N-WAY / FEW-SHOT LEARNING FOR TEXT CLASSIFICATION

MASTER THESIS IN DATA SCIENCE

at ZHAW Zürich University of Applied Science

Winterthur, 30.06.2021

Author

Hunkeler Bruno

Ibergstrasse 99
8405 Winterthur
+41 79 699 80 67
bruno.hunkeler@gmx.ch

Supervisor 1

Prof. Dr. Cieliebak Mark

ZHAW School of Engineering
8400 Winterthur

ciel@zhaw.ch

Supervisor 2

Deriu Jan Milan, PhD Student

ZHAW School of Engineering
8400 Winterthur

deri@zhaw.ch

Table of contents

Table of figures

Tables

1 Management Summary

Unstructured data in the form of text is everywhere: social media, emails, chats, web pages, support tickets, survey responses, and more. Text is an extremely rich source of information but extracting insights or classifying text can be hard and time-consuming due to its unstructured nature. Text classification via Machine- or deep learning can help businesses to automatically structure and analyze their text data, in a quick and cost-efficient way, which leads to enhanced data-driven decisions.

The process of learning good features for Machine Learning applications can be very computationally expensive and data intensive. Today's standard deep neural networks fail in the small data regime, but with many real-world problems there are only a few samples available per class. Collecting additional data is either remarkably time consuming and costly or altogether impossible. Therefore, the need to learn from only a few available samples is omnipresent and would relieve us from the data-gathering burden and would also reduce the requirement of large compute resources.

Deep neural networks have shown great success in the large data domain but fail in small data settings. The field of few-shot learning has recently seen substantial advancements. Few-shot learning encapsulates a family of methods that can learn new concepts with only a handful of data points. Most of these advancements came from casting few-shot learning as a meta-learning problem. Model-Agnostic Meta-Learning (MAML) is currently one of the best approaches for few-shot learning via meta-learning. MAML is simple, elegant and very powerful.
Meta-learning has shown strong performance in computer vision, where low-level patterns are transferable across learning tasks. However, directly applying this approach to text is challenging.

In this paper, we will cast few-shot learning as a meta-learning problem by applying the Model-Agnostic Meta-Learning method to text classification. The nature of few-shot learning requires prior knowledge of an existing task to be able to classify text. A knowledge transfer from similar tasks is crucial to learn a robust model (e.g., transfer learning). However, manual knowledge transfer from one task to another for the purpose of fine-tuning on a new task can be a time consuming and ultimately inefficient process. Meta-learning, or learning to learn, can instead be used to automatically learn across-task knowledge, such that our model can, at inference time, quickly acquire task-specific knowledge from new tasks using only a few samples.

We will show that the approach of combining MAML with few-shot learning will reduce the need for data collection and labelling, hence reducing the time and resources needed to build robust machine learning models, but still being able to generalize well. We describe ways to capitalize on discriminative features to generalize by predicting not just new data, but entire new classes.

With our approach in combining Model-Agnostic Meta-Learning with Convolutional Neural Network architectures by applying just a hand-full of data, usually 5 or 10-samples per class, we can achieve near state-of-the-art performance.

2 Introduction

The human capacity to learn new concepts using only a handful of samples is immense. In contrast, modern deep neural networks need, at a minimum, thousands of samples before beginning to learn representations that can generalize well to unseen data points (Krizhevsky et al., 2012; Huang et al., 2017), and mostly fail when the data available is scarce. The fact that standard deep neural networks fail in the small-data regime can provide hints about some of their potential shortcomings. Solving those shortcomings has the potential to open the door to understanding intelligence and advancing Artificial Intelligence.

Deep learning-based approaches have seen great successes in a variety of fields. Successes have largely been in areas where vast quantities of data can be collected, and where huge compute resources are available. This excludes many applications where data is intrinsically rare or expensive, or compute resources are unavailable.

The field of few-shot learning has recently seen substantial advancements. Few-shot learning encapsulates a family of methods that can learn new concepts with only a handful of data points (usually 5-10 samples per class). This possibility is attractive for several reasons. Few-shot learning would reduce the need for data collection and labelling, hence reducing the time and resources needed to build robust machine learning models. It would potentially reduce training and fine-tuning times to adapt systems to newly acquired data. In many real-world problems there are only a few samples per class available and the collection of additional data is either remarkably time consuming and costly or altogether impossible. Therefore, there is a need to learn from only a few available samples.

Meta-learning has emerged as a promising methodology in computer vision for learning in a low-resource regime. This is especially interesting if the goal is to enable an algorithm to expand to new classes for which only a few training instances are available. These models learn to generalize in these low-resource conditions by recreating such training episodes from the data available. Even in the most extreme low-resource scenario with a single training example per class, this approach shows near state-of-the-art performance.

Given this strong empirical performance, we are interested in employing meta-learning frameworks in Natural Language Processing (NLP). The challenge, however, is the degree of transferability of the underlying representation learned across different classes. In computer vision, low-level patterns and their corresponding representations can be shared across tasks. However, the situation is different for language data where most tasks operate at the lexical level. Words that are highly informative for one task may not be relevant for other tasks. Words highly salient for one class do not play a significant role in classifying others.

Neural network meta-learning has a long history. However, its potential as a driver to advance the frontier of today's deep learning industry has led to an explosion of recent research. In particular, meta-learning has the potential to alleviate many of the main criticisms of contemporary deep learning, for instance, by providing better data efficiency, exploitation of prior knowledge transfer, and enabling unsupervised learning.

Model Agnostic Meta-Learning (MAML) is currently one of the best approaches in the field of few-shot learning via meta-learning but is a rather young research area. The vast majority of research is based in the area of computer vision rather than in text analytics or classification. Applying Model Agnostic Meta-Learning for text classification is a road less travelled.

2.1 Problem description and Objectives

2.1.1 Problem description

Unstructured data in the form of text is everywhere: social media, emails, chats, web pages, support tickets, survey responses, and more. Text is an extremely rich resource of information but extracting insights or classifying text can be hard and time consuming due to its unstructured nature. Text classification via machine or deep learning can help businesses to automatically structure and analyze their text data in a quick and cost-efficient way. Text classification is associated with Natural Language Processing (NLP), which is a field within Artificial Intelligence (AI) and machine learning that combines linguistics and computer science to break down language so it can be analysed by machines. Text understanding, text classification, language translation and question and answering are just a few of the subjects in the NLP space.

As a product manager for financial applications, I am confronted in many ways with lots of unstructured text data, such as news information, cash management reports, texts in invoices, etc. The richness of information contained in text data allows us to enhance and automate our product in many ways. Text classification in that sense is one of the main drivers in my current area of work. Training contemporary deep learning models requires vast quantities of data. In many real-world problems there are only a few samples per class available and the collection of additional data is either remarkably time consuming and costly or altogether impossible. Therefore, learning from only a few available samples (few-shot learning) is omnipresent and would relieve us from the data-collection burden.

Few-shot learning is interesting in many ways. Certain real-world problems have long-tailed and imbalanced data distributions. This means the vast majority of text information is available for certain types of classes (usually just a handful) and rare for most other classes. This may result in poor performance of models/applications based on long-tailed or imbalanced data distributions. A further aspect is the fact that this might reduce the time and resources needed to build robust machine learning models and reduce the computing resources required for training models.

The main problem we intend to solve is to apply a mechanism, which allows one to overcome the problem of classifying text separated in various classes in a low-data regime. This would relieve us from the data-collection burden and the computationally expensive training cycles. Multi-class text classification problems in our application often appear in different flavours. A usual case is to distinguish between 2, 5 or 10 classes. But there are also cases where we need to classify text information, for example, in cash management reports or invoices and assign them to a set a 30, 40 or even 50 classes. However, cash management reports of our customers contain sensitive information and therefore are not always easy to get hold of. Therefore, we will use publicly available news text data with ~40 or even more classes.

The nature of few-shot learning requires prior knowledge of an existing task to be able to classify text. A knowledge transfer from similar tasks is crucial to learn a robust model (e.g., transfer learning). However, manual knowledge transfer from one task to another for the purpose of fine-tuning on a new task can be a time consuming and ultimately inefficient process. Meta-learning, or learning to learn, can instead be used to automatically learn across-task knowledge, such that our model can, at inference time, quickly acquire task-specific knowledge from new tasks using only a few samples. Meta-learning can be broadly defined as a class of machine learning models that become more proficient at learning with more experience, hence learning how to learn. More specifically, meta-learning involves learning at two levels. At the task level, where the base model is required to acquire task-specific knowledge rapidly, and at the meta-level, where the meta-model is required to slowly learn across-task knowledge.

Meta-learning has emerged as a promising methodology in computer vision for learning in a low-resource regime. Model-Agnostic Meta-Learning (MAML), which is a relatively new approach in the field of few-shot learning, is currently one of the best approaches for such a setting. Even though most cases MAML is used for are in computer vision, we will apply this technique to our text classification problem, given the favourable properties of the algorithm.

2.1.2 Objectives

The aim is to apply **n-way/few-shot learning in text classification** (multi-class text classification), where n-way is the number of classes to distinguish between and few-shot the number of samples per class. This should resolve two omnipresent problems: building a classification system with a limited amount of training data and training models on low computational resource constraint environments. With these boundaries, it favors the use of meta-learning based on the Model-Agnostic Meta-Learning (MAML) method. The few-shot setting must be applicable to any news text data in the form of "text, label". The system must allow the flexibility to classify news text data in the form of a hierarchical or non-hierarchical structured with a broader range of class labels (e.g., >30). The system should provide similar accuracy to conventional Machine Learning (ML) or Deep Learning (DL) models for a respective task. Given the fact that our applications are used in central Europe the system must be able to classify English and German news text data.

2.1.3 Delimitations

Scope:
Data considerations:

- The application must be applicable to news data sets in a hierarchical/non-hierarchical structure in the form of "text, label". The class assignment to text data must be disjoint (no multi-label classification).
- The application must be able to process English as well as German news text data.

Text Pre-processing / Feature Engineering:

- Establish a text pre-processing pipeline, which considers text cleaning, such as stop word elimination, stemming, lemmatization, contraction expanding, etc.
- Feature engineering should consider aspects like syntactic parsing, entity extraction, statistical features, word embeddings.

Model Building:

- Establish a baseline, based on conventional machine learning and deep learning models for comparison.
- Build a meta-learning system by favouring the Model-Agnostic Meta-Learning method based on Neural Networks and the associated loss functions for a classification task.

Model comparison:

- Apply appropriate metrics for model comparison, such as accuracy, f1-score, confusion matrix, etc.
- Compare the results of the meta-learning algorithm with conventional machine and deep learning classifiers.

Out of Scope:
Reinforcement Learning, which is a next stage in a few-shot regime, will not be applied. Further languages apart from German and English are not considered.

2.2 Structure of Thesis

Part 1 - Data Preparation, Pre-Processing
The main driver in part 1 is the getting to know the data (explanatory data analysis) and text data pre-processing. The text data preparation part focuses on building a data cleaning pipeline considering steps that fall into the categories of *Noise Removal*, *Lexical Normalization*, and *Standardization*. Aspects in this context are stop word elimination, stemming, lemmatization, contraction extraction, Named Entity Recognition (NER), Part-of-Speech tagging (POS), etc. Other parts covered in this section are data augmentation (imbalanced data), building vocabularies, applying word embeddings, and preparing the data into a digestible form to be used in deep learning models.

Even though data preparation and pre-processing is the first part in a Data Science project, the above-mentioned descriptive parts have been placed in the Appendix, chapter 10.

Part 2 – Meta-learning, Model-Agnostic Meta-Learning
The central subject, meta-learning with respect to Model-Agnostic Meta-Learning, in this thesis is covered in chapters 4 and 5. We address meta-learning and Model-Agnostic Meta-Learning to formalize the high-level problem in order to understand and position these methods. We also will position meta-learning with respect to related topics, such as transfer learning, and provide a brief description of methodologies in terms of meta-representation, and meta-optimizer as model-based, metric-based (non-parametric) and optimization-based methods. This section then focusses on the various aspects of building a meta-learning architecture by discussing loss and optimizer functions.

Part 3 – Baseline of Conventional Machine and Deep Learning Classifiers
Meta-learning as a novel method in supervised learning needs to be compared to a reference system. Baseline classifiers described in chapter 6 focus on building conventional machine and deep learning classifiers as a reference system and provide respective performance metrics. Baseline classifiers include Random Forest, Multinominal Naïve Bayes and XGBoost in a conventional machine learning context, while in deep learning we consider models like Convolutional Neural Network, Sequence Model and Bidirectional Encoder Representations from Transformers (BERT). As part of the model analytics (e.g., scores, error, etc.) provided in chapter 6, we include further analytics in the form of “model explainability” via Lime, which will be available in the Appendix.

Part 4 – Experiment, Challenges, and Conclusion
Part 4, covered in chapters 7, 8, 9, provides insight into the effective Model-Agnostic Meta-Learning experiment, to understand whether MAML really learns to adapt to novel tasks. This includes data set-up, model architectures, optimization, error and loss functions and the training and validation procedure. As part of chapter 8 we will discuss challenges and ways to overcome certain shortcomings in our experiment. Finally, we will conclude the learnings in the thesis and provide potential next steps to consider.

3 Basic Terminology and Definitions

This chapter describes the basic terminology and definitions used throughout this thesis.

3.1 Supervised Learning

In machine learning and artificial intelligence, supervised learning refers to a class of systems and algorithms that determine a predictive model using data points with known outcomes. The model is taught by training through an appropriate learning algorithm (such as linear Regression, Random Forests, or Neural Networks) that typically works through some optimization routine to minimize a loss or error function. Different denominations are used, depending on the output. If the system has a continuous output, the task is typically denoted as regression, while it is called classification for nominal outputs. In case of several quantitative outputs, it is called multivariate regression and for multiple qualitative outputs, it is named a multi-class classification. Unsupervised learning in contrast, has an unknown target output. Throughout this thesis, we will focus on methods for supervised learning and multi-class classification.

3.2 Text Classification

Classification is a process of categorizing a given set of data into classes. It can be performed on both structured and unstructured data. Classes are often referred to as target, label, or categories.
Text classification as such is the task of assigning predefined categories to text documents. Classes are selected from a previously established taxonomy (a hierarchy of categories or classes). News text, for example, is typically organized by subject categories (topics) such as business, health, medicine etc.

3.3 Machine and Deep Learning

3.3.1 Machine Learning (ML)

Machine Learning is a field of computer science that aims to teach computers how to learn and act without being explicitly programmed. More specifically, machine learning is an approach in data analytics that involves building and adapting models, which allow programs to "learn" through experience. Machine learning involves the construction of algorithms that adapt their models to improve their ability to make predictions. It is a sub-field of Artificial Intelligence.

3.3.2 Deep Learning (DL)

Deep learning, a sub-field of machine learning, is a technique that constructs artificial neural networks to mimic the structure and function of the human brain. In practice, deep learning, also known as deep structured learning or hierarchical learning, uses large numbers of hidden layers, typically more than 3 but often many more. Deep learning networks process nonlinear functions to extract features from data and transform the data into different levels of abstraction (representations). Nonlinearities means that the neural network can successfully approximate functions that do not follow linearity or it can successfully predict the class of a function that is divided by a decision boundary which is not linear.

Deep learning drives many Artificial Intelligence (AI) applications and services that improve automation, performing analytical and physical tasks without human intervention. Deep learning technology lies behind everyday products and services (such as image recognition, digital assistants, voice-enabled TVs, and credit card fraud detection) as well as emerging technologies (such as self-driving cars).

3.3.3 Hyperparameter

Every machine or deep learning algorithm requires hyperparameters. Hyperparameters are parameters that need to be set before training a machine learning algorithm, as opposed to normal parameters of an algorithm that are not fixed before execution but optimized while training the algorithm. Hyperparameters are learning rate, number of epochs, momentum, regularization constant and so forth.

3.3.4 Optimization

Optimization is the process of maximizing or minimizing an objective function by finding the best available values across a set of inputs. Some variation of optimization is required for all deep learning models to function, whether using supervised or unsupervised learning. There are many specific techniques to choose from, but all optimization algorithms require as a minimum an objective function $f(x)$, to define the output that is being maximized or minimized. This function can be deterministic (with specific effects) or stochastic (achieving a probability threshold) and inputs that are controllable, in the form of variables like x_1, x_2, etc. Each can be either discrete or continuous. There is a wide variety of optimizers, such as *Gradient Descent*, *Stochastic Gradient Descent*, *Adam*, *MSProp*, *Adagrad* and so forth, each holding specific properties.

3.3.5 Loss Function

Loss functions are used to determine the error ("the loss") between the output of our algorithms and the given target value. Loss functions also take part in optimization problems with the goal of minimizing the loss. Loss functions are used while training Neural Networks by influencing how their weights are updated. The larger the loss is, the larger the update. By minimizing the loss, the model's accuracy is maximized. However, the trade-off between size of update and minimal loss must be evaluated in a respective machine learning algorithm.

3.3.6 Bias, Variance

A supervised machine learning model aims to train itself on the input variables (X) in such a way that the predicted values (Y) are as close to the actual values as possible. This difference between the actual values and predicted values is the error and it is used to evaluate the model. The error for any supervised machine learning algorithm comprises 3 parts: Bias error, Variance error and Noise. Noise is the irreducible error that cannot be eliminated, while bias and variance are reducible errors that can be minimized. There are different strategies to approach high/low bias and variance. Another term often used in the same context is *over- and underfitting*. This situation where any given model is performing too well on the training data, but the performance drops significantly over the test set, is called an overfitting model. On the other hand, if the model is performing poorly over the test and the training set, then we call that an underfitting model.

3.4 Natural Language Processing (NLP)

Natural Language Processing (NLP) is a field of computer science that is primarily concerned with the interactions between computers and natural (human) languages. Major emphases of natural language processing include speech recognition, natural language understanding, natural language generation, language translation and so forth. Machine learning methods can be applied to each of these areas.

3.4.1 Text Pre-Processing

Text in its core is the most unstructured form of available data and is associated with various types of noise. Text pre-processing is traditionally an important step for natural language processing tasks. It transforms text into a more digestible form so that machine learning algorithms can perform better. Pre-processing consists of a number of steps, but generally fall under the broad categories of *Noise Removal*, *Lexical Normalization*, and *Standardization*.

Noise Removal

This step deals with the removal of all types of noise entities present in texts. Any piece of text which is not relevant or does not provide additional information to the context or semantic in data can be specified as noise. Commonly used words in a language as "is", "am", "the", "of", "in", etc. are known as stopwords. URLs, links, mentions, hashtags (in social media entities), punctuations and industry-specific words, also contribute little value in text classification and can therefore, be treated as noise.

Lexical Normalization
Lexical normalization is the task of translating/transforming a non-standard text to a standard register. Textual noise is about multiple representations exhibited by a single word, e.g., “play”, “player”, “played”, “plays” and “playing”, which are different variations of the word “play”. The most common approaches in lexical normalizations are **stemming** and **lemmatization**. Stemming is a rudimentary rule-based process of stripping the suffixes (“ing”, “ly”, “es”, “s”, etc) from a word, while lemmatization is an organized and step-by-step procedure of obtaining the root form of the word, making use of a vocabulary and morphological analysis (word structure and grammar relations).

Standardization
Text data often contains words or phrases which are not present in any standard lexical dictionaries. These pieces are not recognized by search engines and models. If an application does not benefit from these words and is just leading to sparsity issues, one can consider removing these words from the datasets.

3.4.2 Imbalanced Data

An imbalanced classification problem is an example of a classification problem where the distribution of examples across the known classes is biased or skewed. The distribution can vary from a slight bias to a severe imbalance where there is one example in the minority class for hundreds, thousands, or millions of examples in the majority class or classes. When we encounter the problem of imbalanced data, we are bound to have difficulties solving it with standard algorithms. Conventional algorithms are often biased towards the majority class, not taking the data distribution into consideration. In the worst case, minority classes are treated as outliers and ignored. For some cases, such as fraud detection or cancer prediction, we would need to carefully configure our model or artificially balance the dataset, by under- or over-sampling each class. Under-sampling in this context means to reduce the number of samples of a majority class towards the number of samples of a minority class. Over-sampling, in contrast, is the process of creating additional artificial samples for minority classes, so the number of samples per class is balanced. This can be achieved via data augmentation.

3.4.3 Text Data Augmentation

When working on Natural Language Processing applications, such as text classification, manually collecting enough labelled examples for each category can be difficult. There are various strategies to overcome the lack of an equivalent amount of text data per class. Data augmentation helps to prevent overfitting and train more robust models. Several approaches exist to augment text data. The approaches are mentioned in appendix 10.3.

3.4.4 Stratified Sampling

Stratified sampling is a sampling technique where the samples are selected in the same proportion (by dividing the population into homogeneous subpopulations called ‘strata’ based on a specific characteristic, e.g., race, gender, location, etc.) as they appear in the population. Every member of the population should be in exactly one stratum. Implementing the concept of stratified sampling ensures that the training and test sets have the same proportion of the feature of interest as in the original dataset. Doing this with the target variable ensures that the result is a close approximation of generalization error.

3.5 Feature Engineering in NLP

The input to machine learning models usually consists of features and a target variable. The target is the item that the model is meant to predict, while features are the data points being used to make the predictions. Feature engineering in natural language processing is very difficult. The major difficulties are that we do not consciously understand the language ourselves. Another aspect is that a language itself is ambiguous. Linguistic concepts, such as words or sentences, seem to be simple, clear, and well-formed ideas, but in reality, there are so many borderline cases that can be quite difficult to grasp. If we just consider homonyms or synonyms. A single word can have many different meanings, or different words have the same meaning. In order to understand what a sentence means, we have to understand the meaning of the words in that sentence, which is not a simple task. The field of NLP aims to convert the human language into a form of representations that are easy for computers to manipulate. In other words, when we deal with an NLP problem, our input text needs to convert into something that our algorithms can understand. Hence, a feature needs to be converted into a numerical representation.

In general, we can categorize text features into two main categories. One category is meta-features, such as word counts, stop word counts, punctuation counts, the length of characters, the language of text, and many more. Other types of features are text-based features, such as tokenization, vectorization, stemming, part of

speech tagging, named entity recognition and so on. As for tokenization there are several possible ways, like TF-IDF, Unigrams, bigrams, trigrams, character-grams, count vectors and so forth. TF-IDF and count vectors will be used as features with our ML baseline models (see chapter 3.5.1 TF-IDF- and Count Vectors). Feature engineering in NLP is a research area by itself. The process of extracting meaningful features requires domain knowledge. Well-curated features result in machine learning models with higher accuracy.

3.5.1 TF-IDF- and Count Vectors as features

TF-IDF Vectors

TF-IDF score represents the relative importance of a term in the document and the entire text corpus. TF-IDF score is composed of two terms: the first computes the normalized Term Frequency (TF), the second term is the Inverse Document Frequency (IDF), computed as the logarithm of the number of the documents in the corpus divided by the number of documents where the specific term appears.

tf(t, d) = (Number of times term t appears in a document (d)) / (Total number of terms in the document)

$$tf(t,d) = \frac{f_d(t)}{\underset{w \in d}{max}(f_d(w)}$$

idf(t) = log (Total number of documents / Number of documents containing term t)

$$idf(t) = log\ \frac{n+1}{df+1}$$

$$TF - IDF = TF(t,d) * IDF(t)$$

TF-IDF Vectors can be generated at different levels of input tokens (words, characters, n-grams)

Word level:	TF-IDF: Matrix representing tf-idf scores of every term in different documents
N-gram level:	N-grams are the combination of N terms together. This Matrix represents tf-idf scores of N-grams
Character level:	Matrix representing tf-idf scores of character level n-grams in the corpus

Count Vector

Count Vector is a matrix notation of the dataset in which every row represents a document from the corpus, every column represents a term from the corpus, and every cell represents the frequency count of a particular term in a particular document.

Tf-idf, count vector and the associated word, n-gram character representations will only be used with our Machine Learning baseline models (Multinominal Naïve Bayes, Random Forest and XGBoost).

3.5.2 Word embeddings

Many machine learning algorithms and almost all deep learning architectures are incapable of processing strings or plain text in their raw form. A word embedding is a form of representing words and documents using a dense vector representation. The position of a word within the vector space is learned from text and is based on the words that surround the word when it is used. This allows words that are used in similar ways to result in having similar representations, naturally capturing their meaning. Word embeddings can be trained using one's own text corpus. This will require a large amount of text data (usually millions or billions of words) to ensure that useful embeddings are learned. Another option is to use pre-trained word embeddings, such as *Glove*, *FastText*, and *Word2Vec*. Such embeddings have been trained on billions of words and are publicly available. All of them are available in 100, 200, and 300 dimensional versions. For the current problem setting we apply pre-trained embeddings: *GloVe for English (300d)* and *FastText for German (300d)* texts.

4 Meta-Learning

Today's machine learning models can master complex tasks. They are typically trained from scratch for a specific task using a fixed learning algorithm designed to solve a specific problem. Deep learning-based approaches in the supervised learning space have seen great successes in a variety of fields. These successes have largely been in areas where vast quantities of data can be collected or simulated, and where large computing resources are available. These datasets are expensive to create, especially when one needs to involve domain experts.

Such approaches become less effective for domain-specific problems, which still require large amounts of task-specific labelled data to achieve good performance. This was the situation when pre-trained models were introduced. Pre-trained models are beneficial if a new task is similar to the previous task. Using a pre-trained model requires less training and less effort in building a new model architecture, but they fail when a model needs to acquire new skills or adapt to new tasks. We cannot afford to train each skill in each setting from scratch. Instead, we need to teach our system to learn, how to learn new tasks by reusing previous experience, rather than considering each new task in isolation.

Meta-learning provides an alternative paradigm where a machine learning model gains experience over multiple learning episodes mostly covering a distribution of related tasks. Meta-learning uses this experience to improve its future learning performance. This ***learning-to-learn*** can lead to a variety of benefits, such as data and computing efficiency, but furthermore, to the ability to adapt to new unseen tasks. Meta-learning is also much better aligned with human learning, where learning strategies improve both on a lifetime and evolutionary timescale.

Learning-to-learn is an aspect conventional machine learning or deep learning approaches do not attempt. This is what makes meta-learning particularly attractive. Meta-learning has the potential to alleviate many of the main criticisms of today's deep learning algorithms. It provides, for instance, a better data and computing efficiency, exploitation of prior knowledge transfer, and enables unsupervised and self-directed learning. Meta–learning can learn from a handful of samples and adapt to novel tasks quickly. It has the capability to build more generalizable systems.

One of the most popular problems meta-learning helps us to solve is few-shot learning. In a few-shot learning scenario, we only have a limited number of examples on which to perform a supervised learning task. Even more important is the ability to learn effectively from them. This might relieve us from the data-gathering burden.

