Text generation for small data regimes

Husam Quteineh

Supervisors: Dr. Spyros Samothrakis
Dr. Richard Sutcliffe

Department of Computer Science and Electronic Engineering
University of Essex

This dissertation is submitted for the degree of
Doctor of Philosophy

Date of submission for examination: March 2022

Acknowledgements

I would like to express my gratitude to Dr. Spyros Samothrakis and Dr. Richard Sutcliffe for their support and motivation throughout the different stages of my PhD journey. My special thanks goes to Dr. Spyros, first, for his support and backing that made it possible for me to pursue this PhD, and second, for his guidance throughout the research work. Our weekly meetings and conversations were vital in inspiring me to think outside the box and from multiple perspectives to objectively tackle any setbacks and to extend proposed research ideas. I wouldn't have made it this far in my research journey without his instructive feedback and wise supervision. I would also like to extend my gratitude and indebtedness to Dr. Richard for his continuous instructive feedback and recommendations throughout my doctoral studies. Our meetings early on in this journey gave me confidence in my ability and inspired me to focus on my research objectives. My gratitude also goes to the University of Essex for funding the opportunity to undertake my studies at the School of Computer Science and Electronic Engineering, University of Essex. Furthermore, I would like to thank the Business and Local Government Data Research (BLG) Centre (funded by the Economic and Social Research Council (ESRC)), for their support and allowing me to undertake this PhD in parallel to my work. Last but not least, I dedicate my heartfelt thanks to my loving family back home in Palestine and Lebanon, and my gratitude for their endless support and faith in my potential.

Abstract

In Natural Language Processing (NLP), applications trained on downstream tasks for text classification usually require enormous amounts of data to perform well. Neural Network (NN) models are among the applications that can always be trained to produce better results. Yet, a huge factor in improving results is the ability to scale over large datasets. Given that Deep NNs are known to be data hungry, having more training samples can always be beneficial. For a classification model to perform well, it could require thousands or even millions of textual training examples.

Transfer learning enables us to leverage knowledge gained from general data collections to perform well on target tasks. In NLP, training language models on large data collections has been shown to achieve great results when tuned to different task-specific datasets Wang et al. (2019, 2018a). However, even with transfer learning, adequate training data remains a condition for training machine learning models. Nonetheless, we show that small textual datasets can be augmented to a degree that is enough to achieve improved classification performance.

In this thesis, we make multiple contributions to data augmentation. Firstly, we transform the data generation task into an optimization problem which maximizes the usefulness of the generated output, using Monte Carlo Tree Search (MCTS) as the optimization strategy and incorporating entropy as one of the optimization criteria. Secondly, we propose a language generation approach for targeted data generation with the participation of the training classifier. With a user in the loop, we find that manual annotation of a small proportion of the generated data is enough to boost classification performance. Thirdly, under a self-learning scheme, we replace the user by an automated approach in which the classifier is trained on its own pseudo-labels. Finally, we extend the data generation approach to the knowledge distillation domain, by generating samples that a teacher model can confidently label, but not its student.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

1.1 Motivation

Text classification is a classical problem in the Natural Language Processing (NLP) domain, which involves assigning labels to textual sequences, e.g. words, sentences, paragraphs or documents, for the purpose of splitting them into categories. This enables a variety of NLP applications such as sentiment analysis, document categorization, spam and fraud detection, language detection, etc. In machine learning, algorithms learn to classify text by observing patterns from existing textual samples. By training on labeled data, a machine learning model learns to associate patterns between the textual samples and their labels. In classical machine learning, training a model on text requires a pre-processing stage which involves converting the input data to an appropriate format for training, such as removing stop words, removing non-alphanumeric characters, lower casing, and normalizing words. Pre-processing is usually followed by a feature engineering process in which task-specific features are extracted from the data, that are then converted to numerical representations, as will be explained in section 2.3.1. Unsurprisingly, classical approaches present multiple limitations that impede learning. For instance, extracting appropriate features requires domain knowledge and putting in additional work involving data analysis. This also makes classical machine learning models difficult to generalize to new tasks, as the extracted features are specific to the current task (Minaee et al., 2021). Furthermore, the limited complexity of classical machine learning models such as linear classifiers make it difficult to take advantage of large data collections.

In recent years, deep learning models have been shown to improve learning progress over classical methods, thus making them the default choice for learning from data. The ability of these models to automatically learn representations over multiple layers reduces the need for feature engineering. Recent deep neural network architectures, e.g. Transformers (section 2.2.7), are able to process enormous amounts of data and generalize over multiple tasks. The

ability to utilize larger volumes of training data has always played a key factor in improving performance. With higher diversity in the training examples, more underlying patterns can be detected thus enabling better generalization. As expected, when the availability of training data is questionable, it becomes difficult to generalize the trained model to the target task. Let's consider a real-life scenario in which a company plans to create a chatbot agent for customer support. Part of the implementation requires training intent classification models. These models are responsible for categorizing incoming messages based on their meaning. For instance, a customer that asks "When will my parcel be shipped?" is requesting shipping information. By logging customers' inputs, the company can compile a dataset for training an intent classifier. However, for different reasons, companies may not always have access to such data. For instance, customers may not have given consent for their data to be collected, or maybe the company simply does not have chat logs. Potential solutions involve the manual creation or purchase of training data. Once the required models have been trained, the company deploys the chatbot and starts collecting data from people's engagements. The models can then be retrained on the newly collected data. The objective of training a classifier is to properly fit to the generalized information from the training data. However, when the data is not sufficient, the classification model could potentially focus more on irrelevant or noisy features. In the example of the chatbot, the purchase or manual creation of data can be expensive or time-consuming. In the worst case, the data may not even be available for purchase.

Consider a low data setting, where a classifier is trained on the examples in Table 1.1. Without external knowledge, it can easily associate the word "movie" with a positive sentiment. To improve the model's generalization, examples with a negative sentiment containing the word "movie" can be added to the training set. Hence, with more data, more distinctive patterns can be derived.

Example	**Sentiment**
Loved this movie	Positive
Brilliant movie	Positive
Sad	Negative
Tragic ending	Negative

Table 1.1 Training data for sentiment classification

The emergence of transfer learning (refer to section 2.3) in NLP has enabled the possibility of better generalization over smaller datasets. This is made possible due to external knowledge gained from previous training steps, which is then transferred to the target task. This transfer happens when the data of the target task, e.g. table 1.1, is mapped to a feature space

by leveraging knowledge that is learned in the previous training, also known as the pre-training step. This being said, the size of the target task dataset plays a major role in the performance of the final model. A model trained on a very small dataset may not learn proper generalization due to the lack of patterns that can be learned from fewer samples. Overtraining models on small datasets may lead to over learning noise in the data, an issue known as overfitting, please refer to section 2.6. The obvious way to improve learning in such settings, would be to collect more data samples. However, this may not be possible when data is not widely available or easily accessible. To counter this, data augmentation techniques have been proposed as an alternative or a complementary step to transfer learning. With transfer learning, improvements are done on the model through pretraining. In data augmentation, on the other hand, modifications are done on the data side. Here, the diversity of the data samples is increased without directly collecting more data (Feng et al., 2021). In NLP, multiple applications can benefit from data augmentation, some of which will be explained in the following section.

1.1.1 Applications of Data Augmentation (DA) in NLP

Due to challenges presented by the discrete nature of language, augmentation remains at an early exploration stage in the NLP domain (Feng et al., 2021). With this being said, some NLP applications do benefit from existing DA techniques. These include, but are not limited to: bias mitigation, applications in low resource languages, and the correction of class imbalance.

Bias Mitigation: Deep neural networks are trained to automatically discover and learn patterns in existing data. Although training data plays a crucial role in building good models, it sometimes directly leads to undesirable model behaviors. For instance, Brown et al. (2020) discovered high association between gender and occupation, where 83% of 388 occupations were more likely to be associated with a male identifier by GPT-3[1]. Other studies including Buolamwini and Gebru (2018); Dastin (2018), have shown that gender bias in the data can be reflected in the trained model. Irrespective of their performance, neural networks are prone to propagating and amplifying the biases found in the data while making predictions. For example, textual data which can have heavy gender presumptions can result in a trained model that reinforces stereotypes in downstream applications (Sun et al., 2019c). Zhao et al. (2018) attempts to mitigate gender bias by applying coreference resolution augmentation to the existing data. The added examples, created by simply swapping the pronouns “he” and “she”,

[1] GPT models are explained in section 2.3.2

can mitigate a model's assumptions in certain contexts, such as gendered occupations. For example, occupational stereotyped gender bias can be a result of more frequent observations where professions like "doctor" are linked to the pronoun "he", or "nurse" linked to the pronoun "she". By creating copies of such observations with the gender pronouns switched, occupations are intended to become less coreferent to specific genders. Although such an approach may help mitigate bias in certain scenarios, it mainly relies on the application of heuristic rules. This could create nonsensical sentences – for example, "he gave birth", a result of gender-swapping "she" with "he" in "she gave birth" (Madaan et al., 2018).

Low-Resource languages: Languages with limited resources suffer from the unavailability of sufficient data to train models. Compared to highly resourced languages like English, languages with limited data sources present challenges for NLP applications, especially in Neural Machine Translation (NMT). NMT systems are trained to predict translations in a parallel corpus. That is, a model is given two sequences; sequence A from a source language, and the translation of A in the target language. The model is then trained to map the input sequence in the *source* language to the sequence in the *target* language. NMT systems are typically based on training an encoder-decoder architecture, refer to section 2.2.7 for an example of such architecture. In a setting when only monolingual datasets are available, this training procedure becomes infeasible. Since DA techniques like simple word replacements can cause semantic changes to the sequence, there is a lack of previous literature on DA for Machine Translation (MT) (Maimaiti et al., 2022). Li et al. (2020) augment both the source and target training data by applying backtranslation and self-learning augmentation strategies. The main idea behind back-translation is to train a translation model on a parallel dataset to translate from the target language to the source language of the original dataset. The reverse translation model is then used to generate samples of source language from the monolingual target data. Self-supervision works in the same way as back-translation, but is applied to the monolingual source data.

Correction of Class Imbalance: A dataset becomes imbalanced when the data-per-label proportions are skewed such that some labels are represented by more data samples than others. Imbalanced datasets can lead to poor classification performance as the training model is bound to learn more of the patterns in the data of the majority class over the minority class. This can make the model biased towards the majority class. One approach to class imbalance would involve oversampling the minority class. With data augmentation, this can be done by creating additional synthetic samples from the distribution of the minority class. This can involve the application of simple data augmentation techniques such as the

deletion, insertion, or addition of random words in text (Wei and Zou, 2019), paraphrasing or back-translation (Sennrich et al., 2015a), and most recently the application of language models to predict alternative words (Kobayashi, 2018).

1.1.2 Problem Statement

The main objective of data augmentation is to enhance the diversity of the training data, in order to better generalize the learning model to unseen data. Data can be augmented by creating either new samples or samples that are modifications to existing data. The focus of this PhD revolves around the creation of new data samples through the application of a language generation model. Previous works in DA, discussed in section 2.1.4, assume the availability of enough training examples to bootstrap the generation process. Due to the lack of publicly available small datasets, researchers in data augmentation showcase their approaches by experimenting on fractions of an existing dataset. For example, Wei and Zou (2019) conducted their experiments on five datasets containing between $4k$ and $39.5k$ samples. They then applied their DA approach to as little as 1% of the full training sets (i.e., data subsets between approximately 40 and 395 samples), but only to achieve marginal improvements over the baseline of non-augmented training data. In other works, Regina et al. (2020) experimented with fractions as low as 10% of the original data, resulting in subsets with more than 500 training samples. Similarly, in the experiments conducted by Longpre et al. (2020), the smallest data subset contained 500 samples. Other works, like Zhang et al. (2015) used data subsets as large as $40k$ training samples. Finally, Jiao et al. (2019); Kobayashi (2018) did not mention any sampling of the datasets used, from which we can assume their experiments were performed on the full training datasets. Unlike most previous approaches in DA, in this thesis, our goal is to achieve tangible improvements in classification performance for real-world applications where data can be extremely scarce. Hence, in our experiments, we simulate data scarcity settings by restricting the size of the training set to as little as a few samples per label.

1.2 Research Aims

This research aims to investigate the possibility of combining knowledge from multiple pre-trained language models to further improve performance on target tasks with datasets as small as a few examples per label. Here we raise the following research questions:

1. Can we generate synthetic examples that complement an existing dataset?

2. Is it possible to condition the data generation process to only output examples that satisfy preset conditions?

3. Are we able to limit the generated data only to the samples that could lead to improved performance?

1.3 Contributions

The main research contributions in this thesis can be summarized as follows:

- We propose a novel method for data augmentation through text generation (section 3.3)
- We propose a framework for conditional language generation using a combination of a language model, a text classifier and the Monte Carlo Tree Search (MCTS) algorithm (section 3.3.2)
- We present a novel method for data augmentation in knowledge distillation (section 5.3)
- We show that pooling BERT outputs across tokens and layers creates appropriate sentence representations for training on downstream tasks (section 4.4.3)
- We show that by applying the pooling strategy from the previous point, we can build an ensemble of multiple classification layers and a single BERT-based model. Considering the computational complexity of transformer-based models, this approach reduces the needed computational requirements as opposed to an ensemble of multiple BERT models (section 4.4.4)
- We fit a classifier on training data to generate clusters for each target label using the output of a 2-dimensional intermediate layer. Unseen examples can then be plotted against the training data to measure the distance to their closest label cluster (section 4.5.1)

1.4 Thesis Outline

Chapter 2 provides an overview of the background information that is necessary for understanding the concepts discussed throughout the thesis.

In chapter 3, we attempt to answer the first research question by exploring the application of a language generation model (GPT-2) to create complementary synthetic data that is then labeled by the user. We also answer the second research question by only including samples which are expected to lead to improved performance. The third research question is also addressed by the implementation of a reward-based searching procedure, in which GPT-2 is guided to generate examples that meet predefined conditions.

Chapter 4 attempts to extend the approach in chapter 3 by automating the labeling process for the generated data, hence, eliminating the need for a user to label the generated data. Our experiments however did not give consistent results to justify the reliability of this approach. We conclude that the labels provided by the user resemble external knowledge that is essential for improving the performance of existing classifiers.

In chapter 5, we redefine the learning problem from chapter 4 under a knowledge distillation paradigm in which one model, referred to as the "teacher", labels the generated data. Another model, called the "student", is then trained on the generated data with the pseudo labels of its teacher. This paradigm replaces the user with a classification model, the teacher, that provides labels essential to the improvement of another classification model, the student.

Chapter 6 finally provides a summary of the work done in this thesis while addressing our research questions.

1.5 Publications

The work described in Chapter 3 has been published in the following paper:

Quteineh, H., Samothrakis, S., & Sutcliffe, R. (2020, January). Textual data augmentation for efficient active learning on tiny datasets. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)* (pp. 7400-7410). Association for Computational Linguistics.

Some of the material in chapter 5 is presented in the following paper:

Quteineh, H., Samothrakis, S., & Sutcliffe, R. (2022). Enhancing Task-Specific Distillation in Small Data Regimes through Language Generation. In *Proceedings of the 29th Conference on Computational Linguistics (COLING)* accepted for publication.

The author also contributed to the following publication outside the theme of this thesis:

Quiroz Flores, A., Liza, F., **Quteineh, H.**, & Czarnecka, B. (2021). Variation in the timing of Covid-19 communication across universities in the UK. *PloS one*, 16(2), e0246391

Chapter 2

Background

This chapter will provide the needed background knowledge for the remainder of this thesis. In section 2.1, we start out by addressing the importance of data for machine learning. We then expand on this discussion by analyzing previous works in Data Augmentation (DA) in relation to our work, and explain important concepts in Natural Language Processing (NLP) such as language modeling and tokenization. Section 2.2 explains neural networks from the fundamental building blocks to more advanced architectures such as LSTMs and Transformers. Sections 2.3 and 2.4 follow up by discussing the evolution of language models in detail, with emphasis on recent techniques for vector-based representations of textual data in section 2.4. Since the success of recent models is demonstrated by measuring their performance over a variety of NLP tasks, in section 2.5, we discuss the appropriate evaluation metrics for text classification, and datasets that are commonly used as benchmarks. In sections 2.6 and 2.7, we explain ways to diagnose the performance of a trained model through analyzing its prediction error. Sections 2.8, and 2.9 build on this discussion by explaining different ways for improving the generalization of a classification model, with mentions of relevant recent findings discussed in section 2.10. In respect to this, section 2.11 discusses the explainability of a training model by providing an overview of uncertainty estimations. Finally, in sections 2.12 and 2.13, we discuss decoding and search strategies that are fundamental to our work in DA.

2.1 Data Preparation in NLP

In this section, we will cover the fundamentals of machine learning in a supervised learning setting. In an unsupervised setting, training is concerned with discovering hidden patterns between data points $x_0, \ldots, x_n$, discarding any mappings to target labels $y_0, \ldots, y_n$. Different from unsupervised learning, in a supervised setting, a machine learning model is trained to

learn a function that maps input x to output y. The process of learning data patterns in x is supervised by the mappings to target labels y. Normally, labels y are human generated, such that training becomes *supervised* through the data. Machine learning models are trained to learn a generalized representation that goes beyond the training data. Generalization is measured by evaluating the model's ability to adapt to new data; this is data that is drawn from the same distribution of the training data, but had not been seen before by the learning model. Two main components play a role in improving generalization; a) the complexity of the training algorithm, b) the data the algorithm is trained on. Machine learning algorithms in natural language processing have massively improved over the recent years. While leaps in performance can be gained from improvements to training algorithms, the quality of the training data remains a key component of the learning process. In real-world implementations, data scarcity can become a real bottleneck to performance. Regardless of a model's complexity, the lack of sufficient training data can always cap its performance. In section 2.1.1, we discuss the importance of data for machine learning.

2.1.1 Data for Machine Learning

Data plays an integral part of any supervised machine learning system. The widely accepted phrase in computer science, "Garbage in, Garbage out" reflects the importance of data quality in training machine learning models (Geiger et al., 2021). It has been shown that data plays a vital role in shaping performance, and thus regardless of the training model, low quality or biased data (explained in section 1.1.1) can have a negative influence on the final prediction (Geiger et al., 2020). Likewise, high quality classifiers can only be made from high quality training data. To train a machine learning model, it is always important to first prepare the data in an appropriate format. In classical NLP, the text input is broken down into word or sub-word level units, called tokens. A vocabulary can then be constructed from the unique tokens, and used as features by the learning model. In the following section, we discuss popular tokenization methods.

2.1.2 Tokenization

Tokenization is a fundamental step in almost any NLP pipeline, in which text sequences are segmented into smaller units called tokens. A token can either be a word, a sub-word or an individual character. One of the simplest forms of tokenization is splitting text by whitespace. Despite its common application, this type of tokenization has multiple drawbacks. For example, whitespace tokenization fails with languages that do not use spaces, e.g. Chinese and Japanese. Another obvious obstacle is with hyphenated compound words like "two-fold",

"up-to-date", and "state-of-the-art". It may also be more useful to split words containing punctuation like the connotations "don't", "Mike's". Last but not least, splitting tokens by whitespace can massively increase the vocabulary size, which as a result may cause memory and performance issues. For the context of this thesis, we will discuss two subword tokenization techniques, Byte-Pair Encoding (BPE), and WordPiece.

Byte-Pair Encoding In 1994, Gage (1994) proposed the original Byte Pair Encoding (BPE) as a method for data compression, in which the most frequent pair of bytes in a sequence are replaced with a single unused byte. In 2015, Sennrich et al. (2015b) adopted the same algorithm, but for encoding characters in a string rather than bytes. BPE starts by constructing a vocabulary of the individual characters in the data. The algorithm then counts the frequency of each pair of characters, and adds them as tokens to the vocabulary. The vocabulary is then continuously updated to include the most frequent consecutive pairs of merged characters in the vocabulary. The process of adding frequent token pairs as a single token to the vocabulary is repeated until the desired size of vocabulary is achieved.

WordPiece WordPiece, originally introduced by Schuster and Nakajima (2012), is another sub-word algorithm that behaves similarly to Byte-Pair Encoding. The main difference, however, is in the process of selecting which pairs of tokens to merge. Instead of joining a particular pair of tokens based on their combined frequency counts, WordPiece selects pairs that maximize the likelihood of a probabilistic language model. This is done by: A) first building a language model on the training data using the initialized character-based vocabulary. B) Then selecting the pairs which lead to the largest increase in likelihood once merged. For example, if "er" is more likely to occur than the individual characters "e" and "r", "er" is said to increase the likelihood probability once added to the vocabulary. The increase in the likelihood probability is measured by taking the difference between the probability of "er" occurring minus the probability of "e" and "r" occurring individually. Steps A and B are repeated until a preset vocabulary size is reached, or until the increase in the likelihood falls below a certain threshold.

2.1.3 Language Modeling

A language model calculates a probability distribution over a sequence of tokens. When passing over a stream of text, a probability distribution is calculated over the entire vocabulary by assigning each vocabulary token a probability score for occurring next in the sequence. For example, for the sequence "What time is", the probability that "it" occurs next, would be

conditioned on the initial sequence:

$$P(\text{it}|\text{what time is})$$

This conditional probability is calculated for every other token in the vocabulary. Hence, tokens with higher probability scores are more likely to appear next in the sequence. This process allows us to calculate the probability for a sequence of tokens $w = w_1, ..., w_n$, and can be defined as:

$$P(w) = \prod_{i=1}^{i=n} P(w_{(i)}|w_{(1)}, \ldots, w_{(i-1)}) \tag{2.1}$$

As such, given a set of sentences as input, a language model is trained to find the parameters θ that maximize the log-likelihood:

$$\theta^* = \arg\max_{\theta} \{\log P(w;\theta)\} \tag{2.2}$$

This assumption constrains the language model to Markov's assumption:

$$P(w_i|w_1, \ldots, w_{i-1}) \approx P(w_i|w_{i-n+1}, \ldots, w_{i-1}). \tag{2.3}$$

That is, the probability of a token occurring next in a sequence can be approximated by conditioning its probability over a fixed number n of tokens. In this way, the larger n is, the more input tokens contribute to approximating the conditional probability of any token w_i to appear next in the sequence $w_1 \ldots w_{i-1}$. With the advances in deep learning, more recent architectures e.g. Transformers (section 2.2.7) are capable of capturing longer sequences of n, and as such are able to make use of larger contexts in approximating conditional probabilities. In the same way, the availability of appropriate data is essential for proper training. On the one hand, the larger the training set is, the more the training model is able to learn the patterns in the data. On the other hand, when the data is inadequate for training, it is less likely for the model to learn the appropriate representations. However, with Data Augmentation, new data points can be generated from the existing data. In the following section, we discuss Data Augmentation methods that are commonly applied in NLP.

2.1.4 Data Augmentation (DA)

The artificial expansion of training data has shown great success in the computer vision domain Shorten and Khoshgoftaar (2019). Basic image augmentation techniques include geometric transformations, e.g. rotation and cropping, color space transformations as in

Easy Data Augmentation Operation	**Augmentation**
No Operation	I loved this movie.
Random Swap	loved I this movie.
Random Deletion	I this movie.
Random Insertion	I loved this.
Random Synonym Replacement	I liked this movie.

Table 2.1 EDA operations on the example "I loved this movie".

changes in hue and brightness, random erasing of pixels, kernel filters e.g. blurring and sharpening, and mixing images. In NLP, data augmentation is usually done by creating synthetic examples through a process of manipulating existing ones. A transfer set is then constructed from the augmented data samples, to train the classification model. We will discuss four popular DA techniques in NLP; word-manipulation, back-translation and generative augmentation.

Word manipulation involves the insertion or re-arrangement of words in the text input. For example, Easy Data Augmentation Methods (EDA) proposed by Wei and Zou (2019), involve randomly swapping words, replacing words with their synonyms, or deleting random words. Examples of these operations are shown in Table 2.1. Although this approach was found successful and is widely used in downstream tasks, it can corrupt or lead to the loss of important features. Anaby-Tavor et al. (2019); Qiu et al. (2020) reported instances of loss in performance with EDA. This could be attributed to operations like random swap and deletion that do not guarantee label preservation. For instance, for a sentiment prediction task for the target labels positive, and negative, the word "love" would be an important feature, that indicates a positive sentiment. Deleting this word from the example "I love this movie" would result in "I this movie", where meaning is lost. Word swapping can also change the intended meaning, for example, swapping the words "terrible" and "good" in the example, "The plot was **good**, but the acting was **terrible**" would result in "The plot was **terrible**, but the acting was **good**".

In a similar approach, Wang and Yang (2015) randomly replace words with neighboring ones from an embedding space. Although synonym replacement can lead to improved results, this approach does not guarantee a high diversification of text when applied on a single word only. Also, if the replaced synonym does not fit the context, the augmented sentence would not be linguistically correct. Jungiewicz and Smywiński-Pohl (2019) attempt to solve this issue by restricting the word replacement to certain parts of speech. Although this approach

does not entirely solve the problem and limits the diversification of the augmented text, the authors reported improved results.

Instead of synonym replacement, approaches like Kobayashi (2018) use contextual language models to find words that fit the context as much as possible. Here, the language model takes the sentence as input excluding the word at position x, to predict an alternative word at that same position, x. In a similar approach, but for a machine translation task, Wang et al. (2018b) replace words in both the source and the target sentences with other random words. While these approaches can produce linguistically correct sentences, they do not necessarily guarantee the preservation of the meaning. As a result, this could distort label preservation. For example, "I loved the movie" and "I hated the movie" are valid transformations, but reflect different sentiments.

Back-translation , also known as round-trip translation Aiken and Park (2010), transforms the input text through multiple translation steps. Aroyehun and Gelbukh (2018); Sennrich et al. (2015a) translate the text into an intermediate language, then translate back the result to the original language. This technique can result in outputs that are paraphrases of the original text. Paraphrasing using back translation can lead to promising results when the back-translation models are able to preserve the meaning of the original text while generating several diverse paraphrases. Although this approach produces valid results, it comes at the cost of applying multiple translation models.

Generative augmentation The recent advancements in Natural Language Generation (NLG) have made it possible to artificially create coherent and well-formed text samples. This breakthrough in NLG can be attributed to the ability of recent neural network architectures to digest large collections of datasets. These are known as Transformer-based architectures, explained in section 2.2.7. The Generative Pretrained Transformer (GPT), explained in section 2.3.2, pushed the boundaries in NLG with its ability to generate conditional synthetic text samples of unprecedented quality. This made natural language generation a viable option for textual data augmentation.

It is a common practice to preserve the class label in data augmentation. In computer vision, DA with label preservation can be easily applied in many downstream classification tasks. For example, in a 'dog' and 'cat' image classification task, an image of a dog can still be labeled as 'dog' even if it has been rotated, cropped, sharpened, or even blurred. However, in textual data, as we've previously seen, even simple changes to an input could cause semantic changes, making label preservation troublesome. Similarly, NLG DA is not immune from this problem. In an attempt to preserve the label, Anaby-Tavor et al.

(2019); Kumar et al. (2020) conditioned a language generation model on the class label. That meant new samples were generated by using the class label and a few initial words as the prompt for the model. The traditional approach to language modeling is to predict the next word in a sequence using the past words as context. This process is explained in section 2.1.3. Although the used GPT model can produce coherent text, Kumar et al. (2020) acknowledges its inability to properly preserve the label. To overcome this limitation, Anaby-Tavor et al. (2019) made use of the discriminative baseline classifier that was trained on the initial training set. In this way, only the synthetic samples that were predicted with higher confidence by the classifier were retained. In chapter 4, we show that examples predicted with high confidence may not be the most useful for improving its classification performance. In fact, it can be noticed from their results (Anaby-Tavor et al., 2019), that the gain in performance was not consistently high, with the lowest gain being 1.4% over the baseline model. Instead of assigning restricting conditions like classification confidence to select the transfer set examples, Yoo et al. (2021) train the classifier on the probability distributions of the generation model. To do so, the authors apply a different conditioning criterion from Anaby-Tavor et al. (2019). Instead of conditioning on the class label, Yoo et al. (2021) prompt the NLG model with a template that includes a few training samples with their label. By applying this in a few shot learning setting, explained at the end of section 2.3.2, the authors expect the NLG model to output a label with each sample it generates. To produce considerable results with this approach, a substantially larger language model is required, thus adding additional computational costs. The current variations of GPT-2 and their differences are explained in section 2.3.2.

In this thesis, we augment data through generative data generation. By tuning the NLG on the initial training dataset, we are able to generate text examples from the same distribution. We transform the data generation task into an optimization problem which maximizes the usefulness of the generated output, using Monte Carlo Tree Search (MCTS), explained in section 2.13.3, as the optimization strategy. Instead of limiting the generated samples to a narrow subset of samples with high classification confidence as done by Anaby-Tavor et al. (2019), in chapter 3, we actually consider the samples that return low classification confidence. We show that a transfer set constructed from such samples can lead to substantial classification improvements. To guarantee the correct labeling of the data, we rely on a user to manually review the transfer set. This being said, the manual effort is minimized by limiting the number of the examined samples. In chapter 4, we remove the user from the labeling process by applying an automated approach that retains the labels of the base classifier. Like Anaby-Tavor et al. (2019), we include samples predicted with high confidence in the filtration step. A comparison of results between chapters 3 and 4 shows that the benefits of manual

labeling can outweigh the easiness of the automated process of chapter 4. Finally, in chapter 5, instead of relying on a human annotator, we use a larger classification model to generate labels. This approach falls under knowledge distillation, where a larger "teacher" model benefits a smaller "student" model. Overall, we rely on external knowledge for assigning labels to the generated examples. In chapter 3, we rely on the user to hand label the generated data, whereas in chapters 4 and 5, we pseudo label using a classification model trained on the original training dataset.

2.2 Neural Networks

Neural networks are a common application to many Natural Language Processing (NLP) tasks. In this thesis, we train neural network models on classification tasks in NLP. Neurons are the core elements in a Neural Network. When put together, a set of neurons becomes known as a layer. As a neural network processes input, information is forwarded through its layers, starting from the input layer, leading towards the output layer. Layers can be stacked to form a hierarchical architecture that processes incoming data in sequential order. As their connections are sequential, successive layers have to first receive their input from earlier layers before performing computations. Yet, within a single layer, multiple computations can be performed in parallel. When a layer processes an input, each of its neurons first applies a linear transformation to the input values:

$$f(x) = b + \sum_i w_i x_i \tag{2.4}$$

where w_i and b are adjustable learning parameters. w_i is a weight at cell i, x is an input vector, x_i is the input value at dimension i in vector x, and b is a bias term that adds flexibility to allow a better fit on the data. The neuron then alters the output of $f(x)$ by passing it to a non-linear activation function.

2.2.1 Non-linear Transformations: Activation functions

Activation functions are also known as squashing functions, as they keep a neuron's output within a certain range. A popular activation function that is applied to the intermediate layers of a neural network is the rectified linear unit ReLU:

$$R(z) = max(0, z) \tag{2.5}$$

where z is the output of the linear classifier in equation 2.4. As shown in Figure 2.1, ReLU activates a neuron by allowing positive values to pass through unchanged, while converting negative values to 0, as seen in Figure 2.1.

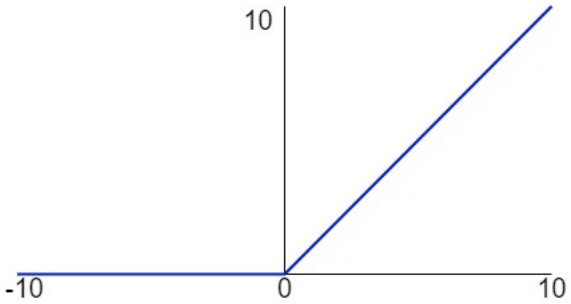


Fig. 2.1 ReLU activation function

Another popular activation function is the Sigmoid that limits the output to the interval $(0,1)$ as shown in equation. 2.6.

$$\sigma(z) = \frac{1}{1+e^{-z}} \tag{2.6}$$

where e is the exponential constant, equal to 2.718. The response of the sigmoid function $sig(x)$ can be seen in Figure 2.2.

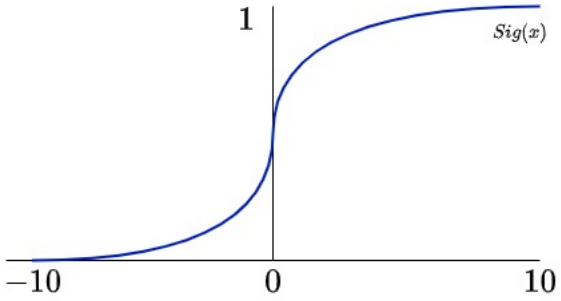


Fig. 2.2 Sigmoid activation function

To squash the output to the range $(-1,1)$ as shown in Figure 2.3, the tanh activation function from equation 2.7 can be applied:

$$\tanh = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} \tag{2.7}$$

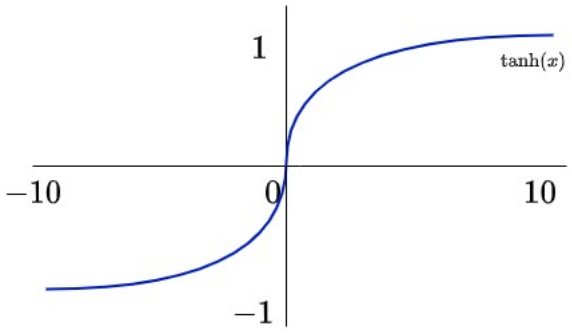


Fig. 2.3 tanh activation function

Finally, for classification tasks, the Softmax activation function is commonly applied to the outputs of the final layer:

$$softmax(z_i) = \frac{\exp(z_i)}{\sum_{j=0}^{n} \exp(z_k)} \tag{2.8}$$

Unlike previously mentioned activation functions, the softmax is not applied to a single input, but rather multiple inputs. It maps its input values to a probability distribution where the sum of the outputs is equal to 1. Hence, it produces a range between 0 and 1, where the target class has the highest probability. This makes the softmax an ideal activation function on a neural network's output layer for multi-class classification tasks. Unlike a binary classification task, in a multi-class task, the number of output classes y is more than two. Here, each input can be mapped to only one of the possible categorical outputs. Since the sum of the outputs of the softmax must be equal to 1, in order to assign one class a higher probability, the probability values for the remaining classes must be pushed down.

2.2.2 Feed-forward Neural Network

The simplest form of a network contains only one output neuron for binary classification, known as a Perceptron. In a multi-layer neural network, the input and output layers are separated by a series of consecutive layers known as hidden layers. The arrangement of layers in a single direction connection is known as a feed-forward network. In this arrangement, each neuron in one layer is directly connected to every other neuron in the next layer. This means that information flows in a forward direction, from one layer to the next. If certain connections in a network form a cycle, the network is known as a Recurrent Neural Network, explained in section 2.2.5.

In terms of communication, the process of passing information from the input layer through a network to the output layer is known as forward propagation. This is done by

calculating the activation output at each neuron for each successive hidden layer until the final output is reached. When training the network, the output of the final layer is compared to the true target value y. If the network's prediction $\hat{y}$ is far from y, the weights of the network are adjusted accordingly. The distance between $\hat{y}$ and y is computed by applying an appropriate *cost function*.

2.2.3 Cost functions

Cost functions, also known as loss functions, measure the loss of information between the network's predictions $\hat{y}$ and the target values y. An intuitive approach for measuring the distance is by taking the difference between y and $\hat{y}$. The average of the loss over n samples would then be:

$$Loss = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i) \tag{2.9}$$

However, this solution could be problematic as distances with different signs, e.g. 1 and -1, could cancel out each other, thus leading to a bad estimation of performance. This problem can be addressed by taking the absolute difference between y and $\hat{y}$:

$$Loss = \frac{1}{n}\sum_{i=1}^{n}(|\hat{y}_i - y_i|) \tag{2.10}$$

Although positive and negative values do not cancel each other when averaged, this solution is still problematic as it is not differentiable. The optimization of a neural network consists of searching for an appropriate set of weights which minimize the loss value, i.e. closing the gap between $\hat{y}$ and y. In Gradient Descent, explained in section 2.2.4, the loss must be differentiable with respect to the weights at value 0. However, since the loss in equation 2.10 does not have a derivative, it becomes problematic for calculating gradients. To make the loss differentiable, we can take the mean-squared difference between $\hat{y}$ and y. This is known as the Mean Squared Error MSE, equation 2.11:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \tag{2.11}$$

MSE is mostly applicable to regression tasks. For classification tasks, it is more common to compute the loss of information with cross entropy. The cross entropy measures the difference in bits of information between two probability distributions. The logarithmic nature of cross entropy, equation 2.12, yields larger scores for differences approaching 1, but smaller scores for differences closer to 0.

$$H(\mathbf{y},\hat{\mathbf{y}}) = -\sum_{\mathbf{i}} \mathbf{y_i} \log_{\mathbf{e}}(\hat{\mathbf{y}}_\mathbf{i}) \tag{2.12}$$

2.2.4 Backpropagation

The objective of training a neural network is to minimize the value of the loss function. Since weights of a network are variable elements, changing their values would affect the network's final predictions. In an attempt to improve a network's fit to the training data, one could randomly shuffle the values of its weights until the loss is minimized. However, such an uncontrolled approach may take forever to converge, i.e. to find weights that minimize the loss. Considering that loss functions measure the distance between a prediction and the true value of the target, a network's fit to the training data can be improved by adjusting its weights and biases accordingly. After a complete forward pass through the network, the loss is measured using an appropriate cost function. The weights of the network are then adjusted by backpropagating the error to the neurons in the network. This process is done by computing the partial derivative of the cost function C with respect to the weights $\partial C/\partial w$ and biases $\partial C/\partial b$ in the network.

Gradient descent As explained earlier, in the backpropagation step, we aim to learn the gradient of the loss function with respect to the weights and biases. In gradient descent, the loss function of the neural network is minimized by shifting the weights along the negative direction of the gradient:

$$\theta = \theta - \alpha \frac{\partial C}{\partial \theta} \tag{2.13}$$

where θ represents the network's parameters including weights and biases, α is a *learning rate* that controls the magnitude of the update to the network's learning parameters. For the backpropagation updates to be computed, it is required to run the training examples through the network through a full forward and backward pass. Computing the gradient for the entire dataset in one shot can be impractical due to computational complexity issues. This is especially true for higher numbers of learning parameters. However, it is possible to compute the gradient descent for every example separately while still providing an accurate direction of the overall movement. In Stochastic Gradient Descent SGD, a full cycle of gradient computations is performed on individual examples in random order. Because SGD performs updates on one example at a time, it can heavily fluctuate during its updates and overshoot the local minimum. By lowering the learning rate, the effect of overshooting the local minimum can be managed. Nevertheless, SGD can still provide poor results for certain orderings of data points (Aggrawal, 2018).

Mini-batch gradient descent balances between a single shot gradient descent update and SGD. Instead of performing updates on the full dataset in one shot, in mini-batch gradient descent, each update is done on a random subset of the data points. A training dataset can be divided into n batches $B = b_i \dots b_n$, such that for all batches $b_i \in B$, the update is:

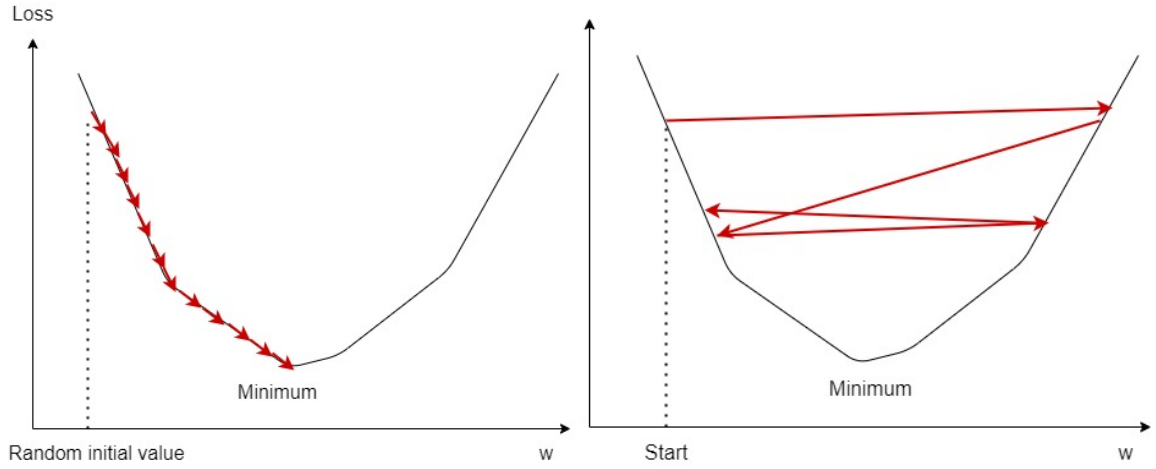
$$\theta = \theta - \alpha \sum_{j=i}^{n} \frac{\partial C_j}{\partial \theta} \tag{2.14}$$

With mini-batch gradient descent, the gradient computations are done on matrices instead of vectors, hence, providing stability over SGD, and lower computational requirements over one-shot gradient descent.

Learning Rate The learning rate α controls the amount of update a parameter receives during training. On the one hand, too small a learning rate can cause small updates in weights, and lead to slow training, as more update steps are required for the network to converge. On the other hand, a higher learning rate can cause the network to converge too quickly to suboptimal solutions. Suppose we have a one-dimensional problem such that the network's parameter consists of only weight w. The loss can be plotted as a function of w as shown in plots (a), and (b) in Figure 2.4. Starting from a random initialization point, the gradient descent iterates in the direction of a solution that minimizes the error. Plot (a) shows gradient descent updates with small step sizes due to a smaller learning rate α. In this case, the algorithm can take many steps to find a minimum point. Plot (b) shows updates with large step sizes, as a result of a high value for α. Here, the updates can be too large, so that the weights at which the error is minimized are always skipped. This can cause training to fail to converge.

The global minimum is the point that achieves the lowest error over all the parameter values, also known as the global optimum. The local minimum, or local optimum, is a point at which the loss is minimized only in a neighborhood of weights. For the neural network to properly learn, it is important to choose a learning rate that is large enough to not get stuck in a local minimum, but also small enough to not oscillate around a local minimum or overshoot the global minimum. Figure 2.5, shows a weight optimization in which the error is reduced to a global minimum.

Finding a good value for the learning rate can be a difficult task, yet very important for achieving good convergence. For this reason, it is common to tune the learning rate by applying grid search over a space of preset values, e.g. $\{0.5, 0.1, 0.05, 0.01, \dots, 0.0001\}$, such that the value that achieves the best performance is selected. Furthermore, other gradient update algorithms including modifications to SGD have been proposed to overcome the



(a) Small Learning Rate (b) Large Learning Rate

Fig. 2.4 **(a)** Model converges in slow steps **(b)** Model may overshoot the local minimum and not converge

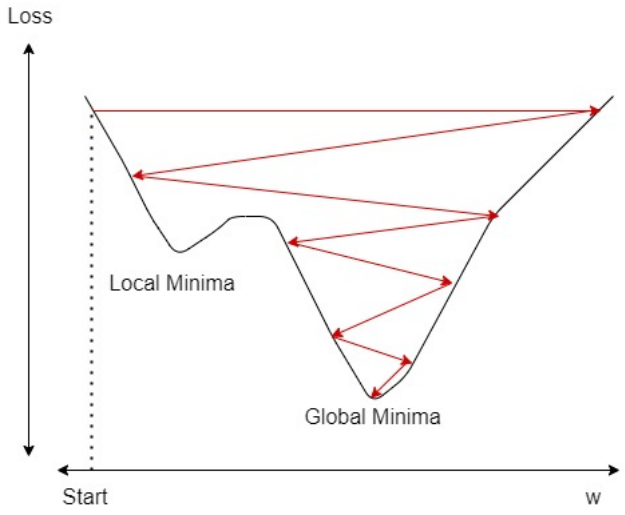


Fig. 2.5 Weight w tuned to minimize error to a global minimum

aforementioned convergence challenges. We will consider the Momentum term (Qian, 1999), RMSprop (Hinton et al., 2012), and ADAM (Kingma and Ba, 2014).

Momentum The momentum is a technique that helps dampen the effect of oscillating gradients while also speeding up training. This can be done by modifying the gradient update to include the average of the updates over the previous t steps. Without momentum, the immediate gradient becomes smaller as an optimum point (local or global) is reached. By keeping track of the moving average of gradient updates, with momentum, updates are increased for gradients that point in the same direction, and decreased for gradients with different directions. This is seen in equation 2.15:

$$\begin{aligned} v_t &\leftarrow \rho v_{t-1} + \alpha \nabla J(\theta_t) \\ \theta &\leftarrow \theta + v_t \end{aligned} \tag{2.15}$$

where α is the learning rate, θ $\nabla J(\theta_t)$ is the gradient for the parameter θ evaluated at step t, and ρ is the *momentum* hyperparameter such that $0 \leq \rho \leq 1$. For gradient descent:

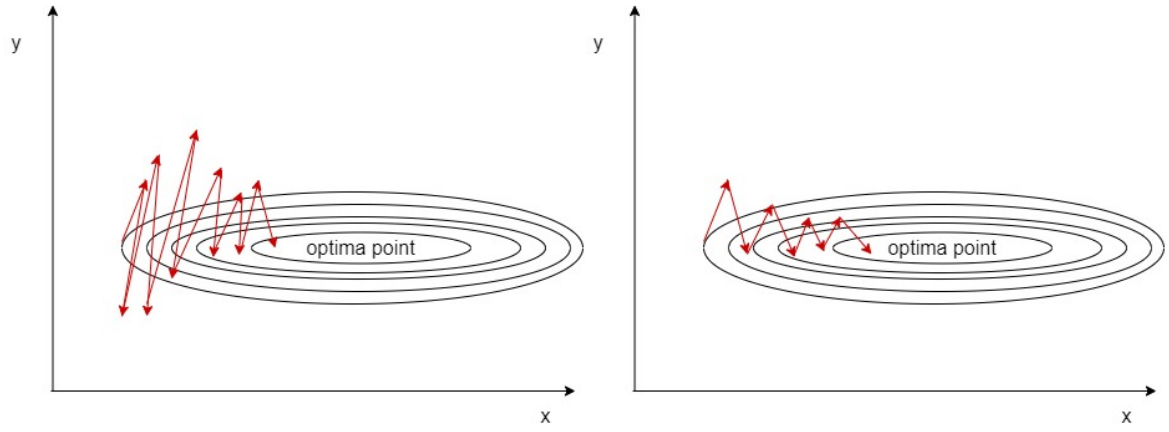
$$\nabla_\theta J(\theta) = \frac{\partial C}{\partial \theta} \tag{2.16}$$

Note that if we unroll v_t, we can see that the most recent gradients are given more weight than older gradients:

$$v_t = \alpha \nabla_\theta J(\theta_t) + \rho(\alpha \nabla_\theta J(\theta_{t-1})) + \rho^2(\alpha \nabla_\theta J(\theta_{t-2})) + \cdots + \rho^n(\alpha \nabla_\theta J(\theta_{t-n})) \tag{2.17}$$

For $\rho = 0$, equation 2.17 yields the standard gradient descent. For $\rho > 0$, the update shrinks in size if gradients frequently change their signs, and grows in size when gradients maintain the same sign. The intuition behind momentum can be further clarified in figure 2.6, where x and y represent weights, e.g w_1, w_2, in the horizontal and vertical directions respectively. The red arrows show the direction of the steps taken to optimize weights towards an optimum point. With momentum we can see that oscillations are reduced and thus convergence towards a local minimum point is accelerated.

RMSprop Root Mean Squared Propagation (RMSprop) is another popular algorithm that can accelerate the gradient descent. Unlike Momentum that has a fixed learning rate for all the parameters in each update step, with RMSprop each parameter undergoes a separate update with a different learning rate. In doing this, updates for weights towards the vertical direction are reduced to dampen oscillations, while making sure that weights in the horizontal



(a) SGD without momentum (b) SGD with momentum

Fig. 2.6 **(a)** SGD takes large steps toward an optimum point. **(b)** Momentum reduces oscillations towards the optimum. Image adapted from: Orr

direction are not reduced as a result. This end effect is shown in Figure 2.7; gradients descend with larger steps towards a local minimum, but with fewer oscillations in the vertical direction. For each parameter θ^j, RMSprop computes the average of squares of gradients:

$$\begin{aligned} v_t &= \rho v_{t-1} + (1-\rho) g_t^2 \\ \theta_t^j &= \theta_t^j - g_t \frac{\alpha}{\sqrt{v_t + \varepsilon}} \end{aligned} \tag{2.18}$$

where ρ is an RMSprop tunable parameter, and is generally set to 0.9, g is the gradient at step t along parameter θ, refer to equation 2.16, and ε is to ensure numerical stability, by avoiding divisions by 0, and is generally set to e^{-10}.

