UNIVERSITY OF CALIFORNIA

Los Angeles

# Fine-Tuning BERT for Sentiment Analysis

A thesis submitted in partial satisfaction
of the requirements for the degree
Master of Science in Statistics

by

Michelle Lu Wang

2024

ABSTRACT OF THE THESIS

Fine-Tuning BERT for Sentiment Analysis

by

Michelle Lu Wang
Master of Science in Statistics
University of California, Los Angeles, 2024
Professor Yingnian Wu, Chair

The introduction of transformer models have vastly improved the performance of machine learning methods on natural language processing (NLP) tasks. Transformer models use a self-attention mechanism which allow the model to weigh the importance of different words in a sequence when making predictions. They also introduce positional encodings which allow the model to be highly parallelizable, expanding the capacity of data they are able to learn from.

This paper explores the fine-tuning of the pre-trained transformer model BERT (Bidirectional Encoder Representations from Transformers) for sentiment analysis on e-commerce reviews. Traditional fine-tuning approaches which involve updating every parameter of a pre-trained model's hundreds of millions of parameters can be inefficient and unnecessary. A parameter-efficient fine-tuning approach is proposed to enhance the pre-trained BERT's performance in discerning between positive and negative sentiments within the diverse user-generate reviews.

The methodology begins by preprocessing the data, including text cleaning and tokenization, to prepare the dataset for training. Subsequently, fine-tuning and hyperparameter tuning techniques are applied to the model in order to tailor BERT to the specific qualities of

the dataset. Smaller subsets of data are fine-tuned on in order to find optimal hyperparameter settings for fine-tuning the full dataset. Three BERT based models will be explored: $\text{BERT}_{BASE}$, $\text{RoBERTa}_{BASE}$, and DistilBERT. Each model will be fine-tuned and evaluated in order to find the model which achieves the highest test accuracy rate. The paper will also delve into the obstacles of training with a large dataset, proposing solutions and techniques to circumvent the problems.

The findings of this paper show the variations of the models which perform the best greatly depend on the needs of the dataset. The larger dataset analyzed in this paper requires a faster, lighter model in order to process in its entirety. The experiment also explores more robustly optimized and larger models, which yield adequate results using a smaller data subset, but are not suitable for bigger data sets. Hyperparameter settings are shown to affect the performance of the model, but not impact the model in any distinct patterns. The exception to this is the number of epochs the data is trained for, which almost always positively influences model accuracy rates.

The thesis of Michelle Lu Wang is approved.

Hongquan Xu

Frederic R Paik Schoenberg

Yingnian Wu, Committee Chair

University of California, Los Angeles

2024

*To my family and friends, mentors and peers - I do not see this is an accomplishment completed solely on my own, but rather a display of the efforts I can achieve with the support of the wonderful people I've been lucky enough to surround myself with.*

# Table of Contents

# List of Figures

# List of Tables

# CHAPTER 1

# Introduction

## 1.1 Natural Language Processing

Natural language processing (NLP) tasks lie at the forefront of bridging artificial intelligence with human language. NLP combines linguistics and machine learning in an attempt to create a machine that comprehends language and responds as a real human would. It has two main components: natural language understanding and natural language generation, which together have the potential to seamlessly construct communication between computers and humans. NLP algorithms take vast amounts of data and train a machine based on the existing text in order to learn to generate, understand, and classify new passages of text. These algorithms have evolved in the past decades to create models that have grown exponentially in their ability to accurately mimic the complex nuances of human linguistics. This paper will discuss the fine-tuning and use of a pre-trained transformer model to perform sentiment analysis. Sentiment analysis is a sector of a larger collection of NLP tasks which aim to classify the type of emotion conveyed by a piece of text [Gun23].