Therefore, meta-learning proposes an end-to-end, deep learning algorithm that can learn a representation better suited for few-shot learning. It is similar to the pre-trained network approach, except that it learns an initialization that serves as a good starting point for a handful of training data points. In a few-shot classification problem for text classification, one could leverage training data that is available from different news text classes, for instance, text passages describing subjects (classes) like sports, medicine, business, etc. The model could then build up prior knowledge such that, at inference time, it can quickly acquire task-specific knowledge with only a handful of training examples. This way, the model first learns parameters from a given training dataset that consists of texts from one type of class, and then uses those parameters as prior knowledge to tune them further, on a new set of classes, based on the limited training set.

But how can a model learn a good initial set of parameters which can then be easily adapted to a downstream task? The answer lies in a simple training principle, which was initially proposed by Vinyals et. al. [1]

Train and test conditions must match

The idea is to train a model by showing it only a few examples per class, and then test it against samples from the same classes that have been held out from the original dataset, much the way it will be tested when presented with only a few training samples from novel classes. Each training sample, in this case, comprises pairs of train and test data points called tasks or episodes.

This is a departure from the way that data is set up for conventional supervised learning. The training data (meta-training data) is composed of train and test samples, alternately referred to as the support and query set. The figure below uses icons as a visual explanation. A more formal description of the data set-up is outlined in chapter 5

[1] Vinyals et al. Matching Networks for One Shot Learning (2016), arXiv:1606.04080

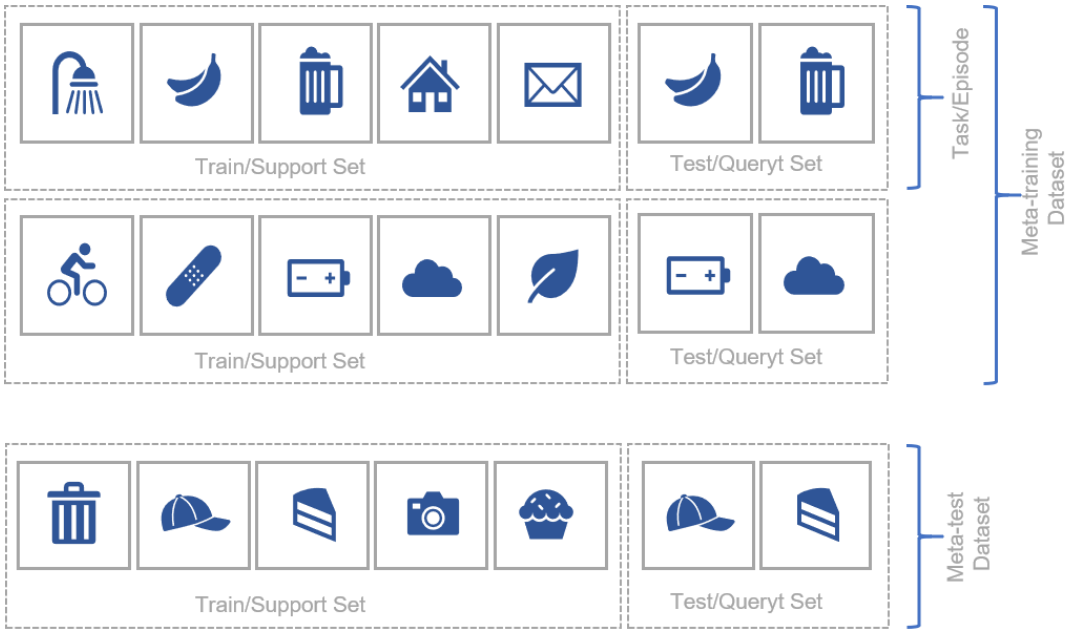

Figure 1: Meta-data set, adopted from Optimization as a Model for few-shot learning

> The number of classes (N) in the support set defines a task as an N-class classification task or N-way task, and the number of labeled samples in each class (k) corresponds to k-shot, making it an N-way, k-shot learning problem.

4.1 Meta-learning – learning to learn

Meta-learning is commonly understood as *learning-to-learn*, which refers to the process of improving a learning algorithm over multiple learning tasks. In contrast, conventional machine learning considers the process of improving model predictions over multiple data instances. In a meta-learning regime, an inner learning model (base algorithm) solves a task, such as text classification, defined by a dataset and an objective. During meta-learning, an outer model (algorithm) updates the inner learning model, such that the model learned by the inner model improves an outer objective. A meta-learning model should be trained on a variety of tasks, and then optimized further for novel tasks. A task, in this case, is basically a supervised learning problem. The idea is to extract prior information from a set of tasks that allows efficient learning on new tasks. For our text classification problem, the ideal set-up would include many classes, with at least a few samples each. These can then be used as a meta-training set to extract prior information, such that when a new task comes in, the model can perform it more efficiently. Meta-Learning models can be decomposed into two stages: meta-learning and adaptation. In the meta-learning phase, the model learns an initial set of parameters slowly across tasks; during the adaptation phase, it focuses on quick acquisition of knowledge to learn task-specific parameters. Since the learning happens at two levels, meta-learning is also known as learning to learn.

A variety of approaches have been proposed that vary based on how the adaptation portion of the training process performs. These can broadly be classified into three categories: *model-based*, *metric-based*, and *optimization-based* approaches.

A ***model-based*** approach simply trains an entire neural network, given some training samples in the support set and an initial set of meta-parameters, and then makes predictions on the query set. The model-based setting approaches the problem as supervised learning task, although there are approaches that try to eliminate the need to learn an entire network.

Metric-based approaches usually employ non-parametric techniques (for example, k-nearest neighbors) for learning. The core idea is to learn a feature representation (e.g., learning an embedding network that transforms raw inputs into a representation, which allows similarity comparison between the support set and the query set). Hence, the performance depends on the chosen similarity metric (like cosine similarity or Euclidean distance).

Optimization-based approaches treat the adaptation part of the process as an optimization problem. In this thesis we will focus only on the *optimization-based* approach. In the following, we will briefly describe how such a process works.

During training, we iterate over datasets of tasks or episodes. In meta-training, we start with the first task, and the meta-learner takes the training set (support set) and produces a learner (model) that will take as input the test set (query set) and make predictions on it. The meta-learning objective is based on a loss function (in our case cross-entropy) that is derived from the query set samples and will backpropagate through these errors. The parameters of the meta-learner (meta-parameters) are then updated based on these errors to optimize the loss. In the next step, we pick up the next task, train with the support set samples, make predictions on the query set and update meta-parameters. This procedure is repeated with all available tasks. In attempting to train a meta-learner this way, we are trying to solve the problem of generalization. The samples in the query set are not part of the training set, so the meta-learner is learning to extrapolate. The conceptual approach of meta-learning learn-to-learn as mentioned above is depicted in **Figure 2**: Learn-to-learn.

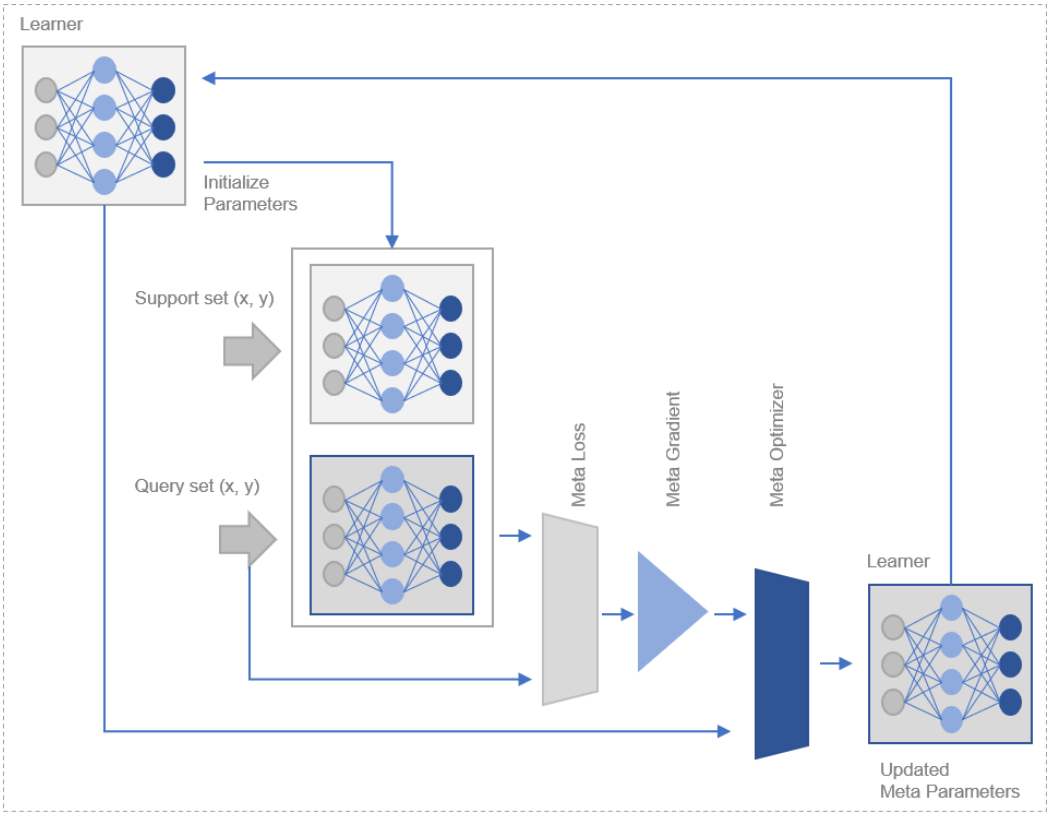

Figure 2: Learn-to-learn

4.2 Transfer Learning for NLP

With the advancement in deep learning, neural network architectures, like Recurrent Neural Networks and Long Short Term Memory (RNN and LSTM) and Convolutional Neural Networks (CNN), have shown a decent improvement in performance in solving several Natural Language Processing (NLP) tasks like text classification, language modelling, machine translation, etc. However, this performance of deep learning models in NLP pales in comparison to the performance of deep learning in computer vision.

One of the main reasons for this slow progress could be the lack of large, labelled text data sets. Most of the labelled text datasets are not big enough to train deep neural networks because these networks have a huge number of parameters and training such networks on small datasets will cause overfitting.

Another quite important reason for NLP lagging behind computer vision was the lack of transfer learning in NLP. Transfer learning has been instrumental in the success of deep learning in computer vision. This happened due to the availability of huge, labelled datasets like ImageNet[2] on which deep CNN-based models were trained and later they were used as pre-trained models for a wide range of computer vision tasks.

Transfer learning is a technique where a deep learning model is trained on a large dataset, learning the weights for the initial model. Then it is used to perform similar tasks on another dataset. We call such a deep learning model a pre-trained model. The most renowned examples of pre-trained models are the computer vision deep learning models trained on the ImageNet dataset. So, it is better to use a pre-trained model as a starting point to solve a problem rather than building a model from scratch.

So, the need for transfer learning in NLP was at an all-time high. In 2018, the transformer was introduced by Google in the paper “Attention Is All You Need”, which turned out to be a ground-breaking milestone in NLP.

BERT (Bidirectional Encoder Representations from Transformers) is a large neural network architecture, with a huge number of parameters, that can range from 110 million to over 340 million parameters. So, training a BERT model from scratch on a small dataset would result in overfitting. But using the pre-trained model and applying those weights to another downstream task, is computationally less expensive and reduces the risk of overfitting by using smaller datasets.

4.3 Transfer learning vs Meta-learning

The terms transfer learning and meta-learning are often the source of confusion in the literature. Transfer learning is a machine learning method where a model is developed for one downstream task and is reused as the starting point for a model on another downstream task. It uses past experience of a source task to improve learning. In today’s neural network context, it often refers to a particular methodology of parameter transfer plus optional fine-tuning. The benefits of transfer learning are that it can speed up the development and training process by reusing predefined modules and pre-trained models. Transfer learning is simply freezing all the trained layers but substituting the fully connected layer (FC) with the new to-be-trained model. While transfer learning can refer to a problem area, meta-learning refers to a methodology which can be used to improve transfer learning as well as other problems.

Transfer learning as a methodology is differentiated to meta-learning as the prior is extracted by learning on the source task without the use of a meta-objective. In meta-learning, the corresponding prior would be defined by an outer optimization that evaluates how well the prior performs when helping to learn a new task. More generally, meta-learning deals with a much wider range of meta-representations than solely model parameters.

*Bidirectional Encoder Representations from Transformers **(**BERT)*[3] is an excellent example of transfer learning. BERT is a transformer-based machine learning technique for natural language processing (NLP) developed and pre-trained by Google (published in 2018). BERT is pre-trained on unlabeled text (Wikipedia - 2.5 billion and Book Corpus - 800 million words). A pre-trained BERT comes in two flavors:

BERT - Base: 12-layers (transformer blocks), 768-hidden, 12-attention-heads, 110M parameters
BERT - Large: 24-layers (transformer blocks), 1024-hidden, 16-attention-heads, 340M parameters

[2] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems, (2012).

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding (2018). arXiv:1810.04805

Training such a model is nearly impossible, as long as one cannot access the required computing resources. Therefore, such pre-trained models are commonly used with transfer learning. Pre-trained weights are applied and then fine-tuned on a downstream task.
Transfer learning with a BERT model is available as a baseline implementation based on the HuffPost data set. It is only available as a Colab implementation since training a BERT model on a downstream task even by using a pre-trained model is computationally expensive.

5 Model-Agnostic Meta-Learning (MAML)

Model-Agnostic Meta-Learning (MAML) (Finn et al., 2017) [4] is a meta-learning framework for few-shot learning. Model-Agnostic, means that it is applicable to any model trained with gradient descent and, therefore, can be applied to a variety of different learning problems, including classification, regression, and reinforcement learning. MAML is elegantly simple yet can produce state-of-the-art results in few-shot regression and classification learning problems. The objective of MAML is to learn an internal feature representation that is broadly applicable for all the tasks T_i in a task distribution $p(T)$, using only a few data points (k-shot) and training iterations, rather than a single task. It optimizes a set of parameters such that when a gradient step is taken with respect to a particular task T, the parameters θ_{i} are close to the optimal parameters for task i. This is achieved by minimizing the total loss $\sum L_{T_i}$ across all the sampled tasks from task distribution $p(T)$. At the end of meta-training, new tasks are sampled from $p(T)$, and meta-performance is measured by the model's performance after learning from k samples. Generally, tasks used for meta-testing are held out during meta-training.

Formally, we consider a model represented by a parametrized function f_θ with parameters θ. When adapting to a new task T_i, the model's parameters θ become θ_i^*. In our method, the updated parameter vector θ_i^*.
is computed using one or more gradient descent updates on task T_i,. For example, when using one gradient update:

$$\theta_i^* = \theta - \alpha \nabla_\theta \mathcal{L}_{T_i}(f_\theta)$$

The step size (learning rate) α may be fixed as a hyperparameter or learned via meta-learning. The model parameters are trained for the performance of $f_{\theta_i^*}$ with respect to θ across tasks sampled from $p(T)$. The meta-objective is as follows:

$$\min_\theta \sum_{T_i \sim p(T)} \mathcal{L}_{T_i}(f_{\theta_i^*}) = \sum_{T_i \sim p(T)} \mathcal{L}_{T_i}(f_{\theta - \alpha \nabla_\theta \mathcal{L}_{T_i}(f_\theta)})$$

Note that the meta-optimization is performed over the model parameters θ, whereas the objective is computed using the updated model parameters θ^*. The proposed method aims to optimize the model parameters such that one or a small number of gradient steps on a new task will produce maximally effective behavior on that task.

The MAML meta-gradient update involves a gradient through a gradient. Computationally, this requires an additional backward pass via function f to compute vector products, which is supported by standard deep learning libraries, such as TensorFlow. In our experiments, we will use PyTorch in cooperation with the higher library instead of TensorFlow or PyTorch. In our project, we use the higher framework to apply MAML. Higher is a library providing support for higher-order optimization, hence it ensures the second order gradient handling.

The meta-optimization across tasks is performed via stochastic gradient descent (SGD), such that the model parameters θ are updated as follows:

$$\theta \leftarrow \theta - \beta \nabla_\theta \sum_{T_i \sim p(T)} \mathcal{L}_{T_i}(f_\theta)$$

β is the step size (learning rate). Further aspects regarding optimization are discussed in chapter 5.3.2 Optimization.

[4] Chelsea Finn, Pieter Abbeel, Sergey Levine. Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks (2017)

5.1 Meta-learning task description

In the typical machine learning setting, we are interested in a dataset D and usually split D so that we optimize parameters θ on a training set D_{train} and evaluate its generalization on the test set D_{test}. In meta-learning, however, we are dealing with meta-sets $\mathcal{D}$ containing multiple regular datasets, where each $D \in \mathcal{D}$ has a split of D_{train} and D_{test}.
We consider an N-way, k-shot classification task, where for each dataset D, the training set consists of *k* labelled examples for each of *N* classes, meaning that D_{train} consists of $k * N$ examples, and D_{test} has a set number of examples for evaluation. Vinyals et al. used the term *episode* to describe each dataset consisting of a training and test set.

In meta-learning, we have different meta-sets for meta-training, meta-validation, and meta-testing ($\mathcal{D}_{meta-train}$, $\mathcal{D}_{meta-validation}$ and $\mathcal{D}_{meta-test}$). On $\mathcal{D}_{meta-train}$, we are interested in training a learning procedure (the meta-learner) that can take as input one of its training sets D_{train} and produce a classifier (the learner) that achieves high average classification performance on its corresponding test set D_{test}. Using $\mathcal{D}_{meta-validation}$ one can perform hyperparameter selection of the meta-learner and evaluate its generalization performance on $\mathcal{D}_{meta-test}$.[5]
For this formulation to correspond to the few-shot learning setting, each training set in datasets $D \in \mathcal{D}$ will contain few labelled samples (we consider *k* = 5 or *k* = 10), that must be used to generalize to good performance on the corresponding test set.

5.2 Formalizing meta-learning

Conventional machine learning
In conventional supervised machine learning, we are given a training dataset $D = \{(x_1, y_1), \ldots, (x_n, y_n)\}$, such as (input text, output label) pairs. We can train a predictive model $\hat{y} = f_\theta(x)$ parameterized by θ, by solving:

$$\theta^* = arg \min_\theta \mathcal{L}(D; \theta, \omega)$$

where L is a loss function that measures the match between true labels and those predicted by $f_\theta(x)$. We include the condition ω to make it explicit that there exists a dependency in this solution on factors such as choice of optimizer for θ or function class for f, which we denote by ω. Generalization is then measured by evaluating several test points with known labels.

The conventional assumption is that this optimization is performed from scratch for every problem D; and furthermore that ω is pre-specified. However, the specification ω of how to learn θ can dramatically affect generalization, data-efficiency, computation cost, etc.

Meta-learning
The meta-learning problem, is defined by the following formula (max log probabilities), which we need to solve in order to find the required parameters:

Meta-learning step:

$$\theta^* = arg \max_\theta log\ p(\theta | \mathcal{D}_{meta-train}, \omega)$$

$$\mathcal{D}_{meta-train} = \{(D_1^{tr}, D_1^{ts}), \ldots, (D_n^{tr}, D_n^{ts})\}$$
$$D_1^{tr} = \{(x_1^i, y_1^i), \ldots, (x_k^i, y_k^i)\}$$
$$D_1^{ts} = \{(x_1^i, y_1^i), \ldots, (x_l^i, y_l^i)\}$$

Adaptation step:

$$\phi^* = arg \max_\phi log\ p(\phi | D^{tr}, \theta^*, \omega) \leftrightarrow \phi^* = f_{\theta^*}(D^{tr}, \omega)$$

learn θ such that $\phi = f_{\theta^*}(D_i^{tr}, \omega)$ is good for (D_i^{ts}, ω)

$$\theta^* = \max_\theta \sum_{i=1}^{n} log\ p(\phi_i | D^{ts}, \omega)\ where\ \phi_i = f_\theta(D_i^{ts}, \omega)$$

final meta-learning formulation

[5] Sachin Ravi and Hugo Larochelle. Optimization as a model for few-shot learning (2017)

5.3 Model Architecture in Model-Agnostic Meta-Learning

Model-Agnostic Meta-Learning has the fine property to allow any kind of model to be applied. If the model is trained using gradient descent, the approach does not place any constraints on the model architecture or the loss function. This characteristic makes it applicable to a wide variety of problems, including regression, classification, and reinforcement learning.

In our setting, we propose the use of convolutional neural networks in conjunction with Model-Agnostic Meta-Learning. Convolutional neural networks are generally used in computer vision; however, recent advances showed promising results on various NLP tasks (chapter 5.4).

A deep learning neural network learns to map a set of inputs to a set of outputs from training data. We cannot calculate the perfect weights for a neural network; there are too many unknowns. Instead, the problem of learning is cast as a search or optimization problem and an algorithm is used to navigate the space of possible sets of weights the model may use in order to make good or good enough predictions.

Neural networks are usually trained using stochastic gradient descent. One needs to choose a loss function when designing and configuring a model. There are many loss functions to choose from and it can be challenging to choose the best loss function to solve a given problem.

5.3.1 Loss function, multi-class cross-entropy loss

Cost or loss functions are used to determine the error (loss) between the output of our algorithms and the given target value and also in optimization problems with the goal of minimizing the loss. Loss functions are used while training neural networks by influencing how their weights are updated. The larger the loss is, the larger the update. By minimizing the loss, the model's accuracy is maximized. However, the trade-off between size of update and minimal loss must be evaluated in a respective machine learning algorithm and is related to the bias/variance subject discussed in chapter 5.5.

The common loss functions used for supervised classification tasks is cross-entropy, which is described below. In our solution, we apply cross-entropy loss, but other supervised loss functions can also be used:

$$\mathcal{L}_{T_i} = \sum_{x^i, y^i \sim T_i} y^{(i)} \, log \, f_\phi(x^{(i)}) + (1 - y^{(i)}) \, log \, (1 - f_\phi(x^{(i)}))$$

5.3.2 Optimization

Mathematical optimization is the process of maximizing or minimizing an objective function by finding the best available values across a set of inputs. Some variation of optimization is required for all deep learning models to function, whether using supervised or unsupervised learning. There are many specific techniques to choose from, but all optimization algorithms require as minimum an objective function $f(x)$, to define the output that is being maximized or minimized. This function can be deterministic (with specific effects) or stochastic (achieving a probability threshold).
In a meta-learning setting, two independent optimizers are required for the inner and the outer loop. The optimizers applied in our project are a Stochastic Gradient Descent and an ADAM optimizer.

5.3.2.1 Stochastic Gradient Descent (SGD)

Stochastic gradient descent is a method to find the optimal parameter configuration for a machine learning algorithm. It iteratively makes small adjustments to a machine learning network configuration to decrease the error of the network.

Error functions are rarely as simple as a typical parabola. Most often they have lots of hills and valleys (local and global minima). Stochastic gradient descent attempts to find the global minimum by adjusting the configuration of the network after each training point. Instead of decreasing the error, or finding the gradient, for the entire dataset, this method merely decreases the error by approximating the gradient for a randomly selected batch (which may be as small as a single training sample). In practice, the random selection is achieved by randomly shuffling the dataset and working through batches in a stepwise manner.

5.3.2.2 Adam Optimizer

Adam is an alternative optimization algorithm that provides more efficient neural network weights by running repeated cycles of adaptive moment estimation. Adam extends on stochastic gradient descent to solve non-convex problems faster while using fewer resources than many other optimization algorithms. It is most effective in extremely large datasets by keeping the gradients “tighter” over many learning iterations.

Adam combines the advantages of two other stochastic gradient techniques, Adaptive Gradients and Root Mean Square Propagation, to create a new learning approach to optimize a variety of neural networks.

5.3.2.3 Adam vs Stochastic Gradient Descent

With Stochastic Gradient Descent (SGD), a single learning rate (called alpha) is used for all weight updates. In addition, the learning rate for each network parameter (weight) does not change during training.

SGD separately calculates the individual adaptive learning rates for different weights by taking estimates of the first (the mean) and second (the uncentered variance) moments of the gradients.

Adam, on the other hand, adapts the parameter learning rates in real-time based on the average of the first and second moments by calculating an exponential moving average of the gradient as well as the squared gradient. Then the parameters beta1 and beta2 can control the decay rates of both moving averages. Finally, bias correction estimates are run before updating the learning parameters.

5.4 Convolutional Neural Network (CNN / ConvNet)

Convolutional neural networks are generally used in computer vision; however, recent advances showed promising results on various NLP tasks. Given the promising results we decided to use CNNs throughout all our meta-learning experiments.

Convolutional neural networks are very similar to ordinary neural networks. They are made up of neurons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity. The whole network still expresses a single differentiable score function: from the raw input at one end to class scores at the other end.

Neural networks receive an input (a single vector) and transform it through a series of *hidden layers*. Each hidden layer is made up of a set of neurons, where each neuron is fully connected to all neurons in the previous layer, and where neurons in a single layer function completely independent and do not share any connections. The last *fully connected layer* is called the “output layer” and in classification settings it represents the class scores.

Convolutional neural networks take advantage of the fact that the input consists usually of raw text and they constrain the architecture in a more sensible way. In particular, unlike a regular neural network, the layers of a ConvNet have neurons arranged in 3 dimensions: width, height, depth. (Note that the word depth here refers to the third dimension of an activation volume, not to the depth of a full neural network, which can refer to the total number of layers in a network.).

A ConvNet is a sequence of layers, and every layer of a ConvNet transforms one volume of activations to another through a differentiable function. We carved multiple ConvNets for our implementation but will take just one to describe the set-up.

In our CNN, we use different types of layers to build ConvNet architectures: *Embedding layer*, *Convolutional layer*, *Pooling layer*, *Batch Normalization*, *Dropout* and *Fully Connected layer* (much the way it is used in regular neural networks). We will stack these layers to form a full ConvNet architecture.

Example architecture of a CNN:

```
nn.Sequence(nn.Embedding()
            nn.Conv2D()
            nn.RELU()
            nn.MaxPool1D()
            nn.BatchNorm1d()
            nn.Conv2D()
            nn.RELU()
            nn.MaxPool1D()
            nn.BatchNorm1d()
            nn.Conv2D()
            nn.RELU()
            nn.MaxPool1D()
            nn.BatchNorm1d()
            nn.Dropout()
            nn.Linear()
            nn.SoftMax()
           )
```

A brief description of individual layers used in our model architectures is attached in the Appendix, chapter 10.6

5.4.1 Weight initialization

The weight initialization step can be critical to the model's ultimate performance, and it requires the right method. Initializing all the weights with zeros leads the neurons to learn the same features during training. In fact, any constant initialization scheme will perform very poorly. Consider a neural network with two hidden units, and assume we initialize all the biases to 0 and the weights with some constant α. If we forward propagate an input (x_1, x_2) in this network, the output of both hidden units will be ReLU ($\alpha x_1 + \alpha x_2$). Hence, both hidden units will have identical influence on the cost, which will lead to identical gradients. Thus, both neurons will evolve symmetrically throughout training, effectively preventing different neurons from learning different things. Choosing proper values for initialization is necessary for efficient training. A too large initialization leads to exploding gradients, while a too small initialization leads to vanishing gradients.

