-art named entity recognition models have been conducted that resemble the *few-shot* setting and the *fine-grained* label characteristics of a semi-automatic annotation system, where the user can define arbitrary label categories beforehand. The results of the experiments confirm that few-shot named entity recognition is a challenging task, even for state-of-the-art models. Additionally, this thesis describes the design and development of a prototype for a semi-automatic annotation system.

Contents

1	Introduction	5
1.1	Motivation	5
1.2	Problem Statement	5
1.3	Approach	6
1.4	Research Questions	6
1.5	Structure of this Thesis	7
2	Background	8
2.1	Deep Learning	8
2.1.1	Neural Networks	8
2.1.2	Supervised & Unsupervised Learning	9
2.1.3	Few-Shot Learning	10
2.1.4	Meta-Learning	10
2.1.5	Deep Learning in Natural Language Processing	11
2.2	Transformer	13
2.3	Transformer-Based Language Models	20
2.4	Named Entity Recognition	23
2.4.1	Few-Shot NER	25
2.4.2	Fine-Grained NER	25
3	Related work	26
3.1	Metric-Based Approaches	26
3.2	Prompt-Based Approaches	27
3.3	Model Architectures	28
3.3.1	ProtoBERT	28
3.3.2	StructShot	29
3.3.3	TemplateNER	31
3.3.4	EntLM	33
3.3.5	Adapter	35
4	Experiment and Dataset	39
4.1	Dataset	39
4.2	Evaluation Metrics	41
4.2.1	Precision	41
4.2.2	Recall	42
4.2.3	F1	42
4.3	Experiments	42
4.3.1	Baseline Experiment	43
4.3.2	ProtoBERT & StructShot	44
4.3.3	TemplateNER	45
4.3.4	EntLM	48
4.3.5	Adapter	50

4.4	Summary	52
4.4.1	ProtoBERT & StructShot	52
4.4.2	TemplateNER	53
4.4.3	EntLM	53
4.4.4	Adapter	53
5	Implementation	55
5.1	Functional Overview	55
5.1.1	Upload a Document	56
5.1.2	Selecting a Model	56
5.1.3	Creating a new Task	57
5.1.4	Selecting Tags	57
5.1.5	Annotation	58
5.1.6	Training	58
5.1.7	Get Training Status	59
5.2	Implementation Details	60
5.2.1	Front End	60
5.2.2	Back End	63
6	Future Work	67
6.1	Multiple Models	67
6.2	Expansion of Possible Annotations	67
6.3	Prompt-Based Few-shot NER	67
7	Conclusion	69
7.1	Research Questions Revisited	69
7.2	Summary and Conclusion	70
	Bibliography	72

1 Introduction

This chapter introduces the topic of this thesis by giving a motivation for the research, formalizing the problem statement, and relating it to the corresponding field of natural language processing. The overall research approach of this thesis is given and three questions to guide this research are defined.

1.1 Motivation

Venkat N. Gudivada (2018) defines a corpus as "a large collection of texts of written or spoken language, stored in a machine-readable format". Corpora enable several useful applications. Being historically rooted in linguistics, they offer insights into how language is used in different situations depending on the corpus origin and time of creation. Especially with the rapid rise of machine learning and data science, such corpora play an important role in science fields other than plain linguistics, as well as in the industry given the fact that the availability of textual data and the resources to process such data aren't obstacles anymore. Companies might be for example interested in transcriptions of meetings, customer e-mails, or analysis of technical logs. To add interpretative value to the unstructured text data, text corpora are often annotated, for example, with labels that indicate the assignment of a certain word to a word class or paragraphs to a certain textual or semantic category. However, annotating text data by hand is often too time-consuming and crowdsourcing is expensive. This thesis, therefore, explores the possibilities regarding a flexible semi-automated annotation system for large text corpora. Such a system takes as input only a few manually-made annotations by the user or a domain expert, tries to find semantically similar spans, and annotates them according to the examples provided by the user.

1.2 Problem Statement

For this thesis, the problem of automatically annotating text is reduced to the well-known but still heavily researched natural language processing problem of *named entity recognition* (NER). The goal of NER is to locate and classify named entities that occur in a text document into pre-defined categories. While early approaches to NER mainly used rule-based systems and heuristics to detect and classify named entities, the advent of machine learning and especially deep learning techniques - the latest and most successful being transformer models - has led to great progress in that research field, especially since named entity recognition is a subtask of other natural language processing problems like information extraction or question answering. However, a semi-automatic annotation system adds a variety of additional challenges to the standard named entity recognition task: First, usually, most NER models are trained on a lot of labeled data beforehand. However, a semi-automatic annotation system is expected to be able to annotate textual data based on only a few annotations that were done manually by the user. The underlying model, therefore, is required to achieve adequate labeling accuracy while only

being provided with a few labeled examples. This problem is commonly referred to as *few-shot learning*.

Furthermore, a semi-automatic annotation system is expected to be able to annotate textual data based on categories defined by the user. While standard NER datasets often differentiate only between coarse-grained categories, for example, "Location", "Person" or "Organization", the labels that the user would want the system to annotate might include more finer-grained, semantically more similar categories which are possibly even technical or domain-specific terms. This task is usually denoted as *fine-grained named entity recognition* and poses an additional challenge to the underlying model.

Lastly, the user of the annotation system will likely often switch annotation tasks (and therefore the labels they want the system to annotate). This requires the system to be able to save and switch between corresponding models more often. The underlying NER model should therefore allow for annotation tasks to be switched quickly and without a large computational or storage-specific overhead.

1.3 Approach

In order to address the above challenges that a semi-automatic annotation system imposes on its underlying models, first, comprehensive research is done on current state-of-the-art named entity recognition systems. This research will focus on systems explicitly developed for the purpose of few-shot and/or fine-grained named entity recognition. For the experiments, currently available datasets will be researched and compared against each other in terms of their suitability regarding the training of a model for a semi-automatic annotation system. Here, a special focus will be placed on the number and the granularity of the labels the dataset provides. After researching suitable models and datasets, experiments are designed that best represent a realistic setting of an annotation system. The designed experiments will be conducted on all chosen models, and their results will be compared to further determine how well different models deal with the various challenges an annotation system imposes on them. Finally, a prototype of a semi-automatic annotation system shall be designed and developed.

1.4 Research Questions

Three research questions can be inferred from the problem statement:

RQ1: What state-of-the-art methods exist for few-shot (fine-grained) named entity recognition?

Current state-of-the-art methods for few-shot and fine-grained NER are to be researched and compared against each other. They shall also be analyzed concerning the requirements of a semi-automatic annotation system.

RQ2: How many support samples are needed to achieve adequate results on an automatic annotation task based on entity classes and how does the number of classes relate to the number of required support samples?

The minimum number of support samples per class that yield adequate results shall be determined empirically.

RQ3: How does the presence of semantically more similar labels affect the overall performance of NER models?

A semi-automatic annotation system will likely deal with many finer-grained labels. Experiments shall determine the ability of NER models to differentiate between labels that are semantically more similar.

1.5 Structure of this Thesis

State-of-the-art natural language processing systems heavily utilize deep learning techniques. In order for the reader to understand the systems presented in this thesis, Chapter 2 introduces and explains the most important technical details. The concept of deep learning is introduced and neural networks and related concepts are explained briefly. Then, the history of deep learning techniques used in the field of natural language processing is outlined and core concepts are explained, starting at recurrent neural networks and sequence-to-sequence models and ending at the currently best and most widely used models - transformer. As transformers play a key role in all analyzed systems of this thesis, core concepts, building blocks, and their functionality are explained in a more detailed manner. Chapter 2 also briefly defines named entity recognition, its history, and related concepts. Chapter 3 first offers an overview of current state-of-the-art few-shot NER systems. It then explains the five systems that were chosen for the experiments in more detail. Chapter 4 then first describes the dataset that was used for the experiments as well as the metrics that were used for the evaluation of the experiments. Then, for each analyzed system, implementation details are described and the results of the experiments are presented. A final section compares all the experiments performed and presents conclusions regarding their suitability in a semi-automatic annotation system. Chapter 5 describes the prototype of such a system that was developed during the course of this thesis. Here, the first section explains the functionality of the system and the second section explains the implementation details of both front and back end. Chapter 6 briefly explains possible future research. Finally, Chapter 7 offers an overall conclusion, summarizing the findings of this thesis.

2 Background

This chapter aims to introduce fundamental background knowledge in order for the reader to understand the different technologies and model architectures on which the experiments were conducted. Section 1 will give an overview of deep learning in general, its recent advancements, and its relevance in the context of natural language processing (NLP). Building on Section 1, Section 2 will introduce transformer, which played a big role in the recent success of NLP applications, such as Chat-GPT¹ and others and will also be used for the experiments of this thesis. Section 2 first explains the general transformer architecture and then explains the two transformer models BERT and BART in more detail. Section 3 will explain some fundamental terms of natural language processing, that are relevant for this thesis.

2.1 Deep Learning

In recent years, deep learning has emerged as a powerful subfield of machine learning and has revolutionized many scientific fields. This section aims to provide a brief overview of what deep learning is and how deep learning changed the way we can approach problems in the field of natural language processing. Unlike traditional machine learning methods that rely on handcrafted features, deep learning algorithms have the capacity to automatically learn and extract patterns and representations from the data itself, thus shifting the focus from carefully designing features and algorithms to using large amounts of data and computational resources while letting the algorithm itself build up useful representations of the data.

2.1.1 Neural Networks

At the core of all deep learning approaches are neural networks, which are briefly explained in the following. A neural network is a computational model which is inspired by the structure and functioning of the human brain. Figure 2.1 shows the basic structure of an artificial neuron. It is composed of interconnected modules called *artificial neurons*. In its most basic form, artificial neurons are parametrized by learnable weights w_1, \dots, w_n , receive input signals x_1, \dots, x_n , compute a weighted sum $x_1 w_1 + x_2 w_2, \dots, + x_n w_n$, and generate output y by passing the weighted sum through an activation function f . Typically, a scalar, learnable *bias* term b is also added to the weighted sum before passing it through the activation function, allowing the model to shift the result of the activation function regardless of the input. Formally, the output y_k of a given neuron with n input signals x_j and weights w_{kj} can be computed as:

$$y_k = \phi\left(\sum_{j=0}^n w_{kj}x_j + b\right). \quad (2.1)$$

1. <https://chat.openai.com/>

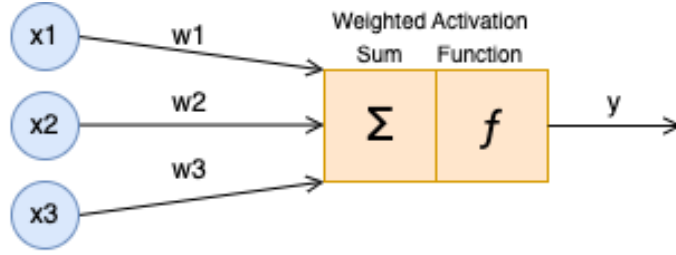


Figure 2.1: A simple artificial neuron with inputs x_1, x_2, x_3

Neural networks typically consist of millions or even billions of artificial neurons that are arranged in layers. The input for each artificial neuron in all layers but the first are the outputs of all neurons of the previous layers. The goal of the network is now for an input vector x to output a desired output vector y . The network learns by adjusting its weight matrix W in a process called backpropagation: A loss function \mathcal{L} computes the discrepancy between the desired output y' and the output of the network y . The choice of \mathcal{L} is task-dependent. The goal is to minimize the discrepancy between y and y' as much as possible. To achieve this, the gradient - the vector that indicates the direction of the steepest ascent - of the loss function with respect to the weights is calculated and each weight is updated by a negative fraction of it. For classification tasks, usually, the output y of the last layer needs to be interpreted as a probability distribution, where each value y_i represents the probability that input x belongs to class C_i . This is achieved by the *softmax* function (Equation 2.2), which takes a real-numbered input vector z and normalizes it such that its values sum up to 1 and therefore allows for it to be interpreted as a probability distribution.

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^n e^{z_j}}. \quad (2.2)$$

A typical loss function for classification tasks is for example the cross-entropy loss, which measures the similarity between a predicted distribution $q(x)$ and a true distribution $p(x)$:

$$H(p, q) = - \sum_x p(x) \log(q(x)). \quad (2.3)$$

2.1.2 Supervised & Unsupervised Learning

Two commonly occurring categories of learning algorithms are *supervised* learning and *unsupervised* learning. Supervised learning, in general, deals with n training samples $(x_1, y_1), \dots, (x_n, y_n)$ where x_i represents the input vector of the i th training example and y_i the respective output vector. The assumption in supervised learning is that the training samples are produced by a function $f: X \rightarrow Y$, and the training objective of the learning algorithm is now to approximate this function as closely as possible, with the goal of it being able to generalize this knowledge to instances (x_j, y_j) not seen during training. Unsupervised learning, in contrast, denotes learning algorithms that learn from unlabeled data. The main goal of unsupervised learning is to learn representations of the input data. Unsupervised algorithms can, for example, be used for classification (clustering approaches, where the model learns cluster centroids from the input data) or data exploration and outlier detection. In the world of natural language processing, often *unsupervised pre-training* is mentioned in the context of large transformer models. Such models are pre-trained by feeding them large amounts of unlabeled text data, corrupting a portion of

it, and giving the model the task to repair the corrupted input in some way. While this is not exactly unsupervised learning in the classic sense (because in practice, there are labels, which are created automatically from the language data during training), it certainly relates to it, since the data itself is unlabeled and the model learns structure in this unlabeled data.

2.1.3 Few-Shot Learning

Usually, machine learning systems and especially deep learning systems are trained on large amounts of training data. This need for training data, however, is a non-trivial hurdle to overcome. First, labeled training data is expensive. To achieve a high-quality data standard, data labeling needs to be done manually by domain experts, who are expensive. Furthermore, annotating large amounts of data can take a long time. Secondly, even large amounts of unlabeled data might not be available in every case, for example in very specific or new domains. *Few-shot learning* is a branch of machine learning that explicitly deals with the scenario where only a small amount of training data is available and the model needs to make predictions based on this small number of samples. A few-shot classification task is usually described by two variables N and K : In an N -way K -shot classification task, the model needs to learn to classify N classes while being provided K training samples per class.

2.1.4 Meta-Learning

Meta-learning is a commonly used approach to learning in a few-shot setting. Meta-learning approaches train the model on episodes, where each episode resembles one few-shot learning task. Formally, in an episodic N -way K -shot learning setting, N is the number of classes the model needs to differentiate between and K is the number of examples that are given per class. Now, for each training episode, N classes and K examples per class are randomly sampled in order to build a support set $\mathcal{S}_{\text{train}} = \{\mathbf{x}^{(i)}, \mathbf{y}^{(i)}\}_{i=1}^{N \cdot K}$ and K' examples for every of the N classes are sampled in order to construct a query set $\mathcal{Q}_{\text{train}} = \{\mathbf{x}^{(i)}, \mathbf{y}^{(i)}\}_{i=1}^{N \cdot K'}$ and $\mathcal{S} \cap \mathcal{Q} = \emptyset$ (Ding et al., 2021). The system is then trained by predicting labels of the query set $\mathcal{Q}_{\text{train}}$ with the information of $\mathcal{S}_{\text{train}}$. The more different tasks (episodes) the model sees, the more it "learns to learn" better from each episode. Thus during inference, it has learned to gather information from \mathcal{S} to correctly classify the samples in \mathcal{Q} on the basis of the support set \mathcal{S} . Generic meta-learning can be described as the following:

$$\theta^* = \arg \min_{\theta} \sum_{i=1}^n \mathcal{L}(\phi_i, \mathcal{Q}_i) \quad (2.4)$$

(Levine, 2021), where n is the number of episodes and $\phi_i = f_{\theta}(\mathcal{S}_i)$ is a learned representation that can be used to classify the specific samples in \mathcal{Q} . This abstract concept can be concretely implemented in different ways. Snell et al. (2017), for example, do meta-learning by building up representations of the samples in \mathcal{S} , and classify examples in \mathcal{Q} based on the distance to each class representation (this approach will be explained further in Chapter 3). Other approaches utilize *gradient-based meta-learning*, where each task in itself involves a gradient step on the parameters of the model such that for each task \mathcal{T}_i , the model's parameters θ become θ_i (Finn et al., 2017).

2.1.5 Deep Learning in Natural Language Processing

Deep learning had a transformative impact on many scientific fields, including natural language processing. While traditional NLP methods relied heavily on handcrafted features and rule-based approaches, deep learning allows models to learn directly from raw text data, omitting the often complex feature and rule engineering part of traditional methods. Over the years, a lot of neural models have been employed in the field of NLP.

Recurrent Neural Networks

One neural architecture that had a big impact on NLP are Recurrent Neural Networks (RNNs). RNNs, as described in Elman, 1990, are, at their essence, layers of neurons, in which the output of one neuron partially depends on its previous output, thus representing a mechanism of capturing relationships between sequential pieces of input. As natural language is an inherently sequential construct, RNNs naturally are a good fit for all kinds of language-related computational tasks. Given a sequence $\mathbf{x} = x_1, x_2, \dots, x_n$, at timestep t , RNNs compute a hidden state h_t based on the previous hidden state h_{t-1} and the current input x_t . At each timestep, also an output is generated that is based on h_t :

$$h_t = \sigma(W_2 h_{t-1} + W_1 x_t + b_1) \quad (2.5)$$

and

$$y_t = W_3 h_t + b_2, \quad (2.6)$$

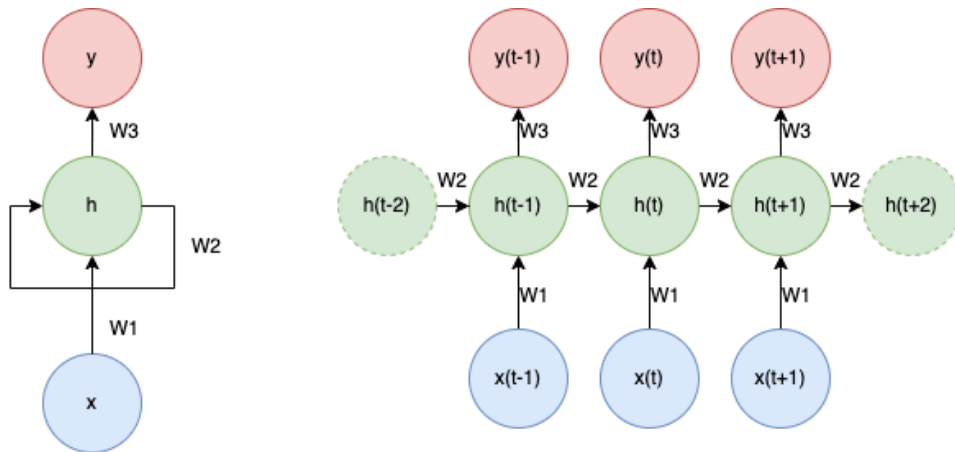


Figure 2.2: Schematic illustration of an RNN (left) and how it unfolds over time (right)

where σ is an activation function, b_1 and b_2 are bias vectors, and W_1, W_2, W_3 are weight matrices for input to hidden state, hidden state to hidden state, and hidden state to output, respectively. Note that the three weight matrices are shared across all timesteps t_n . This process can be seen in Figure 2.2. RNNs learn by utilizing backpropagation-through-time (BPTT), a variant of the standard backpropagation algorithm, where the gradient is computed after the RNN has been unfolded into a series of connected networks, one for each step of the input. This, however, leads to two problems known as the *vanishing gradient problem* and the *exploding gradient problem*. The vanishing gradient problem describes a scenario during backpropagation, in which gradients become extremely small, due to the large amount of layers they are propagated through. Its

inverse, the exploding gradient problem, occurs when gradients become exponentially large during backpropagation-through-time. This complicates the learning process immensely. More sophisticated network architectures like LSTM (Long-Short-Term Memory, Hochreiter and Schmidhuber (1997)) and GRU (Gated Recurrent Unit, Cho et al. (2014)) have been introduced to enhance RNNs, minimizing the vanishing (and exploding) gradient problem and improving their capability to capture long-term dependencies. Furthermore, Schuster and Paliwal (1997) introduced *bidirectional RNNs*, enabling RNNs to also take into account future timesteps x_{t+i} . This improves the contextual understanding of RNNs further since tokens ahead of the current timestep t often also carry important semantic information.

Sequence-to-Sequence Models

It is also notable that, depending on the application, not all outputs y_i of the RNN are needed. A sentiment classification task, for example, will only need the last hidden state of the RNN to classify the sequence. Other tasks, like named entity recognition, need all intermediate outputs. More sophisticated NLP tasks like machine translation or text summarization, however, involve the generation of output sequences using input sequences where the length of the sequences often doesn't match. To gain more flexibility with regard to the sequence structure, Sutskever et al. (2014) suggested *sequence-to-sequence* (seq2seq) models. Seq2seq models consist of an encoder and a decoder component, both of which typically are RNNs with LSTM cells. The encoder component takes the input sequence and produces a fixed-length vector, capturing semantic information of the input sequence. That fixed-length vector is fed into the decoder network, generating the desired output sequence (cf. Figure 2.3). During training, both models are trained jointly on pairs of (X, Y) , where X is the input sequence and Y is the desired output sequence. In the decoder, instead of building the sequence on its own predicted words y'_i (as is the case during inference), at each timestep t , the decoder is provided the corresponding desired word y_{t-1} as its input. This approach allows for varying input and output sequence lengths and also improves the flexibility of the overall model as the encoder and decoder are, although jointly trained, different models and can therefore be subject to different architectural design choices.

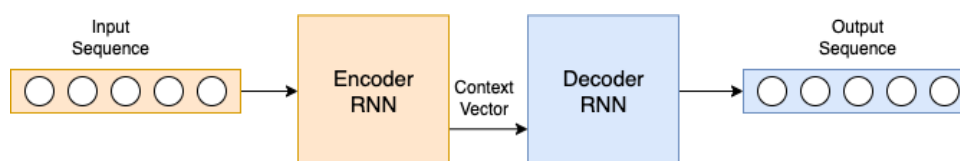


Figure 2.3: Seq2Seq Model

Attention

Although seq2seq models solved a lot of problems of vanilla RNNs, they still exhibit problems with increasing sequence length. According to Bahdanau et al. (2014), the reason for this is that the fixed-length context vector that is produced by the encoder poses a bottleneck that becomes increasingly more significant the longer the input sequence is. Bahdanau et al. (2014) proposed a solution to eliminating this bottleneck by allowing the model to glance back at the input sequence at each decoding step. This technique is called *attention*. Using attention mechanisms, each decoder output depends not only on the last decoder state but on a weighted combination of all the input states as well. Thus, a sequence-to-sequence model with attention

is capable of quantifying the relevancy of each input symbol x_i to the current output y_t . This is achieved in the following way: Instead of one context vector c being handed over to the decoder, a distinct context vector c_i is computed at each time step i . Each c_i is a weighted combination of vectors h_i called *annotations* (Bahdanau et al., 2014) and each annotation h_k summarizes the input sequence until timestep k . Due to the property of RNNs to represent the more recent inputs better than inputs longer ago, each annotation automatically focuses on the more recent inputs. To also capture future inputs, such that each annotation h_k represents the input sequence around timestep k (shortly before and shortly after), Bahdanau et al. (2014) uses a Bidirectional RNN. Let $(\vec{h}_1, \dots, \vec{h}_t)$ be the sequence of forward hidden states and $(\overleftarrow{h}_1, \dots, \overleftarrow{h}_t)$ the sequence of backward hidden states produced by the Bidirectional RNN. Now, for each input symbol x_k , the corresponding annotation is obtained by concatenating the forward hidden state \vec{h}_k and the backward hidden state \overleftarrow{h}_k such that $h_k = [\vec{h}_k; \overleftarrow{h}_k]$. The context vector at each time step c_i is computed like this:

$$c_i = \sum_{j=1}^n \alpha_{ij} h_j. \quad (2.7)$$

Here, the α are the weights of each annotation which are computed using an alignment model e_{ij} , describing how well the inputs around position j and output at position i match:

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^n \exp(e_{ik})} \quad (2.8)$$

$e_{ij} = f(s_{i-1}, h_j)$ is calculated based on the decoder hidden state s_{i-1} and the j -th annotation h_j of the input sentence. To calculate e_{ij} , a feed-forward neural network is used that is jointly trained with the rest of the model (Bahdanau et al., 2014). A larger alignment e_{ij} means that s_{i-1} and h_j match well. This will lead to a larger α which will in turn lead to the corresponding annotation being of more importance for the decoder when generating c_i and therefore y_t . This alignment model ensures that the encoder does not need to capture the whole input sequence in a single vector. Instead, the decoder learns to *attend* to certain parts of the input that are more important for the generation of the output sequence. Figure 2.4 illustrates this process.