The idea of language as a science has spanned for multiple decades, catalyzed by the introduction of Alan Turing's prominent "Turing Test" in 1950. Turing was an accomplished mathematician who strived to uncover the limits of computation. He surmised that if a computer could indistinguishably mimic human language from an actual human through means of chat, that the computer could be considered intelligent. Though it has been shown that the Turing Test exhibits problems which challenge its actual legitimacy [TUR50], there is no doubt that this theory lied at the head of one of the most prevalent topics at the front

lines of machine learning today.

In current research in the field of NLP, language processing is studied through the lens of creating machines which can mimic human language and intelligence, rather than aiming to possess a quality of conscious intelligence. There are a plethora of natural language processing tasks, all of these centered on the foundational goal of developing machines that are able to imitate human language. A few common ones are:

- Text Classification: Text classification assigns a category to a text. Many common NLP tasks fall under this category including sentiment analysis, spam detection for emails, and common categorization tasks prevalent in everyday life (news categorizations, book and movie genres, and many more).

- Machine Translation: Machine translation converts text from one language to another. Early forms of machine translation were rule-based, but with the rise of neural machine translation, adaptive translation that fully encompasses the complicated meaning and context behind multilingual phrases is possible.

- Text Summarization: Text summarization takes large text documents and outputs summaries, retaining the most important information and underlying meanings of the original text.

- Question Answering (QA): Question answering takes an input of a question and outputs an answer. There are three main types of question answering tasks. Extractive QA answers questions based on a given text, where the model is able to extract the answer from the text. Open generative QA is also based on a given text, but is allowed to openly generate an answer based on the context of the text. Closed generative QA generates an answer based only on the model, without any other given context. Chatbots, which have recently risen in popularity, fall under this category.

- Semantic Textual Similarity (STS): STS compares different texts to determine if they are similar in meaning. Usually STS takes inputs of two different texts, and outputs a

numerical score ranking their similarity. STS is a particularly difficult NLP task since it prioritizes the semantic meaning of two texts, which could linguistics wise have very different words.

Of these many tasks, text classification makes up an important sector of NLP. Within text classification, sentiment analysis aims to design a model which can classify the underlying polarity or emotion of a given text. Based on the specific goals of a study, the scale of sentiment analysis can vary from simple positive or negative sentiments, to more complicated graded sentiments (very positive, positive, neutral, negative, very negative).

Sentiment analysis is widely used in many different sectors, and is particularly useful for businesses. The shift toward e-commerce and online platforms in the last decade has restructured many companies and business models. The rise of social media and online platforms have created a ubiquitously accessible environment for customers to express their opinions on products. In the modern world, for any given business or product, it is possible that there exists extensive amounts of publicly available information that may both help or hinder a business. By monitoring social media, reviews, and other online platforms, businesses have the unique opportunity to receive real time reactions from consumers as well as research on competitors. The problem has quickly developed from finding a lack of genuine market data to experiencing a lack of procedure to process an influx of large amounts of data.

NLP algorithms are particularly useful in theses cases. Language models provide an opportunity for companies to quickly and almost automatically turn complex text into clear data points. Language models are trained to allow for an automated processing of large amounts of text data at once. Tasks that would take hours or days to complete manually, can be performed by machines trained on language models with similar accuracy and performance within minutes.

## 1.2 Overview of Language Models

Transformer models are one of the most popular choices in working with language model algorithms. They are relatively recent developments in the machine learning sphere. First introduced in 2017 by Vaswani et al. in *Attention is All You Need* [VSP17], transformer models have transformed the way machine learning models approach text data. They distinguish themselves from other deep learning algorithms prevalent in NLP tasks due to their ability to connect long-range dependencies and their highly-parallelizable structure. Transformers are valuable for longer lengths of text, where the entire content of the text must be considered concurrently in order to extract a meaningful representation of the full context of the text.

Before the introduction of transformers, other widely used methods include Recurrent Neural Networks (RNN) and Convolutional Neural Networks (CNN).

### 1.2.1 Related Methods

For a long time, RNNs were the primary method employed in NLP tasks, particularly for sequential data. Sequential data is any data where the order of the sequence affects the meaning of the model, such as the order of the words in a sentence. Because RNNs retain a memory of the past data they receive, they perform well in predicting the next element in a sequence.