Therefore, to prevent the gradients of the network's activations from vanishing or exploding, we will stick to the following rules of thumb:

- The *mean* of the activations should be zero.
- The *variance* of the activations should stay the same across every layer.

There is a large variety of initialization methods. Xavier initialization works with *tanh* activations. But since we are using ReLU, a common initialization is *Kaiming He* initialization (He et al., Delving Deep into Rectifiers) [6], in which the weights are initialized by multiplying the variance of the Xavier initialization by 2.

In our models, we initialize all *biases* with constant weights 0.01 and *Batch Normalization* with a value 1. *Conv2D* and *Linear layers* have been initialized via Kaiming normal distribution.

5.4.2 Learning schedule

Learning schedule is a way to dynamically assign learning rates to an optimizer on given conditions, usually after a certain number of epochs. We did not apply a learning schedule in our meta model. Instead, we applied a static value.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification (2015). arXiv 1502.01852.

5.5 Bias-Variance

5.5.1 Bias

Bias is the difference between the average prediction of our model and the correct value which we are trying to predict. A model with high bias pays very little attention to the training data and oversimplifies the model. It always leads to high error on training and test data.

5.5.2 Variance

Variance is the variability of model prediction for a given data point or a value which tells us about the spread of our data. A model with high variance pays a lot of attention to training data and does not generalize on the data which it has not seen before. As a result, such models perform very well on training data but have high error rates on test data.

5.5.3 Bias- Variance trade-off

If our model is too simple and has very few parameters, then it may have high bias and low variance. On the other hand, if our model has a large number of parameters then it is going to have high variance and low bias. So, we need to find the right balance without overfitting and underfitting the data. This trade-off in complexity is why there is a trade-off between bias and variance. An algorithm cannot be more complex and less complex at the same time.

5.5.4 Potential approaches to prevent bias / variance in Neural Networks

Small neural networks, with fewer parameters are more prone to underfitting, but are computationally cheaper, while large neural networks are computationally more expensive, but are more prone to overfitting. There are strategies to approach the bias-variance problem.

Solving bias / variance problem			
	Fixing the high variance problem		Fixing the high bias problem
[1]	Get more training data	[5]	Apply additional features
[2]	Smaller set of features	[6]	Increase number of epochs
[3]	Increase λ (in the regularization)	[7]	Decrease λ (in the regularization)
[4]	Change the model architecture	[8]	Change the model architecture

Table 1: Approaches to prevent bias-variance

There are several solutions mentioned above that can take part in our solution in case of a bias-variance problem. The only thing not applicable is item [1]. The intention in our problem setting is to use as little data as possible. Other aspects, such as adding or removing features, in our case increasing or decreasing the number of words, are possible, but need to be handled with care since adding more features is computationally more expensive. In general, we are left with the possibility of changing the model architecture or using regularization (e.g., dropout, early stopping, etc.) in case of a bias-variance problem (see chapter 5.5.5).

5.5.5 Regularization

Regularization is a technique to reduce the complexity of the model. It does so by adding a penalty term to the loss function (cost function). The most common techniques are known as L1 (Lasso Regression) and L2 (Ridge Regression) regularization:

The L1 penalty aims to minimize the absolute value of the weights.

$$L(x, y) = \sum_{i=1}^{n} (h_\theta(x_i) - y_i)^2 + \lambda \sum_{i}^{n} |\theta_i|$$

The L2 penalty aims to minimize the squared magnitude of the weights.

$$L(x, y) = \sum_{i=1}^{n} (h_\theta(x_i) - y_i)^2 + \lambda \sum_{i=1}^{n} \theta_i^2$$

5.5.5.1 Dropout

Dropout proposed by (Geoffrey Hinton et al.)[7] is a regularization technique that prevents neural networks from overfitting. Regularization methods like L1 and L2 reduce overfitting by modifying the cost function. Dropout, on the other hand, modifies the network itself. It randomly drops neurons from the neural network during training in each iteration. When we drop different sets of neurons, it is equivalent to training different neural networks. The different networks will overfit in different ways, so the net effect of dropout will be to reduce overfitting. Dropout introduces an extra hyperparameter, the probability of retaining a unit. This hyperparameter controls the intensity of dropout. $p = 1$, implies no dropout and low values of p mean more dropout. Typical values of p for hidden units are in the range 0.5 to 0.8. For input layers, the choice depends on the kind of input.

5.5.5.2 Early stopping

"Early stopping", is also a regularization technique. The idea is very simple. The model optimizes the loss given the training data, by tuning the model parameters. The validation set, which contains unseen samples and is used for validation as we train the model, keeps a record of the loss function on the validation data. Model training is stopped, rather than training all the epochs, when there is no further improvement on the validation set. This strategy of stopping early based on the validation set performance is called Early Stopping. It does not only protect against overfitting, but it also needs a considerable smaller number of epochs to train a model.

[7] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting (2014).

6 Baseline classifiers

Measuring the performance of an algorithm given a dataset is just one side of the coin. A reference system is required to finally draw conclusions. Therefore, several machine learning as well as deep learning baseline classifiers have been created. There is no specific reason for choosing Random Forest, Multinominal Naïve Bayes or an XGBoost algorithm. All of them are well-known and so-called commodity models. The only requirement was to choose algorithms of different families to see if one type performs better than others. In our case, we used Random Forest as it is an ensemble technique, a Naïve Bayes because it uses a probabilistic classifier and a gradient boosting algorithm (ensemble meta-algorithm) for primarily reducing bias, and variance. The situation with deep learning models is a little different. I went the route of creating CNN, BERT and Sequential Models, since those are state-of-the art in text classification.

6.1 Experiment details

6.1.1 Dataset

HuffPost headlines consists of ~136'000 news headlines and short descriptions, published on HuffPost between 2012 and 2018 (Misra, 2018). Headlines and descriptions are shorter and less grammatical than formal sentences. Each headline/short description is assigned to one out of 41 classes, where each class has at least 800 samples. Some minor Explanatory Data Analysis is available in Chapter 10.2.1 HuffPost data analysis.

6.1.2 Data set-up

We apply an 80/20 split on the given dataset. In that case the training data receives ~109'000 samples while the test data contains ~27'000 samples. The dataset has been stratified. In that way we can ensure that the class distribution contains the same proportions in the training and test set.

6.1.3 Features

Features for machine learning models

Our machine learning and deep learning models will receive different feature sets. The original data has been pre-processed based on our NLP pipeline. There are two independent pre-processing pipelines for English and German texts (see chapter 10.1 Data pre-processing).

The pre-processed data has then been transformed into *word level tf-idf*, *n-gram level tf-idf, character level tf-idf vectors* and *count vectors* (see chapter 3.5.1 TF-IDF- and Count Vectors). These vector representations have then been applied to our ML models (Random Forest, Multinominal Naïve Bayes and XGBoost).

Features for deep learning models

Our deep learning baseline models receive the same pre-processed text data as our machine learning models, but instead of transforming the text into tf-idf vectors, we use 300 dimensional embeddings (GloVe for English and FastText for German). As with our machine learning models, certain transformations are required to work with our deep learning models. To transform our data, we use libraries from the TensorFlow (Keras) ecosystem. We created a vocabulary (word index) of size 20'000 by tokenizing our text data, created a mapping between the word indices and the embedding indices which resulted in an embedding matrix. The embedding matrix had then to be padded to ensure equal length vectors. The maximum sequence length of our features has been set to 150 words. Those vectors where then applied to our Sequence, CNN and BERT model.

6.2 Machine Learning Classifiers

6.2.1 Random Forest (RF)

Random Forest (RF) is an ensemble learning technique consisting of the aggregation of a large number T of decision trees, resulting in a reduction of variance compared to a single decision tree. In this paper we consider the original version of RF first described by Breiman (2001) [8], while acknowledging that other variants exist, for example, RF based on conditional inference trees (Hothorn et al., 2006) [9], which address the problem of variable selection bias investigated by Strobl et al. (2007) [10]. Our considerations are, however, generalizable to many of the available RF variants and other methods that use randomization techniques. A prediction is obtained for a new observation by aggregating the predictions made by the T single trees. In the case of regression RF, the most straightforward and common procedure consists of averaging the prediction of the single trees, while majority voting is usually applied to aggregate classification trees. This means that the new observation is assigned to the class that was most often predicted by the T trees.

Parameter settings Random Forest classifier	
Parameter	Setting
bootstrap	true
max_depth	80
max_features	"auto"
min_samples_leaf	3
min_samples_split	8
n_estimators	100
random_state	0

Table 2: Parameter settings Random Forest classifier

6.2.2 Multinominal Naïve Bayes (NB)

Naïve Bayes classifier is a probabilistic classifier based on Bayes' theorem, which assumes that each feature makes an independent and equal contribution to the target class. NB classifier assumes that each feature is independent and does not interact with others, such that each feature independently and equally contributes to the probability of a sample belonging to a specific class. NB classifier is simple to implement and computationally fast and performs well on large datasets having high dimensionality. NB classifier is beneficial for real-time applications and is not sensitive to noise. NB classifier processes the training dataset to calculate the class probabilities and the conditional probabilities, which define the frequency of each feature value for a given class value divided by the frequency of instances with that class value. NB classifier best performs when correlated features are removed because correlated features will be voted twice in the model leading to the overemphasis of the importance of the correlated features. The parameter settings for a Multinominal Naïve Bayes classifier are rather simple.

Parameter settings Multinominal Naïve Bayes classifier	
Parameter	Setting
alpha	0.1

Table 3: Parameter settings Multinominal Naïve Bayes classifier

[8] Leo Breimann. Random Forests (2001)

[9] Hothorn T, Hornik K., Zeileis A. Unbiased Recursive Partitioning: A Conditional Inference Framework (2006)

[10] Carolin Strobl, Anne-Laure Boulesteix, Achim Zeileis and Torsten Hothorn. Bias in random forest variable importance measures: Illustrations, sources and a solution (2007)

6.2.3 XGBoost (XGB)

XGBoost stands for “Extreme Gradient Boosting”, where the term “Gradient Boosting” originates from the paper Greedy Function Approximation: A Gradient Boosting Machine, by Friedman. XGBoost is a decision-tree-based ensemble machine learning algorithm that uses a gradient boosting framework. XGBoost algorithm was developed as a research project by Tianqi Chen and Carlos Guestrin in 2016[11]. The most important factor behind the success of XGBoost is its scalability in all scenarios. The system runs more than ten times faster than existing popular solutions on a single machine and scales to billions of examples in distributed or memory-limited settings. The scalability of XGBoost is due to several important systems and algorithmic optimizations. These innovations include: a novel tree learning algorithm is for handling sparse data; a theoretically justified weighted quantile sketch procedure enables handling instance weights in approximate tree learning. Parallel and distributed computing makes learning faster, which enables quicker model exploration.

Parameter settings XGBoost classifier	
Parameter	Setting
n_estimators	100
max_depth	5
min_child_weight	1
gamma	0
eta	0.3
subsample	0.8
colsample_bytree	0.8
objective	multi:softmax
nthread	4
learning_rate	1e-5
Epochs	50

Table 4: Parameter settings XGBoost classifier

6.3 Deep Learning Classifiers

The baseline models have been developed with the Keras Framework. Keras is a deep learning framework for Python that provides high-level building blocks to define and train almost any kind of deep learning model. It does not handle low-level operations, such as tensor manipulation and differentiation. Instead, it relies on a specialized, well-optimized tensor library, serving as the backend engine of Keras. The implementation at hand uses the TensorFlow backend. It allows the same code to run seamlessly on CPU or GPU. It has a user-friendly API that allows for quickly prototyping deep learning models. The Keras framework also provides further functions which need to be considered while developing deep learning classifiers such as callback, loss and optimizer function. Loss and optimizer functions have been discussed previously but will be mentioned again for the sake of completeness. The functions mentioned below have been applied to all our deep learning baseline models.

Callback function
When training a model, there are many things one cannot predict from the start. In particular, one cannot tell how many epochs will be needed to get to an optimal validation loss.

A much better way to handle this is to stop training when the validation loss in no longer improving. This can be achieved using a Keras callback function. A callback is an object (a class instance implementing specific methods) that is passed to the model. A callback is called by the model at various points during training. It has access to all model parameters which reflect the state of the model and its performance. A callback function can take actions as to interrupt training, save a model, load a different weight set, or otherwise alter the state of the model.

We applied two callback functions *ModelCheckpoint* and *EarlyStopping*. ModelCheckpoint takes the role of storing the best model while training. The EarlyStopping verifies if validation loss in no longer improving and stops training if such a condition appears. EarlyStopping is often used for regularization reasons.

Loss function
Keras provides a wide variety of loss functions to apply to models. A *cross-entropy loss function* is usually used for classification tasks, but this would require applying a one-hot encoded array to match the output shape. With the help of the *sparse categorical crossentropy loss function*, we can skip that step and keep integers as target labels.

11 Tianqi Chen and Carlos Guestrin. XGBoost: A Scalable Tree Boosting System (2016)

Optimizer function
Machine learning as well as deep learning algorithms require an optimization function in order to learn weights. Adam is an optimization algorithm that can be used instead of the classical stochastic gradient descent procedure to update network weights iterative based in training data. An important aspect for Optimization algorithms is the learning rate and the learning rate decay. These parameters need to be selected carefully (not big or to small) in such a way that the algorithm is able to converge.

6.3.1 Sequence Model (SEQ)

A sequential model groups a linear stack of layers into a model. A sequential model provides training and inference features in this mode. The model architecture has been intentionally kept simple. The applied architecture is listed in the table below. The output layer (dense layer) applies a softmax activation function with a vector size 41, which reflects the number of classes to predict.

6.3.1.1 Model architecture

Sequence model architecture based on Keras		
Layer (type)	Output Shape	# Parameters
Input layer	[(None, None)]	0
Embedding layer	(None, None, 300)	6'000'000
GlobalAveragePooling1D	(None, 300)	0
Dense (activation = relu)	(None, 128)	38'528
Dropout	(None, 128)	0
Dense (activation = softmax)	(None, 41)	5289
Total params	**Trainable params**	**Non-trainable params**
6'043'817	43'817	6'000'000

Table 5: Sequence model architecture

Parameter settings sequence model	
Parameter	Setting
Feature size	150
Optimizer	Adam
- learning rate / epsilon	1e-4 / 1e-8
Loss function	Sparse categorical crossentropy
Model Callbacks	ModelCheckpoint / EarlyStopping
Epochs	50

Table 6: Parameter settings sequence model

6.3.1.2 Sequence Model - results

The model has been trained on 50 epochs. The training accuracy at the end of 50 epochs is approximately. 62%, while the validation accuracy levels out at about 54% (see Figure 3). The accuracy of 62% compared to 54% seems feasible for a text classification task given the 41 classes. There are also several factors, that tend to influence the performance, such as the applied optimizer or learning rates.
The data set has not been balanced, but since we stratified the data, the proportion of classes and samples should not matter much. Both, model accuracy as well as the loss curve indicate a slight overfitting (bias-variance problem). There are several applicable options to approach a bias/variance problem. Such options are mentioned in chapter 5.5. Regularization is one such option. The applied dropout layer which takes the role of regularizing the model has a dropout factor $p = 0.5$ (dropping 50 percent of the neurons). Typical values of p for hidden units are in the range 0.5 to 0.8. So, it might be appropriate to change the factor p to 0.7, which is still in the range of typical values. Other options are changing the model architecture to a more sophisticated model, which might help, but also will enlarge training times.

We will take the given accuracy for granted for the time being, since the objective is not a benchmark. We rather take these values as a baseline to compare to our meta learning approach.

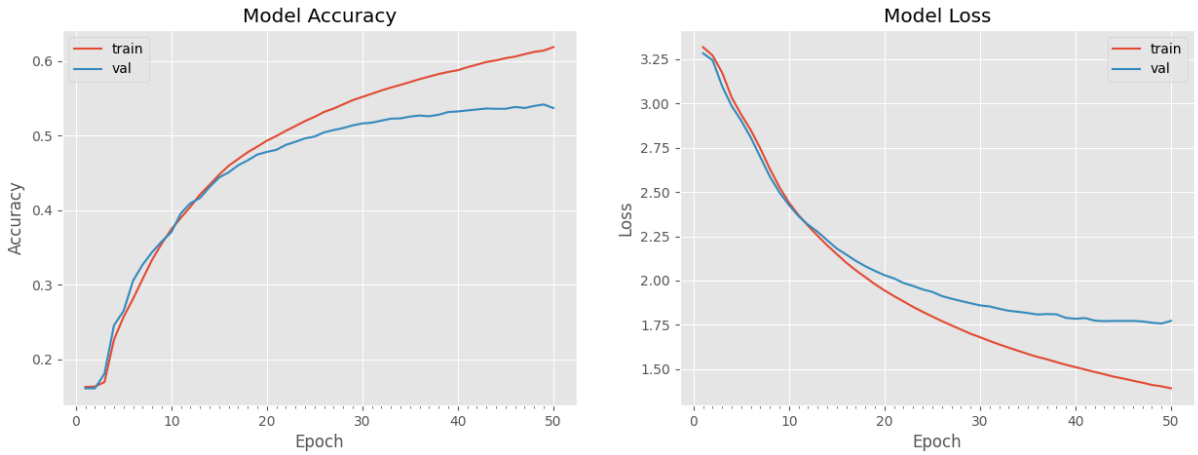

Figure 3: Sequence model accuracy, loss curves

6.3.2 Convolutional Neural Network (CNN, ConvNet)

A convolutional neural network, or CNN, is a deep learning neural network designed for processing structured arrays of data, such as images. CNNs are widely used in computer vision and reached state of the art in many visual applications, like image classification. In recent years CNNs have also shown groundbreaking results in the field of natural language processing. Such networks are excellent at picking up patterns. Unlike earlier algorithms, CNNs can directly operate on raw data (no tedious pre-processing required). A ConvNet is a feed-forward neural network, which often consists of more than 20 layers.

The magic in convolutional neural network comes from a special kind of layer called the convolutional layer. Convolutional neural networks contain many convolutional layers stacked on top of each other, each one capable of recognizing more sophisticated shapes.

6.3.2.1 Model architecture

The present CNN has intentionally been kept simple, so we are not supposed to learn too many parameters. It consists of 14 layers. Here we also use an embedding layer, which represents the words. Then there is 3 times the same structure of layers: Convolution, Batch Normalization and Pooling. Then again, as in the previous model architecture, a dense layer with a ReLU activation, a dropout layer and a dense layer with a softmax activation.

CNN Architecture based on Keras		
Layer (type)	Output Shape	# Parameters
Input layer	[(None, None)]	0
Embedding layer	(None, None, 300)	6’000’600
Conv2D	(None, None, 128)	192’128
BatchNorm2D	(None, None, 128)	0
MaxPooling1D	(None, None, 128)	0
Conv2D	(None, None, 128)	82’048
BatchNorm2D	(None, None, 128)	0
MaxPooling1D	(None, None, 128)	0
Conv2D	(None, None, 128)	82’048
BatchNorm2D	(None, None, 128)	0
GlobalMaxPooling	(Global (None, 128)	0
Dense (activation = relu)	(None, 128)	16’512
Dropout	(None, 128)	0
Dense (activation = softmax)	(None, 41)	5289
Total params	**Trainable params**	**Non-trainable params**
6’378’625	378’025	6’000’600

Table 7: Convolutional Neural Network architecture

Parameter settings CNN model	
Parameter	Setting
Feature size	150
Optimizer	Adam
- learning rate / beta 1 / beta 2 / weight decay	1e-3 / 0.9 / 0.99 / 1e-5
Loss function	Sparse categorical crossentropy
Model Callbacks	ModelCheckpoint / EarlyStopping
Epochs	50

Table 8: Parameter settings Convolutional Neural Network

6.3.2.2 Convolutional Neural Network – results

The model has been trained on 30 epochs. The training accuracy at the end of 30 epochs rises to approximately 80%, while the validation accuracy drops to about 53% (see Figure 4).

The training loss decreases, while the training accuracy increases with every epoch. That is what you would expect when running gradient descent optimization. The quantity we are trying to minimize should be less with every iteration. In our case, that applies while training the model. During validation, the validation loss starts rising at around the 8th epoch, while the accuracy starts dropping they diverge with every additional epoch. A model that performs better on the training data is not necessarily a model that will do better on data it has never seen before. The given model has an extreme overfit. We are overoptimizing on the training data and will end up learning representations that are specific to the training data but do not generalize to unseen data.

The best suited training accuracy is at about 62%, while the validation accuracy shows 57%. Those figures seem about right when we compare them with the other baseline models.

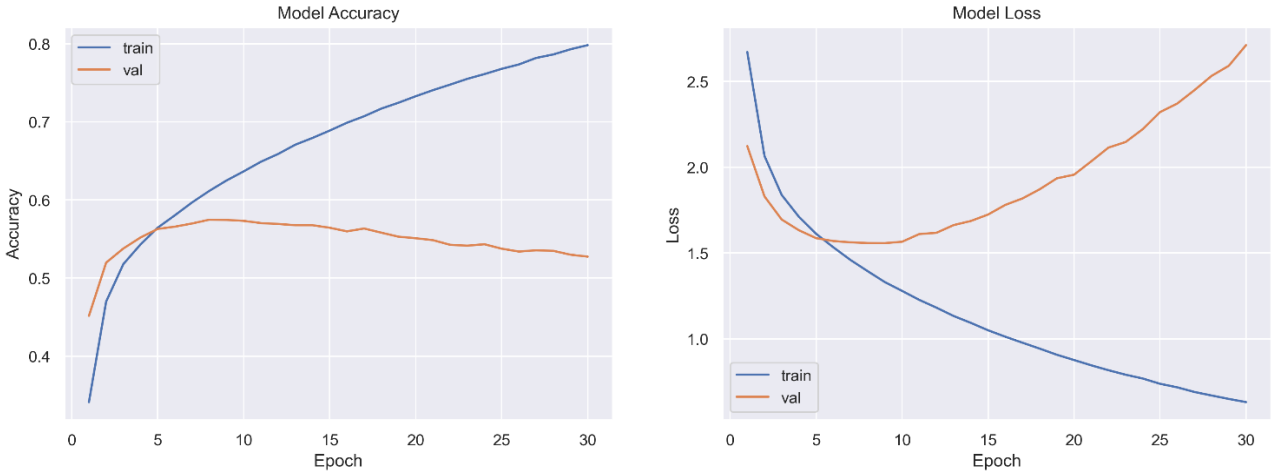

Figure 4: CNN model accuracy-, loss curve

6.3.3 Bidirectional Encoder Representations from Transformers (BERT)

2018 was a turning point in the field of natural language processing. A series of deep learning models appeared on the horizon that achieved state-of-the-art results on NLP tasks ranging from question answering, language translation to sentiment classification. Most recently, Google's BERT algorithm has emerged as a sort of "one model to rule them all", based on its superior performance over a wide variety of tasks.

BERT builds on two key ideas that have been responsible for many of the recent advances in NLP: the transformer architecture and unsupervised pre-training. The transformer is a sequence model that forgoes the recurrent structure of RNNs for a fully attention-based approach, as described in the instant classic *Attention Is All You Need*. [12] BERT's pre-trained weights are learned in advance through two unsupervised tasks, *masked language modelling* (predicting a missing word given the left and right context) and *next sentence prediction* (predicting whether one sentence follows another). BERT does not need to be trained from scratch for each new task; its weights are instead fine-tuned. Attention models are quite complex systems, but using a pre-trained model, makes the architecture simple as reflected below.

[12] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin. Attention Is All You Need (2018)

There is a slight difference compared to conventional input data preparation. Attention frameworks require 3 input layers (input_ids, input_mask, input_token) unlike normal neural networks. The inputs need to be pre-processed towards adding additional attention-related tokens [CLS] and [SEP] with each sentence.

6.3.3.1 Model architecture

BERT Architecture		
Layer (type)	Output Shape	# Parameters
Input layer (input_ids)	[(None, 50)]	0
Input layer (input_mask)	[(None, 50)]	0
Input layer (input_token)	[(None, 50)]	0
bert (TFBertMainLayer)	[(None, 768)]	109'482'241
Dense (activation = relu)	(None, 64)	49'216
Dropout	(None, 64)	0
Dense (activation = relu)	(None, 32)	2080
Dropout	(None, 32)	0
Dense (activation = softmax)	(None, 41)	1353
Total params	**Trainable params**	**Non-trainable params**
109'534'890	109'534'889	1

Table 9: BERT Architecture

The BERT experiment is only available as a colab notebook. The model has been trained on 10 epochs and a reduced feature size of 50, given the complexity of the model and the time consumption for training. The settings had to be reduced to be able to train the model within 12 hours. Google's Colab platform retrieves resources at the latest after 12 hours.

Parameter settings BERT model	
Parameter	Setting
Feature size	50
Optimizer	Adam
- learning rate / epsilon	1e-4 / 1e-8
Loss function	Sparse categorical crossentropy
Model Callbacks	ModelCheckpoint / EarlyStopping
Epochs	10

Table 10: Parameter settings BERT model

6.3.3.2 BERT Model - results

As mentioned before, the model has been trained on 10 epochs. The training accuracy at the end of 10 epochs is approximately 67%, while the validation accuracy reaches its top at about 61%. The validation accuracy where we need to focus on is about 8% higher than the one with the sequence model. This also makes sense. BERT models in general tend to show better performance given the pre-trained model. With this model as well, model accuracy and loss curve indicate an overfitting. The model starts overfitting around the 5th epoch. The loss curve shows the classical behaviour when a bias-variance problem exists. Ways to overcome are discussed in chapter 5.5.4.

BERT significantly improves classification performance, compared to the other baseline models. But in general, we expected an even better performance based on other experiences. The reason might be that news text is shorter and less grammatical than formal sentences. We postulate that this discrepancy arises because relation classification is highly contextual, while news text classification is mostly key-word based.