2.2 Transformer

While the attention mechanism by Bahdanau et al. (2014) improved seq2seq models by enabling the decoder to attend to certain parts of the input, it still brought some limitations because of its sequential processing of input tokens. The biggest limitation is that it precludes parallel processing within training examples (Vaswani et al., 2017) leading to inefficiencies with longer input sequences. With Vaswani et al. (2017) a groundbreaking approach to sequence-to-sequence tasks was introduced: Transformer. Unlike traditional seq2seq models which rely on recurrent connections to capture sequential dependencies, transformer employ a so-called *self-attention* mechanism that allows them to capture global dependencies in an input sequence without sequentially processing the input. Transformer models quickly established themselves as state-of-the-art for many NLP tasks including machine translation, question answering, sentiment analysis, speech recognition, or named entity recognition. Figure 2.5 shows the architecture of transformer as described in Vaswani et al. (2017). On a high level, a transformer as suggested by Vaswani et al. (2017) consists of 6 encoder components stacked on top of each other and 6 decoder components stacked on top of each other. Each encoder component has two sub-layers:

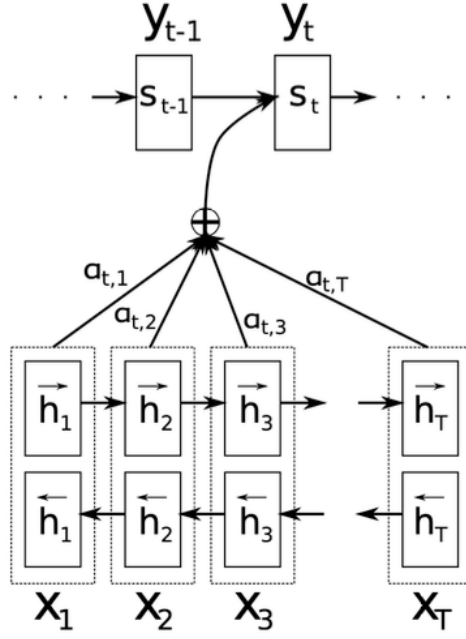


Figure 2.4: Illustration of how Bahdanau et al. (2014) utilize annotation vectors h_i to generate context vectors c_i , Source: Bahdanau et al., 2014

One *multi-head* self-attention layer and one fully connected feed-forward network. The decoder component also has a multi-head self-attention layer and a feed-forward layer. To attend to parts of the encoder, the decoder also has another multi-head attention layer that receives the input from the encoder. For training stability, residual connections are added between sub-layers, and each sub-layer is followed by a normalization layer both in the encoder and the decoder.

In the following, the core components of transformer models are explained in more detail.

Scaled Dot-Product Attention

Attention is a general concept that allows a system to focus on specific elements of the input. However, the specific implementation of attention used in Vaswani et al. (2017) is slightly different than the one from Bahdanau et al. (2014) that was described above. Vaswani et al. (2017) use a concept that is called *dot-product attention*, in contrast to Bahdanau et al. (2014) who use *additive attention*. Let X be an input sequence given to the first layer of the transformer encoder, the *embedding layer* (as seen in Figure 2.5). The embedding layer transforms each token x_i into a learned embedding v_i of dimension d_k . By multiplying the input embedding matrix with each of the three trainable weight matrices W_Q, W_V, W_K , the three matrices Q, K, V are computed. The attention itself is now computed as follows:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V. \quad (2.9)$$

This equation is illustrated in Figure 2.6. Each entry $(QK^T)_{ij}$ represents the dot-product similarity of query vector q_i with key vector k_j (effectively quantizing, how much input i and input j relate to each other). QK^T is now scaled with $\sqrt{d_k}$. According to the authors, this

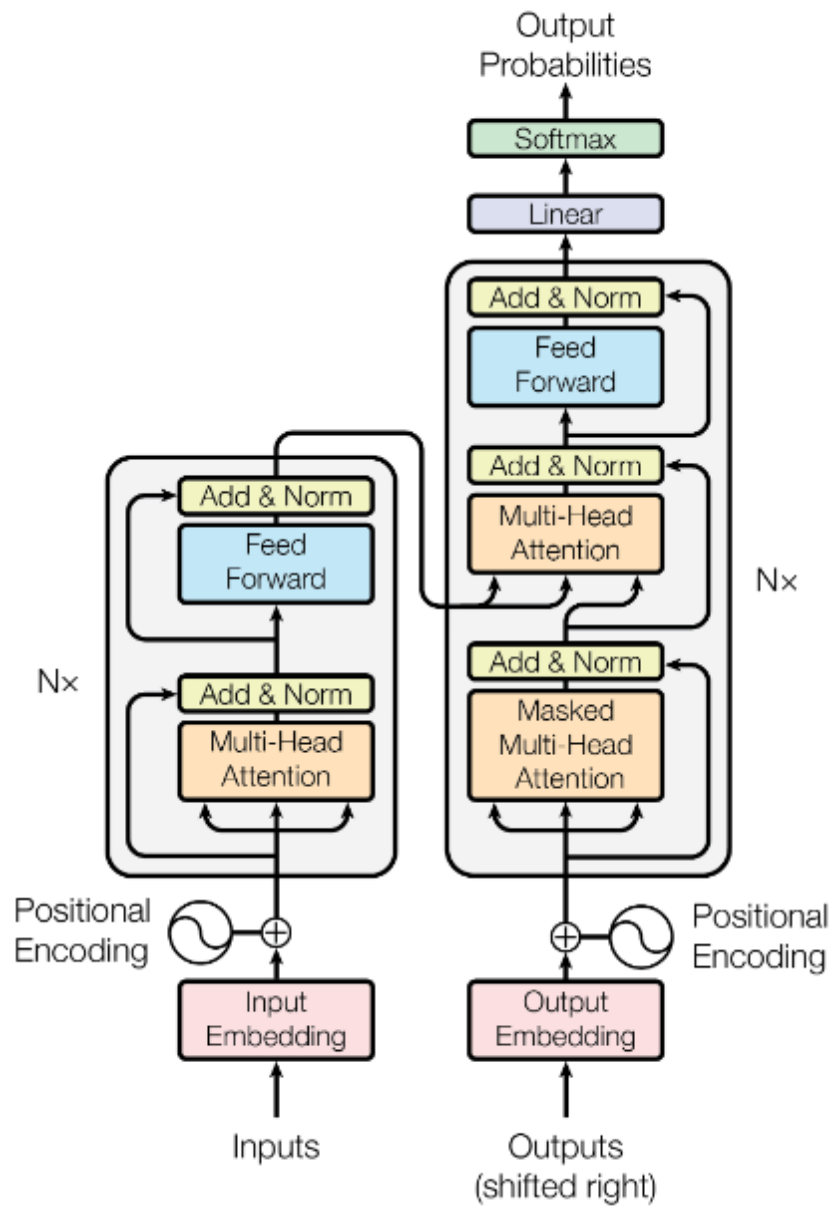


Figure 2.5: The general architecture of transformer as described in the original publication (Source: Vaswani et al. (2017))

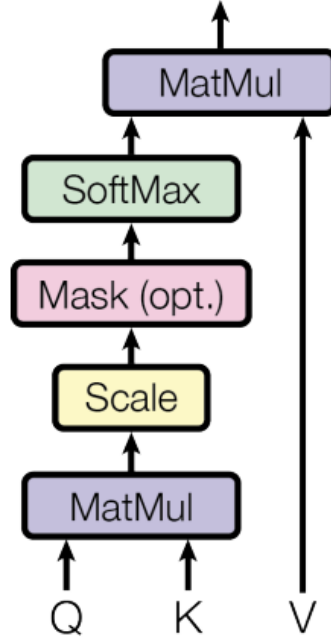


Figure 2.6: Illustration of the scaled dot-product attention (Source: Vaswani et al. (2017))

down-scaling ensures that the magnitude of the softmax does not become too large, which in turn would lead to very small gradients (Vaswani et al., 2017). $\text{softmax}(\frac{QK^T}{\sqrt{d_k}})$ in the equation is used to squash the similarity values between 0 and 1, effectively producing a pseudo-probability over values QK^T , which are then used as weights. This *compatibility weight matrix* is now multiplied with the value matrix V . This matrix multiplication produces a sum of the values v_i that is weighted by the entries of QK^T . The output y_i of the attention layer then effectively represents a summary of how each token in the input relates to token v_i , because more important tokens have a higher score in QK^T and thus influence the weighted sum more substantially. It is important to note that in the encoder, in the first layer the v_i are the input embeddings and in subsequent layers the v_i are the outputs of the previous attention layer. This concept is often called *self-attention* because instead of the classic seq2seq attention, where the attention mechanism was only built into the decoder to attend to certain encoder steps during decoding, here, the attention is built also into the learning process of the encoder itself, allowing the model to weigh the significance of different input tokens against each other and adjusting their influence on the next layer.

Multi-head attention

Vaswani et al. (2017) suggest not only using one attention mechanism, but multiple attention "heads" in parallel, which allows the model to simultaneously attend to different parts of the sequence, providing it with an even richer capacity for modeling complex relationships between tokens. This concept is called *multi-head attention* and it is computed as follows:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_n)W_O. \quad (2.10)$$

Vaswani et al. (2017) set the number of heads h to 8, so instead of having only weight matrices W_Q, W_K, W_V per attention layer, one attention layer now has $W_{Q_1}, W_{K_1}, W_{V_1}$ to $W_{Q_8}, W_{K_8}, W_{V_8}$. Let the input dimensionality be d_{model} . The dimensionality of the weight matrices is set to $\mathbb{R}^{d_{\text{model}} \times d_k}$

for W_{Q_i} and W_{K_i} and $\mathbb{R}^{d_{\text{model}} \times d_v}$ for W_{V_i} , where $d_k = d_v = d_{\text{model}}/h$. The authors used $d_{\text{model}} = 512$. The reduced dimensionality of Q, K, V leads to the computational cost being similar to the cost of one single attention head with full dimensionality d_{model} . After the scaled dot-product of each attention head is computed, the h results of each head are concatenated, resulting in the original output dimension of d_{model} . After concatenating, the result is once again projected by a linear layer W_O , delivering the final output of the multi-head attention layer. Figure 2.7 illustrates the concept of multi-head attention.

Positional Encodings

The order of input is essential for language problems, as the same word in different positions can change semantics drastically. Recurrence in standard seq2seq models - as described in this chapter - provides a natural way for the model to retain input order, since the input is processed token by token. In contrast to this, transformer models perform their computations on the whole input sequence at once and therefore require a mechanism to receive information about the input order. Vaswani et al. (2017) solve this issue by providing the initial word embeddings with *positional encodings* that are added on top of them. The positional encodings are calculated as follows:

$$PE_{(pos, 2i)} = \sin(pos/10000^{2i/d_{\text{model}}}) \quad (2.11)$$

and

$$PE_{(pos, 2i+1)} = \cos(pos/10000^{2i/d_{\text{model}}}). \quad (2.12)$$

In this equation, pos is the position of the token and i is the specific dimension index of the embedding. Intuitively, positional encodings shift word embeddings slightly into groups that correspond to their respective position, allowing the model to incorporate positional aspects into its computation. The choice of sinusoidal functions with large wavelengths as positional encodings offers the advantage that the values of the encodings are between 0 and 1, not changing the word embedding too much. Furthermore, the authors hypothesize, that this representation of positional information will enable the model to attend to relative positions, since for every k , PE_{pos+k} can be represented by a linear transformation of PE_{pos} . Other approaches for injecting positional information are for example found in Gehring et al. (2017), where during training positional embeddings are trained alongside the rest of the model. However, Vaswani et al. (2017) found that fixed positional encodings do not influence the performance of the model negatively compared to learned positional encodings, while they, due to their periodic nature, also allow the model to generalize over possible longer sequence lengths than seen during training.

Position-wise Feed-Forward Layer

The position-wise feed-forward layer is another building block of transformer components and is placed after the multi-head attention layer. It is simply a fully connected layer that applies two linear transformations to the output of the previous layer. After the first transformation, a ReLU activation function is computed. The computation of the position-wise feed-forward layer is as follows:

$$FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2. \quad (2.13)$$

Here, $\max(0, xW_1 + b_1)$ stands for the ReLU function $f(x) = \max(0, x)$ being applied to the first feed-forward layer. It can be seen that the same weight matrices W_1, W_2 are applied along all

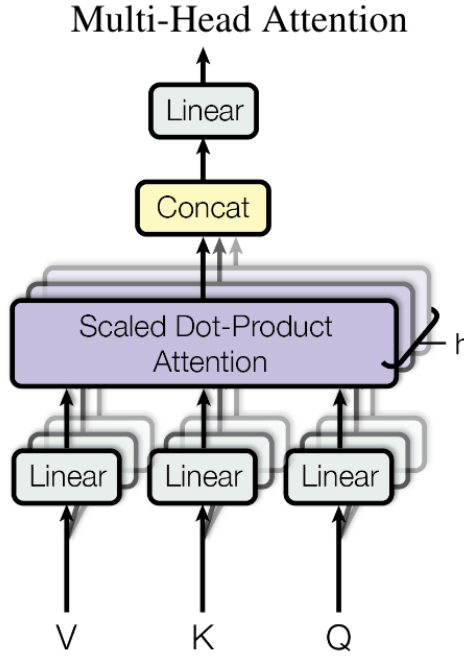


Figure 2.7: Multi-head attention. Q, K, V are projected in their smaller subspaces and each scaled dot-product attention is computed independently. After, the results are concatenated and once again projected (Source: Vaswani et al. (2017))

positions (hence position-wise). In the original paper, internally, the layer has a dimensionality of 2048 ($4 \cdot d_{\text{model}}$), so W_1 has dimensionality 512×2048 and W_2 has dimensionality 2048×512 .

Add & Norm Layer

In Figure 2.5 it can be seen, that after each multi-head attention layer, as well as after each feed-forward layer there are Add & Norm layers. Those layers are related to two concepts of deep neural networks, *layer normalization* and *residual connections*. Generally, normalization in deep learning has the purpose to scale input data to a more practical range, usually centered around zero, in order to stabilize the training of the network during gradient descent. While multiple approaches to normalization exist, two of the most popular are batch normalization (Ioffe and Szegedy, 2015) and layer normalization (Ba et al., 2016). The main difference between batch normalization and layer normalization is that batch normalization performs normalization over each batch dimension, i.e. given batch $x = \{x_1, \dots, x_n\}$ each $x_{(k)_i}$ of batch dimension k is normalized using the associated per-dimension mean and variance. Layer normalization, on the other hand, is calculated per sample, i.e. each sample x_i is normalized by using the associated in-sample mean and variance. It is notable that the actual calculation process of normalization is more nuanced and is explained in more detail in the corresponding publications Ba et al. (2016) and Ioffe and Szegedy (2015), respectively. In their implementation, Vaswani et al. (2017) use layer normalization.

Before the normalizing step, however, residual connections are added to the output of the corresponding transformer sublayer. Residual connections were first introduced by He et al. (2016). They discovered a *degradation* problem, where deep networks, at a certain number of layers, were outperformed by their shallower counterparts, even though in theory, the deep

networks would only need to learn an identity function in layers that are not present in their shallower counterpart. To counteract this degradation problem, they introduced a mechanism in which identity skip connections are introduced to the network and empirically showed that residual connections improve the performance of deep neural networks. Let x be the input of a layer \mathcal{F} in a deep neural network. The output of \mathcal{F} , $\mathcal{F}(x)$ is then added to x , representing the new output that flows through subsequent layers. This process is depicted in Figure 2.8. The common hypothesis as to why residual connections improve training stability in deep neural networks is that they, during training, represent shortcuts for the gradient to flow through, thus minimizing the vanishing gradient problem. They also ensure that deep networks do not suffer from the degradation problem, as learning the identity function for sublayers with residual connections can now be done by setting all outputs of \mathcal{F} to zero which is easier for the network than approximating it through weight combinations.

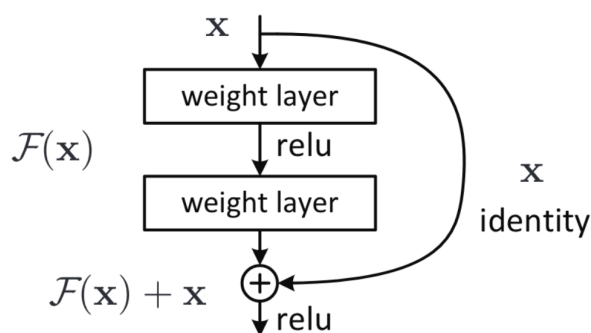


Figure 2.8: The residual connection. The input x flows through the layer \mathcal{F} and is additionally passed around it and added to its output afterwards (Source: He et al. (2016))

Decoder

The encoder and decoder components in transformer, as described in Vaswani et al. (2017) have a lot in common, as can be seen in Figure 2.5: Both use multi-head attention layers followed by position-wise feed forwards layers, where after each of them, the sum of the residual connection and the output is normalized and fed into the next layer. However, there are some differences between the encoder and the decoder. Like in classic seq2seq processing, the encoder is in charge of finding good vector representations of the input tokens (using multi-head attention as described earlier) and handing them to the decoder. The decoder, in turn, takes these representations and is in charge of generating an output sequence with them. For this, the decoder needs its previous output at every step (as can be seen in Figure 2.5). Formally, at every step t , the decoder computes $p_{\theta_{dec}}(y_t | y_{0:t-1}, x_{1:n})$, where y_t is the output of the decoder at the t -th step and $x_{1:n}$ is the representation of the input sequence as computed by the encoder component. For this, the decoder has two types of multi-head attention layers. The first one processes its previous outputs in a self-attention manner, similar to the encoder. The output of this attention layer then represents the V of the next attention layer. However, the Q and K of the next attention layer are computed from the output of the last encoder layer. This step is therefore commonly not called self-attention, but *encoder-decoder attention* because in this step, essentially, the decoder attends to its previous outputs given the encoder representations. The final linear layer and the softmax layer in the decoder component are used to actually transform the output of the last decoder in the decoder stack into words. For this, the linear

layer projects the last decoder output into a $|\mathcal{V}|$ -dimensional space, where $|\mathcal{V}|$ is the size of the vocabulary and by taking the softmax over the output of the linear layer, its output can be interpreted as a probability distribution over all words known by the transformer. As explained, the transformer computes sequences in a parallel manner in contrast to standard encoder-decoder seq2seq models, which compute the output in a sequential manner. Because of this, a special mechanism called *masked attention* is necessary. Masked self-attention, essentially, simulates sequential processing in the first decoder layer, where the decoder only has access to its past predicted tokens $y_{i-j}, j \leq 1$. Because the process, in reality, happens in parallel, future tokens are masked, essentially putting them to $-\text{inf}$ such that the softmax of such future tokens will always be zero, hence effectively preventing the decoder from deriving useful information from future tokens. Figure 2.9 illustrates the concept of *masked attention*.

	<start>	Evan	buys	a	car	<end>
t=1	Evan	0.2	-inf	-inf	-inf	-inf
t=2	buys	0.4	0.1	-inf	-inf	-inf
t=3	a	0.01	0.02	0.1	-inf	-inf
t=4	car	0.4	0.6	0.02	0.2	-inf
t=5	<end>	0.4	0.5	0.001	0.4	0.2

Figure 2.9: Illustrated concept of masked attention. At timestep t , the decoder only has access to words at timestep t or before. Other values are set to $-\text{inf}$, which leads to them being zeros after the softmax layer.

2.3 Transformer-Based Language Models

The section above explained all core components of transformer, as they were introduced by Vaswani et al. (2017). While the core principles, mainly parallel computation of multi-head attention followed by position-wise feed-forward layers, are fundamental across all actual transformer implementations, different concrete transformer implementations have arisen over the years. While the original transformer was developed with the task of machine translation in mind, other approaches use the parallel power of self-attention to construct powerful language models which offer a general language understanding and which can be fine-tuned to all kinds of NLP-related tasks. The following section explains two transformer implementations, that have been very impactful in the world of natural language processing, BERT (Devlin et al., 2019) and BART (Lewis et al., 2020) in more detail.

BERT

BERT stands for **B**idirectional **E**ncoder **R**epresentations from **T**ransformers and has been introduced by Devlin et al., 2019. At that time, it achieved state-of-the-art results on many common NLP tasks such as sentiment analysis, text prediction, or named entity recognition. In contrast to the original transformer that used a decoder component for generating an output sequence, BERT only makes use of the transformer encoder components, stacked up to 12 (for the *bert-base* model), or 24 (for the *bert-large* model) times. BERT introduces two pre-training objectives, the masked language model objective, and the next sentence prediction objective. The goal of those pre-training objectives is to teach BERT a general language understanding by training it on large amounts of unlabeled unilingual text data. More specifically, the authors used the English Wikipedia (~2.500 M words) and the BooksCorpus (Yukun Zhu et al., 2015) to pre-train the BERT model on the two pre-training tasks.

Pre-training task *Masked Language Modelling* (MLM): In order to achieve a bidirectional representation of words (i.e each word representation is influenced by left and right context), the authors corrupt original sentences by choosing a random token at the i th position at 15% probability and replace that token with either a [MASK] token (80% of the time), a random token (10% of the time) or they leave the token unchanged (10% of the time). The task for the model is now to predict the correct token of the sentence. Not always actually using the [MASK] token for masked words but instead also using random words and the actual word at masked positions reduces the mismatch between pre-training and fine-tuning, because usually in fine-tuning, no [MASK] tokens are observed. The MLM objective effectively leads to the model being trained in a bidirectional fashion because for every [MASK] token it predicts its context both from the left and the right side of the sequence available. While masking 15% of all tokens yields very good results on a lot of tasks, a recent study showed that this number is not optimal in all cases and that other masking ratios and more sophisticated masking strategies can improve the model performance further (Wettig et al., 2023).

The second pre-training task is *Next Sentence Prediction* (NSP). In order for BERT to better learn relationships between sentences and not just between words, BERT is also trained on a binary classification task where it is given two sentences A and B with the task to determine whether or not these sentences follow each other. Here, 50% of the time, this holds true, and 50% of the time B is just a random sentence of the corpus.

After pre-training the model to equip it with a strong general language understanding, it can now be trained in a supervised manner on various downstream tasks, utilizing the language understanding it gained in the pre-training stage. This concept of training a previously trained language model on a specific task is called *fine-tuning*. To fine-tune a BERT model on any language-related downstream task, a task-specific layer is simply added to the last encoder of the BERT model and either trained together with the whole model or trained alone while the parameters of the pre-trained model are frozen. The most important upside of this approach is that the pre-trained model can be shared (for example on platforms like huggingface²) and only the fine-tuning for any downstream tasks needs to be done, saving time (BERT's pre-training time was around 4 days (Devlin et al., 2019), while most often the fine-tuning process doesn't take up more than an hour), money (fine-tuning can be done on conventional GPUs rather quickly) and minimizing the large carbon footprint that pre-training causes.

2. <https://huggingface.co/models>

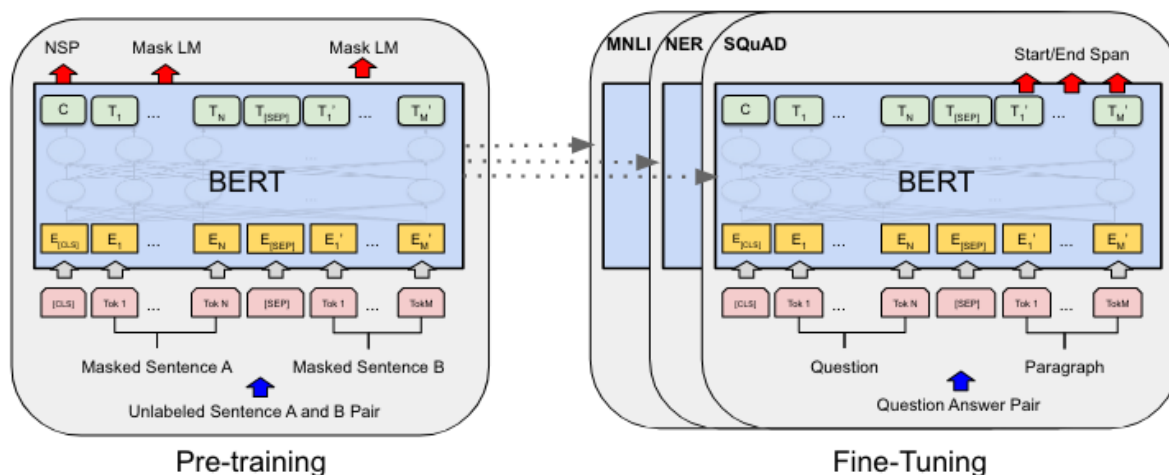


Figure 2.10: BERT pre-training and fine-tuning. During fine-tuning, a task-specific output layer is added and all parameters are fine-tuned. This leverages the language understanding acquired during pre-training for several (possibly low-resource) downstream tasks. (Source: Devlin et al. (2019))

BART

BART (Lewis et al., 2020) (**B**idirectional **A**uto-**R**egressive **T**ransformer) is a transformer-based encoder-decoder architecture that brings the pre-training paradigm to seq2seq models while also using the bidirectional encoder in a BERT-like fashion. While BERT uses only the encoder components of the original transformer architecture, BART uses the encoder as well as the decoder components, as originally suggested by Vaswani et al. (2017). In order to leverage unsupervised language data, the model is pre-trained using corrupted documents with the task to reconstruct these documents. Unlike BERT, which is trained by predicting tokens at positions, BART's objective is to generate the output sequence in an auto-regressive fashion, where the decoder generates each token based on its previously generated output. For pre-training, the authors introduce 5 sequence-based pre-training tasks:

1. *Token Masking*

This objective is similar to BERT's pre-training task, where random tokens are replaced with the [MASK] token.

2. *Token Deletion*

This objective deletes random tokens from the input and the model has to also decide which positions are missing.

3. *Text Infilling*

This objective is related to the *Token Masking* objective but instead of a single word, a span is replaced with a single [MASK] token. This introduces the additional difficulty for the model to predict how many tokens are missing. The span length is drawn from a Poisson distribution with $\lambda = 3$, where 0-length spans correspond to the insertion of [MASK] tokens.

4. Sentence Permutation

For this training objective, a document of shuffled sentences is given to the model and it has to predict the right sequence of sentences.

5. Document Rotation

This objective rotates a randomly chosen token at the beginning of the document. According to the authors, this teaches the model to identify the start of a document.

Even though BART was trained on sequence generation pre-training objectives, it can also be fine-tuned for sequence or token classification tasks. For classification tasks, it receives the same input at the encoder and the decoder, and the final hidden state is classified by a linear classifier. The authors report, that BART performs on par with (at the time) state-of-the-art language classification approaches. However, due to its generative nature, BART is used more often for tasks that involve sequence generation.