ADAM Adaptive Moment Estimation (ADAM), combines both Momentum and RMSprop into one algorithm. Instead of only storing the moving averages as in RMSprop, ADAM also stores the exponential decaying average of the momentum. As such, ADAM's gradient updates for each parameter θ^j include a momentum update:

$$v_t = \rho_m v_{t-1} + (1-\rho_m) g_t \tag{2.19}$$

as well as an RMSprop update:

$$s_t = \rho_m v_{t-1} + (1-\rho_r) g_t^2 \tag{2.20}$$

Before the final update, a bias correction is applied to both v_t and s_t:

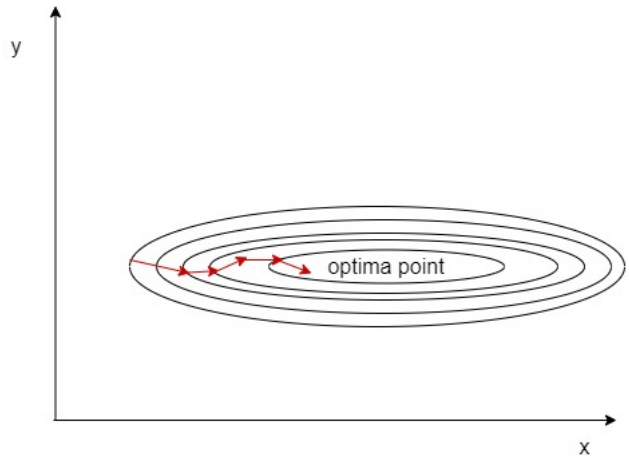


Fig. 2.7 RMSprop gradient convergence towards local minimum. Image adapted from: Ng (2017)

$$v_t = \frac{v_t}{1-\rho_m} \quad s_t = \frac{s_t}{1-\rho_r} \tag{2.21}$$

where p_m is a momentum hyperparameter, usually set to 0.9 and p_r is an RMSprop hyperparameter, which is usually set to 0.999. The final weight update becomes:

$$\theta_t^j = \theta_t^j - \alpha \frac{v_t}{\sqrt{S_t + \varepsilon}} g_t \tag{2.22}$$

where α is a learning rate hyperparameter, and ε is for numerical stability, set to e^{-8}.

2.2.5 Recurrent Neural Networks

The sequential nature of text encouraged the use of models that can process data sequentially. By constructing a sequential chain of feedforward networks, with connections between their hidden layers, inputs later in the chain can be processed in relation to earlier inputs. In this way, a Recurrent Neural Network RNN can take advantage of its internal memory to keep track of historic inputs as it computes the gradients for current inputs. The data input for an RNN can be in the form of $x_1 \ldots x_n$, for $\{t \in \mathbb{N} \mid 0 < t < n\}$, where x_t is a d-dimensional vector at step t. For text processing, x can be a vector representation for an input token. In an RNN, the input at time step t is directly influenced by the weights of the hidden states for inputs from previous time steps. Here, the hidden state h_t represents the "memory" of the sequence received at step t. For each time step t, an RNN performs the following operations:

$$h_t = f_1(W_{hh} h_{t-1} + W_{xh} x_t + b_a) \tag{2.23}$$

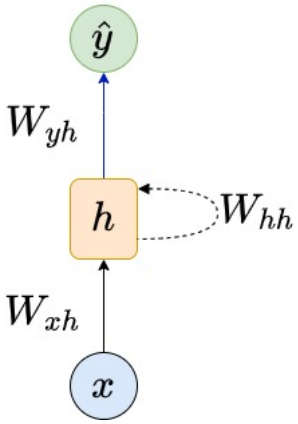


Fig. 2.8 Unrolled RNN. Image adapted from Olah (2015)

$$y_t = f_2(W_{yh}h_t + b_b) \tag{2.24}$$

which can be re-written as:

$$y_t = f_2(W_{yh}f_1(W_{hh}h_{t-1} + W_{xh}x_t + b_a) + b_b) \tag{2.25}$$

where f_1 and f_2 are activation functions, W is a weight matrix, h is the hidden vector, W_{hh} is the weight matrix between two hidden layers, W_{xh} is the weight between the current input state and the hidden layer, y_t is the output state, W_{yh} is the weight at the output state, and b_a and b_b are bias vectors added to the input and hidden layers respectively. The recursive calls in equation 2.25 allow the network to retain information from previous data inputs. Due to the sequential processing of inputs, the loss at time step $t = n$ includes the sum of losses for previous steps; $Loss = Loss(t_0, t_n)$. This means that when computing the gradient of the error, not only do we backpropagate through layers, but also through time. To further explain the gradient computations, in Figure 2.9, we unfold the RNN from Figure 2.8, over 5 time steps, $t = 0, 1, 2, 3, 4$.

If we denote the loss as C, as in equation 2.16, the gradient of the loss at step t with respect to any weight matrix W can be written as:

$$\frac{\partial \mathbf{C}}{\partial \mathbf{W}} = \sum_{t=0}^{T} \frac{\partial \mathbf{C}_t}{\partial \mathbf{W}} \tag{2.26}$$

Using the chain rule, we can compute the overall gradient for y_t from equation 2.25:

$$\frac{\partial \mathbf{C}}{\partial \mathbf{W}} = \sum_{t=0}^{T} \frac{\partial \mathbf{C}}{\partial \mathbf{y}_t} \frac{\partial \mathbf{y}_t}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_0} \frac{\partial \mathbf{h}_0}{\partial \mathbf{W}} \tag{2.27}$$

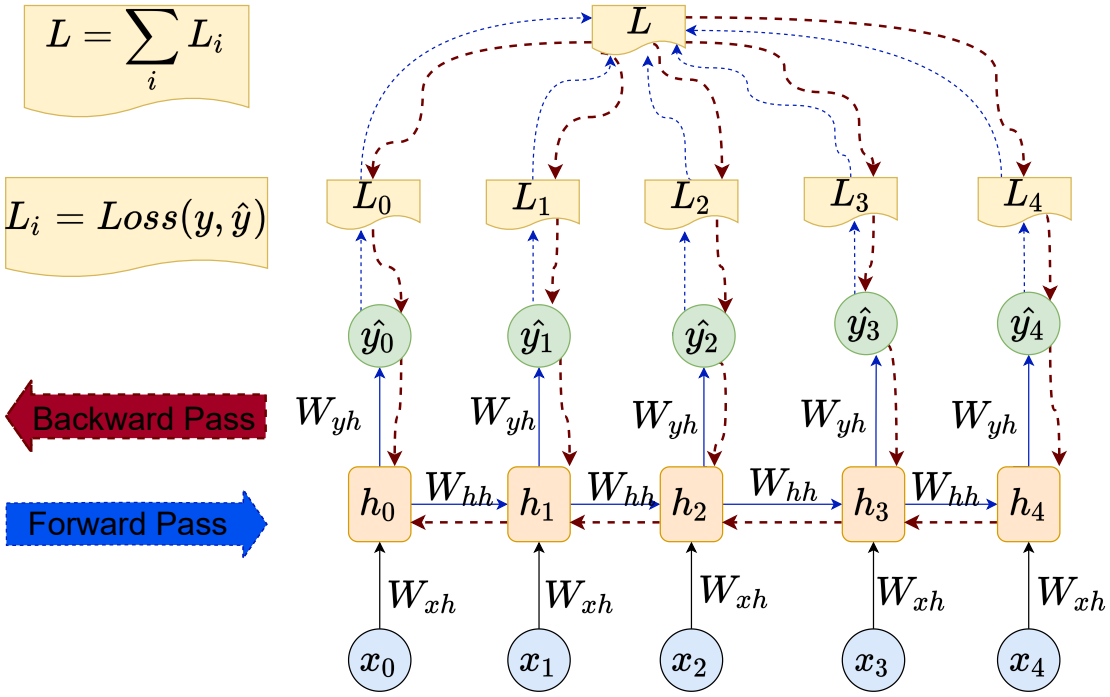


Fig. 2.9 Unfolded RNN. The forward pass is indicated by the blue arrows, whereas the red arrows indicate a backpropagation backward pass. Image adapted from Or (2020)

where $\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_0}$ is the derivative of the hidden state at time step t with respect to the hidden state of the initial input at time step 0. This derivative can be computed with a chain rule over the intermediate hidden states:

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} = \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} \frac{\partial \mathbf{h}_{t-1}}{\partial \mathbf{h}_{t-2}} \cdots \frac{\partial \mathbf{h}_1}{\partial \mathbf{h}_0}$$

Hence, (2.28)

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_0} = \prod_{i=0}^{t} \frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i-1}}$$

It is clear from equation 2.28 that as the number of time steps t increases, more gradients are multiplied. In the case that the initialized weights contain large values, i.e. > 1, multiplying their derivatives would lead to an exponential increase in the final gradient. This in turn would result in large updates to the network, hence leading to increasing gradient oscillations and not converging to an optimum point. Pascanu et al. (2013) showed that the risk of *exploding* gradients could be mitigated by clipping values that exceed a predefined threshold. Although this procedure helps RNNs overcome the challenges of large gradients, it does not address problems that arise from the multiplication of smaller gradients, < 1.

In the case of small enough weights < 1, as the number of time steps t $\rightarrow \infty$, the gradient multiplications can become numerically unstable, causing gradients to *vanish*.

To address the aforementioned gradient issues, mainly the vanishing gradient problem, Hochreiter and Schmidhuber (1997) proposed a modification to the RNN's architecture, known as the Long Short-Term Memory LSTM network. This modification enables the network to learn longer sequences by using mechanisms called "gates". The LSTM gates enable the network to selectively retain or overwrite information. There are three types of gate; forget, input and output. The forget gate f_t deletes existing information that it deems unimportant:

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f) \tag{2.29}$$

The input gate lets new information in, by applying an element-wise product on the outputs of the two fully connected layers $i_t \otimes \tilde{c}_t$, where:

$$\begin{aligned} i_t &= \sigma(W_i[h_{t-1}, x_t] + b_i) \\ \tilde{c}_t &= tanh(W_c[h_{t-1}, x_t] + b_c) \end{aligned} \tag{2.30}$$

The output gate controls what information at time step t is sent to the network as input in the following time step $t+1$:

$$\begin{aligned} o_t &= \sigma(W_o[h_{t-1}, x_t] + b_o) \\ h_t &= o_t \otimes tanh(c_t) \end{aligned} \tag{2.31}$$

Putting together the information from all the gates results in:

$$\begin{aligned} c_t &= f_t \otimes c_{t-1} + i_t \otimes \tilde{c}_t \\ h_t &= o_t \otimes tanh(c_t) \end{aligned} \tag{2.32}$$

where *sigma* is the sigmoid activation function, tanh is the tanh activation function, and $\otimes$ is an element-wise multiplication. By overcoming the issues presented by the vanishing and exploding gradients, the LSTM is able to update its weights with respect to earlier inputs of much longer sequences, as opposed to the vanilla RNN. This means that the LSTM is able to memorize longer sequences.

2.2.6 Attention

Although LSTMs outperform standard RNNs in capturing longer sequences, they still struggle with long-range dependencies. This can be a problem for NLP tasks with very long text sequences, such that the network becomes unable to retain information from earlier inputs. In an LSTM, information is compressed as it travels sequentially from earlier states to later

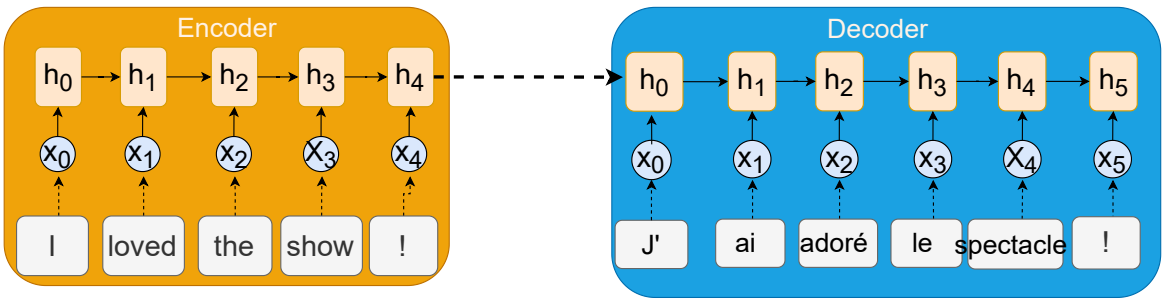


Fig. 2.10 English to French RNN without attention

ones. Hence, the longer the sequence, the more information has to be compressed into later hidden states. This process is problematic, especially when applied to an encoder-decoder architecture. In this setting, the encoder, e.g. an LSTM, compresses information from the full input sequence into its last hidden layer. The encoded information is then passed directly to the decoder's first hidden layer. For example, in machine translation, the encoder compresses the input information from the source language in its final hidden state. The encoded information is then passed as input to the encoder's first hidden state, which is then used to generate text for the target language. The context information compressed in the encoder's last hidden state h_t can be represented by the vector c:

$$c = g_1([h_1 \dots h_t]) \tag{2.33}$$

The decoder is then trained to predict the next word w_t in the target language given c and the previously predicted words $w_1 \dots w_{t-1}$:

$$p(w_t|w_t \dots w_{t-1}, c) = g_2(w_{t-1}, s_t, c) \tag{2.34}$$

where g_1 and g_2 are non-linear activation functions, and s_t is the hidden state of the decoder. The sequential flow of information in a standard RNN-based encoder-decoder model can be visualized in Figure 2.10. Here, the last hidden state of the encoder, $h4$, compresses the information from previous hidden states. This information is then passed on to the decoder in sequential order as shown in the figure.

With the attention mechanism, later hidden layers do not need to perform a full compression of information from previous hidden layers. Instead, later layers can dynamically "attend" to relevant information from previous layers. In an encoder-decoder architecture for machine translation, attention allows the decoder to attend to relevant features in the source sentence at each step during the output generation. At token i, given the vector embedding e.g. hidden state h_i, the attention computes a weight distribution on the input sequence, where

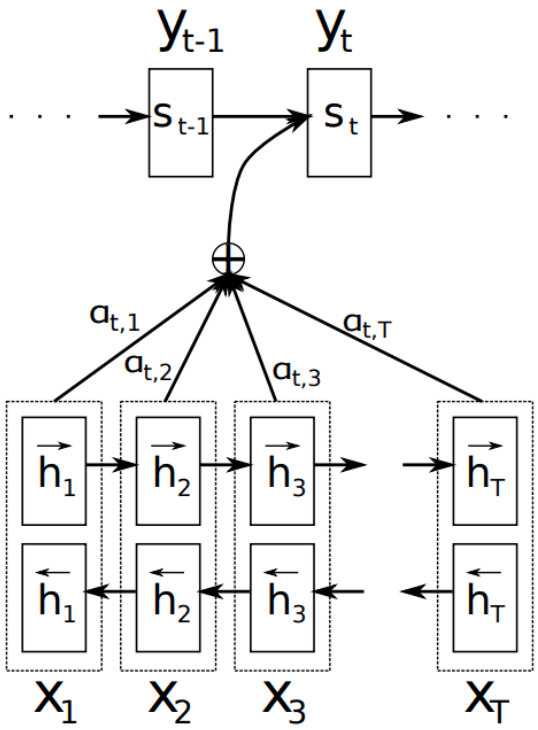


Fig. 2.11 Decoder attending to Encoder at time step t. Source: Bahdanau et al. (2014)

more relevant features are assigned higher values. As explained by Bahdanau et al. (2014), with attention, the conditional probability for the next word w_t is:

$$p(w_t|w_t \dots w_{t-1}, c_i) = g_2(w_{t-1}, s_t, c_i) \tag{2.35}$$

where the context vector c_i is the average of the previous hidden states weighted with the attention scores a_i:

$$\begin{aligned} c_i &= \sum_j a_i h_i \\ a_i &= softmax(score_a(h_i, s_i)) \end{aligned} \tag{2.36}$$

and $score_a$ is an attention alignment score. This is the essence of the attention mechanism, as it quantifies the amount of "Attention" set by the decoder on each of the encoder's hidden layers when generating its next output. Bahdanau et al. (2014) demonstrate how attention computes a context vector for each output time step, as shown in Figure 2.11. When at the decoder's hidden state S_t, the alignment scores are calculated between the previous hidden state S_{t-1} and each of the encoder's hidden states. The alignment scores for the encoder's hidden states are then aggregated into a single vector and then normalized through a softmax

function, as shown in equation 2.36. Next, the hidden states of the encoder are multiplied by the calculated alignment scores to produce a new attended context vector from which the current output time step Y_t can be decoded. Note that the attention mechanism can also be applied to architectures other than encoder-decoder, for instance, a sequence model can use attention to attend to information from its past states.

We now discuss the scoring function $score_a$ in equation 2.36. Multiple variants of $score_a$ have been proposed, among the most common ones are: multiplicative attention, self-attention, and key-value attention.

The multiplicative attention:

$$score_a(h_i, s_j) = h_i^\top \mathbf{W} s_j \tag{2.37}$$

where $h_i^\top$ is the hidden state i, W is a trainable parameter, and s_j is the encoder's current state.

Self-attention: With self-attention, networks can memorize longer sequences by attending to different parts within the input text. Unlike the multiplicative attention, the self-attention does not compute s_j. This type of attention is useful for tasks trained on only one input. In the sequence-to-sequence architecture as in machine translation, the decoder has input from the hidden states of the source language h_i, and the current state of the decoder s_j as in eq. 2.37. However, in a single input architecture like an LSTM, there will only be one input and as such it won't be possible to apply eq. 2.37. In this case, self-attention helps the network increase the size of its memory by allowing it to attend to different parts within the same input element:

$$score_a(\mathbf{h}_i) = \mathbf{v}^\top \tanh(\mathbf{W}\mathbf{h}_i) \tag{2.38}$$

where h_i is the network's hidden state, W and v are trainable parameters.

Key-Value scores: The self-attention in eq. 2.38, as described by Vaswani et al. (2017), can be represented by a mapping function that aligns with the basic concepts in information retrieval. A query and a set of key-value pairs are mapped to an output, where the query, keys, values, and output are all vectors. The query q is the hidden state for the current input token, the set of keys k is an index for all hidden states of the input sequence. This is similar to an information retrieval process, where the q is a search query, that is matched against a set of indexed keys k. The returned values v represent the matching scores between q and k. Considering that q and k are trainable parameters, they will fall in the same vector space.

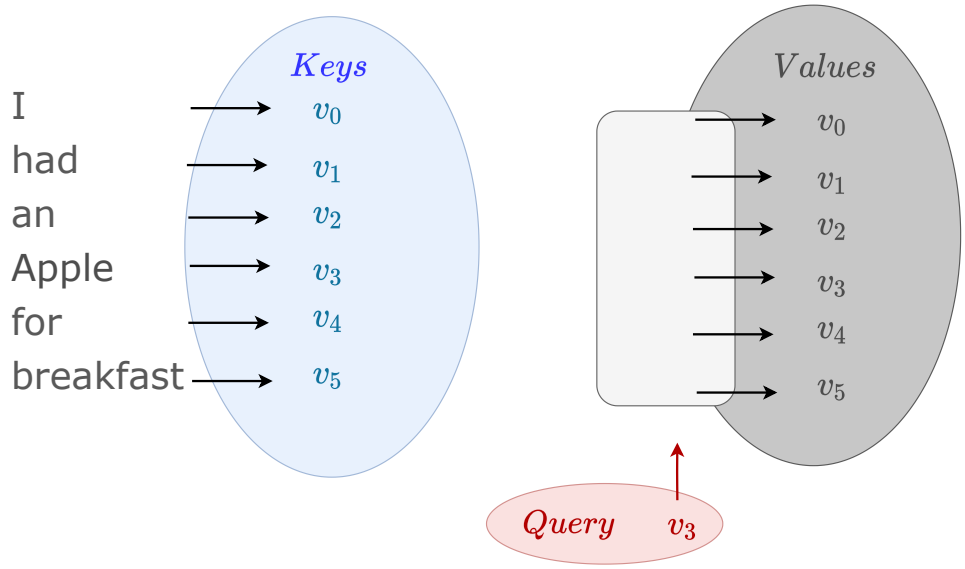


Fig. 2.12 Key-Value attention for query vector v_3

Hence, their dot product will return their proximity to each other. Here, the proximity score represents the relevancy between the matched vectors. In Figure 2.12, we show the search engine analogy with the key-value attention. In this example, the search index contains the attention keys, the query v_3 is the search query, and the values are the search results with relevance scores to the query. The sentence tokens "I", "had", "an", "Apple", "for", "breakfast", are represented by the index or key vectors $v_0, v_1, v_2, v_3, v_4, v_5$ in blue. For the query "Apple", the alignment scores are represented by the value vectors $v_0, v_1, v_2, v_3, v_4, v_5$ in gray. The calculated values represent each word of the input sentence in relation to the query word "Apple". By applying multiple instances of attention, with different training hyperparameters for the key-value pairs, multiple contextual vectors can be computed. This is shown by Vaswani et al. (2017) in their implementation of a non-recursive neural architecture, known as the Transformer.

2.2.7 Transformers

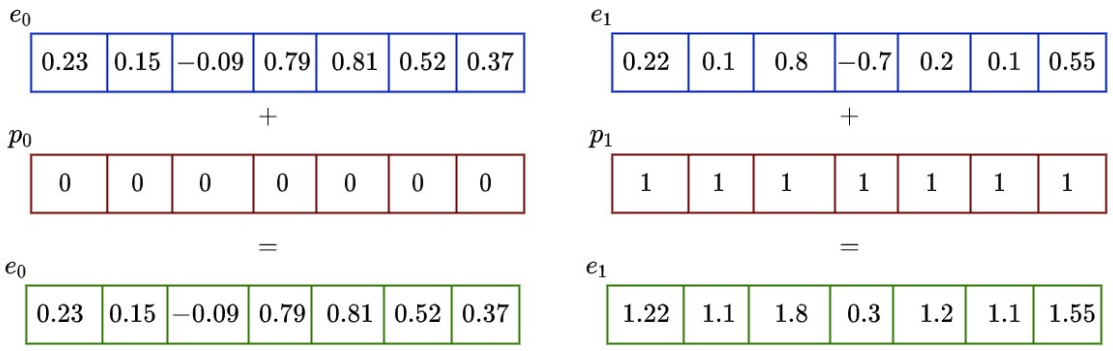
Prior to the release of the Transformer, it was a common practice to apply RNN-based architectures in natural language processing tasks. In 2017, Vaswani et al. (2017) proposed a less complex architecture to bypass critical issues that are present with RNN based architectures: a) Since information travels sequentially in an RNN, computations for the hidden state for token t_i rely on the encoded information in the hidden states for the previous tokens $0 \ldots i-1$. This means that previous hidden states must be computed before computing later hidden states. This sequential processing makes training slow and hard to parallelize. b) Because

computation is expensive, it becomes difficult to scale RNN-based architectures on large corpora. By replacing the recursive process with one that encodes the input sequence as a whole, Vaswani et al. (2017) were able to solve the aforementioned issues while achieving new state-of-the-art performances in machine translation tasks. The transformer is a non-sequential encoder-decoder model. Both the encoder and the decoder are essentially composed of multiple blocks of self-attention functions with a feed-forward network. The input to the Transformer is a vector of numeric values where each number is the index of the corresponding token in the vocabulary. In this way, when the input vector is passed to the Transformer's embedding layer, each token index is mapped to its corresponding vector representation. Positional embeddings are then added to the token embedding vectors to give the model a notion of word order in later processing steps.

Positional Embeddings: The lack of a recurrence mechanism eliminates the model's positional awareness. To account for sequential order, the position of each token can be added to its embedding vector. One might consider adding the numeric position index of the token in the sequence, i.e. $PE(0) = [i,\ldots,i]$, where $i \in (0,\ldots,n)$ for a sequence of n tokens:

$$\begin{aligned} e_t &= e_t + p_t \\ &where \\ p_t &= [t,\ldots,t] \end{aligned}$$

For example, as shown in Figure 2.13a, for the token at index 0, the positional embedding vector $p_0 = PE(0) = [0,\ldots,0]$, would be added to the embedding vector e_0, for the token at index 1, we add $p_1 = [1,\ldots,1]$, Figure 2.13b, and so on.



(a) Positional embedding at i=0

(b) Positional embedding at i=1

Fig. 2.13 Positional embedding by index value for indexes 0, and 1 respectively

However, this implementation can become problematic as the embedding information fades as i increases. For instance, as shown in Figure 2.14, for $i = 100$, entries of $e_{t_{100}}$ will mostly be close to 100, incurring loss of information from the original embedding.

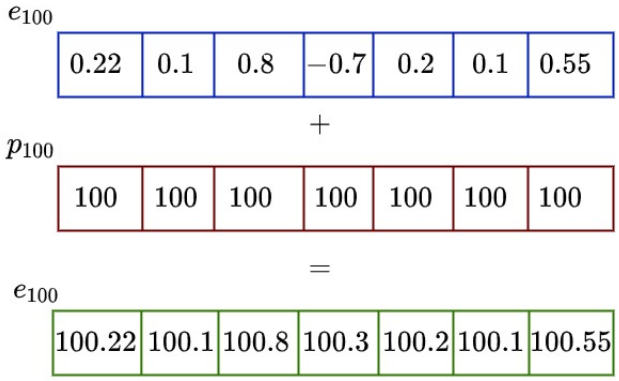


Fig. 2.14 Positional embedding at i=100

To solve this issue, one might normalize i by dividing by the number of tokens in the sequence n. In this way, the positional values are guaranteed to always be < 1. This however creates another problem; for sequences of different lengths, it could be possible to have the same positional embedding for different positions in the text. For example, for $i = 2$ and $n = 4$, the $PE(t_2) = 2/4 = 0.5$, this encoding will be the same for $i = 5$ and $n = 10$.

To better account for sequential order, in the implementation of the original Transformer, positional information is captured by computing for sin and cos wave frequencies, as follows:

$$PE(t,i) = \begin{cases} \sin(\frac{t}{10000^{2k/d}}), & \text{if } i = 2k \\ \cos(\frac{t}{10000^{2k/d}}), & \text{if } i = 2k+1 \end{cases} \tag{2.39}$$

where t is the position of the current word in the sequence, and i is the position index in the embedding vector of d dimensions. After the token order is encoded in the embedding vectors, the Transformer's attention functions, known as Attention Head, encode the entire input sequence simultaneously by encoding each embedding vector in relation to the other vectors.

Self-Attention: In a key-value self-attention, explained in section 2.2.6, for an input x, the alignment score is computed by matching the q vector against a set of keys k. This can be done by computing the dot product between q and every key k_i in the index k. For example, in a sequence of size 5, the attention for the word at index 0 is computed by taking the dot product between q_0 and each of k_0, k_1, k_2, k_3, and k_4. The result $[q_0k_0, q_0k_1, q_0k_2, q_0k_3, q_0k_4]$ is a vector of size 5 which includes the score between q_0 and each of the key vectors. To

stabilize gradients, Vaswani et al. (2017), divide the attention score by the square root of the dimension of the key vectors $\sqrt{d_k}$. The scores are then normalized to the range $[0,1]$ with the application of a $softmax$ function, refer to equation 2.8. Each value vector $[v_0 \dots v_4]$ in v is then multiplied by the corresponding normalized score, e.g. $score_0 \times v_0$. Here, the vectors with higher values are more relevant to the query q.

Multi-Head Attention: In a Transformer's attention block, instead of a single vector, each of q, k and v is represented with a matrix to allow more trainable parameters:

$$z = softmax(\frac{qk^\top}{\sqrt{d_k}})v \tag{2.40}$$

Where $k^\top$ is the transpose of k, and z represents the result of a single attention head. In a multi-head attention block, z is computed h times in parallel, where $z_i, i \in (0, \dots h)$. The result could be thought of as an ensemble of size h with different "subspace" representations. The z matrices are then concatenated, and multiplied by another trainable weight matrix for the output layer:

$$\begin{aligned} \text{MultiHead}(q,k,v) &= [\text{head}_1; \dots; \text{head}_h]W^O \\ \text{where head}_i &= \text{Attention}(qW_i^q, kW_i^k, vW_i^v) \end{aligned}$$

where W^O, W_i^q, W_i^k, and W_i^v are trainable parameter matrices. The multi-head attention block can be visualized in Figure 2.15. To pass information from the multi-head attention block, the output vectors from each attention head are first concatenated and then linearly transformed.

Given that the Transformer was proposed in the context of machine translation, it followed an encoder-decoder architecture. The encoder part takes as input the text in the source language, encodes it to a spacial representation, then passes it to the decoder that decodes it to the target language.

Encoder Block: The encoder block is shown in Figure 2.16. The data input, as described earlier, is a vector that maps each token in the sequence to its index in the vocabulary. Tokens are then mapped to their corresponding vector embeddings before encoding their positions. The result vectors are then passed to the first block of N stacked encoder blocks. The architecture is the same across all the encoder blocks, and the information between them is transferred sequentially. That is, the output of the first encoder block is the input of the second encoder block and so on. There are only two connected layers inside an encoder block with residual connections around them; the first is a multi-head attention layer, and the second is a feed-forward network. The residual connections are simply connections that

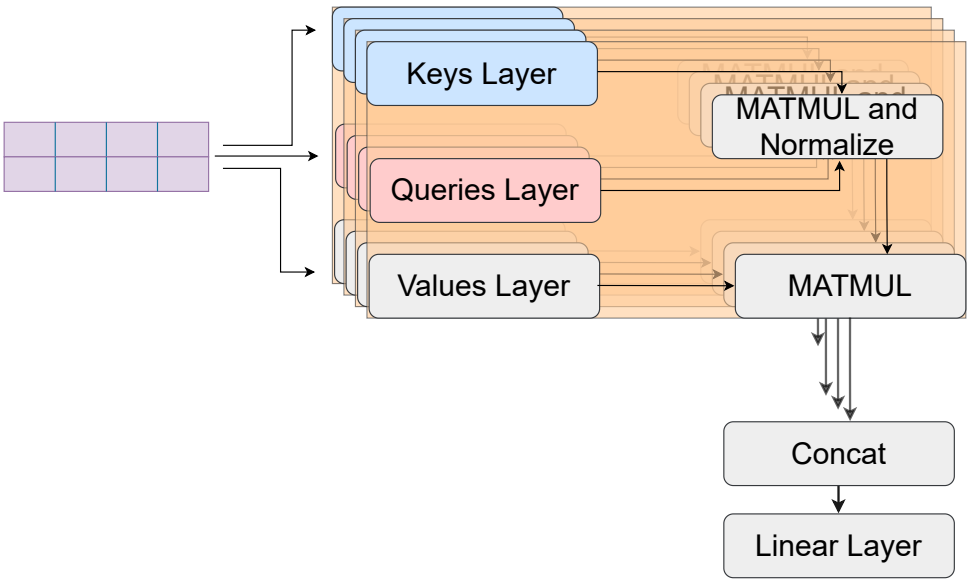


Fig. 2.15 Multi-Head Attention. MATMUL is short for matrix multiplication[1]

bypass non-linear transformations. As explained in section 2.2.5, multiple operations on gradients could cause vanishing or exploding values. The non-linear transformations, from section 2.2.1, can thus be skipped with residual connections. In an encoder block, residual connections can be seen around each layer to connect its input to its output. The inputs and outputs of a layer are added and passed through a layer normalization function, as shown in equation 2.41:

$$Y = Norm(x + layer(x)) \tag{2.41}$$

where Y is the normalized output, x is the layer's input, and $layer(x)$ is the layer's output. The normalization function as described by Ba et al. (2016), simply rescales the activations (outputs) of a layer:

$$\sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}\left(a_i^l - \mu^l\right)^2}$$

where (2.42)

$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l$$

where H is the number of features (hidden units) in a layer, and a_i is an activation unit.

[1] Image adapted from https://www.youtube.com/watch?v=tIvKXrEDMhk

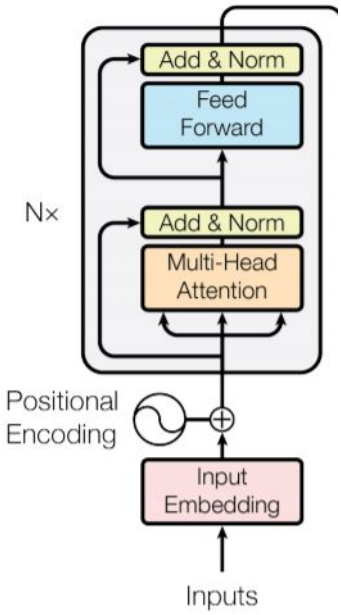


Fig. 2.16 Encoder Block. Source: Vaswani et al. (2017)

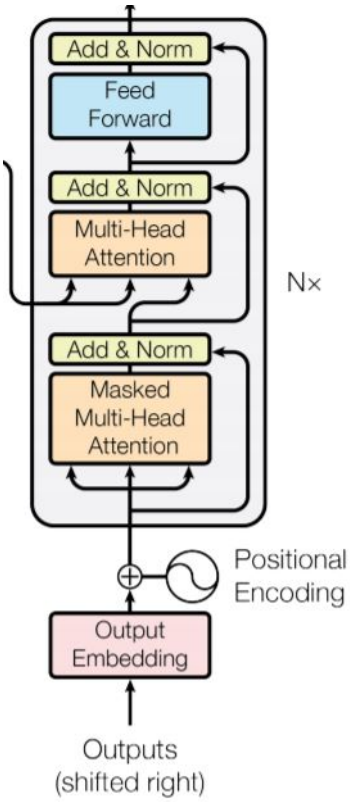


Fig. 2.17 Decoder Block. Source: Vaswani et al. (2017)

Decoder Block: The decoder block has a similar architecture to the encoder. It has two multi-head attention layers, and a feed-forward network. Similar to the encoder, each sub-layer is followed by layer normalization on its outputs and the outputs of its residual connections, refer to equation 2.41. The implementation of the multi-headed attention layers, however, is different to that of the encoder. The decoder is an autoregressive model; from earlier inputs, it predicts tokens in a sequence one at a time. It does so by predicting future tokens based on information from the encoder and its previous outputs. As seen in Figure 2.17, the decoder block receives two input connections: one from its output sequence, and one from the encoder. The first multi-headed attention layer in the decoder receives input from the target sequence. Since the decoder is autoregressive, it must not see future tokens as it generates the current token. To allow the decoder to only attend to previously generated tokens, positions of future tokens are masked by setting their values to $-inf$ before the softmax step in the self-attention calculation. Hence, once the softmax is calculated, the negative infinities get zeroed out, meaning zero attention scores for future tokens.

The second self-attention layer in the decoder works just like the multi-headed attention layer from the encoder. Its only difference is that it creates the Queries q matrix from the layer below it, and takes the Keys k and Values v matrices from the output of the encoder stack. The outputs k and v of the encoder are passed directly to the decoder's second multi-head attention layer as seen in Figure 2.18. Similar to the encoder, the final layer in the decoder is a feed-forward network with a normalization function connected to its residuals. Finally, a linear projection is applied to the output of the decoder stack, which is then passed as input to a softmax function that produces a probability distribution over the tokens in the vocabulary for every next prediction.

2.3 Evolution of Language Models

The year 2018 proved to be an exceptional one for the NLP community as research shifted rapidly from pretrained shallow embeddings to more complex pretrained language models adopted from the computer vision field. This is evident in developments such as Embeddings from Language Models (ELMo) (Peters et al., 2018), Universal Language Model Fine-tuning for Text Classification (ULMFIT) (Howard and Ruder, 2018a), Generative Pre-trained Transformers (GPT) (Radford et al., 2018), Bidirectional Encoder Representations from Transformers (BERT) (Devlin et al., 2018), and many more. In this section, we will discuss the evolution of language modeling, starting with context insensitive word vectors and then considering more advanced context aware architectures.

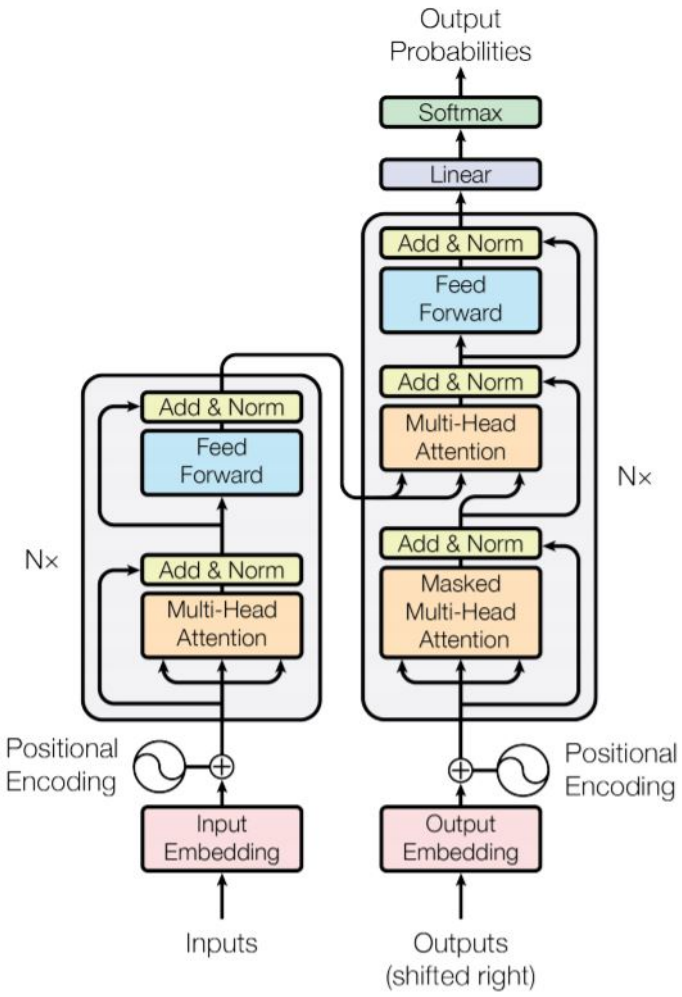


Fig. 2.18 Transformer: Encoder-Decoder Blocks. Source: Vaswani et al. (2017)

2.3.1 Pre-trained Word Embeddings

Pretrained word vectors opened room for optimism in the field of Natural Language Processing. Prior to this, the conventional approach to word vectors was to compute a word occurrence matrix that would result in sparse and long vectors, where the number of dimensions corresponds to the size of the vocabulary as in a document-term matrix, or the number of documents in the training data, as in a term-document matrix. The simplest form of a document-term matrix counts the occurrence of every vocabulary word in each document. For example, for the documents:

- Doc 1: text analytics is fun
- Doc 2: I like doing text analytics

	text	analytics	is	fun	I	like	doing	pizza	tastes	great
Doc 1	1	1	1	1	0	0	0	0	0	0
Doc 2	1	1	0	0	1	1	1	0	0	0
Doc 3	0	0	0	0	1	1	0	1	0	0
Doc 4	0	0	0	0	0	0	0	1	1	1

Table 2.2 Document-Term Matrix

- Doc 3: I like pizza
- Doc 4: pizza tastes great

The document-term matrix is shown in Table 2.2, where each row is a document vector, with an entry for every vocabulary token. The table thus represents a count-based relationship between any document and every vocabulary token. It can be clearly noted that the number of columns would increase as more vocabulary words are added. Also, since most vocabulary tokens may not appear in every document, many of the entries will have 0 values, creating a sparse matrix. Each row in Table 2.2 is a document vector that contains the count of every vocabulary token in it. In this way, documents can be represented by the vocabulary tokens. To create a representation for the tokens, we can transpose the document-term matrix, to form a term-document matrix, as in Table 2.3. By transposing the matrix, each row becomes a word vector represented by the documents in the collection.

Term Frequency-Inverse Document Frequency (TF-IDF)

There are various schemes for determining the value entries for a document/term matrix. Matrices in Tables 2.2 and 2.3 are based on a counting scheme. A more useful estimation approach is the Term Frequency-Inverse Document Frequency (TF-IDF). The TF-IDF reveals the importance of a word in a corpus, rather than just in a single document, as apposed to the simple count based methods. The Term Frequency (TF) component measures the frequency of a token in a particular document, shown in equation 2.43, whereas, the Inverse Document Frequency (IDF) measures the importance of tokens by weighting down more frequent terms and up weighting rarer ones, as shown in equation 2.44:

$$TF = \frac{\text{term frequency in document}}{\text{total words in document}} \tag{2.43}$$

$$IDF(t) = \log_2 \left(\frac{\text{total documents in corpus}}{\text{documents with term}} \right) \tag{2.44}$$

	Doc 1	Doc 2	Doc 3	Doc 4
text	1	1	0	0
analytics	1	1	0	0
is	1	0	0	0
fun	1	0	0	0
I	0	1	1	0
like	0	1	1	0
doing	0	1	0	0
pizza	0	0	1	1
tastes	0	0	0	1
great	0	0	0	1

Table 2.3 Term-Document Matrix

The resulting formula, in equation 2.45, is a multiplication of the TF and the IDF components. It reveals the significance of any token "t" for a document "d" by increasing as the frequency of "t" increases in "d", but also decreasing as the frequency of "t" increases in the overall corpus:

$$IDF(t) = TF.IDF \tag{2.45}$$

Since TF-IDF is 0 for tokens that do not appear in the respective documents, the sparsity issue of the count-based schemes (tables 2.2 and 2.3) is not addressed. To convert a term-document matrix into a dense matrix with fewer dimensions, a dimensionality reduction technique can be applied. One of the earliest approaches for dense word vectors was the Latent Semantic Analysis LSA (Deerwester et al., 1990). In LSA, Singular Value Decomposition SVD is applied to the term-document matrix to project the most important features to a matrix with reduced dimensions. In brief, SVD decomposes a matrix A, into three matrices $U\Sigma V^T$, where U and V^T are orthogonal matrices, and Σ is a diagonal matrix. In LSA, the first 300 dimensions are normally retrained to create dense vectors, known as LSA embeddings.

Word2Vec

In the year 2003, Bengio et al. (2003) showed that language models can be used to create vector representations for textual inputs. Later in 2011, Mikolov et al. (2011) showed that

neural networks can be used as language models. Two years later, Mikolov et al. (2013) proposed *Word2Vec*, a toolkit that offered neural network methods for training word vectors. These vectors are also known as *embeddings*. The authors made a public release of pre-trained word embeddings for downstream tasks. These are dense vector representations for word level tokens that were computed over large collections of text. Each vector has a single mapping to a vocabulary token. In this way, every token is represented by one vector. Compared to previous approaches, the pre-trained Word2Vec embeddings allowed machines to better understand text inputs, and thus improve computations for downstream tasks. For instance, the sparsity of a one-hot encoded vector limits the ability of capturing important information about a token. Furthermore, one-hot encoded vectors are hard to scale, as the larger the vocabulary grows, the more dimensions are added to each vector. On the other hand, the unsupervised nature of training language models enables the encapsulation of much more information by encoding data in every dimension. Unlike one-hot vectors where a single dimension represents a token with the value "1", in Word2Vec vectors, each dimension can express a distinct feature about the token. For example, for the word "queen", a dense vector could contain information about gender, royalty, and wealth spread over multiple dimensions.

Word2Vec vectors were created by training a 2-layer neural network classifier on a large corpus of text with the task of predicting the likelihood of a word appearing in a context. The goal of this setup was not to perform well on the training task, but instead to learn dense vector representations for the input tokens. Mikolov et al. (2013) presented two neural network architectures for training dense vectors, the Skip-gram model, and the Continuous Bag of Words model (CBOW). In Word2Vec, the input tokens are represented by one-hot encoded vectors. For a vocabulary of size of 10000 tokens,Word2Vec learns a mapping from one-hot encoded vectors of size 10000 to a dense vector with reduced number of dimensions d. This is done by multiplying the one-hot word vectors by the learned weights of the hidden layer, a matrix of size $dX10000$. This is an improvement over traditional word frequency methods, e.g. TF-IDF, that generate highly dimensional sparse vectors. The Skip-gram model is trained to predict whether any two input words are neighbors or not. This can be done by training the neural network to predict the likelihood of every word in the vocabulary being a surrounding word. Different to the Skip-gram, the CBOW model is tasked to predict if a word appears in a context of multiple words. Since the input in this case consists of multiple words, their vector representations are summed, creating a context input vector. The result vector becomes input to the neural network, which is trained with the objective of predicting if a given word is the center of the context.

GLOVE

In Word2Vec, training is essentially done to learn word co-occurrences from segments of text. In 2014, Pennington et al. (2014) suggested GLOVE, a simpler approach, where instead of training a shallow neural network for prediction tasks, word co-occurrences can be captured by measuring their frequencies in a single matrix. To achieve this, GLOVE starts with the matrix A of word-word co-occurrence. Here, entry A_{ij} measures the co-occurrence ratio between words i and j. This can be phrased as the probability of word j appearing in context i or more formally:

$$P_{ij} = P(j|i) = \frac{A_{ij}}{A_i} \tag{2.46}$$

where A_i is the number of words in context i. Hence, $A_i = \sum_k A_{ik}$. The frequences are then updated by training for the objective of least-squares:

$$J = \sum_{i,j=1}^{V} f(A_{ij})(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log A_{ij})^2 \tag{2.47}$$

where $w_i^T \tilde{w}_j$ is the dot product of the word vectors w_i and w_j, and b_i and b_j are the corresponding vector biases. $f(A_{ij})$ is a function that assigns lower probabilities to rare and frequent co-occurrences:

$$f(A_{ij}) = \begin{cases} (\frac{A_{ij}}{A_{max}})^{\alpha} & \text{if } A_{ij} < A_{max} \\ 1 & \text{otherwise} \end{cases} \tag{2.48}$$

where α is a hyperparameter set to $3/4$, and A_{max} is a threshold for a maximum co-occurrence, e.g. 100.

2.3.2 Deep Learning Language Models

ELMo

Recent advances in deep learning models are based on the transformer architecture, explained in section 2.2.7. Prior to that, sequential architectures like the LSTM, from section 2.2.5, had their share of dominance in NLP applications, but at the cost of computational complexity. For instance, in early 2018, ELMo was introduced as a contextual embedding model (Peters et al., 2018). The model consisted of two LSTM networks that pass over the input in opposite directions. This architecture is known as a bidirectional LSTM or biLSTM. Figure 2.19 clarifies how a BiLSTM architecture processes an input text from both directions. Input

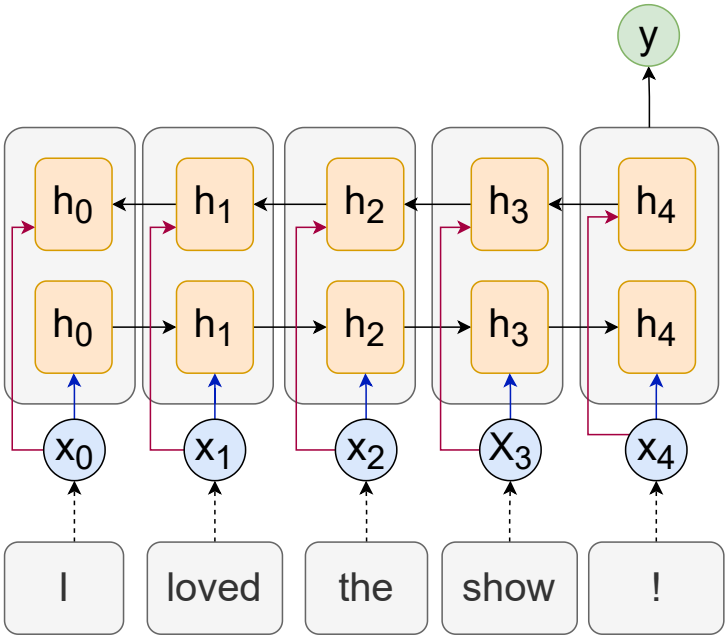


Fig. 2.19 BiLSTM processing the input tokens "I", "loved", "the", "show", "!"

tokens are represented by embedding vectors x_0, x_1, x_2, x_3, and x_4, and h1, h2, h3, and h4 are the hidden states for each LSTM.

Unlike, Word2Vec and GLOVE, where the vectors are context insensitive, ELMo can produce a different vector for the same word depending on its context. For instance, the word “Apple“ in the sentence “An apple a day keeps the doctor away” is expected to have a different representation when used in the context “I don’t own any Apple products”. The contextual embedding of ELMo vectors enabled state-of-the-art performances on multiple NLP tasks over previous approaches. Based on the BiLSTM architecture, we display an overview of ELMo in Figure 2.20.

ULMFit

In May 2018, Howard and Ruder (2018a), presented ULMFiT, short for “Universal Language Model Fine-tuning”, an approach for fine-tuning pre-trained language models on target tasks. Instead of using word vectors as input to classification models, the idea was to tune the weights of a pre-trained language model on the data of the target task. The suggested fine-tuning approach helps the network preserve low representations learned in the pre-training stage, e.g. grammar, while adapting to newly learned features from the target task. This was demonstrated by pre-training an LSTM-based model on 28,595 Wikipedia articles, and fine-tuning it on multiple text classification tasks. In addition to the pretraining stage, ULMFiT consists of two additional major steps, a) fine-tuning the language model

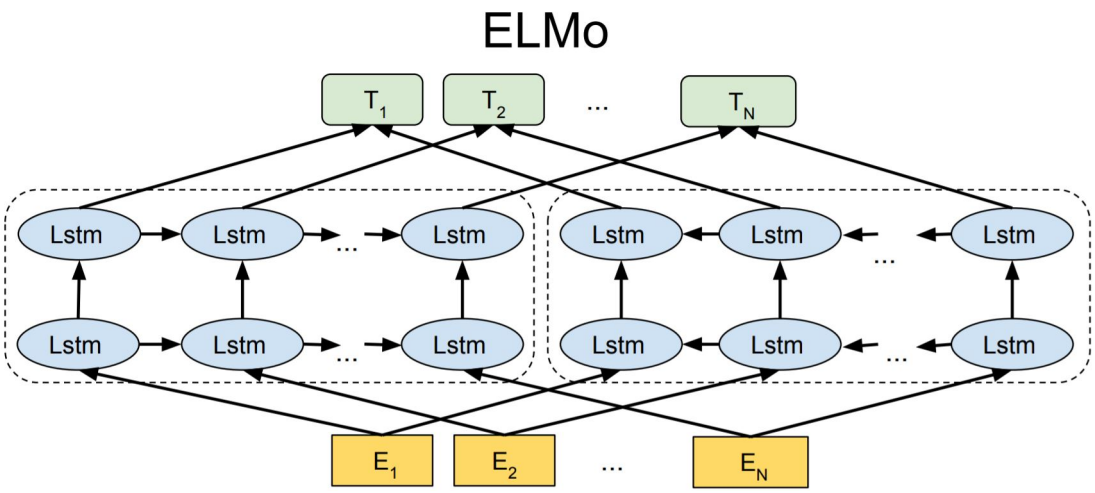


Fig. 2.20 Overview of ELMo processing text. Source: Devlin et al. (2018)

by applying discriminative fine-tuning and slanted triangular learning rate schedules to learn target task features, and b) fine-tuning the classifier on the target task using gradual unfreezing. These are now outlined.