RNNs are based on traditional feed-forward neural networks. Feed-forward neural networks are the simplest form of artificial neural networks (ANN). ANNs consist of interconnected nodes which process input data and produce an output data. Data flows through an input layer, which receives raw data, to hidden layers. In these hidden layers, the input data gets multiplied by the neurons with a weight and added to a bias, followed by an activation function, which introduces an element of non-linearity. The number of hidden layers vary depending on the complexity of the problem, and it is in these hidden layers where the model attempts to learn the patterns of the data in order to produce an accurate output.

Each neuron transforms the input in order to minimize the difference between the predicted output and actual output, sometimes referred to as the target output. During training, these weights and biases get updated and adjusted as the model learns more about the data. The ANN then outputs its predicted values based on what it learns during training.

A feed-forward neural network is simply an ANN where the information is processed unidirectionally forward, every element in the input is processed at once, and moves in a forward pass through the hidden layers to the output layer, and never in any other direction [Saz06]. Because of this, feed-forward neural networks do not hold any perception of order within their input data.

RNNs, however, set themselves apart from feed-forward neural networks by moving information in cycles [Sch19]. Each element in an input sequence of the RNN gets processed independently and sequentially. The inputs in RNNs can loop through their hidden layers multiple times, capturing information about previous inputs and maintaining a sort of memory. Given an input sequence of elements, at each time step, an element passes through the hidden layer and maintains a hidden state. The element then receives the hidden state of the previous time step, and combines it with its current hidden state to create a new hidden state. Thus, at each time step, the RNN maintains information about previous steps, and holds a memory about past information. A loss function is then calculated and a backward pass, referred to as back propagation, then updates the error to the network to update the weights of the model.

The introduction of memory to machines plays a significant role in their evolution to providing more accurate predictions. This memory allows machines to capture dependencies within inputs. In the realm of NLP, it facilitates a way for words to relate to other words in a sequence. In text tasks, words commonly have different meanings according to the context in which they appear in. The study of semantics deals with the meanings of words. Different words in different parts of a sentence can and often times do have very different meanings. However, the study of semantics becomes complex, as oftentimes the context required to fully understand the meaning of a single word does not appear in neighboring words, but

rather much further after or before the initial appearance of the word in a sequence. In this case, the ability to remember the content of the entire sentence is imperative to the understanding of a portion of the sentence. As tasks become more complex, and inputs grow to longer lengths, the relevant context can exist sentences away from the initial position of the word.

While standard RNNs hold advantages over traditional feed-forward neural networks, they still suffer from some of their problems. In particular, RNNs suffer from short-term memory loss. RNNs are shown to perform well for simpler tasks with shorter sequences. However, with more complex tasks with longer sequences, traditional RNNs have been shown to provide sub-optimal results [KCH19].

A large reason for the memory loss of RNNs is due to the fact that they are particularly susceptible to vanishing or exploding gradients. Gradients describe how a neural network learn. Models whose output changes vastly in response to a small input change have higher gradients, and they also learn faster. Lower gradients indicate models that learn very slowly; their outputs change slowly in response to input changes. Gradients of the loss function are calculated during the back propagation step where RNNs calculate their error. When RNNs experience vanishing gradients, they suffer from a sort of memory loss, where the model "forgets" earlier parts of the sequence, and struggles to maintain long-range dependencies.

When RNNs perform their back propagation step, they compute the gradients of the loss function with respect to the model's parameters. The gradients are then multiplied by the model's weights at each time step. After many time steps, if the model weights are small enough, the gradient decreases exponentially, essentially vanishing. This means that the memory from earlier time steps also vanish and earlier inputs no longer influence current predictions. On the other side of the spectrum, the gradients can explode after many time steps when the model weights grow very quickly. At this point, the model becomes extremely unstable, and model weights can be so large they are incapable of being updated. Vanishing and exploding gradients are especially problematic for natural language processing tasks where sequences tend to be long and the words at the beginning of a sequence hold

importance to words later on in the sequence.