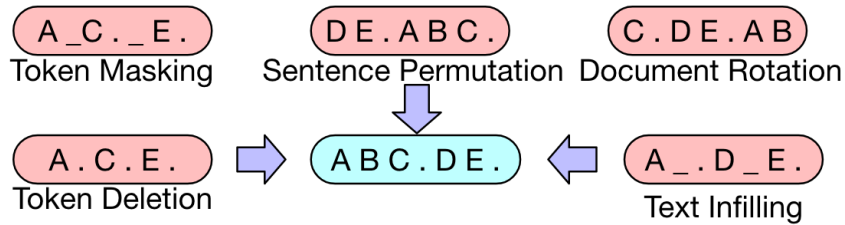


Figure 2.11: The pre-training objectives used for training BART. The authors suggest five ways of corrupting an input sequence and the goal is always to predict the original sequence (Source: Lewis et al. (2020))

2.4 Named Entity Recognition

Named entity recognition (NER) is a research field of natural language processing. The goal of NER is to locate and classify named entities that occur in a text document into pre-defined categories. The term "Named Entity" was first coined in 1996 in the context of Information Extraction tasks (Grishman and Sundheim, 1996). Roughly speaking, named entities are any words that can be referred to with a proper name (Jurafsky and Martin, 2008). Typical examples of named entities are for example "person", "location", or "organization". However, this definition is not rigid and in practice, some datasets include words that exceed this definition, for example, categories for time or money value (Weischedel et al., 2013). Named entity recognition is oftentimes used as a first step in NLP-based problems like question answering or information extraction. In contrast to part-of-speech tagging, where each token is assigned one tag, named entities are usually more sparsely distributed over the sentence and oftentimes span multiple tokens, making the problem more difficult because it basically requires the two sub-tasks of *entity detection* and *entity naming*. Most of the time those subtasks are only implicitly learned by models, however, there exist approaches that explicitly divide the NER problem into those two subtasks, for example in Ma et al. (2022). Formally, the problem of named entity recognition can be defined as follows: Given an input sentence of n tokens, $X = x_1, x_2, \dots, x_n$ the goal is to obtain a triplet (l, r, t) for each entity occurring in X where l and r indicate the left and right inclusive boundary word indices of the entity in X and $t \in \mathcal{T}$ indicates the type of the entity,

where \mathcal{T} is the set of entity types (Wang et al., 2022). The special type \circ is usually used for tokens that don't belong to any entity. For example, given the sentence, $X = \{\text{"The Eiffel tower is located in Paris."}\}$, a NER system might identify the entities $e_1 = (1, 2, \text{"building"})$ and $e_2 = (6, 6, \text{"location"})$.

A *tagging scheme* in the context of NER denotes, how entities are marked in a NER data set. The most popular tagging scheme is BIO tagging, where the beginning and the inside of an entity are marked, as well as the outside token. Another scheme is the BIOES tagging scheme, which differentiates between single-token entities (S) and entities that span multiple tokens. It also introduces a tag that explicitly denotes the last token of an entity span. A simpler approach is the IO scheme, which only decides whether a token is part of an entity or not. In practice, the choice of a tagging scheme matters and is very task and dataset dependent. Ding et al. (2021), for example, found that the IO scheme is working best for their dataset. Alshammari and Alanazi (2021) analyzed the impact of different annotation schemes on the NER task and also empirically found, that the IO scheme works best. However, the drawback of the IO scheme is that it is unable to mark entities that directly follow each other.

Sentence:	Gardemeister achieved podiums at the Monte Carlo Rally, Rally Sweden, Acropolis Rally and the Tour Corse.									
IO:	person		event	event	event	event	event	event	event	event
BIO:	B-PER		B-EV	I-EV	I-EV	B-EV	I-EV		B-EV	I-EV
BIOES:	S-PER		B-EV	I-EV	E-EV	B-EV	E-EV		B-EV	E-EV

Figure 2.12: Visualization of popular tagging schemes BIO, IO and BIOES

In early NER research, a common approach to the problem was the use of hand-crafted rules and heuristics that were applied to a text. Such rules could for example involve looking at the capitalization of a word, using word lists of known entities, preceding or subsequent words, or parsing regular expressions to capture frequently occurring patterns. The obvious disadvantages of rule-based approaches are that those rules are very scenario specific and crafting a well-performing rule set for a different domain might be a very time-consuming task. Later, supervised machine-learning techniques for NER replaced strictly rule-based models. A popular choice to approach NER was to train a Conditional Random Field (CRF) (Lafferty et al., 2001) in a supervised manner, but this approach also required extensive feature engineering to incorporate expert knowledge into the model, basically yielding similar disadvantages as the rule-based approaches. While those approaches can achieve satisfactory results, the resulting models are highly associated with the categories it was trained on, and adding new categories or switching domains is a hard and often infeasible problem making this approach unfit for a semi-automated annotation system where different people work on different domains frequently. Since the advent of large language models such as BERT (Devlin et al., 2019), it is common practice to use a large pre-trained model and fine-tune it on the downstream task NER. While this solution does achieve very high results on the NER task, its success is highly dependent on the availability of a large enough entity corpus. For the purpose of a semi-automated annotation system, however, this is clearly a non-satisfying option because, in practice, a semi-automated annotation system deals with a lot of custom tags and cannot provide enough training data for each tag to employ a simple label-oriented supervised approach.

There exist various datasets for the NER task, for example, CoNLL2003 (Tjong Kim Sang and De Meulder, 2003) based on News articles, I2B2 (Stubbs and Uzuner, 2015) based on medical

data, WNUT2017 (Derczynski et al., 2017) based on social data or OntoNotes 5.0 (Weischedel et al., 2013) based on various genres of text.

2.4.1 Few-Shot NER

Realistically, a system for semi-automatic annotation of text documents needs to be able to predict annotations while only being given a small number of annotations by a human annotator. This problem is commonly known as few-shot named entity recognition. Few-shot NER focusses on the NER scenario where a system is trained on annotations of one or more source domains \mathcal{D}_S and is then tested on a target domain \mathcal{D}_T being only provided with a few labeled support samples per entity class. Yang and Katiyar (2020) formalize the problem of K -shot NER as follows: Given an input sentence, $x = x_{t=1}^T$ and a support set \mathcal{S} for the target tag set \mathcal{C}_T that includes K example samples for each entity in \mathcal{C}_T find the best tag sequence $y = y_{t=1}^T$ for x . Since it is often a hard problem to sample *exactly* K examples per class, Ding et al. (2021) proposed a greedy sampling algorithm that guarantees no more than $2K$ example samples per class. Ding et al. (2021) therefore define the few-shot task as N -way $K \sim 2K$ -shot. While a lot of few-shot learning problems are approached in a setting of episodic learning, recent studies have also used prompt-based learning for few-shot NER. Prompt-based learning is a paradigm in natural language processing, where the particular task at hand is reformulated in such a way, that the pre-trained language model can be used to solve it directly (Liu et al., 2022). The aim of such approaches is to better leverage the language understanding of the model in a more direct manner without the need for additional task-specific layers. While this approach is more straightforward for tasks that don't operate on token level, recent research also studied prompt-based approaches for token-based tasks like named entity recognition. Chapter 3 explores different state-of-the-art techniques of few-shot NER in more detail.

2.4.2 Fine-Grained NER

Most annotated NER datasets only differentiate between entity types of a few coarse categories. The popular CoNLL2003 data set (Tjong Kim Sang and De Meulder, 2003) for example only differentiates between the types PER (person), LOC (location), ORG (organization) and MISC (miscellaneous). Most NER datasets usually differentiate between 4 (e.g. CoNLL2003) and 18 (OntoNotes 5.0) entity types. However, a system for automatic text annotation should be able to differentiate between more fine-grained entity types, depending on the task at hand. For example, a user might not only want to extract all entities of the type *Organization* but might also want to differentiate between cultural, political, or educational organizations. Depending on the target domain, almost any level of fine-grained entity representation could potentially be desired by the user. A step towards a fine-grained dataset is Few-NERD (Ding et al., 2021), which differentiates between eight coarse- and 66 fine-grained entity classes using a manually annotated Wikipedia corpus. Few-NERD aims to be the de-facto standard data set for fine-grained NER and has already been adopted by recent works such as Das et al. (2022), Ma et al. (2022) or Chen et al. (2022). This thesis also adopts the Few-NERD dataset for the experiments that are described in Chapter 4.

3 Related work

This chapter aims to provide an overview of recent research regarding techniques for few-shot NER. First, a general overview of recently published approaches to few-shot NER is given. Then, the five techniques that are used for the experiments in this thesis are explained in a more detailed manner.

3.1 Metric-Based Approaches

Metric-based approaches typically make use of episodic training, where each episode resembles one few-shot learning task (as explained in Chapter 2). The core idea of these approaches is for the model to compute representations of each class using a support set and then perform some kind of similarity calculation of the examples in the query set to those representations.

Fritzler et al. (2019) tackled the task of few-shot named entity recognition using Prototypical Networks (Snell et al., 2017) to learn intermediate representations of words that cluster well into named entity classes. The model learns a prototype for unseen classes by averaging the representations of the support samples for that class. Wiseman and Stratos (2019) applied few-shot classification to NER by proposing a label-based nearest-neighbor approach: Considering an example (support) sentence x' and a corresponding tag sequence y' . For a token x_i in sequence x and x'_i in sequence x' , the probability $y_i = y'_i$ depends on the similarity of the trained word embeddings of x_i and x'_i . Ziyadi et al. (2020) proposed an approach they call example-based, also modeling the correlation between a set of support samples and a query sentence for which the entities need to be determined. They obtain BERT representations of each token in the query and support set and measure the similarity between the query sentence and each of the entities in the support set. Additionally, they combine this token-level-similarity approach with a sentence-level similarity measure. Yang and Katiyar (2020) identified a problem in previous prototypical-based approaches which is that the outside class \circ does not represent any unified semantic meaning. For models that are solely based on prototypes, this leads to them learning noisy prototypes for \circ . Instead of only learning prototypes for each entity class, they represent each token by its contextual BERT representation in the sentence. Additionally, their model *StructShot* makes use of abstract tag transitions distribution (Hou et al., 2020) thus being able to transfer some knowledge about tag transitions from the source to the target domain using a Viterbi (Forney, 1973) decoder component. Yamada et al. (2020) proposed a new contextualized entity representation based on BERT. They achieve this by applying a new pre-training task to BERT which involves predicting randomly masked words and entities in an entity-annotated corpus (in contrast to only predicting masked words) and obtaining contextualized entity representations. They also introduced an entity-aware self-attention mechanism that is able to differentiate between words and entities while computing attention scores. Yu et al. (2021) employ a retrieval-based method, matching token spans in the input to the most similar labeled spans in a retrieval index. Huang et al. (2021) explored among other things, how leveraging freely available data (such as web data) as supervised pre-training data can be beneficial for the few-shot NER

setting and conducted experiments using *WiFiNE* (Ghaddar and Langlais, 2018), a heuristically annotated Wikipedia corpus as noisy supervised pre-training data. They concluded that it can significantly boost accuracy in the few-shot setting. Ding et al. (2021) proposed the first large fine-grained NER data set with eight coarse- and 66 fine-grained entity types consisting of 188,238 manually annotated sentences from Wikipedia. They also implemented ProtoBERT, a Prototypical Network based on BERT representations and the approach mentioned above by Yang and Katiyar (2020), and concluded that few-shot fine-grained NER is a challenging task that is far from being solved. Additionally, they provided three different benchmark tasks (SUP, INTER, INTRA) in order to better compare the knowledge transfer capability of different models on their data set. Das et al. (2022) proposed entity representations based on Gaussian Embeddings (Vilnis and McCallum, 2014) instead of fixed vectors. This is supposedly increasing the generalizability of tags with a low number of support samples. It is trained by using contrastive learning: Decreasing the distance of the Gaussian token embeddings of similar entities while increasing it for dissimilar ones. Classification is performed via nearest neighbor inference on the Gaussian Embeddings. They also combine the nearest neighbor classification with Viterbi decoding like Yang and Katiyar (2020). J Ma et al. (2022) also use a similarity measure, but incorporate label semantics into the measure by using two BERT encoders - one for the document and one for the labels. The model then learns to match token representations to label representations. This approach allows leveraging the semantics of a label class into the decision process. Ma et al. (2022) approach the problem of few-shot NER by training a label-agnostic span detection network and an entity-typing network independently. At inference time the two modules first are fine-tuned on the support set. Then the span-detection network outputs spans of the query sentence which are sent to the entity typing network.

3.2 Prompt-Based Approaches

Over the last few years, a new paradigm emerged in the world of natural language processing: *Prompt-based learning*. This paradigm is all about reformulating the task in such a way that a language model like BERT (Devlin et al., 2019) or BART (Lewis et al., 2020) can understand it because it is similar to its pre-training task. (Liu et al., 2022). For example, a model with the task of detecting emotions of a social-media post would get the input sentence "*I missed the bus today*" and in addition, the prompt "*I felt so _*". The language model is then supposed to fill in the blank, just like it does during the pre-training task. R Ma et al. (2022) formalize a prompt as consisting of a template function $T_{prompt}(\cdot)$ that converts the input x to a prompt input $x_{prompt} = T_{prompt}(x)$ and a set of label words \mathcal{V} which are connected with the label space through a mapping function $\mathcal{M} : \mathcal{Y} \mapsto \mathcal{V}$. The template then is a string with two unfilled slots, one slot X to fill the input and one answer slot Z for the model. For a sentiment classification task, the template might look like this: "[X] It was [Z]". For a named entity recognition task, another slot [S] is needed to specify the token which is labeled. A template for a NER task could look like this: "[X] [S] is a [Z] entity". This approach takes away the fine-tuning to downstream tasks via objective engineering and, in theory, offers a flexible solution to leverage the raw language understanding of the pre-trained model. However, it shifts the engineering work to be done to the prompts since finding a good and effective prompt depends a lot on the task at hand, and finding an adequate one is often simply a trial-and-error process.

Schick and Schütze (2021) proposed to reformulate the input to a cloze-style question effectively trying to provide task descriptions for the model and utilizing it for supervised tasks in a few-shot

setting. They do this by employing a technique called *pattern-exploiting-training*, where first, an ensemble of models is fine-tuned on the cloze-style patterns and is then used on a larger unlabelled dataset to infer soft labels. Finally, the resulting labeled dataset is used to train a classifier. Gao et al. (2021) improved upon their approach using automatically generated prompts and showed that leveraging automatically generated prompts together with task demonstrations as part of the input context enables language models to be used for text classification tasks. Cui et al. (2021) trained a BART encoder-decoder network on templates, leveraging the generative output of such models. During training, if a token c belongs to entity e , the template $t = "c \text{ is a } e \text{ entity}"$ is generated and given to the decoder as the desired output. During inference, first, all possible n -grams in the sentence are generated (up to $n = 8$), then for each template T , a score is generated and c is assigned to the entity of the highest scoring entity template. Lee et al. (2022) also augment the original input by appending automatically created task demonstrations and feeding them alongside the tokens to the language model. Wang et al. (2022) also inject the task instruction directly at the input, together with the input sentence and labeling options. To boost performance they additionally define two auxiliary tasks *entity extraction* and *entity typing* that together make up the named-entity recognition task. R Ma et al. (2022) identify two main disadvantages of manual template crafting for prompt-based learning in the context of named entity recognition: It is inefficient as the search space can grow quite large and obtaining the label of each token requires enumerating all possible spans (as seen, for example, in Cui et al. (2021)). Instead of templates, they propose the *EntLM* fine-tune objective where they train a BERT model to replace an entity name with a label word, which is a representative of the corresponding entity class. For example, the input sentence "Obama was born in America" will generate "John was born in Australia", with "John" and "Australia" being label words for the classes "Person" and "Location". An advantage of this approach is that all entities can be determined in one pass through the model (in contrast, for example, to Cui et al. (2021)). They also propose methods for finding appropriate label words.

3.3 Model Architectures

After general research with regard to few-shot named entity recognition, five state-of-the-art systems have been chosen to run further experiments on. The systems have been picked according to their reported results on the few-shot NER task. To achieve a balanced overview of current approaches to few-shot NER, two meta-learning models (ProtoBERT (Ding et al., 2021) and StructShot (Yang and Katiyar, 2020)) and two prompt-based models (TemplateNER (Cui et al., 2021) and EntLM (R Ma et al., 2022)) have been chosen for further experiments. Furthermore, experiments have been done using adapter-transformer (Pfeiffer, Rücklé, et al., 2020), as their design naturally makes them a good fit for quickly switching annotation task contexts without much storage overhead. This section describes the chosen approaches in a more detailed manner.

3.3.1 ProtoBERT

ProtoBERT is one of the models that were implemented in the original Few-NERD publication (Ding et al., 2021). It is based on Prototypical Networks, which were first described by Snell et al. (2017). Like many solutions for few-shot learning, they are based on an episodic learning approach. The core assumption of Prototypical Networks is, that there exists an embedding

representing a class, around which all examples of that class cluster. To receive these cluster centroids, they internally build up prototype representations of each class of the support set. An example is then classified by computing some distance of that example to the prototype. The authors report squared Euclidian distance to work best but state that any distance metric is permissible (Snell et al., 2017). To classify examples, a non-linear mapping of the input space to an embedding space is learned using an embedding function $f_\phi : \mathbb{R}^D \rightarrow \mathbb{R}^M$. The prototype of a class C simply is computed as the mean of all classes x_i in the support set, where $y_i = C$. Classification is performed by computing a softmax function over distances to class prototypes.

More formally: Prototypical Networks compute an M -dimensional representation $\mathbf{c}_k \in \mathbb{R}^M$ for each training example through an embedding function $f_\phi : \mathbb{R}^D \rightarrow \mathbb{R}^M$ with learnable parameters ϕ . A prototype for a class is the mean vector of all the support points that belong to this class:

$$\mathbf{c}_k = \frac{1}{|S_k|} \sum_{(\mathbf{x}_i, y_i) \in S_k} f_\phi(\mathbf{x}_i). \quad (3.1)$$

A distance function $d : \mathbb{R}^M \times \mathbb{R}^M \rightarrow [0, +\infty)$ is used to find the prototype for a specific class by computing a softmax function over the distances to the prototypes in the embedding space:

$$p_\phi(y = k|\mathbf{x}) = \frac{\exp(-d(f_\phi(\mathbf{x}), \mathbf{c}_k))}{\sum_{k'} \exp(-d(f_\phi(\mathbf{x}), \mathbf{c}_{k'}))}. \quad (3.2)$$

During the training phase, the negative log-likelihood is used as a loss function:

$$J(\phi) = -\log p_\phi(y = k|\mathbf{x}). \quad (3.3)$$

ProtoBERT (Ding et al., 2021) implements this concept by using BERT (Devlin et al., 2019) as an encoder model for class representations.

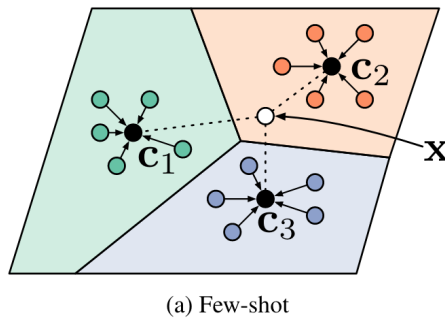


Figure 3.1: The classification process in a Prototypical Network. c_1, c_2, c_3 are the centroids of the prototype clusters and x is the example to classify (Source: Snell et al. (2017)).

3.3.2 StructShot

StructShot is the second model that the authors of Few-NERD implemented in their original publication. It was first proposed by Yang and Katiyar (2020). The core idea of StructShot is that, instead of employing class representations to classify samples by calculating a mean of

the classes in the support sample and then calculating distances to the class representatives (as ProtoBERT does), it simply makes use of a token-level nearest neighbor classification. Given a sentence \mathbf{x}_1^T and a support set $\mathcal{S} = (\mathbf{x}_n, \mathbf{y}_n)_{n=1}^N$ of N sentences, the model uses a token embedder $f_\phi(x)$ to obtain contextual representations for all tokens in the sentence. The model employed in Ding et al. (2021) uses a BERT encoder (Devlin et al., 2019) trained on a source domain as token embedders. A similarity score between each token of the sentence in question and each token in the support set is calculated and the token x is assigned the tag c that corresponds to the most similar token in the support set:

$$y^* = \arg \min_{c \in 1, \dots, C} d_c(\hat{x}) \quad (3.4)$$

and

$$d_c(\hat{x}) = \min_{x' \in \mathcal{S}_c} d(\hat{x}, \hat{x}'). \quad (3.5)$$

To compute the distance, the squared Euclidian distance $d(\hat{x}, \hat{x}') = \|\hat{x} - \hat{x}'\|_2^2$ is used.

According to the authors, this token-level similarity measure solves a problem of prototype-based approaches: Prototype-based approaches learn prototypes even for the outside class \circ . However, in contrast to entity types, the prototypes for the outside class are noisy because they do not carry any semantic meaning whatsoever. In certain cases, noisy \circ prototypes may lead to inaccuracies during classification, causing an outside sample to be falsely classified as an inside sample because, accidentally, its embedding is more similar to some class prototype than to the \circ prototype.

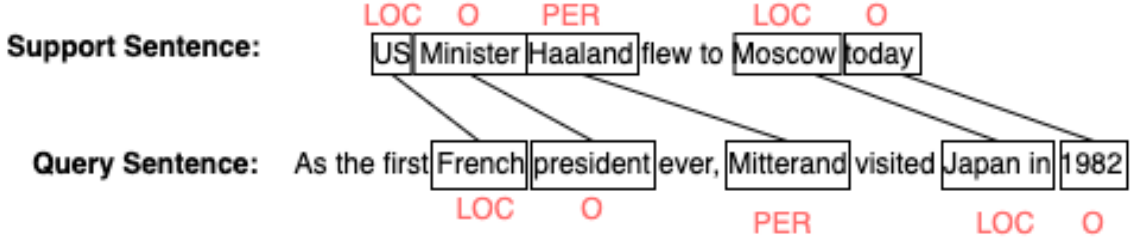


Figure 3.2: The token-level based inference as described in Yang and Katiyar (2020)

Additionally, during inference an optimization step is included in StrutShot, using a Viterbi (Forney, 1973) decoder component. During training on the source domain, a transition probability is computed by counting the number of times each transition occurred. Since the tags during on-domain training might differ from the tags that are used during the actual few-shot training, only abstract transition probabilities between \circ , I , and I -other - for the case that two different entity types directly follow each other - are counted. With the emission probabilities given as

$$p(y = c|x) = \frac{e^{-d_c(\hat{x})}}{\sum_{c'} e^{-d_{c'}(\hat{x})}} \quad (3.6)$$

and the abstract transition distribution $p(y'|y)$, the Viterbi decoder is used to solve the following problem:

$$y^* = \arg \max_y \prod_{t=1}^T p(y_t|x) \cdot p(y_t|y_{t-1}). \quad (3.7)$$

Finally, they introduce a hyper-parameter τ to normalize the transition probabilities to a similar scale.

3.3.3 TemplateNER

TemplateNER is a prompt-based approach to few-shot named entity recognition. As presented above, the underlying assumption of those approaches is that the model already learned a sufficient understanding of natural language in the pre-training stage and that this knowledge can be used in a more direct fashion than fine-tuning it on supervised training data, leading to a more efficient and direct way of classification. Cui et al. (2021) introduced a template-based method to better exploit the language understanding of pre-trained language models for the task of NER. They fine-tune a BART model on pre-defined templates, filled with words from the training set. For example, a common template for named entity classification would have the form of

<sentence>. X is a Y entity.

Another example of a possible template would be:

<sentence>. X belongs to the Y category.

For example, given the sentence $\mathbf{X} = \textit{This year, we traveled to Spain}$, where "Spain" is labeled "Location" and the other words are labeled *Outside* ("O"), using the first template \mathbf{T} above we would derive the following pair (\mathbf{X}, \mathbf{T}) :

This year, we traveled to Spain. Spain is a location entity.

A BART model would then be fine-tuned to predict "*Spain is a location entity*" when it encounters that sentence. To balance the model out, in order for it to not predict too many entities, a template for non-entities is needed as well. A template for non-entity spans might, for example, look like this:

<sentence>. X is not a named entity.

Applying the negative template to the example sentence above, would, for example, look like this:

This year, we traveled to Spain. we is not a named entity.

Formally: First, given a label set $\mathbf{L} = l_1, \dots, l_n$, a mapping function transforms \mathbf{L} to a set $\mathbf{Y} = y_1, \dots, y_n$ of natural words (e.g. "LOC" becomes "location"). This step is important because in order to leverage the language understanding of generative models, the generated classification sentences need to carry a semantic meaning, whereas "LOC" only has a symbolic meaning. For datasets like Few-NERD, this step is not necessary, because it already uses natural language labels. Then, for each $y_i \in \mathbf{Y}$, a positive Template $\mathbf{T}_{y_i}^+$ is created (e.g. <sentence>. X is a y_i

entity.). Additionally, a negative template (as seen above) \mathbf{T}^- is created. The ratio of negative to positive templates is a hyperparameter and was set to 1.5 in the original paper.

For inference, given a sentence $\mathbf{X} = x_1, \dots, x_n$ all possible candidate spans are generated and corresponding templates are filled. The candidate spans are all 1-grams, 2-grams, ..., n -grams of the sentence (given the example sentence above, they would be: "This", "year", ..., "Spain", "This year", ..., "to Spain", ... "This year, we", ..., "traveled to Spain", and so forth). In order to avoid unnecessary complexity, the original authors capped n at 8. Then, using BART, a score $f(\mathbf{T}_{y_k, x_{i:j}}^+)$ is calculated for each template and for each generated span. For each span, the entity type with the largest scoring template is assigned to the span. The detection of nested entities is not provided. For overlapping spans, that predict different entities, a scoring function is used to predict the entity whose template has a larger score (Cui et al., 2021). The inference process of TemplateNER is depicted in Figure 3.3.