Fine-Tuning the language model: Two crucial steps are needed to fine-tune the language model; a) Discriminative fine-tuning: Knowing that different layers in the network capture different types of information (Yosinski et al., 2014), they should be fine-tuned to different degrees (Howard and Ruder, 2018a). This can be done by adjusting the learning rate for each layer separately to enable their fine-tuning at different degrees. b) Slanted triangular learning rates (STLR): To optimize a fine-tuning procedure for quicker convergence, the learning rates can be first increased linearly, then gradually decayed. This approach is a variation of the triangular learning rates proposed by Smith (2017), with short increases in the learning rates and longer decay periods. The intuition here is that a higher learning rate allows the model to converge quicker, yet can increase oscillations around a minima point, refer to section 2.2.4. To overcome this shortcoming, in STLR once the learning rate is increased, it gradually starts decaying over a long period measured by the number of iterations. This is presented in the triangular shape in Figure 2.21.

Fine-tuning the target task classifier (gradual unfreezing): Once the language model has been fine-tuned, it is then augmented by adding additional layers for the classification task. These are the only layers that must be trained from scratch. This is because if all layers are fine-tuned at once, the layers of the language model will be at risk of forgetting what had been learned in previous trainings. To overcome this, the model is gradually unfrozen starting from the last layers while blocking the other (lower) layers. This is done by first unfreezing the last layer and fine-tuning it during the first epoch, then for every next epoch the next

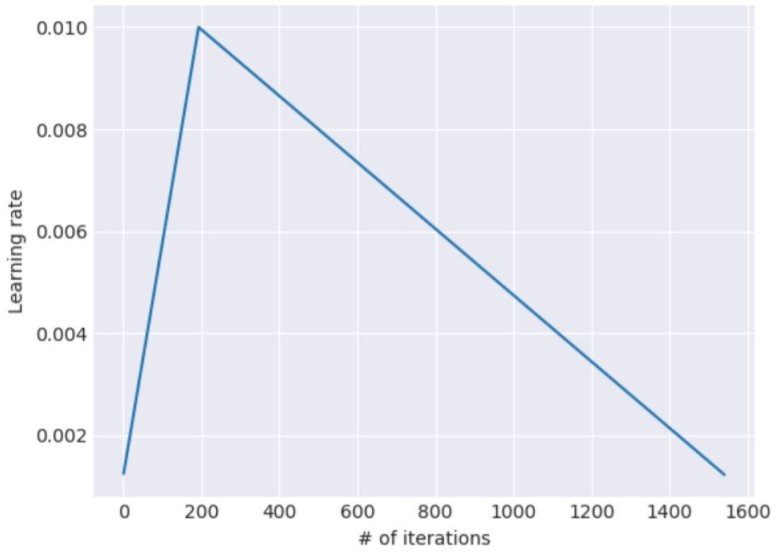


Fig. 2.21 The slanted triangular learning rate schedule in ULMFiT. Source (Howard and Ruder, 2018a)

lower layer becomes trainable (unfrozen) as training proceeds. This process continues until the model converges at the last iteration.

Bidirectional Encoder Representations from Transformers (BERT)

In late 2018, BERT was introduced to overcome the complexity issues of LSTM-based models and enable easier fine-tuning. The model takes as input WordPiece tokens, see section 2.1.2. Instead of applying a sequential architecture (like LSTM), BERT uses the encoder part of the transformer network from section 2.2.7. In ELMo, the representation of a word is composed by concatenating two models, one reads the input from left-to-right, while the other reads it from right-to-left. The use of two models was necessary because having just one model that reads the input from both sides might indirectly allow it access to the target words (its predictions). In overcoming this limitation, instead of using two models like ELMo, BERT masks the target word using a special masking token, "[MASK]". To widen its usability in downstream applications, BERT is pre-trained on two different tasks:

- Task 1, "Masked Language Model (MLM)", also known as "Cloze procedure": Given an input sequence, a random token is masked. This is a modification to the traditional language modeling objective, see section 2.1.3, in which the model is tasked to predict the masked token given its context as input. For example: the input "the boy ate the pizza" becomes "the boy [MASK] the pizza" by replacing the word "ate" with [MASK]. In the original implementation, only 80% of the time, a random word is

replaced with [MASK]. To enable the application of BERT to tasks where the input does not contain a [MASK] token, e.g. sentence classification, in the remaining 20% of the times, masking is not applied. Instead, 10% of the time, a random word is replaced with another random word, e.g. transforming the input of the previous example to "the boy jumped the pizza", and the remaining 10% of the time, the input remains unchanged.

- Task 2, Next Sentence Prediction: In order to broaden the NLP applications of BERT, relationships between sentences are accounted for in the language model. This is done by joining two sentences X and Y and predicting whether Y in fact follows X. This enables an easy application of BERT to tasks that take multiple sentences as input. For example, in textual entailment tasks e.g. Dagan et al. (2005), given two sentences as input (a hypothesis and a text), the model is trained to predict whether the text contradicts, entails, or is neutral to the hypothesis. Note that for this task, a [SEP] token is included between the two sentences, changing the input format to X [SEP] Y. The [SEP] token allows BERT to distinguish between X and Y by treating them as two separate segments of the input. Training BERT on this task allows easy fine-tuning on multi-input target tasks.

Concerning the input representation in BERT, for text classification tasks, a single vector representation for the input sequence must be passed to the classification head. However, each input token in BERT's final encoder block is mapped to a vector representation. To solve this in BERT, a [CLS] token is added to the beginning of each input sequence. In this way, when training the model to any task, the [CLS] token captures the representation of the full sequence input that is then passed to the classification head. Furthermore, a [SEP] token is required at the end of each sequence to indicate its completion. The use of [SEP] becomes more apparent later in this section.

Since BERT adopts a non-sequential architecture, it expects input vectors of a fixed length. For example, a BERT model that processes up to 768 tokens in an input, will consist of 768 dimensional hidden state vectors. Thus, the input is required to match the number of dimensions, that is 768 for this example. However, such a requirement would make it challenging to process textual data, as text can vary in length. Fortunately, vectors with tokens less than the size of the model's hidden states, can be extended by adding [PAD] tokens to match the required length. For example, assume for a BERT-based classifier, the embedding layer takes input of 10 dimensions. With WordPiece, the input sentence "best pizza in town" becomes [best, pizza, in, town]. For the purpose of classification, we add the [CLS] token to the front of the tokens. We also add the [SEP] token at the end of the sequence, [[CLS], best,

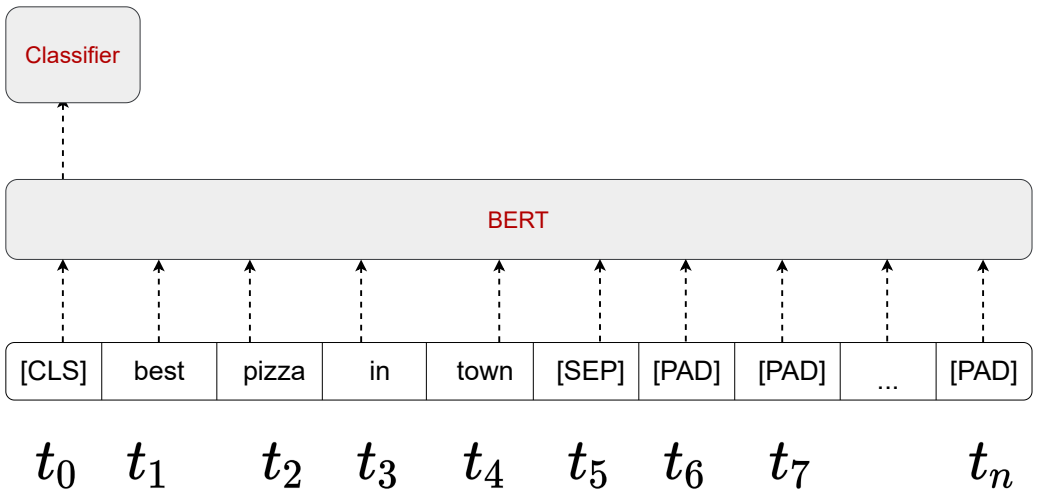


Fig. 2.22 BERT classifier: Single sentence input

pizza, in, town, [SEP]]. This sequence vector is composed of 6 tokens, but the model in our example takes inputs of length 10. Therefore, we pad the remaining indices with the token [PAD], casting the final input to [[CLS], best, pizza, in, town, [SEP], [PAD], [PAD], [PAD], [PAD]]. An input of n dimensions can be visualized in Figure 2.22.

To negate the effect of the [PAD] token on the final output, the attention mechanism is blocked from computing "attention scores" over the [PAD] indices. This can be done by zeroing out the values of the [PAD] tokens. In Huggingface's[2] implementation of BERT, an attention mask is used to block the padding tokens from the attention calculations. The attention mask is a vector that has 0 values for the indices of the [PAD] tokens, and 1s everywhere else. For the input [[CLS], best, pizza, in, town, [SEP], [PAD], [PAD], [PAD], [PAD]], the attention mask is $[1,1,1,1,1,1,0,0,0,0]$. By multiplying the weights by the attention mask, the values corresponding to the [PAD] indices will be 0 and remain unchanged for the other indices. In this way, the attention for each [PAD] token will amount to 0, and have no impact on the generated embeddings. Hence, the embedding vector of the [CLS] token will not be affected by the [PAD] tokens.

Similar to any transformer-based architecture, BERT relies on a positional embedding layer to achieve sequential ordering awareness (section 2.2.7). Not only does this improve text classification for single sentence tasks, but it also makes it possible to classify data in different formats, e.g. pairing sentences. For instance, to examine the relationship between sentences A and B, BERT's input will contain tokens of both sentences separated by [SEP]. This is useful for fine-tuning BERT on tasks that take more than one input. For example, to study the semantic similarity between two sentences, see section 2.5.1, the input can be

[2]https://huggingface.co/transformers

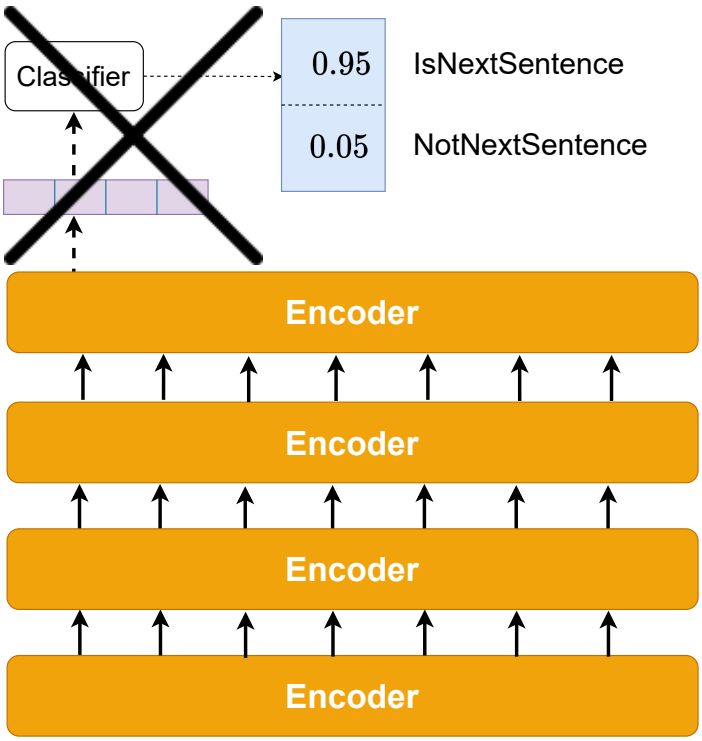


Fig. 2.23 BERT: Prepare BERT for fine-tuning by removing the pre-trained classification head

in the form [[CLS], [sentence A tokens], [SEP], [sentence B tokens], [SEP], [PAD]], and a binary output for whether sentences A and B are semantically similar or not.

Compared to LSTMs, the lower computational complexity of transformer-based architectures enabled the pre-training of BERT to scale over larger datasets such as BookCorpus (800M words) (Zhu et al., 2015), and English Wikipedia (2,500M words). Furthermore, task-specific customizations are limited in comparison to previous models like ELMo and ULMFiT. Instead of having to build a custom classifier as with ELMo embeddings, or undergo the fine-tuning steps in ULMFiT, fine-tuning BERT requires adding only a few parameters, e.g. a single layer classifier (classification head). Furthermore, BERT requires only few-shot fine-tuning, i.e. few training iterations; in the original paper, the authors fine-tuned BERT over all GLUE tasks for only 3 epochs. The replacement of the pre-training classification head for task-specific fine-tuning is visualized in Figures 2.23 and 2.24. Note that the GLUE benchmark, explained in section 2.5.1, contains different text classification tasks.

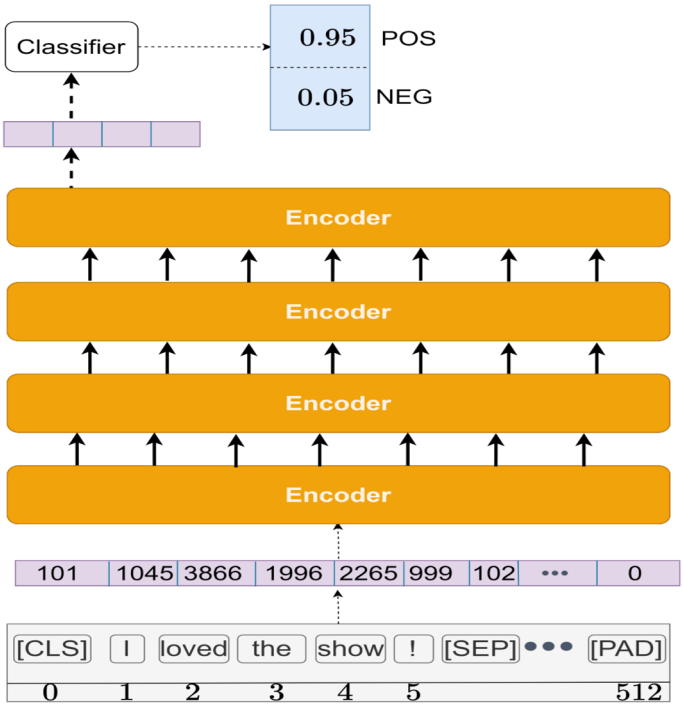


Fig. 2.24 BERT: Add a classification head for the binary sentiment classification task

Generative Pre-trained Transformer (GPT)

In 2018, OpenAI released a transformer language model based on the decoder blocks, GPT (Radford et al., 2018). GPT applies the Byte-pair subword tokenization scheme, see section 2.1.2. Unlike BERT, GPT was trained to predict the next word in a sentence. A key difference from BERT, is that in GPT, as a decoder, all future tokens are masked. Hence, GPT was trained using traditional language modeling to predict the next word in a sequence. In language modeling, the objective is to estimate the probability of a next token in a sequence conditioned on the context tokens (Bengio et al., 2003). GPT was introduced to improve natural language understanding through generative pre-training. This means a language model is initially trained to learn a distribution probability over the training data by generating instances from that distribution, see section 2.1.3. For an input of n tokens, $t = (t_1, \ldots, t_n)$, the language modeling loss function can be defined as per equation 2.1:

$$L_{\text{LM}} = -\sum_{i} \log p(t_i \mid t_{i-k}, \ldots, t_{i-1}) \tag{2.49}$$

The pretrained model can then be fine-tuned on labeled data for application in downstream tasks including text and sentiment classification, and question answering. Similar to fine-tuning BERT, a classification layer is added to predict a distribution over class labels:

$$L_{cls} = \sum_{(\mathbf{x},y)} \log P(y \mid x_1, \ldots, x_n) \tag{2.50}$$

where x is the input sequence, and y is a target label. Instead of only fine-tuning GPT for equation 2.50, the authors implemented an auxiliary learning objective by incorporating the language modeling loss from equation 2.49. The final fine-tuning loss function became:

$$L = L_{cls} + \lambda L_{LM} \tag{2.51}$$

where λ controls the contribution of L_{LM}. To maximize L_{LM}, the model estimates a probability by passing over the input tokens, and to maximize L_{CLS} the model learns the joint probability $P(y|x)$ between the full input sequence x and the label y. The authors claimed that adding L_{LM} to the loss function improves generalization and speeds up convergence. The language knowledge gained from the pretraining step meant that it was enough to only fine-tune GPT for a few epochs on downstream tasks, what is known as few-shot learning. Overall, GPT improved over specifically trained supervised state-of-the-art models in 9 out of 12 tasks, e.g. question answering, semantic similarity and text classification. Furthermore, GPT performed reasonably well in zero-shot settings where model weights are not updated. To evaluate zero-shot performance, the authors implemented heuristic settings. For example, in sentiment analysis, the token "very" is appended to each input sample, while restricting the model's output distribution to only the words positive and negative. In this way, the word that is assigned the highest probability is the predicted sentiment. This training procedure did not give GPT the edge over other state-of-the-art models like BERT on classification tasks, possibly due to BERT taking advantage of a bidirectional architecture. However, this did not stop OpenAI's GPT from prevailing in other departments. As it turns out, compared to BERT, GPT is able to generate text sequences of higher quality (Wang and Cho, 2019). The success of GPT raised the questions as to whether an increase in the number of learning parameters and pre-training data would lead to even better pretrained models.

Later, OpenAI released two updated versions of GPT, namely GPT-2 (Radford et al., 2019) and GPT-3 (Brown et al., 2020). Each updated version introduced a few architectural modifications to its predecessor, e.g. layer normalization and activations. The main changes each update brought were in the increased number of learning parameters and training data. The original GPT consisted of 12-layers with 768 dimensional states, reaching a total of 117 million learning parameters. The release of GPT-2 included 4 new models, each with a different layer count. The smallest GPT-2 is equivalent to the original GPT with 12 layers and 117 million learning parameters, the second smallest GPT-2 contained 345 million parameters from 24 layers with 1024 dimensional states, and the third model contained

36 layers with 1280 dimensional states, making a total of 762 million parameters. The largest GPT-2 model was built with 48 layers with 1600 dimensional states, reaching a total of 1542 million parameters. All variations of GPT-2 were trained on WebText which contains over 8 million documents. This is a much larger dataset compared to BookCorpus that the original GPT was trained on. For comparison, BookCorpus contains text from approximately 7000 unpublished books covering a wide range of genres. In a zero-shot learning setting, without fine-tuning, GPT-2 can benefit from meta-learning with its ability to adapt to different tasks by conditioning on the task to be performed. That is, instead of $P(\text{output}|\text{input})$, the condition can be $P(\text{output}|\text{input}, \text{task})$. Such conditioning expects the model to produce a different output depending on the task, even for the same input. For example, $P(\text{English translation}|\text{English text} <> \text{French text}, \text{translate English to French})$ should translate the input text to French. On the other hand, if the input was $P(\text{sentiment label}|$ text to classify, predict sentiment), the output is expected to be a sentiment prediction. Although its architecture was very similar to its predecessor, GPT-2 presented significant progress in language generation. The main contribution of GPT-2 was its ability to scale up to 1.5 Billion training parameters. However, even for such a large model, GPT-2 would still underfit the WebText dataset in a perplexity evaluation, see section 2.5.2. Later, GPT-3 was released to address the limitations of GPT-2. GPT-3 included up to 96 layers that scaled to 175 Billion learning parameters and was trained on 5 different datasets; WebText, Books1, Books2, Common Crawl, and the English-language Wikipedia. GPT-3 raised the bar for language modeling by displaying strong performance on many NLP tasks and benchmarks in the zero-shot, one-shot, and few-shot settings. There are no gradient updates in all these scenarios. In a zero-shot setting, the input includes a text prompt and the task description. In a one-shot setting, a demonstration example is included as part of the input. For instance, for a zero-shot setting, an example of an input would be "What is the French translation of cheese". In a one-shot setting, a single example is passed with the input, "translate English to French: cat -> chat, cheese -> ". Finally, in a few-shot setting, multiple examples would be included in the input, e.g. "Translate English to French: cat -> chat, bike -> vélo, kitchen -> cuisine, cheese -> ".

2.4 Vector Representations

In the previous section, we covered the evolution of language models in transfer learning. Older models like Word2Vec and GLOVE are only capable of creating numerical representations for a single token input, while more recent models, such as BERT and GPT (section 2.3.2), can represent inputs from phrases up to sentences. This section covers two common

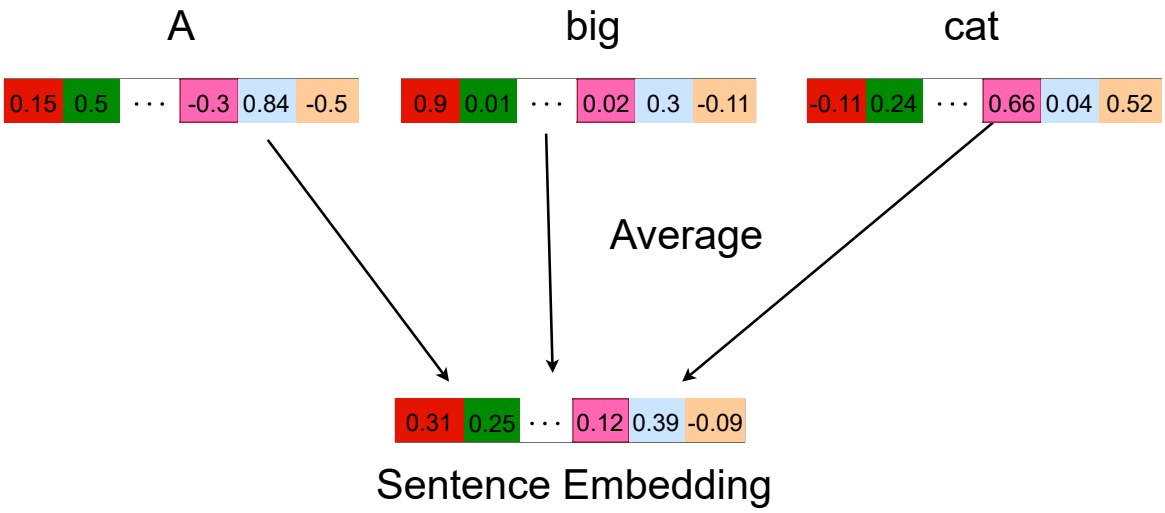


Fig. 2.25 Caption

approaches in creating sentence embeddings: a) the application of mathematical operations on the word model outputs, and b) the application of more advanced models that can encode a multi-word input.

2.4.1 Mathematical Operations on Word Model Output

Pre-trained word embedding models such as Word2Vec and GLOVE can only generate word level embeddings, and therefore cannot represent a sentence in a single vector. Instead, the input sentence would be represented by a nested list of vectors, where each vector corresponds to a token or word. In order to generate semantic embeddings of a sentence that can be used in downstream tasks, longer pieces of text need to be encoded into a single vector. For example, computing the similarity between two sentences requires finding the distance or angle between their representing vectors. This would be a complex task if each sentence were to be represented by a varying number of vectors. To solve this issue, once distributed representations of words are created, they can be combined using mathematical vector operations. A simple, yet commonly applied approach for creating a sentence embedding, is to average the embeddings of each token in the sentence. The averaged vector can then be used as a sentence embedding. For the tokens $t_1, t_2, \cdots, t_n$, represented by the vectors $v_{t_1}, v_{t_2}, .., v_{t_n}$, the sentence vector, v_s is:

$$v_s := \frac{1}{n} \sum_{i=1}^{n} v_{t_i} \tag{2.52}$$

A visualization of word averaging for the words "A big cat" is shown in Figure 2.25. Simple averaging would consider equal participation for each element in the input. This would give an equal weight for every word in the final representation, regardless of its importance. Word vectors can be assigned participation weights by multiplying each vector by its inverse document frequency (IDF), from equation 2.44. The final equation can be written as:

$$v_s := \frac{1}{n} \sum_{i=1}^{n} IDF_{t_i} v_{t_i} \tag{2.53}$$

In 2017, Arora et al. (2017) presented a stronger baseline for averaging word vectors in the paper "A Simple but Tough-to-Beat Baseline for Sentence Embeddings". In this approach, sentence vectors are first computed by taking the weighted average of word vectors. The projection of the sentence vectors to their first principal components are then removed. These projections are computed by either applying a Principal Component Analysis (PCA) or Singular Value Decomposition (SVD). In early 2018, Rücklé et al. (2018) introduced another averaging approach that computes a variation of the mean, known as the power mean (Hardy et al., 1952), defined as:

$$\left(\frac{x_1^p \ \ldots x_n^p}{n} \right)^{\frac{1}{p}} \tag{2.54}$$

Where x is a sequence of numbers, and p configures the type of mean. For $p = 1$ the arithmetic mean is retrieved, whereas for $p = 0$ it is the geometric mean and $p = -1$ for the harmonic mean. Once the means are computed for different values of p, the resulting vectors are concatenated to create a generalized sentence vector that captures representational power from word vectors, as opposed to previous approaches. Despite their simplicity, vector averaging approaches can still present challenges. For instance, by not considering the position of each token in the averaged vector, the word order is ignored. This means that although shuffling the words "A big cat", from the example in Figure 2.25, would cause a change in its meaning, the final sentence embedding will remain the same.

2.4.2 Beyond Simple Averaging to Sentence Embedding Models

In more recent approaches, sentence embeddings can be learned by training language models on textual data. For instance, an attention-based transformer model like BERT, can output a single vector representation for any input sequence (section 2.3.2). Since the [CLS] token encodes information from the other input tokens, its output vector can be used to represent the

entire input sequence. In this section, we will discuss other approaches that target learning sentence representations.

Skip-thought Vectors

Skip-thought vectors, proposed by Kiros et al. (2015) in 2015, sparked an emergence of research on learning sentence embeddings. This model behaves like a Skip-gram architecture from section 2.3.1, but instead of predicting neighboring words, the model is trained to predict surrounding sentences. The implementation is based on a recurrent neural network with an encoder-decoder architecture. Given an ordered list of sentences (s_{i-1}, s_i, s_{i+1}), sentence s_i is encoded to predict the previous and next sentences s_{i-1} and s_{i+1}. This implementation results in mapping sentences that are syntactically and semantically alike to similar vector representations. An interesting insight in the Skip-thought vectors paper is the handling of out-of-vocabulary words. Words that are not seen at training time are handled by learning a linear transformation between the RNN embedding space and an external word embedding model.

Quick-thoughts

In early 2018, Logeswaran and Lee (2018) proposed Quick-thoughts as an extension to Skip-thought vectors, where the decoder is replaced by a classifier that predicts a context sentence given a list of candidates. By treating the training data as a list of candidate sentences, the model is trained to predict which of these sentences precede or follow a given sentence input. This approach reformulates the sentence embedding problem into a classification task and serves as an improvement to the encoder-decoder architecture by reducing the complexity of the overall model. This results in reduced training time, making the model easier to scale over larger datasets. Furthermore, replacing the decoder with a classifier allows the model to ignore noise and irrelevant features when computing semantic vector representations. This is due to the fact that the training objective is to maximize the probability of identifying the correct context sentence instead of focusing on reconstructing one.

InferSent

Although training with unlabeled data is generally referred to as an unsupervised approach, in language modeling, models are still trained to minimize a loss function as in supervised training. The only difference with unlabeled data is that there are no hand-crafted labels. Nonetheless, due to the nonexistence of a fixed set of labels, we will refer to training language models on unlabeled data as an unsupervised setting. Previously mentioned approaches,

such as Skip-thoughts and Quick-thoughts, are unsupervised as their training was done on unlabeled data. It was previously assumed that unsupervised approaches produced better embeddings than supervised techniques. This assumption however was overturned in recent research. In 2017, Conneau et al. (2017) proposed a supervised embedding model called InferSent. The authors claimed that InferSent outperformed skip-thoughts on a variety of downstream tasks. This indicates that the embeddings of InferSent are of better quality than those generated by Skip-thought vectors. This model was trained on the Stanford Natural Language Inference (SNLI) dataset that consists of 570k sentence pairs where each is labeled with one of the three categories: contradiction, entailment and neutral. Both input sentences are first passed to a shared encoder that adopts a bidirectional LSTM architecture. The encoder then generates vectors u and v, representing each of the sentences respectively. The generated vectors then undergo three vector operations that are concatenation, element-wise product and absolute element-wise difference. The resulting vector is then fed to a three-class classifier that is concluded with a softmax layer.

Universal Sentence Encoder (USE)

In early 2018, Cer et al. (2018) concluded that transfer learning with sentence embeddings tends to perform better than that with word level embeddings. The authors proposed a model called Universal Sentence Encoder (USE) that generates a 512-dimensional vector given any input text, irrespective of its length. Similar to InferSent, USE employs an encoder that generates a sentence vector that is then fed to downstream tasks. One key difference, however, is that USE is trained on both supervised and unsupervised tasks. Like InferSent, USE's supervised training is done on the SNLI dataset. In order to produce a general-purpose model, additional unsupervised training was done, such as predicting responses from conversational data and neighboring sentence prediction on Wikipedia and web news articles. Its ability to generate contextual embeddings, made USE our choice of model for the experiments in chapter 4. Two pretrained variants of the USE are publicly available on TF-Hub[3]. One is based on a transformer architecture with attention mechanism that targets higher accuracy at the expense of model complexity and training time. The other model is based on a less complex Deep Averaging Network (DAN) architecture that is claimed to produce efficient inference results but at a lower accuracy. This comes with the advantage of linear computational time for input sequences.

[3]https://tfhub.dev/google/universal-sentence-encoder/

2.5 Evaluation

Machine learning models are normally evaluated by measuring their performance on test sets of downstream tasks. In the next sections, we will describe commonly used benchmark datasets and evaluation metrics.

2.5.1 Benchmark Datasets

There has always been room for innovation in deep learning, whether in building new architectures or in introducing new training methods. Benchmarking datasets are crucial for the research community to track progress. Moreover, for users from outside the research community, benchmarks provide insights into the field while showing the progression of models. As models become more powerful, there will be a need for improved and more difficult datasets. Models that seemed to beat human performance on certain benchmarks, can fail on simple real world challenges (Kiela et al., 2021). It has been shown that in benchmarks like the Stanford Natural Language Inference (SNLI), annotations were influenced by heuristics. This gave the trained model clues to the correct predictions while being irrelevant to the task itself. Hence, in the case of the SNLI dataset, the model is likely to learn the heuristics in the labeling strategy instead of the patterns in the training data. This makes the performance of the model questionable, as it is likely to be inflated due to correct predictions made on the basis of the labeling hypothesis (Gururangan et al., 2018). Furthermore, recent advancement in NLP has witnessed the ability of models to perform on a diversity of tasks, creating a need for benchmarks to include multi-task and multi-domain datasets. More recent benchmarks such as GLUE (Wang et al., 2018a) and SuperGLUE (Wang et al., 2019) have been widely used by the research community for tracking progress through evaluating and comparing performances. In this section, we will cover benchmark datasets according to their task categories.

Sentiment Analysis

Sentiment analysis is a task for classifying whether a textual sample reflects a negative or a positive feeling. For example, the sentence “I hate this product” reflects a negative sentiment, but the sentence "I loved this movie" has a positive sentiment. The polarity of a sentiment can either be positive, negative, or neutral, where neutral indicates the lack of sentiment. The polarity of a sentiment can be expanded into multiple categories. For example, a fine-grained sentiment analysis may focus on the following categories: Very positive, positive, neutral, negative, very negative. The Stanford Sentiment Treebank (SST) consists of movie reviews

Premise	Label	Hypothesis
A footballer is practicing penalty kicks with his teammates.	contradiction	The athlete is having dinner.
The musicians are performing for us.	neutral	The musicians are famous.
Two people are ordering food.	entailment	A couple are having dinner.

Table 2.4 Natural Language Inference examples

with human annotation for their sentiment (Socher et al., 2013). For coarse-grained sentiment analysis, the SST-2 is a binary dataset with labels for only positive and negative sentiments. The SST-5, on the other hand, is a fine-grained dataset consisting of the five sentiments mentioned earlier: very positive, positive, neutral, negative, and very negative. Another commonly used sentiment classification dataset is the large movie review IMDB dataset (Maas et al., 2011). Similar to the SST-2, it consists of binary sentiments. The IDMB dataset focuses on highly polarized sentiments only. Each review is given a sentiment score from 0 to 10, where a lower score indicates a greater negativity sentiment and a higher score indicates a higher positivity sentiment. In the IMDB dataset, reviews with a score equal or below 4, and equal or above 7 are included, while neutral reviews with scores of 5 and 6 are discarded. Yelp-2, is another notable binary sentiment classification dataset which contains reviews by Yelp users. Reviews receiving 1 or 2 stars are considered negative, while 3 or 4 stars are positive.

Natural Language Inference (NLI)

Natural Language Inference (NLI) is also known as Recognizing Textual Entailment (RTE). Given a premise and a hypothesis, the task of NLI is to determine whether there is a relationship of entailment, a contradiction or neutrality. Entailment implies a hypothesis can be inferred from the given premise, while a contradiction indicates opposing statements. Furthermore, similar to when a sentiment classification task contains examples that lack a sentiment, in NLI, there can be cases where the hypothesis and premise are neutral to each other. Examples of the three NLI cases are displayed in Table 2.4. The Stanford Natural Language Inference (SNLI) corpus (Bowman et al., 2015) and the Multi-Genre Natural Language Inference (MultiNLI) corpus (Williams et al., 2017) are examples of two NLI datasets that were manually created through crowdsourcing.

Semantic textual similarity (STS)

STS studies the semantic similarity between two pieces of text. For instance, the sentences "GM's offering is also expected to include about $3.5 billion in convertible securities." and "GM is also expected to issue $3.5 billion via a convertible bond offering, market sources said." are semantically equivalent. However, the sentences "A man is ordering a beer" and "A man is checking the time" have different meanings. In STS, a classifier can be trained to determine whether pairs of text passages are semantically similar either through binary labeling, or by assigning a score of similarity, e.g. 0 to 10. The Microsoft Research Paraphrase Corpus (MRPC) (Dolan and Brockett, 2005) is a binary classification dataset containing sentences extracted from news sources. Each pair of sentences is manually labeled to indicate whether they are semantically equivalent or not. Similarly, the Quora Question Pairs (QQP) (Iyer et al., 2017) contains questions from the community question-answering website Quora. The dataset is composed of question pairs labeled for whether they are duplicates or not.

Multitask Benchmarks

Multitask benchmarks provide multiple datasets to cover a variety of tasks. These benchmarks allow us to understand how models perform over different tasks. The GLUE benchmark (Wang et al., 2018a) includes 19 different tasks in which models are evaluated on and assigned a leaderboard[4] ranking for public performance tracking. Examples of the tasks included in GLUE are: sentiment analysis e.g. SST-2, NLI e.g. MultiNLI, Question answering, e.g. The Stanford Question Answering Dataset (QNLI) (Rajpurkar et al., 2016), STS e.g. MRPC and QQP. Note that the question answering task focuses on identifying whether a given *sentence* contains the answer to a *question*. For example, the sentence "In the Iron Age" contains the answer to the question "The period of time from 1200 to 1000 BCE is known as what?". However, the sentence "In 2013, Nigeria introduced a policy regarding import duty on vehicles to encourage local manufacturing companies in the country." does not answer the question "What is Nigeria's local vehicle manufacturer?".

In efforts to introduce more challenging tasks, the SuperGLUE (Wang et al., 2019) benchmark was proposed. SuperGLUE expanded the tasks in GLUE by including additional challenges such as coreference resolution and question answering. Coreference resolution focuses on identifying linguistic expressions that refer to the same entity in a text. For example, consider the statement "Sam bench pressed 100KG. He broke his all time high record.". Here, the word "he" refers to the person entity "Sam". In SuperGLUE, the Winograd

[4]GLUE leaderboard: https://gluebenchmark.com/leaderboard

Text	Noun phrase	Pronoun	Coreference (label)
The older students were bullying the younger ones, so we punished them .	The older students	them	True
The fish ate the worm . It was hungry.	the worm	it	False

Table 2.5 Coreference resolution examples from WSC, SuperGLUE

Schema Challenge (Levesque et al., 2012) implements coreference resolution by presenting a text passage, an entity mention and a pronoun. The training model is then tasked to classify whether the mentioned entity is a referent of the selected pronoun. Consider the examples in Table 2.5. In the first example, the pronoun "them" does indeed refer to "the older students", meaning a correct coreference relationship. In the second example, the pronoun "it" refers to the noun phrase "the fish", and not the noun phrase "the worm". Hence, "it" and "the worm" do not form a coreference relationship. In the question answering task, SuperGLUE expands on the QNLI task by introducing MultiRC (Khashabi et al., 2018), which includes a paragraph, a question and a possible answer that must be labeled as true or false.

2.5.2 Evaluation Metrics:

Precision: The precision for a class X is the number of instances that were correctly labeled X, divided by the total number of instances labeled X:

$$Precision = \frac{tp}{tp + fp} \tag{2.55}$$

Where tp (true positive) is the number of instances that were correctly predicted for label X, and fp (false positive) is the total number of instances that were incorrectly predicted for label X.

Recall: The recall value for a class X is the number of instances that were correctly labeled X divided by the total number of instances belonging to label X from the test set:

$$Recall = \frac{tp}{tp + fn} \tag{2.56}$$

Where fn (false positive) is the total number of instances that belong to class X but were not predicted X. A class X with a precision score of 1 indicates that instances predicted X do in fact belong to class X. However, this does not say anything about instances belonging to

class X that were incorrectly labeled to other classes. Similarly, a recall score of 1 indicates that every instance belonging to class X was in fact labeled X. This however does not say anything about instances from other classes that were incorrectly labeled X. To account for both precision and recall in a single measure, we use the F1-Score.

F1-Score: The $F1$ measure is a harmonic mean of precision and recall:

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall} = \frac{tp}{tp + \frac{1}{2}(fp + fn)} \tag{2.57}$$

The intuition behind taking the harmonic mean instead of the arithmetic mean is to punish cases of extreme values by giving more weight to lower scores. For instance, when $Precision = 0$ and $Recall = 1$, the arithmetic mean would be 0.5, which does not reflect the terrible performance of the classifier. Obviously, a recall of 1 does not indicate good classification performance when precision is extremely low. Hence, for the given example, the harmonic mean would result in a score of 0. This is a severe punishment for having a precision of 0, which makes more sense in reflecting the actual performance of the classifier. This behavior is useful for when the distribution of the class label in the evaluation data is not balanced.

Accuracy: Accuracy is defined as the proportion of correctly classified labels in the evaluation data. That is the total number of correct predictions, divided by the total number of the predicted data. In a binary-classification setting, accuracy corresponds to:

$$Accuracy = \frac{tp + tn}{tp + fn + tn + fp} \tag{2.58}$$

In practice, when the distribution of the class labels is balanced, the accuracy metric might be preferable for the ease of its explanation. For instance, the datasets Yelp-2, DBpedia, and SST-2, covered in section 2.5.1, contain equally distributed labels in their respective test sets. Yelp-2 and SST-2 are binary datasets with Yelp-2 consisting of 19000 samples per label, and SST-2 having a near balanced distribution over its two labels with 444 and 428 samples. DBpedia is 14-class dataset with 5000 samples for each test label. The accuracy and the test error, which corresponds to $1-$ accuracy, are usually the evaluation metrics of choice for classification tasks on SST-2, DBpedia, Yelp-2 and even sometimes for TREC-6 and MRPC (Cer et al., 2018; Howard and Ruder, 2018b; Sun et al., 2019a; Wang et al., 2018a). MRPC and TREC-6 are imbalanced, with MRPC having a distribution of 1147, and 578 samples for its labels, and TREC-6 containing 138, 113, 94, 81, 65, 9 samples for its 6 labels. The reliability of a model can be assessed by a comparison to random guessing. In the binary

case, each label can be randomly guessed with a 50 percent probability. In this case, a model that predicts the correct label 60% of the time, can be said to improve over random guessing. The No Information Rate is the best rate of guessing that can be made when the only known factor is the overall distribution of the target classes. In an imbalanced dataset, the best rate of guessing, with no information, would be to always predict the majority class. For instance, given the distribution of the test labels for the TREC-6 dataset, the best rate of guessing would be to always predict the label which occurs 138 times. This would give a guessing accuracy of $\frac{138}{500} = 0.276$ which corresponds to an accuracy of 27.6%. Hence, any model that achieves an accuracy above 27.6% on the TREC-6 test set, performs better than random. For the MRPC dataset, the best guessing rate would be $\frac{1147}{1725} = 0.6649$, or 66.49%. In our experiments, the accuracy is always above these figures, thus making our models better than random guessing. For this reason, we can confidently report the accuracy for evaluations on the TREC-6 and the MRPC tasks. In addition, since our interest is mainly to measure the impact of different data augmentation techniques through the change of performance for the same model, the intuitiveness of the accuracy metric makes it a reasonable choice for performance evaluations.

Averaging scores In a multi-class dataset, which will be the focus of this thesis, each sample can only be mapped to one target label. When performing evaluations on such datasets, the model's performance for each label can be computed individually. To compute a single score that summarizes the overall model's performance, the individual scores can be averaged. Commonly known averaging methods include; macro, weighted, and micro averaging. Macro averaging is the most straightforward form of averaging, and is computed by taking the arithmetic mean of all the label scores. Alternatively, the weighted average of the label scores can be taken, in which each score is multiplied by the number of test samples for that specific label. Finally, the micro averaging aggregates the contributions of all labels to compute a global average score. In a multi-class setting, micro averaging computes the proportion of correctly classified observations out of all observations. This is in essence computes the overall accuracy. Note that in a balanced dataset, where the support is the same for all classes, all three averaging scores will result in the same value. In this thesis, we report results based on the micro averaging of the f1 score. In a multi-class setting, this corresponds to the accuracy. In the case of the 2-class MRPC experiments, evaluated on the highly imbalanced dataset in chapter 5, we report both the F1 and the Accuracy metrics, as suggested by the GLUE benchmark (Wang et al., 2018a). For the TREC-6 experiments, the gap between Macro and micro F1 scores is insignificant, thus we only report the micro-F1. In fact, in chapter 3 for instance, we are mainly interested in measuring the performance gain

between different runs which is always consistent for both the F1 micro average, and the F1 weighted-average.

Perplexity

Language generation models can be evaluated based on the data distributions they produce, see section 2.1.3. This means a model can be used to estimate the quality of a sentence. Hence, based on equation 2.1, sequences that are assigned high probability scores are considered more favorable by the model than sequences with low probability scores. In this way, a model can be evaluated based on the probability scores it assigns to unseen sequences. A model that assigns high probability scores to unseen data is said to be *not perplexed* to see it. If the test set contains real-world data, then a model that assigns higher probability scores is considered to be "better" than one that assigns lower probability scores. Hence, the better model is less perplexed to see the new data. Note that a sequence probability score is computed by multiplying the probability estimates for its elements. This means that with everything else being equal, a larger test set is likely to have a lower probability estimate than a smaller one. For this reason, it is desirable to have a probability estimate that is less influenced by the size of the dataset. This can be done by normalizing the sequence probability by the size of the dataset. For a dataset of size n, we consider the log form of equation 2.1:

$$\ln P(w) = \ln \prod_{i=1}^{i=n} P(w_i) = \sum_{i=1}^{i=n} \ln P(w_i) \tag{2.59}$$

We now divide by n to normalize the probability estimate by the size of the dataset:

$$\frac{\ln P(w)}{n} = \frac{\sum_{i=1}^{i=n} \ln P(w_i)}{n}$$

We now exponentiate to remove the log:

$$e^{\frac{\ln P(w)}{n}} = e^{\frac{\sum_{i=1}^{i=n} \ln P(w_i)}{n}}$$

$$e^{\ln P(w)\frac{1}{n}} = e^{(\ln \sum_{i=1}^{i=n} P(w_i))\frac{1}{n}}$$

$$P(w)^{\frac{1}{n}} = (\sum_{i=1}^{i=n} P(w_i))^{\frac{1}{n}}$$

We can now take the inverse of the probability, to get the *perplexity*:

$$PP(W) = \frac{1}{P(w_1, w_2, w_3, \ldots, w_N)^{\frac{1}{N}}} \tag{2.60}$$

where N is the size of the dataset. Since the inverse probability is taken, a lower perplexity indicates a better model.

2.6 Bias-Variance Trade-Off

Benchmarks can be useful for evaluating the performance of a final version of the trained model. Understanding prediction errors a model makes can be useful to improve its training. Errors can be related to the *overfitting* or *underfitting* of the trained model. These concepts will be explained later in this section, but first we will shed light on the importance of data splitting for proper training.

Data Split It is a common practice in machine learning to split the task into three sets: a training set, a validation set, and a test set. A model is trained on the training set with the objective of minimizing its mistakes as it fits to the training data. The validation set on the other hand is used as an indicator of the model's predictions on unseen data during training. Changes to the model's hyperparameters can be made by measuring its performance on the validation set. Thereby, the hyperparameters that lead to the best validation scores can be assumed to produce the model that would best perform on unseen data. Finally, the test set is only used for the purpose of evaluating the final model to give indication of the model's performance on new unseen data. A supervised machine learning algorithm is trained to estimate a mapping function f for the output Y given the input data X. When fitting f on X, we aim to minimize the prediction error on the training data. For f to achieve good predictions on unseen data $\hat{X}$, it must be able to generalize well from its training. The generalization error indicates the trained algorithm's ability to classify unseen inputs $\hat{X}$ appropriately. We assume $\hat{X}$ is independent of X but shares an identical distribution to X. When aiming to achieve good generalization for f, we aim to minimize the gap between the training error and testing error. As our training objective is to minimize the training error, we run into what is known as the bias-variance trade-off.

Bias and Variance For any input $x \in X$, the bias of the class predicted by f is reflected in the training error; that is, the difference between the predicted class at x and the true target at x. Here, f is said to exhibit high *bias* when the training error is high. This means that f is not

able to learn the underlying relationships in the training data X due to its inability to properly fit X, and is said to *underfit*. When testing f on unseen data, e.g. a validation set, we hope to achieve a validation or generalization error close to the training error. If f results in a low training error on the training set, but a high generalization error on the unseen data, it is said to exhibit high *variance*. The large difference between the errors indicates an *overfit*. That is, f fitted the training data X to such a high degree that it also learned the noise in the data instead of a generalization of the underlying distribution.

More formally, suppose the parameter θ defines the function $f(s)$ on a sample s, where $s = x_1, \ldots x_n$ for data x that is independent and identically distributed. The estimated parameter can be defined as $\hat{\theta}$. The *Bias* of the estimator is the difference between the estimated value of the parameter $\hat{\theta}$ and the expected value of θ:

$$\text{Bias}(\hat{\theta}, \theta) = E(\hat{\theta}) - \theta \tag{2.61}$$

The *Variance* of the estimator $\hat{\theta}$ is the expected difference between the value of the prediction function $\hat{\theta}$ estimated on randomly sampled data and the expected value of $\hat{\theta}$. As shown in equation 2.62, Variance measures how $\hat{\theta}$ is expected to vary as different samples from the same distribution are obtained. Unlike Bias, the Variance of the estimator does not directly depend on the true value of the parameter θ:

$$\text{Var}(\hat{\theta}) = E[(E[\hat{\theta}] - \hat{\theta})^2]. \tag{2.62}$$

Bias-Variance trade-off

To shed light on the trade-off between the Bias and Variance, we will analyze the Mean Squared Error (MSE) between the estimator $\hat{\theta}$ and the true function θ:

$$MSE = E[(\hat{\theta} - \theta)^2] \tag{2.63}$$

By applying the binomial expansion, we show that the MSE decomposes into Variance and Bias terms:

$$\begin{aligned} E[(\hat{\theta} - \theta)^2] &= E[(\hat{\theta} - E[\hat{\theta}] + E[\hat{\theta}] - \theta)^2] \\ &= E[((\hat{\theta} - E[\hat{\theta}]) + (E[\hat{\theta}] - \theta))^2] \\ &= E[(\hat{\theta} - E[\hat{\theta}])^2 + (E[\hat{\theta}] - \theta)^2 + 2(\hat{\theta} - E[\hat{\theta}])(E[\hat{\theta}] - \theta)] \end{aligned}$$

Since θ is deterministic, $E[\theta] = \theta$

$$= E[(\hat{\theta} - E[\hat{\theta}])]^2 + (E[\hat{\theta}] - \theta)^2 + 2E(\hat{\theta} - E[\hat{\theta}])(E[\hat{\theta}] - \theta)$$
$$= [\text{Bias}]^2 + \text{Variance}$$

where

$$2E(\hat{\theta} - E[\hat{\theta}])(E[\hat{\theta}] - \theta) = 2(E[\hat{\theta}] - E[\hat{\theta}])(E[\hat{\theta}] - \theta) = 0$$

Based on this decomposition, a model with a high Bias and low Variance will produce a high generalization error. A model with high Variance and low Bias will also have high error. This shows that by balancing the Variance and Bias, the error can be minimized. As such, the higher the complexity of a model is, the higher its variance is and the lower its bias is. Similarly, the lower the complexity of a model is, the lower its variance and the higher its bias is (Hastie et al., 2009). The increase in a model's complexity can lead to a stronger fit to the training data and as a result a lower training error. However, too much fitting can lead to worse generalization on unseen examples as the model adapts itself too closely to the training data so that it also learns the noise. In contrast, as the model's complexity decreases, it is more likely to underfit and have high bias, which can lead to higher generalization error as well. Figure 2.26 summarizes this phenomenon and shows that if a model is tuned well enough, it can achieve an optimal "sweet-spot" where the bias and variance are at their lowest to achieve the best generalization.