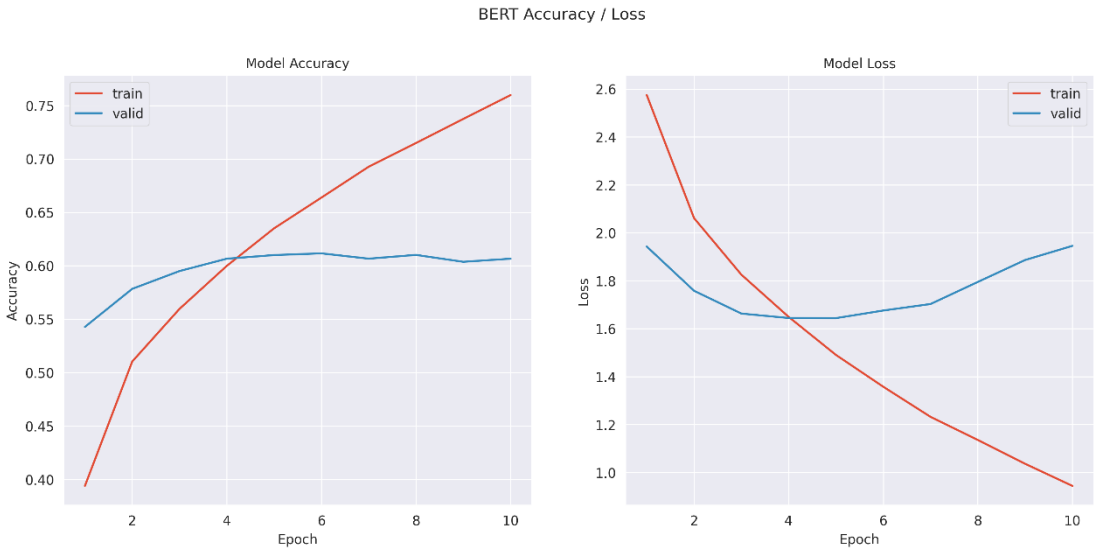

Figure 5: BERT model accuracy, loss curve

6.4 Baseline model results

The baseline models are meant to be a reference towards the representation of our meta-learning algorithms. All baseline models have been evaluated on the HuffPost Dataset (~136'000 samples, 41 classes). The conventional machine learning models received features in different forms as count vector and Word, n-gram, character level tf-idfs as described in a chapter 3.5.1. Deep learning models on the other hand receive raw-text data, words translated into a vector representation, so the models can cope with the given data.
The results for machine learning models show that none of the types, be it an ensemble, probabilistic or boosting algorithm, showed better results. All are in the same range with slight differences. It also indicates that applying different feature representations did not much improve in one over another case. None of the baseline models stands out. The accuracy ranges in the high 30s up to the mid 40s (39% - 45%).

The situation looks slightly different with the more sophisticated deep learning algorithms. The results here range from 53% up to 61%, about 8% - 16% higher than the ML models, considering the highest accuracy.
The models applied did not receive a thorough parameter fine-tuning. A fine-tuning cycle would certainly improve the accuracy of each individual model.

Contextualized representation (BERT)
In general, we expected an even better performance with the BERT algorithm (previous experience). A BERT model takes the contextual representation into consideration. The reason might be that news text is shorter and less grammatical than formal sentences. We postulate that this discrepancy arises because relation classification is highly contextual, while news text classification is mostly key-word based.

Results – Random Forest					
ML Model	Features	Accuracy [%]	Precision [%]	Recall [%]	F1_score [%]
Random Forest	Count Vector	35.80	38.50	57.10	38.50
	Word level	36.10	40.20	54.50	40.20
	N-gram level	16.70	22.40	55.30	22.40
	Char level	39.60	45.20	53.90	45.20
Best model	**Char level**	**39.60**	**45.20**	**53.90**	**45.20**

Results – Multinominal Naïve Bayes					
ML Model	Features	Accuracy [%]	Precision [%]	Recall [%]	F1_score [%]
Naïve Bayes	Count Vector	45.60	51.70	55.10	51.70
	Word level	43.20	49.80	56.20	49.80
	N-gram level	29.40	33.50	48.90	33.50
	Char level	36.90	43.30	50.20	43.30
Best model	**Count Vector**	**45.60**	**51.70**	**55.10**	**51.70**

Results – XGBoost					
ML Model	Features	Accuracy [%]	Precision [%]	Recall [%]	F1_score [%]
XGBoost	Count Vector	33.80	36.10	46.80	36.10
	Word level	43.20	49.80	56.20	49.80
	N-gram level	29.40	33.50	48.90	33.50
	Char level	36.90	43.30	50.20	43.30
Best model	**Word level**	**43.20**	**49.80**	**56.20**	**49.80**

Table 11: Results – machine learning models

Results – deep learning models				
DL Model	Accuracy [%]	Loss	Validation accuracy [%]	Validation loss
Sequence	61.69	1.3991	53.68	1.7728
CNN	80.55	0.6113	57.48	2.7111
BERT	66.54	1.3463	61.16	1.6756
Best model	**66.54**	**1.3463**	**61.16**	**1.6756**

Table 12: Results –deep learning models

7 Meta-Learning Experiment

The MAML paper explores the approach for multiple problems, such as regression, classification, and reinforcement learning. In this thesis we focus solely on text classification to gain a better understanding of the algorithm and investigate whether MAML really learns to adapt to novel tasks. We also intend to answer the question if Model Agnostic Meta-learning is a suitable approach for productive use and if it gives a better initialization for classification. We will apply two different datasets, HuffPost and Swiss Economic Archive (see chapter 7.1.1) to test the technique.

All of the experiments were performed using ***PyTorch*** (which allows for automatic differentiation of the gradient updates) along with the Higher library. ***Higher***[13], is a library providing support for higher-order optimization, e.g., through unrolling first-order optimization loops of "meta" aspects of such loops. It provides tools for turning existing “torch.nn.Module” instances "stateless", meaning that changes to the parameters can be tracked, and gradient steps with regard to intermediate parameters can be taken. It also provides a suite of differentiable optimizers, to facilitate the implementation of various meta-learning approaches. It extends PyTorch to enable the easy and natural implementation of such meta-learning approaches at scale. There are also other libraries available on the market, like *Torch-Meta (Facebook)*, *learn2learn* or *Meta-datasets (Google)*. Those have also been considered. Torch-Meta (Deleu et al., 2019) [14], for example, provides a library aiming to assist the implementation of meta-learning algorithms, supplying useful data loaders for meta-training. However, unlike the convenience provided by Higher, it requires re-implementation of models using their functional/stateless building blocks, and to reimplement the optimizers in a differentiable manner. Therefore, the Higher library seemed the most promising.

7.1 Experiment details

7.1.1 Dataset

HuffPost headlines consist of ~136’000 news headlines and short descriptions, published on HuffPost between 2012 and 2018 (Misra, 2018). Headlines and descriptions are shorter and less grammatical than formal sentences. Each headline/short description is assigned to one out of 41 classes, where each class has at least 800 samples. Some minor explanatory data analysis is available in Chapter 10.2.1 HuffPost data analysis.

The 41 classes were randomly split into tasks for $D_{meta-train}$, $D_{meta-valid}$ and $D_{meta-test}$. The training and evaluation were performed on the meta-training and meta-validation set. Where the meta-validation set was mostly used for hyperparameter tuning. The meta-test set measured the generalization to new tasks.

HuffPost class split	
$D_{meta-train}$	20 classes
$D_{meta-valid}$	10 classes
$D_{meta-test}$	11 classes

Table 13: HuffPost class split

[13] Edward Grefenstette, Brandon Amos, Denis Yarats, Phu Mon Htut, Artem Molchanov, Franziska Meier, Douwe Kiela, Kyunghyun Cho, Chintala Soumith. Generalized Inner Loop Meta-Learning (2019). arXiv:1910.01727

[14] Tristan Deleu, Tobias Würfl, Mandana Samiei, Joseph Paul Cohen, Yoshua Bengio. Torchmeta: A Meta-Learning library for PyTorch (2019). arXiv 1909.06576

Swiss Economic Archive (SWA) consists of news articles published in various news media starting from 1950. The news articles are split among 485 classes. Multiple labels are assigned for certain news articles. The focus in this thesis is not to predict multiple labels per class. Therefore, two datasets have been created. Dataset 1 contains only samples that have an assignment to a single class. Hence, Dataset 1 contains 146 classes, where Dataset 2 uses the 146 classes with a single class assignment, but also samples where 2 class assignments existed, but only the first of the two class assignments have been used. This leaves us with 246 distinct classes in dataset 2. We will initially stick to dataset 1 with its 146 classes.

Swiss Economical Archive class split:

Swiss Economic Archive class split	
$D_{meta-train}$	100 classes
$D_{meta-valid}$	20 classes
$D_{meta-test}$	26 classes

Table 14: Swiss Economic Archive class split

7.1.2 Data set-up

We consider the N-way, k-shot classification setting where a meta-learner trains on many related but small training sets of k samples for each of the N classes (5-way/10-shot). We first split the list of all classes in the data into disjoint sets and assign them to each meta-set of meta-training, meta-validation, and meta-testing dataset $D = (D_{train}; D_{valid}; D_{test}) \in \mathcal{D}$.

A dataset, e.g., training set, consists of any number of batches. A batch is formed by 32 tasks. To generate each instance of a N-class, k-shot task dataset, we initially create a book-keeping dictionary, which contains a class-sample index mapping (samples that belong to a respective class). Then ran through the following steps:

1) Sample N classes from the list of classes corresponding to the meta-set we consider $(D_{train}; D_{valid}; D_{test})$.
2) Sample k examples from each of those classes. These k samples per N-classes composes a task in the training set D_{train}.
3) Compose a batch, by aggregating 32 tasks into a batch
4) The number of batches varies for meta-train, meta-valid and meta-test

The same procedure is then applied for D_{valid} and D_{train}. The difference between a training, validation, and test set lies in the number of batches, while for training 1000 batches are allocated, we only allocate 250 each for validation and test set. An experiment consists of the following setting:

Training, validation, test set			
Unit	Type	(5-way/10-shot)	(5-way/5-shot)
1	task	50 samples	25 samples
1	batch	32 tasks	32 tasks
1000	batch	32 tasks à 50 samples {1000} * 32 * 50 = 1'600'000 samples	32 tasks à 25 samples {1000} * 32 * 25 = 800'000 samples

Table 15: Data setup – training, validation, test set

Time consumption for training, validation, and testing

In terms of time consumption, we can allocate the following figures. The times greatly depend on the size of model and can, therefore, vary slightly. On average the time to process a (5-way/10-shot) batch is about ~30sec. resp. 18sec. for a (5-way/5-shot). The figures are based on the given system configuration: Nvidia GeForce RTX3080 and CUDA 11.1

Training, validation, and test time consumption				
Phase	(5-way/10-shot)		(5-way/5-shot)	
Training	1000 * [27…32] sec	~7.5 - 9h	1000 * [15…23] sec	~4 – 6.5h
Validation	250 * [27…32] sec	~2 – 2.5h	250 * [15…23] sec	~1 – 1.5h
Testing	250 * [27…32] sec	~2 – 2.5h	250 * [15…23] sec	~1 – 1.5h

Table 16: Time consumption – model training

7.1.3 Features

PyTorch has been used for all meta-learning models as mentioned in the introduction. So, I used the torchtext, library a package out of the PyTorch ecosystem. Torchtext provides a set of convenience functions to handle NLP tasks. We used torchtext because it nicely integrates with PyTorch. It allows to quickly apply embeddings and create a vocabulary.
The steps to create features were in general the same as we did with our baseline models. We also use the pre-trained FastText (Joulin et al., 2016) [15] and GloVe (stanford.edu) embeddings. The embeddings are applied depending on the used language in a respective data set (GloVe for English and FastText for German). The mapping between our embedding matrix and the vocabulary is done by torchtext behind the scenes. We also use a vocabulary size of 20'000 words and set the maximum sequence length of our features to 150 words.

7.2 Model architecture

In terms of model architecture, 5 hand-woven convolutional networks have been created. For that purpose, multiple convolutional blocks comprised of Convolutional, MaxPooling, Dropout, Batch Normalization and Fully Connected layers have been stacked on top of each other.

The individual models differ in minor ways. Some of the models are comprised of 1D convolutional layers, while the other use 2D convolutional layers. One uses Batch Normalization, while the others do not. Some models use a different number of filters. Common to all models is the use of a ReLU as intermediate activation function and a "softmax" activation function on the last layer to distinguish between several classes. A stride of 1 and a cross-entropy loss function to calculate the error has been applied to all models.

Model architectures		
Model	Trainable parameters	Layer
Meta-Model 1	332'405	3 x Conv2D, MaxPooling, Dropout, Linear layer, filter size (100)
Meta-Model 2	373'241	3 x Conv1D, MaxPooling, BatchNorm, Dropout, Linear layer, filter size (100)
Meta-Model 3	361'805	3 x Conv1D, MaxPooling, Dropout, Linear layer, filter size (100)
Meta-Model 4	362'405	3 x Conv2D, MaxPooling, Dropout, Linear layer, filter size (128)
Meta-Model 5	994'205	6 x Conv2D, MaxPooling, Dropout, Linear layer, filter size (100)

Table 17: Model architectures

7.2.1 Hyperparameter evaluation

As we saw previously, learning occurs in two stages: gradual learning is performed across tasks, and rapid learning is performed within tasks. This requires two learning rates, which introduces difficulty in choosing hyperparameters that would help to achieve training stability. A parameter optimization cycle has been applied to find the best settings. Finding the optimal parameters is a tedious and time-consuming task but finding the best set of optimization parameters is vital and might lead to substantially better results. Meta-learning algorithms are also suitable for finding optimal parameters, but that technique has not been applied yet. Parameter fine-tuning is more of an art than an exact science. With more experience, one tends to get a gut feeling as to which settings might be better suited than others. In my own case, I lack the experience as well as the required time to run a full-fledged tuning cycle. Despite this, the following ranges have been verified:

Hyperparameter evaluation	
SGD learning rate	[0.1, …, 0.0001]
SGD momentum	[0.5, …, 0.9]
Adam learning rate	[0.01, …, 0.0001]
Adam beta 1	[0.5, …, 0.9]
Adam beta 2	[0.5, …, 0.99]

Table 18: Hyperparameter evaluation

[15] Armand Joulin, Edouard Grave, Piotr Bojanowski, Matthijs Douze, Hérve Jégou, Tomas Mikolov. FastText.zip: Compressing text classification models (2016). arXiv 1612.03651

The optimizer settings which seemed best are listed in the table below:

Hyperparameter settings						
Data set	Learning loop	Optimizer	Learning Rate	Momentum	Beta 1	Beta 2
HuffPost	Inner loop	SGD	0.1	0.9		
	Meta loop	Adam	0.001		0.9	0.99
SWA	Inner loop	SGD	0.005	0.9		
	Meta loop	Adam	0.001		0.9	0.99

Table 19: Hyperparameter settings

7.2.2 Model-Agnostic Meta learning algorithm

The Model-Agnostic Meta-Learning algorithm is the central part of this thesis. The following pseudo code describes the mechanism of our meta-learning application (see Figure 6: MAML pseudo code). A detailed implementation of the algorithm using the Higher library is attached in the appendix chapter 10.7.
The MAML approach fine-tunes its model using gradient descent each time for a new task. This requires it to backpropagate the meta-loss through the model's gradients, which involves computing derivatives of derivatives, i.e., second derivatives. While the gradient descent at test time helps it extrapolate better, it does have its costs. Since we use the Higher library, we do not have to worry about second derivatives. All is covered by the library.

Algorithm: MAML for few shot supervised learning

Require: $p(T)$: distribution over tasks
Require α, β: step size hyperparameters

Randomly initialize θ
while not done **do**
  Sample batch of tasks $T_i \sim p(T)$
  for all T_i **do**
    Sample K datapoints $D = \{x^i, y^i\}$ from T_i
    Evaluate $\nabla_\theta \mathcal{L}_{T_i}(f_\theta)$ using D and T_i [1]
    Compute adapted parameters with gradient descent: $\theta_i^* = \theta - \alpha \nabla_\theta \mathcal{L}_{T_i}(f_\theta)$ [2]
    Sample datapoints $D_i' = \{x^i, y^i\}$ from T_i for meta update [3]
  end for
  Update: $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{T_i \sim p(T)} \mathcal{L}_{T_i}(f_\theta)$ using each D_i' and $\mathcal{L}_{T_i}$ [4]
end while

Figure 6: MAML pseudo code

We take k datapoints (support and query set). We optimize the likelihood of the support set by taking a few gradient steps with respect to the model's parameters. In our case, we apply 5 inner gradient steps on the support set. This adapts the model's meta-parameters to the task. Higher, which takes a central role in managing the model's parameters, is able to automatically keep copies of the network's parameters as they are being updated. The final set of adapted parameters will induce some final loss and accuracy (loss and accuracy are logged as mentioned later) on the query set. These will be used to update the model's meta-parameters during backpropagation.

Finally, we update the model's meta-parameters to optimize the query losses across all of the tasks sampled in this batch. This unrolls through the gradient steps.

Gradient steps:
Backpropagating through many inner steps can be computationally and memory intensive. With only a few gradient steps, it might be a less time-consuming endeavor, but it may not be the best solution for scenarios that require a higher number of gradient steps at test time. So, we applied a medium-sized inner gradient step value of 5 to balance the situation.

Logging of loss and accuracy
Our implementation keeps track of the loss and accuracy on a task and batch level. This is done via the two simple methods. Task-level logging contains the *query loss* and *query accuracy* per task, while the data stored on batch level contains averaged values of all tasks in a batch. This makes the curves smoother and hence more readable. Task and batch metrics are stored in csv files, for visualization reasons.

```
task_metric  = per_task_metric(task_cnt, query_loss, query_accuracy, mode)
batch_metric = per_batch_metric(batch_idx, batch_loss, batch_accuracy, mode)
```

7.3 Meta-training, validation, and testing

Following Vinyals et al. (2016), we consider the k-shot, N-class classification setting where a meta-learner trains on many related but small training sets of k examples for each of N classes. We first split the list of all classes in the data into disjoint sets and assign them to each meta-set of meta-training, meta-validation, and meta-testing. During inference, we use the meta-trained model to predict on the meta-test set, except this time although the meta-trained model undergoes additional gradient steps to help classifying the query set examples, the learner parameters are not updated.

Meta-training, validation and testing will be performed on a 5-way/10-shot and 5-way/5-shot setting. The assignment of classes to a train, validation, and test set is usually a random assignment. Table 20 shows an initial random assignment of classes into training, validation, and testing. The given class assignment will from now on be kept static, to ensure comparability between the different models and settings. The setting will not be 100% equivalent, since we cannot control the sample assignment to a given class. Those will always be random. Building on the idea of predicting seen as well as unseen classes, we formulate 2 scenarios.

Meta-training, validation, testing class split	
Training classes	[13, 1, 25, 6, 12, 24, 8, 28, 18, 19, 22, 9, 33, 3, 40, 30, 34, 0, 11, 2]
Validation classes	[29, 16, 14, 39, 7, 17, 21, 4, 32, 35]
Testing classes	[5, 36, 15, 38, 23, 27, 20, 37, 26, 10, 31]

Table 20: Random class split

Validation, in a conventional meta-training, validation, and test setting, is used for fine-tuning. Table 20 shows that our class distribution for training, validation and testing is indeed disjoint. We will change the conventional procedure towards the following settings:

Meta-training, validation, testing class split	
Training classes	[13, 1, 25, 6, 12, 24, 8, 28, 18, 19, 22, 9, 33, 3, 40, 30, 34, 0, 11, 2]
Validation classes	[29, 16, 14, 39, 7, 17, 21, 4, 32, 35]
Test classes	[13, 1, 25, 6, 12, 24, 8, 28, 18, 19, 22, 9, 33, 3, 40, 30, 34, 0, 11, 2, **5, 36, 15, 38, 23, 27, 20, 37, 26, 10, 31**]

Table 21: Class split scenario 1

Scenario 1:
Validation is more an academic scenario, where we only validate on unseen classes. In a production setting we will almost never have the situation that during inference only unseen classes need to be predicted. The usual case during inference is, that seen as well as unseen classes need to be identified. Therefore, our test cycle will contain a combination of seen and unseen classes. For that purpose, we merge the 20 classes assigned to training and the 11 classes assigned to testing (bold) to form our final test set. It will be a mixture of seen and unseen samples, while neither classes nor samples out of the test class assignment (bold) have ever appeared during training.

Scenario 2:
In our 2nd scenario, we train on samples from 30 classes (training and validation of the initial class assignment) and perform testing on 41 classes. In that case we cover all 41 classes (30 classes seen, 11 classes unseen).

Meta-training, validation, testing class split	
Training classes	[13, 1, 25, 6, 12, 24, 8, 28, 18, 19, 22, 9, 33, 3, 40, 30, 34, 0, 11, 2, 29, 16, 14, 39, 7, 17, 21, 4, 32, 35]
Testing classes	[13, 1, 25, 6, 12, 24, 8, 28, 18, 19, 22, 9, 33, 3, 40, 30, 34, 0, 11, 2, 29, 16, 14, 39, 7, 17, 21, 4, 32, 35, **5, 36, 15, 38, 23, 27, 20, 37, 26, 10, 31**]

Table 22: Class split scenario 2

7.3.1 Evaluation Metrics

We will only use accuracy (acc) to evaluate the classification performance, in all our text classification experiments. A confusion matrix is often used in machine learning and specifically by visualizing statistical classification. A confusion matrix, also known as an error matrix, is a specific table layout that allows the visualization of the performance of an algorithm, typically in supervised learning. Confusion matrices are very useful when the performance of a handful of classes need to be visualized. A graph becomes confusing, when visualizing 41 classes as in our case.

7.3.2 Results on HuffPost dataset

The following section describes how MAML performs on the 5-way/10-shot and a 5-way/5-shot setting on the HuffPost dataset.

In both sections, we illustrate accuracy and loss curves for training, validation, and testing. The individual plots yield 4 curves which reflect the individual models and their respective performances.

As mentioned before, training has been taken on 1'000 tasks, which adds up to 1'600'000 in a 5-way/10-shot and 800'000 datapoints in a 5-way/5-shot setting, while we only applied 250 tasks for validation and testing which on that side adds up to 400'000 respectively 200'000 datapoints. One such setting qualifies for an epoch. The figure below illustrates a training cycle of 1 epoch. The HuffPost dataset does itself not contain that many samples. The 1'000 tasks yield different combinations of 5 classes and their respective samples.

We expect that a 10-shot setting yields better results than a 5-shot setting on both the meta-validation and meta-test set. Therefore, as the number of shots/examples per class increase, we probably will see a better performance during validation and test time.

Restriction:
Training only on a single epoch (time consumption).

Experiment summary:
Detailed performance figures of the individual models are available in a table in chapter 7.5.

Remark:
Figure 7 is just for illustration reasons. The 2 plots top left and middle depict, in fact, the same data. One shows the loss values per task, while the other is an averaged loss value over all tasks in a batch. It is hard to derive any information from a task-level plot. Therefore, we will only use batch-level data from now on.

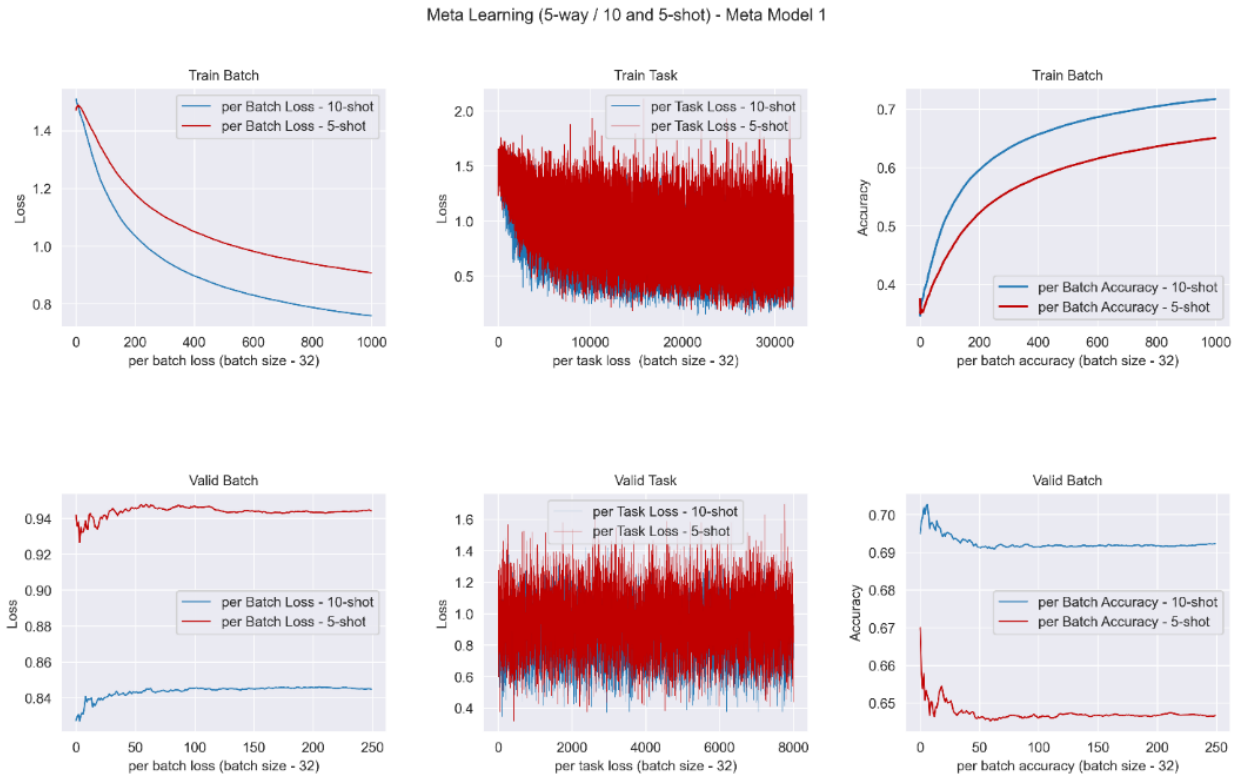

Figure 7: MAML task vs. batch level loss curve

7.3.2.1 Meta-training, validation, and testing 5-way/10-shot

The results described in this section reflect scenario 1. The procedure is described in chapter 7.3.