For training, first, the dataset is transformed into templates. Given a gold sentence \mathbf{X} , where a span $x_{i:j}$ has entity type y_k , a corresponding sentence $\mathbf{T}_{y_k, x_{i:j}}^+$ "X. $x_{i:j}$ is a y_k entity." is created. In a similar fashion, negative templates \mathbf{T}^- are filled. Negative entity spans are sampled randomly (Cui et al., 2021). Then, given a pair of sentences and template (\mathbf{X}, \mathbf{T}) , \mathbf{X} is fed into the encoder component of the BART model and a hidden representation of the sentence

$$\mathbf{h}^{enc} = \text{Encoder}(x_{1:n}) \quad (3.8)$$

is obtained. The output of the decoder at the c th step depends on \mathbf{h}^{enc} and previous output tokens $t_{1:c-1}$:

$$\mathbf{h}_c^{dec} = \text{Decoder}(\mathbf{h}^{enc}, t_{1:c-1}). \quad (3.9)$$

The probability of the next generated word is then defined as

$$p(t_c | t_{1:c-1}, \mathbf{X}) = \text{softmax}(\mathbf{h}_c^{dec} \mathbf{W}_{lm} + \mathbf{b}_{lm}) \quad (3.10)$$

with \mathbf{W}_{lm} and \mathbf{b}_{lm} being the weights and the bias of the decoder model with $\mathbf{W}_{lm} \in \mathbb{R}^{d_h \times |\mathcal{V}|}$ and $\mathbf{b}_{lm} \in \mathbb{R}^{|\mathcal{V}|}$. $|\mathcal{V}|$ represents the size of BART's vocabulary and d_h the dimension of the hidden representation. The training objective is to minimize the cross-entropy between the decoder's output and the original template.

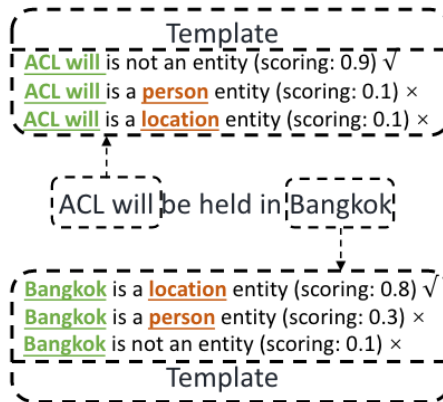


Figure 3.3: The inference process of TemplateNER (Source: Cui et al. (2021))

3.3.4 EntLM

EntLM (R Ma et al., 2022) is another prompt-based approach to few-shot NER. This approach solves the problem of the impracticability of template-based approaches for few-shot NER like Cui et al. (2021), which have to iterate over all possible spans to classify entities, resulting in very long inference times. Instead, they utilize the language understanding of pre-trained language models by specifying an entity-oriented masked LM objective, in which a pre-trained model is fine-tuned to predict class-related label words instead of the original words at the entity positions, while still predicting the original word at none-entity positions (R Ma et al., 2022). With this training objective, the advantage of prompt-based learning, which is that the output of the language model is used directly, without the need for additional layers, is kept, without the problems that template-based approaches like Cui et al. (2021) entail.

Formally: Given an input sentence $\mathbf{X} = x_1, \dots, x_n$ and a label sequence $\mathbf{Y} = y_1, \dots, y_n$, a target sequence $\mathbf{X}^{Ent} = x_1, \dots, \mathcal{M}(y_i), \dots, x_n$ is constructed, in which the token at the position of the entity at position i is replaced with the corresponding label word $\mathcal{M}(y_i)$. The language model is then trained to maximize the probability of the modified sentence \mathbf{X}^{Ent} given the source sentence \mathbf{X} :

$$\mathcal{L}_{EntLM} = - \sum_{i=1}^n \log P(x_i = x_i^{Ent} | \mathbf{X}), \quad (3.11)$$

where $P(x_i = x_i^{Ent} | \mathbf{X}) = \text{softmax}(\mathbf{W}_{lm} \cdot \mathbf{h}_i)$. During inference, given a sentence \mathbf{X} , the probability of labeling a token x_i with class y is given by:

$$p(y_i = y | \mathbf{X}) = p(x_i = \mathcal{M}(y) | \mathbf{X}), \quad (3.12)$$

where \mathcal{M} is the function that maps label words to specific classes (R Ma et al., 2022). The inference process is depicted in Figure 3.4

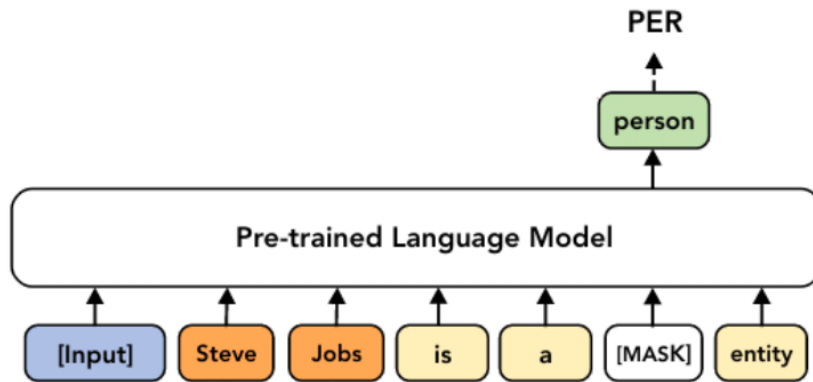


Figure 3.4: The inference process of EntLM (Source: R Ma et al. (2022))

Naturally, this approach implies the need for label words, that appropriately represent the class they stand for, to function well. The authors name three approaches for deriving the most appropriate label words: Searching with data distribution, searching with LM output distribution (R Ma et al., 2022), and a combination of those.

Searching Label Words using Data Distribution

Searching for label words using data distribution is the most intuitive approach to label word searching. When searching for a suitable label word for a class C , $\phi(x = w, y = C)$ describes the frequency of word w belonging to class C . The label word is then the most frequently occurring word w with the label C :

$$\mathcal{M}(C) = \arg \max_w \phi(x = w, y = C). \quad (3.13)$$

Searching Label Words using LM Output Distribution

The second approach for label word searching suggested by the authors leverages the pre-trained language model for label word searching. For this, each sample (X, Y) of the data is fed into the language model to receive the probability distribution $p(x_i = w | X)$ of predicting each word $w \in \mathcal{V}$ at each sentence position j , where \mathcal{V} is the vocabulary of the language model. The label word for a class C is then the word that is most frequently predicted for words that belong to class C . More Formally:

$$\mathcal{M}(C) = \arg \max_w \sum_{(X,Y)} \sum_i^{|X|} \phi_{topk}(x_i = w, y_i = C), \quad (3.14)$$

where $\phi_{topk}(x_i = w, y = C)$ returns 1, if w belongs to the top k predictions for x_i in sample (X, Y) and y_i is C (R Ma et al., 2022). k is a hyperparameter and the authors state $k = 6$ achieves good results throughout all experiments.

Combining label counting in the data and LM output distribution

The authors suggest a third approach for finding appropriate label words, combining the naive counting method with the method that considers LM output distribution:

$$\mathcal{M}(C) = \arg \max_w \left\{ \sum_{(X,Y)} \sum_i^{|X|} \phi(x_i = w, y_i = C) \cdot \sum_{(X,Y)} \sum_i^{|X|} \phi_{topk}(x_i = w, y = C) \right\}. \quad (3.15)$$

Label Word Search using Distantly Matched Labels

In most cases, deriving label representatives from the (few-shot) training corpus won't deliver good results though, as the sample size of words and their labels is too small and the above approaches will not lead to sufficient results. To counteract this problem, the authors suggest making use of larger, unlabeled domain data and employing distant supervised techniques. They specifically recommend BOND (Liang et al., 2020) for this approach. BOND is a RoBERTa-based (Y Liu et al., 2019), two-stage approach for labeling an unlabeled corpus with the help of a knowledge base. First, distantly matched labels of the target domain are generated with the

help of the knowledge base, in this case wikidata¹, using string matching, regular expressions, and handcrafted rules (Liang et al., 2020) and a RoBERTa model is trained on those generated labels. They then employ a teacher-student approach, in which the student model is trained using the pseudo-labels generated by the teacher model, and the teacher is updated by the student accordingly. According to the authors, this additional teacher-student step adds a significant edge to performance compared to the standard distant supervised algorithms (Liang et al., 2020)

3.3.5 Adapter

Adapters for NLP tasks have first been introduced in Houlsby et al. (2019) with the goal of making the process of fine-tuning large language models (like BERT or GPT) more parameter-efficient. Traditionally, when fine-tuning a transformer model for an NLP task, the entire model is fine-tuned by modifying its parameters. Depending on the size of the pre-trained model, this is computationally expensive and time-consuming (GPT-3 (Brown et al., 2020) for example has 175 billion parameters). Additionally, this approach naturally requires saving all the parameters of the fine-tuned model for each task at hand, making it impractical in a scenario where different tasks arise frequently. Houlsby et al. (2019) solve these issues by adding new modules between the layers of a pre-trained transformer. Formally, traditional fine-tuning of a transformer neural network $\phi_{\mathbf{w}}(\mathbf{x})$ is done by adjusting the parameters \mathbf{w} of the pre-trained model. Adapter-transformer introduce a new set of parameters \mathbf{v} to the model resulting in a function $\psi_{\mathbf{w},\mathbf{v}}(\mathbf{x})$, where only the parameters \mathbf{v} are modified during training. The newly added parameters \mathbf{v} make up 0.5% - 8% of the parameters of the original model (Houlsby et al., 2019). This technique makes the training of a pre-trained language model faster while at the same being parameter efficient since the weights of the original model are shared through all tasks. Additionally, Houlsby et al. (2019) show that their approach does not substantially degrade the accuracy of the model. A downside of this approach is that the inference time is slightly increased due to the newly introduced parameters. However, in practice, the advantages of this approach often outweigh this disadvantage.

Adapter Architectures

The approach of adding trainable parameters between transformer layers is in itself a theoretical approach, leaving room for many different architectural choices and configurations. Furthermore, integrating additional layers is not straightforward. In an attempt to make the creation of adapter modules and their different practical implementations, their usage, and the sharing of adapter modules more accessible, Pfeiffer, Rücklé, et al. (2020) released a framework that enables the user to easily plug in adapter layers into an existing huggingface transformer model. They also provide a variety of different adapter architectures and configurations and provide a website, where trained task adapters can be uploaded and shared². The following section will briefly describe different possible adapter configurations provided by Pfeiffer, Rücklé, et al. (2020)

Houlsby et al. (2019) suggest placing the adapter layers after both the multi-head attention layer and after the feed-forward layer of the transformer encoder. The adapter layers are injected directly behind the output of the multi-head attention and the feed-forward layer, respectively.

1. <https://www.wikidata.org/>

2. adapterhub.ml

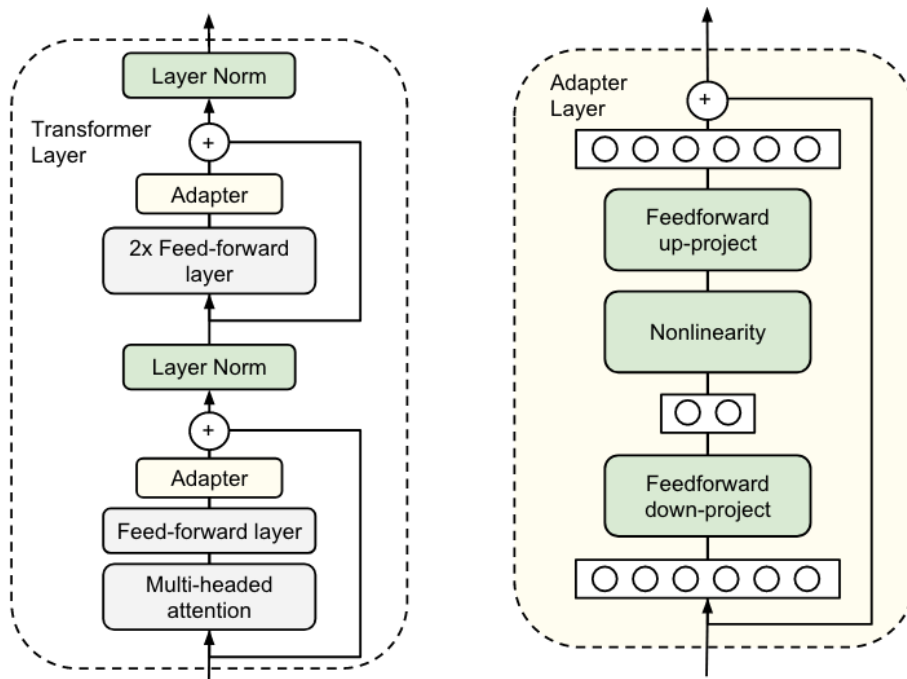


Figure 3.5: Illustration of a basic Bottleneck Adapter module. Adapter layers can be inserted after the multi-head attention layer as well as after the feed-forward layer of a transformer network (left). To save parameters, internally, they project their input down to a lower dimension m , compute a non-linearity function on it, and project it up to dimension d (right) (Source: Houlsby et al. (2019)).

In order to keep the number of parameters small, each adapter layer projects the original d -dimensional input from the layer it is adapted to into a smaller dimension m . After that, a non-linearity is applied and the input is projected up to d dimensions again. Configurations that utilize a down-and-up projection inside the adapter layers are called *Bottleneck Adapters*. A basic bottleneck adapter configuration is depicted in Figure 3.5. On adapterhub.ml, this configuration is represented by the `HoulsbyConfig` class.

Pfeiffer, Vulić, et al. (2020) proposed a slightly different adapter configuration in an attempt to study how adapters can be utilized to transfer knowledge between a multilingual transformer model and a target language. To transfer knowledge between languages, they first train language adapters on unlabelled data of a source as well as the target language using the masked language model training objective on a Wikipedia corpus (Pfeiffer, Vulić, et al., 2020). They then train a language-agnostic task-specific adapter (for example for the NER task) that is stacked on top of the source language adapter. Stacking adapters first has been introduced by Pfeiffer, Kamath, et al. (2020) who showed that stacking adapters can effectively combine the knowledge of each individual adapter to a certain extent, without leading to catastrophic forgetting. During the training of the task adapter, the weights of the language adapter, as well as the weights of the transformer model, are frozen. In a third step, to apply zero-shot transfer to the target language, the source language adapter is switched out with the target language adapter, effectively combining the general knowledge of the target language adapter with the knowledge of the fine-tuned task adapter, without the need for actually having seen labeled examples of the target language. The adapters used in this approach are bottleneck adapters as described by Houlsby et al. (2019). However, Pfeiffer, Vulić, et al. (2020) only use the adapters after the feed-forward layer and not, like Houlsby et al. (2019), also after the multi-head attention layer. On adapterhub.ml, this arrangement of adapter layers is represented by the `PfeifferConfig` class. To capture language-specific token-level transformations, they also introduce *invertible* adapters, which are stacked after the embedding layers, with their respective inverse being stacked after the output embedding layer. Invertible adapters are provided on adapterhub.ml via the `inv_adapter` attribute of the `AdapterConfig` class.

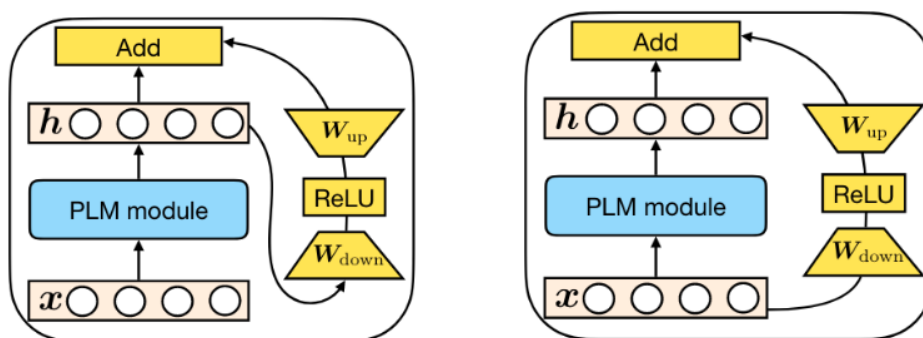


Figure 3.6: Sequential adapter configuration (left) and parallel adapter configuration (right)
(Source: J He et al. (2021))

J He et al. (2021) systematically analyzed state-of-the-art design choices regarding parameter-efficient fine-tuning of language models and introduced a novel framework that describes the injection of additional parameters into the model across four dimensions. They also proposed a new form of inserting additional layers into the pre-trained transformer model in a parallel fashion. Let x be the input to a certain sublayer of a pre-trained transformer model, h the

output of that sublayer, and \mathcal{T} the adapter transformation $f(\mathbf{h}\mathbf{W}_{down})\mathbf{W}_{up}$ where f is a non-linear function like ReLu. Traditional adapter (Houlsby et al., 2019) compute the transformation after x has passed through the sublayer and add the output of \mathcal{T} to the original output h . The parallel approach suggested by J He et al. (2021) - which was also independently suggested by Yaoming Zhu et al. (2021) - computes $\mathcal{T}(x)$ directly and adds $\mathcal{T}(x)$ to h afterwards. This is illustrated in Figure 3.6. Yaoming Zhu et al. (2021) performed extensive empirical research on parallel adapter configurations in the context of neural machine translation and concluded that they can improve the performance of transformer models on various translation tasks. On adapterhub.ml, this configuration can be achieved by using the `ParallelConfig` class.

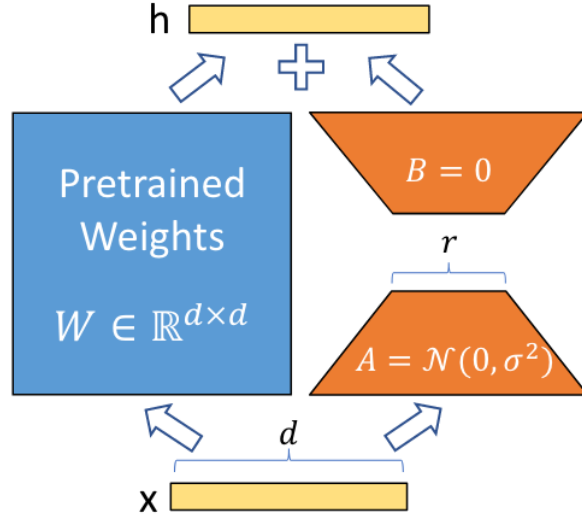


Figure 3.7: Decomposing ΔW into two low-rank matrices A and B (Source: Hu et al. (2021))

A different approach of injecting trainable layers into a pre-trained transformer model is described by Hu et al. (2021). This approach, called LoRA for "**L**ow-**R**ank **A**daption", is inspired by Aghajanyan et al. (2021), who empirically showed that the intrinsic dimensionality of large pre-trained language models is a lot lower than the number of parameters they actually have. Hu et al. (2021) take this assumption a step further and also assume that change in weights during training also has a low intrinsic rank. Let $W_0 \in \mathbb{R}^{d \times k}$ be the pre-trained weight matrix of a transformer model and ΔW be the update to this matrix during training. To restrict possible updates of this potentially very large weight matrix, LoRA decomposes ΔW by representing it with the help of two lower-rank matrices A and B , where $A \in \mathbb{R}^{r \times k}$ and $B \in \mathbb{R}^{d \times r}$ and $r \ll \min(d, k)$. Just like in other adapter configurations, during training, W_0 is frozen and only parameters in A and B are trained:

$$\mathbf{h} = W_0 x + \Delta W x = W_0 x + B A x. \quad (3.16)$$

An important advantage this approach has over adapter like Houlsby et al. (2019), or Pfeiffer, Vulić, et al. (2020) is that A and B can be merged into W at inference time by adding $W = W_0 + B A$ which eliminates the overhead in inference time the other approaches suffer from. Additionally, task switching also demands very little computational overhead: If A' and B' are decomposed matrices for a second task, then switching to that task simply means subtracting AB from W_0 and adding $A'B'$ to it. (Hu et al., 2021).

4 Experiment and Dataset

4.1 Dataset

Named entity recognition datasets are crucial resources in the field of natural language processing and machine learning and typically contain annotated text data, where each named entity is labeled and categorized for training purposes. Over the years, several NER datasets have emerged and established themselves as popular, the most popular datasets being the CoNLL-2003 (Tjong Kim Sang and De Meulder, 2003) dataset and the OntoNotes5 (Weischedel et al., 2013) dataset, providing annotations for four (CoNLL) and 18 (OntoNotes5) different entity types, respectively. However, CoNLL-2003 as well as OntoNotes provide only rough-grained entities: CoNLL-2003 differentiates between Person, Organization, Location, and Miscellaneous entities and OntoNotes differentiates between Person, Organization, Location, Date, Time, Money, Percent, Facility, GPE, Vehicle, Weapon, Event, Product, Work of Art, Language, Law, NORP (Nationality, Religious or Political Group) and Miscellaneous types. In the practical setting of an automated annotation system, however, even more fine-grained entity types might occur. For example, while OntoNotes provides the entity type "Event", there might be cases in which an annotator wants to differentiate between the event of a war, the event of a catastrophe, or a protest. The same holds for example for the type of "Organization", where it's more likely that an annotator may want to differentiate between a political party or, for example, a company, instead of both being labeled Organization. 2021, the Few-NERD dataset (Ding et al., 2021) was released with the goal of providing a higher differentiation factor, making it useful for research that aims at working with more fine-grained entities. Few-NERD provides 66 fine-grained entity types, each type being assigned to one of eight parent entities (for example the parent entity "Person" has child entities like "Actor", "Athlete" or "Artist"). Few-NERD consists of 188,238 sentences from the English Wikipedia and provides a total of 4,601,160 annotated words which makes it the largest NER dataset to the point of writing this (CoNLL includes around 20,000 sentences and OntoNotes includes around 100,000 sentences). The dataset has been annotated by 70 annotators with linguistic knowledge, as well as 10 experienced experts. Each paragraph was annotated by two annotators. After, an experienced expert overlooked the annotated paragraph and made the final decision. Figure 4.1 shows an overview of all the classes that are included in the dataset.

Important to note is, that the authors define the few-shot NER setting as a setting of episodic learning, in which a model is trained on episodes. Each episode contains a query set and a support set and for each episode, the goal of the model is to predict the labels contained in the query set with the help of the support set (as explained in Chapter 2). For example, in a 4-way 5-shot setting, one episode may contain the classes "Person", "Location", "Event", and "MISC", with the support set containing 5 examples of each of those classes. In this case, the query set contains sentences where only those 4 classes occur, and the task of the model is to predict the classes correctly after having seen the classes in the support set. Depending on episode size and training corpus, it is often not possible to sample sentences in such a way that they include exactly K samples for each of the N classes. Ding et al. (2021) tackle this issue by relaxing the

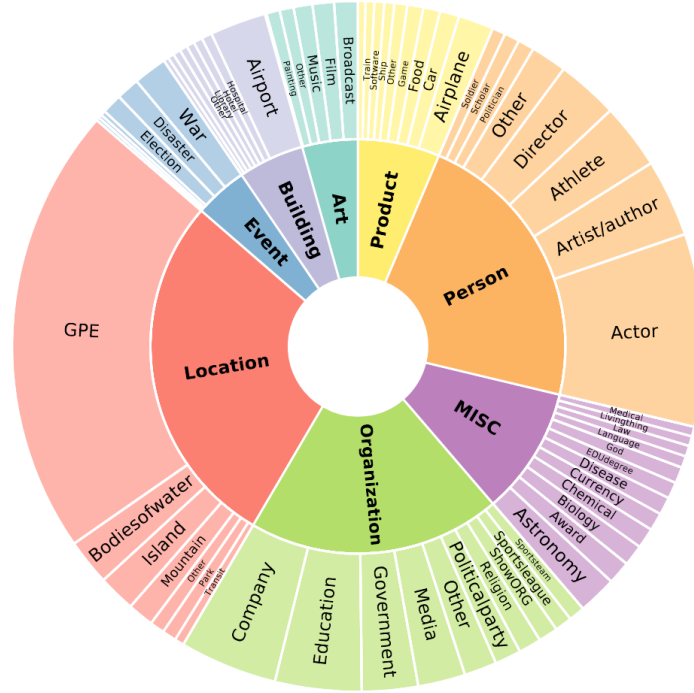


Figure 4.1: Overview of all classes present in the Few-NERD dataset. The inner circle represents the parent classes, the outer circle represents the child classes (Source: Ding et al. (2021))

N -way K -shot setting to an N -way $K \sim 2K$ -shot setting, ensuring that each class occurs at least K times but no more than $2K$ times.

Few-NERD provides three different data splits, SUP, INTER, and INTRA. SUP is the standard supervised split, in which the available data is randomly split into training data (70%), testing data (20%), and validation data (10%). In order to better assess the ability of the model to generalize and transfer knowledge between episodes, the INTER and INTRA split have been developed. In the INTRA split, all entities in different sets (train, test, and development) belong to different coarse-grained entities (e.g. entities that have different parent entities). More specifically, the train set includes the types "People", "MISC", "Art" and "Product", the development set includes the types "Event" and "Building" and the test set includes the types "Organization" and "Location". In the INTER split, only the fine-grained entity types are mutually disjoint, while coarse-grained types are shared, leading to roughly 60% fine-grained types being in the training set, 20% in the test set, and 20% in the validation set. In contrast to other popular NER datasets which most often use the BIO Tagging scheme, Few-NERD makes use of the IO tagging scheme, as the authors achieved slightly better results with it.

For the experiments conducted in this thesis, the Few-NERD dataset was used. Having up to 66 classes is helpful in determining how well a classifier is able to handle more than 18 classes - which was the highest number of classes a NER dataset provided before Few-NERD was released. Its class richness also allows the creation of several sub-datasets, for example, datasets that only contain one or two coarse-grained types, in order to assess how well the model is able to differentiate between the corresponding fine-grained classes without giving it the challenging task of differentiating between 66 fine-grained classes. The hierarchical division into parent

types also allows for experiments in which the model is pre-trained on only the coarse-grained types and is then given a few examples of different fine-grained types.