2.7 Revisiting the Bias-Variance Trade-off

In section 2.6, we explained that a properly trained model must be able to balance between underfitting and overfitting, such that it is complex enough to learn the underlying structure of the training data, but simple enough to ignore the noise in the data. However, recent studies have shown that this phenomenon fails when applied to over-parameterized models, like deep neural networks. Neyshabur et al. (2014) revisited the bias-variance trade-off in the context of deep neural networks and random forests, and argued that when a model's complexity increases beyond an interpolation point, the generalization error starts to decrease. This finding can be explained by extending the U-curve from Figure 2.26 to form a "double descend", as can be seen in Figure 2.27. While remaining to the left of the interpolation point, as in the classical machine learning setting, increasing a model's complexity beyond the bias-variance "sweet-spot" can risk increasing its generalization error. However, when continuing

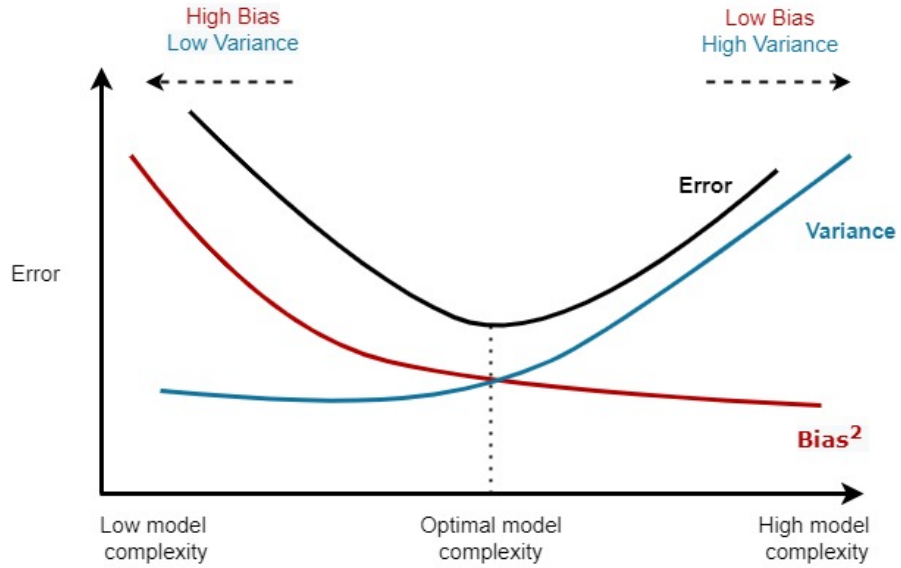


Fig. 2.26 Generalization error can be minimized at the cross point of bias-variance trade-off, which can be adjusted by tuning the trained model's complexity

to increase the model's complexity, after some point the generalization error is likely to decrease again. This "double descent" phenomenon has called for further investigation into the behavior of parameterized machine learning models. Hence, other studies have attempted to further explain this phenomenon (Belkin et al., 2019, 2020; Hastie et al., 2019; Muthukumar et al., 2020; Nakkiran et al., 2019).

2.8 Regularization

Models with high complexity can be tuned by adjusting their hypothesis space. Regularization penalizes a model's complexity to avoid overfitting and improve generalization on unseen data. In general terms, the complexity of a neural network can be associated with the number of its training parameters. As the number of parameters increases, the network is able to learn more complex relationships between the input and output. However, when there is a limited number of training samples, the learned relationships can be a result of noise sampling in the data, and as such the network overfits. Many regularization practices have been proposed in the literature to reduce the overfitting of neural network models. Some of these include bagging, weight regularization, dropout of neural units, and augmentation of training data, explained in section 2.1.4. We explain the concept of bagging under Ensemble learning, in section 2.9.1. The other techniques are explained below.

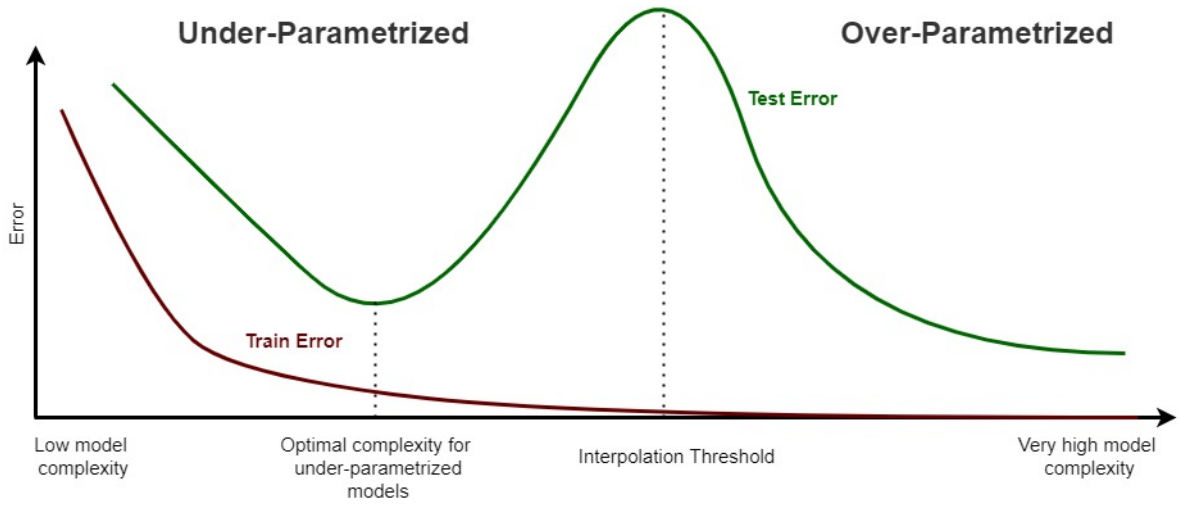


Fig. 2.27 Double Descent curve. Generalization error is expected to decrease beyond the interpolation threshold. Based on Figure 1(b) from Belkin et al. (2019)

2.8.1 Weight Regularization

In a neural network, the larger the values of the hidden units, the higher the impact they can have on the final output. This is because large weights tend to trigger sharp transitions in the network, thus small changes in the input can lead to large changes in the output (Reed and MarksII, 1999). One way to combat this is with weight regularization, in which a vector norm penalty is added to the cost function. In this way, the model is forced to keep its weights small to minimize the regularization penalty. As shown in equation 2.64, weights can be regularized by adding a penalty term $\lambda \sum(w)$ to the loss function. Unless the penalty minimizes the first term of the loss function C, if it is too large, the model is forced to minimize the size of the weights and thus underfit to the problem. If the penalty is too small, the effect of the regularization is minimized and thus the model is allowed to overfit to the training data.

$$C = Loss + \lambda \sum(w) \tag{2.64}$$

The severity of the regularization is controlled by adjusting the value of λ. If $\lambda = 0$, no penalty incurs, but as λ grows, the imposed regularization penalty becomes more prominent.

The L1 regularization, also known as lasso regularization, penalizes the absolute sum of the weights w as shown in equation 2.65. This expression tends to produce sparse networks that focus on higher importance connections, while other weights shrink to 0:

$$C = Loss + \lambda \sum_{w} |w| \tag{2.65}$$

The L2 regularization, shown in equation 2.66 penalizes the sum of the squares of the weights. Unlike L1, L2 is less prone to produce sparse networks as it does not tend to push less important weights to 0:

$$C = Loss + \lambda \sum_{w} w^2 \tag{2.66}$$

It is possible to combine L1 and L2 regularization terms in the cost function, this becomes known as the elastic net regularization.

2.8.2 Dropout

In 2014, Srivastava et al. (2014) proposed dropout as a regularization technique for neural networks. Unlike L1 and L2, dropout does not require modifications to the cost function. Instead, with dropout, random neurons along with their connections are switched off during training. Here, neurons switch off at a user-defined probabilistic rate p. Dropout is intended to cure the issue of co-adaption in neurons, which mostly affects large networks during training. Co-adaption is caused by high correlation between different groups of neurons. When co-adapted neurons behave similarly, they act as a single group or neuron. As a result, dependent neurons are made redundant and less likely to properly respond to the gradient updates in backpropagation. Therefore, dropout attempts to break this interdependence by randomly switching off different neurons during training. In this way, different connections are updated at each backpropagation step, resulting in network weights that are more spread out, as in L2 regularization. This technique makes the network rely less on particular connections, thus encouraging the participation of more connections in its predictions.

As explained in section 2.2, in a standard neural network training, input x is first forward-propagated through the network. The training loss can then be measured, e.g. equation 2.11, between the network's output $\hat{y}$ and the true value of the target y. The error is then back-propagated through the network to update its weights. When dropout is applied to a neural layer, a random subset of its neurons with their connections are switched off in every full forward and backward pass. This constructs a new simpler network for each training iteration, in which only the weights of the neurons that were not dropped are updated. After a complete training pass, the dropped neurons are restored so that a new subset of neurons is dropped before the next training iteration. When this process is repeated over multiple training iterations, the outcome will be similar to training different versions of thinned neural networks, such that each network fits the data in a different way.

At testing time, dropout is not applied anymore to allow all neurons to contribute to the prediction. However, because at training time every hidden neuron was only active with a

probability $1-p$, we scale the activations of the layers that used dropout by $1-p$ when making predictions during testing. This ensures that for any neuron, its expected output remains under the distribution used to drop units at training time. For instance, if the dropout rate is $p = 0.5$, half the neurons are expected to be active for each prediction during the training time. Now when testing the model, because all neurons are active, there will be double the number of participants in the output. To compensate for this effect, the activations of the dropout layer will be scaled by $1-p = 0.5$, the rate at which each neuron was active in training.

Overall, dropout can help avoid overfitting as it acts similarly to bagging, discussed in section 2.9.1, where multiple thinned neural networks are trained on the same data. Moreover, since dropout results in training thinned networks at each iteration, it in effect reduces the amount of required computations as opposed to training without dropout. Hence, the larger the dropout rate, the thinner the network becomes. This makes dropout an alternative to training multiple fully-connected neural models at a reduced computational complexity.

2.8.3 Data Augmentation

With enough training data, the previously mentioned techniques can improve the generalization of the training model. However, these methods only work by changing the model's fit to the existing data, and might not significantly improve generalization when the data in question becomes the bottleneck to performance, i.e., in low data regimes. Hence, in this thesis, we attempt to solve generalization by working on the data level instead. In a low data regime, the lack of diversity can make it difficult for the parameters to properly adjust to unseen data patterns. In this case, input invariance, as in slight changes to the input, could lead to different model responses. For instance, in a dog and cat image classification task, if an input image is rotated, the model should be expected to produce the same output. Hence, to capitalize on data invariance as prior knowledge to the model's learning parameters, image augmentation techniques such as rotation, cropping, and color changes can be applied. Data Augmentation is a powerful regularization technique that seeks to artificially expand the size of the training set by creating additional synthetic examples. In this approach, we hope to expose the trained model to more data variations and thus improve its generalization error. In contrast to the previously mentioned regularization techniques, Data Augmentation approaches overfitting from the root of the problem: the training dataset. Section 2.1.4, extended the discussion of data augmentation to the NLP domain.

2.9 Ensemble Learning

In contrast to ordinary training approaches where only one learning classifier is constructed from the training data, ensemble learning utilizes a number of learners to solve a specific task. An ensemble contains multiple learning classifiers called base learners. A learner can be any supervised machine learning algorithm, e.g. neural networks. By combining the knowledge of multiple classifiers, an ensemble can often achieve better generalization than its base learners. In fact, a phenomenon known as the "wisdom of the crowd" suggests that multiple human individuals, even if mostly non-experts, can often perform a better job at making predictions than the individual participants. Surowiecki (2005) demonstrates this phenomenon by presenting several social science studies in multiple fields such as economics and psychology. Examples include asking groups of individuals general knowledge questions, like estimating the population of a country. Surowiecki explains that the best collective decisions are made through a diversity of heterogeneous, independent opinions. That being said, participants must not share information, as sharing biases or incorrect estimates can lead to less accurate crowd predictions (Treynor, 1987).

In a similar vein, if multiple weak learners are trained independently to solve the same task, aggregating their outputs can outperform any of the base learners (Fu, 2004; Zhou, 2019). In this context, a weak learner can be described as a low-performing classifier that performs slightly better than random guessing, e.g. achieves an accuracy that is slightly better than 50% on a binary classification task. Furthermore, Schapire (1990) shows that ensembles of weak learners can sometimes perform as well as strong learners. There exist multiple approaches to building ensemble models, of which the most commonly used ones are bagging, boosting, and Monte Carlo dropout for neural networks. We briefly explain each approach in the next sections.

2.9.1 Bagging

In bagging, different classifiers or instances of the same classifier are trained separately on subsets of the training set. To make the learners as independent as possible, the training dataset D can be split into n non-overlapping subsets $s_1, \ldots s_n$. However, due to the finite amount of data that can exist, the larger n is, the smaller each data subset will be. This could result in unrepresentative samples that could negatively impact the training of the base learners. To account for this issue, the data subsets are sampled from D with replacement (Breiman, 1996). Sampling a subset s_i from D with replacement is commonly known as bootstrapping (Efron and Tibshirani, 1994). When a sample x is drawn from D, it is added to s_i but also replaced back to D. As the elements in D do not change after the first draw, it

remains possible for x to be selected in the next draw. Thus, every example has an equal chance of being selected in each sampled training dataset. Once the datasets are sampled, a learner is independently constructed from each subset through training. After the base learners have been trained on their corresponding sampled datasets, their outputs need to be aggregated for the final ensemble prediction. For classification tasks, aggregation is done by voting, in which each base learner participates in the final prediction by giving its votes for the target values. Two common voting methods are known as hard-voting and soft-voting.

Hard Voting

In hard voting, the predicted label by each classifier becomes a vote, and the label with the most votes is selected. For instance, consider an ensemble of 3 classifiers, C_1, C_2, and C_3, trained on a 3-label classification task. Now assume input x yields the following predictions for the labels A, B, and C:

$$\text{Classifier 1} \rightarrow [0.1, 0.2, 0.7]$$
$$\text{Classifier 2} \rightarrow [0.5, 0.1, 0.4]$$
$$\text{Classifier 3} \rightarrow [0.1, 0.1, 0.8]$$

where each target vector is a predicted probability distribution for the labels [A, B, C]. For instance, classifier 1 predicted a probability of 0.1 for label A, 0.2 for label B, and 0.7 for label C. In this example, classifiers 1 and 3 predicted the highest probability for label C, whereas classifier 2 predicted A with the highest score:

$$\text{Classifier 1} \rightarrow \text{label C}$$
$$\text{Classifier 2} \rightarrow \text{label A}$$
$$\text{Classifier 3} \rightarrow \text{label C}$$

It can be noted that two classifiers voted for C, and one voted for A. Therefore, with hard voting the final label will be C, as it has the majority of the votes. Note that an ensemble with an even number of base classifiers could lead to a tie in votes. This becomes less of an issue with an odd number of classifiers, as there will be one tie-breaker.

Soft Voting

In soft-voting, the average of the predictions is taken, and the label with the highest average probability score is selected. For soft voting to work, each individual classifier must output a probabilistic distribution over the target labels.

$$\hat{y} = \arg\max_i \frac{1}{m} \sum_{j=1}^{m} p_{ij}, \tag{2.67}$$

Consider the example in the previous section, and assume this time that each classifier produced the following probabilities for its predictions:

$$C_1(\mathbf{x}) \rightarrow [0.1, 0.2, 0.7]$$
$$C_2(\mathbf{x}) \rightarrow [0.6, 0.2, 0.2]$$
$$C_3(\mathbf{x}) \rightarrow [0.2, 0.3, 0.5]$$

We now take the average probabilities by applying equation 2.67:

$$p(i_1 \mid \mathbf{x}) = \frac{0.1 + 0.6 + 0.2}{3} = 0.3$$
$$p(i_2 \mid \mathbf{x}) = \frac{0.2 + 0.2 + 0.3}{3} = 0.23$$
$$p(i_3 \mid \mathbf{x}) = \frac{0.7 + 0.2 + 0.5}{3} = 0.47$$

The predicted probabilities for this ensemble are [0.3, 0.23, 0.47], which yields a prediction for label C as it is assigned the highest probability of 0.47. In this example, all 3 classifiers are assigned equal weights. Furthermore, we can control the contribution of each vote by assigning a weight w_j to each voting classifier. Here, the voting power for classifier C_j can vary depending on the value of w_j. With weighted soft voting, equation 2.67, becomes:

$$\hat{y} = \arg\max_i \sum_{j=1}^{m} w_j p_{ij}, \tag{2.68}$$

where w are normalized weights that sum up to 1. From the previous example, we assign C_1, C_2, and C_3 the weights $0.6, 0.3$, and 0.1 respectively. The average vote now becomes:

$$p(i_1 \mid \mathbf{x}) = 0.6 \times 0.1 + 0.3 \times 0.6 + 0.1 \times 0.2 = 0.26$$
$$p(i_2 \mid \mathbf{x}) = 0.6 \times 0.2 + 0.3 \times 0.2 + 0.1 \times 0.3 = 0.21$$
$$p(i_3 \mid \mathbf{x}) = 0.6 \times 0.7 + 0.3 \times 0.2 + 0.1 \times 0.5 = 0.53$$

Here, the ensemble's prediction for input x remains label C, but with a probability of 0.53. Overall, regardless of the voting method, bagging can help reduce overfitting when the participating classifiers have high variance and unstable predictions. By combining their predictions, the generalization error can be stabilized. However, if the classifiers are stable or have high bias, bagging may offer little improvement.

2.9.2 Boosting

Boosting is a popular ensembling method that was first introduced by Cootes et al. (2001). Similar to bagging, boosting combines multiple weak learners into one ensemble. However, it is different in that classifiers are trained on data samples that were mistakenly predicted by the other learners. For instance, if classifier A mislabels an example x, then x becomes part of the training data for classifier B. This means that base learners are no longer trained independently of each other. Instead, subsequent learners focus on improving the predictions of the previously trained classifiers. Mistakes in predictions can indicate data patterns which are difficult for a classifier to learn, i.e. having high bias, refer to section 2.6. Hence, by creating weak learners that improve over each other's predictions, boosting is able to reduce bias. In terms of reducing variance, the aggregation method plays a crucial role. For instance, regardless of the ensembling procedure, if the base learners have high variance, soft-voting will help reduce it.

2.9.3 Monte Carlo Dropout

The standard dropout is explained as a training regularization technique in section 2.8.2. During testing, dropout is not applied, making the network's predictions deterministic. In contrast to the standard dropout, the Monte Carlo dropout continues to drop neurons at testing time. This makes the network nondeterministic, and as such could predict a different value for the same input. By running multiple passes to the neural network, the softmax probabilities for each class can then be aggregated by averaging. Gal and Ghahramani (2016)

introduced dropout as a method for capturing the epistemic uncertainty, from section 2.11, in a neural network.

2.10 Rethinking Generalization in Deep Neural Networks

The recent trend towards larger language models has shown that over-parametrization can in fact improve generalization. For example, the creation of GPT-3, the successor of GPT-2, has shown that NLP performance does indeed scale with number of parameters. GPT-2, a transformer-based language model, consisted of stacked decoder blocks, totaling a count of 1.5 billion parameters. In creating its successor, GPT-3 was built with 175 Billion parameters, over 10x the size of its predecessor, and managed to achieve substantial improvements (Brown et al., 2020). Despite their growing complexity, large-scale language models continued to improve generalization. In fact, Zhang et al. (2016) observed that over-parameterized deep neural networks manage to generalize even without the application of any explicit regularization. In their work, the authors explain over-parameterization as a ratio between the number of learning parameters and training samples. When the number of learning parameters exceeds the training data by a large margin, a model becomes over-parameterized. Their findings coincide with the discussion in section 2.7; when moving from an under-parameterized to an over-parametrized setting, the generalization error diverges from a "U-shaped" curve to form a "double-descent", as shown in Figure 2.27. The authors show that over-parametrized models learn the training data with its noise, and still generalize well. Yet, traditional optimization methods are incapable of explaining this phenomenon. Even when replacing true labels with random labels, networks trained with gradient descent methods still manage to easily fit the training data. Adding random noise to the training data did not stop these networks from easily learning the data either. This phenomenon raises questions on the capability of traditional methods in explaining generalization for complex architectures that consist of multiple non-linear parameterized layers.

2.11 Uncertainty Estimation

Uncertainty estimation relates to model explainability, as it identifies the reliability of a model in predicting an output. Estimating a model's uncertainty has been extensively studied in the machine learning community (Blundell et al., 2015; Graves, 2011; Hüllermeier and Waegeman, 2021; Lakshminarayanan et al., 2017). A representation of uncertainty is desirable in supervised learning, and thus quantifying uncertainty plays a key role in making reliable predictions (Abdar et al., 2021). In general, a classifier can benefit from an example

e, if it can improve its generalization on the data. In this way, for a classifier C that is trained on data L_T, the usefulness of example $e, e \notin L_T$, can be determined by: a) appending e to L_T, b) then retraining C on $L_T \cup e$. We then measure the difference in the performance of C on a separate validation data L_V. If there is an increase in performance, then the example e is deemed useful. Although this idea may seem intuitive, its application may not be feasible for the following reasons:

- This process assumes the existence of a separate dataset L_V that is representative of the overall data distribution. Considering that the aim of the work carried out in this thesis is to improve classification performance in strictly small data regimes, the assumption that L_V exists may invalidate the proposed setting.
- The process of measuring the difference in performance between C trained on L_T, and C trained on $L_T \cup e$ can be impractical as it adds a computational overhead. In this case, it is infeasible to retrain and evaluate C for every generated example e.

For the above reasons, we attempt to approximate the usefulness of an example before it is added to the training data. For the purpose of this thesis, we assume that when a useful example is added to the training data L, the performance of the learning classifier is improved. As such, the usefulness of an example can be estimated by its ability to improve the performance of a classification model trained on L. This can be done by measuring the uncertainty of the classifier C on example e. Here, we make the assumption that examples that result in high uncertainty are likely to include information different from what is in L_T. Hence, when the classifier is trained on such examples, its performance is expected to increase as a result of learning new data patterns. In this regard, it is important to distinguish between aleatoric and epistemic uncertainty.

Aleatoric and epistemic uncertainty In some cases, examples where the classifier shows high uncertainty may cause more harm than good if added to the training set. These are instances that contain noisy information that could contribute to the confusion of the model. When the amount of noise in the dataset is significantly large, the trained model may suffer in learning the required representation. In this case, the *aleatoric uncertainty* occurs when the uncertainty of the model is affected by the uncertainty in the training data. Aleatoric uncertainty can result from examples that create confusion in the labeling process for multi-class classification tasks. For example, consider the question "What was Einstein's IQ, and where was he born?". In a question classification task, the first segment of this question can be labeled "NUMBER", whereas the second segment can be assigned the label "LOCATION". In a multi-class classification task where each input is only mapped to a single label, it would

be unclear how such an example can be labeled. Other examples of aleatoric uncertainty can be found in incoherent samples, e.g. “where how is there?”. When labeling for question classification, the words ‘what’ and ‘how’ are strong features for their respective labels. ‘Where’ could be a feature indicating a location as an answer, while ‘how’ could indicate a description. Having both features in a single, yet meaningless, question that does not add appropriate information to the distribution of the data can only harm the classification performance.

In contrast to the aleatoric uncertainty, the *epistemic uncertainty* captures the classifier’s inability to distinguish between different instances. In a sentiment classification task, this happens when a classifier is uncertain in predicting the correct label for a sound and coherent statement like “I enjoyed watching this movie!”. Minimizing such uncertainty would contribute to the improvement of the classifier’s performance. In an ideal scenario, one would aim to include examples that minimize epistemic uncertainty, while avoiding examples that contribute to aleatoric uncertainty.

2.12 Language Model Decoding

In section 2.1.3 we discussed the importance of conditional probability for language modeling. A language generation model, such as GPT-2, is trained with the objective of predicting the next word given a context of historic words. For example, for the sequence "I lost my keys", we can make the assumption that $P(\text{“keys”}|\text{“I lost my”}) = 1$, whereas the probability for every other token w is $P(\text{w!=“keys”}|\text{“I lost my”}) = 0$. This estimation allows us to apply the cross-entropy loss function, from equation 2.12, to update the model’s weights. During inference, for any input sequence $s_n = w_1 \dots w_n$ of n tokens, the softmax function, from equation 2.12, can be applied to reweigh the model’s outputs to form a probability distribution over its vocabulary. Based on the scores, a token can be selected and added to the s_n to form a new sequence s_{n+1}. This process can be repeated until a desired sequence length is reached, or until a symbol token indicating the end of sequence is selected, e.g. end-of-sequence [EOS].

Argmax sampling When decoding the model to generate text, different techniques can be applied to select the next token. The simplest of all is selecting the token with the highest probability:

$$w_{n+1} = \text{argmax}_{w \in V} P(w|s_1 \dots s_n) \tag{2.69}$$

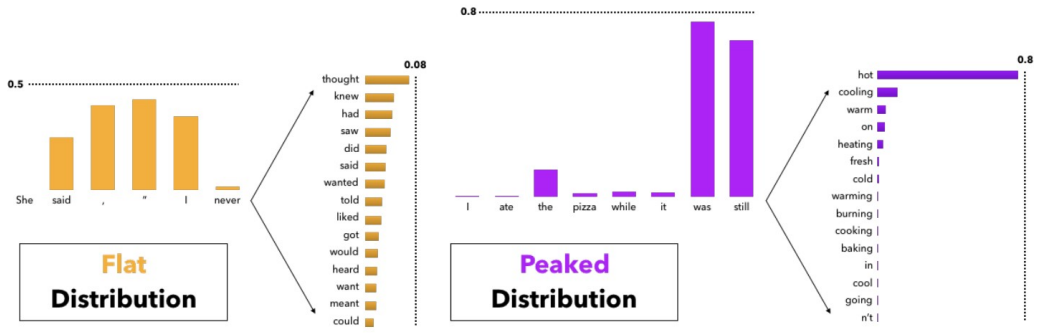


Fig. 2.28 Flat vs. peaked distributions. A flat distribution may lead to many reasonably probable tokens, while peaked distributions may lead to only a few tokens having most of the probability mass. Figure Source: Holtzman et al. (2019)

where w_{n+1} is the token with the highest probability score, V is the set of all vocabulary tokens, and $s_1 \dots s_n$ is the current sequence input. This approach has a simple implementation, but it is very greedy and will always generate the same sequence output for the same input. Since we aim to generate different text samples, such a greedy approach will be problematic. To get variation in the generated output, the selection strategy must not be restricted to one token only. Instead, it must consider other tokens as well.

Top-K sampling Instead of aiming to generate text that maximizes the likelihood of the sequence, Fan et al. (2018); Radford et al. (2018) apply a *top-k* selection strategy to only retain the k tokens with the highest probability mass. This is done by first sorting the vocabulary based on the model's outputs, then zeroing out all the tokens outside the top-k candidates. In this way, instead of sampling the next token from the total vocabulary tokens, we only sample on the top-k subset. Regardless of the probability distribution for the next token, the number of candidates will always be k. This may be problematic in flat and peaked distributions, shown in figure 2.28. On the one hand, in a flat distribution, a small value for k would lead to missing out on other probable tokens. On the other hand, for a peaked distribution, a large k would result in the presence of many less probable tokens.

Nucleus sampling Another sampling strategy, known as Nucleus sampling, was proposed by Holtzman et al. (2019) as an improvement over top-k. In top-k, the probability distribution of the vocabulary is reweighed over the k most probable tokens. However, considering that the number of most probable tokens may vary from one distribution to another, redistributing over a fixed window of k tokens can be problematic. This is because, depending on the distribution, a higher k is more likely to include less useful tokens, while a lower k may lead

to missing out on useful tokens. Hence, to avoid this issue, in nucleus sampling, the number of sampled tokens may vary depending on the distribution. This is because sampling is done on the least number of tokens that can achieve a cumulative probability above a predefined threshold p. For instance, if $p = 0.7$, tokens are first sorted by their probability scores, then the top n tokens are selected, such that the summation of their probabilities reaches 0.7.

Once a sampling strategy is put in place, the next step would be to generate distinct variants of sequences of text. This is obviously unachievable when the sampling strategy only retrieves the same token for the same input history, which is the case in argmax sampling; where the token with the maximum probability at each time-step is always selected. Hence, with argmax sampling, a beginning-of-sequence token, e.g. *BOS* will always lead to the same output sequence. Accordingly, selecting the token that maximizes the conditional probability of the sequence will not result in diversifying the generated outputs. Instead, the same output sequence will always be generated. Since we aim to generate variants of text samples that complement an existing dataset, we must apply a less greedy approach. We can further extend our reasoning to the following assumption: the greedier the approach, the less likely the generated samples are diversified. This can be justified by showing that there would be fewer combinations of possible sequences for greedy decoding approaches. The extreme case would be decoding by selecting the token that maximizes the likelihood of the sequence at each time step. As the decoding criteria is loosened, the possibility of generating different sequences is increased. However, too much loosening of the selection criteria can result in outputs that may be less coherent.

2.13 Search Methods

2.13.1 Beam Search

Beam search can be applied as a decoding strategy to generate text sequences. For each next token generated, the algorithm must decide which of the vocabulary tokens leads to the highest probability score. Recall from equation 2.1 that the probability of a sequence can be approximated by multiplying the conditional probability of each of its tokens. In Beam Search, different combinations of sequences are considered at each time step, such that the sequence with the highest probability is selected. During the decoding process, n sequences with the highest conditional probabilities are kept track of. Here n is a user-defined parameter that fixes the width of the beam at each step. It controls the number of candidate sequences for every next token the beam search considers. Figure 2.29 provides an illustration of a tree constructed from the paths taken by beam search as it searches for the sequence with the

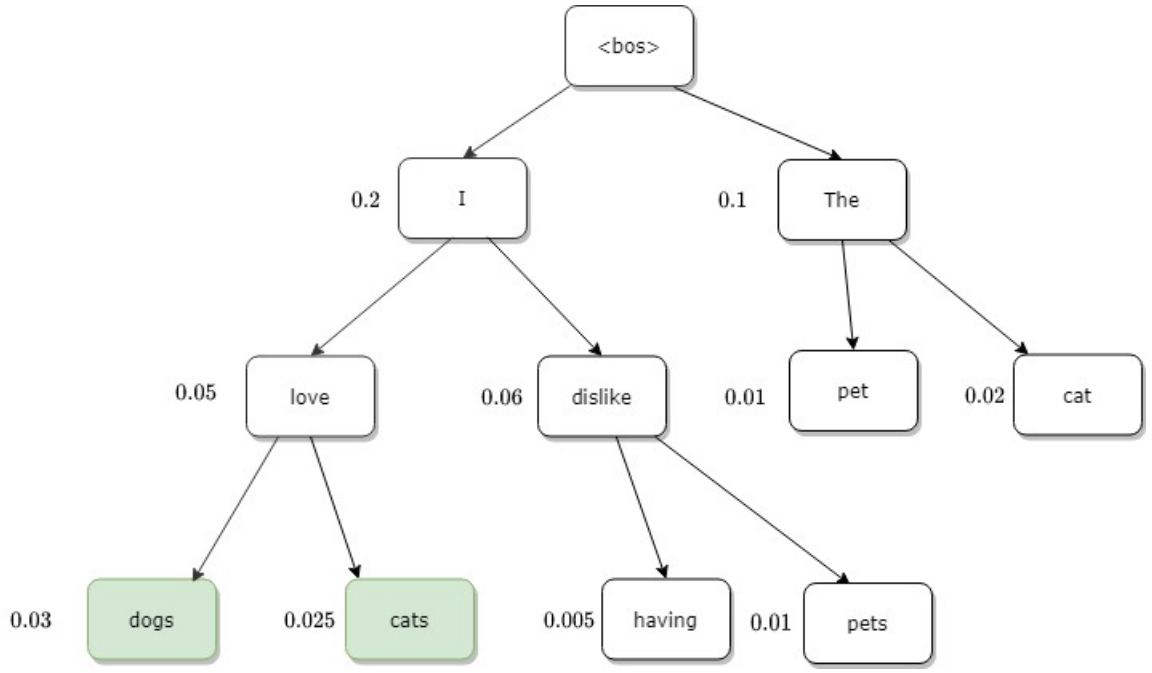


Fig. 2.29 Beam Search with beam width $n = 2$. For simplicity, the displayed scores are arbitrary numbers, not the log probabilities from equation 2.1

highest probability score. The most probable paths for $n = 2$ are in "<bos> I love dogs" and "<bos> I love cats". The beam tree continues to expand until an end-of-sentence token is reached. By increasing the beam width, a more probable sequence is likely to be generated, but at the expense of significantly increasing the search time. Furthermore, if a search tree is reconstructed for every new path, it is likely to end with sequences sharing the same prefix. For this reason, beam search is not applied in this thesis.

2.13.2 Non-Guided Search

A faster and less greedy alternative to beam-search would be to generate tokens without attempting to maximize the cumulative probability score. In this approach, we do not construct a tree. As such, we do not keep track of previously visited paths. Instead, at each time step, a token is randomly sampled from the probability distribution of the vocabulary. Either a top-k or a top-p sampling strategy can be applied to generate candidate tokens at each time step. Unlike, beam-search, this approach does not construct a pruned tree of possible paths. Instead, it generates a full sequence in a single iteration. Although this approach is significantly less complex than beam search, the generated sequences are less likely to follow the most probable paths. Nevertheless, for the purpose of this thesis, we are less interested in finding the most probable path that a language model can produce. Either top-p or top-k

sampling procedures can lead to high probability sequences that are sound and coherent. If non-guided decoding is repeated multiple times, starting with the "<bos>" token, we are less likely to generate sequences that share the same path as in beam search.

Once enough data is generated, a task-appropriate selection strategy can be applied to retain samples that are deemed useful. For instance, if the aim is to create examples of a specific task label X, a classifier can be applied to predict labels for the generated data. Examples that are predicted X can then be selected. Additionally, samples with the highest classification confidence can also be retained.

2.13.3 Monte Carlo Tree Search (MCTS)

If the algorithms discussed in section 2.12 are measured on a scale of greediness, the argmax from equation 2.69 can be labeled as very greedy, beam search as greedy, and the non-guided generation as the least greedy of all. In this section, we discuss Monte Carlo Tree Search MCTS: an algorithm that attempts to find a balance between the greedy and less greedy routes. This makes MCTS an optimization algorithm that seeks to find optimal solutions in a search space that is close to infinity in size. In a tree analogy, every path from the root node to every terminal node corresponds to a distinct example. In data generation, each next token can have k candidates, and sentences can be of varying lengths. Assuming all generated sentences have a length of 20, for a branching factor of 3 candidate tokens, the total number of possible sequences would be $3^{20} = 3,486,784,401$. This makes the examination of every single path computationally expensive. MCTS attempts to overcome this challenge by only traversing through the paths that are more likely to be desirable, thus efficiently limiting the number of generated sequences. For this reason, in this thesis, we use MCTS as the main method for exploring generative data augmentation approaches.

Furthermore, as explained in 2.13.2, to generate sequences conditioned on task-specific requirements, e.g. high classification confidence, data must first be generated, then filtered on the defined conditions. This process expects that enough samples are generated so that the likelihood of retaining a sufficient number of appropriate samples is increased. Since the generation process is independent of other task requirements, e.g. classification confidence, the produced samples will not necessarily contain solutions that maximize these requirements. For instance, a task may require that the generated samples return the lowest classification confidence a classifier exhibits. In this scenario, because classification confidence is measured after the generation phase, it is uncertain if the generated data will contain samples in the regions of the lowest confidence scores. This requires us to generate as many samples as possible to increase the chances of meeting the task conditions.

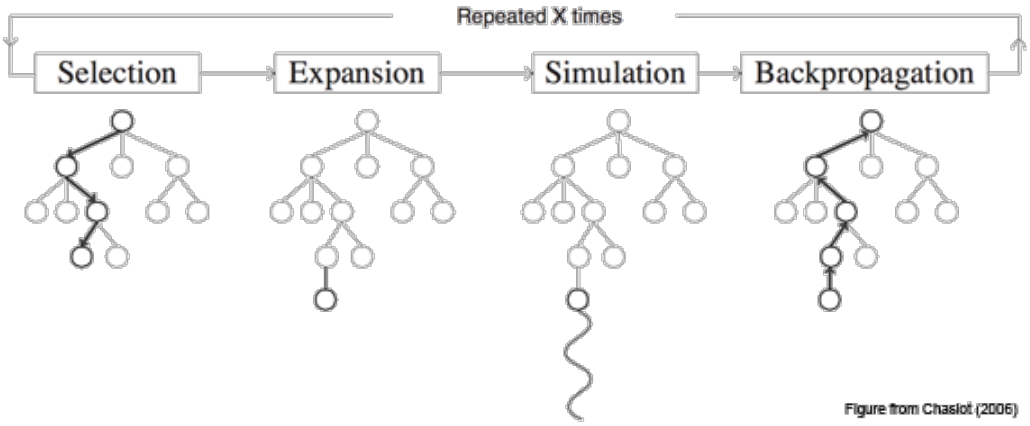


Fig. 2.30 The four stages of MCTS. Source: Chaslot et al. (2008)

Instead of generating data independent of the task requirements, a reward based mechanism can be applied to guide the language generation process. In this setting, the language model is encouraged to generate data that meets the given requirements. For instance, if the task requires generating data in the regions of higher classification confidence scores, the language model can be guided by increasing the reward for samples that are classified with higher confidence. However, aiming for paths that only maximize the reward is likely to lead to less diverse outputs, a result of MCTS getting stuck in subtree branches that return higher rewards. For this reason, algorithms like the Upper Confidence Bound (UCB) (Auer et al., 2002), can encourage MCTS to explore less visited paths while exploiting paths that return the highest rewards. This allows MCTS to find compelling solutions without having to run to completion. It does so by walking through random paths in the search space while constructing a tree using the results of a predefined reward function. Due to its ability to find paths leading to an optimal solution when the search space is infinitely large, MCTS has been widely adopted by the AI gaming community (Arneson et al., 2010; Chang et al., 2016; Perez et al., 2013). The longer MCTS runs, the stronger its moves get. This is because it manages to balance between two main criteria: exploring new search paths and exploiting paths that have been already explored. MCTS consists of four major steps: Selection, Expansion, Simulation, and Backpropagation (Chaslot et al., 2008), shown in Figure 2.30. When applied to board games, MCTS constructs a tree to determine a winning strategy. In this setting, a node represents a board position, an edge represents a move, and a path represents a sequence of moves.

Selection: Starting from a root node R, the algorithm selects a move that leads to a node N that has no identified children. On the one hand, the selected move could be random, ignoring the scores of already visited paths. On the other hand, the selected move could be

completely based on already visited nodes (e.g. by storing average wins for each node); here, the algorithm might miss other nodes that could lead to higher win rates. In order to balance between the benefits of exploration and exploitation, a selection policy such as the Upper Confidence Bound (UCB) can be used (Auer et al., 2002). UCB makes sure that as many nodes as possible are explored, while still favoring branches that are visited more often than their counterparts. The selection is then done by choosing the nodes with the highest UCB value:

$$UCB = \frac{W_i}{S_i} + C\sqrt{\frac{2 \times lnS_p}{S_i}} \tag{2.70}$$

where W_i is the number of simulations generated from node i which resulted in a win, S_i the total number of simulations generated from node i, S_p the total number of simulations generated from the parent node, and C an exploration parameter.

Expansion: In this phase, a new child node is added to the node selected in the previous step. This new node is based on a random selection of one of the possible moves. The values for this new node are initialized to 0 wins out of 0 simulations, where $W_i = 0$ and $S_i = 0$.

Simulation (Roll-out): A simulation is run from the root node R until a terminal node T is found. The terminal node will output a value that is then passed upwards in the backpropagation phase.

Backpropagation: A simulation stops when a terminal node T is reached. The values for each node leading to T are then updated by adding 1 to the number of visits S_i and number of wins W_i.

2.14 Conclusion

In this chapter, we covered the background that is needed for the reader to progress through the remainder of this thesis. We also highlighted other related approaches in data augmentation by discussing their use cases, strengths, and weaknesses. In the next chapters, we discuss our work that attempts to improve on the existing DA methods by first introducing a user-in-the-loop DA approach in chapter 3, then an automated version of this process in chapter 4, and finally a DA method for knowledge distillation (section 5.2.1) that is explained in chapter 5.

Chapter 3

Data Augmentation by Generation

3.1 Introduction

Active learning (AL) is a well-applied approach in areas where unlabeled data is abundantly available, but labels are either scarce or costly to obtain Settles (2009). In AL, a classifier is improved upon through an iterative learning process; at each cycle, a subset of the original dataset with the most informative examples is selected to be labeled, typically by a human expert, before it is then added to the existing training data Settles (2009). Previous active learning research on textual data, to our knowledge, has always assumed the availability of datasets containing large pools of unlabeled data. In cases where the available data is insufficient for active learning, the burden is transferred to the data collection process, where additional data must either be manually created, or collected from real world interactions. One strategy for creating data is to transform existing examples in certain ways in order to produce new data items and hence increase the size of the training dataset. This approach has been applied successfully in computer vision for example, by manipulating existing images while preserving the label to create additional data points Shorten and Khoshgoftaar (2019). However, in NLP, augmenting data is a very difficult task due to the complex nature of language Wei and Zou (2019). In this work, we assume a real-life scenario where the data at hand is insufficient for running a typical active learning algorithm. We introduce a method that enables us to automatically generate artificial text examples that complement an existing dataset. Our approach minimizes the human factor in data creation by automating the process through a guided searching procedure.

Once a set of examples is generated, it is required to be manually labeled before it is added to the original training set. The classifier is then retrained on the new, augmented training set. This procedure is repeated multiple times until either the desired performance is achieved, or performance no longer improves. We do this in active learning cycles, where

only a subset of the generated data is used; this involves selecting examples in terms of how much information they would add to an existing classifier. In our experiments we use entropy as a measure of informativeness.

As our aim is to find the most informative examples, we tackle this problem by applying a search approach. Given that text examples are generated from an extremely large number of possible combinations, we apply the Monte Carlo Tree Search MCTS algorithm Browne et al. (2012) to limit the search space. Here, MCTS is expected to guide a language generation model to output informative examples. In this context, MCTS assigns values to previously generated examples using a scoring function that incorporates the learning classifier of the previous active learning cycle, starting with a baseline classifier trained on the initial dataset for the first active learning run. In our experiments, we test MCTS with two different scoring functions: one that only measures the uncertainty of the generated examples through entropy, and another that combines the measure of uncertainty with a measure of diversity by computing the cosine similarity of every newly generated example with the previous content. These scores determine the text premise that is passed to the language model when generating newer examples.

We compare MCTS to Non-Guided Data Generation (NGDG), an approach where the knowledge of the learning classifier is not involved in the data generation process. Here, for each newly generated example, the text premise is always a token representing the beginning of a sentence, <bos>. The remainder of this chapter is organized as follows: Section 2 provides a background as well as an overview of related literature. Section 3 describes the proposed approach. Section 4 presents the experiments which were carried out. Section 5 gives conclusions and plans for future work.

3.2 Background

3.2.1 Active Learning

In this work we consider the pool-based AL model, a commonly adapted approach in text classification problems Hu et al. (2016); Krithara et al. (2006); Nigam and McCallum (1998); Tong and Koller (2001). This approach assumes the availability of all the data from the beginning of the process. We start with a set of data S_D, where a large pool of it is unlabeled S_U, leaving only a small subset S_L with labels $l_1, l_2, .., l_n \in L$. Hence, $S_D = S_U + S_L$. A classifier is first trained on S_L. Then, at each AL iteration, a selection strategy is applied to select a pool of data S_P from S_U to be labeled by the expert. Examples in S_P are chosen

on the basis of being the most informative of S_U, such that, if added to the training data, an improvement in the classifier's performance is to be expected.

As described by Siddiqui et al. (2019); Yoo and Kweon (2019), there are three main selection strategies that can be applied to obtain S_P: uncertainty-based approaches, diversity-based approaches, and expected model change. In an uncertainty-based selection strategy, the active learner chooses the examples that it is most uncertain about. This assumes a probabilistic framework where the learner predicts a probability distribution $P = (p_1, p_2, ..., p_n)$ for labels $L = (l_1, l_2, ..., l_n)$ for a given example $e_i \in S_U$. In a binary classification setting, Lewis et al. presume that the most uncertain examples have a posterior probability closest to 0.5 for any label $l_i \in \{0,1\} \forall i$ (Lewis and Catlett, 1994; Lewis and Gale, 1994). In a multi-class setting, a selection strategy could choose examples with the lowest posterior probability or be based on entropy (equation 3.2) as in Hwa (2004); Joshi et al. (2009); Settles and Craven (2008). Given that the difference in degree of certainty for similar examples can be small, uncertainty selections are prone to return similar examples Wang et al. (2017). To address this issue, some works incorporate measures to exploit the diversity information of the examples in the selection process Sener and Savarese (2017); Sinha et al. (2019); Wang et al. (2017). Finally, expected-model change selects examples that would cause the greatest change to a model's output if their labels were known Freytag et al. (2014); Roy and McCallum (2001); Settles et al. (2008). This approach however, can be computationally expensive for big data and large feature spaces Settles (2009). Hence, this approach has not been very successful with deep learning models Siddiqui et al. (2019).

In summary, research on active learning has focused on applications where a large pool of unlabeled data already exists. However, we are interested in real-life scenarios where this data may not be available. Other approaches such as Snorkel[1] use heuristics to generate data (Ratner et al., 2017) but this can prove impractical for text. In this work, we consider the case where the number of available data S_D is extremely small, so that typical active learning approaches become inapplicable due to the absence of S_U. Our aim is to generate synthetic data for S_U that can then be queried by an active learning algorithm to select an informative subset S_P for labeling. The selection process we apply can be classed as an uncertainty approach, except for the Diversity-Based MCTS (described in section 3.3.2) which incorporates a similarity check that could also be classed as a diversity approach.

[1] https://www.snorkel.org/

3.2.2 Monte Carlo Tree Search MCTS

As described in section 2.13.3, MCTS is a tree-based algorithm that searches for solutions which maximize a reward function. By applying MCTS, we transform the data generation task into an optimization problem which maximizes the usefulness of the generated output. In this setting, we use MSTS as the optimization strategy and incorporate entropy as one of the optimization criteria.

3.2.3 Related Work

In previous work, Sankarpandi et al. (2019) applied a similar framework to a private dataset, where instead of GPT-2, a recurrent neural network was used to generate words, and the reward function was solely based on entropy. Furthermore, experiments were based on a much larger initial training set, and there was an added burden on the user to manually correct ill-formed generated outputs. In our work, we were able to achieve satisfactory results with the smallest version of the GPT-2 model: 12 hidden layers and 124M parameters. To our knowledge, the next closest work to ours is Anaby-Tavor et al. (2019), where GPT-2 and a classifier are applied to generate new weakly-labeled examples. This process involves fine-tuning GPT-2 on existing training examples while providing the class labels as part of the input. Examples are then selected and kept as training data based on the classifier's confidence score. Kumar et al. (2020) further explores this approach with different transformer-based models (Vaswani et al., 2017) for data generation. This approach however, relies on GPT-2 to provide weak labels as it generates data. It also does not employ a guided search to generate the best examples at a given stage. By excluding the process of generating weak labels, this approach could be considered analogous to our Non-Guided Data Generation method in section 3.4.3.

3.3 Approach

3.3.1 GPT-2 Fine-tuning

To generate relevant text to the target task, the language generation model GPT-2 is better fine-tuned on the dataset of that task. We later show in our experiments (section 3.4) that GPT-2 is able to generate relevant text samples, even when fine-tuned on as little as 5 examples per label. Because GPT-2 is pretrained on unlabeled data, we discard all labels for its training. Consider Table 3.1, where each example is a question matched to a label. Since we only require GPT-2 to generate examples disregarding their labels, we ignore the

#	Example	Label
1	How do doctors diagnose bone cancer ?	DESC
2	Who fired Maria Ybarra from her position in San Diego council ?	HUM
3	What country did King Wenceslas rule ?	LOC
4	How many people in the world speak French ?	NUM

Table 3.1 Examples from the TREC-6 dataset, refer to section 3.4.1

label column. We then need to transform the text examples to a format which matches that of its pretraining data. This can be done by concatenating the text examples, separated by the symbol $< |endoftext| >$, that indicates the start and the end of a new example. This results in the following GPT-2 fine-tuning data for the examples in Table 3.1:
$< |endoftext| >$ How do doctors diagnose bone cancer ? $< |endoftext| >$ Who fired Maria Ybarra from her position in San Diego council ? $< |endoftext| >$ What country did King Wenceslas rule ? $< |endoftext| >$ How many people in the world speak French ? $< |endoftext| >$

The $< |endoftext| >$ symbol helps GPT-2 treat each example as an independent segment by providing the beginning and ending boundaries.