Long-Short Term Memory (LSTM) is one way that RNNs can solve their vanishing gradient problem. LSTM incorporates a gate into the model which determines whether an input is useful or not, and then erases the unimportant inputs. This helps RNNs to solve their long-range dependency problem.

However, as sequences grow longer, RNNs that use LSTM become extremely computationally expensive. Typically, more complex data favor models which are parallelizable. Parallelizable models are able to perform multiple computations on multiple data points simultaneously, creating a highly efficient environment for learning. Since RNNs are sequential models by nature, they are restricted by order in which their input sequence must be processed, and must be trained on one element at a time. Thus, RNN models cannot train multiple data points at once and can only feasibly become so complex under computational restrictions.

CNNs are a highly parallelizable and popular pattern recognition model worth mentioning. However, they are mostly used within computer vision systems rather than language tasks. Convolution leverages two different functions to create a third one. CNNs consist of three layers: a convolutional layer, a pooling layer, and a fully connected layer. In the convolutional layer, the network learns how to detect patterns by applying a convolutional function to the input data using kernels. The pooling layer reduces complexity within the model by reducing spatial size. Finally, the fully connected layer connectes neurons from each layer to neurons in the next layer [ON15].

Although there are variations of CCNs which perform better on NLP tasks, traditional CNNs may not perform as well as other models on sequential tasks and tasks requiring long-range dependencies [KRG23]. They leverage matrix calculations and focus mainly on grid data, such as pixels in images, where the spatial relationships between neighboring elements are important. Text data is normally not structured in a way that the neighboring words are the most important to their comprehension.

Transformer models do not face many of the disadvantages of RNNs and CNNs. The introduction of transformers has become such a widespread success due to their ability to capture long-range dependencies so well, while also maintaining a highly parallelizable environment.

### 1.2.2 History and Evolution

The history of NLP models span far before transformer models, RNNs, and CNNs. Early methods began with simple text vectorization methods, ways to convert words into numerical representations [Rud18].

One of the earliest models was the Bag of Words (BoW) model. The BoW model provides a way to represent a document in a numerical representation. Using this model, a document is converted to a "bag" of words - a list of vocabulary disregarding sequence. Each unique word is represented by a number indicating its frequency. The BoW approach is effective when dealing with relatively simple tasks. For simple sentiment analysis tasks, a document can be analysed by looking for frequency of positive or negative associating words. Of course, there are many limitations with this method as well. Order and context are not in any way considered. But for tasks requiring only a rough understanding of text, it is a straightforward and computationally inexpensive method.

Another evolution of text vectorization is Term Frequency-Inverse Document Frequency (TF-IDF). TF-IDF is a way to quantify how important a word is to a document within a group of documents, referred to as a corpus. It multiplies term frequency, a metric describing how often a word appears in a document, and inverse document frequency, a metric describing how often the word appears across the entire corpus. TF-IDF assigns a higher rate to a word that appears frequently in a certain document, but not as frequently in the entire corpus. This usually indicates that the word is particularly informative to the document which it appears in many times. Words that are more common throughout the entire corpus receive lower scores.

Prediction based modeling such as RNNs and LSTM were eventually introduced and found valuable spaces in NLP tasks. Word embeddings used along with neural networks provide a powerful way to predict text. Word embeddings are text vectorizations which consider the meanings of words. They create similar representations for words with similar meanings. Word2Vec was prominent word embedding to surface. Word embeddings such as Word2Vec provided a powerful, new way to represent words. While previous vectorizations of text into numerical vectors didn't consider semantic meanings of the words, Word2Vec uses an unsupervised learning method to learn understand semantic meanings. Word2Vec and similar methods showed the importance of consideration of linguistic meanings within vectorizations.