4.2 Evaluation Metrics

The most basic metric for evaluating classification tasks would be to compute the percentage of correctly classified instances over all instances (accuracy). However, in unbalanced datasets like NER datasets, this metric is not ideal. In order to get a deeper understanding of what the classifier can and cannot do, it is more common to take into account the number of *true positives*, *false positives*, *false negatives*, and *true negatives* and use them to compute the metrics *precision*, *recall*, and *F1*. Given an entity class C , the true positives of class C are all tokens that belong to C and are labeled as C , the false positives are all tokens that don't belong to C , but are labeled as C , the false negatives are all tokens that belong to C but are not labeled as C and the true negatives are all tokens that don't belong to C and are not labeled as C . Figure 4.2 visualizes those terms using a 2-class scenario.

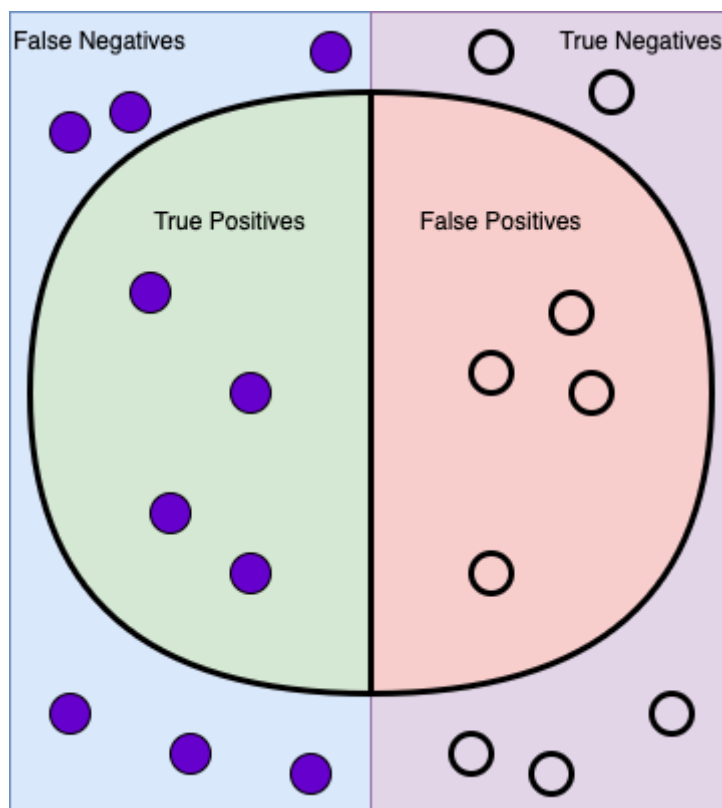


Figure 4.2: Visualisation of true positives, false positives, true negatives, and false negatives in a 2-class setting. The circle represents the classifier.

4.2.1 Precision

Precision measures the fraction of correctly predicted entities over all predicted entities. A high precision score indicates that, if the model did predict an entity, that prediction is most likely reliable and the number of erroneously predicted entities is low. On the other hand, a low

precision score suggests that the system tends to overpredict certain classes (larger number of *false positives*).

$$Precision = \frac{TruePositives}{TruePositives + FalsePositives}$$

4.2.2 Recall

Recall specifies the fraction of correctly predicted entities over the total number of entities. A high recall indicates that the system is capable of identifying most of the entities that are present in the sample, whereas a low recall score indicates that the system tends to miss a significant number of entities it should have predicted.

$$Recall = \frac{TruePositives}{TruePositives + FalseNegatives}$$

4.2.3 F1

As both precision and recall are very important metrics for evaluating a classification task, the F1 score combines both of them into a single metric and therefore represents the most important metric when evaluating classification models. A high F1 score indicates that the model has a high precision in classified samples, while also keeping the number of misclassified samples low. The F1 score is the harmonic mean of precision and recall.

$$F1 = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall}$$

4.3 Experiments

After researching different state-of-the-art methods for few-shot named entity recognition, it was decided to conduct experiments with ProtoBERT (Ding et al., 2021), StructShot (Yang and Katiyar, 2020), TemplateNER (Cui et al., 2021) and EntLM (R Ma et al., 2022) as all of those systems were reported to produce state-of-the-art few-shot NER results. Furthermore, this selection of systems contains meta-learning approaches and prompt-based approaches and thus delivers the broadest possible insights on what state-of-the-art few-shot NER models are capable of. Experiments were also conducted using adapter-transformer (Pfeiffer, Rücklé, et al., 2020). While they, in their basic form, are not built specifically for few-shot learning, adapters naturally support quick task switching while at the same time being very resource efficient, making them a good fit for a semi-automated annotation system.

As described in Chapter 3, ProtoBERT and StructShot use meta-learning approaches with episodic training. The conducted experiments follow the experiments done by the authors of the Few-NERD dataset and aim to test how well meta-learning approaches perform in a fine-grained few-shot setting. As described in Section 4.1, the authors define two data splits, INTER and INTRA, where each data split is defined by how much knowledge is shared between training

Parent Class	Child Classes	
Art	Broadcastprogram	Film
Building	Hospital	Airport
Event	Battle	Disaster
Location	Bodies of water	GPE
Organization	Company	Education
Other	Astronomything	Award
Person	Actor	Artist/Author
Product	Airplane	Car

Table 4.1: The 16 classes of the Few-NERD dataset that were used in all experiments along with their parent classes

Building						
Hospital	Hotel	Library	Airport	Restaurant	Sportsfacility	Theater

Table 4.2: The 7 classes of the Few-NERD dataset that were used for a fine-grained analysis

and testing episodes. Experiments have been carried out on both the INTER and the INTRA split.

For the two prompt-based systems, as well as for the Adapter setup, two experiments have been designed: a first experiment differentiates between 16 classes in which two of each belong to one coarse-grained category of the Few-NERD dataset. The classes used in this experiment are depicted in Table 4.1. Additionally, to determine the ability of the model to differentiate between fine-grained classes of the same parent category, another experiment was conducted using classes of the parent category "building". The classes used in this experiment are depicted in Table 4.2. To examine the number of samples needed to achieve adequate results, those experiments were done with $N = 10, 20, 50, 100, 500$. For those experiments, the SUP split from the Few-NERD dataset was used (filtered for relevant classes).

4.3.1 Baseline Experiment

A preliminary baseline experiment was conducted using the transformer library by huggingface¹ and the Few-NERD data set (Ding et al., 2021). This experiment was performed to investigate how well a simple fine-tuned BERT model performs on the named entity recognition task with all fine-grained classes of the Few-NERD dataset. For comparison, the model was also tested on the standard coarse-grained NER task. For this, all 66 fine-grained classes in the data set were mapped to their coarse-grained base class, leaving the eight classes "Person", "Art", "Organization", "Product", "Event", "Other", "Location" and "Building". The pre-trained model used was DistillBERT (Sanh et al., 2019). For both experiments, the learning rate was set to $5e-05$ and the batch size was set to 32. The result after three epochs of training are reported in Table 4.3. On the eight parent classes, it achieved an F1 score of 84.5%. In contrast, the DistillBERT that was trained on all 66 fine-grained classes achieved an F1-score of 66.7% (Table 4.3) indicating that the presence of many fine-grained classes poses severe problems to the model in the standard supervised setting.

1. <https://huggingface.co/>

	Precision	Recall	F1
Coarse-grained (8 Classes)	0.84	0.85	0.85
Fine-grained (66 Classes)	0.65	0.69	0.67

Table 4.3: The results of training DistilBERT on the coarse-grained (eight different classes) and on the fine-grained (66 different classes) Few-NERD data set.

4.3.2 ProtoBERT & StructShot

As described in Chapter 3, ProtoBERT and StructShot are trained and tested on episodes, where each episode resembles one few-shot learning task. Following the notation by Ding et al. (2021), in an N way $K \sim 2K$ shot experiment, each episode shows the model N classes, where for each class there are a minimum of K and a maximum of $2K$ examples. The task of the model is then to correctly label the sentences in the query set. The following episodic experiments were conducted: 5-way 1 \sim 2 shot, 5-way 5 \sim 10 shot, and 10-way 1 \sim 2 shot, each in the INTER as well as in the INTRA setting. Since episode size scales with $N \cdot K$, experiments with higher N and K resulted in an out-of-memory error, even for batchsize=1, and could therefore not be conducted.

Implementation Details

For the experiments, the code that was published by Ding et al. (2021) was used.² All models use uncased BERT (Devlin et al., 2019) as the backbone encoder. The hidden size is 768, and the number of layers and heads is 12 (Ding et al., 2021). The models were implemented using huggingface transformer³ and PyTorch⁴. The implementation uses AdamW optimizer⁵, the learning rate was 1e-4 for all experiments. As the episodes can be large, especially for larger N and K , to avoid memory issues during training, the batchsize for all experiments has been set to 1. For all N, K combinations, 10000 episodes were used for training, and 5000 episodes were used for testing. The hyperparameter τ for the Viterbi decoding in StructShot was taken from the original authors who set it to 0.434 for 5 \sim 10 shot setting and 0.32 for 1 \sim 2 shot setting. The labeling schema used was IO as the original authors found it to perform better than BIO.

Results

The results reported in Ding et al. (2021) could be reproduced approximately. They are reported in Table 4.4 and 4.5: For lower K (numbers of examples per class), both ProtoBERT and StructShot achieved comparably good results and the inference and training time is quite fast because of the simplicity of the algorithms. As expected, the INTER episode split performs better than the INTRA episode split, because of the knowledge of the parent classes being shared through all splits. However, the episodic learning nature of these models limits the possibility to assess how well they deal with larger K and larger N , and therefore comparability with the

2. <https://github.com/thunlp/Few-NERD>

3. <https://github.com/huggingface/transformer>

4. <https://pytorch.org>

5. <https://www.fast.ai/2018/07/02/adam-weight-decay/adamw>

Model	INTER								
	5 way 1~2 shot			5 way 5~10 shot			10 way 1~2 shot		
	P	R	F1	P	R	F1	P	R	F1
ProtoBERT	0.32	0.48	0.38	0.49	0.63	0.55	0.26	0.42	0.32
Structshot	0.53	0.51	0.52	0.46	0.32	0.38	0.45	0.4	0.42

Table 4.4: Results of ProtoBERT and Structshot experiments using INTER setting

Model	INTRA								
	5 way 1~2 shot			5 way 5~10 shot			10 way 1~2 shot		
	P	R	F1	P	R	F1	P	R	F1
ProtoBERT	0.12	0.25	0.16	0.36	0.50	0.42	0.10	0.19	0.13
Structshot	0.30	0.25	0.27	0.46	0.32	0.38	0.26	0.17	0.21

Table 4.5: Results of ProtoBERT and Structshot experiments using INTRA setting

following experiments is only partially given. The experiments showed, though, that ProtoBERT and StructShot are able to perform comparably well for low K while at the same time being very efficient both in training and inference.

4.3.3 TemplateNER

As described in Chapter 3, TemplateNER (Cui et al., 2021) represents a prompt-based approach to named entity recognition. The original authors conducted experiments using the MIT Movie (J Liu et al., 2013), MIT Restaurant (J Liu et al., 2013), and ATIS dataset (Hemphill et al., 1990), both in a resource transfer setting in which the BART model was first fine-tuned on CoNNL2003 and without prior fine-tuning. In order to explore the capabilities of TemplateNER in a more fine-grained few-shot setting and in order to compare the results to the other systems, experiments were conducted using the Few-NERD dataset (Ding et al., 2021).

Implementation Details

For the experiments with TemplateNER, the code⁶ that was published by Cui et al. (2021) was used and modified in order for the experiments to be done on the Few-NERD dataset. As described in Chapter 3, TemplateNER is based on a generative model (BART), that is trained on sentences in the form of

<sentence>. X is a Y entity

As stated in the original paper, other templates performed comparably or worse so the experiments on TemplateNER were only conducted with the template above. To run the experiments with the Few-NERD dataset, the dataset first had to be transformed to the required input scheme. Additionally, parts of the code for inference had to be modified, since the corresponding templates have to be hardcoded in order to evaluate the output. The authors implemented their

6. <https://github.com/thunlp/Few-NERD>

model using PyTorch and huggingface transformer. As generative model, the *bart-large*⁷ model was loaded from the huggingface hub. For the learning rate, $4e-5$ was used, with a warmup ratio of 0.06. To avoid overfitting, early stopping was used with an early stopping delta set to 0 and early stopping patience set to 3. The batch size in training was set to 32. The evaluation was done on 10000 sentences taken from the Few-NERD test split.

Table 4.6: Experiments on TemplateNER with and without prior fine-tuning

K=	P	R	F1		K=	P	R	F1
10	0.42	0.62	0.47		10	0.45	0.83	0.58
20	0.48	0.7	0.55		500	0.53	0.83	0.62
50	0.51	0.78	0.60					
500	0.51	0.81	0.61	(a) The results of TemplateNER on the 16-class experiment without prior fine-tuning				
								(b) The results of running the 16-class experiment on a BART model that has been fine-tuned on the eight parent categories of the Few-NERD dataset for $K = 10$ and $K = 500$

Results

Using TemplateNER (Cui et al., 2021), the 16-class experiment, as well as the 7-class fine-grained experiment were conducted as described at the beginning of this chapter. The results can be seen in Table 4.6a, Table 4.7, and Figure 4.3. It can be observed that the average F1 score starts for $K = 10$ at 0.47 but then improves only by 14% to 0.61 when using $K = 500$ samples, indicating that more samples will likely not lead to a substantial improvement in F1 score. For the 7-class fine-grained experiment, the results are similar to the 16-class experiment. The F1 score starts at 0.53 for $K = 10$ and improves by 7% for $K = 500$.

As the authors in the original paper suggested, prior fine-tuning on a larger dataset can improve the predictions of the model. To confirm this, the 16-class experiment was also executed on a BART model that has been finetuned on the eight parent classes of Few-NERD before the actual few-shot task. The dataset used for fine-tuning was derived from the training split of the original Few-NERD dataset, where all classes were mapped to their respective parent classes and the sentences used for few-shot learning were removed. As shown in Table 4.6b the prior fine-tuning achieved an improvement of 11% for $K = 10$. However, for $K = 500$ the prior fine-tuning only led to an improvement of 1%, indicating again that the F1 score for 16 classes might converge in the $F1 = 0.6$ region.

7. <https://huggingface.co/facebook/bart-large>

K=	P	R	F1
10	0.39	0.82	0.53
20	0.44	0.86	0.58
50	0.44	0.9	0.59
500	0.5	0.9	0.6

Table 4.7: Results for the 7-class fine-grained experiment on TemplateNER

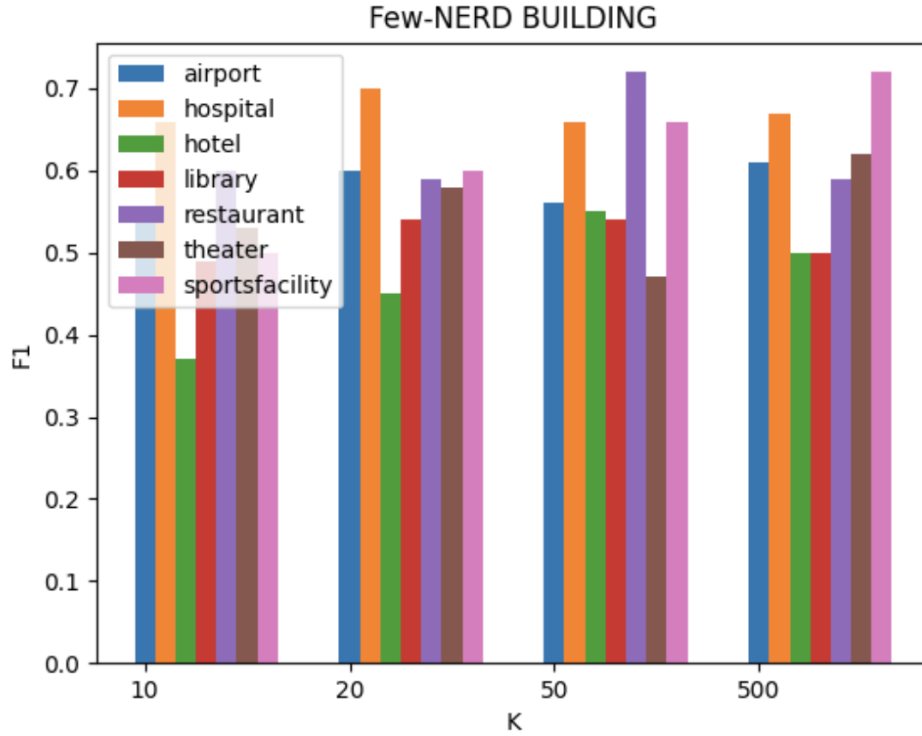


Figure 4.3: Class-specific F1 scores for the fine-grained experiments on TemplateNER.

It is interesting to note that all models achieved a comparably high recall score, while delivering a fairly low precision score, indicating that the model tends to overpredict certain classes. A factor that might influence this is the number of available "non-entities" in the training sentences. Following the original paper, the number of non-entity-type sentences, e.g. templates of the form of

`<sentence>. X is not a named entity`

was chosen to be 1.5 times the number of entity-type sentences. However, the high recall score indicates that the model tends to predict more entities than present in the actual text. To counteract this behavior, future experiments could choose the number of non-entity-type sentences to be larger.

An additional observation is, that experiments that used the original Few-NERD label structure of `<parentclass-childclass>` delivered worse results than experiments that were conducted using only the child class as the label.

While the results, especially for lower K look quite promising, a big disadvantage of this system is the time that is needed for inference. The reason for this is that at inference time, it has to iterate over every n -gram of the sentence in question to get the corresponding probabilities. For example, if we want the model to annotate entities in the sentence "I traveled to Japan last fall to see a jazz concert", for every possible n -gram in the sentence (i.e. "I", "I traveled", "I traveled to", ..., "traveled", "traveled to", ..., "to", "to Japan", ..., "Japan", "Japan last", ... "I traveled to", "traveled to Japan", "to Japan last", ..., and so forth) the corresponding template needs to be filled out for every class. This means, that for inference of this sentence (which has 66 n -grams, $n = 1, \dots, T$, where T is the number of words), the model needs to calculate the possibility for

$66 * N$ templates, where N is the number of classes, which especially means that the inference time increases significantly with the number of classes. Even though the authors capped n at 8, predicting entities in 10000 sample sentences took over three hours, making this system not very scalable with regard to annotating large text corpora.

4.3.4 EntLM

The second prompt-based approach with which experiments were conducted was EntLM (R Ma et al., 2022). As described in Chapter 3, EntLM utilizes the BERT pre-training task for predicting label representatives at entity positions, thus omitting the need for costly n -gram iterations during inference. Similar to TemplateNER, the original authors conducted experiments with ConNNL2003, OntoNotes5, and MIT Movie. To compare the few-shot capabilities of EntLM to the other systems mentioned in this thesis, both the 16-class experiment and the 7-class fine-grained experiment (as described above) were conducted on the Few-NERD dataset.

Implementation Details

R Ma et al. (2022) published the code for executing experiments with EntLM [github](https://github.com/rtmaww/EntLM)⁸. However, the original authors only conducted experiments using the CoNNL2003 and OntoNotes5 datasets, which primarily means that the label representatives (as described in Chapter 4) for the Few-NERD dataset are not available in the original repository. To receive the label representatives, the original authors adopted BOND (Liang et al., 2020), an algorithm for distantly annotating large datasets with the help of a knowledge base. Instead of adopting the BOND algorithm on the Few-NERD dataset, for this experiment, the test split of the Few-NERD dataset was used to receive the label map for Few-NERD labels, using the provided scripts of the original authors. While this approach is obviously less noisy than a distantly supervised algorithm and doesn't mimic a practical setting, for the scope of this series of experiments it was seen as sufficient, especially since the authors stated in their paper that the quality of the distantly annotated corpus doesn't affect the outcome of their algorithm in a serious manner. This is mainly because the distantly annotated data is only used as an indicator of the data distribution and the model is not trained directly on it (R Ma et al., 2022). The model was implemented using huggingface transformer and PyTorch, the learning rate was set to $1e-4$ for all experiments. The batch size was set to 4. The model was trained for 20 epochs for K up to 50 and for 50 epochs for K larger than 50, using AdamW optimizer. The labeling scheme adopted was the IO scheme. For the label word selection algorithms, the hyperparameter k for selecting the top k high-frequency words was set to $k = 6$, as suggested by the authors (R Ma et al., 2022).

Results

The results of the experiments can be seen in Table 4.8a and 4.8b. For the 16-class experiment, the results start at a F1 value of 0.24 for $K=10$ and go up to $F1=0.56$ for $K=500$. The results for the fine-grained experiment on the parent class BUILDING start at an F1 of 0.045 for $K=10$ and build up to an F1 of 0.21 for $K=500$. It is conspicuous that for the fine-grained experiment, the precision score only grows by $\sim 10\%$ from $K=10$ to $K=500$, while the recall score grows by $\sim 63\%$, indicating that the model tends to predict too many entities. Furthermore, in Figure 4.4 it can be seen that some fine-grained classes generally perform better than others. For example,

8. <https://github.com/rtmaww/EntLM>

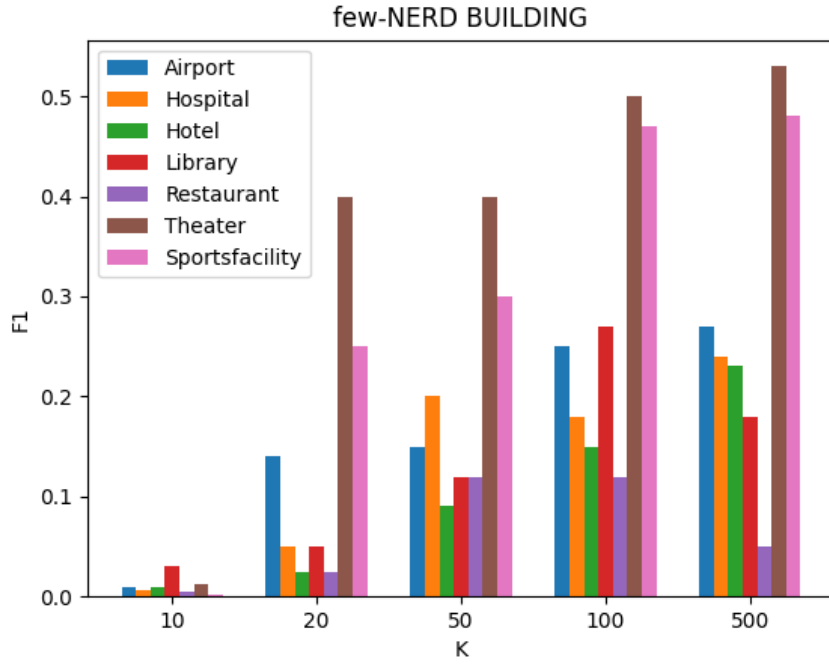


Figure 4.4: The F1 values of the BUILDING classes for different K for EntLM.

the classes "Theater" and "Sportsfacility" perform well for $K = 20$ to $K = 500$ while the class Restaurant performs comparably poorly, even for $K = 500$. To counter this, further engineering of the label words might prove useful.

In contrast to TemplateNER, EntLM has a big advantage regarding the inference time. Because the model only predicts a label word at the entity position, the whole sentence can be processed at one time and iterating over all potential entities is not needed. However, the results of the original paper could not be reproduced for the Few-NERD dataset. Especially at lower K EntLM gets outperformed by other few-shot techniques such as ProtoBERT or TemplateNER. Furthermore, EntLM requires sophisticated methods to gather adequate label words that the model predicts for each entity type, making it unfit for an annotation system, in which classes to annotate might change in a more rapid fashion.

Table 4.8: The results for both experiments on EntLM

K=	P	R	F1	K=	P	R	F1
10	0.3	0.2	0.24	10	0.025	0.105	0.045
20	0.3	0.3	0.3	20	0.047	0.366	0.084
50	0.39	0.34	0.36	50	0.123	0.524	0.199
100	0.46	0.52	0.49	100	0.171	0.536	0.260
500	0.51	0.61	0.56	500	0.122	0.730	0.21

(a) The results of the 16-class experiments on EntLM

(b) The results of the fine-grained Few-NERD BUILDING experiment on EntLM

4.3.5 Adapter

The last system that was explored for the few-shot NER setting are adapter-transformer. Adapter-transformer are a modification of the standard transformer architecture, aiming to make fine-tuning of large transformer models more efficient and flexible. With the goal of an automated annotation system in mind, adapters have the advantage of providing task-specific layers that adapt to one shared pre-trained model, allowing for efficiently storing different adapters for different annotation tasks, without having to store whole fine-tuned transformer models for every task. Experiments were conducted using the Few-NERD dataset. For comparability with the other systems, the 16-class experiment and the fine-grained experiment were conducted. Additionally, because of the good overall performance of adapter-transformer, experiments were conducted with 32-classes, 48-classes and 66-classes from Few-NERD, each for $K = 10, 20, 50, 100, 500$. Besides that, experiments were also conducted using $K = 1000$. In this setting, however, it could not be guaranteed to have exactly 1000 examples of every class. This results from the fact that samples are always sentence-based and a sentence usually includes more than one entity, which makes the sampling of exactly 1000 examples per class not feasible, especially for a larger number of classes N . To determine the ability to transfer knowledge from prior fine-tuning, in an additional experiment, a BERT model that was trained on the eight coarse-grained classes of Few-NERD was used as the base model of the adapter.

Implementation Details

The code for the Adapter experiments was written using huggingface transformer, adapter-transformer⁹, and PyTorch. As pre-trained transformer model, huggingfaces' *bert-base-cased*¹⁰ was used. For all experiments, the learning rate was set to $1e-4$ and the batch size was set to 8. The number of epochs was set to 20. No overfitting could be observed at this number of epochs. For adapter configurations, the configuration modes `pfeiffer`, `parallel` and `lora` were tested, where `parallel` seemed to perform slightly better than the other configurations and therefore was set as the configuration for all reported experiments. The models were tested on the test split of the Few-NERD dataset (edited to include only the classes that it was trained on). As labeling scheme, BIO was used as it produced slightly better results in most cases, in contrast to the findings of Ding et al. (2021).