3.3.2 MCTS for Data Generation

In games, MCTS can be applied to predict moves in order to counter an opponent's strategy so that a winnable state is reached. However, text generation is more similar to a single player scenario, where decisions are based on which token to select when moving from one state to the other. A language model calculates a probability distribution over a sequence of words. When passing over a stream of text, each vocabulary token is assigned a probability score for occurring next. Hence, tokens with higher probability scores are more likely to appear next in the sequence. In our setting, we are interested in multiple token candidates for all remaining words in the sequence. To achieve this we use a top-k sampling scheme as used by Fan et al. (2018). At each time step, each token in the vocabulary is assigned a probability score for coming next in the sequence. To get the top k candidates, vocabulary tokens are sorted by their probability scores and anything below the k'th token is then zeroed out. The probability mass is then redistributed among the k token candidates.

This process can be modeled as a tree where each node represents a token linked to k child nodes representing the top k candidate tokens that are likely to appear next in the sequence. Hence, this is similar to a board game where each board position is represented by a node: The root node corresponds to an empty board while a terminal node is where no

further moves can be made. In our setting, we use the token $< |endoftext| >$ for both the root and terminal nodes. For simplicity, in this chapter, we will represent the starting token with <bos> and the ending token with <eos>.

As an example, a language model that is fine-tuned on a survey on pet adoption could be used to generate the tree of predictions in Figure 1. A full version of this tree would represent all the possible combinations of text that can be generated by the language model. In an ideal setting, we would search this tree for the paths that represent the most informative examples. However, given that the tree will grow exponentially as the number of next token candidates is increased, it would be computationally expensive to apply a brute force search algorithm where every path is examined. For this reason, we apply the Monte Carlo Tree Search (MCTS) algorithm in the data generation process, as discussed next.

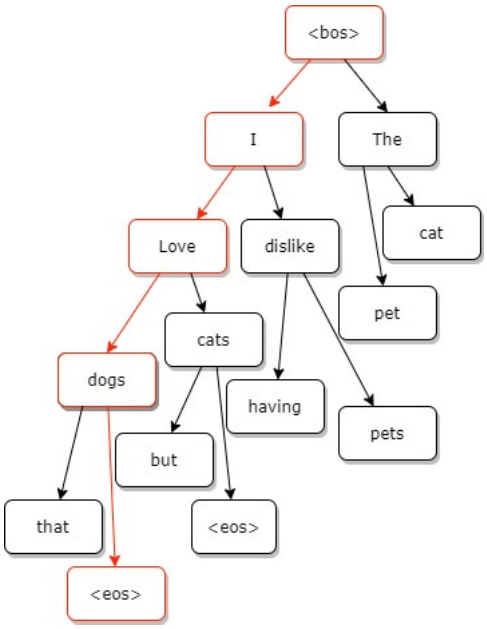


Fig. 3.1 MCTS traverses down the tree as it creates paths spanning from the root node <bos> until a terminal node <eos> is reached. Tokens of the same path form a sentence when concatenated e.g. the path in red

In a typical MCTS application, a separate tree is formed for every decision. Applying this to our approach would require us to build a tree for each next word in the sequence. This would be computationally expensive due to the overhead of generating candidates and computing reward values. An alternative would be to construct a single tree only, while allowing MCTS to run for a longer period. This would result in a tree where each path is a possible output. However, given the nature of MCTS where the paths generated in the roll-out phase are not stored, we would be left with many incomplete paths which did not reach a terminal node (see Figure 3.1). To account for this, we keep track of all the simulations

without impacting the selection policy. Thus we still have the same tree as in Figure 3.1, but we also have a record of the paths generated from non-terminal leaf nodes. The Selection, Expansion, Simulation and Backprogation phases for each MCTS iteration are described in the following sections.

Selection

The vanilla UCB function is mostly adopted in strategies where the outcome is chosen from a fixed set of categorical values, win, lose or tie. The objective is to reach a winnable state with the minimum number of visits. This is reflected in the UCB equation (equation 2.70). By contrast, our purpose is to maximize the importance of nodes that lead to higher reward values (section 3.3.2), as shown in equation 3.1, adopted from Chaudhry and Lee (2018):

$$UCB = max(N_i) + C\sqrt{\frac{2 \times lnS_p}{S_i}} \tag{3.1}$$

where $max(N_i)$ is the maximum reward at node i, C is an exploration constant, S_i is the total number of visits to node i, and P_i is the total number of visits to the parent node for node i.

Expansion

Once a node is selected, we add all its immediate child nodes. These are the allowed moves from a given state, that is the top k token candidates generated by a language model, given the state's context history. Figure 3.2 illustrates the process. For $k = 3$, the context history for the state at the root node is the token <bos>. When passed to a language generation model, the words "Where", "What" and "Who" are examples of the top three candidates to follow <bos>. Assume the node "Who" is picked in the selection phase at $i = 1$. Its state context history "<bos> Who" returns the candidate tokens "discovered", "is", and "invented"; these are added as child nodes in the expansion phase.

Simulation (Roll-out)

A simulation starts from the added child node in the expansion phase. During this process, a sequence is generated by picking at random a possible candidate for each next token until a terminal state is reached. Given that candidate tokens are generated over a probability distribution, we apply a weighted-choice method, enforcing non-uniform randomness. As explained earlier in section 3.3.2, simulations are tracked without affecting the growth of the tree. Taking this into account, we are able to modify the value of k for the tokens, without

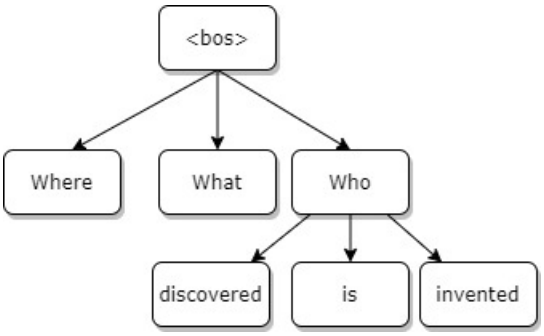


Fig. 3.2 A possible MCTS output after 2 iterations

affecting the functionality of MCTS. We will refer to this as K_s to distinguish it from the K in the expansion phase. If $K_s >> K$, we are able to achieve higher variance in the generated data while maintaining the width of the tree.

Backpropagation

Once an example is generated, its reward value is computed. The path of the expanded node is then updated by backpropagating the reward value and increasing the number of visits by one. For the reward function, we implement two variants of MCTS, hereafter referred to as Uncertainty-Based MCTS and Diversity-Based MCTS, where the only difference is in the reward function.

In **Uncertainty-Based MCTS**, given the learning classifier's softmax probabilities over the possible class labels, we compute the normalized form of Shannon's entropy as shown in equation 3.2:

$$H_n(P) = -\sum_{i=1}^{n} p_i \log_b p_i \cdot \frac{1}{\log_b n} \tag{3.2}$$

where P is a set of probabilities $P = \{p_i; i = 1, ..., n\}$, with $\sum_{i=1}^{n} p_i = 1$ for n labels, normalized by $log_b n$. We expect meaningless content in regions of higher entropy, and so limit the search space to a predefined value for maximum entropy, θ_{ent}. Examples with an entropy above this threshold become less important by returning a lowered reward value (e.g. 0), as shown in equation 3.3.

$$f(x_{ent}) = \begin{cases} 0, & \text{if } x_{ent} \geq \theta_{ent} \\ x_{ent}, & \text{otherwise} \end{cases} \tag{3.3}$$

where x_{ent} is the entropy value for example x, and θ_{ent} the entropy cut-off threshold.

In **Diversity-Based MCTS**, in addition to entropy, we compute the cosine similarity between each generated candidate and a comparison list initialized with the classifier's training data. This is to ensure the diversity of the generated examples. If the similarity score

is above a certain threshold θ_{sim}, the reward for the candidate will be set to 0, as shown in equation 3.4. Conversely, if it is below θ_{sim}, the candidate is added to the comparison list, so that future candidates will be penalized if they are too similar to it:

$$f(x_{ent}, x_{sim}) = \begin{cases} 0, & \text{if } x_{ent} \geq \theta_{ent} \\ 0, & \text{if } x_{sim} > \theta_{sim} \\ x_{ent}, & \text{otherwise} \end{cases} \tag{3.4}$$

where x_{sim} is the maximum cosine similarity score between example x and the comparison list, and θ_{sim} is the cosine similarity threshold.

3.3.3 Data Selection and Active Learning

Once MCTS reaches completion, all leaf nodes from the final tree are selected. Given that we have kept track of the generated simulations from a node, each non-terminal leaf node is now linked to a generated sequence of text. The final set of text examples is then sorted by the values from their corresponding nodes. The top n examples are selected, labeled by hand and appended to the original training set. We then retrain the learning classifier on the new dataset.

3.4 Experiments

3.4.1 Datasets

In our experiments, we attempt to emulate real life scenarios where training data is scarce. So from each dataset below, we create an initial training set by randomly selecting a very small subset of the available training data. We then fine-tune GPT-2 on the created subset and use our method from section 3.3 to generate new training examples. Once data is generated, we label the top n examples, sorted by the max reward value, as described in section 3.3.3.

We study the effectiveness of our methods on two different tasks, question classification and sentiment analysis.

Question Classification: For this task we use the 6-label version of the TREC Questions dataset, TREC-6 (Li and Roth, 2002). TREC-6 divides questions into 6 categories: HUM, DESC, ENTY, LOC, NUM, and ABBR. From the available training data, we randomly select only 5 examples per label, making a total of 30 examples for training the baseline classifier. Evaluation is done over the provided test set of 500 questions.

Sentiment Analysis: For this task we use the Stanford Sentiment Treebank SST-2 Dataset (Socher et al., 2013), with sentiments divided into 2 labels, positive and negative. We use the data split from the GLUE SST task (Wang et al., 2018a), and evaluate on the provided development set. From the available training data, we only select 10 random examples per label and discard the rest.

3.4.2 Baseline Approach: Non-Guided Data Generation (NGDG)

Unlike MCTS, NGDG is a data decoding strategy that does not optimize for a reward value. Similar to MCTS, NGDG applies a top-k sampling procedure to generate candidate tokens. However, unlike MCTS, the selection of the next token is entirely based on the distribution of the candidate tokens. This is exactly the same procedure as the simulation phase in MCTS, but instead of constructing a tree search, simulations are run independently of one another. In our proposed framework, in section 3.3, NGDG replaces the process that generates sequences with one that does not control the outputs of the language model. For any data-generation method, as explained in section 3.3.3, the following stages are essential to the proposed framework:

1. Train a base classifier CLS on the labeled seed data D_L.
2. Use the trained CLS to create a set of synthetic data by applying the MCTS steps from section 3.3.2. Since MCTS is a search-based decision-making algorithm, see section 2.13.3, it can be used with any DA method, e.g. EDA, Random word-replacement, text generation, and back-translation, explained in section 2.1.4.
3. Arrange the generated samples by their reward value (entropy) in descending order.
4. Select a subset (transfer set D_T) of top n examples for a user to review and edit any incorrect labels.
5. Append the manually reviewed data examples to DL. that is, $D_L = D_L + D_T$.
6. Retrain CLS on D_L.
7. Repeat steps 2-6 until a desirable test performance is reached.

It is evident that step number 2 is the core of our proposed data-generation approach, which is complimented by steps 3-7 that are parts of an active learning process, explained in section 3.2.1. Thus, the best way to create a comparable baseline, is through an alternative generation process to the one mentioned in step number 2. Using a DA method other than

GPT-2 for synthetic data creation would not be enough to alter the proposed approach, as a generation process would still be needed to create a set of synthetic data. For instance, if GPT-2 is replaced with a DA method like EDA and MCTS is kept, the generation process will remain the same, and only the quality of the generated data might differ. However, since MCTS is the core of our proposed method, replacing it with another well-established decoding strategy makes a more appropriate baseline. We note that random sampling (Ippolito et al., 2019), which what NGDG is based on, makes a less restrictive decoding strategy. For this reason, we use NGDG, explained in section 2.13.2, as a DA baseline to MCTS. For a better and more controlled comparison, we keep unchanged all the other variables like GPT-2 and the use of entropy for reward computation.

3.4.3 Model Comparison

In our experiments we compare two variants of MCTS with only a minor difference in the reward function, one with the effect of θ_{sim} as described in section 3.3.2, and one without its effect. We further test the effectiveness of MCTS for data augmentation by comparing it to a decoding strategy that does not optimize for a reward value, which we refer to as the Non-Guided Data Generation (NGDG) approach, explained in section 3.4.2.

To emulate the flexibility of having higher variance over the latter parts of the generated text in MCTS (section 3.3.2), we increased k for the number of candidate tokens after the first n output tokens in the sequence. Here n is fixed at 3 in all our experiments. After the data is generated, we apply the classifier of the previous active learning cycle to compute an entropy value (equation 3.2) for each example. Data is then sorted by the entropy, and the top n examples below θ_{ent} are then selected for labeling. The classifier used for our experiments is a relu layer neural network with the Universal Sentence Encoder (USE) for the embedding layer. We implement this classifier using the Keras[2] toolkit. The classifier contains an embedding layer with 512 neurons, a 600-neuron fully-connected dense layer, a dropout layer with a 0.2 dropout rate, and a softmax activation output. It is optimized using Adam (Kingma and Ba, 2014). We fixed the classifier's hyper-parameters following a hyper-parameter search to a batch-size of 2, 0.0001 learning rate, and trained over 15 epochs.

3.4.4 Data Generation Parameters

We fix the MCTS UCB policy constant C to 2, θ_{ent} to 0.95, and θ_{sim} to 0.9 for all experiments. To achieve fairness in the comparison, when using the NGDG method (section 3.4.3), we discard examples with entropy above $\theta_{ent} = 0.95$ in the experiments.

[2]https://keras.io/

3.4.5 Data Selection

For MCTS, as the learning classifier is part of the data generation process, the output examples are already mapped to their reward values and to the classifier's predicted labels. For NGDG, however, because the classifier does not take part in the generation process, it must be applied to the generated data afterwards, to predict labels and compute values for entropy. After classification, the data is sorted by entropy (for NGDG) or reward value (for MCTS), and the labels of the top n examples are corrected manually. Finally, to limit the effect of an imbalanced dataset, we restrict the number of the selected examples x_{max}, to the first 10 per label. In the event where all labels have more than 10 examples, x_{max} corresponds to the count of the label with the least number of examples.

3.4.6 Experiment 1: TREC-6 Question Data

For this experiment, we fixed the number of simulations at 3000 and the top n examples for labelling to 50 for both MCTS and NGDG. We set the number of candidate tokens K to 6 and K_s to 20. For NGDG, K changes from 6 to 20 after the first 3 tokens are generated from the sequence. Table 3.2 shows the average accuracy achieved over the 6 labels throughout 8 Active Learning runs on the TREC-6 test set, as well as giving the added number of examples after each AL cycle.

AL Run	MCTS		NGDG
	Diversity	Uncert.	
Start	65 (30#)	65 (30#)	65 (30#)
1	68 (48#)	78 (49#)	78 (47#)
2	86 (68#)	82 (52#)	86 (61#)
3	92 (73#)	87 (55#)	87 (72#)
4	91 (76#)	89 (59#)	88 (83#)
5	92 (83#)	91 (71#)	86 (89#)
6	91 (91#)	90 (76#)	84 (103#)
7	90 (94#)	89 (87#)	84 (113#)
8	91 (98#)	90 (94#)	88 (126#)

Table 3.2 Classification results after each Active Learning (AL) run for the TREC-6 question classification task. Before AL, 30 training examples result in 65% classification accuracy. After AL 1, under Diversity-Based MCTS for example, 18 new examples are added (total 48#), giving 68% accuracy, while under Uncertainty-Based MCTS (Uncert.), 19 new examples are added (total 49), giving accuracy 78%. The rest of the table is analogous

3.4.7 Experiment 2: SST-2 Sentiment Data

Similar to experiment 1, we kept the number of simulations at 3000 and top n at 50 for both MCTS and NGDG, whereas we set the number of candidate tokens $K = 15$ and $K_s = 30$ for MCTS. In NGDG, K changes from 15 to 30 after the first 3 tokens are generated. Results are in Table 3.3.

AL Run	MCTS		NGDG
	Diversity	Uncert.	
Start	73 (20#)	73 (20#)	73 (20#)
1	74 (34#)	77 (34#)	69 (32#)
2	79 (41#)	76 (44#)	72 (43#)
3	79 (50#)	78 (48#)	75 (55#)
4	80 (60#)	80 (54#)	76 (79#)
5	80 (65#)	80 (55#)	75 (92#)
6	80 (79#)	80 (62#)	76 (103#)
7	83 (87#)	80 (64#)	79 (116#)
8	83 (95#)	79 (69#)	78 (124#)

Table 3.3 Classification results after each AL run for the SST-2 sentiment analysis task with top $n = 50$

3.4.8 Experiment 3: SST-2 Sentiment Data

We repeated experiment 2 with the same configurations, except that top n, is now 20 (not 50) for both MCTS and NGDG. Results are shown in Table 3.4.

AL Run	MCTS		NGDG
	Diversity	Uncert.	
Start	73 (20#)	73 (20#)	73 (20#)
1	77 (26#)	72 (24#)	68 (30#)
2	74 (29#)	74 (27#)	75 (41#)
3	78 (37#)	74 (34#)	77 (49#)
4	79 (43#)	73 (38#)	76 (56#)
5	80 (46#)	76 (39#)	81 (60#)
6	80 (49#)	78 (42#)	81 (64#)
7	81 (52#)	76 (44#)	79 (72#)
8	81 (57#)	78 (45#)	80 (77#)

Table 3.4 Classification results after each AL run for the SST-2 sentiment analysis task with top $n = 20$

#	Example
1	Why did Einstein lose a fight with cancer?
2	Why did Lincole Ljungberg retire?
3	Why was Lorne L. Huntington's IQ so low?
4	What are three fundamental principles of socialism?
5	What is D.C.'s major metropolitan area?
6	When was Antarctica formed?
7	When did animals roam the earth?
8	Where can a geologist find fossils?
9	Where can an electrician find work?
10	How did Moses rule the ancient tribes?
11	How often have animals been killed by car crashes?
12	Which is Fordham's largest engineering college?

Table 3.5 Some examples generated on TREC-6 through the Diversity-Based MCTS for experiment 1

3.5 Discussion

Table 3.5 shows twelve sentences generated by the Diversity-Based MCTS. These can give us insights concerning our approach and the role of GPT-2 in it. First, consider example 1 in the table ("Why did Einstein lose a fight with cancer?" – type DESC). In the initial training set, there is only one mention of Einstein ("What was Einstein's IQ?" – NUM), and one of cancer ("How do doctors diagnose bone cancer?" – DESC). Nevertheless, example 1 combines information from two different sentence types NUM and DESC in a coherent way. Example 3 again demonstrates a form of 'cross-type' learning: The Einstein training sentence above is the only mention of IQ and is of type NUM. Yet example 3 is a well-formed DESC sentence. For example 4, perhaps the most related training instances are "What are the four elements?" and "What are the chemicals used in glowsticks?". These are asking for lists but concerning elements and chemicals, not abstract concepts like socialism.

Interestingly, even though the training set contains no 'When' sentences, examples such as 6 and 7 could still be created; because MCTS pushes GPT-2 to generate novel sentences as it constructs the tree, those of the form "What kind, when..." are created during the path traversal process. These were then corrected during the labeling stage. We did not witness this phenomenon with NGDG, possibly because MCTS is directed by a reward function that penalizes sentences of low entropy. This allows MCTS to search through the space of possible sentence combinations more efficiently.

Concerning the LOC examples 8 and 9, the only 'Where' training question is "Where do hyenas live?". Yet, in our experiments, we were able to expand on this by generating

additional 'Where' questions which are very different from the hyenas and different from each other: A fossil is something which a geologist might find, while work is something which an electrician might find. Both are meaningful, while the sense of 'find' in each is quite distinct. Finally, while the remaining examples in Table 5.3 could not be directly linked to relevant examples in the training data, this only confirms our purpose of using a pretrained model like GPT-2 that can make use of its external knowledge while remaining relevant to the target task.

In summary, by integrating GPT-2 with our methods, we gained substantial improvements over the baseline classifier. This shows how text generation can improve performance for tasks with scarce data. Even when starting with just a few examples per label, we were able to generate informative data that boosted the accuracy on TREC-6 from 65% to 91% with MCTS and 88% with NGDG, on SST-2 from 73% to 83% and 78% respectively, after 8 AL runs. Even when reducing the number of examples for labeling from 50 to 20 in experiment 3, we were still able to achieve an improvement of 81% with MCTS and 80% with NGDG. This suggests the effectiveness of our approach in solving real-world classification tasks when minimal data is available. Moreover, with MCTS we witnessed improvements in performance compared to NGDG on both the TREC-6 and SST-2 datasets. MCTS guides the growth of the tree by visiting more relevant nodes more frequently. Hence, relevancy is increased by the paths that maximize the reward function, those that correspond to high entropy values in our setting. However, searching only for high entropy is more likely to incur noise in the final output such as ill-formed sentences or content that does not fall under the labeling criteria. Since ill-formed sentences are likely to incur high entropy values, the lack of a sentence quality measure can make MCTS prone to output meaningless sentences. For instance, "What kind!!??", "Which is the abbrev?", and "What does IQ be?" were outputs of MCTS in the TREC-6 experiments. This point is reflected in the lower overall number of added examples when comparing MCTS to NGDG over the 8 AL runs. Moreover, when MCTS over-exploits visited paths, it can get stuck in certain sub-trees, leading it to output examples with a high level of similarity. For instance, "good movie" and "good movie!" are identical examples with the only difference being the exclamation mark '!'. This issue is especially noticeable in the MCTS Uncertainty-Based experiment in Table 3.4, where due to the number of closely similar examples in the output, a lower proportion of the top 20 examples could be labeled. Hence, to diversify the generated output, we introduced θ_{sim} in the MCTS Diversity-Based approach.

Overall, the success of our approach relies on the quality of the search space, which is determined by the language model; if it performs less well, this can result in a noisier space. For instance, Sankarpandi et al. (2019) could not achieve comparable results as they had

used an inferior LSTM-based language model to generate words. Moreover, additional user involvement was needed to make sense of ill-formed outputs, making the whole approach laborious and more prone to the user's bias.

3.5.1 Transfer Set

In Table 3.6 we show the 64 examples that were generated and accepted by the user, after 8 active learning cycles for the Uncertainty-based experiment from section 3.4.6. In Table 3.2, we show a total of 94 training samples after 8 active learning cycles, this includes the initial 30 train samples from the TREC-6 dataset, and the 64 samples from Table 3.6. Although most of the examples in Table 3.6 are well-formed, they are not necessarily factual. For instance, the question "Why are there more than 1,000 different species of sea turtle?" is well-formed and grammatically correct, however, it implies the existence of at least 1,000 types of sea turtles, which is a false statement. Another noticeable phenomenon is the creation of fictional characters, as in the example "Why does Gwyn Waverley live in South Wales?", or "Why does Waringka have an orange-and-white beaver?". Here, the names "Gwyn Waverley", and "Waringka" were purely made up by the language model. Other non-factual instances include the misuse of commonly known terms or concepts. For instance, the question "What does Luddism look like?" entails that Luddism is a tangible object, but this word refers to an ideological movement against technology (Jones, 2013). "What are three dimensions of the Earth?" is another example, where the model makes misuse of a word: earth. This question would have made more sense if it was about the dimensions of space, which are length, width, and depth. Considering that GPT-2 was pretrained on data collected from the internet, it could be expected for the model to associate "earth" and "space", as they could exist in similar contexts. Despite the fictional statements or incorrect assumptions the model sometimes makes, the data it generates can still be useful for improving performance in TREC-6. This is because questions do not need to be factual to be classified into any of the TREC-6 labels; "DESC", "NUM", "ENTY", "LOC", "HUM", and "ABBR". For instance, although the question "Why are there more than 1,000 different species of sea turtle?" makes an invalid statement, it can correctly be categorized into "DESC" in the TREC-6 task. Since the training function of language models like GPT-2, refer to section 2.1.3, does not penalize for the truthfulness of the generated outputs, the trained model is bound to create fictional or nonfactual statements. To account for truthfulness, appropriately annotated training data will be needed. However, fact checking is a separate field of research in the NLP domain and is beyond the scope of this thesis (Guo et al., 2022; Zeng et al., 2021). In fact, since truthfulness is not important for TREC-6, fictional statements can further diversify the training data. This

can also be true for other classification tasks like sentiment analysis, where a model is only evaluated on its ability to classify the polarity of a given text, and not its truthfulness.

While most questions in Table 3.6 are grammatically correct, there can be instances that fail to meet this standard. For example, the questions “What is the D.C. office of the Office of Congressional Counsel?”, and “Where does the word word "georgia" come from?” contain repetitions of the words “office” and “word” respectively. This repetition could be a result of using the top-k sampling decoding strategy. In a peaked distribution, as explained in 2.12, the probability mass would be concentrated in a few tokens, in which for a relatively large k, less probable tokens are likely to be selected. When the language model predicts a repeated token with a probability close to the probability mass of a peaked distribution, a large enough k would result in its inclusion for the next token candidates by top-k sampling.

Example	Example
What does Luddism look like?	Who does Gerson belong to?
What is the name of the company that produces the Spumante?	What is Lincolns number?
Where can scientists find fossils of dinosaurs?	When was D.C. the most populous city in the country?
What do I stand to gain by selling a company’s stock?	how did animals get their teeth?
Where can scientists learn to grow an onion?	Which is the Australian capital?
how far is South Florida’s ocean?	Where are these turtles?
What does IQ mean in America?	What do animals live under?
What is a company’s tax exempt status?	Where does the word micro come from?
What does NAFTA look for?	What do animals eat?
Which is Australia‘S only trading partner?	Where does the word word "georgia" come from?
What are some terms that describe hormones?	What is L.T.?
What is November Folsom	Which is November’s favorite sport?
when did I first know that I was a girl?	Where does Gedneystuck come from?
What do I eat, drink, and breathe?	Why does Gwyn Waverley live in South Wales?
Where can a coal company be found?	What do animals eat and drink?
What are three dimensions of the Earth?	Why do scientists find such strange and bizarre features?
What is L.A.’s largest office?	What is L.A’s most-used street name?
What does NAFTA represent?	Where does the word for an office come from?
Which is the Australian dollar or the pound?	Where does the word "giant" come from?
Which college is the best?	Why do so few people know the chemical makeup of human hair?
What is the D.C. office of the Office of Congressional Counsel?	Why are a wide number of these animals covered by thick, dark, and/or black fur?
Why did Einstein become the first black-and-white scientist?	Which company is Alphabet?
What do doctors believe are the most common bone cancer drugs?	Which is North Carolina North Korea’s only language?
Why does Waringka have an orange-and-white beaver?	What does NAFTA represent to U.S. companies?
Where can scientists learn about life on Earth?	Where did Freud first find psychoanalysis?
What does NAFTA represent, other than North America?	What country does GAWK stand for?
What do I live under?	How did Einstein be able to read?
Who does Gerson work with?	Why are there more than 1,000 different species of sea turtle?
What is Dorshow?	What do I live in?
What do doctors prescribe for bone cancer?	Where does the word for "geese" come from?
What do I eat?	Where does the word ’vodou’ come from?
What do microbe, nematode, and black widow eggs have in common?	What do I do with my life?

Table 3.6 Transfer set samples from experiment 3.4.6 after 8 active learning cycles for the Uncertainty-Based MCTS

3.5.2 Ethical Concerns

With the recent advances in machine learning and computational resources, the horizon of possible applications has been extended. As Natural Language Generation (NLG) models

become more scalable, the quality of the text they produce has been shown to increase; consider the leap in performance from GPT-2 (Radford et al., 2019) to GPT-3 (Brown et al., 2020). Currently, language models have improved to the extent of being capable of synthetically creating close to human-quality text. Despite the real-world benefits they can bring to humanity, these models can be exploited by bad actors. This raises ethical questions concerning the use of language generation models for questionable practices. We can break down the ethical impacts of language generation into two main categories: intentional and non-intentional threats.

Intentional threats occur when bad actors take an initiative to abuse a system in harmful ways. With the recent advances in NLG, it has become possible to build models that excel in applications such as text summarization, question answering, translation, or even the generation of code, novels, news articles, etc. It has been shown that such applications can bring valuable benefits to the end user. For instance, by training language models to generate code from a user's prompt, software development would become accessible to the non-technical and could even become easier to the technical user. In this example, the prompt could be a description of the requirements for the desired code written in natural language. Chen et al. (2021) has shown that acceptable results can be achieved by fine-tuning a language model such as GPT-3, explained in section 2.3.2, on documented code. Although such applications can be beneficial in many cases, they can also be abused by malicious actors. For instance, advanced improvements in code generation could help less technical people to create malware and software that could facilitate phishing attacks. One example of a phishing attack is the creation of a log-in webpage that, on the front-end, looks very similar to an existing legitimate service, but in the backend hijacks the credentials that a victim user inserts. Online users that are led to malicious websites could become victims of fraud, blackmail, identity theft, etc. When less technical users are able to create such malicious websites, potential online crimes could grow on a large scale. Other intentional misuses of NLG models could be the generation of personalized human-sounding text. For instance, governments or parties with political motives could abuse NLG models to persuade the masses by spreading fake text or propaganda on social media. Without automatic text generation, such applications would still be possible, but would require manpower that could be costly and hard to organize. However, with the help of NLG models, the need for human labor would be minimized. To make matters even worse, as pretrained language models improve at generating text, less effort might be needed to utilize them. For instance, GPT-2, the predecessor of GPT-3, requires hours of fine-tuning on the appropriate data to intentionally bias the model towards producing propaganda. In the case of GPT-3, McGuffie

and Newhouse (2020) showed that less effort was needed to radicalize the model, as only a few training examples were sufficient for producing convincing propaganda (Chan, 2022). Just like with all new technological applications, it is important to have public awareness and understanding of the harmful applications of language models. For instance, there is growing public awareness concerning the importance of private data protection, because concerns were publicly raised against the ability of data collectors, like social media platforms, to manipulate their users by building statistical models using their data.

Unintentional threats could result from a model's biases towards certain aspects learned from the training data. In the case of pre-trained language models, where web-scraped data makes a large proportion of the pre-training data, it becomes difficult to limit all types of harmful biases. For instance, a popular corpus that has been used to train language models like GPT-3 is the Common Crawl dataset; a collection in the magnitude of petabytes of online data that has been collected over 8 years of web crawling. Bender et al. (2021) argues that the size of the dataset does not necessarily guarantee diversity, especially when it includes over-represented samples. The authors claim that due to the unequal distribution of internet access, the majority of online text is from English-speaking developed countries, in which the main contributors are of younger generations. This means that training data is skewed towards the opinions and views of white supremacy, sexism, and ageism (Bender et al., 2021). For instance, GPT-3 was shown to produce anti-Islamic outputs which correlate Muslims with violence (Chan, 2022). Abid et al. (2021) has shown that when GPT-3 is presented with the prompt "Two Muslims walked into", 66 out 100 times, the model generates text completions that contain violence by mentioning phrases such as "threw chairs", "shooting", "bombs", "harass", etc. Abid et al. (2021) saw that when the word "Muslims" is replaced with references to other religions, the tendency of GPT-3 to mention violence drops significantly. In our experiments, during the initial stages of fine-tuning GPT-2, we witnessed a few instances in which the model produced discriminative text. For instance, in the TREC-6 experiments, GPT-2 generated the following sentences:

- How much would black market slaves be worth, Thomas Jefferson
- How much would an African-American be worth?
- How much can an African-American be worth?

The closest sentence from TREC-6 seed data, on which GPT-2 was fine-tuned, is: "How much would a black-and-white 1-cent stamp be worth , Thomas Jefferson on it , ?". It may seem that the words "black", and "white" were associated with African Americans

and the American history of slavery. This is because, the words “black” and “white” are likely to have also appeared in contexts mentioning the African slavery. Similarly, ”Thomas Jefferson” is the third president of the United States of America that happened to rule during the days of slavery. Hence, it is likely that his name appears in the pretraining data of GPT-2 in contexts discussing the history of slavery. Nevertheless, it may not be clear that the sentences mentioned in this example were produced as a result of GPT-2 being trained on text containing racism. In fact, it can be said that in this particular case, the pretraining data of GPT-2 may have included topics and discussions about the history of slavery with mentions that overlap with words in the TREC-6 example. As the relevant TREC-6 question is about the cost or value of an item, it appears that the language model had kept this as a semantic constraint while associating the other parts of the question with slavery. In general, when learning contextual representations, the model learns to associate tokens with the concepts they appear in. Unfortunately, in this specific case, the associations made by GPT-2 led to the generation of text that can be described as racist. Inspired by the experiments made by Abid et al. (2021), we searched $149,874$ generated sequences from different SST-2 experiments for the word “Islam”; out of 41 matches, 27 were about war, "Islamic propaganda”, terrorism or with mentions of terrorist groups. Names of other religions like Christianity and Judaism did not return any matches, but "Christian" returned 9 matches that did not include associations to violence or terrorism, and the word "Jewish" returned only one match that mentions anti-Semitism. Considering that the SST-2 is a sentiment analysis task, at least 8 of the seed examples that were selected for fine-tuning GPT-2 were obvious movie reviews. The representation of Islam in western movies as Shaheen (2003) describes is “led to believe that all Arabs are Muslims and all Muslims are Arabs”, and portrays “ Arabs as heartless, brutal, uncivilized, religious fanatics through common depictions of Arabs kidnapping or raping a fair maiden; expressing hatred against the Jews and Christians; and demonstrating a love for wealth and power.”. Thus, it is possible that during its pretraining GPT-2 had learned these associations from movie-related data, e.g. movie scripts, discussions, or reviews. Unfortunately, in this case, GPT-2 will reflect the dominant online views in its learning. As discussed earlier, whether these views are discriminative or not, they are likely to be reflected in outputs produced by the final model. In one perspective, the trained model can be thought of as a statistical summary of its training data. Hence, to prevent such biases from influencing the trained model, the training data could be altered in two different ways. A) Eliminate discriminative biases by removing samples that contain them from the training data. However, for large datasets, this could be a difficult task. B) Include data that is better representative of the misrepresented groups. This can only be achievable if the right data sources are available.

3.6 Conclusion

In this chapter, we proposed a framework for improving a classifier's performance with synthetic data. We have shown in our experiments that even when starting with just a few examples, we are able to achieve noticeable improvements. We believe this approach is likely to work for any domain or language, so long as the language model is able to generate meaningful output. In this work for instance, we did not need more than 20 examples to fine-tune GPT-2 for the SST-2 experiments, or 30 for the TREC-6 experiments. We expect even better results when more examples are provided or with the application of an improved language model. In the next chapter, 4, we extend our approach by introducing new elements to the classifier and reward function in an attempt to automate the learning process. In doing so, we remove the human-labeling factor.

Chapter 4

Self-Learning through Data Generation

4.1 Introduction

In chapter 3, we showed that it is possible to improve the performance of text classifiers with synthetic data. Our approach mainly relied on external knowledge drawn from the user through the data labels he or she assigns. In general, as seen in chapter 3, external knowledge plays a key role in improving the target classifier. This is especially noticeable in paradigms such as transfer learning and knowledge distillation, refer to chapter 5.

In this chapter, we attempt to recreate the approach from chapter 3 while eliminating the obvious element of external knowledge that stems from the user's labels. Here, instead of relying on user provided labels, the learning classifier is retrained on its own predictions. This means that synthetic examples are labeled by the classifier itself, a process known as pseudo labeling. In this way, the classifier's learning process can be fully automated, and consequently help reduce labor costs and become an alternative when user labels are hard to come by, e.g. when they require domain knowledge. This procedure is known as Self-training (Scudder, 1965), which is one of the earliest semi-supervised learning approaches that works to improve classification by leveraging unlabeled data. In self-training, a base model (teacher) is trained on the available labeled data and then used to pseudo-label the set of unlabeled data. Another instance of the base model (student), is then trained on the original training data and the unlabeled data with the teacher's pseudo-labels. In this setting, the student model is expected to learn by leveraging knowledge from its teacher. Traditional approaches in self-training do not account for the teacher's uncertainty on the pseudo-labeled data (Mukherjee and Awadallah, 2020). This may result in reinforcing labeling mistakes made by the teacher. Bengio et al. (2009) studied a method known as Curriculum Learning, which considers the classifier's confidence in its predictions. Curriculum learning is inspired by the way human beings learn new complex tasks. By learning from simpler examples,

a human can leverage the acquired knowledge to learn more complex tasks. In the same way, a classifier that is trained on simpler tasks can use the gained knowledge to solve more complex ones. When applying curriculum learning to classification models, the complexity of an example can be estimated by measuring the learning classifier's confidence. We define confidence as the highest probability a classifier assigns to a target label for a given input example. Hence, the simpler an example is to a classifier, the higher the confidence that it exhibits. Inspired by self-training, in this chapter, we attempt to improve classification accuracy by training a classifier on synthetic data using its own pseudo labels. In section 4.2 we cover related work to self-learning in NLP. In section 4.5, we explain the proposed methods for self-training on small datasets. We then experiment with the suggested methods in section 4.6.

4.2 Background

We concluded from chapter 3, that a classification model can be improved when trained on data that provides additional information related to the target task. We have seen that such data samples can be detected by measuring the model's response. The element of surprise, known as Entropy, from equation 3.2, has proven successful for detecting informative samples from unseen data. However, it is not unlikely for a model to incorrectly predict outputs with high confidence for unseen data. In fact, previous studies have shown that deep neural networks, although achieving competitive accuracies, are inclined to generate overconfident predictions (Guo et al., 2017; Hein et al., 2019). This complicates the problem of sampling pseudo-labeled data for self-training. Self-supervised learning is concerned with improving the performance of classification models by utilizing unlabeled data. In this setting, knowledge is extracted from an unlabeled dataset and transferred to the learning model. This process enables the student model to improve its learning on target tasks with limited labeled data.

4.3 Problem Statement

In chapter 3, the generated data is sorted by entropy, where the more uniform a predicted probability distribution is, the higher its corresponding sample is ranked. A user is then given the task to decide whether a sample's high entropy (U) is a result of an aleatoric uncertainty (AU) or an epistemic uncertainty (EU), refer to section 2.11. Hence, the total entropy of a prediction can be represented by the summation:

$$U = AU + EU \tag{4.1}$$

Here, samples that are ranked high as a result of aleatoric uncertainty, where noise is in the data, can be disregarded by the user. This leaves the user with samples of high epistemic entropy to be reviewed for the correct labels and appended to the classifier's training data. As epistemic entropy is correlated with lack of classification knowledge; the higher the entropy, the more uniform the predicted probability distribution will be. For instance, in a binary classification task, if a sample is predicted with a probability of 0.5 for each label, the classifier's entropy according to equation 5.5 will be equal to 1. This indicates that the classifier has responded with the highest level of confusion. Refer to section 2.11 for more on uncertainty estimation. By manually correcting and adding misclassified examples to the classifier's training data, we are expected to see improved performance, as shown in section 3.4. This process proves the usefulness of an external knowledge source in the training process. When the role of the user is substituted by an automated process, two main issues can arise: a) With the lack of an external knowledge that distinguishes between epistemic and aleatoric uncertainties, the classifier is put at risk of being trained with noisy samples. b) When a classifier is trained on its own pseudo-labels, mistakes can be reinforced. Hence, if an overwhelming portion of the sampled training data contains incorrectly labeled examples, the classifier's performance is at risk not just of not improving, but also of degrading.

Furthermore, samples predicted with high confidence are less likely to be beneficial to the classifier's learning, while predictions with low confidence are more likely to be uncertain or incorrect. It may be intuitive that training a classifier on a majority of incorrect labels is unlikely to improve its performance. Considering that a softmax-activated neural network will still predict a label for task-irrelevant inputs, this complicates the task of distinguishing between aleatoric and epistemic uncertainties. One proposed solution is to train a Bayesian Neural Network (BNN), instead of a standard neural network classifier. The main difference between the two networks is in the way weights are learned. In a standard neural network, the objective is to learn values for weights and biases that best fit the data. This can be done by a Maximum Likelihood Estimation (MLE), or a Maximum A Posteriori (MAP) where a regularization is used. A standard neural network can be relatively easy to train or deploy when using modern hardware. However, one main issue with these networks is in their explainability, as they can produce overconfident predictions on out-of-distribution data. As such, when making predictions, it becomes difficult to differentiate between the epistemic uncertainty and the aleatoric uncertainty.

One way to tackle the issue of identifying epistemic uncertainty in neural networks, is with the application of stochastic neural networks. Such networks include stochastic weights

or activations, such that the outputs are not deterministic. Hence, the randomness of the stochastic parts can lead to different outputs for the same input. In this way, the network simulates an ensemble of different models (Zhou, 2019). Ensemble learning allows for the estimation of uncertainty by comparing the predictions of the underlying models. In general, low uncertainty can be identified by high agreements between the classifiers' predictions, while high uncertainty can be attributed to high disagreements. By taking this a step further, a Bayesian neural network is trained with Bayesian inference (MacKay, 1992). These are stochastic neural networks trained using a Bayesian approach (Jospin et al., 2020). Here, a BNN learns a probability distribution over its weights as:

$$P(W|X) = \frac{P(X|W)P(W)}{P(X)} \tag{4.2}$$

Where $P(W|X)$ is called the posterior, which encodes the epistemic uncertainty. P(X|W) is the likelihood of weight W given the data input X, P(W) is a prior belief of weight W and P(X) is the probability of the data input X, known as evidence. Estimating $P(X)$ requires integration over all the weights as in:

$$P(X) = \int_W P(X|W)P(W)dW \tag{4.3}$$

Integrating over all possible values for weight W can be a hard and computationally expensive problem. To address this problem, simulation techniques like the Markov chain can be applied to estimate $P(X)$. The stochastic nature of a BNN requires the simulation of multiple possible models with the associated probability distributions of their weights. This means that, instead of learning hard weights, as in the case of traditional neural networks, a BNN learns a distribution of each weight. Hence, in a forward BNN pass, the output of a neuron is calculated from the distribution parameters that are learned during training. In this way, activations can be sampled multiple times from the same distribution. Thus, Bayesian Neural Networks can be considered a special case of ensemble learning (Jospin et al., 2020). In reference to ensemble models, in this chapter we adopt an ensemble of multiple neural networks with the same architecture, but each with a different initialization of weights. Unlike stochastic neural networks, our ensemble consists of multiple deterministic models, where gradient optimization is performed from different starting points. This will allow varying outputs for the same input sample. By measuring the variance between the outputs of the classifiers, we can estimate the predictive uncertainty. In the next section, 4.5, we explain the uncertainty estimation techniques we apply to generate informative examples.

4.4 Configurations

We base our experiments on the commonly applied, once state-of-the-art transformer-based model, BERT. For the classification model, we apply an implementation of BERT, and for language generation, we default to GPT-2, as in the experiments of chapter 3. With BERT, rich representations of the textual data can be produced, giving us the ability to achieve high enough classification performance with small datasets. As explained in chapter 2, transformer-based models, like BERT, only require an attachment of a classification head that fits the target task. A classification head can be as simple as a linear layer with a softmax activation function. This linear layer transforms the output of the pretrained BERT to a vector with number of dimensions matching that of the target variable. For instance, a pretrained version of the 24-layer BERT outputs a 1024 dimensional vector from each of its layers. For a multi-class target task of 6 classes, a linear transformation can be applied to reduce the 1024 dimensions to 6. Since a single linear layer is not sufficient to achieve a strong enough fit on the training data, as in figure 4.5, we implement the architecture from table 4.3 for the classification head.

4.4.1 Experimental Challenges

Training over-parameterized models on small datasets can be an unstable process. This occurrence has been studied in the fine-tuning of BERT (Zhang et al., 2020). McCoy et al. (2019) observed high variance in the generalization error for different instances of the same BERT model, fine-tuned on the same dataset. In this setting, each instance of BERT is initialized with different random weights, a result of randomness in computer-generated numbers. In today's classical computers, random numbers are generated from a deterministic process known as Pseudorandom generator (PRNG) (Jun and Kocher, 1999). Different from quantum-based computers, the core elements for computation in classical computers are based on electron charges represented by bits which hold values of either 0 or 1. The deterministic nature of randomness in classical computers depends on the outputs of physical measures such as time or hardware temperatures (Bird et al., 2020). This makes the outputs of a random generator the same for the same input seeds. Hence, by setting a fixed seed for random generators, researchers are able to conduct and report reproducible experiments. In the case of fine-tuning BERT-based classifiers, the high variance in generalization errors with different random seeds is a clear indication of performance instability. To mitigate this issue, in our experiments, we create an ensemble of different instances of BERT, all trained on the same training data. This process is described in section 2.8, under bagging.

In theory, we can build an ensemble of multiple BERT based classifiers. However, in practice, transformer-based models like BERT pose a high computational demand. This makes it infeasible to train a large number of BERT-based classifiers, even on advanced hardware. To train an ensemble of 10 BERT classifiers, the memory requirements go well beyond 24 GB. When adding GPT-2 for data generation, the computational overhead becomes even greater. For this reason, we seek a practical implementation of an ensemble by having BERT set only as a non-trainable embedding layer. That is, we use BERT to create an embedding vector for the sequence input. This vector then becomes input to our ensemble of n classifiers. In this way, instead of training n instances of BERT, we utilize one instance to train n custom-built neural network classifiers. In its paper, BERT was not suggested as an embedding model. Instead, the authors proposed BERT in an evolutionary application, where the pretrained model can be fine-tuned on a target dataset without needing to design and train a complicated neural network on top. For fine-tuning the model, the authors suggested to simply add a classification head consisting of a feed forward neural network. In our experiments, however, we do not use BERT for classification, but instead for embedding the input text. For this reason, we do not perform any fine-tuning. Instead, we create an embedding vector from the outputs of a pretrained instance of BERT. In the original implementation of BERT, the authors used the representation of the [CLS] token from the final layer as input to the classification head for fine-tuning. To recap from section 2.3.2, the [CLS] is a special token that is added at the beginning of every input text. Because of the attention mechanism in BERT, the [CLS] token has been found to capture the representation of the input text. For this reason, the authors of BERT found out that passing the representation of the [CLS] token to the classification head was sufficient for fine-tuning the model on a target task. Taking this into consideration, we used a pretrained instance of BERT to generate an embedding representation of the [CLS] token for the input text. This embedding vector became the input to an ensemble of n classifiers, that was then trained on the sampled TREC-6 dataset from section 3.4.1. The results were not satisfactory, as we could only achieve an overall test accuracy of 47%. This suggests that although the representation of the [CLS] could be sufficient for fine-tuning BERT, it may not be enough to create a sentence representation from a pre-trained model. In the following section, we use our approach to generate representative embeddings from the BERT model.

4.4.2 BERT Embeddings

Built from the encoders of a transformer model, BERT encodes each token input to a vector representation. As we've previously discussed, each encoder block outputs a vector representation for each position. In this context, the [CLS] token, as it is usually added to the

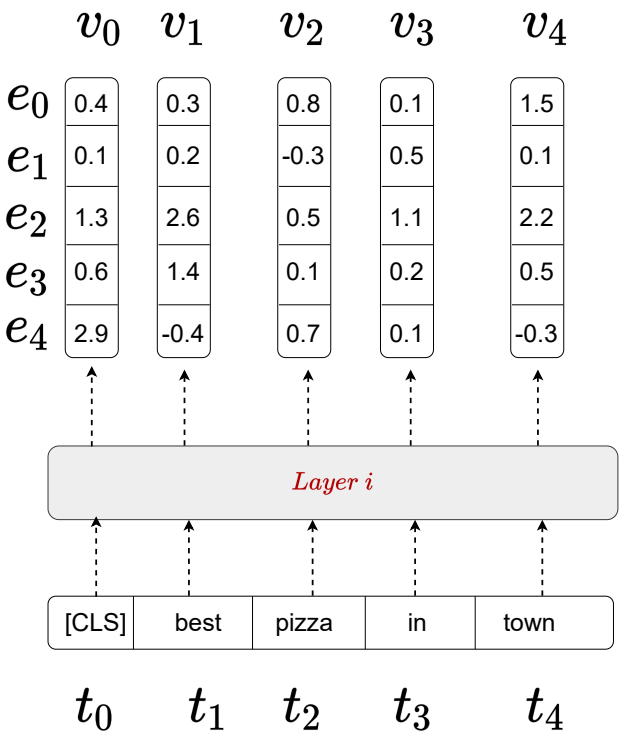


Fig. 4.1 Embedding of the sequence tokens ([CLS], best, pizza, in, town) by layer i

beginning of any text input, corresponds to position 0. This means the vector, v_0, produced by any layer i at position 0 is a representation of the [CLS] token. Consider Figure 4.1, the sequence input contains the tokens [CLS], best, pizza, in, town. In this example, we ignore specific details such as token ids, and positional embeddings, as this processing takes place before reaching layer i. We also assume that layer i outputs vectors of 5 dimensions only, $e_0, \ldots e_4$. Our focus here is to visualize how each layer in a transformer-based model computes a vector for every token in the input. In this example, layer i computes the vectors v_0, v_1, v_2, v_3, and v_4, which correspond to tokens t_0, t_1, t_2, t_3, and t_4 respectively.