Word2Vec was mainly trained with two different architectures. The first one, Continuous Bag of Words (CBOW), trains the model to predict the target word based on context. The model takes its input as all words in the sequence except for the target word, and the target word as its output. The hidden layers in the model attempt to learn the word embeddings. The second architecture, skip-gram, flips the model around, and takes the target word as its input, and trains the model to predict the output as all the context words.

Embeddings from Language Models (ELMo) was the next breakthrough in NLP modeling. When ELMo vectorizes a certain word, the vector comes from a function of the entire sentence that the word appears in. This was a breakthrough because ELMo was one of the first models to be able to represent different meanings based on context within the same word. ELMo uses a Bidirectional LSTM model to process the sentence forward and backward in order to capture dependencies in both directions. Its bidirectional architecture greatly contributes to its efficacy by holding the ability to capture context in both directions of a word. Many unidirectional sequential models lose a certain amount of complexity by only capturing dependencies in one direction.

More recently, transformers have been introduced and have left a great impact within the machine learning sphere. They solve many of the problems faced by their predecessors. Transformer models leverage an attention mechanism to understand the context of and

consider all past information, so they don't suffer from the same memory loss issues as their counterparts. Transformers' abilities to process all the information they receive at once without memory loss makes them particularly powerful and sets them apart from other models.

## 1.3 Overview of Transformer Models

Transformers have quickly risen to the forefront method of choice in modeling for NLP tasks in current day research. Although originally intended for language tasks, they've also been proven to be useful in many computer vision tasks as well [KNH22].

Transformer models introduce a self-attention mechanism, which is the foundation to the model's exceptional performance. The self-attention mechanism allows the model to assign a special weight, or pay attention, to the parts of a text it deems most important to the contribution of understanding of the full text. Thus, it is able to process longer pieces of text, holding special consideration to the more important components, without suffering from short-term memory loss, as many of its neural network counterparts do. It also introduces positional encodings, information about the word's position in the sequence. This makes it unnecessary for the model to process one word at a time, allowing for an entire sentence to be processed at once, making the model parallelizable and more efficient. Transformer models in nature are exceptional at capturing long-range dependencies. Unlike RNNs, transformers do not operate on recursion, they create a highly efficient architecture which allows them to be trained on vast amounts of data and learn quickly. The combination of these qualities has led to the growth of the transformer model as the dominant model in current NLP tasks.

There are currently many pre-trained transformer models that exist. Pre-trained models are generic models that are trained on large datasets for general tasks. These pre-trained models are trained for many days with powerful processing units. They use extremely large datasets and have many model parameters, encompassing and representing large amounts of knowledge. Pre-trained models take on the computationally expensive and extensive training

of foundational language structures necessary for all NLP tasks. Once a pre-trained model possesses the basics, it can be even further fine-tuned on specific datasets for more specific tasks.

One of the first prominent pre-trained transformer models to arise was GPT (Generative Pre-Trained Transformer) by OpenAI. GPT is a unidirectional transformer model, meaning it attends to one token associated with a word at once, before then attending to the token to the right of it. After GPT, BERT (Bidirectional Encoder Representation from Transformers) was introduced by Google AI in 2018 [DCL2019]. By creating an encoder-only model, BERT is able to bidirectionally attend to any word of its input sentence. GPT and BERT have since evolved into many updated versions (GPT-2, GPT-3, GPT-4 and RoBERTa, DistillBERT, XLNet, etc., respectively). Subsequent versions have made improvements in both model performance and efficiency using a variety of techniques. They trained existing models with more data and for longer times, modified number of parameters, adjusted learning objectives and training methods.

This paper will focus on fine-tuning BERT. BERT is a strong pre-trained model, but was created with the intention of further fine-tuning to be performed on specific tasks. Subsequent BERT based models, roBERTa and distilBERT will also be fine-tuned. RoBERTa was introduced shortly after BERT [LOG19] . It further trains BERT on a more comprehensive dataset as well as tailoring some of BERT's design choices. DistilBERT was introduced in [SDC20] to provide a smaller, faster, and lighter version of BERT.