Table 4.9: The results of both experiments for adapter-transformer

K=	P	R	F1		K=	P	R	F1
10	0.19	0.21	0.20		10	0.26	0.29	0.27
20	0.32	0.26	0.28		20	0.36	0.39	0.37
50	0.59	0.54	0.57		50	0.57	0.58	0.57
100	0.61	0.62	0.62		100	0.59	0.62	0.61
500	0.68	0.7	0.69		500	0.67	0.68	0.67
>500	0.72	0.7	0.71					

(a) The results of the 16-class experiments on adapter-transformer

(b) The results of the fine-grained Few-NERD BUILDING experiment on adapter-transformer

9. <https://docs.adapterhub.ml/>

10. <https://huggingface.co/bert-base-cased>

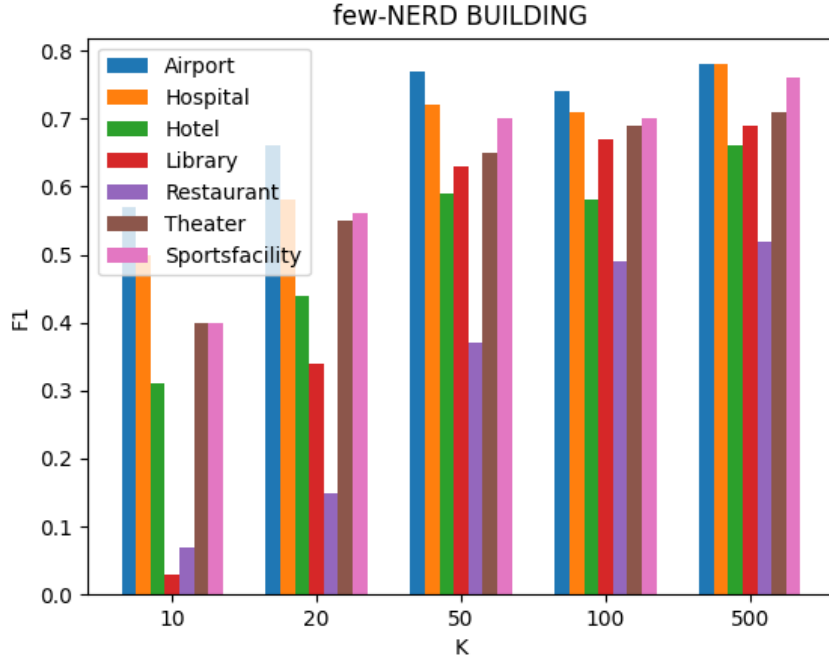


Figure 4.5: The F1 values of the BUILDING classes for different K for adapter-transformer

Results

The results of the experiment on adapter-transformer can be seen in Table 4.9a and 4.9b. For the 16-class experiment, the F1 score starts at 0.2 for $K = 10$ and moves up to 0.7 for $K = 500$. For $K > 500$ (with the limitations described above), the average F1 score improves only by 4 points, indicating that it will converge somewhere around 0.7. For the fine-grained experiment, the F1 score for $K = 10$ starts at 0.26 and moves to 0.67 for $K = 500$. The biggest improvement in F1 score can be observed going from $K = 20$ to $K = 50$, with a jump from 0.28 to 0.57 with only 30 additional samples per class, while adding another 450 samples per class leads only to an increase of 10 points. This holds true for the 16-class experiment, as well as for the fine-grained experiment. Looking at the per-class F1 score for the experiments done with adapter-transformer (Figure 4.5), it can be seen that for $K = 10$, some classes already achieve comparably high F1 scores, while some classes do not. However, for larger K , those imbalances start to diminish. Table 4.12 shows the results of further experiments with adapter-transformer, in which the model should gradually differentiate between more classes. It can be seen that the results of those experiments look approximately the same for all N , indicating that the number of classes is not the bottleneck for those models. The model performs only 14% worse when it differentiates between 66 classes than when it differentiates between 16 classes. A last experiment was conducted in order to determine the effect of prior fine-tuning of the adapter base model. For this, the BERT model used as the base for the Adapter experiments was trained on the eight parent classes of the Few-NERD dataset before conducting the few-shot experiments with the fine-grained classes using adapters. In order to avoid the model observing the same sentences during prior fine-tuning and few-shot training, sentences that were used for sampling few-shot data have been removed. Figure 4.6 shows the results of this experiment. It can be seen that using a transformer model that has been fine-tuned on NER data of the same domain as the base model can improve the performance of the adapter, especially for lower K .

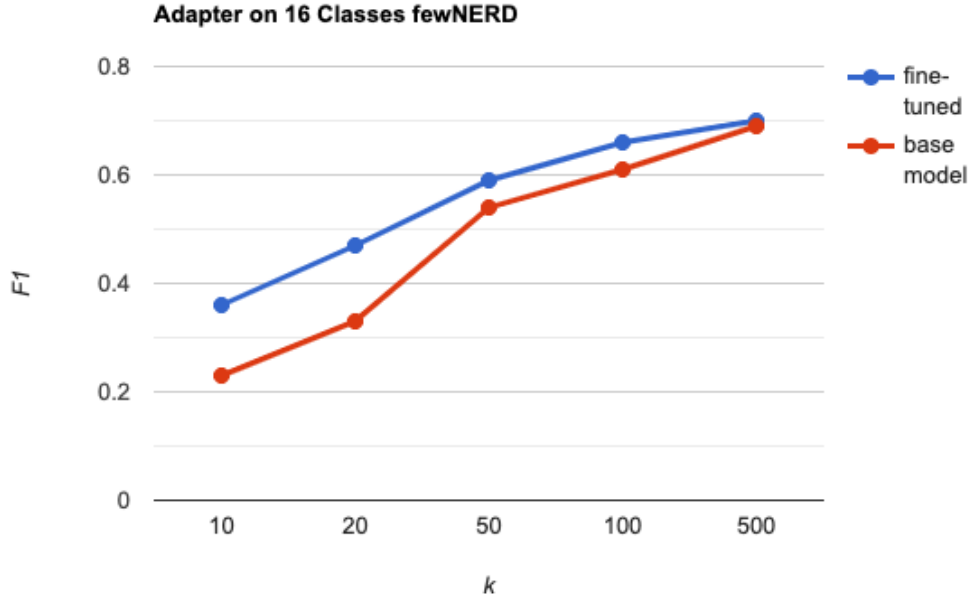


Figure 4.6: F1 values of the 16-class adapter experiments, both for the base model and prior fine-tuned model

4.4 Summary

This chapter described the named entity recognition experiments that were done on ProtoBERT & Structshot (Ding et al., 2021), TemplateNER (R Ma et al., 2022), EntLM (R Ma et al., 2022) and adapter-transformer (Pfeiffer, Rücklé, et al., 2020). All experiments were conducted using the Few-NERD dataset (Ding et al., 2021). For ProtoBERT & Structshot episodic learning was conducted using the training episodes of the original authors. For the other systems, two experiments were designed using the Few-NERD dataset. The first experiment includes 16 classes, where each of the two classes belongs to one parent category of Few-NERD. To determine the ability of the model to differentiate between labels that are semantically closer together, the second experiment includes 7 classes, where all classes belong to the same parent category. The results of the experiments on TemplateNER, EntLM, and Adapters are summarized in Table 4.10 and 4.11.

4.4.1 ProtoBERT & StructShot

On ProtoBERT and StructShot, episodic learning experiments were conducted, following the original implementation of Ding et al. (2021). For ProtoBERT and StructShot, the results of the original paper could be reproduced approximately. The experiments demonstrated, that, by using meta-learning, both models were able to predict new classes based on only up to ten examples per class. On the INTER split (where some knowledge is shared between training and testing), StructShot, for example, achieved an F1 score of 52% in a 5-class setting using only up to 2 examples per class. As expected, on the INTRA split (that shares little to no knowledge between train and test data), both models performed worse. However, while the results were

good, it is not clear how both models perform for gradually larger N and K , because evaluation on larger N and K was not possible in a straightforward manner due to memory issues.

4.4.2 TemplateNER

The first prompt-based system, TemplateNER, performed comparably well on both 16 classes and 7 fine-grained classes, with an F1 score of around 50% for $K = 10$ examples per class. However, the score only improved for another 10% for $K = 500$ examples per class, indicating that this system is usable in the few-shot setting but does not benefit much from more training data. An additional experiment confirmed the findings of Cui et al. (2021), that the few-shot ability of TemplateNER does benefit from prior fine-tuning on a resource-rich NER dataset (in this case the eight parent classes of Few-NERD). Throughout all experiments, TemplateNER showed imbalances between precision and recall with the recall being much higher in all cases. To counteract this, the ratio of entity vs. non-entity samples in the training data could be adjusted. TemplateNER achieved good results, especially for lower examples per class. However, at inference time, the system requires to iterate over all possible n-grams of the sentence, which results in high inference time. Using such a model in a semi-automatic annotation system therefore might prove impractical for larger text corpora.

4.4.3 EntLM

The second prompt-based system, EntLM, achieved comparable results to TemplateNER for $K = 500$ examples per class on the 16-class experiment. However, for lower K the results are much worse (TemplateNER achieved around 50% F1 score, while EntLM achieved around 25% F1 score for $K = 10$ examples per class). This tendency continues for the 7-class fine-grained experiment, where EntLM achieved only 0.4% F1 score for $K = 10$ examples per class and 21% for $K = 500$ examples per class, indicating that classes that are semantically closer together pose severe problems for the system. This and the fact that EntLM requires an additional algorithm for finding representative label words makes this system unfit for the purpose of a semi-automated annotation system, where different, finer-grained classes can be expected to occur frequently.

4.4.4 Adapter

The last analyzed system were adapter-transformer (as described in Chapter 3). In the 16-class experiment, adapter achieved an F1 score of 20% for $K = 10$ examples per class. However, it could be shown that prior fine-tuning of the base transformer model on a resource-rich NER dataset improves the few-shot capability of adapters for lower K from 20 to 40% F1 score, almost putting them on par with the good results of TemplateNER. The results of the 7-class fine-grained experiments are comparable to the results of the 16-class experiment in terms of F1 score, both achieved around 20% for $K = 10$ examples per class and around 70% for $K = 500$ examples per class. To determine if more examples per class will lead to a higher F1 score, additional experiments were conducted using up to 1000 examples per class. It could be shown that a larger K only improves the performance marginally. Moreover, the effect of prior fine-tuning of the base model diminishes with a larger K . While for $K = 10$ examples, prior fine-tuning leads to double the F1 score compared to no prior fine-tuning, for $K > 500$ examples the effect of prior fine-tuning is not noticeable at all, indicating that for the 16-class

Model	K=				
	10	20	50	100	500
TemplateNER	0.47 (0.58)	0.55	0.6	-	0.61 (0.62)
EntLM	0.24	0.3	0.36	0.49	0.56
Adapter	0.2 (0.39)	0.28 (0.49)	0.57 (0.60)	0.61 (0.63)	0.69 (0.69)

Table 4.10: Results (F1) of TemplateNER, EntLM, and Adapter models on the 16-class experiment. Results in brackets denote the results for the same experiment with a prior fine-tuned base model

Model	K=				
	10	20	50	100	500
TemplateNER	0.53	0.58	0.59	-	0.6
EntLM	0.05	0.08	0.20	0.26	0.21
Adapter	0.27	0.37	0.57	0.61	0.67

Table 4.11: Results (F1) of TemplateNER, EntLM, and Adapter models on the fine-grained experiment

experiment, 70% F1 score represents indeed a limit for adapters. Finally, experiments were conducted with 32 classes, 48 classes, and 66 classes for the adapter to differentiate between. The results of those experiments are comparable to the results of the initial 16-class experiment. Out of the five experiments done, adapter-transformer seem to be the most flexible option for the purpose of a semi-automatic annotation system: They are fast at inference and training and are inherently built for efficiently switching task contexts without the need of saving a whole transformer model for each task. Furthermore, they are able to achieve comparably good results while differentiating between a large number of classes. Since they are not specifically built for a few-shot learning setting, adapter-transformer fall behind in performance for lower K when being compared, for example, to ProtoBERT or TemplateNER. However, it has been shown, that prior fine-tuning of the adapter base model on a resource-rich domain can alleviate this issue.

K=	P	R	F1	K=	P	R	F1	K=	P	R	F1
10	0.24	0.20	0.22	10	0.20	0.25	0.22	10	0.24	0.30	0.27
20	0.35	0.32	0.33	20	0.29	0.33	0.31	20	0.31	0.37	0.34
50	0.54	0.52	0.53	50	0.45	0.49	0.47	50	0.45	0.49	0.47
100	0.57	0.58	0.57	100	0.49	0.51	0.5	100	0.47	0.51	0.49
500	0.64	0.65	0.64	500	0.56	0.57	0.57	500	0.54	0.57	0.57

(a) 32-class experiment (b) 48-class experiment (c) 66-class experiment

Table 4.12: Additional experiments with adapter-transformer

5 Implementation

During the course of this thesis, a functional prototype of an automatic annotation system has been implemented. This chapter describes the functionality of the implemented application, as well as implementation and architectural details.

5.1 Functional Overview

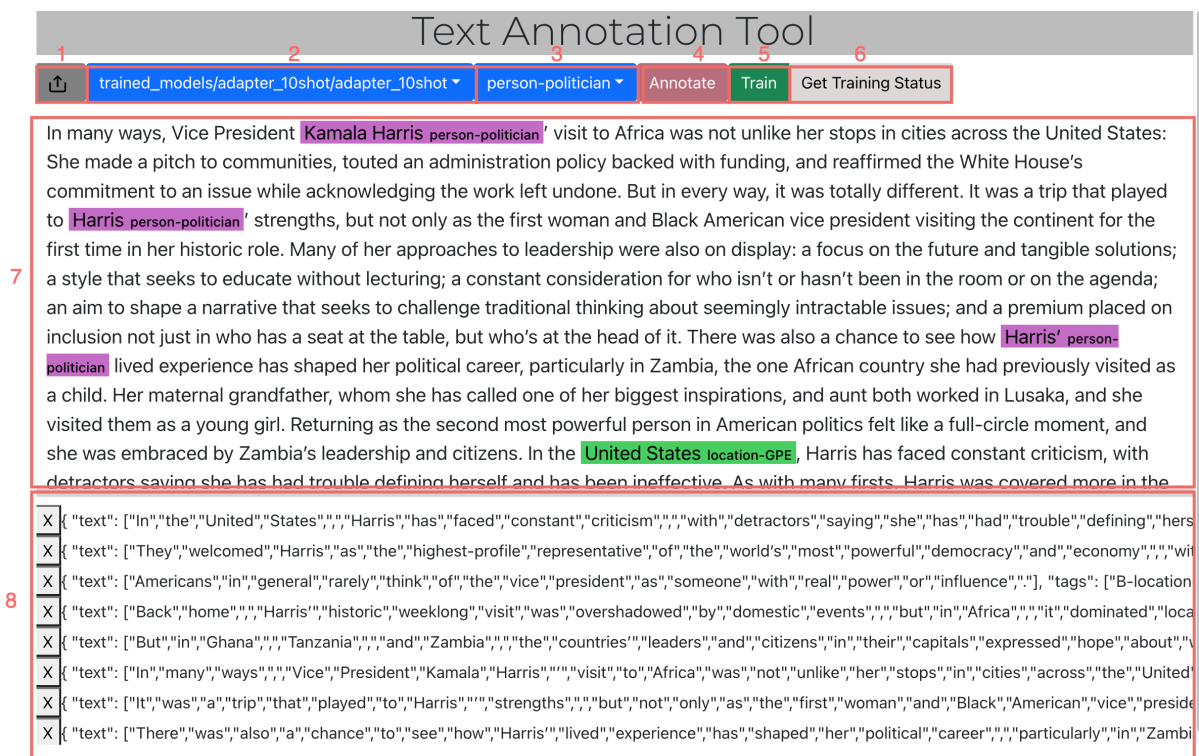


Figure 5.1: Overview of the annotation system. The Interface consists of the Upload Document Button (1), Choose Model Dropdown (2), Choose Tag Dropdown (3), Annotate Button (4), Train Button (5), Get Training Status Button (6), as well as the Annotation Area (7) and the Training Data Staging Area (8)

Figure 5.1 shows an overview of the implemented application. The application is a web-based annotation tool, that lets the user upload a text document and annotate it using pre-defined categories. The annotated sentences are collected as training data for a NER model that runs on the backend and can be used to train that model. The trained model can then be used for inference, effectively extending the user-made annotations to the whole document. When the automatic annotation process is finished, the user has the possibility to correct the annotations that were wrong and send the corrected sentences as training data to the model again. In the following, the core features of the application are described from a user's perspective.

5.1.1 Upload a Document

At startup, the only available button is the *Upload Document* Button (1 in Figure 5.1). A click opens a prompt for the user to select the document to annotate (Figure 5.2).

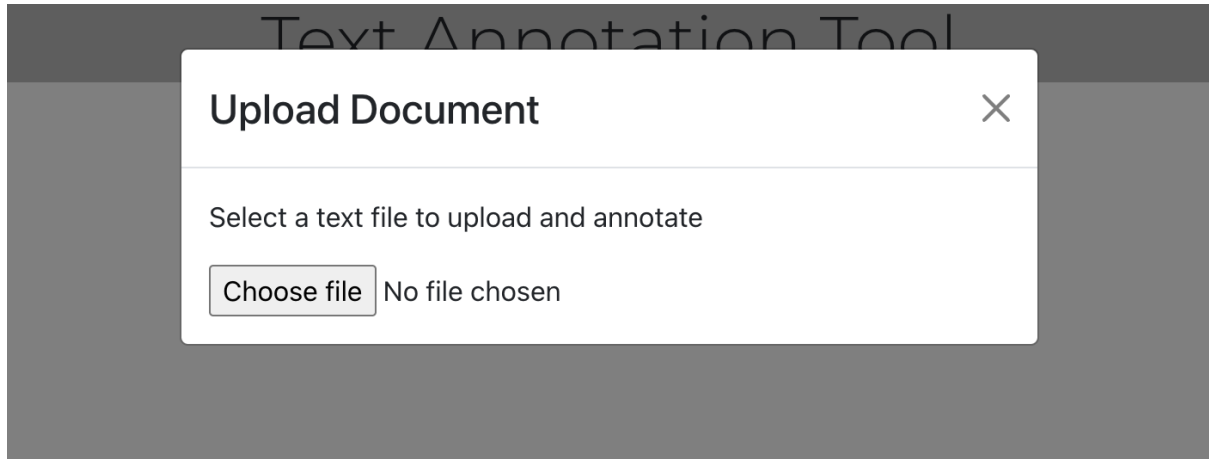


Figure 5.2: The *Upload Document* dialogue

After selecting the document to annotate, the main application screen opens. The main application screen consists of the *Annotation Area* (7), the buttons *Annotate* (4), *Train* (5), *Get Training Status* (6), two dropdown menus (2 and 3) and the *Training Data Staging Area* (8)

5.1.2 Selecting a Model

The dropdown menu *Select a Model* (2) offers the user the choice to select a trained model that will be used for the annotation task or to create a new model from scratch (Figure 5.3)

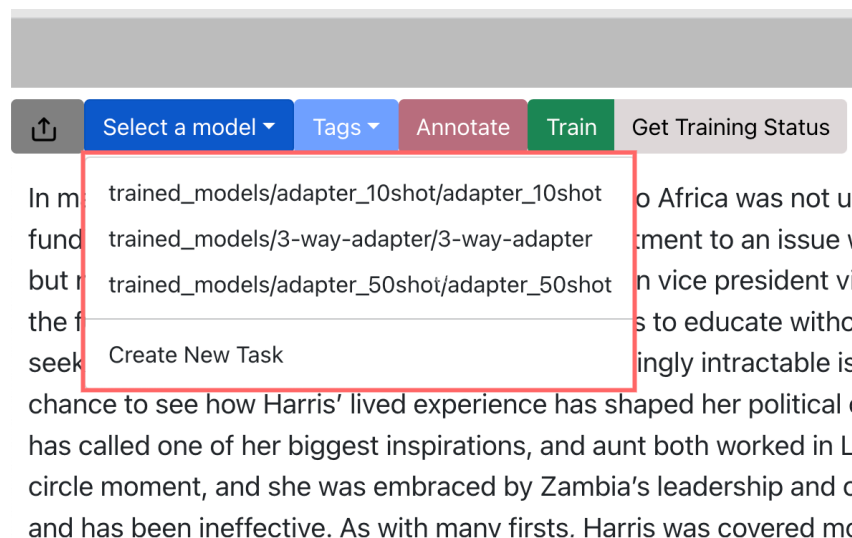
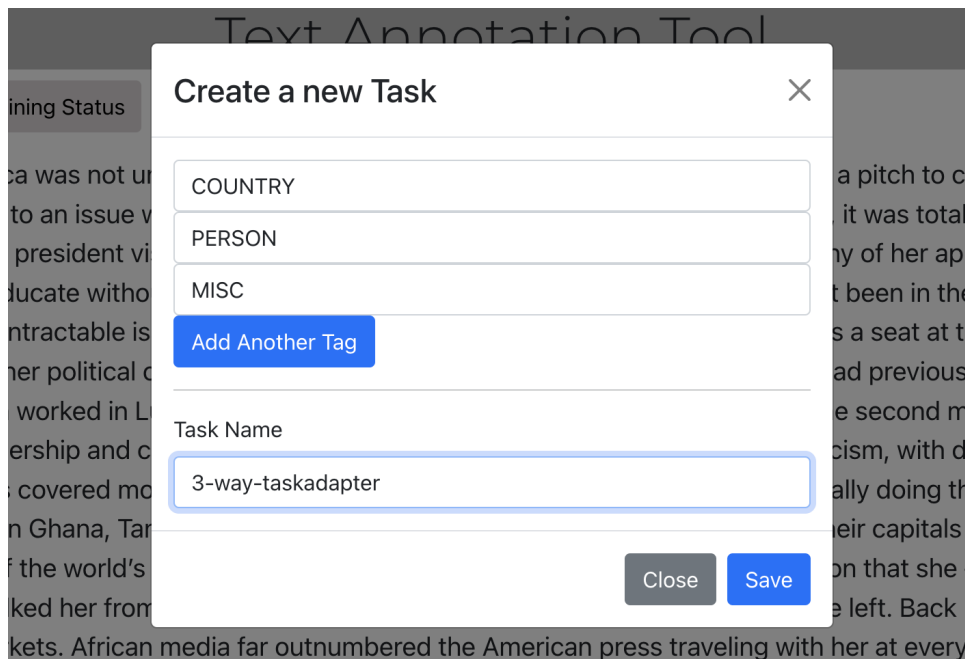


Figure 5.3: The *Select Model* dropdown, where the user can pick a saved model to use for annotation

5.1.3 Creating a new Task

If the user wants to create a model for a new task, she can click the *Create New Task* option. This will trigger a dialogue to open, in which the user can enter a name for the new task. Furthermore, the user can enter the tags which they want the model to differentiate (Figure 5.5)



The image shows a 'Create a new Task' dialog box overlaid on a background of text. The dialog box has a title bar with a close button (X). It contains three input fields for tags: 'COUNTRY', 'PERSON', and 'MISC'. Below these is a blue button labeled 'Add Another Tag'. There is a 'Task Name' label followed by an input field containing the text '3-way-taskadapter'. At the bottom right are 'Close' and 'Save' buttons.

Figure 5.4: Dialogue for creating a new model;The user can enter the desired tags as well as the task name

5.1.4 Selecting Tags

After selecting the desired model, or after creating a new model with a new set of tags, the second dropdown menu becomes clickable. Here, the user can select specific tags in order to annotate the document. The selectable tags reflect the tags that the model was trained on (in case the user picked a previously saved model) or the tags that the user entered during the *Create New Task* dialogue. The user can select portions of the text to annotate by just clicking, holding the left mouse button and dragging over the desired area. The system then marks this portion of the text in a predefined color and writes the corresponding label name next to the annotated word.

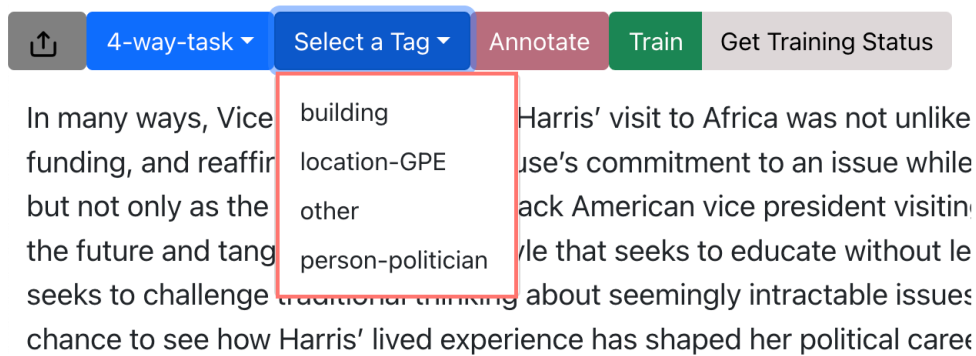


Figure 5.5: The *Select Tag* dropdown menu; Here, the user can select specific tags to annotate the document with

5.1.5 Annotation

The button *Annotate* (4) will trigger the selected model to annotate the text in the Annotation Area (7) (Figure 5.6).