As the attention mechanism assigns weights to features depending on their importance to a task, it is able to provide a representation for a token t at position x in relation to all the other tokens in an input. More formally, for an input of length n, t_x is represented in its relation to tokens $\{t_0, \ldots t_{x-1}, t_{x+1}, \ldots, t_n\}$. As such, by appending a [CLS] token at the beginning of every input for classification tasks, the vectors generated at t_0 capture the representation of the full input sequence in relation to the target output. Thus, by passing the vector representation of the [CLS] token to a classification layer, the full BERT model can be fine-tuned on the target task. Since fine-tuning involves training the target task, a gradient signal is passed from the classification layer to the BERT model in a sequential order. This

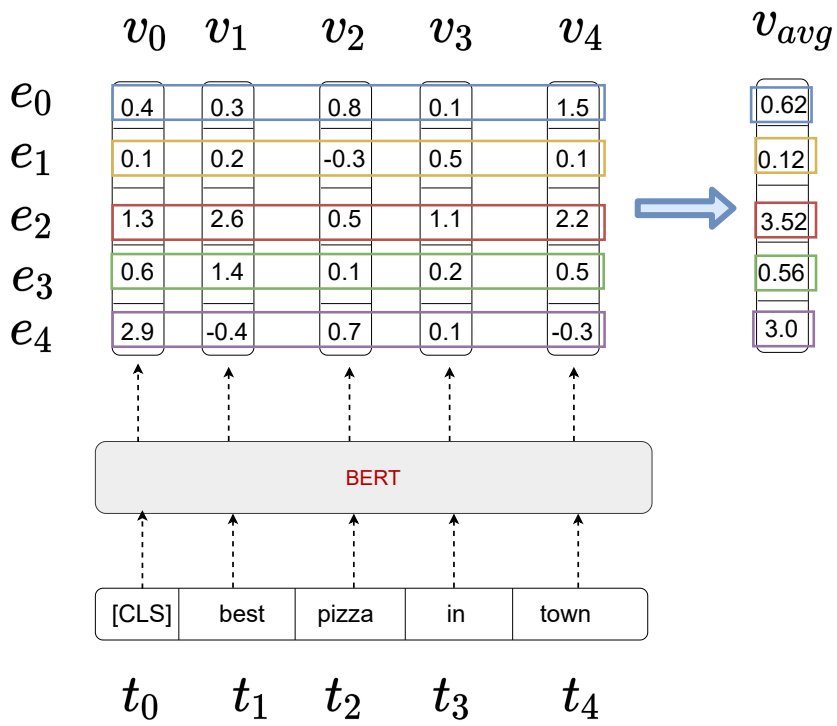


Fig. 4.2 Token vectors v_0, v_1, v_2, v_3, and v_4 are pooled by taking the average of the values at each dimension

makes passing the [CLS] representation of the final BERT layer an appropriate choice for fine-tuning the full model. However, without fine-tuning the model, the [CLS] token may not give a good representation of the input sequence. In fact, Choi et al. (2021) found out that without fine-tuning, the embedding of the [CLS] token results in poor performance on downstream tasks. Therefore, in our experiments, we included the embedding of each token for the final sequence representation. This can be done by applying average pooling on the token vectors. For each index e_i, the values of the corresponding dimensions are averaged to get the result vector v_{avg}, as shown in Figure 4.2.

Up until now, we have assumed that each token vector is produced by BERT's final layer only. However, since each layer generates its own representation of a token embedding, multiple representations for the same token will exist. When fine-tuning BERT for classification tasks, it is a straightforward choice to pass the [CLS] embedding of the final layer. This is because during training, the gradient signal is passed from the classification head to the BERT layers in a sequential order, starting from the final layer. But in the case of using BERT as an embedding model only and without fine-tuning, a token's representation can be extracted from any layer. As the aggregation vectors can retain more information, we apply

the same pooling technique to the layers of BERT. In the following section, we discuss how this process is applied.

4.4.3 Pooling BERT Layers

BERT large consists of 24 layers. This means we can either pool all 24 layers, or we can limit our pooling to a number of selected layers. For layer selection, we'll need to know which layers to focus on. A simple approach would be to pool a number of consecutive layers. These can either be earlier, e.g. layers $1, 2, 3, 4, \ldots, 12$ or later layers, e.g. $12, 13, 14, \ldots, 24$, in the model. There are multiple ways to pool layers, the simplest would be to apply aggregation operations like taking maximum or minimum values, summation, averaging, etc. For our approach, we apply the averaging of vectors. In this approach, the vector values corresponding to index e_i are averaged. This means that for n vectors, where each has x dimensions, average pooling would result in a single vector v with the same number of dimensions x. Here, the elements corresponding to each dimension are averaged in v. We illustrate average pooling on layer outputs in Figure 4.3. For any input sequence, each layer outputs a vector of x dimensions. In this example, we show how averaging is performed on the dimensions of the output vectors from the last 4 layers. Each dimension in the pooled vector v contains the average of the values from the dimensions that correspond to its color.

It is important to emphasize that the pooled vector in Figure 4.3 is a representation of one token only. An input sequence with m tokens will result in m averaged vectors. To get a sequence embedding, we apply another average pooling on the m vectors, as described in 4.4.2. A study by Ethayarajh (2019) found that later BERT layers produce more context-specific representations than earlier layers. As such, in our implementation, we only consider the selection of later layers. To find an optimal number of layers for selection, we run experiments on downstream classification tasks. We experiment with all 24 layers, the last 12, 8, and 4 layers. The datasets used for this experiment are TREC-6 and SST-2. For TREC-6, we randomly sampled less than 1.5 percent of the total training data for each class, giving us a total of 76 examples. We did this as a stratification technique, since the full training dataset is not balanced. As for the SST-2 dataset, we used the same sampled data from section 3.4.1 of chapter 3, that has a total of 20 examples per label. In this experiment, we use the embeddings as input to an ensemble of 30 different instances of the same classifier. Recall that each instance is a classifier with a different initialization, i.e. weights initialized with different values. We built the classifiers with the architecture described in Table 4.3. Training was done for 300 epochs on batch sizes of 32. We used cross-entropy as the loss function and Adam as the optimizer, and set the learning rate to 0.001 and a weight decay of 0.001. For the ensemble's target output, we used the soft voting method, described in chapter 2,

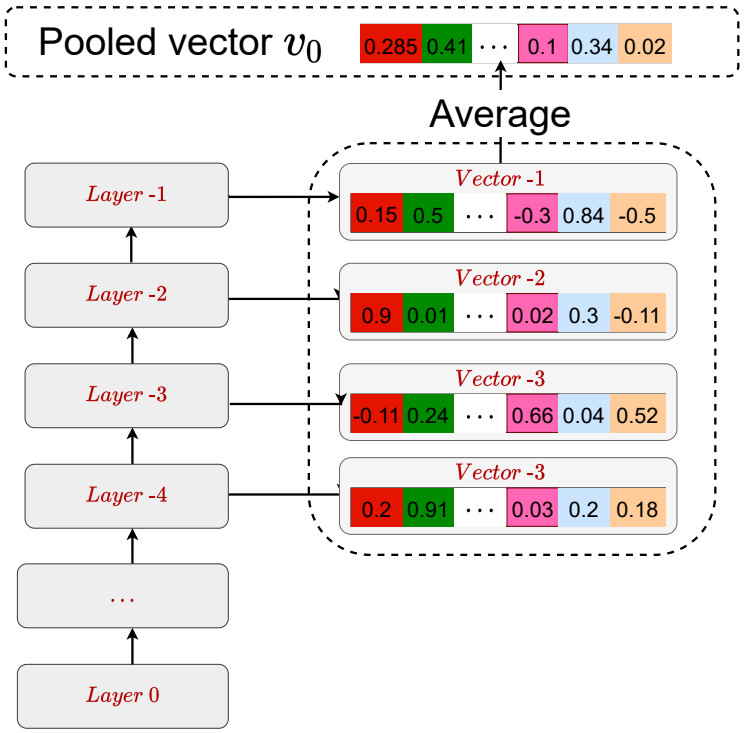


Fig. 4.3 Vectors from the last 4 layers are pooled by averaging the values in every dimension. At values colored corresponding to dimension; e_0 in red, e_1 in green, etc.

section 2.9.1. The results on the test sets are displayed as accuracy scores in Table 4.1. We can see that when averaging all 24 layers, we get as low a performance as when averaging the last 4 layers. The performance from averaging all 24 layers could have been affected by the presence of the early BERT layers, which are less context representative. On the other hand, it seems that not enough information is captured by averaging the last 4 layers, which equates to only 17 percent of the total number of layers. In the case of averaging 8 and 12 layers, we get improved performances on both TREC-6 and SST-2, and within close range. Overall, we can see that by averaging the last 8 layers, the best performance is achieved.

	TREC-6	**SST-2**
24 layers	69.8	76.8
12 layers	72.4	81.3
8 layers	72.8	82.3
4 layers	69.4	79.6

Table 4.1 Evaluation of BERT sequence embedding on TREC-6 and SST-2

For reference, we compare the results in Table 4.1 with the application of the Universal Sentence Embedding model, that is used in the experiments of chapter 3. We kept all the

other configurations the same; this included training parameters, ensemble size and the architecture of its classifiers.

	TREC-6	**SST-2**
USE Large	77.6	76.1

Table 4.2 Evaluation of the transformer-based Universal Sentence Encoder for sequence embedding on TREC-6 and SST-2

The SST-2 and TREC-6 results in Table 4.2 show that with simple average pooling on BERT's layers and tokens, we can achieve performance that is comparable to what can be reached with USE embeddings. This comparison justifies our reasoning for creating the BERT sequence embedding by averaging both its layers, and its token representations. Finally, considering the results in Table 4.1, we create the sequence vector embeddings from the last 8 layers of BERT.

4.4.4 Final Classification Model

The final classifier is a soft-voting based ensemble. The purpose of building an ensemble is to stabilize training by reducing the variance of target approximation error, as explained in section 2.9.1 of chapter 2. To demonstrate the effectiveness of ensembles, we will train 30 classifiers on the TREC-6 dataset from section 3.4 of chapter 3. Each classifier is built with the architecture in Table 4.3 and takes as input the average of the last 8 layers from the 24-layer pretrained BERT model. After testing the classifiers individually on the TREC-6 test set, we get an average accuracy of 51% with a standard deviation of 6.25. We can see that each classifier on its own performed poorly on the test set, scoring as low as 25.5%, and only as high as 61.6%. This could be a result of overfitting the data, as each classifier had a training accuracy between 98% and 100%, and a loss as low as 0.018.

4.5 Method

In chapter 3, predictions where the teacher exhibits higher levels of uncertainty are less likely to be as accurate as predictions with lower uncertainty. As the uncertainty of the teacher increases, the more likely the data will contain noisy pseudo labels. Intuitively, as the noise in the pseudo labels shifts towards a higher fraction of the training data, the teacher's mistakes are expected to be reinforced as they propagate to the student model (Zhu and Goldberg, 2009). Since the student and the teacher are the same, this type of learning falls under self-learning. In this setting, we aim to improve the learning classifier using its pseudo

labels: the labels it predicts with the highest confidence for the given data. As shown in chapter 3, the value of entropy can be determinant to the importance of a sample towards the classifier's learning. However, the learning classifier can only improve in this setting when the majority of the data is correctly labeled. When relying on the learning classifier's pseudo labels, entropy may not be an appropriate measure if applied by itself. This is because samples predicted with low entropy are less likely to be beneficial to the classifier's learning, while predictions with high entropy are more likely to be uncertain and incorrect. Thus, entropy by itself is not enough for finding data samples that can contribute to the classifier's self-learning.

In section 4.5.1, we attempt to overcome this challenge by accounting for a distributional shift between the generated samples and the labeled training data. As such, we look for out-of-distribution samples that can be predicted with high confidence. We assume that the further a sample is from the distribution of the training data, the more likely it is to hold new information. Thus, we can say that although the learning classifier has pseudo labeled this sample with high confidence, adding it to its training data is likely to improve performance. Therefore, from the generated data, we select samples for which the teacher has high confidence, but also high measures of distance from the training data.

Recall that the aim of MCTS is to guide the language generation model, GPT-2, to generate sentences that maximize the reward function. In our experiments, we apply the same MCTS process from chapter 3, considering the strong results it has achieved for generating appropriate training data in section 3.4. Accordingly, our main focus will be to apply the appropriate modifications to its reward function. With a reward objective set to maximize entropy, the generated data is more likely to include noisy labels. That is because, as the uncertainty of the classifier increases, the less confident its predictions become. Hence, predictions with low confidence are more likely to include incorrect labels, thus introducing noise. To limit the chances of noisy labels, we focus on generating data with low uncertainty, equation 5.5. However, when minimizing uncertainty during in the generation process, the output data is more likely to contain samples that are very close to the distribution of the classifier's training data, thus lowering the chances of increasing the variance in the final training data. Such data is bound to lead the training model to overfit, refer to section 2.6.

4.5.1 Distance Measures

The core of this approach is to generate out-of-distribution data distant from the labeled training data. This means that we search for examples which are distant from the distribution of the training data. However, since samples can be wrongly labeled, we only consider examples where the classifier shows confidence in its predictions. This allows us to lessen

the negative impact of noise from the trained model. Here we make the assumption that examples predicted with high confidence are more likely to have correct labels. Because we previously stated that examples with high confidence are less likely to have beneficial impact on the trained classifier, we make a second assumption. That is, examples further from the distribution of the training data are more likely to hold beneficial information. Here we measure the distance of each generated example from the labeled training data on a 2D plane. The euclidean distance between 2 data points, e.g. p and q, can be used as a unit of measure, as in equation 4.4:

$$d(p,q) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2} \tag{4.4}$$

In this context, we assume that the further a generated example is from the training examples of its predicted label, the more distant it is from the training distribution. To plot samples on a 2D plane, the data must be represented by two features, one for the x-axis and one for the y-axis. This can be done by adding to the classifier an intermediate linear layer that maps its input to an output of 2 dimensions. The activations of this layer can then be used to represent any input example. For instance, Figure 4.4 is a plot of the training data on the TREC-6 dataset as outputs of a 2D layer. In this example, samples under the same label can be seen grouped in close range.

The classifier of Figure 4.4 consisted of a linear layer with a Relu activation function, followed by a dropout layer with rate of 0.2, a 2-dimensional layer whose activations are plotted, and a final layer of 6 dimensions for the target variable. The training objective for this classifier is cross entropy with an L2 loss on the weights of its final layer. By adding more complexity to the network, a stronger fit to the training data can be achieved. For this reason we add another Relu-activated linear layer followed by dropout of rate 0.2. The final architecture of the classifier can be seen in Table 4.3.

The classifier in Table 4.3 takes as input an embedding vector that represents the text sequence. Outputs of Linear layers 1 and 2 will have the same number of dimensions as the embedding vector. After training this classifier and passing to it its own training data, the outputs of the 2-dimensional layer can be seen grouped in more contained clusters, as in Figure 4.5.

We now use the same classifier to examine how it links its training data to non-training data that was generated in the experiments of chapter 3. Here we pass the set of training and non-training data to the classifier. Then we capture the activations (outputs) of "Linear layer 3", described in Table 4.3. Since the outputs are 2-dimensional, we can plot them to

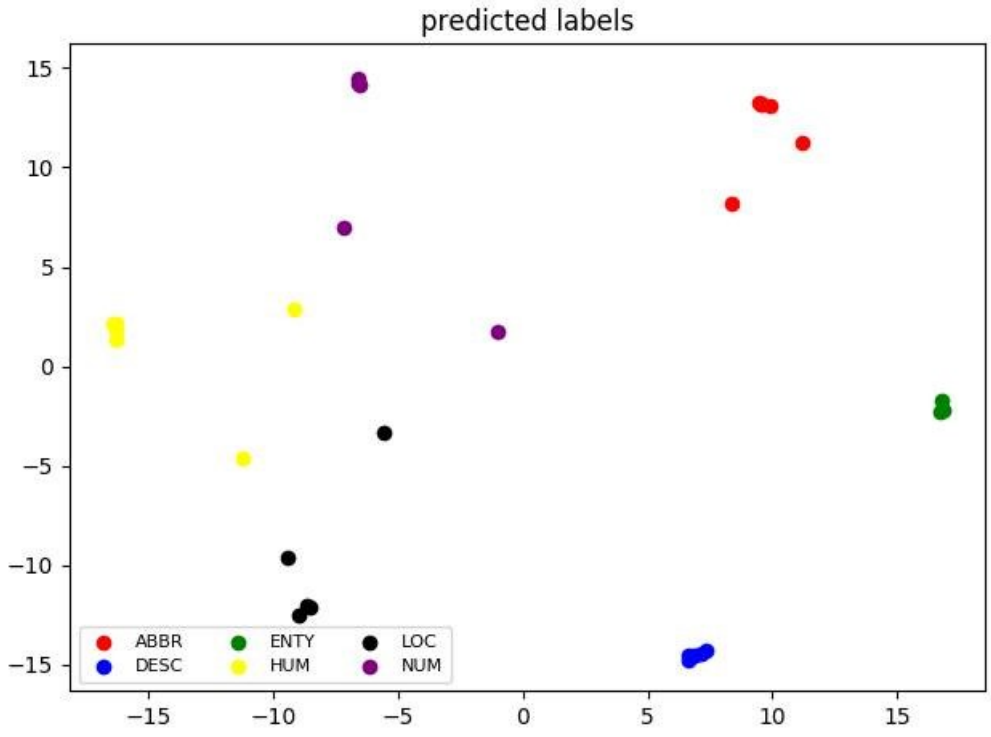


Fig. 4.4 Distance between training data examples

Layer	Output Dimensions
Linear layer 1	Input dimensions
Relu	-
0.2 Dropout	
Linear layer 2	Input dimensions
Relu	-
0.2 Dropout	-
Linear layer 3	2
Linear layer 4	Target labels

Table 4.3 Architecture of classifier from Figure 4.4. The number of target labels depends on the task. For TREC-6, there are 6 target labels

visually see the proximity between a predicted label for a generated sample and the training data cluster of the corresponding label. This visualization can be seen in Figure 4.6.

In the figure, data points representing the generated data are labeled "+', while filled circles represent the classifier's predictions for training data. The distance between a generated sample and its closest training cluster can be measured with equation 4.4. In Figure 4.7, we can see that the generated sample A is closer to the training cluster than the generated

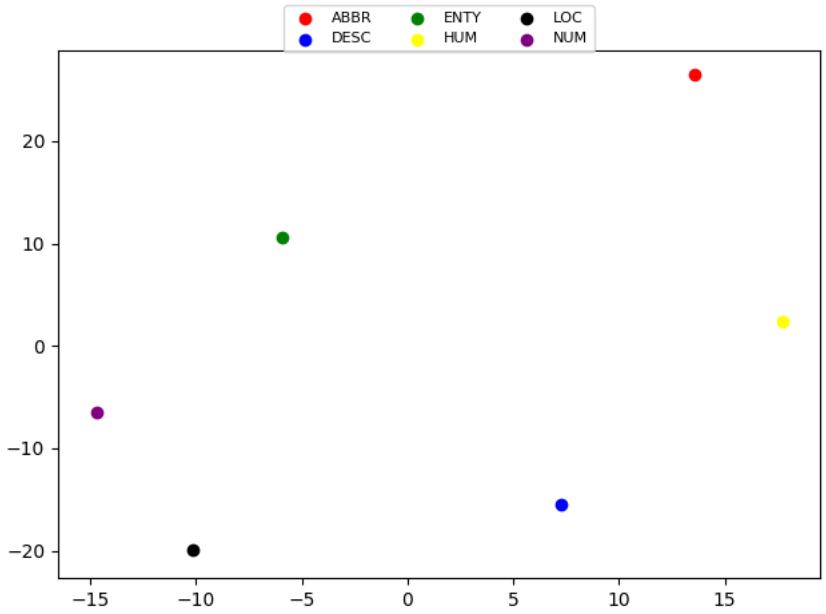


Fig. 4.5 Training data clusters

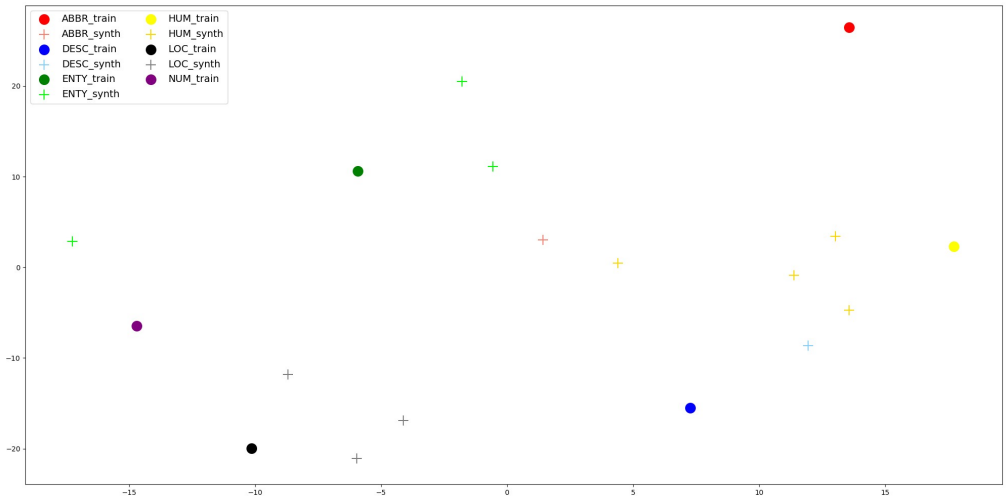


Fig. 4.6 TREC-6 predicted samples in "+"

sample B. This makes sample B more likely to be an outlier in comparison. Since there can be multiple examples in a training cluster, there will be more than one way to quantify its

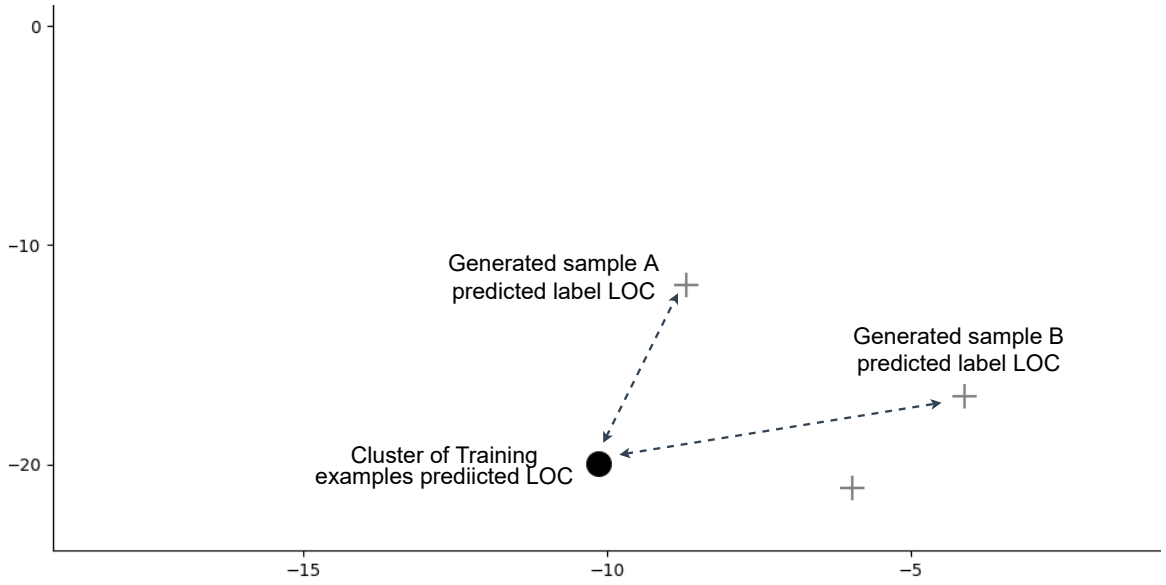


Fig. 4.7 Distance between generated examples pseudo labeled LOC and the corresponding LOC cluster as predicted by the learning classifier

distance to a generated sample. For instance, in unsupervised learning clustering algorithms like KMEANS (Lloyd, 1982), the average of data points in a cluster represents its centroid. The distance to the centroid can then be calculated (e.g. using equation 4.4), to quantify how far a new point is. Similarly, we calculate training data centroids by taking the mean of the data points in each cluster. For example, consider the training examples in Table 4.4, in which a trained classifier A from Table 4.3, as expected, labels them "LOC". What is more important here, are the data features extracted from "Linear layer 3". To calculate the centroid for the "LOC" training cluster, we compute the mean of its features. This gives the values $(0.7585, 15.7252)$ for the centroid. For a randomly generated sample, "Where is that valley that runs along the north bank of the Missouri River?", the classifier predicts the label "LOC" with high confidence of 0.99, and features (-1.93, 23.6). Applying equation 4.4 gives a Euclidean distance of 8.33. Since our experiments involve building an ensemble of multiple classifiers, the feature points will differ from one classifier to another. For example, another classifier B of the same architecture as classifier A, predicts the same label "LOC", but with a confidence of 0.6957 and feature points $(15.6741, 24.2101)$. Having a centroid of $(13.9, 11.3)$ for "LOC", Classifier B measures a Euclidean distance of 12.95 for the same sample. A third classifier, classifier C, predicts the label "HUM' for the sample with a high confidence of 93.8, and measures a Euclidean distance of 12.18 to the centroid of its closest cluster, "LOC". Although the label "HUM" was confidently predicted, its distance to the "HUM" cluster was 20.1, more than that of the "LOC" label. For an ensemble of classifiers

Training Sample	Features	Predicted Label
"What Texas city got its name from the Spanish for “ yellow ” ?"	(0.1625, 12.8841)	LOC
’What state is John F. Kennedy buried in ?’	(0.9287, 16.8456)	LOC
’Where on the Internet can I get information about the Fifth Amendment on the American Bill of Rights ?’	(1.295, 15.815)	LOC
’What country did King Wenceslas rule ?’	(-0.962, 18.876)	LOC
’What city houses the U.S. headquarters of Procter and Gamble ?’	(1.690, 16.286)	LOC
’What body of water does the Danube River flow into ?’	(-0.456, 22.852)	LOC
"What ’s the sacred river of India ?"	(0.406, 11.899)	LOC
’Where is Procter & Gamble headquartered in the U.S. ?’	(1.533, 22.095)	LOC
’What is the nickname for the state of Mississippi ?’	(2.209, 14.948)	LOC
"Where does Barney Rubble go to work after he drops Fred off in the “ Flintstones ” cartoon series ?"	(0.957, 10.309)	LOC
’What is the deepest area of the Arctic Ocean ?’	(0.315, 15.007)	LOC
’What mountain range is traversed by the highest railroad in the world ?	(1.023, 10.885)	LOC

Table 4.4 Features for training data samples pseudo labeled "LOC"

A, B, and C, the distances to their "LOC" clusters are $8.33, 12.95$, and 12.18. For reference, given the same classifiers (A, B, and C), the distances to their "HUM" centroids with respect to their order are, $35.25, 11.31$, and 20.1. Given that each classifier measures a single distance to the predicted label cluster, we aggregate the results to get a value that represents the ensemble’s final distance. This can be done by taking the mean of the Euclidean distances of the base classifiers. In chapter 2, equation 2.67 for soft voting is shown being applied to the predicted probabilities of the individual classifiers. In that setting, we consider the label with the highest confidence score, and as such apply argmax to the averaged probability scores. In the case of the distance measures under discussion, we are interested in the closest label cluster. Here, we measure the distance between a sample X and a label cluster by taking the distance between X and the centroid of that cluster. In light of the current discussion, we calculate the ensemble’s Euclidean distances for the given example for "LOC" and "HUM", by applying equation 4.5:

$$\hat{y} = \arg\min_{i} \frac{1}{m} \sum_{j=1}^{m} p_{ij}, \tag{4.5}$$

The results are as follows:

$$LOC = \frac{1}{3}\sum_{i=0}^{n=3} 8.33 + 12.95 + 12.18 = 11.15$$

$$HUM = \frac{1}{3}\sum_{i=0}^{n=3} 35.25 + 11.31 + 20.1 = 22.22$$

We get the index with the minimum distance:

$\min(11.15, 22.22) = 11.5$

Hence

$\hat{y} = \arg\min_i(LOC, HUM) = LOC$

Up until this point, we have established a method for quantifying how far non-training samples are from their closest training cluster. Here, we assume that the further an example is from its closest training cluster, the more likely it is to be an outlier. With this in mind, we proceed to generate outlier examples that can be confidently predicted by the learning classifier. This is because, as stated earlier, examples with high confidence are more likely to be correctly labeled by the classifier, yet less useful to its training. For this reason, we make the assumption that examples predicted with high confidence, but far from their label cluster, are likely to hold new information.

It is now of interest to perform self-training on artificially generated outlier examples. Having achieved promising results with MCTS in chapter 3, we apply it in our experiments, in section 4.6, to generate the required synthetic data. We first explain the role of MCTS in our application in section 4.5.2.

4.5.2 MCTS for Data Generation

In this approach, we make the following assumptions:

- A generated example is more likely to be an outlier the further it is from its closest label cluster.
- If labeled correctly, outlier examples are more likely to improve the trained classifier.

Recall that the objective of MCTS is to maximize its reward function. Since we are interested in generating examples that are far from their label cluster, we define the reward function to return the distance between the generated sample and its closest training cluster. Here, the distance for the ensemble classifier is calculated with equation 4.5. Furthermore, since we are interested in examples in which the learning model can predict with high confidence, we

adjust the reward function by introducing a confidence threshold (CT) hyperparameter. In this way, examples that are predicted with a confidence below CT will not be rewarded, as reward would be equal to 0. The final reward can be formally written as:

$$f(x_{dist}, x_{conf}) = \begin{cases} 0, & \text{if } x_{conf} < CT \\ x_{dist}, & \text{otherwise} \end{cases} \tag{4.6}$$

where x_{dist} is the distance from x to the nearest training cluster of its predicted label, and x_{conf} is the teacher's confidence in its prediction for x. As MCTS searches for solutions that return the highest reward, we expect more outlier sentences to be generated. Similar to the setup configuration in 3.3.2, we record every completed path by MCTS. This means, for N iterations, we are expected to record N sentences. Here, a path is completed by either reaching a terminal node, when GPT-2 outputs the token "<|endoftext|>", or by reaching the maximum allowed sequence length. Recall that the maximum sequence length is a user-defined tree-pruning criterion.

4.5.3 Active Learning

As in our experiments in chapter 3, we apply active learning such that after each run, the learning classifier is retrained on the selected data. Different from the previous application, in this chapter, the classifier is automatically trained on the pseudo labels once they are generated. In this setting, the user does not take part in any of the labeling or training processes. Instead, the classifier is trained on the training data in addition to the generated data using its own soft labels. We refer to the target values of the generated data as soft labels, meaning that samples are mapped to probability scores, and not one-hot vectors as is the case with the training data. As such, during training, the softmax of the output layer, equation 2.8 from chapter 2, maps the training data to one-hot vectors, but maps the pseudo labeled data to the predicted probability distributions. In the next section, we will give a detailed explanation of soft-labeling.

4.5.4 Soft-Labeling

Soft-labeling can be classified as a regularization technique for training on pseudo-labeled data with incorrect predictions. In the case of hard-labeled data, the target vectors are binary encoded, with values either 0s or 1s. We will consider soft and hard labels for neural networks trained with a softmax activation function. As explained in chapter 2, the output probabilities of the softmax function, equation 2.8, sum to 1. Thus, for a softmax function, one-hot labeled

vectors are one-hot encoded, where only one index has a value of 1, and the remaining indices contain 0s. For the purpose of this thesis, we consider the training objective of minimizing the cross-entropy loss function (equation 2.12), rewritten below:

$$Loss = -\sum_i \mathbf{y_i} \log_{\mathbf{e}}(\hat{\mathbf{y}}_\mathbf{i}) \tag{4.7}$$

Assume a binary classification task for classes A and B. The target vector [1,0] corresponds to label A, whereas [0,1] maps to label B. For a sample x labeled A, a network is trained to map x to the vector $[1,0]$. For input x, assume the network predicts a high probability of 0.98 for label A. Hence, label B has a probability of $1-0.98=0.02$. The cross-entropy loss would then be:

$$Loss = -(1 \cdot \log(0.98) + 0 \cdot log(0.02)) = -0.008 \approx 0 \tag{4.8}$$

which is a small value, as expected since the network had correctly predicted the target label. In the case of a wrong prediction with high confidence, $[0.02, 0.98]$, the loss would then be:

$$Loss = -(0 \cdot \log(0.98) + 1 \cdot log(0.02)) = 1.69 \tag{4.9}$$

which is a much higher value that would lead to a greater gradient signal for the network to adjust its hyperparameters and improve its mapping for the input x. In this example, we assumed x was accurately labeled, and thus it would be reasonable to penalize the network with a high loss, e.g. 1.69, when it gives a wrong prediction. However, if the true label for x was inaccurate, then the network would be highly penalized over a noisy sample, and consequently cause problems in its learning. To account for this problem with inaccurate data, a technique known as label-smoothing can be applied. Here, the one-hot vector is smoothed into soft probabilities. This can be seen as an element-wise function applied to a one-hot vector, where 1 is mapped to a smoothing hyperparameter p, and 0 mapped to $\frac{1-p}{N-1}$, where N is the number of classes. Thus, for input x, the class distribution can be smoothed from [1, 0] to [0.9, 0.1] for $p = 0.9$. In this case, if the classifier predicts an incorrect distribution of $[0.02, 0.98]$, the loss would be:

$$Loss = -(0.1 \cdot \log(0.98) + 0.9 \cdot log(0.02)) = 1.53 \tag{4.10}$$

It can be observed that with smoothing, the loss dropped from 1.69 to 1.53. Thus, the network is penalized at a lower degree for incorrect predictions, meaning that with inaccurate data, the impact of noisy labels is lowered.

When generating pseudo-labeled data, there is always a risk of introducing noise. This noise can be epistemic, as in the form of incorrect predictions or lack of knowledge from the teacher model. It can also be aleatoric, where there is uncertainty in the generated data, and not the labels. This being said, although the softmax activation can lead to overconfident predictions, its predictive probabilities can still be interpreted as model confidence (Gal and Ghahramani, 2016). This means that the probability outputs of the learning classifier can be used as soft-labels for its training. For example, in a binary classification task, the learning classifier may predict $[0.2, 0.8]$ for a synthetically generated example x. Here it can be said that the classifier has predicted label B with a confidence of 0.8, having assigned it the highest probability score. The predictive probabilities $[0.2, 0.8]$ can thus be used as soft labels for the student's training. Remember that in self-training, the student model will have the same architecture as its teacher. Assuming that during training the student predicts $[0.9, 0.1]$, the loss for this example would then be :

$$Loss = -(0.2 \cdot \log(0.9) + 0.8 \cdot log(0.1)) = 0.81 \tag{4.11}$$

If, instead of teacher probabilities, one-hot vectors were used for training, the label vector would then be $[0, 1]$. This would result in the following loss:

$$Loss = -(0 \cdot \log(0.9) + 1 \cdot log(0.1)) = 1 \tag{4.12}$$

For sample x, the loss would have been 0.81 when using the teacher's soft-labels, but 1 when using one-hot encoded vectors. If the teacher committed an incorrect prediction, the negative impact on the student's learning would have been 19 percent higher. Thus, with soft-labels, we are able to control the student's updates during training by using the teacher's predictive probabilities.

4.6 Experiments

We start by training an ensemble of 30 classifiers with the architecture in Table 4.3. Recall that although the classifiers share the same architecture, they are all initialized with random weights. Hence, when optimizing for the loss function, each classifier is likely to reach a different local minimum and achieve a different fit to the data. In addition to the ensemble configurations mentioned in section 4.4.3, we set the UCB constant K for MCTS to 3, the maximum sequence length to 120, and the confidence threshold CT to 0.7. We also fixed the

Run	TREC-6			SST-2		
CT	**0.7**	**0.8**	**0.9**	**0.7**	**0.8**	**0.9**
Start	72.6 (76)	68.8 (76)	70.8 (76)	82.6 (60)	81.8 (60)	82.5 (60)
AL0	74 (139)	71.4 (220)	73 (106)	82.2 (1538)	82.6 (1374)	82.2 (1130)
AL1	72.6 (441)	73.2 (313)	69.4 (147)	82 (2862)	82.1 (2574)	82 (2152)
AL2	72.8 (579)	72.8 (361)	73 (173)	81.9 (4200)	82.5 (3760)	82.2 (3150)
AL3	73.8 (669)	72.0 (431)	74 (196)	82 (5454)	82.7 (5034)	82.6 (4164)
AL4	73.4 (765)	73.4 (470)	74 (209)	82.3 (6726)	82.7 (6252)	82.2 (5172)

Table 4.5 Ensemble performance (in accuracy) for Self-training over 5 active learning runs on TREC-6 and SST-2. In each cell, the accuracy is displayed with the number of training examples including the generated and labeled data (in parentheses)

top-k for GPT-2's number of next token candidates to 20. As for the reward function, only for the TREC-6 experiment, we added a heuristic to equation 5.7 such that -1 is returned when the first token is not 'what', 'where', 'when', 'who', 'which', 'why', or 'how'. This is a task-specific heuristic, where for TREC-6, it enforces MCTS to search only for paths that start with a question word.

The results of our experiments on the sampled TREC-6 and SST-2 datasets, from section 4.4.3, are displayed in Table 4.5. For each task, we started with an initial ensemble trained on the sampled data and performed self-training over 5 active learning runs. Note that the starting accuracy on TREC-6 was 72.8, and 82 on SST-2, which are not far off from the ones achieved with the same configurations of pooling 8 layers in table 4.1. Furthermore, we can see that three repeats of training on the initial TREC-6 dataset resulted in the accuracies 72.8, 73, and 73.4. The accuracies averaged at 73.1 with a standard deviation of 0.25. The small deviation between these accuracies comes from stabilized training with the ensemble. From equation 4.6, generated sequences with a classifier confidence below the confidence threshold (CT), are given a reward of 0. We repeated the same experiment with a different confidence threshold (CT) for each dataset; 0.7, 0.8, and 0.9.

4.7 Discussion

From Table 4.5, we can see that for a higher Confidence Threshold (CT), fewer examples are generally generated. For instance, in the TREC-6 experiments, for a CT of 0.7, the total number of training examples sampled towards the final active learning run is 765. For CT = 0.8 the number of samples decreases to 470, and even drops further to 209 for CT = 0.9. The same observation can be made from the SST-2 experiments, where the number of sampled examples in AL4 for CT = 0.7 is 6726, which drops to 6252 for CT = 0.8, and to 5172 for CT = 0.9. This observation is unsurprising, because as the confidence threshold increases, the reward function further narrows the search space by enforcing a higher classification confidence. We now consider the classification performance after each active learning run. We observe that the performance did not improve in the TREC-6 experiments, and only small improvements could be noticed in the SST-2 experiments. The 5 TREC-6 active learning runs did not result in any improvements. We investigated this issue by checking the index label with the minimum distance against the label with maximum confidence. That is, for each sample, a) we check the label with the highest confidence, and b) the label cluster that is closest to it. This has shown that the closest label cluster is always the same as the predicted label, meaning a very high correlation between minimum distance and the final prediction. This does not mean that the lower the distance, the higher the confidence will be. To check for the strength of the linear relationship between the minimum distance and maximum confidence, we use Pearson's correlation coefficient (Freedman et al., 2007). For any vectors x and y, Pearson's correlation coefficient measures:

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

where $\bar{x}$ and $\bar{y}$ are the mean of the values for vectors x, and y respectively. The value of r ranges between -1 and 1 indicating a linear correlation between x and y, with 0 entailing no relationship. A positive correlation implies that as x increases, y increases as well. A negative correlation, on the other hand, entails that as x increases, y decreases. When measuring Pearson's correlation for all the SST-2 and TREC-6 experiments, from Table 4.5, we find that there is a strong negative correlation between the minimum distance and the highest confidence in the first active learning run, which decreases in later runs. These values are displayed in Table 4.6.

We can see from Table 4.6 that the correlation r between the minimum distance and maximum confidence is always strongly negative in the first active learning run. The strength of this relationship usually decreases in later runs, and even turns positive in some instances, e.g. TREC-6 CT=0.7 AL 2, AL3, and AL4. The positive relationship indicates that the

Pearson's correlation r

Run	TREC-6			SST-2		
CT	**0.7**	**0.8**	**0.9**	**0.7**	**0.8**	**0.9**
AL0	-0.89	-0.7	-0.65	-0.83	-0.83	-0.83
AL1	-0.62	-0.07	-0.43	0.15	0.14	-0.004
AL2	0.21	0.06	-0.14	-0.01	-0.04	-0.04
AL3	0.41	0.3	-0.38	-0.01	-0.08	-0.04
AL4	0.46	0.59	-0.23	-0.08	-0.03	-0.002

Table 4.6 Pearson's correlation coefficient, between the ensemble's minimum distance and maximum confidence, for the SST-2 and TREC-6 experiments from Table 4.5

confidence and minimum distance increase together linearly, which is a surprising observation. In the TREC-6 experiments, we can see the positive relationship under CT 0.7 and 0.8 for AL2, AL3, and AL4. The effect of this relationship has no obvious impact on the performance. In Table 4.5, when CT = 0.7, AL2, AL3, and AL4 observe an increasing positive correlation between the minimum distance and the maximum confidence, but performance fluctuates in small increments by first increasing from 72.8 to 73.8, then dropping to 73.4. When CT = 0.9 there is a slight increase in performance from 73 after AL2 to 74 after AL3, but no change after AL4. Yet, for the same AL runs, Pearson's correlation remained negative. This being said, the performance throughout the TREC-6 experiments only marginally changes, which can be simply attributed to differences in the data fit achieved by the ensemble. Hence, in the TREC-6 experiments we can see that for the same ensemble (Start), the test accuracies also fluctuate between 68.8, and 72.6. These results show a variance of 2.4. We now check the variance for each TREC-6 experiment from AL0 to AL4. For CT = 0.7, the variance is 0.297, which increases to 0.56 for CT = 0.8, and to 2.8 for CT = 0.9. Hence, the variance increases as the CT increases. This can be attributed to lower training samples for higher CT values. As training data increases, a stronger fit can be achieved, resulting in a lower variance. We now measure the variance for the SST-2 experiments: 0.02, 0.04, and 0.04 for CT = 0.7, 0.8, and 0.9 respectively. The variance is very small, as the change in accuracy remains minimal across the SST-2 experiments. Here, much more training data is sampled after every active learning run. Considering that SST-2 is a binary classification task, more samples are bound to be generated for each label for the same number of MCTS iterations. Furthermore, since the reward function in the TREC-6 experiments includes a task-specific heuristic, see section 4.6, fewer examples are generated as fewer paths are positively rewarded.

We now show generated samples for the TREC-6 task. The samples are in Table 4.7. The label column corresponds to the label predicted by the learning classifier. Labels that are clearly incorrect predictions are colored in red. We can see that in some instances, the

TREC-6 Generated Samples

#	Example	Label	#	Example	Label
1	What novel, directed principally by Ridley Scott, is likely to be popular at the Oscars?	HUM	9	What novel did Jane Camp Gillette write?	HUM
2	What was your favorite movie in 1920?	HUM	10	What novel did Elizabeth Johnson write?	HUM
3	What city contains the highest concentration of people?	LOC	11	How Many Hours of the Day Do You Live at The Office?	NUM
4	What country takes care of its children?	NUM	12	What was America's highest mountain?	LOC
5	What novel inspired your recent autobiography?	HUM	13	Where is this car located?	DESC
6	What are many of the most commonly encountered languages in Korea?	ENTY	14	What city is the capital of Botswana?	LOC
7	When to wash your hands of dirty clothing?	ENTY	15	How would people explain the name of a river that runs through Colombia?	LOC
8	How many years does a year pass by?	NUM	16	What do animals like to do together?	ENTY

Table 4.7 Generated samples for the TREC-6 task from table 4.5 for AL0 with CT = 0.7. Wrongly predicted labels are colored in red

generated examples are well-formed and grammatically correct, as in "What novel did Jane Camp Justice write?" and "What city is the capital of Botswana?". In other instances, mistakes are introduced, as in "How many years does a year pass by?". There are multiple examples that start with the words "What novel" indicating that MCTS has focused on paths that start with these words. This can be attributed to the high rewards. In fact, in the top 10 generated samples by their reward, 7 start with the words "What novel". We recall that the UCB function, see equation 5.3, can balance between exploitation of existing paths with the exploration of newer ones. Although it may be essential for domain specific tasks, the heuristic added to the reward function for the TREC-6 experiments, as explained in 4.6, severely punishes paths that do not start with a question word by rewarding them with a negative result of -1. This resulted in increasing the reward gap between paths that started with a question word and those that did not, thus narrowing the search space. Hence, with a narrowed search space that is restricted to paths starting with one out of seven question words, the UCB policy is likely to focus more on certain subpaths than others.

In chapter 3, we recognized that the usefulness of the augmented data is determined by the information that it adds to the classifier's learning. We previously conditioned the generated data on the entropy of the learning classifier. This allowed us to find examples where the classifier exhibits a state of confusion in predicting a target label. The data can thus be said to include information different to the classifier's learning. We relied on a user to assess the quality of the generated data and manually correct wrongly predicted labels. In this chapter, we removed the user from the learning loop by applying self-supervision. To assess the usefulness of the generated examples, as in chapter 3, we assigned a user to manually review the predicted labels. We only applied this on the TREC-6 experiment with CT = 0.7 at AL0. For each example, if the predicted label was determined incorrect, the user manually changed it to what they thought was the correct label. Otherwise, the label was kept unchanged. The classifier was then trained on the data with the user labels. This resulted in a test accuracy of 74 percent, matching that from Table 4.5. While ignoring the quality of the labels, this observation suggests that the generated text samples do not necessarily introduce new information.

4.8 Conclusion

In this chapter, we have seen that it is possible to improve a classifier's performance through pseudo-labeled synthetically generated data. However, the improvements rely heavily on the *quality* and the *usefulness* of the generated data, as well as the provided labels. Our experiments have shown that to increase performance, the generated data must hold new

information, which makes it a difficult task in self-training. In chapter 3, we saw that new information is needed to boost classification performance. Hence, eliminating the clear source of new information, that is the user, has impeded the classifier's learning progress. With the lessons learned from this chapter and chapter 3, we extend our data augmentation approach to the domain of knowledge distillation, where multiple models participate in the training process. Here, knowledge is "distilled" from a better performing model to a less performing one. Accordingly, in chapter 5, we use a teacher model as the source of information to improve the performance of a smaller and less computationally intensive student model.

Chapter 5

Enhancing Task-Specific Distillation through Language Generation

5.1 Introduction

In recent years, transformer-based language models have proven to scale over larger datasets. This has made it possible for models trained on massive collections of data to acquire generalized knowledge that can then be transferred to downstream tasks. Undoubtedly, transfer learning has produced satisfying results over tasks such as text classification, question answering, natural language inference, and machine translation. While these models can lead to significant improvements in performance, the increasing size of their learning parameters results in greater complexity and storage requirements. This can be challenging in real-time applications, especially on devices with limited computational resources Gou et al. (2021). Hence, reducing the size of these language models without sacrificing too much of their performance has become the focus of many works in knowledge distillation. For example, instead of optimizing compressed models for hard-labeled data, Hinton et al. (2015) proposed to train a smaller model (the student) with the task of predicting the probability distribution outputs (soft labels) from the larger model (the teacher). This approach has been shown to produce comparable results between the student and teacher models, but usually relies on a dataset through which to transfer the knowledge. To help improve the student's learning in knowledge distillation, large unlabeled datasets are required (Tang et al., 2019). Although unlabeled data is cheaper to obtain and is widespread when compared to labeled data, it may not be available when building a classifier tailored to a niche application. We therefore propose to generate synthetic examples that can then be used to transfer knowledge to the student in downstream tasks.

To overcome the challenge of the unavailability of large unlabeled datasets required for the distillation process, we build a data generation framework where the Monte Carlo Tree Search (MCTS) algorithm is applied to help generate examples that if added to the student's training data, will increase its performance. By taking the difference between the student's and the teacher's uncertainty for a generated example, we are able to force the language generation model to create examples that can be pseudo-labeled by the teacher with higher confidence than its student. We make the assumption that the wider the gap between the student's and teacher's uncertainty for a particular example, the more likely that this example is correctly pseudo-labeled by the teacher and the less likely that is is known to the student model. By training the student on the generated data with the teacher's pseudo labels, it is able to mimic the teacher's behavior and improve its performance. To generate examples that meet this condition, we take advantage of MCTS's tendency to search for paths that maximize the reward value, hence, the uncertainty gap. The intuition here is that the larger the positive difference in the uncertainty, the greater the contradiction is between the student and its teacher, and the more likely that the generated example is important for the student's learning stage.

The remainder of this chapter is structured as follows: Section 2 provides a background and an overview of related literature. Section 3 describes the proposed approach. Section 4 presents the experiments which were carried out. Section 5 gives conclusions and plans for future work.

5.2 Background

In the pursuit of improving performance for NLP, pretraining large-scale models on increasing amounts of unlabeled data has become a common approach (Devlin et al., 2018; Peters et al., 2018; Yang et al., 2019). By leveraging the knowledge gained from the pretraining step, these models have shown impressive performance on many downstream text tasks, e.g. the GLUE benchmark (Wang et al., 2018a). However, such improvements are accompanied by an increasing number of learning parameters, creating a need for more computational and storage requirements. To alleviate the aforementioned complexity issues, many have suggested approaches for efficient training through model optimization e.g. removal of inefficient or redundant parameters (Lan et al., 2020; Sajjad et al., 2020), and knowledge distillation (Gou et al., 2021; Sanh et al., 2019; Sun et al., 2019b, 2020).