# CHAPTER 2

# Background and Architectures

## 2.1 Overview of Sentiment Analysis

This paper focuses on the sentiment analysis of reviews. With the rise of online resources, reviews have become a unique and crucial focal point of many companies. Reviews provide a particularly important value to consumers and sellers alike and have transformed the way that many businesses market and provide customer service.

They are not only the forefront platform for companies to receive direct feedback, they also provide a platform for potential customers to receive unbiased and real advice on the goods or services before they buy them. By analyzing a company's own reviews as well as potential competitors' reviews, a company can acquire valuable product research. However, reviews are also particularly difficult to process. Vast amounts of text encompassing high word counts make it difficult to be manually monitored by humans. Furthermore, the nature of user-generated content makes it difficult for even humans to discern meaning at times [Gun23].

Sentiment analysis as a study has existed through many transformations. Early forms of sentiment analysis were lexicon-based, sentences were labeled based on certain word related rules. A common method, similar to the BoW approach, creates a list of "positive" and "negative" terms, and processes sentences to keep a frequency count of both sentiments. The most simple form of this model then calculates a sentiment score (StSc) in the form of an average:

$$StSc = \frac{\#\ of\ Positive\ Words - \#\ of\ Negative\ Words}{Total\ \#\ of\ Words}$$

where a negative score deems the sentence negative, and a positive score deems the sentence positive. Subsequent variations of this method record domain specific lexicon tools in order to better label sentiments. Rather than using a general list of positive and negative words, this method can be modified to fit more distinct tasks.

However, sentiment analysis faces some unique challenges within natural language processing that are better suited for machine learning based models rather than rule-based models to solve. Since the data used in this study is entirely user-generated, it is not as simple as picking and choosing textbook words associated with "positive" and "negative" emotions. Human language is a complicated study even when closely monitored by real humans. The choice of model must properly capture human emotion and represent the complex nature of language.

Language nuances such as sarcasm and irony present to be a unique problem that tends to be difficult to capture for some models and even humans alike. Negation also can be difficult for some models to understand, especially in instances where the word intended to be negated occurs later down the sequence from the negation word. Transformer models are able to handle these complications particularly well. The extensive amount of data they are able to process and train from make them well-suited to learn the patterns and distinctions of human emotion through text.

## 2.2 Model Architectures

To understand the architecture of transformer models, it is helpful to take a look at the inner workings of the models that preceded them.

Sequence to sequence models based on traditional RNNs employ an encoder and a decoder step to map an input to an output. The encoder takes a sequence of elements, and maps

them to a numerical vector, which represents the RNN's hidden state. This vector is also referred to as the context, as it provides an encoding of the previous information received by the model until that point. The vector size of the context is set to the number of hidden units within the encoder. At each time step, the RNN processes an element of the input sequence and then updates its hidden state. Once the entire input sequence is processed, it sends the context, now containing the last hidden state which holds memory of all the previous hidden states, to the decoder. Conditioned on this context, the decoder then begins to generate an output sequence of symbols, one element at a time.

As explored earlier in this paper, problems tend to arise with the traditional RNN model, particularly when dealing with longer input sequences. When processing these longer sequences, the RNN's hidden states struggle to retain relevant information about earlier time steps.

With the introduction of attention models, this issue is mitigated by giving focus to the more important parts of the input sequences. Similar to a traditional RNN model, the encoder in an attention model still processes elements and updates its hidden states in the same way, except now, it passes on all the hidden states to the decoder, as opposed to only the last hidden state. The decoder now has all of the encoder hidden state vectors, along with its own initial decoder hidden state. Then, the decoder in the attention model scores each hidden state it receives to assign attention weights to each state. Next, it softmaxes each score and multiplies each hidden state vector by its softmax, creating a series of weighted vectors. Now, the model can focus its attention to the more important hidden state vectors, which are weighted higher than the less important ones.