Scenario 1:
Meta-training on 20 classes (seen classes) and meta-validation on 10 classes (unseen classes). In meta-testing we perform tests on 31 classes (20 seen classes from meta-training and 11 unseen classes from meta-testing)

Meta-training, validation, testing class split	
Training classes	[13, 1, 25, 6, 12, 24, 8, 28, 18, 19, 22, 9, 33, 3, 40, 30, 34, 0, 11, 2]
Validation classes	[29, 16, 14, 39, 7, 17, 21, 4, 32, 35]
Testing classes	[13, 1, 25, 6, 12, 24, 8, 28, 18, 19, 22, 9, 33, 3, 40, 30, 34, 0, 11, 2, **5, 36, 15, 38, 23, 27, 20, 37, 26, 10, 31**]

Meta-training:
The individual models show a relatively diverse performance. The performance after one cycle ranges from ~64% up to ~72%. The complexity of the individual models rises with the number assigned to each model name. This means meta_model_1 is a rather simple model, while meta_model_5 has a higher complexity. The initial assumption was that the model performance would improve with additional layers. As we add more layers, the model should be able to learn different (more diverse aspects) from the given data. But our assumptions seem to be incorrect. The model with the lowest complexity reaches the best training accuracy, while meta_model_5, which has about twice the number of convolutional layers, does not do very well. But in fact, we should not focus on the training accuracy too much. The more important performance measures are the results with the validation and test set.

The one thing all our models have in common is the fact that the loss curve gradually degrades. Training loss decreases, while the training accuracy increases with every epoch. That is what one would expect when running gradient descent optimization. The quantity we are trying to minimize should be less with every iteration. The expectation is that we see an asymptotic behaviour towards 0. This certainly applies only as we run more training cycles. Meta_model_5 fails in this case. The loss curve levels out after training about 600 tasks.

Figure 8: Meta-learning 5-way/10-shot accuracy, loss curves

Meta-validation:
The validation accuracy verifies how good the learned weights effectively are. We apply unseen classes and samples in the validation cycle, as mentioned in the introduction to this chapter. Here we see a relatively static behaviour of the individual models. The figures below show the same sequence as we saw in the training phase. Meta_model_1 yields the best result, while meta_model_5 has the lowest performance. All curves show a slightly lower performance than in the training cycle. In conventional machine learning models this would indicate a slight overfitting. A model that performs better on the training data is not necessarily a model that will do better on data it has never seen before. When we see overfitting, we overoptimize on the training data, and one ends up learning representations that are specific to the training data and do not generalize to data outside of the training set.
In a meta-learning context, we would imply that our models generalize well. Consider meta_model_1 which yields a training accuracy of ~72% compared to the ~69% during validation or if we consider meta_model_3 with training/validation accuracy of ~70% / ~68% This in general indicates a robust model that generalizes well (see all results in chapter 7.5 Experiment summary).

Figure 9: Meta-validation 5-way/10-shot accuracy, loss curves

Meta-testing:
The meta testing results are somewhat interesting. We merged the 20 classes the model has been trained on (seen classes) with the 11 unseen classes, which we initially put aside. So, we test on 31 classes. We expected the performance to be somewhere between the training and the validation results, a little lower than the training performance, but better than the validation results. The figures yield a different view. The results, and this virtually applies to all models, are even a little higher than the training performance, which seems rather odd.

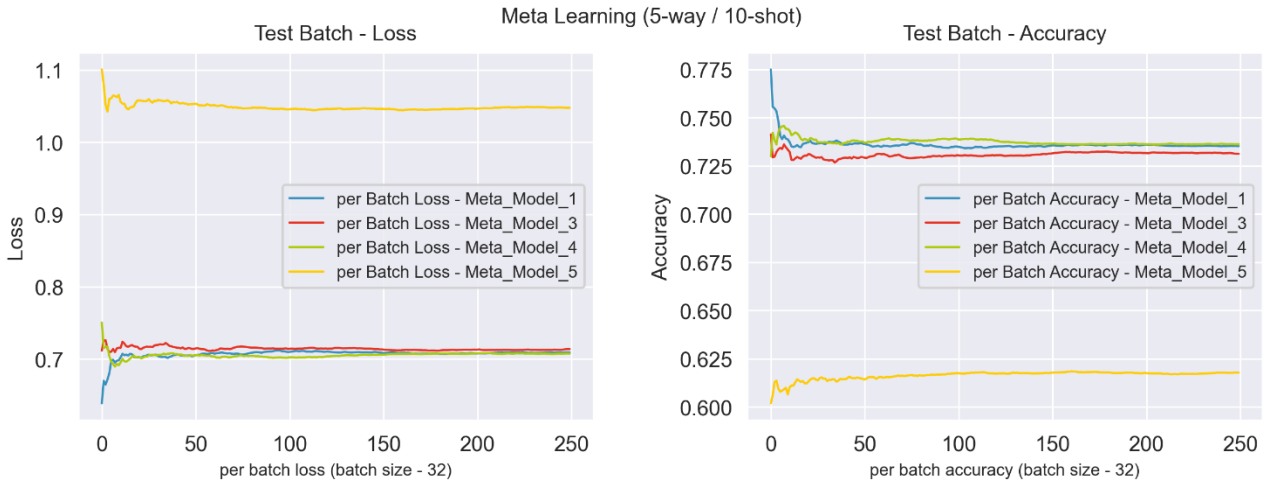

Figure 10: Meta-testing 5-way/10-shot accuracy, loss curves

7.3.2.2 Meta-training, validation, and testing 5-way/5-shot

The results described in this section reflect scenario 1. The procedure is described in chapter 7.3.

Meta-training:
The performance with 5-way/5-shot settings is, as we expected in the introduction, lower than the ones with the 5-way/10-shot setting. The performance after one cycle ranges from ~59% up to ~65%, which is about ~5% -~7% lower than in the 10-shot case. The situation here looks a little different. Meta_model_5, which did not perform well in the previous setting, tells a completely different story. It shows almost the same performance as our simple model (meta_model_1). A reason might be, that the newly assigned samples contain words, that have a better predictive power and therefore can better be allocated to a given class. In general, all loss curves degrade gradually as one would expect.

Figure 11: Meta-training 5-way/5-shot accuracy, loss curves

Meta-validation:
The meta-validation tells us an equivalent story as described in the above section. Our meta_model_1 generalizes quite well. The meta-training and meta-validation performance of that model is virtually the same (meta-train: ~65.07%, meta-validation: ~64.72%), while the other models slightly overfit. **This proves, that given the prior knowledge, we in fact can predict entirely new classes very well.**

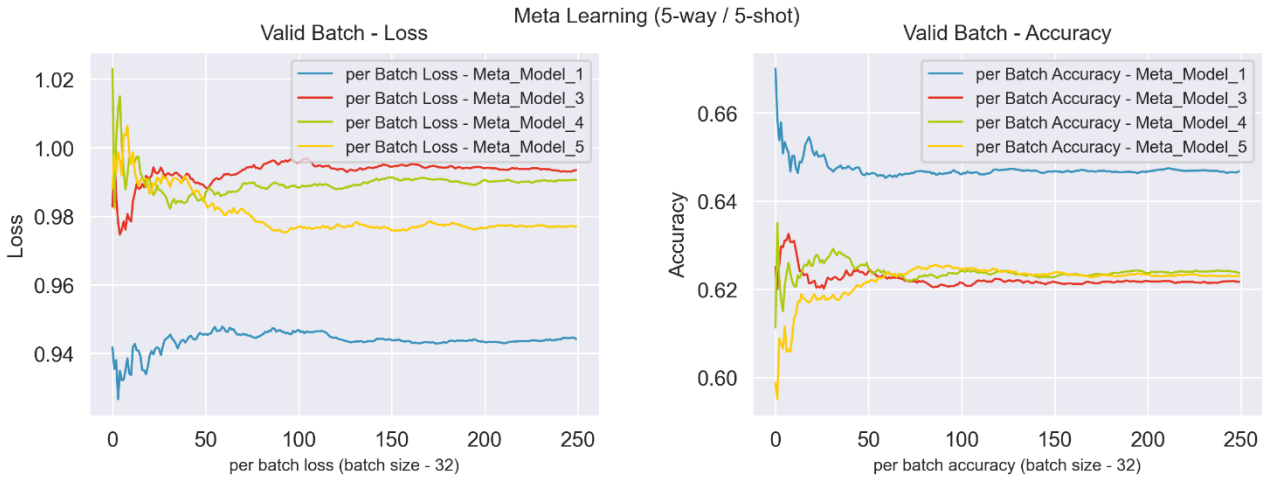

Figure 12: Meta-validation 5-way/5-shot accuracy, loss curves

Meta-testing:
Our expectation with the 5-shot setting is the same as with the 10-shot setting. We expected the performance to be somewhere between the meta-training and the meta-validation results. A little lower than the training performance, but better than the validation results. Meta_model_4 shows an equivalent performance between meta-training and meta-validation, while meta_model_1 and meta_model_5 even exceed the meta training performance. This basically yields an indication that a slight bias exists.

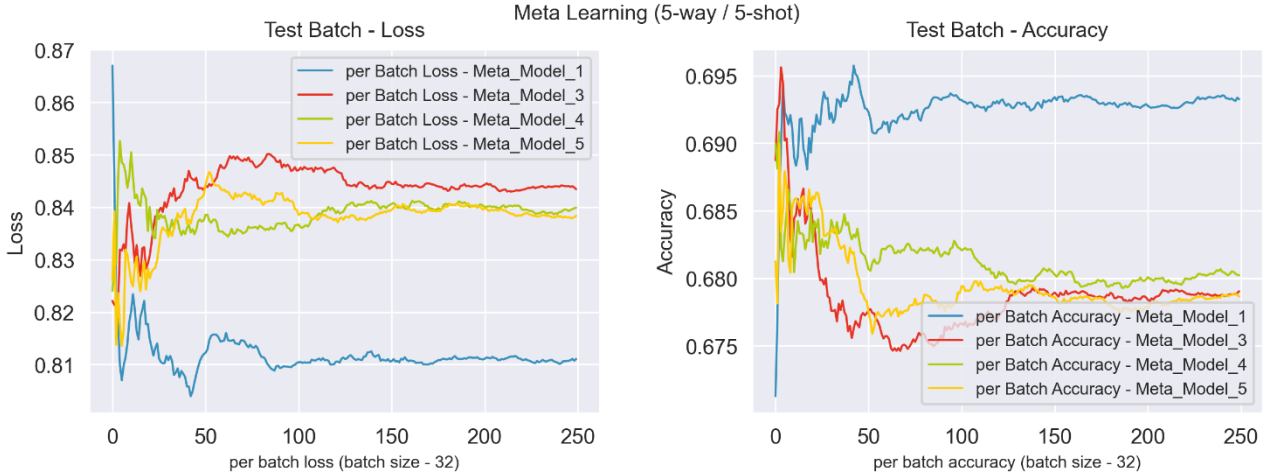

Figure 13: Meta-testing 5-way/5-shot accuracy, loss curves

7.3.2.3 Meta-training, and testing 5-way/10-shot

Scenario 2:
In our 2nd scenario, we train on samples from 30 classes (training and validation) and perform testing on 41 classes. In that case we cover all 41 classes in our dataset (30 classes seen, 11 classes unseen).

Meta-train, -validation, -test class split	
Train classes	[13, 1, 25, 6, 12, 24, 8, 28, 18, 19, 22, 9, 33, 3, 40, 30, 34, 0, 11, 2, 29, 16, 14, 39, 7, 17, 21, 4, 32, 35]
Test classes	[13, 1, 25, 6, 12, 24, 8, 28, 18, 19, 22, 9, 33, 3, 40, 30, 34, 0, 11, 2, 29, 16, 14, 39, 7, 17, 21, 4, 32, 35, **5, 36, 15, 38, 23, 27, 20, 37, 26, 10, 31**]

Meta-training:
In scenario 2 as we defined above or the introduction in chapter 7.3, we apply the full stack of classes. We see a similar situation as with our scenario 1 in chapter 7.3.2.1. The assumption that the training accuracy is smaller on the 30 classes than on the 20 classes, does not hold true. The spread of training accuracy is tighter over all models. The lowest training accuracy improved by ~3% while the top accuracy dropped by ~2% compared to the similar situation in scenario 1. Accuracies range from ~67% up to ~70%.

Figure 14: Meta training 5-way/10-shot accuracy, loss curves

Meta-testing:
For our final inference scenario we preload each individual meta-trained model. As before our models undergo additional gradient steps to help classifying the query set examples, but the learner's parameters are not updated. The result depicted below shows a rather positive image. The test accuracies range from ~73.6% – ~75%.
It is actually incredible how good the learned weights are, if we consider the fact that only 10 samples per class have been used. Initially I had the feeling something is wrong with the setup, but if we consider the data preparation setup, everything is plain vanilla.

Our application provides 3 independent data loaders (training, validation, and test data loader). In case of training, the data loader takes 30 classes and assigns 10 samples per class. Our book-keeping method ensures, that a class does not get the same sample assigned twice. In fact, we assign 20 samples per class, but they are split 50/50 into a support and query set later on, to ensure that they are coming from the same class distribution.

The test data loader does the same thing but on 41 classes. The 11 unseen classes for sure get samples the system has never seen before, while the remaining 30 classes might get samples it has seen before. But when we consider our dataset. The number of samples per class range from ~800 up to >20'000 samples. By even only considering the least number of samples of ~800 in one class, we can be sure that the overlay (match) is minor.

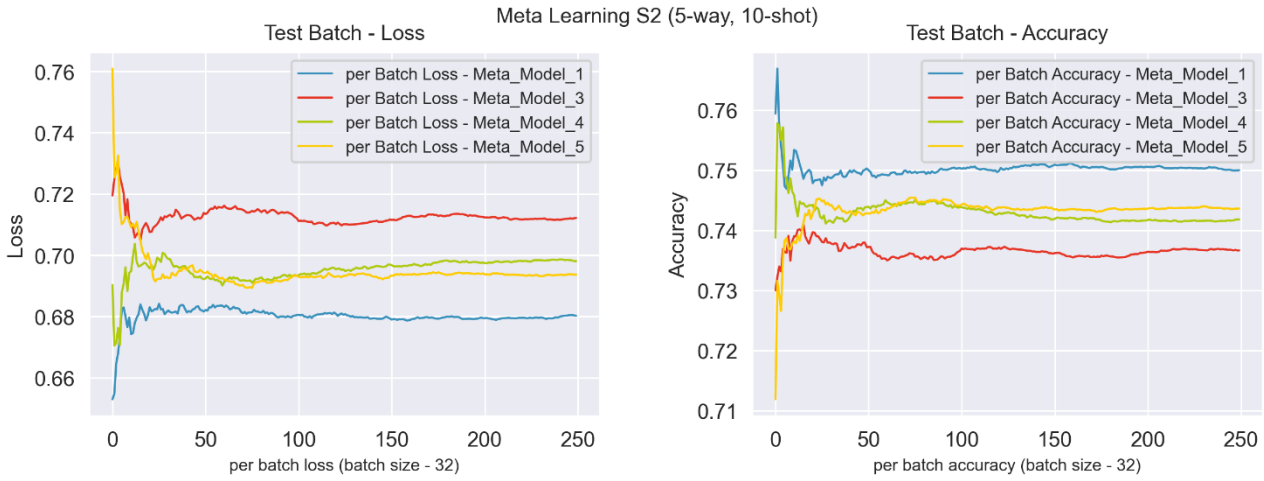

Figure 15: Meta-testing 5-way/10-shot accuracy, loss curves

7.3.2.4 Meta-training, and testing 5-way/5-shot

Meta-training:
The performance with 5-way/5-shot settings is as we expected, lower than the ones with the 5-way/10-shot setting. The performance after one cycle ranges from ~59% up to ~62%, which is about ~8% lower than in the 10-shot case. The situation here looks a little different. Meta_model_5 which did not perform well, shows almost the same performance as our simple model (meta_model_1). Here as well, we can only assume if the reason is related to the newly assigned samples and the respective words. The fact, that it behaves the same way as in the 5-way/5-shot scenario trained on 20 classes, does not support our assumption. So, there must be a different reason. In general, all loss curves degrade gradually as one would expect.

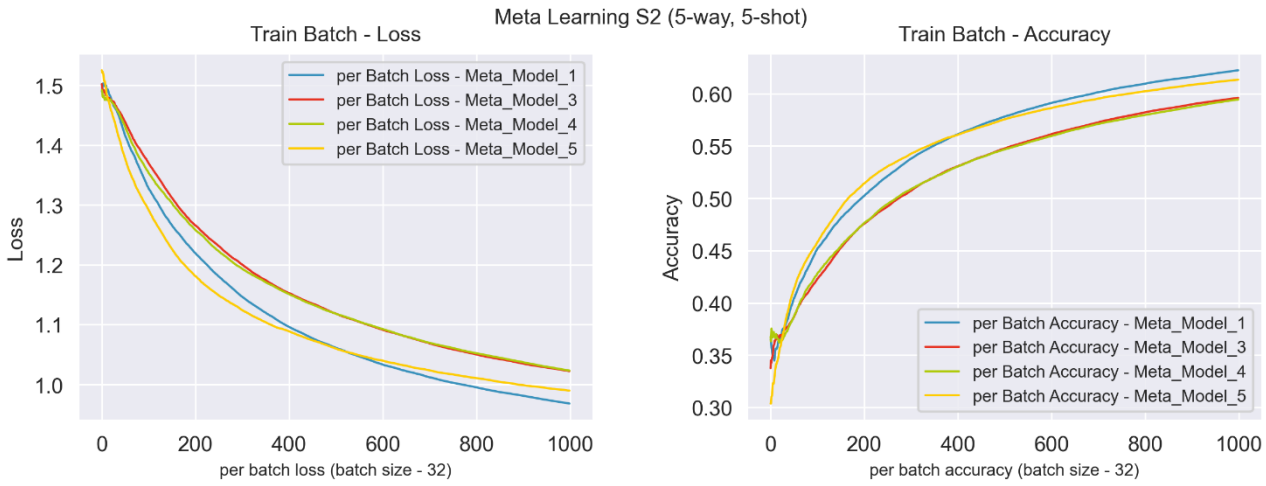

Figure 16: Meta-training 5-way/5-shot accuracy, loss curves

Meta-testing
In this setting we see again the expected behaviour. Accuracies with 5-shots are lower than with 10-shots. Accuracies range from ~66.7% up to ~69.5%. The loss is basically a mirror of the accuracy. Highest accuracies have the lowest loss and vice versa.

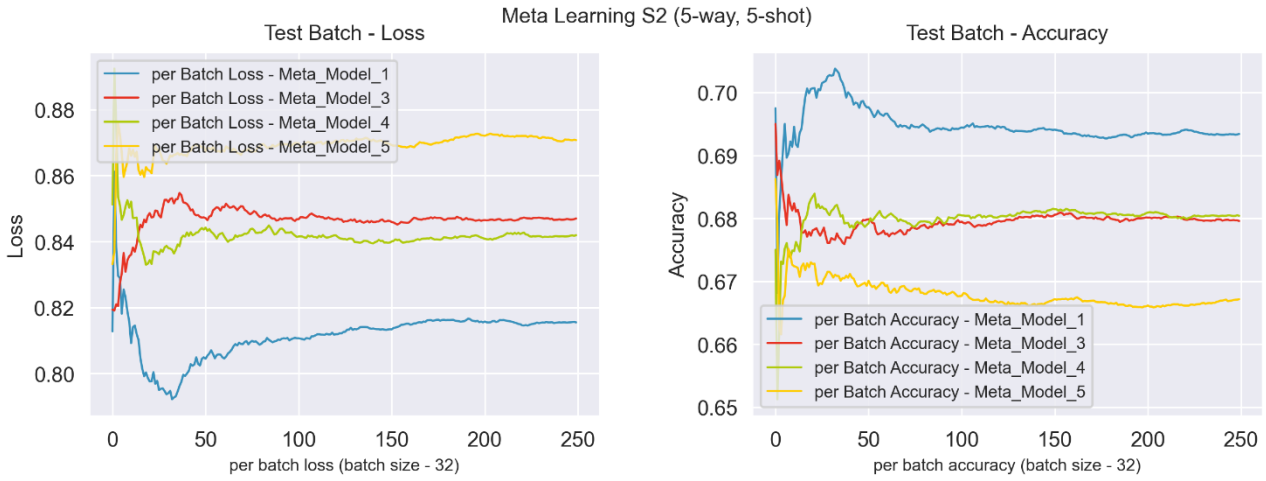

Figure 17: Meta-testing 5-way, 5-shot accuracy, loss curves

7.3.3 Results on SWA dataset

The results on the Swiss Economic Archive dataset had to be skipped given the time constraint to finalize this thesis. We only ran one training cycle, which resulted in a training accuracy of ~58%.

7.4 Production setup

When one consults the currently available research papers on MAML in an N-way/few-shot setting, one will not find any hint on how MAML is supposed to be used in a production set-up. I racked my brain on that subject, but the only solution I could come up with, was that the MAML N-way/few-shot pre-trained models can be used as a good initialization in a transfer learning context. But I was not sure if that assumption really holds true. So, I wrote my question to two researchers (Sergey Levine[16] and Antreas Antoniou[17]) with the expectation not to get any feedback. Interestingly, I got an invitation from Antreas for a video call.

We discussed the subject on "How to apply MAML in a productive setting", which would suit as a good title for a research paper. It was a fruitful discussion, where I received some valuable hints. He also mentioned that the subject is not discussed anywhere in the research community.

The following points came out of our discussion:

- Use MAML as a well-trained starting point for transfer learning by retraining our model on a newly assigned output layer (softmax), which covers all available classes.
- Build a model architecture that provides an output layer (softmax), which covers all available classes and apply meta-learning as an N-way/few-shot setting.
- Generating softmax parameters for task-specific classification, by dynamically allocating a changing output layer (softmax) while meta-training (depending on the number of classes in each task).

Transfer learning
Transfer learning would be a straightforward task. The primary step is to load a pre-trained model, then we need to obtain access to all layers and parameters. Having access, we can freeze the respective layers by setting the parameters *requires_grad* to *False*. This would prevent calculating the gradients for these parameters in the backward step, which in turn prevents the optimizer from updating them. And finally apply the new output layer.

Generating softmax parameters for task-specific classification
Existing applications of MAML consider few-shot learning with a fixed N-way assignment. This limits the applicability to a reduced number of applications. Classification tasks, which would need a different set of N classes with each individual task could not be covered. The remedy to this, would be to generate task-dependent softmax parameters (both linear weights and bias). Given the training data, $D_i^{tr} = \{(x_i, y_{i)}\}$, for a task T_i, we first would need to partition the input into the N_i number of classes for the task (available in D_i^{tr}) an then obtain the softmax classification weights and bias for task T_i. Currently I have no clear picture how to address this situation. I would need to add more brain power into it, since many open questions remain as calculating and concatenating the weights and biases, class remapping etc.

Model architecture with a 41-class output layer
For our final experiment we will apply approach 2 from the list above. It is the fastest way, given our model architecture. We just need to provide the number of classes in the output layer (softmax) covering all 41 classes and switching off the remapping by creating the $D_{meta-train}$ and $D_{meta-test}$. Remapping of classes is required when applying a 5-way classification with a model architecture containing an output layer to distinguish between 5 classes. Since we will cover all classes in our current setting, a remapping of classes for meta-training and validation, as we did in a conventional 5-way classification, will not be required.

[16] Chelsea Finn, Pieter Abbeel, Sergey Levine. Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks (2017)

[17] Antreas Antoniou, Harrison Edwards, Amos Storkey. How to train your MAML (2019).

7.4.1 Meta-training and testing 5-way/10-shot

We will perform meta-training on the same classes as in scenario 2. We again run experiments on 5-way/5-shot and 5-way/10-shot. The only difference is that the output layer (softmax) of our model consists now of 41 classes. A remapping is not required in such a setting as mentioned above.

Meta-training, validation, testing class split	
Training classes	[13, 1, 25, 6, 12, 24, 8, 28, 18, 19, 22, 9, 33, 3, 40, 30, 34, 0, 11, 2, 29, 16, 14, 39, 7, 17, 21, 4, 32, 35]
Test classes	[13, 1, 25, 6, 12, 24, 8, 28, 18, 19, 22, 9, 33, 3, 40, 30, 34, 0, 11, 2, 29, 16, 14, 39, 7, 17, 21, 4, 32, 35, **5, 36, 15, 38, 23, 27, 20, 37, 26, 10, 31**]

Meta-training

Meta-training in our production scenario should show around the same results as in the previous section scenario 2, since we use the same class initialization. There could be a slight difference because of different sample assignments and certainly weight initializations. The meta-training accuracy of our models is in the range of ~71% to ~74%, which is, surprisingly, a few percent higher than the previously trained models in scenario 2. Our assumption here was that we will get similar performance or even a little less. When it comes to the training accuracy of our simplest and most complex model, they are again in a comparable range.

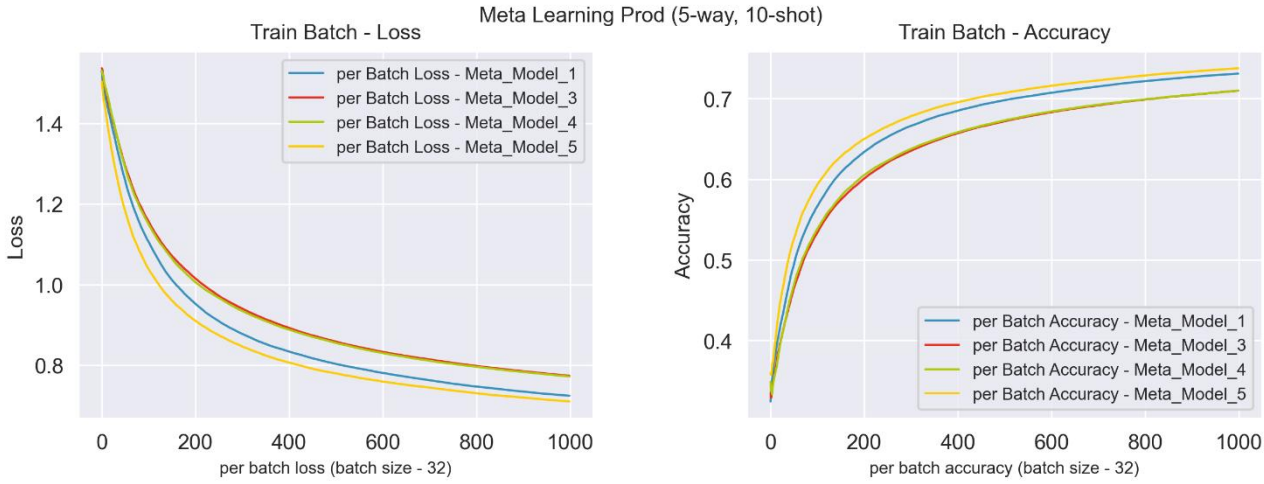

Figure 18: Meta-training production 5-way/10-shot accuracy, loss curves

Meta-testing
When we now compare the meta-testing performance, we see a different situation. The model performance of all models drops to a range between ~58% and ~59.6%, which in my view is still surprisingly good. If we compare our BERT baseline model, which reached an accuracy of ~61%. A model, which we in fact consider a state-of-the-art model only shows a ~1.5% improvement compared to our best meta-trained model.
Taking into consideration the number of samples we applied to our meta model (10 samples per class), I would call our performance near state-of-the-art. One thing to recognize is the fact, that our simplest model again performs best. The remaining question to answer here is the difference between meta-training and meta-testing performance. The models in general were rather robust and did generalize quite well, as we saw in previous experiments. So, why is there such a drop?