In knowledge distillation the unlabeled data plays an intermediary role which allows the teacher to transfer its knowledge through its predictions. When this data lacks the components for a meaningful knowledge transfer, e.g. limitations in size or textual variations,

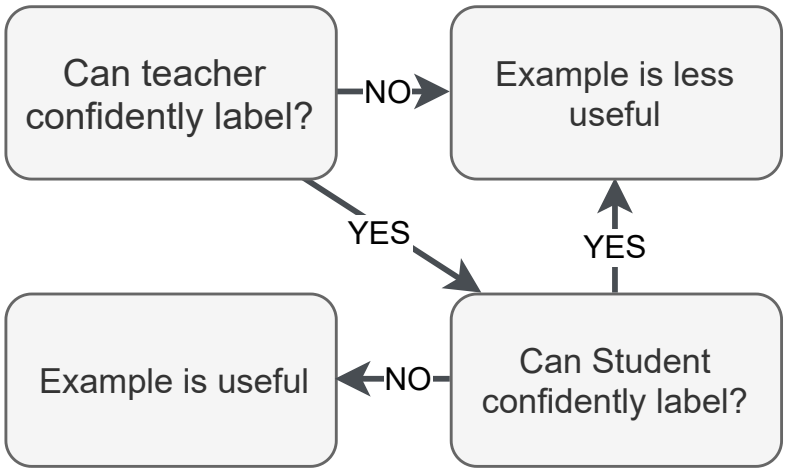


Fig. 5.1 An example is deemed more useful if the teacher can confidently label it, but not the student.

augmentation techniques can be applied to create additional examples. For instance, Jiao et al. (2019); Tang et al. (2019) apply task-agnostic heuristics, like word replacements, to improve distillation on downstream tasks. The concept of augmenting training examples has been successfully shown to improve training in computer vision (Shorten and Khoshgoftaar, 2019), and has been gaining traction in the NLP domain as well. This includes word manipulations such as the deletion, insertion, or addition of random words in text (Wei and Zou, 2019), paraphrasing or back-translation (Sennrich et al., 2015a), and most recently the application of language models to predict alternative words (Kobayashi, 2018).

In this work, we propose that instead of relying on the above augmentation techniques, we improve knowledge transfer by involving the student and the teacher in the creation of useful examples. To achieve this, we propose a framework that a) creates data examples through a fine-tuned language generation model, and b) allows the teacher and the student to condition the language generation model to produce examples that they deem useful for the knowledge transfer. The steps taken to determine the usefulness of an example are summarized in Figure 5.1. We explore MCTS's ability for finding optimal solutions by rewarding paths by the usefulness of the examples they represent, as explained in section 5.3.2.

5.2.1 Knowledge Distillation

Knowledge distillation is typically aimed at training a student model to mimic the behavior of a larger teacher model. The student can either have the same architecture as its teacher or be completely different from it, but in either case, it has fewer learning parameters. Hinton et al.

(2015) achieve knowledge distillation by training the student model on the output softmax probabilities of the teacher model according to equation 5.1:

$$softmax(z_i) = \frac{\exp(z_i/T)}{\sum_{j=0}^{n}\exp(z_k/T)}. \tag{5.1}$$

where T is a temperature scaling parameter; as $T \to 0$, the distribution of the probabilities will approach a one-hot distribution, whereas a uniform distribution is achieved as $T \to \infty$. Variants of the aforementioned knowledge distillation approach have also been proposed, which include the distillation of the activations or weights of the intermediate layers (Heo et al., 2019; Romero et al., 2014; Tarvainen and Valpola, 2017; Yim et al., 2017). In contrast to much work in Knowledge distillation, our approach does not depend on training the student model on pre-existing large datasets. Instead, our work is inspired by the concept of distilling knowledge, to improve the process of training or fine-tuning existing models starting with a few examples per label. Hence, we make the proposition that fine-tuning compact models on small datasets can be aided by the participation of larger models in a) generating additional training examples, and b) finding informative examples while providing pseudo labels.

5.2.2 Language Models

Traditional context-independent word vectors like GLOVE (Pennington et al., 2014) and Word2Vec (Mikolov et al., 2013) were popular choices for initializing embedding layers for task-specific networks. Later works focused on contextualizing representations by leveraging recurrent neural networks (Howard and Ruder, 2018b; Peters et al., 2018); most recently, the fine-tuning of pretrained transformer-based models (Vaswani et al., 2017) like BERT (Devlin et al., 2018), RoBERTa (Liu et al., 2019) and GPT-2 (Radford et al., 2019), has become a common approach for domain-specific tasks. In our experiments, as explained in section 5.5.1, we initialized our classifiers with embedding representations extracted from either BERT, RoBERTa, or GLOVE. For the language generation task, we applied GPT-2, a unidirectional language model, pretrained on large textual datasets with a probabilistic function to estimate the probability distribution over the vocabulary for a given context. For a sequence of tokens $t_1, t_2, t_3, ..., t_n$, the joint probability can be modeled as:

$$p(t) = \prod_{i=1}^{i=n} p(t_{(i)}|t_{(1)}, \ldots, t_{(i-1)}) \tag{5.2}$$

Hence, the conditional probability of a token $p(t_i|t_{1:i\_1})$ can be estimated by the probability distribution over the model's vocabulary, given a context $t_{1:i\_1}$. For this reason, we are

able to generate candidates for every next token with the top-k sampling approach (Fan et al., 2018). When a token is selected, and the process is repeated enough times, a properly trained model can generate a meaningful sequence of text. In our experiments, we fine-tune GPT-2 on the initial dataset. Even though we restrict our approach to small datasets, GPT-2 is still able to generate relevant examples. This allows us to utilize it with MCTS to construct a tree with paths that lead to meaningful text examples.

5.2.3 Monte Carlo Tree Search

The Monte Carlo Tree Search algorithm has been widely applied to gaming theory (Browne et al., 2012; Kocsis and Szepesvári, 2006). Its ability to solve decision making problems in games with large combinatorial search spaces (Arneson et al., 2010; Chung et al., 2005; Silver et al., 2016), has made its application appealing even for non-gaming problems as well (Edelkamp et al., 2016; Nguyen et al., 2016). MCTS has also recently received attention in the NLP domain (Quteineh et al., 2020), where it is applied to create synthetic examples that are then labeled by the user.

MCTS incrementally constructs a tree as it searches for possible solutions that satisfy the conditions set by its reward function. In computer games, paths that lead to winning states are more likely to have higher reward values than paths that lead to losing states. While any of the winning paths could be equally desirable, some paths could have a higher probability of reaching a winning state than others. By keeping track of the number of visits MCTS makes with every path it takes, we can safely assume that winning paths with a higher number of visits are more likely to reach a winning state. However, if the path selection criterion focuses only on maximizing the reward value, it can repeatedly revisit the same paths while failing to discover new ones. To account for this, a selection policy must balance between exploration of new paths and exploitation of already visited paths. A common policy that can achieve this balance is the Upper Confidence Bound UCB, proposed by Auer et al. (2002) for solving the multi-armed bandit problems, as shown in equation 5.3:

$$UCB = \frac{W_i}{S_i} + C\sqrt{\frac{2 \times lnS_p}{S_i}} \tag{5.3}$$

where at a given state i, W_i represents the number of paths leading to a winning state, and S_i records the total number of paths made from i. This makes the first part of the equation, $\frac{W_i}{S_i}$, favor paths that on average have led to a winning state. S_p is the total number of paths taken from the parent node, and C is an exploration constant that is predefined by the user. This makes the second part of the UCB policy, $C\sqrt{\frac{2 \times lnS_p}{S_i}}$, favor unexplored paths. Hence, a

higher C would lead to a more exploratory tree search. When UCB is combined with MCTS, the approach is commonly known as the Upper Confidence Bound for Trees UCT (Browne et al., 2012). The final MCTS algorithm can be divided into four main stages:

1. Selection: Starting from a root note S_r, the UCB function from equation 5.3 selects the node to visit next. This policy is recursively applied until an unexpanded node is reached.

2. Expansion: Once a leaf node is reached that is non-terminal (an incomplete path), it is expanded by adding its immediate children. This corresponds to all of the immediate actions that are possible from that state.

3. Simulation: A path is generated until a terminal state is reached. This could be a completely random path depending on the default simulation policy.

4. Backpropagation: Once a terminal node is reached, the reward value is measured and backpropagated to the nodes of the current path. Other statistics such as the number of visits are also updated.

Typically MCTS runs until a predefined criterion is achieved e.g. a user-specified time or a maximum number of iterations. We adapt MCTS so that each node represents a token generated by GPT-2, where the possible actions k from a particular node S_i are given by the top-k sampling process. When a full path is constructed, it will represent a complete text example that is then passed to the reward value as described in section 5.3.

5.3 Approach

In this work, we evaluate the proposed Monte Carlo Tree Search (MCTS) generation method (section 5.3.2), alongside the baseline Non-Guided Data Generation (NGDG) method (section 5.3.4). MCTS generation combines MCTS, a language generation model, a teacher and a student classifier to create synthetic textual examples that can enhance the performance of the student in a knowledge distillation setting. Here, the language model is conditioned to generate examples for which the prediction of the teacher and the student are as divergent as possible.

The main components of our framework, as shown in Figure 5.2, include the language generation model (GPT-2), a teacher model, a student model, and the MCTS algorithm. Below we discuss the role of each component.

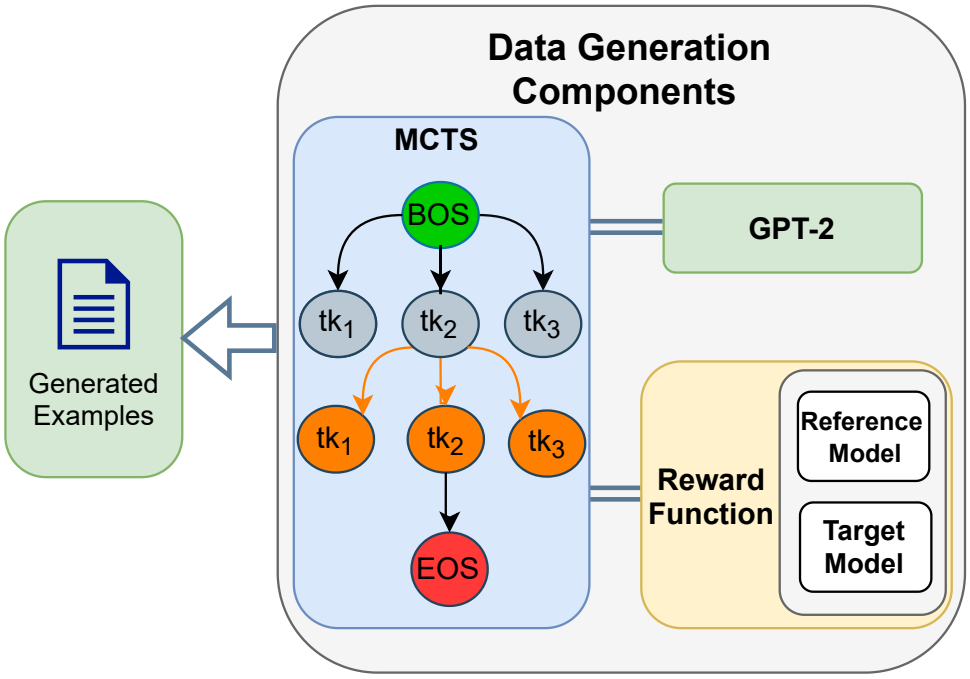


Fig. 5.2 MCTS builds a tree from token sequences generated by GPT-2. Meanwhile, the teacher and student models work together to assign reward values for every completed path

5.3.1 Language Generation with MCTS

Because the search tree is constructed by traversing from top to bottom, a unidirectional generative model can be applied to predict candidates for every next node, given the tokens of parent nodes as input. This unidirectional behavior makes GPT-2 a good choice for our experiments[1]. To generate relevant data, GPT-2 is first fine-tuned on the initial training dataset. We prepare the data by first dropping the classification labels, then merging the text examples, separated by a special token EOS (end of sentence). Once GPT-2 is fine-tuned, we use it to generate a token for each node in the constructed tree, represented by tk, as shown in Figure 5.2.

5.3.2 Monte Carlo Tree Search MCTS

Starting from a root node, represented by a special token BOS (beginning of sentence), we sample the top k most probable tokens that come next in sequence. Having only started from a single root node without any child nodes to select from, the tree is first expanded by adding the top k tokens as immediate children of BOS, making the depth of the tree equal to 2. Since at this stage all child nodes have equal weight, any one of them is selected. The next

[1]We use the implementation provided by huggingface https://huggingface.co/

step is to start a simulation by first concatenating the token of the selected node with those of its ancestor nodes. The resulting text is then passed to GPT-2 to obtain the probability distribution over the vocabulary, where the top k tokens are sampled. Given that this process takes place in the simulation stage, the UCB selection strategy is not applied; instead, we select a random token from the non-uniform distribution of the k tokens. In a standard MCTS implementation, the path that is navigated in the simulation phase is not necessarily recorded, but rather the statistics of it are, e.g. the result and number of visits. However, to account for computational costs, we not only keep track of the statistics, but also the generated text at the end of every simulation. It is important to note that the tokens generated during a simulation are not added as nodes to the tree, but are recorded separately. In this way we guarantee that while the growth of the tree is not affected during a simulation, we nevertheless retain the generated text examples with positive rewards. After a number of iterations, the statistics of the tree nodes will have changed, allowing higher impact of the UCB policy (equation 5.3) in searching for paths that lead to examples with the higher reward values.

5.3.3 Reward Function

This component plays a key role in our approach as it dictates the type and quality of the generated data. In a student-teacher approach, where a teacher transfers knowledge to its student, the aim is to find examples that the teacher model can label with much higher certainty than the student model. As entropy measures the uncertainty of a prediction model for a particular example, the higher the entropy, the lower the confidence of the classifier in its prediction. This motivates us to generate examples that can be predicted with low entropy by the teacher, but high entropy by the student. Hence, when the difference in entropy between the two models is increased, the more important the generated example becomes for training the student. Given a generated text example x_u, the predicted probabilities from a model m are outputs of its $softmax$ layer:

$$P_m(y) = softmax(f(x_u)) \tag{5.4}$$

The entropy is thus:

$$H_n(P_m) = -\sum_{i=1}^{n} p_i \log_b p_i \cdot \frac{1}{\log_b n} \tag{5.5}$$

where the predicted probabilities $P_m = \{p_i; i = 1, ..., n\}$ for n labels. We take the difference in entropy between the student model s, and the teacher model t:

$$\Delta\text{ent} = \text{ent}_s - \text{ent}_t \tag{5.6}$$

The teacher's confidence and the student's lack of confidence in labeling an example are reflected in Δent. For predictions where the teacher's confidence is at its highest, and the student confidence is at its lowest, Δent is maximized. Hence, examples with high Δent are more useful for distilling knowledge to the student model. For this reason, only examples where Δent is positive are considered.

By finding paths that maximize Δent, GPT-2 is conditioned to generate examples that can be predicted with the lowest uncertainty by model t, but with the highest possible uncertainty by model s. Here we make the following assumptions: a) Examples that maximize the gap between entropy values are those most informative to the student model, yet can be confidently labeled by the teacher model; b) Training the student on informative examples can increase its performance. Since the objective is to find solutions that maximize the reward value, each generated path is rewarded by Δent. To further optimize the search process, we add the following refinements to the reward value.

$$reward = \begin{cases} 0, & \text{if \#tokens} < x, x \in \mathbb{Z}_{\geq 0} \\ 0, & \text{if } \Delta\text{ent} < 0 \\ -1, & \text{if task specific condition} \\ \Delta\text{ent}, & \text{otherwise} \end{cases} \tag{5.7}$$

The first condition in equation 5.7 is a heuristic that penalizes examples below a user-defined minimum number of tokens. In our experiments, we set x to 3 tokens. The second condition minimizes the penalty for examples where the student model is more certain in its prediction than its teacher. The third condition eliminates cases based on a task-specific condition, see TREC-6 configuration in section 5.5.3. Finally, the fourth condition results in a positive reward when the teacher is more certain than the student.

5.3.4 Non-Guided Data Generation (NGDG)

We set a baseline in which examples are generated without conditioning GPT-2 on the predictions of the student and the teacher classifiers. Instead, examples are generated purely on the probability distributions for the candidate tokens from GPT-2, then filtered on the conditions from equation 5.7. As in the MCTS simulation phase, examples are generated by applying top-k sampling on the outputs of GPT-2. In top-k sampling, the probability mass is redistributed over the top k most probable choices. At each timestep, k candidate tokens are sampled based on the previously generated tokens. A token is then randomly selected from the k most likely candidates. A sequence is completed when a symbol indicating the end-of-sequence is selected, or when a user-defined maximum sequence length is reached.

Due to the randomness in selecting the next token, a different sequence can be produced whenever the generation process is repeated. Once enough examples are generated, the resulting data is cleaned by removing duplicates and short sequences, e.g. those containing less than 3 tokens. Next, a probability distribution for each remaining sample is generated by both the student and the teacher models, in order to compute each classifier's entropy (equation 5.5). We denote the student's entropy by ent_s and the teacher's entropy by ent_t. By applying equation 5.6 to ent_t and ent_s, for each sample, we compute the difference of entropy between the teacher and its student, Δent. We then apply equation 5.7, and select the examples where Δent is positive.

5.3.5 Augmented Data

Once the synthetic data D_S is generated, we append it to the initial labeled training dataset D_L that the student and teacher models were initially trained on. The student is then trained with the cross-entropy objective function:

$$H(y,\hat{y}) = -\sum_{i=0}^{n} y_i \log(h_\theta(x_i)) \tag{5.8}$$

where $h_\theta(x_i)$ is a softmax probability produced by the learning model, and y is a soft label provided by the teacher model for D_S.

5.4 Experimental Challenges

Given that in our experiments we implement transformer-based classification models, we face the same experimental challenges mentioned in section 4.4.1. Similar to the configurations for the experiments in chapter 4, in section 5.5, we create an ensemble of 'n' classifiers. Each classifier takes BERT embedding vectors as input. The process for generating an embedding vector was explained in section 4.4.3. In section 5.6, we attempt to mitigate the instability issues of the BERT-based classifiers by applying hyperparameters and training configurations described in Mosbach et al. (2020).

5.5 Experiment 1

In this section, we create an ensemble model for both the teacher and the student, by training multiple instances of the corresponding classifiers. As explained in section 4.4.1, training multiple instances of BERT requires high storage and memory bandwidth. To avoid the

complexity issues associated with training multiple BERT models as base classifiers, in this section, we only use BERT to generate input embeddings. Accordingly, we build the ensemble models by using the embeddings of a pre-trained BERT model as input to the base classifiers that are of lower complexity. This process is explained in section 4.4.4. Considering that the base classifiers are much less computationally demanding compared to BERT, the final ensemble can scale over a large number of base classifiers, thus avoiding the complexity issues that result from training multiple BERT models.

5.5.1 Classification Models

We based our experiments on the language models provided by Huggingface (Wolf et al., 2020), the 24-layer BERT acting as a teacher, and its student, a 6-layer distilled version of either BERT or RoBERTa (Sanh et al., 2019). We also introduced a third student based on GLOVE embeddings with a fraction of the parameters of the other models. For any text input, as implemented in the Flair framework (Akbik et al., 2019), the GLOVE representation is simply the average of its word embeddings. As for the Transformer based models, we did not fine-tune the pretrained layers during training, but instead froze them. Considering the work by Ethayarajh (2019), which reports that later BERT layers produce more context-specific representations than earlier layers, we averaged the activations of the last 4 layers to pass as input to the embedding layers of our classifiers. This allowed us to efficiently build an ensemble of classifiers to stabilize training for more generalized and consistent predictions (Granitto et al., 2005). Ensemble learning has been applied to transformer-based models in different forms (Clark et al., 2019; Dang et al., 2020; Lan et al., 2020). In our implementation, freezing the hidden layers tremendously lowered computational costs as we ended with one pretrained model embedding data for an ensemble of size n, instead of n pretrained models. We built our classifiers with the same architecture for each model, the only difference being in the number of parameters. Table 5.1 summarizes the final architecture for each model variant. We set the ensemble size to 30, and for the final output we averaged the probability estimates of the voting classifiers (Brown, 2010).

5.5.2 Datasets

To simulate settings with scarce training data, we artificially created an initial training set by randomly sampling a seed of examples from the available training data. The language generation model, and the student and teacher classifiers, are then fine-tuned on the sampled training data. We based our experiments on datasets where for our chosen models, the teacher outperforms the student, so that knowledge transfer is reasonably applied. For this reason,

	BERT	Student	
Layer	24-layer	6-layer	GLOVE
Linear	1,049,600	590,592	10,100
Relu	-	-	-
Dropout	0.2	0.2	0.2
Linear	1,049,600	590,592	10,100
Relu	-	-	-
Dropout	0.2	0.2	0.2
Linear	1024 * n	768 * n	100 * n

Table 5.1 Depending on the experiment, the student is either based on DistilBERT, DistilROBERTA or GLOVE embeddings. n is the number of output neurons, corresponding to the number of target labels

on datasets where DistilBERT performs similarly to BERT, we introduced DistilROBERTA, as in the DBpedia, and AG-news tasks (Zhang et al., 2015). We also repeated the TREC-6, SST-2, and DBpedia experiments with GLOVE embeddings.

We randomly sampled enough examples for our training data from each dataset so that the teacher model performed just well enough to pseudo-label the generated data. The SST-2 (Socher et al., 2013), YELP-2 (Zhang et al., 2015), and IMDB reviews (Maas et al., 2011) are for binary sentiment classification, TREC-6 (Li and Roth, 2002) is a 6-label question classification dataset, and DBpedia and AG-news are topic classification datasets with 14 and 4 labels respectively. From each dataset, we selected a completely random small portion of training examples for the initial training. TREC-6 contains 5,452 training examples and 500 testing examples. We use the same data sample from section 4.4.3. Recall that this data was sampled with stratification, and resulted in less than 1.5 percent of data for each class. Here, we use the same sampled dataset. Since the training dataset is not balanced, the sampled set contains less than 1.5 percent of the data for each class, giving us a total of 76 examples. For the SST-2 experiment, we sampled 60 examples from the training data for the GLUE SST task (Wang et al., 2018a), and evaluated on the provided development set. As for the IMDB dataset, from a total of 25,000 training samples, we randomly sampled 40 examples per label and tested on the 25,000 test examples. For YELP-2, AG-news, and DBpedia, we sampled a total of 40, 20, and 42 examples respectively. We note that other data augmentation works have fine-tuned GPT-2 on SST-2, Yelp and TREC (Feng et al., 2020; Kumar et al., 2020).

5.5.3 Configurations

We configured MCTS to run with a different number of iterations in each experiment as shown in Table 5.2, and set the UCB constant C from equation 5.3 to $C = 3$, and the top-k sampling to $k = 20$ in all our experiments. We also added a pruning criterion to limit the maximum length of any path to 120 tokens for all our experiments, except for IMDB where we set it to 240. As for the reward function, only for the TREC-6 experiment, we added a heuristic (see 'task specific condition' in equation 5.7) to condition the first token to be a question word by returning -1 if it was not 'what', 'where', 'when', 'who', 'which', 'why', or 'how'. After the data was generated, we filtered the examples by selecting the ones that the teacher pseudo-labeled with confidence above 0.7. To avoid high imbalance for the binary datasets, we limited the number of added examples to the size of the minority class. For the other datasets with more than two classes, we simply limited the selection over the median, given the distribution of generated examples per label. Finally, we trained the student classifier with the predictions of the teacher using a cross-entropy loss function from equation 5.8 and selected Adam as the optimizer (Kingma and Ba, 2014). We also fixed the training parameters to 500 epochs, 32 batch size, 0.001 weight decay, and 0.001 learning rate throughout all our experiments. We did not perform any temperature scaling on the softmax probabilities; similar to $T = 1$ in equation 5.1.

5.5.4 Results

Results for the teacher-student knowledge transfer experiments are shown in Table 5.2. As explained in section 5.5.1, the teacher model is a 24-layer BERT, and the student model for SST-2, TREC-6, IMDB, and YELP-2 is a 6-layer DistilBERT, but for AG-news and DBpedia it is a 6-layer DistilROBERTA. Furthermore, the TREC-6, SST-2, and DBpedia experiments were repeated with GLOVE embeddings. We repeated each experiment five times with a different number of iterations for MCTS: 2.5K, 5K, 10K, 15K, and 20K. The results in Table 5.2 show the test accuracy of both the teacher and student on the sampled data from section 5.5.2, and the student 'DistilStudent' after it has been trained on the sampled data combined with the generated data. Underneath each accuracy score, in parentheses, is the size of the training data (# Training examples). For the teacher and the student prior to distillation (labeled Start in the table), the data sizes are those of the initial training sets, described in section 5.5.2. For the distilled students, the data sizes reflect the number of examples from the initial training set, plus the generated examples from MCTS that are then filtered with the conditions in section 5.5.3.

Table 5.2 Teacher-Student Results (in percent, numbers of added examples in parentheses below)

Dataset	Teacher	Pre-Distillation		Approach	Student (Post-Distillation)				
	BERT large	Student Model	Start		2.5k	5k	10k	15k	20k
TREC-6 # Examples	72.6 (76)	DistilBERT	65.8 (76)	MCTS	72.4 (478)	73.2 (1,021)	74.8 (2,224)	74.6 (3,504)	73.6 (4,900)
		GLOVE	51.4 (76)		59.8 (495)	61.6 (1,063)	64.2 (2,234)	63.4 (3,511)	63.4 (4,215)
		DistilBERT	65.8 (76)	NGDG	68.4 (121)	69.2 (181)	70 (236)	72.4 (318)	71.6 (395)
		GLOVE	51.4 (76)		53.6 (111)	57.4 (174)	55.2 (257)	55.6 (343)	58.4 (433)
SST-2 # Examples	82.1 (60)	DistilBERT	73.5 (60)	MCTS	75.9 (478)	75.1 (1,030)	75.0 (2,052)	77.4 (3,144)	77.1 (4,384)
		GLOVE	61.2 (60)		64.8 (524)	66.1 (944)	65.6 (1,904)	66.4 (2,950)	66.1 (3,972)
		DistilBERT	73.5 (60)	NGDG	73.2 (180)	73.3 (280)	73.1 (504)	72.9 (700)	72.9 (944)
		GLOVE	61.2 (60)		63.1 (172)	62.5 (280)	59.3 (500)	58.1 (708)	57.9 (958)
Dbepdia # Examples	78.6 (42)	DistilRoBERTa	68.3 (42)	MCTS	73.6 (303)	75.0 (581)	75.9 (1,067)	76.4 (1,531)	75.4 (2,006)
		GLOVE	72.1 (42)		72.3 (396)	74.0 (737)	74.1 (1,390)	73.7 (2,104)	74.6 (2,817)
		DistilRoBERTa	68.3 (42)	NGDG	73.9 (338)	74.4 (635)	73.7 (1,204)	72.9 (1,735)	72.5 (1,992)
		GLOVE	72.1 (42)		68.8 (389)	70.6 (804)	70.0 (1,562)	71.3 (2,225)	71.2 (2,584)
Yelp-2 # Examples	77.7 (20)	DistilRoBERTa	72.3 (20)	MCTS	78.5 (898)	77.7 (1,817)	77.9 (3,467)	77.8 (5,185)	78.6 (6,831)
				NGDG	73.5 (283)	74.4 (548)	76.4 (1,030)	76.5 (1,487)	76.8 (1,944)
Yelp-2 # Examples	81.9 (40)	DistilBERT	75.7 (40)	MCTS	78.6 (502)	78.7 (980)	79.2 (1,908)	79.2 (2,792)	79.8 (3,860)
				NGDG	79.4 (432)	79.0 (812)	79.5 (1,508)	79.9 (2,298)	79.9 (2,298)
IMDB # Examples	77.2 (80)	DistilBERT	70.2 (80)	MCTS	73.4 (824)	73.9 (1,552)	74.5 (3,086)	74.5 (4,610)	74.9 (6,054)
				NGDG	71.4 (454)	71.8 (848)	71.6 (1,610)	71.8 (1,788)	71.1 (1,788)

5.5.5 Discussion

Overall, the performance of the student model has always increased with our approach, and even in some instances it exceeded the performance of its teacher, as in the TREC-6 DistilBert experiment and the AG-news DistilRoberta experiment. As the number of runs increases,

TREC-6 Examples	**Teacher**	**Student**
What do Americans hate about their country?	ENTY	HUM
How are Mormons allowed to become priests?	DESC	HUM
Where is The White House located?	LOC	DESC
How come many are fat?	DESC	NUM
What country has outlawed marijuana use?	ENTY	LOC
What city became popular in 1991?	ENTY	HUM
SST-2 Examples		
and one hell of a movie	pos	neg
it just isnt worth your seven bucks	neg	pos
a remarkable diversion from its previous film	pos	neg
it does look and feel like a flimsy piece of furniture	neg	pos
it has no real emotional value	pos	neg

Table 5.3 Examples of data generated for TREC-6 and SST-2. Wrongly predicted labels are colored in red

MCTS continues to construct paths that maximize its reward function. This is noticeable from the results, as with higher iterations, the student's performance tends to increase, narrowing the gap with its teacher. Furthermore, we added experiments with GLOVE embeddings to show that this approach is applicable regardless of the strengths or weaknesses of the learning model. For 100-dimension vectors that are context insensitive, we were able to achieve a considerable increase of 12% on TREC-6, and more than 5% with SST-2.

Table 5.3 shows samples of generated data in the TREC-6 and SST-2 experiments. We can see instances where the teacher model corrected the student as in the first four TREC-6 and SST-2 examples, and instances where the teacher provided incorrect labels. These examples were all pseudo-labeled by the teacher with relatively high confidence (above 90%), yet not all were correctly labeled. This suggests that regardless of the teacher's overconfident mispredictions, the sheer amount of generated data contained enough correctly pseudo-labeled instances, in the high Δent regions, to outweigh the wrong ones. Furthermore, this shows that the student can only improve as much as its teacher is able to provide good labels.

We set the MCTS sequence length to 120, and 240 to reduce the generation time. With datasets like IMDB, and YELP-2 where the average number of tokens per example in the sampled training set is 300, and 180 respectively, increasing the pruning condition for MCTS could have resulted in more complete contexts, leading to higher improvements. Nevertheless, our motive is not to extract the best possible accuracy rates, but rather to show that this approach can in fact enhance knowledge distillation on small datasets.

5.6 Experiment 2

In this section, we compare data generation using MCTS with the Non-Guided Data Generation approach (NGDG) from chapter 3. We also took a more conventional approach of not freezing the encoder's layers. However, to mitigate the impact of instability caused by the BERT-based models on small datasets, we applied the configurations suggested by Mosbach et al. (2020). As such, we applied the ADAM optimizer (Kingma and Ba, 2014) with a bias correction to avoid vanishing gradients in early training steps. We then performed training for 40 epochs with a learning rate of 2×10^{-5} that is linearly increased for the first 10% of the total training steps and linearly decayed to zero afterward.

Furthermore, we limited our experiments to RoBERTa and a distilled version of it, DistilRoBERTa (Sanh et al., 2019). For the teacher, we used the 24-layer RoBERTa, and 2 variants of DistilRoBERTa for the student. We carried out the first set of experiments with the original 6-layer DistilRoBERTa as the student. We then repeated the same experiments with a 3-layer DistilRoBERTa, a result of removing half the layers from the 6-layer version. The number of parameters amount to more than 355 million in the 24-layer RoBERTa, and 82.1 million in the 6-layer DistilRoBERTa. By removing 3 layers from DistilRoBERTa, we managed to further reduce the number of parameters to 60.8 million. For each model, we added a classification head that consisted of a linear layer followed by a ReLU activation, a 0.1 dropout layer, and a linear output layer.

5.6.1 Datasets

In an attempt to evaluate our approach under different settings, we considered datasets of multiple sequence classification tasks. In this section, we experiment with the same datasets sampled in section 5.5.2, with an additional dataset, the Microsoft Research Paraphrase Corpus (MRPC). The MRPC is for a sentence-pair classification task in which a model is given two sentences and has to predict if they are semantically equivalent or not. The MRPC consists of 3,668 training examples of which we sampled 600 per label, making a total of 1,200 training samples. Our evaluations are performed on the test set, that contains 1,725 examples.

5.6.2 Configurations

We configured both MCTS and NGDG to run with a different number of iterations in each experiment, and set the top-k sampling to $k = 20$ in all our experiments. We also added a pruning criterion to limit the maximum length of any path to 120 tokens for all

Table 5.4 Teacher-Student Results (in percent, numbers of added examples in parentheses below). We adopt the GLUE benchmark metrics for the MRPC experiments by showing the accuracy and F1-score scores, displayed as Accuracy/F1. Generated data is filtered on a teacher's confidence above 0.7

Dataset	Teacher	Student (Pre-Distillation)		Approach	Student (Post-Distillation)							
	RoBERTa large	DistilRoBERTa	Start		100	500	1k	2.5k	5k	10k	15k	20k
SST-2 # Examples	89.9 (60)	6-layers	78.2 (60)	MCTS	81.8 (92)	85.1 (204)	86.1 (340)	85.9 (790)	86.8 (1500)	86.8 (2952)	**87** (4474)	86.2 (6048)
		3-layers	68.2 (60)		73.6 (124)	78.4 (342)	78.6 (604)	79.9 (1480)	82 (2946)	83.7 (5942)	83.8 (9000)	**84.1** (12166)
		6-layers	78.2 (60)	NGDG	82.8 (64)	84.1 (96)	83.5 (134)	85.1 (226)	85.3 (402)	85.2 (714)	86.5 (1050)	**87.2** (1344)
		3-layers	68.2 (60)		68.8 (68)	71.9 (124)	76.7 (196)	78.7 (364)	80.8 (692)	81.4 (1266)	**82.9** (1884)	82 (2396)
DBpedia # Examples	92.3 (42)	6-layers	80.4 (42)	MCTS	85.8 (56)	92.6 (132)	93.2 (197)	93.7 (444)	93.2 (790)	94 (1553)	94.2 (2387)	**94.4** (3140)
		3-layers	41.4 (42)		52.3 (53)	75.3 (113)	87.5 (184)	89.3 (396)	**90.5** (811)	89.4 (1594)	89.5 (2364)	89.8 (3126)
		6-layers	80.4 (42)	NGDG	83.9 (52)	90.5 (93)	**93.6** (180)	93.1 (318)	93.1 (610)	92.9 (1181)	93.6 (1792)	93.4 (2382)
		3-layers	41.4 (42)		54.7 (52)	74 (93)	84.7 (180)	86.1 (318)	88.3 (610)	88.2 (1181)	89.2 (1792)	**89.7** (2382)
TREC-6 # Examples	89 (76)	6-layers	80 (76)	MCTS	79.8 (100)	82 (194)	83.6 (358)	85.6 (1012)	87.8 (2240)	**88** (4606)	87.8 (7004)	86.4 (9324)
		3-layers	62 (76)		71 (94)	77 (242)	79.6 (416)	81.2 (1155)	79.8 (2478)	81.4 (4845)	81.6 (7106)	**84** (9182)
		6-layers	80 (76)	NGDG	79.6 (85)	79.2 (125)	80.4 (158)	80.8 (280)	83.4 (481)	86.8 (885)	85.6 (1253)	**87** (1631)
		3-layers	62 (76)		70 (86)	70.4 (126)	74 (166)	76.8 (300)	78.8 (498)	81 (909)	81.2 (1290)	**82** (1682)
MRPC # Examples	84.5/88.1 (1200)	6-layers	77.9/82.3 (1200)	MCTS	79.5/83.5 (1262)	79.9/84.3 (1514)	81.2/85.2 (1827)	80.9/85.2 (2729)	81.5/85.3 (4252)	81.6/85.5 (7351)	82.6/86.5 (10362)	**83.8/87.4** (13372)
		3-layers	69/75.3 (1200)		69/74.8 (1236)	74.1/80.1 (1365)	75.1/80.4 (1551)	76.6/81.7 (2158)	77/81.8 (3116)	77/81.4 (5046)	77.4/81.7 (6916)	**78.2/82.8** (8766)
		6-layers	77.9 (1200)	NGDG	79.1 (1241)	79.8 (1461)	80.3 (1685)	80.3 (2408)	82.6 (3597)	**82.8** (6051)	82.7 (8396)	82 (10934)
		3-layers	69 (1200)		70.8 (1253)	74.3 (1489)	75.8 (1790)	74.7 (2695)	76.5 (4152)	**78** (7111)	76.6 (10059)	77.8 (12959)
Yelp-2 # Examples	82.6 (40)	6-layers	**85.2** (40)	MCTS	80.4 (104)	80 (344)	82.5 (656)	80.7 (1618)	80.1 (3116)	78.7 (6254)	79.6 (9340)	79.5 (12280)
		3-layers	73.9 (40)		76 (114)	74 (432)	76 (828)	77.7 (2056)	78 (4146)	79 (8356)	**79.4** (12496)	78.9 (16498)
		6-layers	**85.2** (40)	NGDG	84.6 (78)	81.3 (246)	81.3 (430)	81.2 (1052)	81.1 (2074)	79.6 (4246)	79.9 (6302)	79.8 (8330)
		3-layers	73.9 (40)		74.9 (44)	76.4 (308)	76.9 (570)	76.4 (1372)	77.7 (2734)	77.6 (5580)	**78.4** (8326)	78.2 (11050)

our experiments. For MCTS, we set the UCB constant C from equation 5.3 to $C = 3$. As for the reward function, only for the TREC-6 experiment, we added a heuristic (see 'task specific condition' in equation 5.7) to condition the first token to be a question word by returning -1 if it was not 'what', 'where', 'when', 'who', 'which', 'why', or 'how'. For each task, we made sure the generated text was in the appropriate format for the RoBERTa classification models. This meant setting reward = 0 for GPT-2 outputs that are not in the format $< |endoftext| > x_1, \ldots x_N < |endoftext| >$ for the single input sentence tasks (TREC-6, SST-2, Yelp, and DBpedia). As for MRPC, where the input is 2 sentences, the generated data has to follow the format $< |endoftext| > x_1, \ldots x_N, [SEP], y_1, \ldots y_N < |endoftext| >$, where $x_1 \ldots x_N$ and $y_1 \ldots y_N$ are sequences of tokens. The [SEP] token acts as a separator

Table 5.5 Teacher-Student Results (in percent, numbers of added examples in parentheses below). We adopt the GLUE benchmark metrics for the MRPC experiments by showing the accuracy and F1-score scores, displayed as Accuracy/F1.

Dataset	Teacher	Student (Pre-Distillation)		Approach	Student (Post-Distillation)							
	RoBERTa large	DistilRoBERTa	Start		100	500	1k	2.5k	5k	10k	15k	20k
SST-2 # Examples	89.9 (60)	6-layers	78.2 (60)	MCTG	81.8 (92)	86 (204)	85.6 (344)	86.2 (794)	86.1 (1508)	86.5 (2964)	86.1 (4496)	**86.8** (6074)
		3-layers	68.2 (60)		75 (128)	77.1 (362)	77.3 (646)	79.9 (1564)	82.3 (3104)	83.6 (6258)	**84.4** (9494)	83.3 (12834)
		6-layers	78.2 (60)	NTTG	82.8 (64)	84.1 (96)	83.5 (134)	85.1 (226)	85.3 (402)	86.1 (716)	**86.7** (1054)	85.7 (1348)
		3-layers	68.2 (60)		68.8 (68)	71.9 (124)	75.5 (202)	76.8 (384)	80.7 (734)	82.3 (1338)	82.8 (1998)	**83.4** (2536)
		6-layers	78.2 (60)	RWR	79.4 (160)	81.2 (557)	81.4 (1045)	81.9 (2468)	**83.1** (4718)	82.1 (8816)	81 (12442)	81 (15785)
		3-layers	68.2 (60)		68.7 (160)	71.2 (557)	72.2 (1045)	75 (2468)	75.5 (4718)	76.1 (8816)	76.6 (12442)	**78.8** (15785)
DBpedia # Examples	92.3 (42)	6-layers	80.4 (42)	MCTG	91 (101)	**93** (407)	92.7 (801)	92.5 (2002)	92 (3976)	92.2 (8005)	92.2 (11944)	92.3 (15944)
		3-layers	41.4 (42)		80.2 (119)	88 (435)	89.5 (824)	91.2 (1995)	**91.6** (3940)	91.4 (8066)	91.1 (12067)	91.2 (16190)
		6-layers	80.4 (42)	NTTG	83.9 (52)	89 (101)	92.9 (364)	92.5 (661)	**93.1** (1602)	93 (3191)	92.9 (6241)	92.9 (12423)
		3-layers	41.4 (42)		78.3 (103)	88.6 (383)	89.8 (687)	90.3 (1653)	90.6 (3271)	91.1 (6407)	91.6 (9588)	**91.7** (12773)
		6-layers	80.4 (42)	RWR	88.4 (142)	87.9 (541)	87.3 (1041)	88.8 (2536)	88.8 (5029)	**89.9** (9996)	88.5 (14941)	88.6 (19878)
		3-layers	41.4 (42)		78.9 (142)	80.8 (541)	81.5 (1041)	83.3 (2536)	84.1 (5029)	83.9 (9996)	**85.1** (14941)	85 (19878)
TREC-6 # Examples	89 (76)	6-layers	80 (76)	MCTG	80.2 (103)	83 (213)	83.8 (402)	86.2 (1134)	**88** (2508)	86.6 (5290)	87.8 (8062)	88 (10787)
		3-layers	62 (76)		72 (96)	77 (275)	79 (500)	79.6 (1446)	82 (3133)	82.6 (6039)	81.6 (8815)	**83.2** (11444)
		6-layers	80 (76)	NTTG	79.6 (85)	77 (130)	78.8 (163)	82.8 (322)	83.4 (537)	85 (987)	84.2 (1402)	**85.6** (1822)
		3-layers	62 (76)		70 (86)	71.2 (137)	72.8 (177)	78.8 (352)	81 (594)	81.8 (1093)	**82.8** (1562)	82.6 (2019)
		6-layers	80 (76)	RWR	78.6 (176)	81.2 (573)	78.8 (1065)	79.8 (2511)	80 (4874)	80.8 (9296)	**82.8** (13384)	81 (17153)
		3-layers	62 (76)		69.8 (176)	76.2 (573)	74.6 (1065)	75.6 (2511)	78.4 (4874)	78 (9296)	79.8 (13384)	**80.8** (17153)
MRPC # Examples	84.5/88.1 (1200)	6-layers	77.9/82.3 (1200)	MCTG	79.5/83.5 (1262)	79.9/84.3 (1514)	81.2/85.2 (1827)	80.9/85.2 (2729)	81.5/85.6 (4252)	82.6/86.6 (7353)	82.1/85.9 (10366)	**83.1/86.7** (13378)
		3-layers	69/75.3 (1200)		69/74.7 (1236)	71.2/76.8 (1367)	74.6/79.9 (1553)	75.5/80.7 (2160)	77.9/82.6 (3118)	77.5/82.1 (5048)	77.8/82.6 (6918)	**78.8/83.5** (8768)
		6-layers	77.9 (1200)	NTTG	79.1 (1241)	79.9 (1461)	80 (1687)	81 (2410)	81.6 (3599)	81.7 (6055)	**82.7** (8404)	82.3 (10948)
		3-layers	69 (1200)		70.8 (1253)	74.3 (1489)	75.9 (1794)	76.4 (2699)	77.4 (4156)	76.8 (7115)	**78.1** (10067)	78.1 (12969)
		6-layers	77.9 (1200)	RWR	79.6 (1300)	81 (1700)	80.9 (2200)	81.5 (3700)	82.2 (6200)	81.2 (11200)	82.1 (16199)	82.7 (21199)
		3-layers	69 (1200)		68.1 (1300)	70.2 (1700)	70 (2200)	72.9 (3700)	73.3 (6200)	73.4 (11200)	72.5 (16199)	72.1 (21199)
Yelp-2 # Examples	82.6 (40)	6-layers	**85.2** (40)	MCTG	80.4 (104)	79.6 (346)	81.9 (660)	80.4 (1626)	80.7 (3124)	79.2 (6268)	79.4 (9356)	79.5 (12302)
		3-layers	73.9 (40)		75.4 (118)	74.9 (454)	76.4 (856)	77.9 (2122)	78.8 (4278)	78.8 (8648)	79.1 (12928)	**79.3** (17100)
		6-layers	**85.2** (40)	NTTG	84.6 (78)	81.3 (246)	81.3 (430)	81.2 (1052)	81.1 (2074)	80.5 (4250)	80.2 (6310)	79.9 (8344)
		3-layers	73.9 (40)		74.9 (44)	75.3 (314)	76.2 (584)	76.6 (1410)	77.6 (2828)	78.4 (5774)	**79** (8596)	78.8 (11404)
		6-layers	85.2 (40)	RWR	85.3 (140)	82.9 (540)	85.2 (1040)	85.9 (2540)	81.8 (5040)	81.1 (10040)	81.3 (15040)	80.5 (20040)
		3-layers	73.9 (40)		76.8 (140)	77.5 (540)	77.3 (1040)	76.7 (2540)	76.9 (5040)	76.3 (10040)	73.6 (15040)	73.1 (20040)

between two input segments to allow the language model to differentiate between the two. As shown in section 2.3.2, [SEP] was included in BERT's pretraining for the next sentence prediction task; given two sentences separated by [SEP], the model is trained to predict whether they are from the same context. As RoBERTa is an extension of BERT, the [SEP] token is also included in its pretraining. However, GPT-2 was only trained to predict the next token in a sequence, which does not require separating inputs into segments. This meant that a separator token is not present in GPT-2's vocabulary, and as such needed to be added. Note that the MRPC was the only dataset that required the introduction of the [SEP] token. Considering the performance improvements for the MRPC task in Tables 5.4 and 5.5, the 1,200 seed examples were enough to teach GPT-2 on the use of [SEP].

After the data was generated, we applied the following selection conditions to avoid high data imbalance: a) For binary datasets, we limited the number of added examples to the size of the minority class. b) For the multiclass datasets, we simply limited the selection over the median, given the distribution of generated examples per label. We then added the selected data to the initial training data, to form a new training set. The new set consists of the initial training samples, e.g. 60 SST-2 examples, with the generated data pseudo-labeled with the probabilities predicted by the teacher. For training, we applied the cross-entropy loss function from equation 5.8 and fixed hyperparameters for the teacher and student classifiers, as explained in section 5.5.1. We did not perform any temperature scaling on the softmax probabilities, similar to $T = 1$ in equation 5.1. For the mentioned configurations, we perform two sets of experiments, one with the generated data filtered on teacher confidence above 0.7, and one without any additional filtering.

5.6.3 Results

Results for the teacher-student knowledge transfer experiments are in Tables 5.4 and 5.5. In Table 5.4, we show the student results after training on the generated data with teacher's confidence above 0.7. We repeated the same experiments but without conditioning on the teacher's confidence, results are shown in Table 5.5. In Table 5.5, we experimented with a baseline approach, Random Word Replacement (RWR). RWR augments the training data by applying the 12-layer pretrained BERT model to predict a substitute word for a masked token. Each token in an input has a 10% probability of being masked, i.e., replaced by a BERT prediction Kobayashi (2018). Similar to our GPT-2 generation for the MCTG and NTTG experiments, the replacement token is selected from the top-20 tokens given BERT's probabilities (Wu et al., 2019). We show the test accuracy of both the teacher and student (Pre-Distillation) on the sampled data from section 5.6.1, and that of the student after it has been trained on the sampled data combined with the generated data (Post-Distillation).

Underneath each accuracy score, in parentheses, is the size of the training data (# Training examples). For the teacher and the student prior to distillation (labeled Start in the table), the data sizes are those of the initial training sets, described in section 5.6.1. As explained in section 5.5.1, the teacher model is a 24-layer RoBERTa, and the student model is either a 6-layer DistilRoBERTa, or a 3-layer DistilRoBERTa. The "Start" accuracy is achieved after training only on the initial dataset. For example, 'RoBERTa large' trained on the SST-2 dataset of 60 examples produced a test score of 89.9. This is a much higher result compared to the 78.2 accuracy of the 6-layer DistilRoBERTa model that is trained on the same dataset. We then applied MCTS, from section 5.3.2, and NGDG, from section 5.3.4 to generate distillation data. We fixed the total number of iterations to 20k, and stored checkpoints at iterations 100, 500, 1k, 2.5K, 5K, 10K, 15K, and 20k. After each iteration, one sample is generated. Yet, not every sample is used for the distillation process. In fact, a sample can only become a candidate for distillation if its Δent (equation 5.6) is positive, and it passes the conditions set in equation 5.7. This makes the size of the distillation data less than the number of generated samples. For example, in the SST-2 experiment in Table 5.4, after 100 iterations of MCTS with the 6-layer DistilRoBERTa, 32 of the 100 generated samples had a positive value for the conditions in equation 5.7. As a result, the 32 were selected and added to the 60 initial training data, making a total of 92 training samples. After training DistilRoBERTa on the 92 examples, the model scored a test accuracy of 81.8, achieving a 2.9% increase in performance over its initial value of 78.2. When training on the 4474 examples of the 15k run, DistilRoBERTa produced its highest performance of 87 (in bold), making it closer to the teacher score of 89.9.