A key component of transformer models lies within their attention mechanisms. Transformer models primarily use self-attention to determine which parts of the sequence are the most important. Self-attention refers to the particular type of attention used specifically within transformer models. It is conceptually similar to the theory of general attention mechanisms, except self-attention focuses on how models weigh different parts of the same input. Meanwhile, regular attention focuses on how models weigh different parts of another

sequence, a process that extends from the encoding to decoding step.

Similar to RNN models, many transformer models also contain a layer of encoders and decoders. The encoder and decoder layers each contain multiple encoders and decoders, respectively, stacked on top of each other. The input sequence flows through each of the encoder's two sublayers: a multi-head self-attention layer and then a feed forward neural network layer, going from encoder to encoder. After passing through all the encoders, the output of the top encoder is transformed into two attention vectors, to be passed onto the decoder layers. It is passed through each decoder layer, where it travels through three sublayers: a masked multi-head self-attention layer, then a encoder-decoder attention layer (also referred to as a multi-head self-attention layer as in the encoder), and finally a feed-forward neural network layer. The output of the decoder then passes through a final linear and softmax layer where it is converted from a numerical vector into a sequence of symbols.

## 2.3 Attention Models

The transformer is an incredibly powerful model, and its attention mechanism plays a large role in its effectiveness. This section delves deeper into attention mechanisms and attention-based models. Attention models assign heavier weights to the more crucial input elements of the model, and lower weights to pieces which may not contribute as much to the performance of the model. The attention mechanism solves the memory problem that RNNs often suffer from when processing inputs with longer lengths. First introduced in 2015 by Bahdanau et al. [BCB16], the attention mechanism transformed models' abilities to capture long-range dependencies.

Bahdanau's paper focused on neural machine learning on French-to-English translations. Common antecedent machine translation methods focused on encoder-decoder methods that encodes an input into a fixed length vector, and then decodes an output translation, which are then jointly trained to produce the output with the highest probability. The requirement to compress all necessary input information into a fixed-length vector left the model to perform

poorly when faced with longer sentences. The neural learning method proposed instead searches for the position in the input sentence where the most important information lies. For every word, the model focuses on context vectors associated with the positions of the important information, as well as all the previous predicted target words, and then predicts its current target word. By doing so, the whole input sentence does not need to be coded into a fixed-length vector, and rather as a sequence of vectors, which can be subset and chosen based on the specific translation.

There are three main types of attention: global, local, and self-attention. Self-attention is the type of attention used by transformer models, and is outlined more extensively in the next section. Global attention refers to a model where all elements of the input are given some sort of attention. While some elements are given more importance and some are given less, by giving all elements attention, the model suffers high computation requirements. All hidden states must be computed and multiplied by their weight matrices. Local attention solves the computation problems that global attention suffers from. Instead of paying attention to every element of the input, it only considers some when generating context vectors. It chooses which elements to pay attention using alignment. When using a monotonic alignment method, local attention will only pay attention to a set of selected information. A predictive alignment method allows the model predict the position of the information that should be paid attention to.

## 2.4 Transformer Model Architecture

Now the transformer model with all necessary equations and notations is formally introduced.

An encoder is given an input sequence of $n$ elements, $\mathbf{x} = (x_1, ..., x_n)$ and maps it to a numerical vector, $\mathbf{z} = (z_1, ..., z_n)$, of hidden states, also referred to as the context. Once the input sequence is processed, the encoder sends the hidden states to the decoder. Conditioned on the context, the decoder then begins to generate an output sequence of elements $\mathbf{y} = (y_1, ...y_n)$, one element at a time [VSP17]. Similarly, $X$ and $Y$ represent the matrix

representations of inputs and outputs. Figure 2.1 shows a map of the transformer encoder and decoder architecture.