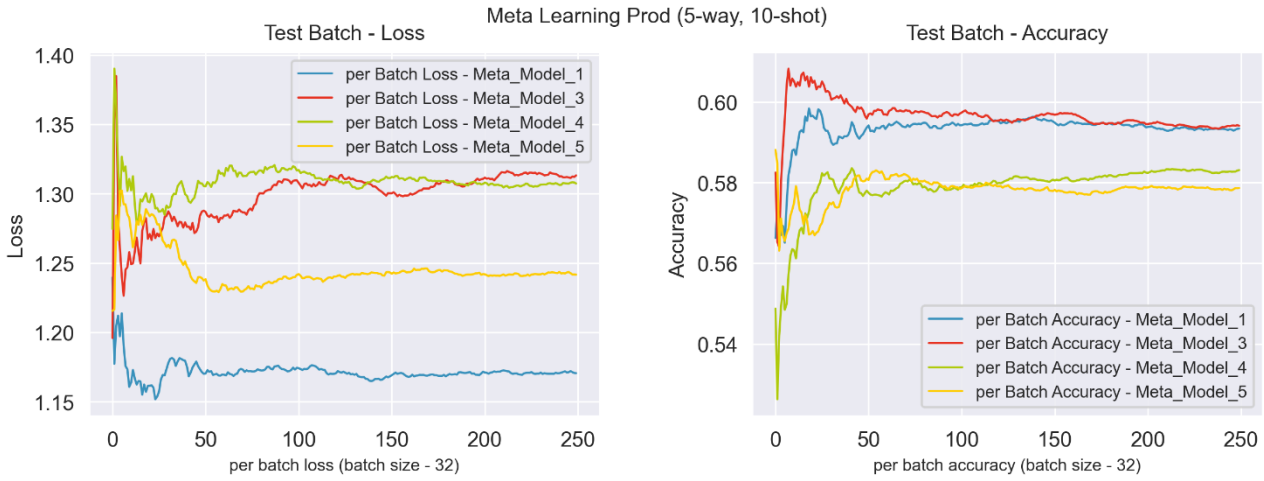

Figure 19: Meta-testing production 5-way/10-shot accuracy, loss curves

7.4.2 Meta-training and testing 5-way/5-shot

Meta-training
We assume to see the same situation with our 5-shot models as we described in the meta-training section for 10-shots. Meta-training is around ~64.6% to ~67.6% which is ok, but the more interesting figures are the testing results.

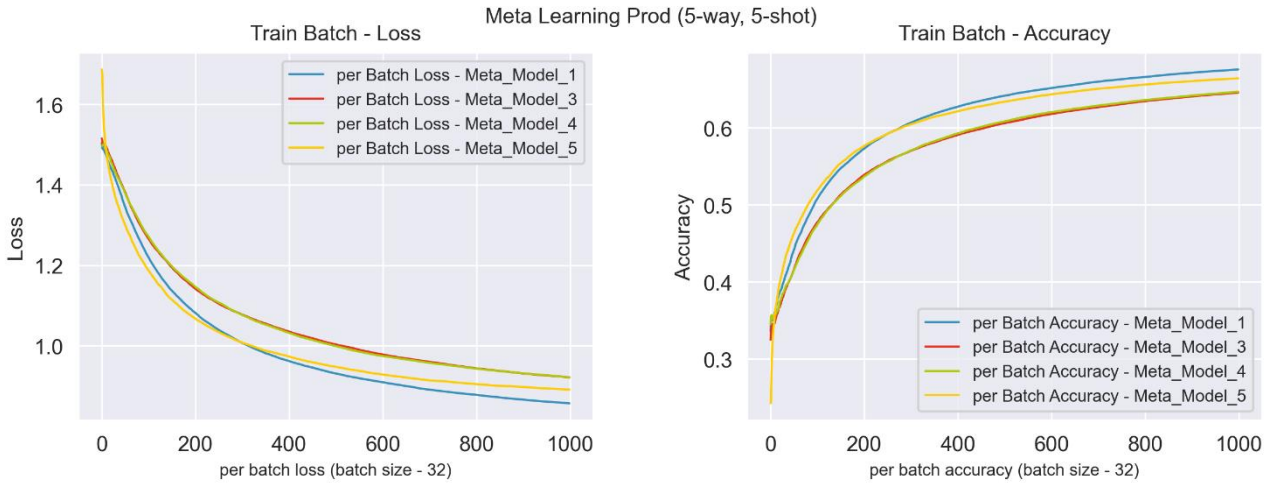

Figure 20: Meta-training production 5-way/5-shot accuracy, loss curves

Meta-testing
The spread of meta-testing results is wider than before. Results are around ~53.2% to 58.7%, which gives a spread of around 5%. If we compare the performance of our best model with 10-shots and 5-shots, we see only a small difference of ~1%. When we now check again our best 5-shot result and compare them to the BERT baseline model, we see again a rather positive situation. Our BERT model only performs better by ~2.3%.

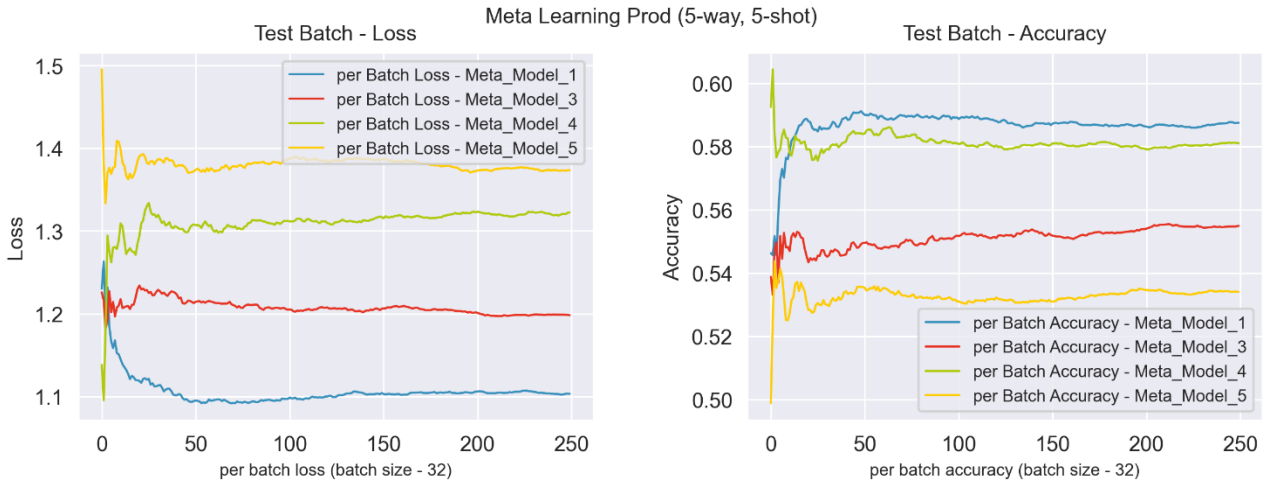

Figure 21: Meta-testing production 5-way, 5-shot accuracy, loss curves

7.5 Experiment summary

The following section aggregates the results acquired in chapter 6 Baseline classifiers and 7 Meta-Learning Experiment. The six baseline classifiers have been evaluated in a conventional machine, and deep learning setting, where an 80/20 split on the data has been applied to separate training from test data. In chapter 7 we evaluated four CNN models in a meta-learning setting where we applied only 5 or 10 samples per class.
A summary of all results is reported in the tables below.

Conventionally trained machine learning models show a rather poor performance with certain model architectures, while the deep learning baseline models reach a substantially better performance. So, if we put the baseline results into context to our few-shot learning results, we see a comparable performance (compared to deep learning baseline models) or even exceeded the performance with the MAML few-shot approach.

Baseline models

Results – Baseline machine learning models					
ML Model	Features	Accuracy [%]	Precision [%]	Recall [%]	F1_score [%]
Naïve Bayes	Count Vector	45.60	51.70	55.10	51.70
Random Forest	Char level	39.60	45.20	53.90	45.20
XGBoost	Word level	43.20	49.80	56.20	49.80
Best model	**Word level**	**45.60**	**51.70**	**55.10**	**51.70**

Results – Baseline deep learning models				
DL Model	Train accuracy [%]	Loss	Validation accuracy [%]	Validation loss
Sequence	61.69	1.3991	53.68	1.7728
CNN	80.55	0.6113	57.48	2.7111
BERT	66.54	1.3463	61.16	1.6756
Best model	**66.54**	**1.3463**	**61.16**	**1.6756**

Meta-learning - scenario 1

Results - 5-way/10-shot	scenario 1			scenario 2	
Meta Models	Max train acc [%]	Mean valid acc [%]	Std Dev	Mean test acc [%]	Std Dev
meta_model_1	71.7434	69.2425	0.1890	73.6086	0.3433
meta_model_3	69.7435	68.1698	0.1479	73.0847	0.1455
meta_model_4	69.8745	69.7306	0.1720	73.7491	0.1754
meta_model_5	63.8355	57.4486	0.1746	61.6377	0.2256
Best model	**69.8745**	**69.7306**	**0.1720**	**73.7491**	**0.1754**

Results - 5-way/5-shot		scenario 1		scenario 2	
Meta Models	Max train acc [%]	Mean valid acc [%]	Std Dev	Mean test acc [%]	Std Dev
meta_model_1	65.0735	64.7239	0.2272	69.2529	0.2001
meta_model_3	62.2178	62.2155	0.1860	67.8646	0.3187
meta_model_4	62.2435	62.3846	0.1823	68.1202	0.1727
meta_model_5	64.7922	62.2000	0.4087	67.9395	0.2416
Best model	**65.0735**	**64.7239**	**0.2272**	**69.2529**	**0.2001**

Meta-learning – scenario 2

Results - 5-way/10-shot			
Meta Models	Max train acc [%]	Mean test acc [%]	Std Dev
meta_model_1	68.9872	75.0241	0.1518
meta_model_3	67.2384	73.6493	0.1155
meta_model_4	66.7008	74.2973	0.2247
meta_model_5	69.8295	74.3173	0.2881
Best model	**69.8295**	**74.3173**	**0.2881**

Results - 5-way/5-shot			
Meta Models	Max train acc [%]	Mean test acc [%]	Std Dev
meta_model_1	62.2355	69.4641	0.3215
meta_model_3	59.6042	67.9522	0.1791
meta_model_4	59.4348	67.9966	0.2616
meta_model_5	61.3382	66.7902	0.2369
Best model	**62.2355**	**69.4641**	**0.3215**

Meta-learning production setting

Results - 5-way/10-shot			
Meta Models	Max train acc [%]	Mean test acc [%]	Std Dev
meta_model_1	73.0900	59.3255	0.4404
meta_model_3	70.9728	59.6567	0.4441
meta_model_4	70.9813	57.8953	0.6950
meta_model_5	73.7565	57.8065	0.3329
Best model	**70.9728**	**59.6567**	**0.4441**

Results - 5-way/5-shot			
Meta Models	Max train acc [%]	Mean test acc [%]	Std Dev
meta_model_1	67.5878	58.6563	0.5934
meta_model_3	64.5836	55.1179	0.3209
meta_model_4	64.6915	58.1110	0.2419
meta_model_5	66.426	53.2604	0.3014
Best model	**67.5878**	**58.6563**	**0.5934**

Our results demonstrate that the **MAML approach** is indeed **beneficial for learning** with only a **few samples**. In the conventional 5-way scenario (meta-train and validation – scenario 1), the tasks indicate an accuracy of around 69% mean validation accuracy on just 10 samples, while the scenario with 5 samples shows an accuracy of approximately 65%. We also see that our second approach with a mixture of seen and unseen classes during meta-testing shows even better results (~73% on 10 samples and ~69% on 5 samples). Such results are remarkable.

When we used **MAML** in a **productive setting**, meaning that we intend to classify 41 classes instead of just 5 classes, the performance dropped down to about 59.6%. But we need to put those results into perspective. Our BERT baseline model achieved an accuracy of ~61%. A model, which we in fact consider a state-of-the-art model. Taking into consideration the number of samples we applied to our meta model (10 samples per class), I would call **our performance near state-of-the-art**. But the question why we experience such a difference between meta-training and meta-testing performance, remains. The models in general were rather robust and did generalize quite well, as we saw in previous experiments.

We were able to reproduce the advantages of MAML by using the prior knowledge to improve its future learning performance. We even experienced that MAML is able to gain experience over multiple learning tasks mostly covering a distribution of related tasks and being able to classify unseen classes.

Does that imply that we learn a better weight initialization with the MAML approach?
I would say, yes.
Given the fact that the MAML algorithm is simple, and its ability to perform a few gradient steps at inference time allowing to generalize quickly to unseen classes (derived from prior knowledge), is remarkable.

Does a productive setup benefit with the MAML approach?
Here as well, I would answer with yes. Our results indicated comparable results in our production set-up. Using the MAML pre-trained weights by applying them in a transfer learning context probably allows to achieve even better results in a productive setting.

There was an important finding while training and testing our approach. The dropout layer, which is used for regularization in the way that it modifies the network by removing neurons during training, contributes to the stability and robustness of a network. In the testing phase, a dropout layer should be eliminated since the learned parameters have already been trained for robustness. Removing the dropout layer during testing was beneficial. It added 4% - 6% to the final accuracy in either setting (5 and 10-shot).

Final words regarding our experiment results:

Meta-learning and especially the MAML approach shows exactly what I intended to achieve. It provides a possibility to relieve us from the data-gathering burden and reduces compute resources. It is certainly not perfect, but the future seems bright.

There is one last approach, which I regret not being able to finalize “yet”. One of my favourite technique is model ensembles. I personally believe, combining ensembles technique and meta-learning could show favourable results. The subject is described in the next chapter.

7.6 Ensemble learning for Deep Learning Neural Networks

A powerful technique for obtaining the best possible results on a given task is an ensemble of models.
Deep learning neural networks are nonlinear methods. They offer increased flexibility and can scale in proportion to the amount of training data available. A downside of this flexibility is that they learn via a stochastic training algorithm, which means that they are sensitive to the specifics of the training data and may find a different set of weights each time they are trained, which in turn produces different predictions. Generally, this is referred to as neural networks having a high variance and it can be frustrating when trying to develop a final model to use for making predictions. A successful approach to reducing the variance of neural network models is to train multiple models instead of a single model and to combine the predictions of these models. This is called ensemble learning and not only reduces the variance of predictions but can also result in predictions that are better than any single model. The idea behind ensembles is that different models learn different areas from the data. Combining these models usually leads to a better performance.

Combining several models is most helpful when the individual models are different from each other and in order to make neural net models different, they should either have different architectures or be trained on different data. However, the obvious idea of averaging the outputs of many separately trained nets is prohibitively expensive.

7.6.1 Ensemble techniques

Ensemble techniques are mainly separated into algebraic and voting methods. We only listed four of the most interesting methods, but there are several more.

Ensemble techniques	
Algebraic methods	Voting methods
Averaging	Majority voting
Weighted Average Ensemble	Weighted majority voting

Table 23: Ensemble techniques

Majority voting
The majority voting method is generally used for classification problems. In this technique, multiple models are used to make predictions for each data point. The predictions by each model are considered as a ‘vote’. The predictions which we get from the majority of the models are used as the final prediction.

Averaging
Similar to the majority voting technique, multiple predictions are made for each data point in averaging. In this method, we take an average of predictions from all the models and use it to make the final prediction. Averaging can be used for regression as well as classification problems.

Weighted average
Weighted average or weighted sum ensemble is an ensemble approach that combines the predictions from multiple models, where the contribution of each model is weighted proportionally to its capability or skill.
The weighted average ensemble is an extension of the averaging method.

Weighted majority voting
The weighted majority voting is a combination of the weighted average and the majority voting method.

8 Challenges and ways to overcome

8.1 MAML / MAML++

The MAML approach fine-tunes its model using gradient descent each time for a new task. This requires it to backpropagate the meta-loss through the model's gradients, which involves computing derivatives of derivatives, i.e., second derivatives. While the gradient descent at test time helps to extrapolate better, it does have its costs.

Backpropagating through many inner steps can be compute and memory intensive. With only a few gradient steps, it might be a less time-consuming endeavor, but it may not be the best solution for scenarios that require a higher number of gradient steps at test time. That said, the authors of the MAML paper also propose a first-order approximation that eliminates the need to compute the second derivatives, with a comparable performance. Another closely related approach is Reptile. Reptile builds on first order MAML but does not need to split the task into support and query sets, making it a natural choice in certain settings. However, approaches to reduce computation time while not sacrificing generalization performance are still at the experimental stage.

As we saw previously, learning occurs in two stages: gradual learning is performed across tasks, and fast learning (fast adaptation) is performed within tasks. This requires two learning rates, which introduces difficulty in choosing hyperparameters that would help achieve training stability. Finding two independent learning rates may lead to the application of a hyperparameter grid search computation, but this is time and resource intensive.

Special care needs to be taken in selecting the learning rate for the adaptation process, because it is learning over only a few examples. In that regard, some solutions or extensions to MAML have been developed to reduce the need for grid search or hyperparameter tuning. Alpha MAML, for example eliminates the need to tune both the learning rates by automatically updating them as needed.

The research paper How to train your MAML (Antoniou et al.)[18] describes an improved variant of the MAML framework (MAML++) that offers the flexibility of MAML along with many improvements, such as robust and stable training and automatic learning of inner loop hyperparameters. The paper proposes updating the query set loss (meta-loss) for every training step in the adaptation process, which can help get rid of the training instabilities. We also see such instabilities in our experiments, but since we average our results, this is not obvious. Another improvement is the "Per-Step Batch Normalization Weights and Biases" which compensates the absence of the "Batch Normalization Statistics Accumulation". The missing "Batch Normalization Statistics Accumulation" has a negative effect on the generalization performance. That was exactly one of the aspects we experienced in our experiments. In addition, they suggest various other steps, which greatly improve the computational efficiency both during inference and training and significantly improved generalization performance.

The MAML++ approach has been implemented in this thesis. The implemented algorithm in general works but could not be thoroughly tested, because we were facing *out of memory issues* while running the code on a GPU. We will keep the investigation on the MAML++ memory issues for later.

A current research paper that performs neural architecture search for gradient-based meta-learners suggests that approaches like MAML and its extensions tend to perform better with deeper neural architectures for few-shot classification tasks. Nevertheless, this should be explored with further experiments, since we saw during our tests that a more sophisticated model does not per se lead to a better performance.

[18] Antreas Antoniou, Harrison Edwards, Amos Storkey. How to train your MAML (2019).

8.2 Text Classification

In computer vision, meta-learning has emerged as a promising methodology for learning in a low-resource regime. These models learn to generalize in low-resource conditions by recreating such training tasks from the data available. Given this strong empirical performance, employing meta-learning in NLP is challenging. The challenge is the degree of transferability of an underlying representation learned across different classes. With language data, most tasks operate at the lexical level. Words that are highly informative for one task may not be relevant for other tasks. If we consider the corpus of HuffPost headlines, categorized into 41 classes, those highly salient for one class do not play a significant role in classifying others. But words informative for one class might also be informative for another class (e.g., “Trump” might be relevant in the classes Politics and Business).

Instead of directly considering words, one way would be to utilize their distributional signatures, characteristics of the underlying word distributions, which exhibit consistent behavior across classification tasks. Within the meta-learning framework, these signatures would enable us to transfer attention across tasks, which can consequently be used to weight the lexical representations of words. One broadly used example of such distributional signatures is tf-idf weighting, which explicitly specifies word importance in terms of its frequency in a document collection, and its skewness within a specific document. Distributional signatures could be utilized in the context of cross-class transfer. This subject is described in the paper Few-shot text classification with distributional signatures.[19]

Different data sets expose specific characteristics. These characteristics have a great impact on the performance of MAML. Therefore, we need to consider many impacting factors when applying MAML on NLP tasks, such as data quality, quantity, similarity among tasks, and the balance between language model and task-specific adaptation.

The one model fits all data approach does not hold true. The HuffPost data set contains headlines and descriptions which are shorter and less grammatical than formal sentences. The Swiss Economic Archive data on the other hand contains more contextual data. Our approach of applying a CNN on a word-level setting shows a surprisingly good performance, while our BERT baseline model, which also has been applied on the HuffPost data set, showed a decent but not excellent performance. Previous experiences with BERT models imply a much better performance. Therefore, an attention-based model like BERT would probably be better suited to the Swiss Economic Archive data, since it provides contextual information in formal sentences.

When it comes to the quantity of data. A 5 respectively, 10-shot setting shows quite substantial difference, as we saw in our experiments. As in a conventional deep learning setting, applying more data would yield a much better performance as well.

Let me get that clear, few-shot learning on 5 or 10 samples per class shows a surprisingly good performance but acquiring 20 or 50 datapoints per class might improve the performance even more and would not lead to a data-gathering burden.

8.3 Data pre-processing (data cleaning, feature engineering)

The previously mentioned approach in applying distributional signatures is one possible way, but when it comes to data quality, the pre-processing step is a vital aspect in improving the performance of a model. Pre-processing (data cleaning and feature engineering) is an art and needs a lot of domain knowledge. A thorough investigation on a given data set might even lead to better results by applying already available mechanisms.

I find there are so many open questions on what mechanisms should be applied or prevented. What pre-processing method does support or reduce the performance? Is it, for example, better to use stemming or lemmatization? Is it helpful to remove person names via Named Entity Recognition (NER), because those will almost never occur in embeddings? Is applying Part of Speech (POS) tagging supportive, or would it be favorable to apply a combination of them all? Should one remove single or double characters, expand contractions, and how should one handle terms in double quotes (e.g., book titles), etc.? The list goes on and on. The correct answer is hard to find and needs to be answered on a case-by-case basis. Our implementation allows the application of almost any combination, by building a pipeline which permits stacking individual pre-processing steps on top of each other. But that still does not answer some of the questions mentioned above. By analyzing the dataset, we found categories which are similar or relatively similar (e.g., “Arts & Culture” and “Culture & Arts” and “Arts”). It such cases it might be wise to aggregate them.

[19] Yujia Baoy_, Menghua Wuy_, Shiyu Changz, Regina Barzilay. Few-shot text classification with distributional signatures (2020)

8.4 Out-of-Vocabulary vector representation (OOV)

Word Embeddings encode the relationships between words through vector representations of the words. These word vectors are analogous to the meaning of the word. A limitation of word embeddings is that they are learned by the natural language model (FastText, GloVe and the like) and, therefore, words must have been seen in the training data before in order to have an embedding. Out-of-vocabulary (OOV) words are words that appear in text that need to be classified but do not receive a vector representation, since they do not exist in the embeddings.

There are many techniques to handle out-of-vocabulary words:

- Typically, a special out-of-vocabulary token <unk> is added to the language model. Often the first word in the document is treated as out-of-vocabulary. In that way, we can ensure the out-of-vocabulary words occur somewhere in the training data and get a positive probability. This is what we applied in our application as a quick fix.
- Another method often mentioned in the literature is sampling values either uniformly from a fixed interval centered at zero, or more often, from a zero-mean normal distribution with the standard deviation varying from 0.001 to 10.
- Another common trick, particularly when working with word embedding-based solutions, is to replace the word with a nearby word retrieved from a synonym dictionary.
- But there are more sophisticated ways, such as applying a model for predicting vector representations of out-of-vocabulary words in a sentence by considering the context of words surrounding the OOV word. There are many research papers available which discuss such possible approaches.

8.5 Ensemble technique

One of my favorite approaches is an ensemble of models. A model ensemble is a powerful technique for obtaining the best possible results on a given task. Deep learning neural networks are nonlinear methods. They offer increased flexibility and can scale in proportion to the amount of training data available. A downside of this flexibility is that they learn via a stochastic training algorithm, which means that they are sensitive to the specifics of the training data and may find a different set of weights each time they are trained, which in turn produces different predictions. Generally, this is referred to as neural networks having a high variance and it can be frustrating when trying to develop a final model for the use of making predictions. A successful approach to reducing the variance of neural network models is to train multiple models instead of a single model and to combine the predictions of these models. This is called ensemble learning and not only reduces the variance of predictions but can also result in predictions that are better than any single model. The idea behind ensembles is that different models learn different areas from the data. Combining these models leads usually to a better performance. More information is available in chapter 7.6.

An implementation has been prepared in this thesis based on the majority voting. Our implementation is based on the applied software architecture for meta-learning, which randomly samples training examples. With majority voting all models require the use of the same sample to finally apply a majority voting. So, final tests could not be performed, since our software architecture does not allow us to control the sequence of samples directly. We would need to refactor our productive code, which is not possible in the given time frame. A short cut to a solution might be the seed technique, but I am not too familiar with that yet.

There are certainly many more improvements which could be tackled to improve the performance.

9 Conclusion

Prior to starting this thesis, I quickly searched the web and thought there were many resources available. Having a closer look later showed that a lot of the resources I initially found were talking about the same thing repeatedly with, to me, minor variations. Most of the meta-learning research falls into the area of computer vision. I found some but not many valuable research papers tackling meta-learning in cooperation with text classification. Meta-learning for text classification still seems to be a road less travelled.

The whole thesis was a challenge. When I chose the topic “N-way/few-shot for text classification”, I was certain that this was going to be quite an endeavour, since I was not familiar with the subject, but my perception was that it is already widely used in the community. Time has taught me otherwise. I soon found out that this is rather a young area and research is only picking up. It turned out that the task has become a research subject rather than applied science. There were not many sources I could rely on. Almost every step I had to take meant creating my own implementation. Thanks to the Higher framework for solving the second order gradient subject.

One aspect that caused some sleepless nights was to understand the departure of the data set-up for conventional supervised learning towards few-shot learning and applying it to a Model-Agnostic Meta-Learning algorithm. There were many stumbling blocks to overcome. The “slough of despond” was wide. But as soon as one’s implementation works, everything seems so clear.

Meta-learning is appealing; its ability to learn from a few examples makes it particularly attractive. A gradient-based approach like MAML puts us in familiar territory: using pre-trained models and fine-tuning them. The MAML algorithm is simple, and its ability to perform a few gradient steps at inference time allows it to generalize quickly to unseen classes.