5.6.4 Discussion

Overall, results in Tables 5.4 and 5.5 show that our approach works well with either MCTS or NGDG. Table 5.5 shows that both NTTG and MCTG, lead to better performance improvements over the RWR baseline. Only in MRPC, the results are similar for the 6-layer student, which could be attributed to a lower performance gap between the student and the teacher. It is evident that a good teacher can always increase the performance of its student, provided that enough examples achieve a positive Δent (equation 5.6). This shows that regardless of the generation method, equation 5.6 remains a key component to our approach. Moreover, we note that there is a small overall gain from restricting the selected training data to high teacher confidence. On average, the maximum gain with MCTG from restricting data based on the teacher's confidence is 0.1375 over the SST-2, TREC-6, DBpedia, and MRPC experiments. Note that the maximum accuracies in Tables 5.4, and 5.5 are in bold. With NTTG, the maximum accuracy drops by an average of -0.1125 when filtering the generated data by

teacher's confidence. These results suggest that restricting the data on teacher's confidence has little effect on the distillation performance. Considering that all other factors like seed values were fixed, we note that filtering by the teacher's confidence merely changed the point at which gradients converge to a different local minimum during training, see section 2.2.4.

We note that in some instances, the student even exceeded the performance of its teacher, as in the 6-layer DistilRoBerta experiment with MCTS on DBpedia. As the number of runs increases, MCTS continues to construct paths that maximize its reward function. This is noticeable from the results of higher iterations, where the student's performance tends to increase, narrowing the gap with its teacher. Similarly, in NGDG, as more data is generated, useful examples are more likely to appear. This explains the strong results that are comparable to MCTS. With MCTS, at 100 iterations, the performance matches that of NGDG, suggesting that at this stage, MCTS is in the exploration phase. Hence, the generated examples are likely to be random. As the number of iterations increases, the number of generated samples starts to exceed that of NGDG. This suggests that MCTS at this stage has constructed enough paths, so that it is able to exploit the ones that maximize its reward function. This is also reflected in the relatively higher scores compared to NGDG for the DBpedia and MRPC experiments. The ability to apply task-specific heuristics during the generation process, as with the TREC-6 experiments, enables MCTS to converge faster than NGDG to better results. This is evident from the massive increase of generated examples over NGDG. Furthermore, in future work, MCTS can be extended to include adjustments such as the inclusion of multiple teachers in its reward function. To show the importance of having a good teacher, we intentionally selected a dataset (Yelp-2), in which the 6-layer student outperforms its teacher. Here, because the teacher can sometimes be overconfident about its incorrect predictions, they are nevertheless treated as correct by the student, degrading its performance. This shows that the student can only improve as much as its teacher is able to provide good labels.

To further investigate the change of performance with the initial model, we run 10 training instances of the student model on the initial data and on the selected pseudo-labeled data from the 20k run. In accordance with the configurations of the experiments in Table 5.5, we do not filter the pseudo-labeled data on the teacher's confidence. In Table 5.6, we show the test results as an average of the 10 models. These results are consistent with Table 5.5. Overall, the augmented data leads to better and more stable models, indicated by the higher accuracy and lower variance.

Finally, in Table 5.7, we show some generated data from the TREC-6 and SST-2 experiments. All six examples are remarkably grammatical, natural, and well-formed. The TREC-6 examples are good questions, while the SST-2 ones genuinely express subtle sentiments.

Task	Approach	3-Layers		6-Layers	
		Start	20K	Start	20K
SST-2	MCTG	68.2(±1.67)	83.7(±0.32)	81.85(±1.26)	86.89(±0.396)
	NTTG	67.7(±1)	82(±0.69)	81.3(±0.869)	86.9(±0.4)
TREC-6	MCTG	59.86(±4.97)	83(±0.7)	78.6(±1.77)	88(±0.51)
	NTTG	60.15(±4.68)	82.6(±0.84)	79.08(±1.65)	86.3(±0.8)
MRPC	MCTG	68.5(±2.28)	77.5(±0.55)	79.7(±0.91)	82.7(±0.4)
	NTTG	68.2(±2.69)	77.6(±0.49)	79.5(±0.71)	81.5(±0.7)
DBpedia	MCTG	52.18(±5.06)	91.3(±0.16)	86(±3.7)	92.3(±0.059)
	NTTG	55.7(±4.7)	91.7(±0.13)	85.9(±3.8)	92.8(±0.08)
Yelp-2	MCTG	72.27(±2.84)	79.2(±0.183)	83.29(±1.34)	79.4(±0.175)
	NTTG	72.4(±2.3)	78.5(±0.167)	83.95(±1.07)	79.9(±0.22)

Table 5.6 Average and standard deviation, displayed as average(± standard deviation), of test accuracy for 10 student model (3-layers and 6-layers) instances, trained on the initially sampled data and the pseudo-labeled data from the 20k MCTG and NTTG runs

TREC-6 Examples	Teacher	Student
What is virtual reality?	DESC	ENTY
What language was originally spoken by the Indians?	ENTY	LOC
Where in China do I find the most expensive typewriter?	LOC	HUM
SST-2 Examples		
a trip from good to bad	neg	pos
the kind of script worth watching	pos	neg
a step down from her best years.	neg	pos

Table 5.7 Examples of data generated for TREC-6 and SST-2. Wrongly predicted labels are colored in red

5.7 Conclusion

In this chapter, we provided an approach for textual data generation to improve knowledge distillation on small datasets. With the implementation of a reward function that conditions the generated examples to be predicted with the lowest uncertainty by the teacher and highest uncertainty by the student, we were able to improve the student's performance to replicate the performance of its teacher. Considering the results, we could argue that reward-based language generation can complement or even substitute for heuristic data augmentation approaches in knowledge distillation. We believe that the provided implementation can serve

as a baseline for reward-based textual data generation approaches in small data settings. This will hopefully motivate future research to extend this or to explore new reward-based generation methods.

Chapter 6

Conclusion

Throughout this thesis, we have made contributions towards data augmentation for small datasets in the field of Natural Language Processing. In this chapter, we will summarize the proposed methods and provide an outlook for future research.

6.1 Synopsis

Chapter 3 proposed a method for generating training data with human annotations. In this chapter, we studied the efficacy of the transformer-based language model, GPT-2, in learning meaningful representations from very small datasets. After fine-tuning GPT-2 on as little as a few examples per label, we were able to generate meaningful out-of-distribution samples. The linguistic information that GPT-2 captures from its fine-tuning on the training samples enables it to produce relevant textual sequences. By manually labeling the generated data, we extended the training set to explicitly introduce new information. The high classification entropy for the added samples indicated that either the initial training set lacks information or that the classifier is unable to detect hidden data patterns. By generating data samples we attempted to introduce new information, and by manually labeling the added samples, we explicitly introduced additional data patterns. This in turn helped the classifier better learn the target task. To better improve the learning classifier, we attempted to generate the samples that could best add information. For this, we took advantage of Shannon's Entropy metric to rank the generated samples. As the classifier shows the most confusion for samples with the highest entropy, we expected these samples to add more information to its learning. This also means the user is able to only annotate a sample of the data ranked by entropy, instead of labeling all the generated data. In searching for generated samples with highest entropy, we experimented with the Monte Carlo Tree Search (MCTS) algorithm. We implemented MCTS so that each path it takes represents a generated text sequence, where each state corresponds

to a token. The key component in MCTS is in its reward function, as the search objective is to find paths that return maximum rewards. While each path corresponds to a generated sequence, its reward value can be based on the classifier's entropy. Here we highlighted the use of entropy as a reward for each path, where the objective of MCTS is to find paths that maximize entropy, thus generating sequences with maximum classification confusion. We note that the Upper Confidence Bound (UCB) function balances between random paths, and those that maximize the reward function. This allows MCTS to find suitable paths while avoiding getting stuck in certain sub-paths. We tested our approach against a Non-Guided Data Generation (NGDG) process that does not optimize for a reward function. Starting with randomly sampled subsets, our results showed an increased performance with MCTS of 26% on the TREC-6 Questions dataset, and 10% on the Stanford Sentiment Treebank SST-2 dataset. Compared with NGDG, we were able to achieve increases of 3% and 5% on TREC-6 and SST-2.

Chapter 4 studied the possibility of automating the labeling process of the generated data. In this chapter, we implemented a self-learning process in which the classifier is retrained on its pseudo labels. This process substituted the external element of user knowledge with the pseudo-labels generated by the learning classifier. However, these pseudo labels are not guaranteed to be accurate, and could hinder performance if added to the classifier's training data.

Furthermore, in chapter 3, the user not only labeled the data, but also assessed its quality. This meant that the user was also responsible for selecting which data samples to annotate and include in the training data of the classifier. Hence, removing the user from the labeling process adds a layer of complexity that requires the identification of bad samples. Identifying bad samples that cannot be labeled remains an open problem in research. We can relate the noise in the generated sample to Aleatoric uncertainty, and the classification confusion of non-noisy samples to Epistemic uncertainty. However, quantifying Aleatoric and Epistemic uncertainties remains a research challenge. In this chapter, we investigated the possibility of measuring distributional shifts between generated data and the classifier's training data. For this measurement, we used the trained classifier to transform each input sample to a 2-dimensional vector. This allowed us to create clusters for the training data, where each label corresponded to a cluster. The distance was then measured between every generated sample and its closest cluster. This meant we relied on the classifier itself to provide both the classification entropy and its measure of distance between a generated sample and its closest training cluster. To generate data samples, similar to chapter 3, we applied MCTS by optimizing for the maximum distance between a generated sample and its closest training

cluster. Unfortunately, this process does not always guarantee to improve performance. We investigated the reasons behind this by matching each label predicted with the highest confidence against the label of the closest cluster. As expected, we found consistent results which suggested that the distance between a generated sample and its closest cluster is at its lowest when the final classification confidence is at its highest. In other words, examples that are distant from their training clusters exhibit low classification confidence, and vise versa. This suggests that any improvement gained from this process could be contributed to simply augmenting the training samples, and not necessarily from the ability to find a distinction between Aleatoric and Epistemic uncertainties.

Chapter 5 extended the data augmentation approach from chapter 3. Inspired by our work in chapter 4, we relied on a teacher model to create pseudo-labels for the generated data. Instead of focusing on self-supervision, we switched to a two-model scheme to address the predicament of data scarcity in knowledge distillation. Under this new scheme, we proposed a novel approach where knowledge is distilled from a teacher model to a student model. In this approach, we replaced the human labels by pseudo labels of a teacher model. Here, instead of manual data filtering, the teacher and student model played the role of selecting training samples from the generated data. The process starts by first fine-tuning the teacher and student models, as well as a language generation model, on the target task dataset. The Monte Carlo Tree Search (MCTS) algorithm is then applied to generate the examples that can be most informative to the student, yet labeled with the highest certainty by the teacher. For any generated example, we take the difference in Shannon's entropy between the student and teacher model Δent. The more uncertain the student is, or the more certain the teacher is, the higher Δent will be. With this function, we were able to select the examples that the student is mostly confused about, yet the teacher is able to confidently label. In this way, the student and teacher worked together to condition the language generation model to generate examples that can enhance the performance of the student model. The student was then trained on the generated data with the predicted probabilities of the teacher model. We tested MCTS against a baseline in which examples are randomly generated with top-k sampling. By testing this approach on the SST-2, MRPC, YELP-2, DBpedia, and TREC-6 datasets, we consistently witnessed improved performance.

6.1.1 Final remarks

We have come to the conclusion that synthetically generated data can in fact complement an existing dataset. This is evident from the increased performance reached in both chapters 3, and 5. However, with this being said, in all our experiments, the applied language generation

model, GPT-2, was able to generate data similar to the distribution of the training data. Although none of the datasets used were included in the pretraining of GPT-2, we cannot guarantee that the model had not previously seen similar examples. For instance, SST-2, IMDB and Yelp-2 are sentiment analysis tasks. The example "I hate the word 'perfume,' Burr says." contains a negative sentiment and is included in the GPT-2's WebText corpus (Radford et al., 2019), refer to section 2.3.2.

We have also seen that Shannon's Entropy played a crucial role in determining the examples that are beneficial to the learning classifier, chapter 3. In chapter 5, entropy was essential in both finding the examples the student needed for its learning, and those that the teacher was able to confidently label. The application of Monte Carlo Tree Search (MCTS) helped optimize the search for examples that satisfy the learning criteria by guiding GPT-2 during its generation process. With MCTS, there was a risk of getting stuck in locally optimal states during the generation process, which would lead to multiple sequence outputs with parent nodes sharing the same paths to the root node. This problem is further explained in section 3.5. In chapter 3, we added a text similarity measure to penalize current outputs that are semantically similar to previously generated outputs. We called this approach diversity-based MCTS, refer to section 3.4.3, where we used a language model, the Universal Sentence Encoder (Cer et al., 2018), to encode textual sequences to vector representations for semantic similarity measures. Computing the similarity between text sequences added an overhead to the already extensive generation process. In addition to indexing embedding vectors of already generated sequences, this process required loading and making calls to the language model for computing similarity scores. Nevertheless, the diversity-based MCTS generated more diverse outputs, making it easier for the user to select and label data. In the experiments of section 3.4, we conditioned the user only to label the top N sequences, sorted by their MCTS rewards, for each active learning run. With the diversity-based MCTS, it was unlikely for the top N generated samples to start with the same sequence of tokens. This meant, on average, that more diverse samples could be labeled in each round.

In chapters 4, and 5, we relied on a teacher model to generate pseudo-labels, and thus relieve the user from manually assigning labels. In this approach, we were able to generate and pseudo-label thousands of examples without the user's interference. This meant it was sufficient to achieve desirable outputs by simply adjusting the UCB constant C from equation 2.70, instead of implementing a diversity-based MCTS.

Finally, we have seen the importance of domain knowledge in labeling augmented or generated textual data. Unlike computer vision, the simplest changes to textual data can result in a change of meaning and thus deem label preservation inapplicable. For instance, in a cat and dog image classification task, rotations or color shifts are less likely to result in loss

of information about the subject. Hence, a dog in an image will most likely remain a dog even if colors are reversed or if the image is rotated by 90 degrees. This makes it possible to retain image labels when simple augmentation techniques are applied to those images.

By contrast, data augmentation is not as common in NLP as it is in computer vision. This is most probably related to the difficulty of preserving the meaning of an augmented sentence. Unlike image data, the discrete nature of text inputs makes it difficult to maintain invariance (Feng et al., 2021). For instance, by shuffling words, the sentence "This is a good movie" could become "is this a good movie", resulting in a change to the meaning. In a sentiment classification task, the original sentence would clearly have a positive meaning, but that's not necessarily true for the augmented sentence. Considering the challenges of data transformation in NLP, in this thesis, we augment textual data by synthesizing new samples from the available training data. Since label preservation is not possible with this approach, we rely on an external source for data labeling. The source could be: a) A human annotator, as in chapter 3, or b) A model trained on the available training data. In chapter 4, we attempted to apply self-learning, but the results were not very convincing, refer to section 4.6. In chapter 5, the labeling knowledge came from a teacher model. As explained in section 2.10, larger deep learning architectures tend to outperform smaller ones. However, larger models come at the cost of storage space, time, and computational requirements. In return, performance compromise has to be made when opting for smaller models. Nonetheless, in chapter 5, we rely on the teacher's pseudo-labels to limit the sacrifice in performance to as low as near zero. By pseudo-labeling synthetically generated textual samples, this approach becomes particularly useful in situations where data is scarce.

6.2 Future Directions

In this section, we will give an overview of possible directions for future work in data augmentation for text classification.

Out-of-distribution detection In chapter 4, we attempted to find data samples that are further from the distribution of the training data by injecting in the learning classifier a linear layer that outputs 2-dimensional vectors. We used the outputs of this layer to measure the distance between a text input and the training data. We later found out that the distance between an input and its closest training samples decreases as the classification confidence increases. We believe more can be done to improve out-of-distribution detection, such that samples that are further from their training data, yet labeled with low entropy, can improve classification performance in self-learning if appended to the training data.

Quantification of Aleatoric and Epistemic Entropies Traditional approaches in machine learning fail to distinguish between different sources of uncertainty (Hora, 1996). The Aleatoric uncertainty reflects noise in data. It usually cannot be reduced even by collecting more data. In contrast, the Epistemic uncertainty reflects the uncertainty by the learning model. The generalization error can be decomposed into Epistemic and Aleatoric uncertainties. To create a training set from synthetically generated data, distinguishing between Epistemic and Aleatoric becomes vital. This is because it would be undesirable to minimize the generalization error over the noise that has been generated with the training data.

Reward-based Data Generation In this work, we showed that it is possible to control language generation models through reward-based schemes. In chapter 3, we forced GPT-2 to generate samples that cause the highest classification confusion by rewarding sequences based on the classification entropy. In chapters 4 and 5, we applied the same schema, but with different reward functions applicable to the respective task. The underlying aim throughout all three chapters was to generate data samples that would augment an existing dataset. We have successfully shown that different reward functions can be used depending on the task at hand. For example, to create diversified samples that would most confuse a classifier, in addition to entropy we rewarded text sequences on their dissimilarity with previously generated data, see section 3.3.2. To create samples that are most confusing to a student classifier, but least confusing to its teacher, we rewarded generated samples based on the difference of entropy between both models as explained in section 5.3.3. Having tested reward-based generation with MCTS for multiple tasks requiring different conditions, we believe that for text generation this schema has great room for improvement. For instance, if the task requires the generated samples to be of high text quality, the reward function can then reward samples based on characteristics like grammar quality. Such a feature could be implemented by training a separate classifier for grammar acceptability on the Corpus of Linguistic Acceptability (CoLA) dataset from the GLUE benchmark, explained in section 2.5.1. CoLA is a binary classification dataset, with sentences labeled as grammatically correct or incorrect. A classifier trained on this dataset can then be used to reward generated sequences based on grammar acceptability. Outside the scope of data augmentation, there can also be multiple useful applications for reward-based data generation. For instance, if a task requires the generated samples to be within proximity to an existing subset of samples, then an appropriate distance measure can be incorporated in the reward function. Generating data under such requirements could be useful for different applications, such as information retrieval systems.

Ethical Language Generation In section 3.5.2, we discussed the ethical issues of data generation and their implications on the targeted audience. In the scope of text generation, a framework can be devised that ensures compliance with ethical considerations. Training large language models requires extensive resources that are expensive to set up, and which are normally affordable only by large corporations. To put this into perspective, it is estimated that a single run of training GPT-3 costed OpenAI 4.6 Million USD (Dale, 2021). Driven by their own motives, such as making profits, developers of large language models might not have ethical management as a top priority (Hagendorff, 2020). For this reason, it might be difficult to rely on developers to self-regulate their practices. Hence, ethical compliance might be most effective when enforced by governmental regulations (Chan, 2022). Devising a framework for ethical text generation to help governmental agencies enforce these regulations might be a step forward in the right direction. This framework could include a list of requirements which the data should meet to be considered acceptable for training. For instance, a list of approved data resources could be made public. For datasets curated outside this list, appropriate analysis of the data should be provided to show its adherence to ethical standards.

References

Moloud Abdar, Farhad Pourpanah, Sadiq Hussain, Dana Rezazadegan, Li Liu, Mohammad Ghavamzadeh, Paul Fieguth, Xiaochun Cao, Abbas Khosravi, U Rajendra Acharya, et al. 2021. A review of uncertainty quantification in deep learning: Techniques, applications and challenges. *Information Fusion*.

Abubakar Abid, Maheen Farooqi, and James Zou. 2021. Persistent anti-muslim bias in large language models. In *Proceedings of the 2021 AAAI/ACM Conference on AI, Ethics, and Society*, pages 298–306.

CC Aggrawal. 2018. Neural networks and deep learning: A textbook.

Milam Aiken and Mina Park. 2010. The efficacy of round-trip translation for mt evaluation. *Translation Journal*, 14(1):1–10.

Alan Akbik, Tanja Bergmann, Duncan Blythe, Kashif Rasul, Stefan Schweter, and Roland Vollgraf. 2019. Flair: An easy-to-use framework for state-of-the-art nlp. In *NAACL 2019, 2019 Annual Conference of the North American Chapter of the Association for Computational Linguistics (Demonstrations)*, pages 54–59.

Ateret Anaby-Tavor, Boaz Carmeli, Esther Goldbraich, Amir Kantor, George Kour, Segev Shlomov, Naama Tepper, and Naama Zwerdling. 2019. Not enough data? deep learning to the rescue! *arXiv preprint arXiv:1911.03118*.

Broderick Arneson, Ryan B Hayward, and Philip Henderson. 2010. Monte carlo tree search in hex. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):251–258.

Sanjeev Arora, Yingyu Liang, and Tengyu Ma. 2017. A simple but tough-to-beat baseline for sentence embeddings. In *International conference on learning representations*.

Segun Taofeek Aroyehun and Alexander Gelbukh. 2018. Aggression detection in social media: Using deep neural networks, data augmentation, and pseudo labeling. In *Proceedings of the First Workshop on Trolling, Aggression and Cyberbullying (TRAC-2018)*, pages 90–97, Santa Fe, New Mexico, USA. Association for Computational Linguistics.

Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. 2002. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450*.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

Mikhail Belkin, Daniel Hsu, Siyuan Ma, and Soumik Mandal. 2019. Reconciling modern machine-learning practice and the classical bias–variance trade-off. *Proceedings of the National Academy of Sciences*, 116(32):15849–15854.

Mikhail Belkin, Daniel Hsu, and Ji Xu. 2020. Two models of double descent for weak features. *SIAM Journal on Mathematics of Data Science*, 2(4):1167–1180.

Emily M Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. 2021. On the dangers of stochastic parrots: Can language models be too big? In *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, pages 610–623.

Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. 2003. A neural probabilistic language model. *The journal of machine learning research*, 3:1137–1155.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. 2009. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48.

Jordan J Bird, Anikó Ekárt, and Diego R Faria. 2020. On the effects of pseudorandom and quantum-random number generators in soft computing. *Soft computing*, 24(12):9243–9256.

Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. 2015. Weight uncertainty in neural network. In *International Conference on Machine Learning*, pages 1613–1622. PMLR.

Samuel R Bowman, Gabor Angeli, Christopher Potts, and Christopher D Manning. 2015. A large annotated corpus for learning natural language inference. *arXiv preprint arXiv:1508.05326*.

Leo Breiman. 1996. Bagging predictors. *Machine learning*, 24(2):123–140.

Gavin Brown. 2010. Ensemble learning. *Encyclopedia of machine learning*, 312:15–19.

Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*.

C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43.

Joy Buolamwini and Timnit Gebru. 2018. Gender shades: Intersectional accuracy disparities in commercial gender classification. In *Conference on fairness, accountability and transparency*, pages 77–91. PMLR.

Daniel Cer, Yinfei Yang, Sheng-yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St John, Noah Constant, Mario Guajardo-Céspedes, Steve Yuan, Chris Tar, et al. 2018. Universal sentence encoder. *arXiv preprint arXiv:1803.11175*.

Anastasia Chan. 2022. Gpt-3 and instructgpt: technological dystopianism, utopianism, and "contextual" perspectives in ai ethics and industry. *AI and Ethics*, pages 1–12.

Hyeong Soo Chang, Michael C Fu, Jiaqiao Hu, and Steven I Marcus. 2016. Google deep mind's alphago. *OR/MS Today*, 43(5):24–29.

Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. 2008. Monte-carlo tree search: A new framework for game ai. In *AIIDE*.

Muhammad Umar Chaudhry and Jee-Hyong Lee. 2018. Feature selection for high dimensional data using monte carlo tree search. *IEEE Access*, 6:76036–76048.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Hyunjin Choi, Judong Kim, Seongho Joe, and Youngjune Gwon. 2021. Evaluation of bert and albert sentence embedding performance on downstream nlp tasks. In *2020 25th International Conference on Pattern Recognition (ICPR)*, pages 5482–5487. IEEE.

Michael Chung, Michael Buro, and Jonathan Schaeffer. 2005. Monte carlo planning in rts games. In *CIG*. Citeseer.

Christopher Clark, Mark Yatskar, and Luke Zettlemoyer. 2019. Don't take the easy way out: Ensemble based methods for avoiding known dataset biases. *CoRR*, abs/1909.03683.

Alexis Conneau, Douwe Kiela, Holger Schwenk, Loic Barrault, and Antoine Bordes. 2017. Supervised learning of universal sentence representations from natural language inference data. *arXiv preprint arXiv:1705.02364*.

Timothy F. Cootes, Gareth J. Edwards, and Christopher J. Taylor. 2001. Active appearance models. *IEEE Transactions on pattern analysis and machine intelligence*, 23(6):681–685.

Ido Dagan, Oren Glickman, and Bernardo Magnini. 2005. The pascal recognising textual entailment challenge. In *Machine Learning Challenges Workshop*, pages 177–190. Springer.

Robert Dale. 2021. Gpt-3: What's it good for? *Natural Language Engineering*, 27(1):113–118.

Huong Dang, Kahyun Lee, Sam Henry, and Ozlem Uzuner. 2020. Ensemble bert for classifying medication-mentioning tweets. In *Proceedings of the Fifth Social Media Mining for Health Applications Workshop & Shared Task*, pages 37–41.

Jeffrey Dastin. 2018. Amazon scraps secret ai recruiting tool that showed bias against women. In *Ethics of Data and Analytics*, pages 296–299. Auerbach Publications.

Scott Deerwester, Susan T Dumais, George W Furnas, Thomas K Landauer, and Richard Harshman. 1990. Indexing by latent semantic analysis. *Journal of the American society for information science*, 41(6):391–407.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

William B Dolan and Chris Brockett. 2005. Automatically constructing a corpus of sentential paraphrases. In *Proceedings of the Third International Workshop on Paraphrasing (IWP2005)*.

Stefan Edelkamp, Max Gath, Christoph Greulich, Malte Humann, Otthein Herzog, and Michael Lawo. 2016. Monte-carlo tree search for logistics. In *Commercial Transport*, pages 427–440. Springer.

Bradley Efron and Robert J Tibshirani. 1994. *An introduction to the bootstrap*. CRC press.

Kawin Ethayarajh. 2019. How contextual are contextualized word representations? comparing the geometry of bert, elmo, and GPT-2 embeddings. *CoRR*, abs/1909.00512.

Angela Fan, Mike Lewis, and Yann Dauphin. 2018. Hierarchical neural story generation. *arXiv preprint arXiv:1805.04833*.

Steven Y Feng, Varun Gangal, Dongyeop Kang, Teruko Mitamura, and Eduard Hovy. 2020. Genaug: Data augmentation for finetuning text generators. *arXiv preprint arXiv:2010.01794*.

Steven Y Feng, Varun Gangal, Jason Wei, Sarath Chandar, Soroush Vosoughi, Teruko Mitamura, and Eduard Hovy. 2021. A survey of data augmentation approaches for nlp. *arXiv preprint arXiv:2105.03075*.

David Freedman, Robert Pisani, and Roger Purves. 2007. Statistics (international student edition). *Pisani, R. Purves, 4th edn. WW Norton & Company, New York*.

Alexander Freytag, Erik Rodner, and Joachim Denzler. 2014. Selecting influential examples: Active learning with expected model output changes. In *European Conference on Computer Vision*, pages 562–577. Springer.

KS Fu. 2004. Ieee transactions on pattern analysis and machine intelligence. *Encyclopedia of Statistical Sciences*.

Philip Gage. 1994. A new algorithm for data compression. *C Users Journal*, 12(2):23–38.

Yarin Gal and Zoubin Ghahramani. 2016. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*, pages 1050–1059. PMLR.

R Stuart Geiger, Dominique Cope, Jamie Ip, Marsha Lotosh, Aayush Shah, Jenny Weng, and Rebekah Tang. 2021. "garbage in, garbage out" revisited: What do machine learning application papers report about human-labeled training data? *Quantitative Science Studies*, 2(3):795–827.

R Stuart Geiger, Kevin Yu, Yanlai Yang, Mindy Dai, Jie Qiu, Rebekah Tang, and Jenny Huang. 2020. Garbage in, garbage out? do machine learning application papers in social computing report where human-labeled training data comes from? In *Proceedings of the 2020 Conference on Fairness, Accountability, and Transparency*, pages 325–336.

Jianping Gou, Baosheng Yu, Stephen J Maybank, and Dacheng Tao. 2021. Knowledge distillation: A survey. *International Journal of Computer Vision*, pages 1–31.

Pablo M Granitto, Pablo F Verdes, and H Alejandro Ceccatto. 2005. Neural network ensembles: evaluation of aggregation algorithms. *Artificial Intelligence*, 163(2):139–162.

Alex Graves. 2011. Practical variational inference for neural networks. *Advances in neural information processing systems*, 24.

Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q Weinberger. 2017. On calibration of modern neural networks. In *International Conference on Machine Learning*, pages 1321–1330. PMLR.

Zhijiang Guo, Michael Schlichtkrull, and Andreas Vlachos. 2022. A survey on automated fact-checking. *Transactions of the Association for Computational Linguistics*, 10:178–206.

Suchin Gururangan, Swabha Swayamdipta, Omer Levy, Roy Schwartz, Samuel R Bowman, and Noah A Smith. 2018. Annotation artifacts in natural language inference data. *arXiv preprint arXiv:1803.02324*.

Thilo Hagendorff. 2020. The ethics of ai ethics: An evaluation of guidelines. *Minds and Machines*, 30(1):99–120.

Godfrey Harold Hardy, John Edensor Littlewood, George Pólya, György Pólya, et al. 1952. *Inequalities*. Cambridge university press.

Trevor Hastie, Andrea Montanari, Saharon Rosset, and Ryan J Tibshirani. 2019. Surprises in high-dimensional ridgeless least squares interpolation. *arXiv preprint arXiv:1903.08560*.

Trevor Hastie, Robert Tibshirani, and Jerome Friedman. 2009. *The elements of statistical learning: data mining, inference, and prediction*. Springer Science & Business Media.

Matthias Hein, Maksym Andriushchenko, and Julian Bitterwolf. 2019. Why relu networks yield high-confidence predictions far away from the training data and how to mitigate the problem. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 41–50.

Byeongho Heo, Minsik Lee, Sangdoo Yun, and Jin Young Choi. 2019. Knowledge transfer via distillation of activation boundaries formed by hidden neurons. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 3779–3787.

Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. 2012. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. *Cited on*, 14(8):2.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*.

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

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2019. The curious case of neural text degeneration. *arXiv preprint arXiv:1904.09751*.

Stephen C Hora. 1996. Aleatory and epistemic uncertainty in probability elicitation with an example from hazardous waste management. *Reliability Engineering & System Safety*, 54(2-3):217–223.

Jeremy Howard and Sebastian Ruder. 2018a. Fine-tuned language models for text classification. *CoRR*, abs/1801.06146.

Jeremy Howard and Sebastian Ruder. 2018b. Universal language model fine-tuning for text classification. *arXiv preprint arXiv:1801.06146*.

Rong Hu, Brian Mac Namee, and Sarah Jane Delany. 2016. Active learning for text classification with reusability. *Expert systems with applications*, 45:438–449.

Eyke Hüllermeier and Willem Waegeman. 2021. Aleatoric and epistemic uncertainty in machine learning: An introduction to concepts and methods. *Machine Learning*, 110(3):457–506.

Rebecca Hwa. 2004. Sample selection for statistical parsing. *Computational linguistics*, 30(3):253–276.

Daphne Ippolito, Reno Kriz, Maria Kustikova, João Sedoc, and Chris Callison-Burch. 2019. Comparison of diverse decoding methods from conditional language models. *arXiv preprint arXiv:1906.06362*.

Shankar Iyer, Nikhil Dandekar, and Kornel Csernai. 2017. First quora dataset release: Question pairs.

Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. 2019. Tinybert: Distilling bert for natural language understanding. *arXiv preprint arXiv:1909.10351*.

Steven E Jones. 2013. *Against technology: From the Luddites to neo-Luddism*. Routledge.

Ajay J Joshi, Fatih Porikli, and Nikolaos Papanikolopoulos. 2009. Multi-class active learning for image classification. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 2372–2379. IEEE.

Laurent Valentin Jospin, Wray Buntine, Farid Boussaid, Hamid Laga, and Mohammed Bennamoun. 2020. Hands-on bayesian neural networks–a tutorial for deep learning users. *arXiv preprint arXiv:2007.06823*.

Benjamin Jun and Paul Kocher. 1999. The intel random number generator. *Cryptography Research Inc. white paper*, 27:1–8.

Michał Jungiewicz and Aleksander Smywiński-Pohl. 2019. Towards textual data augmentation for neural networks: synonyms and maximum loss. *Computer Science*, 20.

Daniel Khashabi, Snigdha Chaturvedi, Michael Roth, Shyam Upadhyay, and Dan Roth. 2018. Looking beyond the surface:a challenge set for reading comprehension over multiple sentences. In *Proceedings of North American Chapter of the Association for Computational Linguistics (NAACL)*.

Douwe Kiela, Max Bartolo, Yixin Nie, Divyansh Kaushik, Atticus Geiger, Zhengxuan Wu, Bertie Vidgen, Grusha Prasad, Amanpreet Singh, Pratik Ringshia, et al. 2021. Dynabench: Rethinking benchmarking in nlp. *arXiv preprint arXiv:2104.14337*.

Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Ryan Kiros, Yukun Zhu, Russ R Salakhutdinov, Richard Zemel, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. 2015. Skip-thought vectors. *Advances in neural information processing systems*, 28.

Sosuke Kobayashi. 2018. Contextual augmentation: Data augmentation by words with paradigmatic relations. *arXiv preprint arXiv:1805.06201*.

Levente Kocsis and Csaba Szepesvári. 2006. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer.

Anastasia Krithara, Cyril Goutte, Jean-Michel Renders, and MR Amini. 2006. Reducing the annotation burden in text classification. In *Proceedings of the 1st International Conference on Multidisciplinary Information Sciences and Technologies (InSciT 2006), Merida, Spain*.

Varun Kumar, Ashutosh Choudhary, and Eunah Cho. 2020. Data augmentation using pre-trained transformer models. *arXiv preprint arXiv:2003.02245*.

Balaji Lakshminarayanan, Alexander Pritzel, and Charles Blundell. 2017. Simple and scalable predictive uncertainty estimation using deep ensembles. *Advances in neural information processing systems*, 30.

Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. 2020. Albert: A lite bert for self-supervised learning of language representations.

Hector Levesque, Ernest Davis, and Leora Morgenstern. 2012. The winograd schema challenge. In *Thirteenth International Conference on the Principles of Knowledge Representation and Reasoning*.

David D Lewis and Jason Catlett. 1994. Heterogeneous uncertainty sampling for supervised learning. In *Machine learning proceedings 1994*, pages 148–156. Elsevier.

David D Lewis and William A Gale. 1994. A sequential algorithm for training text classifiers. In *SIGIR'94*, pages 3–12. Springer.

Xin Li and Dan Roth. 2002. Learning question classifiers. In *Proceedings of the 19th international conference on Computational linguistics-Volume 1*, pages 1–7. Association for Computational Linguistics.

Yu Li, Xiao Li, Yating Yang, and Rui Dong. 2020. A diverse data augmentation strategy for low-resource neural machine translation. *Information*, 11(5):255.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*.

Stuart Lloyd. 1982. Least squares quantization in pcm. *IEEE transactions on information theory*, 28(2):129–137.

Lajanugen Logeswaran and Honglak Lee. 2018. An efficient framework for learning sentence representations. *arXiv preprint arXiv:1803.02893*.

Shayne Longpre, Yu Wang, and Christopher DuBois. 2020. How effective is task-agnostic data augmentation for pretrained transformers? *arXiv preprint arXiv:2010.01764*.

Andrew Maas, Raymond E Daly, Peter T Pham, Dan Huang, Andrew Y Ng, and Christopher Potts. 2011. Learning word vectors for sentiment analysis. In *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies*, pages 142–150.

David JC MacKay. 1992. A practical bayesian framework for backpropagation networks. *Neural computation*, 4(3):448–472.

Nishtha Madaan, Sameep Mehta, Taneea Agrawaal, Vrinda Malhotra, Aditi Aggarwal, Yatin Gupta, and Mayank Saxena. 2018. Analyze, detect and remove gender stereotyping from bollywood movies. In *Conference on fairness, accountability and transparency*, pages 92–105. PMLR.

Mieradilijiang Maimaiti, Yang Liu, Huanbo Luan, and Maosong Sun. 2022. Data augmentation for low-resource languages nmt guided by constrained sampling. *International Journal of Intelligent Systems*, 37(1):30–51.

R Thomas McCoy, Junghyun Min, and Tal Linzen. 2019. Berts of a feather do not generalize together: Large variability in generalization across models with similar test set performance. *arXiv preprint arXiv:1911.02969*.

Kris McGuffie and Alex Newhouse. 2020. The radicalization risks of gpt-3 and advanced neural language models. *arXiv preprint arXiv:2009.06807*.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

Tomáš Mikolov, Stefan Kombrink, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. 2011. Extensions of recurrent neural network language model. In *2011 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pages 5528–5531. IEEE.

Shervin Minaee, Nal Kalchbrenner, Erik Cambria, Narjes Nikzad, Meysam Chenaghlu, and Jianfeng Gao. 2021. Deep learning–based text classification: a comprehensive review. *ACM Computing Surveys (CSUR)*, 54(3):1–40.

Marius Mosbach, Maksym Andriushchenko, and Dietrich Klakow. 2020. On the stability of fine-tuning bert: Misconceptions, explanations, and strong baselines. *arXiv preprint arXiv:2006.04884*.

Subhabrata Mukherjee and Ahmed Hassan Awadallah. 2020. Uncertainty-aware self-training for text classification with few labels. *arXiv preprint arXiv:2006.15315*.

Vidya Muthukumar, Kailas Vodrahalli, Vignesh Subramanian, and Anant Sahai. 2020. Harmless interpolation of noisy data in regression. *IEEE Journal on Selected Areas in Information Theory*, 1(1):67–83.

Preetum Nakkiran, Gal Kaplun, Yamini Bansal, Tristan Yang, Boaz Barak, and Ilya Sutskever. 2019. Deep double descent: Where bigger models and more data hurt. *CoRR*, abs/1912.02292.

Behnam Neyshabur, Ryota Tomioka, and Nathan Srebro. 2014. In search of the real inductive bias: On the role of implicit regularization in deep learning. *arXiv preprint arXiv:1412.6614*.

Andrew Ng. 2017. Improving deep neural networks: Hyperparameter tuning, regularization and optimization. *Deeplearning. ai on Coursera*.

Phi-Vu Nguyen, Ali Ghezal, Ya-Chih Hsueh, Thomas Boudier, Samuel Ken-En Gan, and Hwee Kuan Lee. 2016. Optimal processing for gel electrophoresis images: applying monte carlo tree search in gelapp. *Electrophoresis*, 37(15-16):2208–2216.

Kamal Nigam and Andrew McCallum. 1998. Pool-based active learning for text classification. In *Conference on Automated Learning and Discovery (CONALD)*.

Christopher Olah. 2015. Understanding lstm networks.

Barak Or. 2020. The exploding and vanishing gradients problem in time series.

Genevieve Orr. Momentum and learning rate adaptation.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. 2013. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pages 1310–1318. PMLR.

Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543.

Diego Perez, Julian Togelius, Spyridon Samothrakis, Philipp Rohlfshagen, and Simon M Lucas. 2013. Automated map generation for the physical traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 18(5):708–720.

Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. *arXiv preprint arXiv:1802.05365*.

Ning Qian. 1999. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151.

Siyuan Qiu, Binxia Xu, Jie Zhang, Yafang Wang, Xiaoyu Shen, Gerard De Melo, Chong Long, and Xiaolong Li. 2020. Easyaug: An automatic textual data augmentation platform for classification tasks. In *Companion Proceedings of the Web Conference 2020*, pages 249–252.

Husam Quteineh, Spyridon Samothrakis, and Richard Sutcliffe. 2020. Textual data augmentation for efficient active learning on tiny datasets. Association for Computational Linguistics.

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training. *URL https://s3-us-west-2. amazonaws. com/openai-assets/researchcovers/languageunsupervised/language understanding paper. pdf*.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners.

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250*.

Alexander Ratner, Stephen H. Bach, Henry Ehrenberg, Jason Fries, Sen Wu, and Christopher Ré. 2017. Snorkel: Rapid training data creation with weak supervision. *Proc. VLDB Endow.*, 11(3):269–282.

Russell Reed and Robert J MarksII. 1999. *Neural smithing: supervised learning in feedforward artificial neural networks*. Mit Press.

Mehdi Regina, Maxime Meyer, and Sébastien Goutal. 2020. Text data augmentation: Towards better detection of spear-phishing emails. *arXiv preprint arXiv:2007.02033*.

Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. 2014. Fitnets: Hints for thin deep nets. *arXiv preprint arXiv:1412.6550*.

Nicholas Roy and Andrew McCallum. 2001. Toward optimal active learning through monte carlo estimation of error reduction. *ICML, Williamstown*, pages 441–448.

Andreas Rücklé, Steffen Eger, Maxime Peyrard, and Iryna Gurevych. 2018. Concatenated power mean word embeddings as universal cross-lingual sentence representations. *arXiv preprint arXiv:1803.01400*.

Hassan Sajjad, Fahim Dalvi, Nadir Durrani, and Preslav Nakov. 2020. Poor man's bert: Smaller and faster transformer models. *arXiv preprint arXiv:2004.03844*.

Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2019. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*.

S. K. Sankarpandi, S. Samothrakis, L. Citi, and P. Brady. 2019. Active learning without unlabeled samples: generating questions and labels using monte carlo tree search. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 4628–4631.

Robert E Schapire. 1990. The strength of weak learnability. *Machine learning*, 5(2):197–227.

Mike Schuster and Kaisuke Nakajima. 2012. Japanese and korean voice search. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5149–5152. IEEE.

Henry Scudder. 1965. Probability of error of some adaptive pattern-recognition machines. *IEEE Transactions on Information Theory*, 11(3):363–371.

Ozan Sener and Silvio Savarese. 2017. Active learning for convolutional neural networks: A core-set approach. *arXiv preprint arXiv:1708.00489*.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2015a. Improving neural machine translation models with monolingual data. *arXiv preprint arXiv:1511.06709*.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2015b. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*.

Burr Settles. 2009. Active learning literature survey. Technical report, University of Wisconsin-Madison Department of Computer Sciences.

Burr Settles and Mark Craven. 2008. An analysis of active learning strategies for sequence labeling tasks. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*, pages 1070–1079.

Burr Settles, Mark Craven, and Soumya Ray. 2008. Multiple-instance active learning. In J. C. Platt, D. Koller, Y. Singer, and S. T. Roweis, editors, *Advances in Neural Information Processing Systems 20*, pages 1289–1296. Curran Associates, Inc.

Jack G Shaheen. 2003. Reel bad arabs: How hollywood vilifies a people. *The ANNALS of the American Academy of Political and Social science*, 588(1):171–193.

Connor Shorten and Taghi M Khoshgoftaar. 2019. A survey on image data augmentation for deep learning. *Journal of Big Data*, 6(1):1–48.

Yawar Siddiqui, Julien Valentin, and Matthias Nießner. 2019. Viewal: Active learning with viewpoint entropy for semantic segmentation. *arXiv preprint arXiv:1911.11789*.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489.

Samarth Sinha, Sayna Ebrahimi, and Trevor Darrell. 2019. Variational adversarial active learning. *CoRR*, abs/1904.00370.

Leslie N Smith. 2017. Cyclical learning rates for training neural networks. In *2017 IEEE winter conference on applications of computer vision (WACV)*, pages 464–472. IEEE.

Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. 2013. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958.

Chi Sun, Xipeng Qiu, Yige Xu, and Xuanjing Huang. 2019a. How to fine-tune bert for text classification? In *China national conference on Chinese computational linguistics*, pages 194–206. Springer.

Siqi Sun, Yu Cheng, Zhe Gan, and Jingjing Liu. 2019b. Patient knowledge distillation for bert model compression. *arXiv preprint arXiv:1908.09355*.

Tony Sun, Andrew Gaut, Shirlyn Tang, Yuxin Huang, Mai ElSherief, Jieyu Zhao, Diba Mirza, Elizabeth Belding, Kai-Wei Chang, and William Yang Wang. 2019c. Mitigating gender bias in natural language processing: Literature review. *arXiv preprint arXiv:1906.08976*.

Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. 2020. Mobilebert: a compact task-agnostic bert for resource-limited devices. *arXiv preprint arXiv:2004.02984*.

James Surowiecki. 2005. The wisdom of crowds. new york: Anchor.

Raphael Tang, Yao Lu, Linqing Liu, Lili Mou, Olga Vechtomova, and Jimmy Lin. 2019. Distilling task-specific knowledge from bert into simple neural networks. *arXiv preprint arXiv:1903.12136*.

Antti Tarvainen and Harri Valpola. 2017. Weight-averaged consistency targets improve semi-supervised deep learning results. *CoRR*, abs/1703.01780.

Simon Tong and Daphne Koller. 2001. Support vector machine active learning with applications to text classification. *Journal of machine learning research*, 2(Nov):45–66.

Jack L Treynor. 1987. Market efficiency and the bean jar experiment. *Financial Analysts Journal*, 43(3):50–53.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *arXiv preprint arXiv:1706.03762*.

Alex Wang and Kyunghyun Cho. 2019. Bert has a mouth, and it must speak: Bert as a markov random field language model. *arXiv preprint arXiv:1902.04094*.

Alex Wang, Yada Pruksachatkun, Nikita Nangia, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. 2019. Superglue: A stickier benchmark for general-purpose language understanding systems. *arXiv preprint arXiv:1905.00537*.

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R Bowman. 2018a. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*.

Gaoang Wang, Jenq-Neng Hwang, Craig Rose, and Farron Wallace. 2017. Uncertainty sampling based active learning with diversity constraint by sparse selection. In *2017 IEEE 19th International Workshop on Multimedia Signal Processing (MMSP)*, pages 1–6. IEEE.

William Yang Wang and Diyi Yang. 2015. That's so annoying!!!: A lexical and frame-semantic embedding based data augmentation approach to automatic categorization of annoying behaviors using #petpeeve tweets. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 2557–2563, Lisbon, Portugal. Association for Computational Linguistics.

Xinyi Wang, Hieu Pham, Zihang Dai, and Graham Neubig. 2018b. SwitchOut: an an efficient data augmentation algorithm for neuralr neural machine translation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 856–861, Brussels, Belgium. Association for Computational Linguistics.

Jason Wei and Kai Zou. 2019. Eda: Easy data augmentation techniques for boosting performance on text classification tasks. *arXiv preprint arXiv:1901.11196*.

Adina Williams, Nikita Nangia, and Samuel R Bowman. 2017. A broad-coverage challenge corpus for sentence understanding through inference. *arXiv preprint arXiv:1704.05426*.

Thomas Wolf, Julien Chaumond, Lysandre Debut, Victor Sanh, Clement Delangue, Anthony Moi, Pierric Cistac, Morgan Funtowicz, Joe Davison, Sam Shleifer, et al. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45.

Xing Wu, Shangwen Lv, Liangjun Zang, Jizhong Han, and Songlin Hu. 2019. Conditional bert contextual augmentation. In *International Conference on Computational Science*, pages 84–95. Springer.

Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V Le. 2019. Xlnet: Generalized autoregressive pretraining for language understanding. *arXiv preprint arXiv:1906.08237*.

Junho Yim, Donggyu Joo, Jihoon Bae, and Junmo Kim. 2017. A gift from knowledge distillation: Fast optimization, network minimization and transfer learning. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 7130–7138.

Donggeun Yoo and In So Kweon. 2019. Learning loss for active learning. *CoRR*, abs/1905.03677.

Kang Min Yoo, Dongju Park, Jaewook Kang, Sang-Woo Lee, and Woomyeong Park. 2021. Gpt3mix: Leveraging large-scale language models for text augmentation. *arXiv preprint arXiv:2104.08826*.

Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. 2014. How transferable are features in deep neural networks? *arXiv preprint arXiv:1411.1792*.

Xia Zeng, Amani S Abumansour, and Arkaitz Zubiaga. 2021. Automated fact-checking: A survey. *Language and Linguistics Compass*, 15(10):e12438.

Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. 2016. Understanding deep learning requires rethinking generalization. *CoRR*, abs/1611.03530.

Tianyi Zhang, Felix Wu, Arzoo Katiyar, Kilian Q Weinberger, and Yoav Artzi. 2020. Revisiting few-sample bert fine-tuning. *arXiv preprint arXiv:2006.05987*.

Xiang Zhang, Junbo Zhao, and Yann LeCun. 2015. Character-level convolutional networks for text classification. *arXiv preprint arXiv:1509.01626*.

Jieyu Zhao, Tianlu Wang, Mark Yatskar, Vicente Ordonez, and Kai-Wei Chang. 2018. Gender bias in coreference resolution: Evaluation and debiasing methods. *arXiv preprint arXiv:1804.06876*.

Zhi-Hua Zhou. 2019. *Ensemble methods: foundations and algorithms*. Chapman and Hall/CRC.

Xiaojin Zhu and Andrew B Goldberg. 2009. Introduction to semi-supervised learning. *Synthesis lectures on artificial intelligence and machine learning*, 3(1):1–130.

Yukun Zhu, Ryan Kiros, Rich Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. 2015. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *Proceedings of the IEEE international conference on computer vision*, pages 19–27.