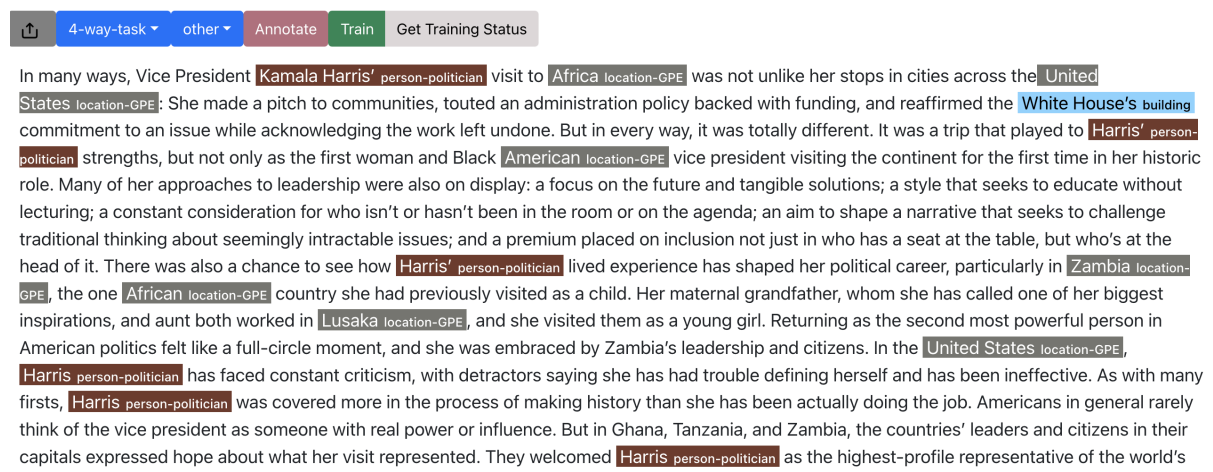


Figure 5.6: The Annotation Area, after a click on the *Annotate Button*

5.1.6 Training

The button *Train* will send training data - which the user has collected by manually annotating the text - to the model in order to train it on the new data. By looking at the annotation area only, it is not clear to the user, what annotations were made by the model during the last iteration and what annotations were made by the user in order to collect training data for the next iteration. Several solutions to overcome this mismatch are imaginable: System annotations could, for example, be identified using a different color scheme, a different font, or be marked with some other kind of symbol. In the spirit of a simple and functional prototype, this issue was solved with the *Training Data Staging Area* at the bottom of the page that just shows the currently collected training data in a verbatim-style manner (Figure 5.8). Each entry in the Training Data Staging Area refers to one training sample that is sent to the model in the next iteration. The user can add data to the Training Data Staging Area by annotating some text in the Annotation Area.

If a sentence contains multiple annotations, the software recognizes this and merges them into one training sample. Similarly, if the user removes one annotation in the Annotation Area but the sentence has more annotations, then the affected training sample is updated. The user can un-stage training samples by removing all annotations of that specific sentence in the Annotation Area, or by clicking on the "X" to the left of each entry in the Training Data Staging Area.

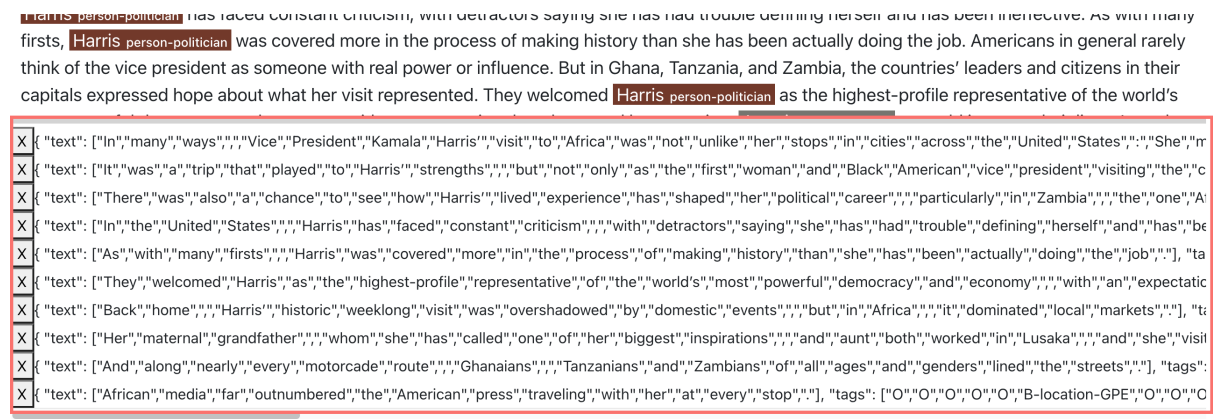


Figure 5.7: The Training Data Staging Area

5.1.7 Get Training Status

Finally, it is important for the user to be able to query the current training status in order to see if the next training iteration can be started, or if any errors occurred during training. This functionality is implemented with the button *Get Training Status*. This button will become active after initializing the first training iteration. It will query the current training status and displays a corresponding message to the user in the form of a simple alert window.

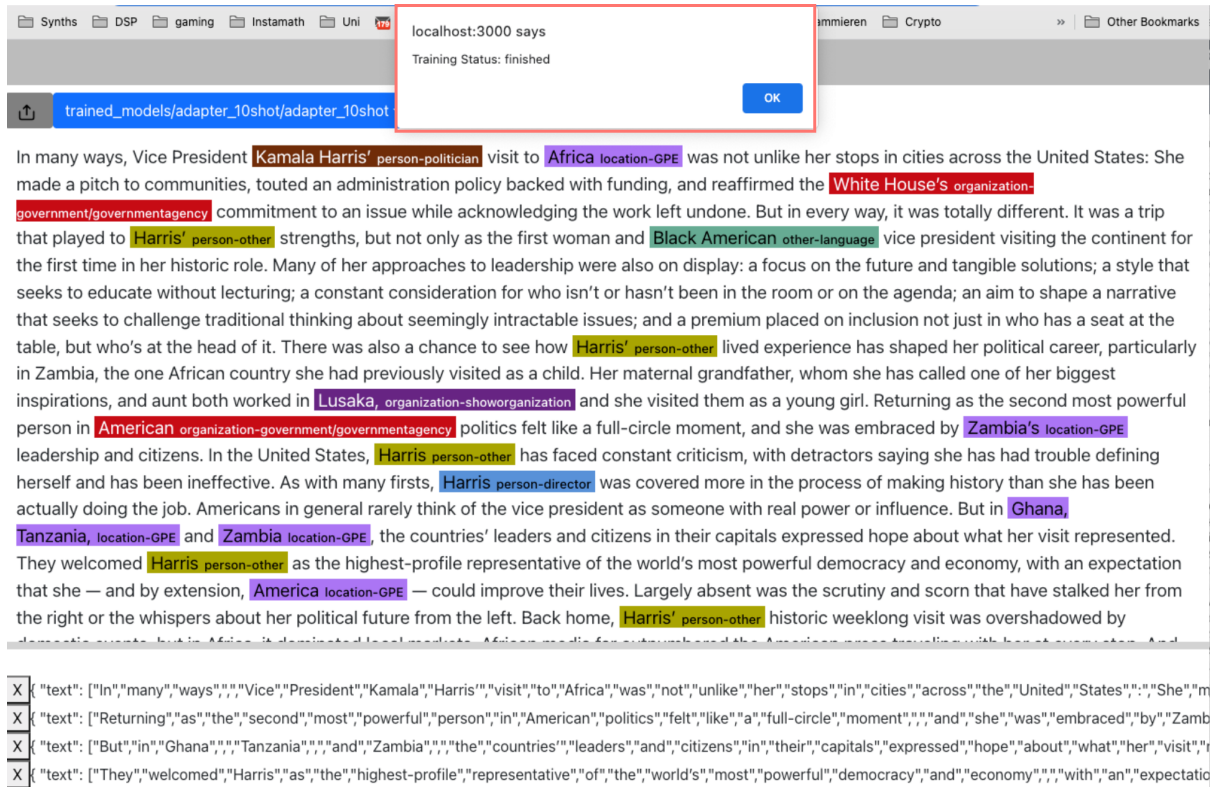


Figure 5.8: A click on the button *Get Training Status* queries the current training status and informs the user via an alert box

5.2 Implementation Details

To provide the reader with a basic understanding of how the application works internally, this section introduces some implementational details of both front end and back end. For the front end, the used library, its interface, and the most important components and functions are explained. The section about the back end explains how the model is managed and how training can be initiated in an asynchronous manner. Also, the API is outlined.

5.2.1 Front End

The front end was created using React 18.2¹. For the Annotation Area, the library *react-text-annotate*² was used. Next to providing the user interface, the most important job of the front end is to provide the annotation functionality via the component offered by the library. This includes keeping track of what annotations are currently active, processing them, and mapping them to correct training data that can be sent to the back end. Similarly, it also includes mapping the received annotation data from the back end (during inference) to an annotation representation. In the following, the most important components of the front end are introduced.

1. <https://react.dev/>

2. <https://mcamac.github.io/react-text-annotate/>

React-Text-Annotate Library

The library *react-text-annotate* provides an interface for basic text annotation via the `TextAnnotate` component and is initialized with the actual text object to be annotated. It represents tags of a document as a list of `AnnotateTag` objects. Each `AnnotateTag` object has the fields `start`, `end`, `text`, `tag` and `color`, where `start` and `end` denote the start and end (character-level) indices of the specific tag, `text` denotes the word(s) being annotated, `tag` denotes the actual label word, and `color` denotes the hexadecimal background color value of the annotation. A callback function `handleChange(value: AnnotateTag[])` is passed to the `TextAnnotate` component which is invoked each time the user annotates a text portion (i.e the `AnnotateTag[]` list is modified). The parameter of this callback function is the new list of tags.

```
[
  {
    "start": 0,
    "end": 13,
    "text": "Kamala Harris",
    "tag": "person-politician",
    "color": "#4b46cd"
  },
  {
    "start": 37,
    "end": 45,
    "text": "America",
    "tag": "location-GPE",
    "color": "#42f5f5"
  },
  {
    "start": 445,
    "end": 451,
    "text": "Japan",
    "tag": "location-GPE",
    "color": "#4b46cd"
  }
]
```

Figure 5.9: Example of how tags for a document are stored internally; *start* and *end* denote the character-level beginning and end indices of the corresponding tag, respectively

Important Components

The front end consists of two main components, the *MainPage* and the *AnnoDocument*.

The *MainPage* is the main component and includes most state variables of the application, like chosen labels, chosen models, training data, tag objects, and different flags. It also implements the user interface and handles all communication with the back end. Notable functions are:

- *handleFileUpload()*

This function is invoked when the user has selected a document. Because training and inference are carried out on a sentence basis, it sends the whole document as a string to

the back end, where it is split into sentences and sent back. The sentences then are passed to the *AnnoDocument* component for further processing.

- *handleInference()*

A call of this function triggers the actual annotation process. It sends the sentences to the back end, where they are handed to the model for inference. The annotated sentences are, once received, converted back into the *AnnotateTag* format and given to the *AnnoDocument* component which then displays the newly received annotations to the user.

The *AnnoDocument* component provides the actual annotation logic of the application, as well as the conversion of the list of *AnnotateTag* objects into actual training data. It manages two state variables: The current list of document sentences and the current annotation state in the form of an *AnnotateTag* list. Notable functions are:

- *handleChange(value: AnnotateTag[])*

This is the callback function of the *TextAnnotate* component. It is invoked every time the user manually annotates the document or deletes an annotation, resulting in a change of the list of *AnnotateTag* objects. It extracts the newly added or removed tag object by computing a delta between the old value of the list of *AnnotateTag* objects and the new one. The extracted tag object is then either added to the list of training data or removed from it.

- *addAnnotation(newValue: AnnotatedSentence)*

This function handles adding a newly annotated sentence to the training data. Its main job is to determine whether the new training sample is already present in the training data (i.e. if that sentence already has an annotation at a different position). If so, it merges the new sentence with the already present sample. If not, it creates a new sample.

- *removeAnnotation(toBeRemoved: AnnotatedSentence)*

This function is the inverse of the *addAnnotation* function. It checks if the sentence we want to remove the annotation from is present in the training data and if yes, if it has more than one annotation. If it has more than one annotation, only the desired tag is removed (i.e. replaced with `O`), while the training sample itself stays. If the annotation to be removed is the only one in that sentence, the training sample is removed.

- *annotateEntity(sentence:string, startIndex:number, endIndex:number, tag:string)*

This function does the actual conversion of *AnnotateTag* objects into usable training data samples. It extracts the annotated substring that is specified in the `[startIndex, endIndex]` interval and builds a list of strings and a list of tags, where the list of tags corresponds to the `tag` parameter (for the parts of the sentence that are in the interval `[startIndex, endIndex]`) or to the `O` tag (for parts of the sentence that are outside the interval). For the substrings within the interval, the BIO tagging scheme is applied. In a second step, it removes all punctuation marks that are part of a token and makes them their own token. Their corresponding entry in the list of tags is either the specified tag (if the punctuation mark was part of an entity annotation, i.e. part of the interval `[startIndex, endIndex]`), or `O` (if they were not part of an annotation).

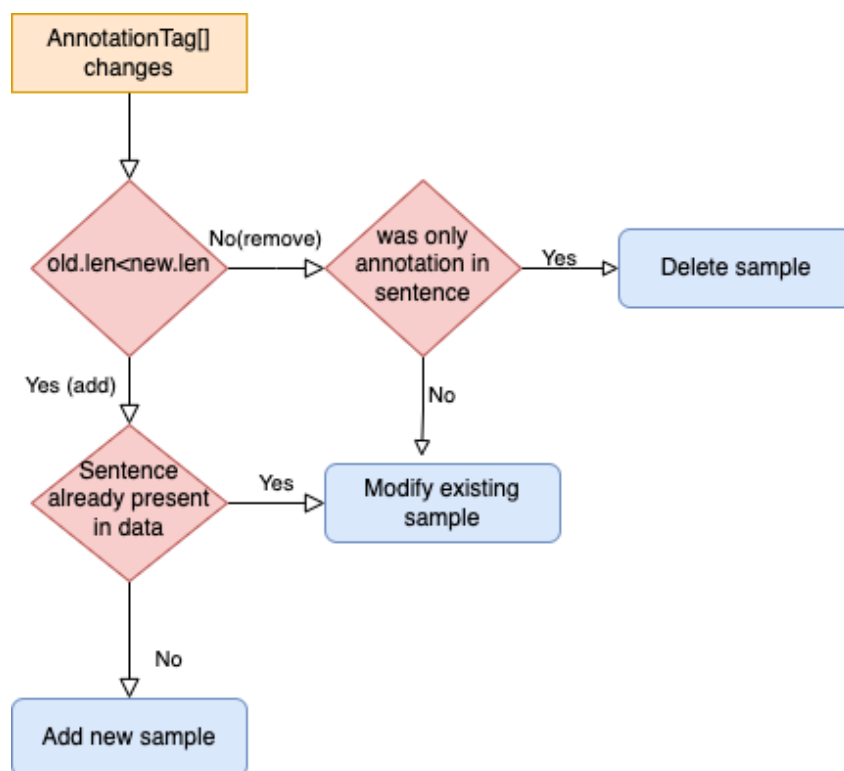


Figure 5.10: Simplified illustration of the conversion of *AnnotateTag* objects into training data

5.2.2 Back End

The back end of the annotation system was implemented using Python 3.9. Its main job is to manage everything that is related to the actual NER model and to provide an API for client communication. This includes initializing the model, loading and saving the model, using the model for inference, and training the model. Currently, only one model type is available, which is an adapter-transformer model. The model is implemented using the *huggingface-transformer* library³ and its extension for adapters⁴. To allow asynchronous training of the model, a task queue was implemented using the library *Redis Queue*⁵, which uses a *Redis*⁶ event queue for queueing jobs and processing them in the background. The API was implemented using *FastApi*⁷ 0.95.1.

Task Queue

Training a model might take some time, depending on the available hardware and the number of training samples. Because of this, it is necessary to introduce asynchronous processing into the back end, in order to avoid timed-out HTTP requests and a bad user experience. The described annotation system realizes asynchronous processing using the library *Redis Queue*. *Redis Queue* (RQ) requires *Redis* and offers an easy-to-use interface for job queueing and background job

3. <https://huggingface.co/>

4. <https://adapterhub.ml/>

5. <https://python-rq.org/>

6. <https://redis.com/>

7. <https://fastapi.tiangolo.com/>

processing building on top of the Redis queue data structure. At initialization, RQ requires a Redis connection object. To queue a job, RQ offers the function *enqueue(ref, params)*, which receives a reference of the function that is supposed to be scheduled along with its parameters and handled asynchronously. To process scheduled tasks, a worker is needed. A worker is a Python process that is started separately in the work directory and waits for new tasks to arrive. A worker can be started with

```
$ rq worker --url redis://rq_redis:6379
```

where `--url` specifies the redis connection. The number of tasks that can be executed asynchronously is limited by the number of available workers. Once a task arrives in the queue, the worker copies the relevant context and starts executing the function. Upon the beginning of execution, RQ returns a *Job* object that can be used by the client to query the current execution state of jobs. If the job is finished, the return value of the executed function is written to Redis under the key `job_id`. If the job has failed, the thrown exception is saved instead.

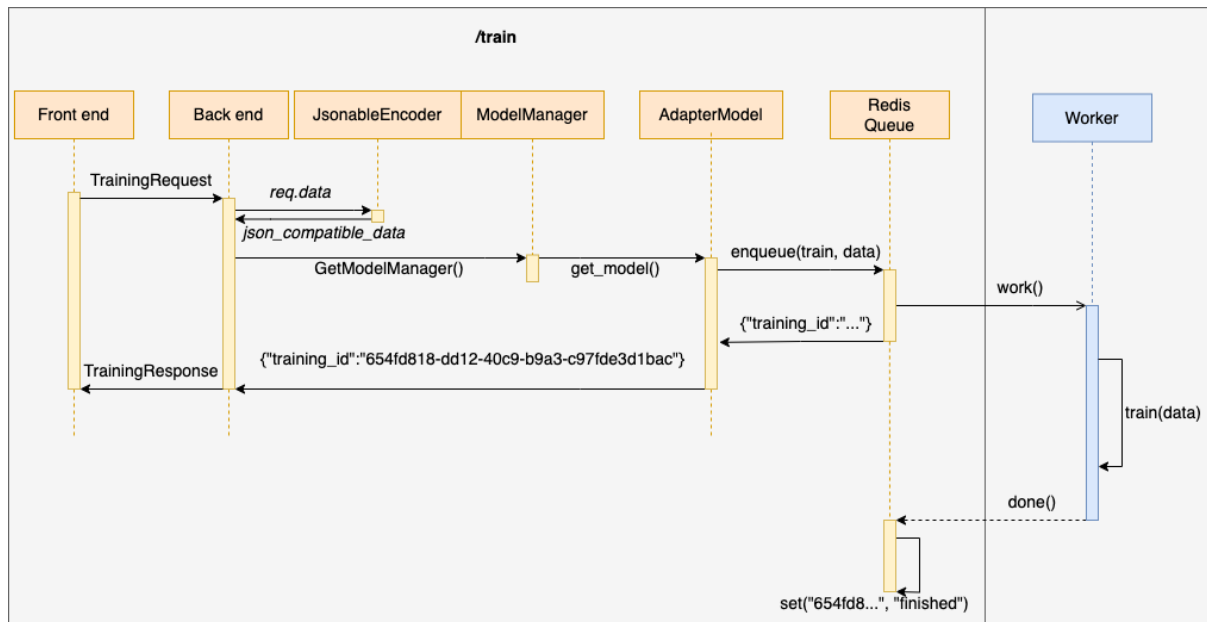


Figure 5.11: Sequence diagram of the processes in the back end after receiving a *TrainingRequest*

Model

To allow for extensibility, the class *Model* represents an abstraction layer over the actual model. Concrete models need to implement the functions *train* and *infer*. The class *ModelManager* then provides a function *getModel()*, which returns the actual implemented model. Currently, the class *AdapterModel* is the only model implemented. It contains all functionality needed to train an adapter-transformer and to do predictions with it. Internally, it manages a huggingface transformer model, which is currently set to be a BERT model (bert-base-cased) and the necessary tokenizer. It also provides the functionality to add different adapters to the model (via the adapter-transformer library) and to remove or modify them.

API

The API enables the client to communicate with the backend via various endpoints. In the following, the most important endpoints are described.

POST /inference

This endpoint is used by the client to start the inference process (i.e. the *Annotate* button has been pressed). The payload of an inference request is the list of strings that the client wants the model to run inference on:

```
#Inference Request
class PredictionRequest(BaseModel):
    sentences: List[string]
```

After successful inference, the model sends back the results as a list of *ClassifiedSentence* objects, where each *ClassifiedSentence* refers to an object with the fields *sentence* and *tags*, where *sentence* is the list of words of the sentence and *tags* is the list of corresponding tags:

```
#Inference Response
class PredictionResponse(BaseModel):
    sentences: List[ClassifiedSentence]

class ClassifiedSentence(BaseModel):
    sentence: List[str]
    tags: List[str]
```

POST /training

This endpoint is used by the client to initiate training and send the training data. It also expects a list of *ClassifiedSentence* objects as the payload, where the *ClassifiedSentence* objects are used as training samples:

```
#Training Request
class TrainingRequest(BaseModel):
    data: List[ClassifiedSentence]
```

Figure 5.11 depicts the entire process in the back end after a training request.

After successfully starting the training, it sends back an ID, which can be used by the client to query the current training status:

GET /training/{training_id}

This endpoint accepts the training ID that the client wants to query the status of and sends back the current training status, where the status can be "started", "finished", or "failed".

POST /settings/loadmodel

This endpoint is used for loading a model by specifying a path in the payload.

POST /settings/createmodel

This endpoint is used for creating a new model. It receives the name of the new model, the label list of the new task, and the path to save the model after training.

#Create a new Model

```
class CreateModelRequest(BaseModel):  
    model_name: str  
    label_list: List[str]  
    path: str
```

GET /settings/availablemodels

This endpoint returns a list of all usable models that are stored on the back end side.

GET /settings/labelmap

This endpoint returns a list of the labels that the currently loaded model was trained on.

GET /split

This endpoint is used by the client to split the document into sentences for further processing. It expects a document and returns the sentences of the document in a list. For sentence splitting, the library *spacy*⁸ is used.

POST /trainingparameters

This endpoint allows to set training parameters of the model.

#Set Training Parameters Request

```
class SetTrainingParameterRequest(BaseModel):  
    params : TrainingArgs  
  
class TrainingArgs(BaseModel):  
    overwrite_output_dir: bool = True  
    learning_rate: float = 1e-5  
    batch_size: int = 32  
    num_train_epochs: int = 10  
    weight_decay: float = 0.01  
    evaluation_strategy: str = "no"  
    save_strategy: str = "no"
```

8. <https://spacy.io/>

6 Future Work

The research presented in this thesis showed that state-of-the-art NER systems pose a multitude of shortcomings when it comes to using them for an automatic annotation system. This affects inference efficiency (Cui et al., 2021), rapid task switching (R Ma et al., 2022) or handling a gradually larger number of N and K (episodic learning approaches). Furthermore, while the results are quite good and roughly represent the results of state-of-the-art few-shot NER research, the accuracy of such models is most likely not high enough for a real production use of an automated annotation system, where it is expected to annotate text using a varying number of labels and a varying number of examples while also maintaining satisfying accuracy most of the time. The following describes some approaches to improve the overall efficiency of a semi-automatic annotation system that could be analyzed in future work

6.1 Multiple Models

Metric-based models like ProtoBERT achieve very good results for lower K . Unfortunately, memory restraints hindered the evaluation of those methods on larger episode sizes. However, to increase the accuracy of the annotation system in the lower sample range, a model-switching mechanism could be implemented, where inference is done for example with a ProtoBERT model up to a specified number of samples, and after that, a second model, which is trained on all previous data, takes over.

6.2 Expansion of Possible Annotations

The work presented in this thesis researched automatic annotation mainly in the context of named entity recognition. However, it is imaginable to expand the functionality of the system from annotating named entities to arbitrary token classification tasks like part-of-speech tagging. Also in some cases, users might want to annotate not only words or entities but also, for example, whole sentences or even paragraphs that share some pre-defined semantic similarities. Future work could research the possibilities regarding the expansion of annotations to more broad classification tasks.

6.3 Prompt-Based Few-shot NER

This thesis analyzed two prompt-based methods, one based on BERT (EntLM (R Ma et al., 2022)) and one based on BART (TemplateNER, (Cui et al., 2021)). Both methods performed comparably well in the few-shot setting, as well as on the task of differentiating finer-grained labels. However, in a realistic setting, it is not enough for an annotation system to get a label right roughly 50 % of the time. Given that the power of prompt-based few-shot NER methods strictly

stems from the language understanding of the model that it inherited during its pre-training phase, it is logical to assume that a better language understanding will substantially improve the model's performance in a few-shot setting. In recent years, a trend in natural language processing developed, where larger and larger language models are released frequently (cf. Figure 6.1). While, for example, the largest BERT model has 340 million parameters, the largest GPT-3 (Brown et al., 2020) model has 175 billion parameters. The GPT-3 researchers showed that their model achieved state-of-the-art results on several NLP tasks without fine-tuning, simply by utilizing prompts and the sheer size of the model. Their research indicated that an increase in parameters alone will increase model performance on most few-shot learning tasks, canceling out the need for sophisticated task-specific fine-tuning altogether. This parameter race is ongoing and companies release new models that achieve new state-of-the-art results frequently.