The approach is applicable to a variety of problems, including regression, classification, and reinforcement learning and can be combined with any model architecture, as long as the model is trained based on gradient descent.

Meta-learning and especially the MAML approach shows exactly what I intended to achieve. Meta-learning for text classification as a way to reach near state-of-the-art performance with just a handful of samples per class was a tremendous learning breakthrough, leading to a brighter future relieving us from the data-gathering burden by training models on systems with reduced computing resources.

While there are many areas for future research on meta-learning, here is a perspective on what could make it more adoptable in real-world scenarios, as well as which areas could benefit from future work.

Some of the successes of meta-learning in solving few-shot learning problems can be attributed to the data set-up for training and testing tasks. In general, learning is much easier if you train the model to do N-way, few-shot learning. The set-up in our experiments is arbitrary. We assumed that during meta-training and meta-testing time we would face a 5-way problem, with each class having either 5 or 10 samples.

But will a real-world inference scenario always match this expectation? Most likely not. Our implementation to apply a configuration setting, which allows control of the creation of any kind of task, is valuable for future experiments.

An area worth exploring is whether it is possible to train on heterogeneous datasets. For example, can we train a meta-learning model to classify a fork based on online news text and text passages from any other document presenting a short description (forks from different training domains or environments)?

How should we define the meta-training set and tasks in such a scenario? Should we consider classes from all the environments to define the meta-training, validation, and test datasets?

There is one interesting paper I came across related to image classification where the authors suggest using classes from one environment for the support set and classes from another environment for the query set. The authors find that meta-learning (MAML) can benefit, especially when there is a high degree of diversity in the data. Applying such profound aspects in text classification, by applying more sophisticated models, such as BERT and the like, could make it the next frontier.

In real-world scenarios, it is often likely that we will have lots of unlabelled data. Is it possible to employ meta-learning algorithms to leverage these unlabelled datasets, under such circumstances? The authors of one research paper propose a solution to this by augmenting a metric-based meta-learning approach to leverage unlabelled samples. In addition to the support set and the query set, the task includes an unlabelled set. This unlabelled set may or may not contain examples from the support set classes. The idea is to use labelled

samples from the support set and unlabelled samples within each task to generalize for a good performance on the corresponding query set.

Over the coming years, we will see additional approaches that will make meta-learning even more adoptable in real-world scenarios; whether it is using meta-learning successfully on heterogeneous data or leveraging unlabelled data, these approaches will make learning and generalizing with fewer labelled samples possible. As previously mentioned, we will hopefully continue to see research that simplifies both the training and inference process but also set ways to use meta-learning in a productive setting. These advances will allow machine learning practitioners to develop even more new ways of designing machine learning systems.

Meta-learning will be an exciting frontier for AI research and can be a big step forward in our quest to achieve Artificial General Intelligence, as computers would have the ability to not only make accurate classifications and estimates but would be able to improve their parameters (hyperparameters) to get better at multiple tasks in multiple problem contexts.

I would like to mention a few loose ends in this thesis, which I had to put aside given the time constraint. The main open point is our investigation on the Swiss Economic Archive dataset, but up front there are other aspects which need to be considered, since they have a direct dependency on text classification results, such as a thorough investigation (data analytics) on text pre-processing and out-of-word-vocabulary initialization. There should also a deep dive into model explainability since this also might give hints on how to improve text pre-processing. As a last point I would refactor my code and apply an enseble technique.

Finally, I would like to thank Prof. Dr. Mark Cieliebak and Jan Milan Deriu, PhD student, for their valuable support during this thesis.

10 Appendix

10.1 Data pre-processing

Data cleaning is one of the major parts in natural language processing. Many decisions need to be taken how to best clean text. The given implementation provides a whole set of text cleaning functions, which can be applied in a *pipeline*. The nlp_helper package contains 4 classes, each containing several static methods to apply in a pipeline. The classes are separated in *NLP_Helper, Remove, Expand, Replace*. Most the functions are removing “noise” (unwanted characters) as mentions, hash tags, html tags, punctuations, numbers etc. But there are further functions to expand contractions, detect named entities (person names, organizations, locations), replace “umlauts” (German words) or lemmatize text. A contraction is a word or phrase that has been shortened by dropping one or more letters. In writing, an apostrophe is used to indicate the place of the missing letters. Contractions are commonly used in speech (or written dialogue), informal forms of writing, and where space is limited, such as in advertising. It usually is wise to expand contractions in written text, since such expressions seldom occur in vocabularies, hence will lead to noise in a text.

Other functions not yet applied are spell checker or thesaurus functions. Such functions might even further improve the quality of text.

Implementation of data cleaning functions:

```
Package nlp.nlp_helper:
   class NLP_Helper()
   class Remove()
   class Expand()
   class Replace()
```

Example pipeline for English text data

```
def clean_text_data_en(text):

   pers, lo, org = NLP_Helper. \
                   find_entities(nlp(text))
   text = Remove.entities(text, pers)
   text = Expand.contractions_en(text)
   text = NLP_Helper.to_lower(text)
   text = Remove.stopwords_en(text)
   text = Remove.mentions(text)
   text = Remove.hashtags(text)
   text = Remove.html_tags(text)
   text = Remove.singlechar(text)
   text = Remove.ascii(text)
   text = Remove.numbers(text)
   text = Remove.punctuation(text)
   text = Remove.nline_and_car_ret(text)
   text = NLP_Helper.lemmatization(nlp(text))
  #text = NLP_Helper.stemming(text,
                       Stemmer ='porter',
                       lang = 'en')
   text = Remove.two_chars(text)
   text = Remove.white_spaces(text)
   return text
```

Example pipeline for German text data

```
def clean_text_data_de(text):

   pers, lo, org = NLP_Helper. \
                   find_entities(nlp(text))
   text = Remove.entities(text, pers)
   text = Expand.Contraction_DE.expand(text)
   text = NLP_Helper.to_lower(text)
   text = Remove.stopwords_de(text)
  #text = Replace.umlaut(text)
   text = Remove.mentions(text)
   text = Remove.hashtags(text)
   text = Remove.html_tags(text)
   text = Remove.singlechar(text)
   text = Remove.ascii(text)
   text = Remove.numbers(text)
   text = Remove.punctuation(text)
   text = Remove.nline_and_car_ret(text)
   text = NLP_Helper.lemmatization(nlp(text))
  #text = NLP_Helper.stemming(text,
                       stemmer = 'snowball',
                       language = 'de')
   text = Remove.two_chars(text)
   text = Remove.white_spaces(text)
   return text
```

10.2 Explanatory data analysis (EDA)

10.2.1 HuffPost data analysis

In this section a few metrics are listed. The metrics have been drawn on the pre-cleaned data. There are more metrics available but have not been applied here, because of their exhaustive character.

HuffPost data summary	
Item	figures
Number of categories	41
Number of samples	135'835
Min number of words	10
Max number of words	136

Table 24: HuffPost data metrics

The categories (classes) available in the data set are listed in Table 25 below. By analysing the list of categories one can see that there are a few relatively similar or even similar categories. Being able to clearly predict a class assignment will be relatively hard since similar classes usually use similar words to describe the respective news. In such a case it is feasible to aggregate similar classes into one class.

Similar categories for example are:
- “Arts”, “Arts & Culture”, “Culture & Arts”
- “World Post”, “World News”, “The World post”
- “College”, “Education”
- “Parenting”, “Parents”

Categories			
Arts	Environment	Parenting	The Worldpost
Arts & Culture	Fifty	Parents	Travel
Black Voices	Food & Drink	Politics	Weddings
Business	Good News	Queer Voices	Weird News
College	Green	Religion	Wellness
Comedy	Healthy Living	Science	Women
Crime	Home & Living	Sports	World News
Culture & Arts	Impact	Style	WorldPost
Divorce	Latino Voices	Style & Beauty	
Education	Media	Taste	
Entertainment	Money	Tech	

Table 25: HuffPost – categories

Imbalanced data
The chart below (see Figure 22) shows the class distribution and the respective number of samples available for a respective class (category). Imbalanced data is usually a problem in text classification, but since our approach only uses few samples per class, we are not bound to balance the classes.

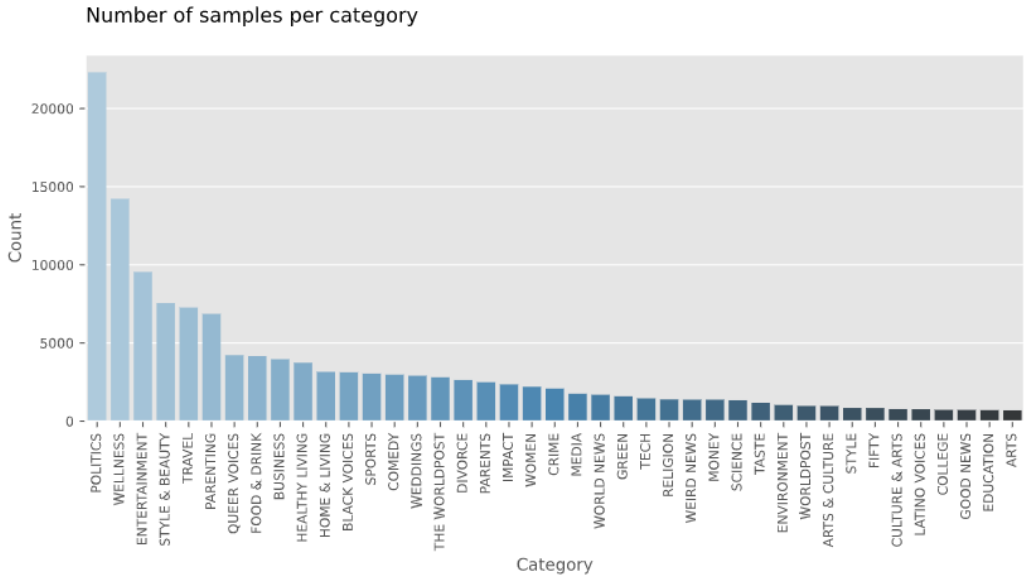

Figure 22: HuffPost - Class distribution and number of samples per class

Average number of words
In the data cleaning phase, we removed the stopwords, punctuation (noise) etc., but also applied lemmatization which unifies words to its base form. These few steps reduce the number of words per sample substantially. The average number of words per class is depicted below (see Figure 23)

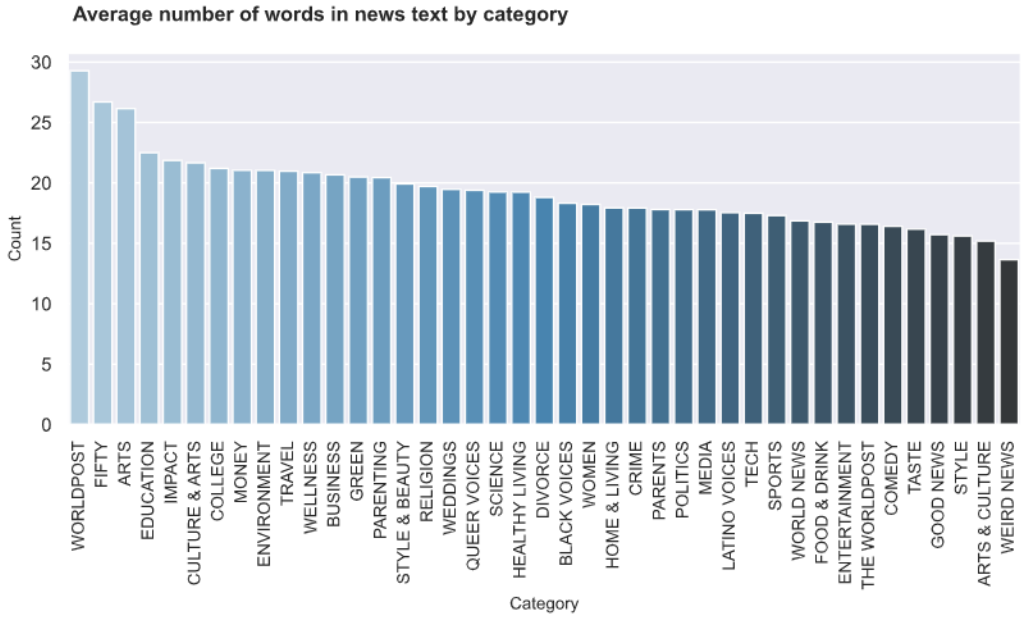

Figure 23: HuffPost - Average number of words per class

Most used words
The graph below (Figure 24) indicates the 30 most used words in all samples. One thing that catches one's eye is the fact that almost none of the words might lead to clear class assignment. This might lead to the fact that different feature engineering steps need to be applied, to retrieve the essence of a news text. One possible way would be to apply distributional signatures[20].

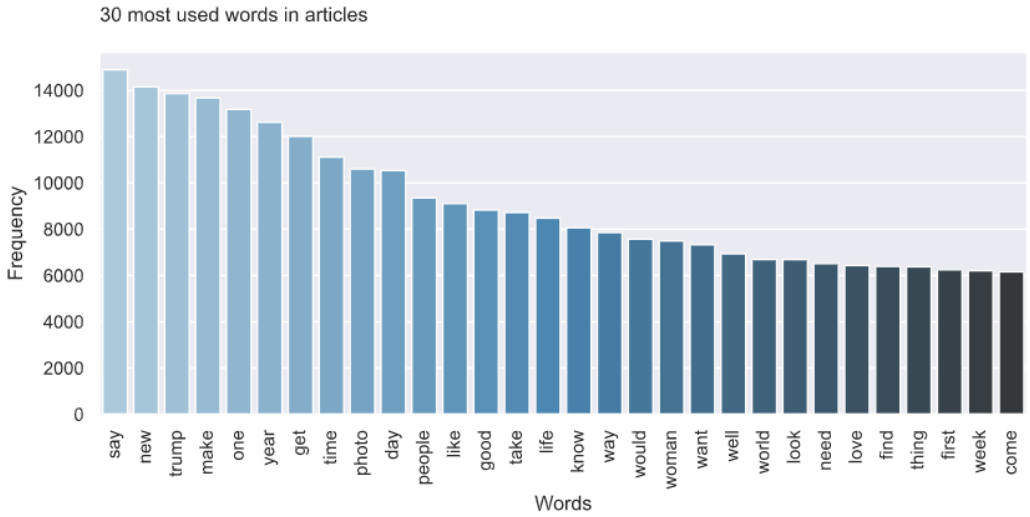

Figure 24: HuffPost - 30 most used words

10.2.2 Swiss Economic Archive (SWA)

In this section a few metrics are listed. The metrics have been drawn on the pre-cleaned data. There are more metrics available but have not been applied here, because of their exhaustive character.
The original Swiss Economic Archive (SWA) dataset is a multi-class/ multi-label data set. The data set contains 485 classes, where samples might be assigned to more than one class. There are samples with up to 9 assigned classes. For our analysis we only consider single class assignments. Therefore, the Swiss Economic Archive (SWA) data comes now in two flavours. Dataset 1 contains news text samples exclusively assigned to one class (146 classes), where dataset 2 also considers samples, which have 2 class assignments, but only one will be used. Dataset 2 contains samples which are split into 248 classes. Initially we only consider dataset 1.

Swiss Economic Archive data metrics	
Item	Figure
Number of categories	146
Number of samples	12'244
Min number of words	21
Max number of words	2040

Table 26: Swiss Economic Archive data metrics

The following list is an excerpt of the 146 classes.

Categories			
Volkswirtschaft. USA	Strassenfahrzeug	Weltwirtschaft	Arbeitsmarkt. Schweiz
Krankenversicherung	Mobilkommunikation	Aussenhandel	Ärzte
Wirtschaftspolitik	Tourismus. Schweiz	Öffentl. Beschaffung	Bank
Aktiengesellschaft	Mineralö	Familienplanung	Steuerstraftat
AHV	'Politisches System	Studienfinanzierung	Nukleare Entsorgung
Militär	Zahlungsverkehr	Glücksspiel	Wasserbau
Lehrkräfte	Verlagsgewerbe	Mehrwertsteuer	Energiepolitik
Finanzausgleich	Bio- und Gentechnik'	Nahrungsmittelgewerbe	Erwerbsunfähigkeit
Europapolitik	Justiz	Soziale Sicherung	Entwicklungshilfe
Klima	Naturkatastrophe	Volkswirtschaft. EU	Wasserkraft
Naturschutz	Flüchtlinge	Sexgewerbe	Volkswirtschaft CH

Table 27: Swiss Economic Archive – excerpt of categories

[20] Yujia Baoy_, Menghua Wuy_, Shiyu Changz, Regina Barzilay. Few-shot text classification with distributional signatures (2020)

Imbalanced data
The chart below (Figure 25) shows the class distribution and the respective number of samples available in a class (category). Imbalanced data is usually a problem in text classification, but since our approach only uses few samples per class, we are not bound to balance the classes.

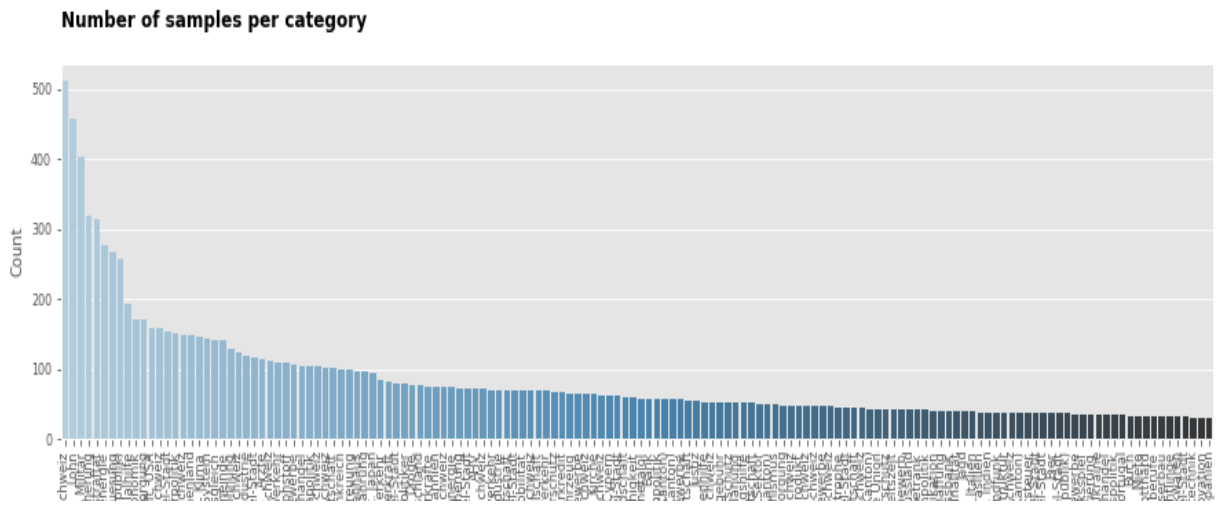

Figure 25: Swiss Economical Archive - Class distribution and number of samples per class

Average number of words
In the data cleaning phase, we removed the stopwords, punctuation (noise) etc., but also applied lemmatization which unifies words to its base form. These few steps reduce the number of words per sample substantially. The average number of words per class is depicted in Figure 26.

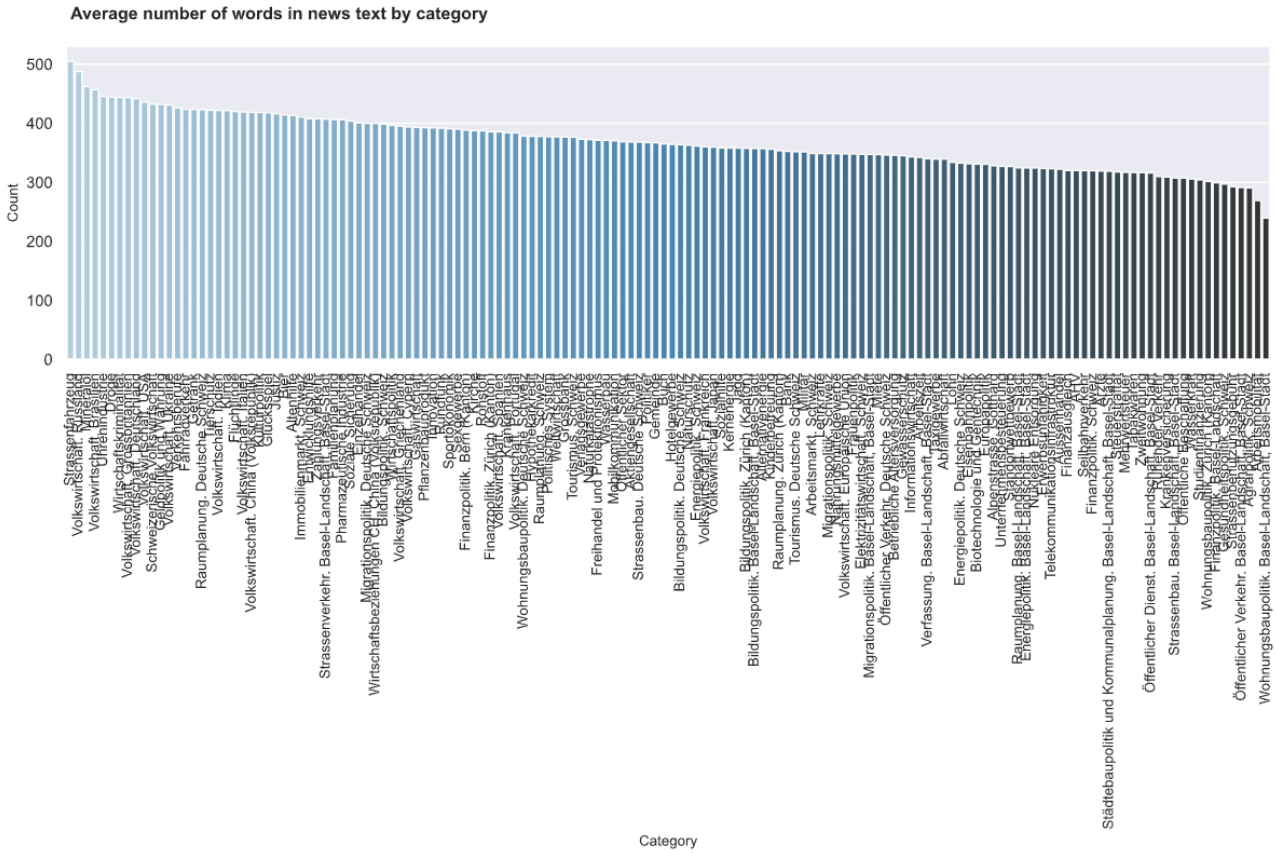

Figure 26: Swiss Economic Archive - Average number of words per class

Most used words
The graph in Figure 24 indicates the 30 most used words in all samples. One thing that catches one's eye is the fact that almost none of the words might lead to clear class assignment. This might lead to the fact, that other feature engineering steps need to be applied, to retrieve the essence of a news text. One possible way would be to apply distributional signatures[21]. Some words should also be reconsidered and potentially be treated as noise.

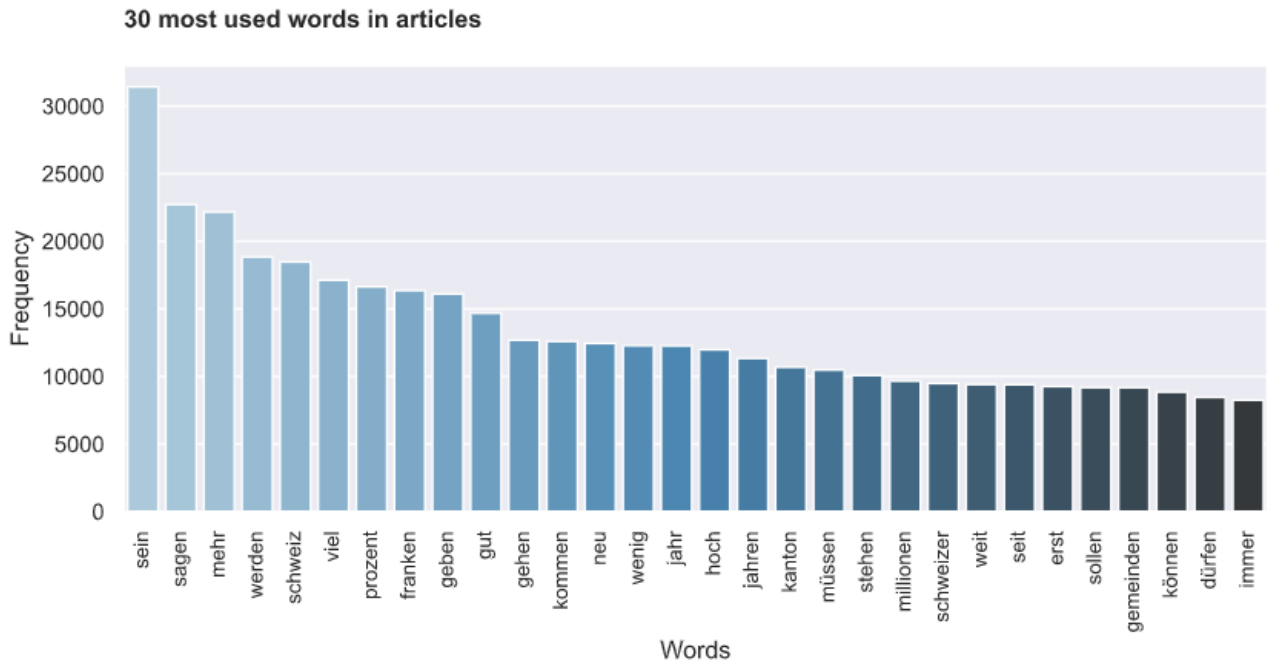

Figure 27: Swiss Economic Archive - 30 most used words

10.2.3 Vocabulary

The vocabulary is a aggregation area for processed text before it is transformed into some representation for the impending task, be it classification, or language modeling, or something else. In NLP, a text corpus (raw text), in our case news text data, needs to be pre-processed and cleaned. These steps are vital for a good vocabulary. Text cleaning is described in chapter 10.1. The next step after pre-processing and cleaning is tokenization, where we split a text into tokens, which is a of segmentation (usually words). Usual vocabularies may contain 10s of thousands of words. In our case we removed the noise (unwanted words as stop words) clean contractions and applied a lemmatization etc. This reduces the number of relevant words substantially. For our purpose we created a vocabulary of 20'000 words. It could also be 30'000 words, but we intend to keep the balance between resource consumption and still be able to run our classification tasks.