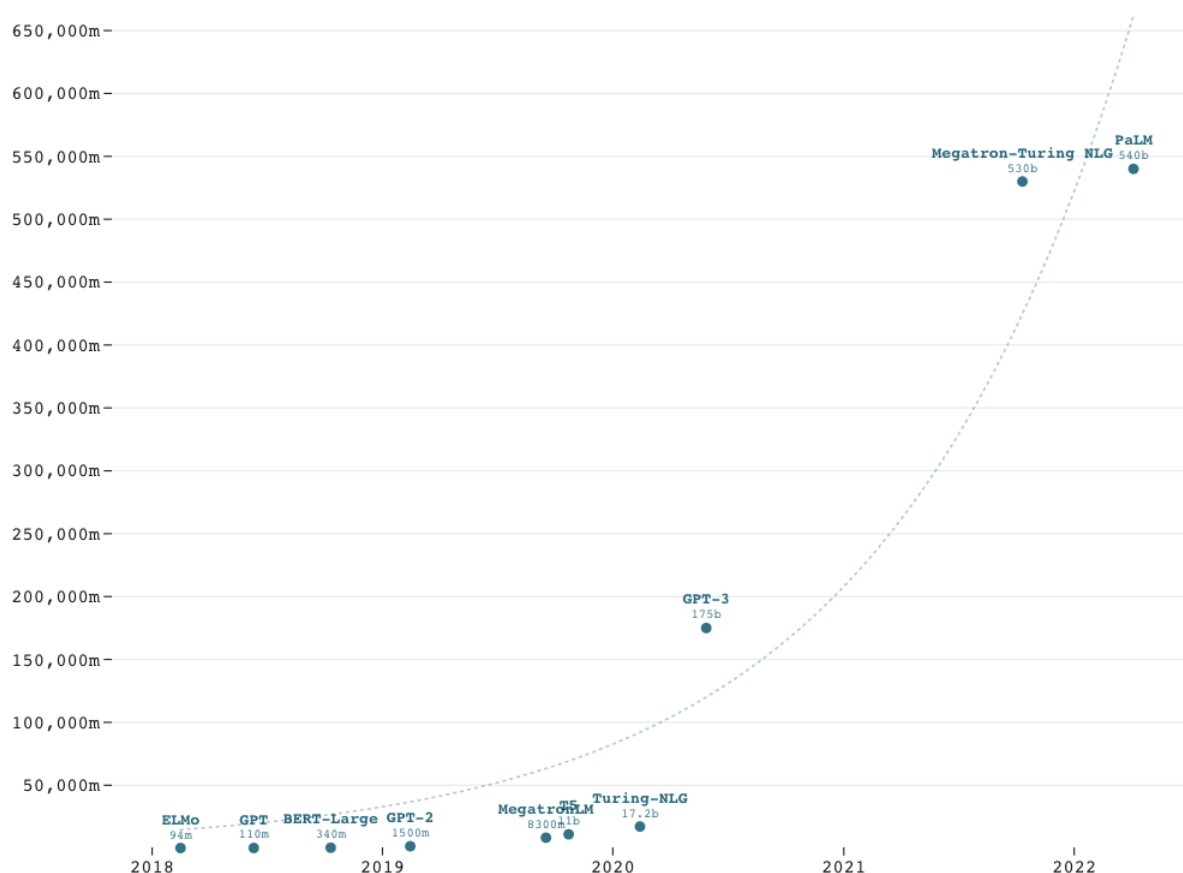


Figure 6.1: Visualisation of how the number of parameters of large language models grew in recent years. Not shown: GPT-4, which, according to a recent leak, has 1.8 *trillion* parameters. Source: Hisamoto (2023)

Moreover, while the largest models at the point of writing this (like GPT-3, PaLM (Chowdhery et al., 2022), PaLM2 (Google, 2023), GPT-4 (OpenAI, 2023) among others) are proprietary and not open for researchers, recently, open source language models (LLaMa (Touvron et al., 2023), Falcon (TII, 2023)) have been released that - according to their publishers - can be seen as serious alternatives to their proprietary counterparts (LLaMA-13B, for example, outperformed GPT-3 on most benchmarks while using only a fraction of the parameters and LLaMA-65B is competitive to PaLM-540B (Touvron et al., 2023)). With the help of these open-source models, future work could try to further leverage the language modeling skills of large language models for the purpose of few-shot named entity recognition.

7 Conclusion

7.1 Research Questions Revisited

At the beginning of this thesis, three questions have been defined, on the basis of which the research done in this thesis was conducted. The questions were:

RQ1: What state-of-the-art methods exist for few-shot (fine-grained) named entity recognition?

After a comprehensive literature study of current state-of-the-art NER models, it became apparent that there are essentially two broad categories of few-shot NER systems: The first category are meta-learning-based systems that are trained episodically and do classification based on some metric calculated using the support set. Meta-learning-based systems are very effective for lower numbers of examples per class and flourish in the setting where the model needs to differentiate between sets of labels frequently given only a small amount of examples. However, episodic learning approaches could not be evaluated for episodes with a larger amount of classes and a larger amount of samples and it is therefore not clear how they can be made compatible with the setting of a semi-automatic annotation system. Examples of meta-learning-based systems are ProtoBERT (Ding et al., 2021) and StructShot (Yang and Katiyar, 2020). The second category of few-shot NER systems are prompt-based systems. Prompt-based systems, instead of using sophisticated classification techniques or additional layers, utilize the language understanding of pre-trained transformer models directly. For this, the task is reformulated in such a way that it resembles the pre-training task of the model. Examples for prompt-based systems are TemplateNER (Cui et al., 2021) and EntLM (R Ma et al., 2022). These prompt-based systems are reported to perform well for few-shot NER tasks. However, they bring other disadvantages, such as high inference time (TemplateNER) or the inability of switching tasks quickly (EntLM). Adapters (Pfeiffer, Rücklé, et al., 2020) offer the advantage of being able to rapidly switch tasks without large storage overhead. They also perform comparably well in the few-shot setting, if the base transformer model has been fine-tuned on a rich-resource dataset. Chapter 3 answers this research question in a more detailed manner.

RQ2: How many support samples are needed to achieve adequate results on an automatic annotation task based on entity classes and how does the number of classes relate to the number of required support samples?

The experiments done during the course of this thesis showed that few-shot named entity recognition is a challenging task, even for state-of-the-art models. For $K = 10$ examples per class and $N = 16$ classes, no system achieved an F1 score over 53 % and for $K = 500$ examples per class (TemplateNER), no system achieved an F1 score over 70 % (Adapter). Furthermore, the additional experiments with $K > 500$ examples that were conducted on Adapters indicate, that the F1 score will converge somewhere in that region. The number of classes present in the training data had only a minor influence on the performance of Adapter: For $K = 10$ examples per class, the model’s performance was roughly equal for 16 classes and for 66 classes, with

around 20-24 % F1 score. For $K = 500$ examples per class, the 16-class Adapter achieved an F1 score of 67 % and the 66-class Adapter achieved an F1 score of 57 %.

RQ3: How does the presence of semantically more similar labels affect the overall performance of NER models?

The results regarding the presence of fine-grained labels in the training data were mixed. Meta-learning-based approaches achieved F1 scores of 42 % (StructShot) and 32 % (ProtoBERT) for the task of differentiating between 10 fine-grained classes which were randomly sampled for each episode, when being given only 2 examples per class. This indicates that meta-learning approaches have the potential to perform well in a fine-grained few-shot setting. TemplateNER achieved an F1 score of 53% on the 7-class fine-grained experiment, which is roughly equal to its score on the 16-class experiment on more coarse categories. Adapter achieved an F1 score of 27 % on the 7-class fine-grained experiment, which also is roughly equal to its score on the 16-class experiment, indicating that the fine-grained task is only slightly harder for those models. EntLMs performance on fine-grained labels on the other hand was very low, indicating that not all prompt-based systems are equally suitable for fine-grained NER tasks.

7.2 Summary and Conclusion

The goal of this thesis was to research state-of-the-art few-shot named entity recognition systems and implement a prototype of a semi-automatic annotation system. First, different state-of-the-art few-shot NER systems have been researched and five systems were selected according to their reported results and how suitable they are for the few-shot NER task of an automated annotation system. For training and experiments, suitable datasets for few-shot NER have been researched. Here, a special focus was set on datasets that best can resemble fine-grained and few-shot named entity recognition tasks. After selecting a dataset, experiments were carried out on all five systems, where the experiments resembled a 16-class few-shot NER task and a 7-class fine-grained few-shot NER task. Meta-learning approaches like ProtoBERT and StructShot achieve good results for lower K while also being efficient during training and inference. Due to their episodic learning nature, they are also good at task switching because the model "learns to learn" from new episodes every time. However, it is not clear how they perform for gradually larger N and K , because evaluation on larger N and K was not possible in a straightforward manner due to memory issues. Leveraging the language modeling capacities of pre-trained language models directly in order to do few-shot NER (*prompt-based learning*) sounds promising at first. While both prompt-based systems delivered adequate results in terms of F1 score on the 16-class experiment, they both revealed their own weaknesses: EntLM - while achieving fast inference times - needs to generate the label representatives based on an unlabeled corpus with the help of knowledge bases, making it impractical for quickly switching tasks in the annotation system. Furthermore, EntLM displayed severe problems during the classification of finer-grained labels. TemplateNER, on the other hand, displayed inference times that are impractical for the use case of an annotation system, with inference times in the order of multiple seconds per sentence. Prompt-based learning approaches scale directly with the power of the underlying language model. Recently released open-source language models like LLaMA (Touvron et al., 2023) or Falcon (TII, 2023) could be analyzed further with regard to their capabilities on few-shot named entity recognition and might be an entry point to improving such a system. Adapters, while exhibiting no specific few-shot NER technique in their vanilla form, offer the advantage of being able to rapidly switch tasks without the need of saving large additional models, because only the

specific Adapter is saved. They also perform comparably well in a low-resource setting, when the base transformer model has been fine-tuned on a rich-resource dataset. The efficient training and inference time of Adapters is also an advantage. During the second part of this thesis, a prototype of an annotation system has been designed and implemented using Adapter models. On the front end, the system displays an interface for the user, with which they can upload a text document, annotate it using custom labels, and initiate training of the model with their own annotations. After training, the system is able to automatically annotate the uploaded document using the labels that the user specified. On the backend, the system manages a huggingface transformer Adapter model, utilizing a task queue for asynchronous training. Overall, it has been shown that a semi-automatic annotation system presents a multitude of challenges to the underlying model and that none of the investigated state-of-the-art models fully meet the requirements of such a system.

Bibliography

- Armen Aghajanyan, Sonal Gupta, and Luke Zettlemoyer. 2021. Intrinsic Dimensionality Explains the Effectiveness of Language Model Fine-Tuning. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, 7319–7328. Online: Association for Computational Linguistics, August.
- Nasser Alshammari and Saad Alanazi. 2021. The impact of using different annotation schemes on named entity recognition. *Egyptian Informatics Journal* 22 (3): 295–302.
- Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450*.
- Dzmitry Bahdanau, Kyunghyun Cho, and Y. Bengio. 2014. Neural Machine Translation by Jointly Learning to Align and Translate. *ArXiv* 1409 (September).
- Tom Brown et al. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, edited by H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, 33:1877–1901. Curran Associates, Inc.
- Yanru Chen, Yanan Zheng, and Zhilin Yang. 2022. Prompt-Based Metric Learning for Few-Shot NER.
- Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*.
- Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*.
- Leyang Cui, Yu Wu, Jian Liu, Sen Yang, and Yue Zhang. 2021. Template-Based Named Entity Recognition Using BART. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, 1835–1845. Online: Association for Computational Linguistics, August.
- Sarkar Snigdha Sarathi Das, Arzoo Katiyar, Rebecca Passonneau, and Rui Zhang. 2022. CONTaiNER: Few-Shot Named Entity Recognition via Contrastive Learning. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 6338–6353. Dublin, Ireland: Association for Computational Linguistics, May.
- Leon Derczynski, Eric Nichols, Marieke van Erp, and Nut Limsopatham. 2017. Results of the WNUT2017 Shared Task on Novel and Emerging Entity Recognition. In *Proceedings of the 3rd Workshop on Noisy User-generated Text*, 140–147. Copenhagen, Denmark: Association for Computational Linguistics, September.
- Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics*, 4171–4186. Minneapolis, MN, USA.
- Ning Ding, Guangwei Xu, Yulin Chen, Xiaobin Wang, Xu Han, Pengjun Xie, Haitao Zheng, and Zhiyuan Liu. 2021. Few-NERD: A Few-shot Named Entity Recognition Dataset. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th Inter-*

- national Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, 3198–3213. Online: Association for Computational Linguistics, August.
- Ding et al. 2021. Few-NERD. <https://github.com/thunlp/Few-NERD>.
- Jeffrey L Elman. 1990. Finding structure in time. *Cognitive science* 14 (2): 179–211.
- Chelsea Finn, Pieter Abbeel, and Sergey Levine. 2017. Model-agnostic meta-learning for fast adaptation of deep networks. In *International conference on machine learning*, 1126–1135. PMLR.
- G.D. Forney. 1973. The viterbi algorithm. *Proceedings of the IEEE* 61 (3): 268–278.
- Alexander Fritzler, Varvara Logacheva, and Maksim Kretov. 2019. Few-shot classification in named entity recognition task. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, 993–1000.
- Tianyu Gao, Adam Fisch, and Danqi Chen. 2021. Making Pre-trained Language Models Better Few-shot Learners. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, 3816–3830. Online: Association for Computational Linguistics, August.
- Jonas Gehring, Michael Auli, David Grangier, Denis Yarats, and Yann N. Dauphin. 2017. Convolutional Sequence to Sequence Learning. In *Proceedings of the 34th International Conference on Machine Learning*, edited by Doina Precup and Yee Whye Teh, 70:1243–1252. Proceedings of Machine Learning Research. PMLR, June.
- Abbas Ghaddar and Philippe Langlais. 2018. Transforming Wikipedia into a large-scale fine-grained entity type corpus. In *Proceedings of the eleventh international conference on language resources and evaluation (LREC 2018)*.
- Google. 2023. PaLM2 Technical Report. Accessed July 1, 2023. <https://ai.google/static/documents/palm2techreport.pdf>.
- Ralph Grishman and Beth Sundheim. 1996. Message Understanding Conference- 6: A Brief History. In *COLING 1996 Volume 1: The 16th International Conference on Computational Linguistics*.
- Junxian He, Chunting Zhou, Xuezhe Ma, Taylor Berg-Kirkpatrick, and Graham Neubig. 2021. Towards a unified view of parameter-efficient transfer learning. *arXiv preprint arXiv:2110.04366*.
- He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 770–778.
- Charles T. Hemphill, John J. Godfrey, and George R. Doddington. 1990. The ATIS Spoken Language Systems Pilot Corpus. In *Proceedings of the Workshop on Speech and Natural Language*, 96–101. HLT '90. Hidden Valley, Pennsylvania: Association for Computational Linguistics.
- Sorami Hisamoto. 2023. Sizes of Large Language Models. Accessed July 1, 2023. <https://observablehq.com/@sorami/sizes-of-large-language-models>.
- Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9 (8): 1735–1780.
- Yutai Hou, Wanxiang Che, Yongkui Lai, Zhihan Zhou, Yijia Liu, Han Liu, and Ting Liu. 2020. Few-shot Slot Tagging with Collapsed Dependency Transfer and Label-enhanced Task-adaptive Projection Network. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 1381–1393. Online: Association for Computational Linguistics, July.

- Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-efficient transfer learning for NLP. In *International Conference on Machine Learning*, 2790–2799. PMLR.
- Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.
- Jiaxin Huang, Chunyuan Li, Krishan Subudhi, Damien Jose, Shobana Balakrishnan, Weizhu Chen, Baolin Peng, Jianfeng Gao, and Jiawei Han. 2021. Few-Shot Named Entity Recognition: An Empirical Baseline Study. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 10408–10423. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, November.
- Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, 448–456. pmlr.
- Dan Jurafsky and James Martin. 2008. *Speech and Language Processing*. 2nd ed. Upper Saddle River, NJ: Pearson.
- John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. 2001. Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data. In *Proceedings of the Eighteenth International Conference on Machine Learning*, 282–289. ICML ’01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.
- Dong-Ho Lee, Akshen Kadakia, Kangmin Tan, Mahak Agarwal, Xinyu Feng, Takashi Shibuya, Ryosuke Mitani, Toshiyuki Sekiya, Jay Pujara, and Xiang Ren. 2022. Good Examples Make A Faster Learner: Simple Demonstration-based Learning for Low-resource NER. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2687–2700. Dublin, Ireland: Association for Computational Linguistics, May.
- Sergey Levine. 2021. CS W182/282A - Meta Learning. Accessed July 1, 2023. <https://cs182sp21.github.io/static/slides/lec-21.pdf>.
- Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. 2020. BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 7871–7880. Online: Association for Computational Linguistics, July.
- Chen Liang, Yue Yu, Haoming Jiang, Siawpeng Er, Ruijia Wang, Tuo Zhao, and Chao Zhang. 2020. Bond: Bert-assisted open-domain named entity recognition with distant supervision. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, 1054–1064.
- Jingjing Liu, Panupong Pasupat, Scott Cyphers, and Jim Glass. 2013. Asgard: A portable architecture for multilingual dialogue systems. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, 8386–8390. IEEE.
- Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*.
- Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2022. Pre-Train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. Just Accepted, *ACM Comput. Surv.* (New York, NY, USA) (September).

- Ma, Huiqiang Jiang, Qianhui Wu, Tiejun Zhao, and Chin-Yew Lin. 2022. Decomposed Meta-Learning for Few-Shot Named Entity Recognition. In *Findings of the Association for Computational Linguistics: ACL 2022*, 1584–1596. Dublin, Ireland: Association for Computational Linguistics, May.
- Jie Ma, Miguel Ballesteros, Srikanth Doss, Rishita Anubhai, Sunil Mallya, Yaser Al-Onaizan, and Dan Roth. 2022. Label Semantics for Few Shot Named Entity Recognition. In *Findings of the Association for Computational Linguistics: ACL 2022*, 1956–1971. Dublin, Ireland: Association for Computational Linguistics, May.
- Ruotian Ma, Xin Zhou, Tao Gui, Yiding Tan, Linyang Li, Qi Zhang, and Xuanjing Huang. 2022. Template-free Prompt Tuning for Few-shot NER. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 5721–5732. Seattle, United States: Association for Computational Linguistics, July.
- OpenAI. 2023. GPT-4 Technical Report. Accessed July 1, 2023. <https://cdn.openai.com/papers/gpt-4.pdf>.
- Jonas Pfeiffer, Aishwarya Kamath, Andreas Rücklé, Kyunghyun Cho, and Iryna Gurevych. 2020. AdapterFusion: Non-destructive task composition for transfer learning. *arXiv preprint arXiv:2005.00247*.
- Jonas Pfeiffer, Andreas Rücklé, Clifton Poth, Aishwarya Kamath, Ivan Vulić, Sebastian Ruder, Kyunghyun Cho, and Iryna Gurevych. 2020. AdapterHub: A Framework for Adapting Transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP 2020): Systems Demonstrations*, 46–54. Online: Association for Computational Linguistics.
- Jonas Pfeiffer, Ivan Vulić, Iryna Gurevych, and Sebastian Ruder. 2020. Mad-x: An adapter-based framework for multi-task cross-lingual transfer. *arXiv preprint arXiv:2005.00052*.
- Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. In *NeurIPS EMC²Workshop*.
- Timo Schick and Hinrich Schütze. 2021. Exploiting Cloze-Questions for Few-Shot Text Classification and Natural Language Inference. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, 255–269. Online: Association for Computational Linguistics, April.
- Mike Schuster and Kuldeep K Paliwal. 1997. Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing* 45 (11): 2673–2681.
- Jake Snell, Kevin Swersky, and Richard Zemel. 2017. Prototypical Networks for Few-shot Learning. In *Advances in Neural Information Processing Systems*, edited by I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, vol. 30. Curran Associates, Inc.
- Amber Stubbs and Ozlem Uzuner. 2015. Annotating longitudinal clinical narratives for de-identification: The 2014 i2b2/UTHealth corpus. *Journal of biomedical informatics* 58S (August).
- Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. *Advances in neural information processing systems* 27.
- TII. 2023. Falcon40B. Accessed July 1, 2023. <https://falconllm.tii.ae/>.
- Erik F. Tjong Kim Sang and Fien De Meulder. 2003. Introduction to the CoNLL-2003 Shared Task: Language-Independent Named Entity Recognition. In *Proceedings of the Seventh Conference on Natural Language Learning at HLT-NAACL 2003*, 142–147.

- Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.
- Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30.
- C.R. Rao Venkat N. Gudivada. 2018. Computational Analysis and Understanding of Natural Languages: Principles, Methods and Applications -. Stanford: Elsevier Science.
- Luke Vilnis and Andrew McCallum. 2014. Word Representations via Gaussian Embedding. *CoRR* abs/1412.6623.
- Liwen Wang, Rumei Li, Yang Yan, Yuanmeng Yan, Sirui Wang, Wei Wu, and Weiran Xu. 2022. InstructionNER: A Multi-Task Instruction-Based Generative Framework for Few-shot NER. *ArXiv:2203.03903 [cs]*, March.
- Ralph Weischedel, Martha Palmer, Mitchell Marcus, Eduard Hovy, Sameer Pradhan, Lance Ramshaw, Nianwen Xue, Ann Taylor, Jeff Kaufman, Michelle Franchini, Mohammed El-Bachouti, Robert Belvin, and Ann Houston. 2013. OntoNotes Release 5.0. V. V1.
- Alexander Wettig, Tianyu Gao, Zexuan Zhong, and Danqi Chen. 2023. Should You Mask 15% in Masked Language Modeling? In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*, 2985–3000. Dubrovnik, Croatia: Association for Computational Linguistics, May.
- Sam Wiseman and Karl Stratos. 2019. Label-Agnostic Sequence Labeling by Copying Nearest Neighbors. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 5363–5369. Florence, Italy: Association for Computational Linguistics, July.
- Ikuya Yamada, Akari Asai, Hiroyuki Shindo, Hideaki Takeda, and Yuji Matsumoto. 2020. LUKE: Deep Contextualized Entity Representations with Entity-aware Self-attention. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 6442–6454. Online: Association for Computational Linguistics, November.
- Yi Yang and Arzoo Katiyar. 2020. Simple and Effective Few-Shot Named Entity Recognition with Structured Nearest Neighbor Learning. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 6365–6375. Online: Association for Computational Linguistics, November.
- Dian Yu, Luheng He, Yuan Zhang, Xinya Du, Panupong Pasupat, and Qi Li. 2021. Few-shot Intent Classification and Slot Filling with Retrieved Examples. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 734–749. Online: Association for Computational Linguistics, June.
- Yaoming Zhu, Jiangtao Feng, Chengqi Zhao, Mingxuan Wang, and Lei Li. 2021. Serial or parallel? plug-able adapter for multilingual machine translation. *arXiv preprint arXiv:2104.08154* 6 (3).
- Yukun Zhu, Ryan Kiros, Rich Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. 2015. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *Proceedings of the IEEE international conference on computer vision*, 19–27.
- Morteza Ziyadi, Yuting Sun, Abhishek Goswami, Jade Huang, and Weizhu Chen. 2020. Example-based named entity recognition. *arXiv preprint arXiv:2008.10570*.

List of Figures

2.1	A simple artificial neuron with inputs x_1, x_2, x_3	9
2.2	Schematic illustration of an RNN (left) and how it unfolds over time (right) . .	11
2.3	Seq2Seq Model	12
2.4	Illustration of how Bahdanau et al. (2014) utilize annotation vectors h_i to gener- ate context vectors c_i	14
2.5	The general architecture of transformer as described in the original paper . . .	15
2.6	Illustration of the scaled dot-product attention	16
2.7	Multi-head attention	18
2.8	Residual Connection	19
2.9	Illustrated concept of masked attention	20
2.10	BERT pre-training and fine-tuning	22
2.11	Pre-training objectives of BART	23
2.12	Visualization of popular tagging schemes BIO, IO and BIOES	24
3.1	The classification process in a Prototypical Network	29
3.2	Token-level based inference of StructShot	30
3.3	The inference process of TemplateNER	32
3.4	The inference process of EntLM	33
3.5	Illustration of a basic Bottleneck Adapter module	36
3.6	Sequential adapter configuration (left) and parallel adapter configuration (right)	37
3.7	Decomposing ΔW into two low-rank matrices A and B	38
4.1	The classes of the Few-NERD dataset	40
4.2	Visualisation of true positives, false positives, true negatives, and false negatives in a 2-class setting	41
4.3	Class-specific F1 scores for the fine-grained experiments on TemplateNER. . .	47
4.4	The F1 values of the BUILDING classes for different K for EntLM.	49
4.5	The F1 values of the BUILDING classes for different K for adapter-transformer	51
4.6	F1 values of the 16-class adapter experiments, both for the base model and prior fine-tuned model	52
5.1	Overview of the annotation system	55
5.2	The <i>Upload Document</i> dialogue	56
5.3	The <i>Select Model</i> dropdown	56
5.4	Dialogue for creating a new model	57
5.5	The <i>Select Tag</i> dropdown menu	58
5.6	The Annotation Area	58
5.7	The Training Data Staging Area	59
5.8	A click on the button <i>Get Training Status</i> queries the current training status and informs the user via an alert box	60
5.9	Example of how tags for a document are stored internally	61
5.10	Simplified illustration of the conversion of <i>AnnotateTag</i> objects into training data	63


5.11	Sequence diagram of the processes in the back end after receiving a <i>TrainingRequest</i>	64
6.1	Visualisation of how the number of parameters of large language models grew in recent years	68

List of Tables

4.1	The 16 classes of the Few-NERD dataset that were used in all experiments along with their parent classes	43
4.2	The 7 classes of the Few-NERD dataset that were used for a fine-grained analysis	43
4.3	The results of training DistilBERT on the coarse-grained (eight different classes) and on the fine-grained (66 different classes) Few-NERD data set.	44
4.4	Results of ProtoBERT and Structshot experiments using INTER setting	45
4.5	Results of ProtoBERT and Structshot experiments using INTRA setting	45
4.6	Experiments on TemplateNER with and without prior fine-tuning	46
4.7	Results for the 7-class fine-grained experiment on TemplateNER	46
4.8	The results for both experiments on EntLM	49
4.9	The results of both experiments for adapter-transformer	50
4.10	Results (F1) of TemplateNER, EntLM, and Adapter models on the 16-class experiment. Results in brackets denote the results for the same experiment with a prior fine-tuned base model	54
4.11	Results (F1) of TemplateNER, EntLM, and Adapter models on the fine-grained experiment	54
4.12	Additional experiments with adapter-transformer	54

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Masterstudiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Unterschrift: 

Ort, Datum: Hamburg, 17.07.2023

Erklärung zur Veröffentlichung

Ich bin damit einverstanden, dass meine Abschlussarbeit in den Bestand der Fachbereichsbibliothek eingestellt wird.

Unterschrift: 

Ort, Datum: Hamburg, 17.07.2023