Out-of-vocabulary words (OOV)
Word Embeddings encode the relationships between words through vector representations of the words. These word vectors are analogous to the meaning of a word. A limitation of word embeddings is that they are learned by the natural language model (Word2Vec, GloVe and the like) and therefore words must have been seen in the training data before, to have an embedding.
The current solution in this thesis uses the pre-trained embeddings GloVe for English (glove.6B.300d) and FastText for German (ft.de.300d.vec) texts. OOV's in both cases are initialized with the special character <unk>, which corresponds to a 0 value. The prediction rate could certainly improve if handling out-of-vocabulary words would be applied. OOV handling is described in the paper "Handling Out-of-Vocabulary Words in Natural Language Processing based on Context"[22].

Metrics: Out-of-vocabulary words – vocabulary size		
Dataset	out-of-vocabulary words	vocabulary size
HuffPost	416	20'002 (including <pad> and <unk>)
Swiss Economic Archive	138	20'002 (including <pad> and <unk>)

Table 28: Metrics: Out-of-vocabulary words – vocabulary size

[21] Yujia Baoy_, Menghua Wuy_, Shiyu Changz, Regina Barzilay. Few-shot text classification with distributional signatures (2020)
[22] Shabeel Kandi, Handling Out-of-Vocabulary Words in Natural Language Processing based on Context (2018)

10.3 Text data augmentation

Data Augmentation is the process that allows to increase the size of training data without collecting the data. Image data augmentation steps such as flipping, cropping, rotation, blurring, zooming, etc. helped tremendously in computer vision. Also, it is relatively easy to create augmented images. Text augmentation in contrast is not as easy due to the complexities inherent in the language. (For example, we cannot replace every word by its synonym, and even if we replace it, the meaning of the sentence might change completely). Text data augmentation is not particularly relevant in this thesis, since we go the route of using as little data as possible and still be able to predict a class given some text. This subject is mentioned here, because it has been used for our baseline models, which rely on a balanced dataset. The following types of text augmentation could be applied:

- Synonym Replacement
- Random Insertion
- Swap and Deletion
- Shuffle Sentences Transform
- Character-, Word- and Sentence level augmentation

In our implementation we applied a swap mechanism. 20% of the words in a sentence are randomly chosen and replaced (swapped) with similar words from pretrained word embeddings (GloVe and FastText).

Excerpt of data augmentation methods:

```
Package data.data_augmentation:
   class Data_Augmentation()
         augment_<dataset_name>(df, samples = 300, pr = 0.2)
         balance_<dataset_name>_classes(df)
         augmentation_check(text)
```

10.4 Error analysis / Error attribution

Building a machine learning pipeline, especially when the pipeline is complex is not easy. Error analysis and error attribution is one of the major aspects while analysing a complex machine learning pipeline, where one needs to improve, the system's performance. Which part of the pipeline should you work on improving? By attributing errors to specific parts of the pipeline, one can decide how to prioritize work.

Our pipeline consists mainly of four pipeline components: data cleaning, vocabulary, feature selection and model architecture. It is possible to spend ages working on improving either of these four pipeline components. How do you decide which component(s) to focus on?

One way to approach this, is by carrying out error **analysis by parts**. One can try to attribute each mistake the algorithm makes to one (or sometimes more) of the four parts of the pipeline.

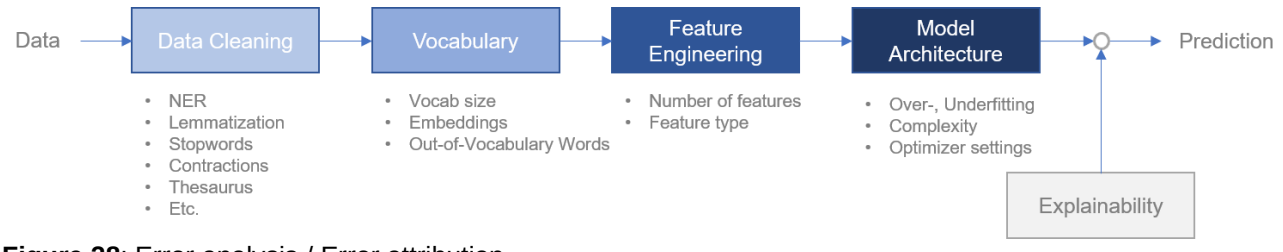

Figure 28: Error analysis / Error attribution

10.5 Model-Agnostic explanations for Machine Learning classifiers

Model-Agnostic Explanations for Machine Learning classifiers also known as *Explainable AI* refers to methods and techniques in the application of artificial intelligence technology (AI) such that the results of the solution can be understood by humans. It contrasts with the concept of the "black box" in machine learning where even its designers cannot explain why an AI arrived at a specific decision.
In the following we will apply the *lime package*, which provides some convenience methods to **shed light** on our *black box model* and find some explanations.

Implementation of explainability functions:

```
Package analytics.explainability:
   class Explainability()
```

Example:
Words in training example 34:
['pray', 'jewish', 'new', 'year', 'big', 'holiday', 'obscene', 'amount', 'food', 'long', 'hour', 'synagogue', 'year', 'something', 'different', 'instead', 'use', 'synagogue', 'prayer', 'book', 'designate', 'take', 'prayer', 'book']
Predicted class: RELIGION
True class: RELIGION

The following plots show, how Lime attributes weights to different classes given the respective words.

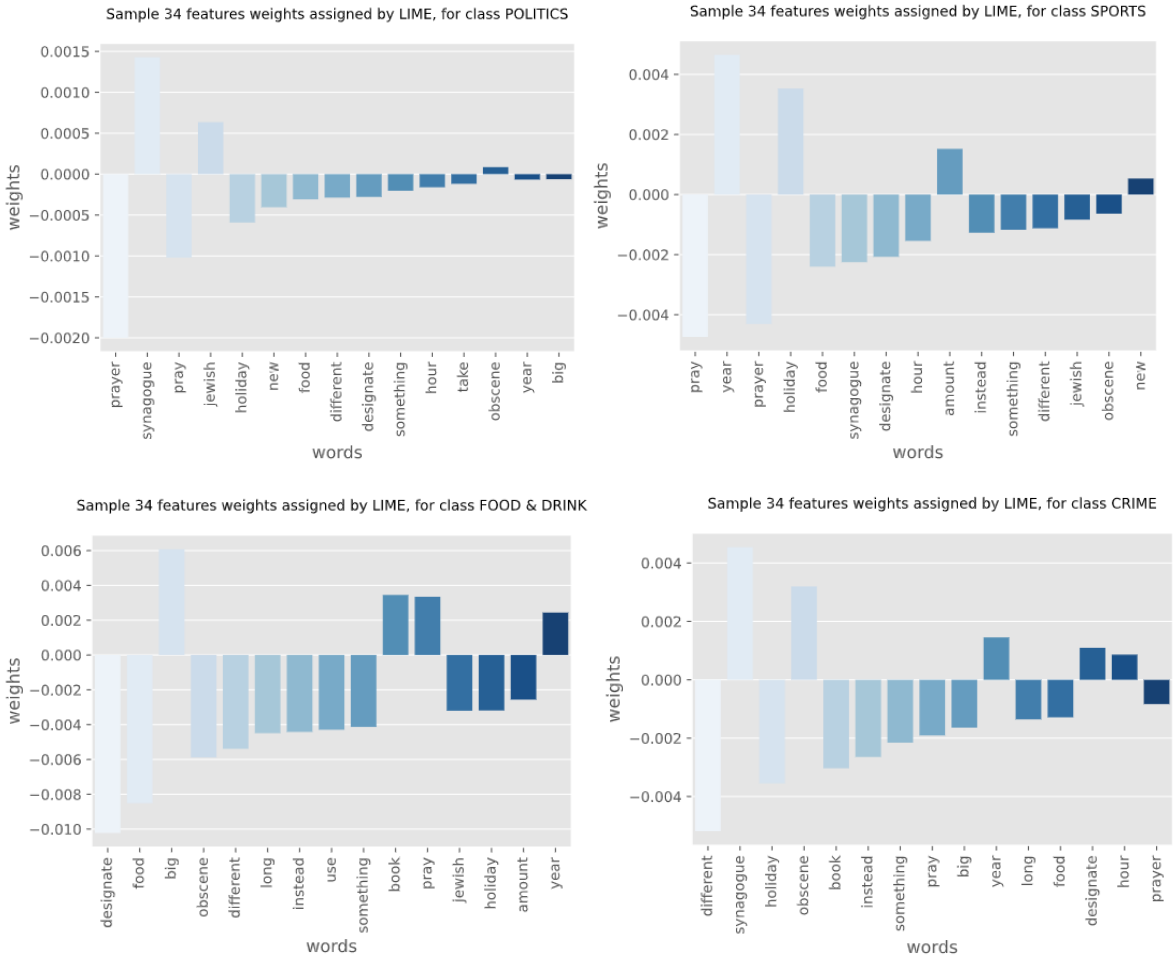

Figure 29: Lime weight attribution

10.6 Convolutional Neural Network layers

10.6.1 Embedding Layer

Embedding layers are mainly used in natural language processing-related tasks, such as language modelling. It is usually the first layer in a neural network when working with textual data. Word embedding is an alternate to one-hot encoding along with dimensionality reduction. The embedded words are represented by dense vectors where a vector represents the projection of the word into a continuous vector space. The position of a word within the vector space is learned from text and is based on the words that surround the word when it is used. Embedding layers adapt pre-trained embeddings like GloVe or FastText.

10.6.2 Convolutional Layer

The Conv layer is the core building block of a convolutional network that does most of the computational heavy lifting. Convolutional layers apply a convolution operation to the input, passing the result to the next layer. A convolution converts all the input values in its receptive field into a single value by applying filters. If we apply a convolution on text data, this will decrease the size as well as bringing all the information in the field together into a single field.

10.6.3 Pooling Layer

It is common to periodically insert a pooling layer in between successive Conv layers in a ConvNet architecture. Its function is to progressively reduce the spatial size of the representation to reduce the number of parameters and computation in the network, and hence to also control overfitting. The pooling layer operates independently on every depth slice of the input and resizes it spatially, using the MAX operation.

10.6.4 Fully Connected Layer

Neurons in a fully connected layer have full connections to all activations in the previous layer. Their activations can hence be computed with a matrix multiplication followed by a bias offset.

10.6.5 Rectified Linear Unit (ReLU)

The Rectified Linear Unit (ReLU) activation function is responsible for transforming the summed weighted input from the node into the activation of the node or output for that input. The ReLU is a piecewise linear function that will output the input directly if it is positive, otherwise, it will output zero. It has become the default activation function for many types of neural networks because a model that uses it is easier to train and often achieves better performance.

10.6.6 Softmax

A softmax activation function is a mathematical function that converts a vector of numbers into a vector of probabilities, where the probabilities of each value are proportional to the relative scale of each value in the vector. Softmax is basically a probabilistic version of the argmax function.

10.6.7 Batch Normalization

Batch Normalization (Ioffe & Szegedy, 2015)[23] is a method to stabilize and speed up learning of deep neural networks by reducing internal covariate shift within the learner's hidden layers. This reduction is achieved by normalizing each layer's pre-activation, by standardizing the values. Standardization is a common requirement for many machine learning classifiers since they might behave badly if the individual features do not more or less look like standard normally distributed data such as Gaussian with zero mean and scaling the variance. During training, the mean and standard deviation are estimated using the current batch being trained on, whereas during evaluation a running average of both statistics calculated on the training set is used. We need to be careful with batch normalization for the learner network in the meta-learning setting, because we do not want to collect mean and standard deviation statistics during meta-testing in a way that allows information leaking between different tasks to be considered. One easy way to prevent this issue is to not collect statistics at all during the meta-testing phase, but just use our running averages from meta-training.

10.6.8 Dropout Layer

The dropout layer in neural networks takes a regularization function and is, therefore, described in the next section.

[23] Sergey Ioffe, Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift (2015).

10.7 Meta learning algorithm

I am pretty certain that plotting code is not the best idea. But be it that way

```
def maml_experiment(self, nt_loader, model_name, epoch, mode = 'train'):

    self._model.to(device = self._device)
    self._model.train()

    # {} placeholder variable initializations
    batches = nt_loader.load_episodes(mode)

    for batch_idx in range(len(batches)):
        tasks = batches[batch_idx]

        num_tasks = len(tasks)
        for task_idx in range(num_tasks):
            task = tasks[task_idx]
            dataloader = torch.utils.data.DataLoader(task, **self._params)

            for task in dataloader:
                # Unpack the inputs from dataloader
                x_support_set, y_support_set, x_query_set, y_query_set = task

            inner_step_size = 5

            # Initialize the inner optimizer to adapt the parameters to the support set.
            self._inner_optimizer = optim.SGD(self._model.parameters(),
                                              lr = self._config['sgd_learning_rate'],
                                              momentum = self._config['sgd_momentum'])
            self._meta_optimizer.zero_grad()

            with higher.innerloop_ctx(self._model,
                                      self._inner_optimizer,
                                      copy_initial_weights = False) as (fmodel, diffopt):

                # Optimize the likelihood of the support set by taking gradient steps w.r.t.
                # the model's parameters. This adapts the model's meta-parameters to the task.
                # higher is able to automatically keep copies of the network's parameters as they
                # are being updated.

                for _ in range(inner_step_size):
                    try:
                        support_logit = fmodel(x_support_set)
                        support_loss  = F.cross_entropy(support_logit, y_support_set)
                        diffopt.step(support_loss)
                    except Exception as e:
                        print('Exception: {0}'.format(e))

                # The final set of adapted parameters will induce some final loss and accuracy
                # on the query set. These will be used to update the model's meta-parameters.

                query_logit = fmodel(x_query_set)
                query_loss  = F.cross_entropy(query_logit, y_query_set) # per_task_loss
                query_losses.append(query_loss.detach().item())         # per_task_losses

                with torch.no_grad():
                    query_accuracy, predictions = self.get_accuracy(query_logit, y_query_set)
                    query_accuracies.extend(query_accuracy)

                metric = self.per_task_metric(task_cnt, query_loss, query_accuracy, mode)
                task_metrics.append(metric)

                # Update the model's meta-parameters to optimize the query losses across all
                # of the tasks sampled in this batch. This unrolls through the gradient steps.

                query_loss.backward()

        self._meta_optimizer.step()

        batch_metrics.append(self.per_batch_metric(**parameters))

    # {} save best model to disk

    return batch_metrics_df, task_metrics_df
```

10.8 Software System

In this section we just give a brief overview of our software system structure:

10.8.1 Application structure

All source code is mainly located in the lib and the respective sub directories. Only packages are listed here.

```
n_way_few_shot
  - configs
    - <config_files>.json
  - data
    - bbc               (pre-processed data)
    - eu_news           (pre-processed data)
    - huffpost          (pre-processed data)
    - meta              (metrics data acquired during training and testing)
    - swa               (pre-processed data)
    - embeddings        (pre-trained embeddings)
  - doc
  - lib
    - mltools
      - analytics       (data analytics classes)
      - data            (data management classes)
      - models          (baseline models)
      - nlp             (nlp related classes)
      - utils           (general purpose classes, methods)
    - meta
      - analytics       (data analytics classes)
      - datasets        (data management classes)
      - meta_higher     (maml, maml++ classes)
      - utils           (general purpose classes, methods)
      - meta_l2l        (learn to learn playground)
      - models          (meta models)
      - transforms      (transformations)
  - model
      - features        (pickle file storage)
      - meta            (pre-trained meta models)
      - weights         (pre-trained baseline models)
  - notebooks           ()
  - resources           (stored images)
  - test                (test classes)
 requirements.txt       (used libraries)
 scripts and notebooks  (main entry point)
```

Python Version:	3.8.5
Libraries:	all used libraries are listed in the requirements.txt file
Development Environment:	Visual Studio Code

Source code: MAS_Master_Thesis on github

10.8.2 Meta-learning core system

Class diagram of the main Meta-Learning core:

The main meta learning algorithm consists of 3 items

- MT_maml (main script)
- Class Meta_System()
- Class Meta_Learner()

MT_maml script primary work:

- Data loading
- Loading embeddings
- Vocabulary creation
- Model initialization

Class *Meta_System()* orchestrates all involved classes:

- Meta system holds an instances of the *Meta_Learner()* and *NewsTextDataset()*
- creates $D_{meta-train}$, $D_{meta-valid}$, $D_{meta-test}$ via a *data_splitter()*
  the data_splitter() is the work horse. It does the book_keeping and class remapping (if required) and creates the tasks respectively batches.
- aggregates the individual data loaders (train, valid and test)
- create meta-train, meta-valid and meta-test set

Class *Meta_Learner()*

- Holds the optimizer inner and outer loop (SGD and Adam)
- Performs the meta-learning and optimization (learning and adaptation)
- Prepares metrics (task and batch accuracies and loss)

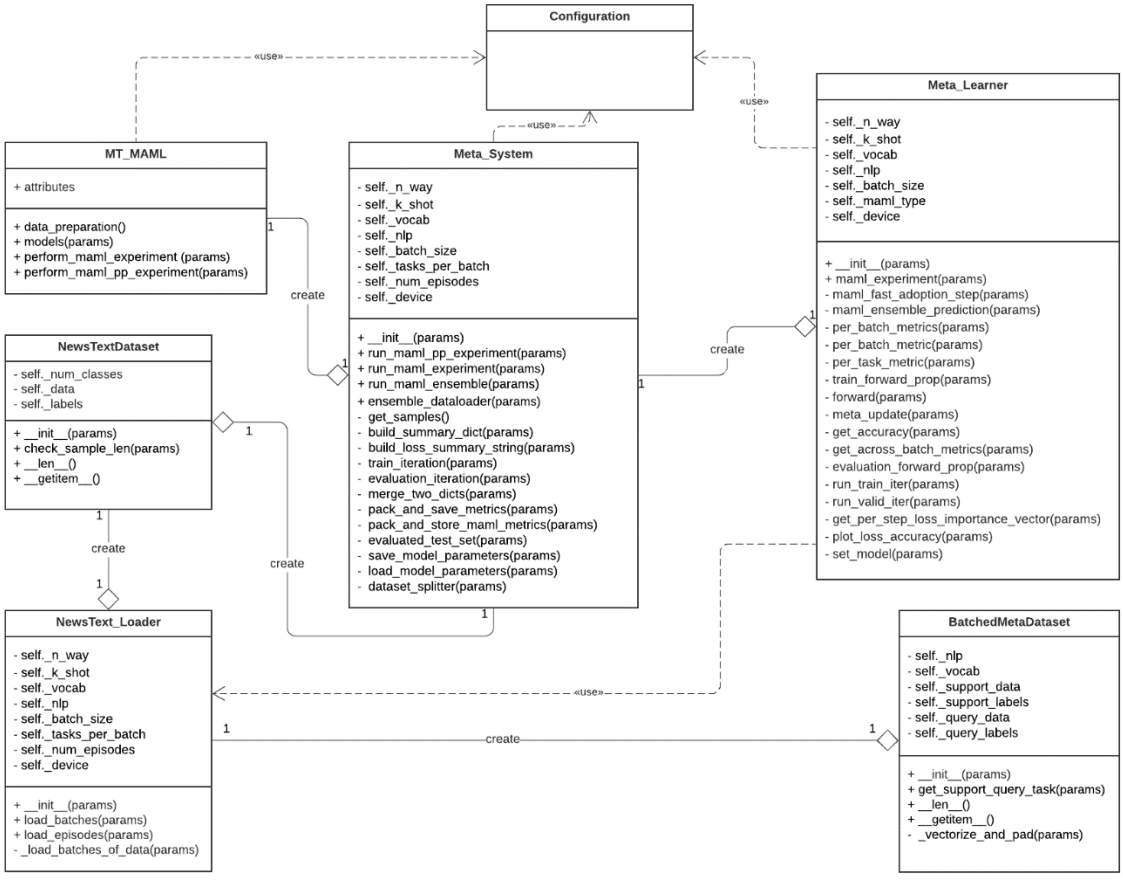

Figure 30: Class diagram learn-to-learn

11 References

[1] Oriol Vinyals, Charles Blundell, Tim Lillicrap, Daan Wierstra. Matching networks for one shot learning. In Advances in Neural Information Processing Systems, (2016). arXiv:1606.04080

[2] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems, (2012).

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding (2018). arXiv:1810.04805

[4] Chelsea Finn, Pieter Abbeel, Sergey Levine. Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks (2017). arXiv 1703.03400

[5] Sachin Ravi and Hugo Larochelle. Optimization as a model for few-shot learning (2017)

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification (2015). arXiv 1502.01852.

[7] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting (2014)

[8] Leo Breimann. Random Forests (2001). Stat. Berkeley

[9] Hothorn T, Hornik K., Zeileis A. Unbiased Recursive Partitioning: A Conditional Inference Framework (2006)

[10] Carolin Strobl, Anne-Laure Boulesteix, Achim Zeileis and Torsten Hothorn. Bias in random forest variable importance measures: Illustrations, sources and a solution (2007)

[11] Tianqi Chen and Carlos Guestrin. XGBoost: A Scalable Tree Boosting System (2016). arXiv:1603.02754

[12] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin. Attention Is All You Need (2018). arXiv 1706.03762

[13] Edward Grefenstette, Brandon Amos, Denis Yarats, Phu Mon Htut, Artem Molchanov, Franziska Meier, Douwe Kiela, Kyunghyun Cho, Chintala Soumith. Generalized Inner Loop Meta-Learning (2019). arXiv:1910.01727

[14] Tristan Deleu, Tobias Würfl, Mandana Samiei, Joseph Paul Cohen, Yoshua Bengio. Torchmeta: A Meta-Learning library for PyTorch (2019). arXiv 1909.06576

[15] Armand Joulin, Edouard Grave, Piotr Bojanowski, Matthijs Douze, Hérve Jégou, Tomas Mikolov. FastText.zip: Compressing text classification models (2016). arXiv 1612.03651

[17] Antreas Antoniou, Harrison Edwards, Amos Storkey. How to train your MAML (2019). arXiv 1810.09502

[19] Yujia Baoy_, Menghua Wuy_, Shiyu Changz, Regina Barzilay. Few-shot text classification with distributional signatures (2020). arXiv:1908.06039

[22] Shabeel Kandi, Handling Out-of-Vocabulary Words in Natural Language Processing based on Context (2018)

[23] Sergey Ioffe, Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift (2015). arXiv 1502.03167

[24] Koch G.; Zemel R. & Salakhutdinov R. Siamese Neural Networks for One-shot Image Recognition (2015)

[25] Edouard Grave, Piotr Bojanowski, Prakhar Gupta, Armand Joulin, Tomas Mikolov. Learning Word Vectors for 157 Languages (2018)

[26] Zequn Liu, Ruiyi Zhang, Yiping Song, Ming Zhang. When does MAML work the Best? An Empirical Study on Model-Agnostic Meta-Learning in NLP Applications (2020)

[27] Trapit Bansaly and Rishikesh Jhaz, Andrew McCallum. Learning to Few-Shot Learn Across Diverse Natural Language Classification Tasks (2020)

[28] Niels van der Heijden, Helen Yannakoudakis, Pushkar Mishra, Ekaterina Shutova. Multilingual and cross-lingual document classification: A meta-learning approach (2021)

[29] Marcin Andrychowicz, Misha Denil, Sergio Gómez Colmenarejo, Matthew W. Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, Nando de Freitas. Learning to learn by gradient descent (2016). arXiv 1606.04474.

[30] Benyamin Ghojogh, Mark Crowley. The Theory Behind Overfitting, Cross Validation, Regularization, Bagging, and Boosting: Tutorial (2019). arXiv 1905.12787

[31] Aniruddh Raghu and Maithra Raghu and Samy Bengio and Oriol Vinyals. Rapid Learning or Feature Reuse? Towards Understanding the Effectiveness of MAML (2020). arXiv 1909.09157

12 Glossary

12.1 Abbreviations

Acc	Accuracy
CNN / ConvNet	Convolutional Neural Network
DL	Deep Learning
EDA	Explanatory Data Analysis
GPU	Graphics Processing Unit
IDF	Inverse Document Frequency
LSTM	Long Short Term Memory
MAML	Model Agnostic Meta Learning
ML	Machine Learning
NLP	Natural Language Processing
OOV	Out-of-vocabulary words
RL	Reinforcement Learning
RNN	Recurrent Neural Network
SNN	Siamese Neural Network
SWA	Swiss Economic Archive (Schweizerisches Wirtschafts Archiv)
TF	Term Frequency

Table 29: Abbreviations

12.2 Terms

f1-score	F1-score is a measure of a model's accuracy on a data set. It is used to evaluate classification systems. The F1-score is a way of combining the precision and recall of the model, and it is defined as the harmonic mean of the model's precision and recall. The F1-score is commonly used for evaluating in in natural language processing. $$f1\_score = 2 * \frac{precision * recall}{precision + recall}$$
precision	Precision is the fraction of true positive examples among the examples that the model classified as positive. $$\text{precision} = \frac{\text{True positives}}{\text{True positives} + \text{False positives}}$$
recall	Recall, also known as sensitivity, is the fraction of examples classified as positive, among the total number of positive examples. $$recall = \frac{\text{True positives}}{\text{True positives} + \text{False n}egatives}$$
feature engineering	Feature engineering is the process of using domain knowledge to extract meaningful features from a data set. The features result in Machine Learning models with higher accuracy.
imbalanced data	Imbalanced data sets are those where there is a severe skew in the class distribution, such as 1:100 or 1:1000 examples in the minority class to the majority class.
undersampling / oversampling	Addressing the problem of class imbalance is to randomly resample the training data set. The two main approaches to randomly resampling an imbalanced data set are to delete examples from the majority class, called undersampling, and to duplicate examples from the minority class, called oversampling.
support set / query set	Consider a *k-shot* / *n-way* classification task: The support and query set contain *k* labelled examples for each of *n* classes for learning and prediction.

Table 30: Terms

13 Index

Wahrheitserklärung

Mit der Abgabe dieser Abschlussarbeit versichert der/die Studierende, dass er/sie die Arbeit selbständig und ohne fremde Hilfe verfasst hat (Bei Teamarbeiten gelten die Leistungen der übrigen Teammitglieder nicht als fremde Hilfe).

Der/die unterzeichnende Studierende erklärt, dass alle zitierten Quellen (auch Internetseiten) im Text oder Anhang korrekt nachgewiesen sind, d.h. dass die Abschlussarbeit keine Plagiate enthält, also keine Teile, die teilweise oder vollständig aus einem fremden Text oder einer fremden Arbeit unter Vorgabe der eigenen Urheberschaft bzw. ohne Quellenangabe übernommen worden sind.

Name des/der Studierenden (Druckbuchstaben)

Bruno Hunkeler
……..

Winterthur, 30. Juni 2021

(Ort, Datum) (Unterschrift der Studierenden / des Studierenden)