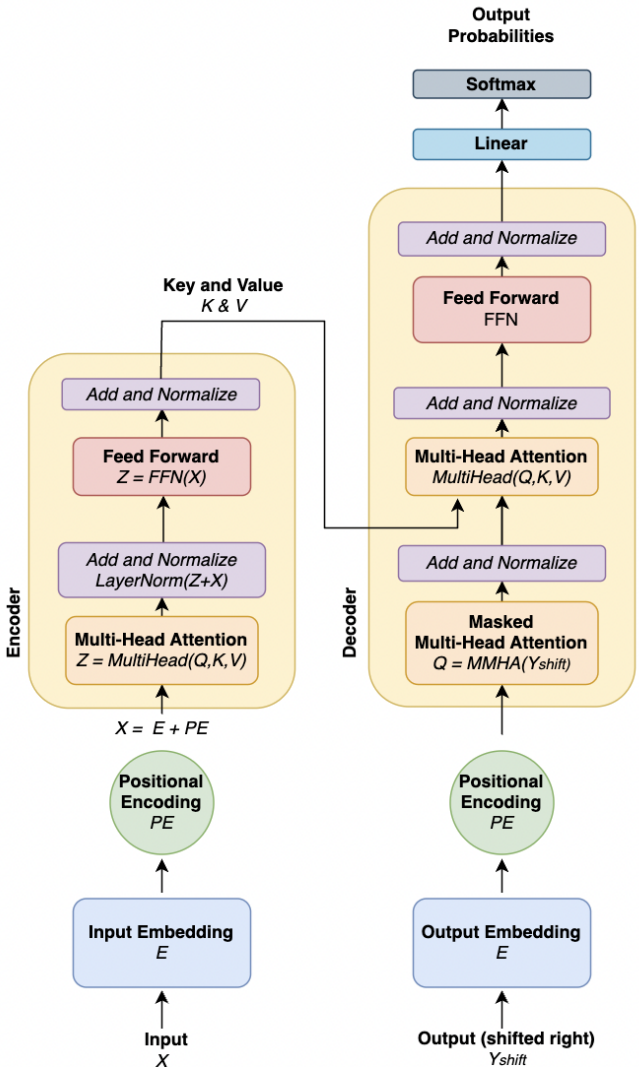


Figure 2.1: Architecture of transformer encoder and decoder.

### 2.4.1 Encoder

The encoder in a transformer is tasked with the job of taking the input sequence and transforming it into a set of contextualized representations. The context holds information about the relationships and dependencies within the input sequence.

Transformer models stack $N$ multiple identical layers of encoders, and the same number of decoders, the exact number varying from model to model based on specific demands of the model. From here on, the original model that was first proposed in *Attention is All You Need* will be referred to as the original transformer. The original transformer employs $N = 6$ encoding and $N = 6$ decoding layers. The remainder of this section will describe a transformer model with six encoding and decoding layers, but it is notable that transformer models can have a different number of layers, and many popular pre-trained models do. This will be furthered explored in later chapters.

#### 2.4.1.1 Embeddings and Encodings

At the bottom encoder, the input is transformed into a numerical vector, through an embedding process, before entering the two sublayers of the encoder. Most neural networks, including transformers, require a continuous valued input. Using word embeddings, an input of elements is mapped to a vector of real numbers that captures both the meaning of the word, as well as the contextual similarity of the word to other words.

The original transformer uses learned embeddings, as do many prominent existing pre-trained transformers. Learned embeddings are trainable and updated based on the task-specific data during training to capture the relationships between words. Pre-trained transformers such as BERT analyzes these relationships bidirectionally to generate meanings based on the context of the entire sentence it occurs in. There are a variety of embedding algorithms, which will be delved into further detail in subsequent chapters.

A positional encoding is also added at this step, which provides information about the relative position of the symbol and gives meaning to the order of sequence of symbols and distance between symbols. This is a key step in the transformer architecture which makes a transformer parallelizable. Due to the nature of parallel processing, the model is unable to learn positional information from data as it processes all words simultaneously. The positional encoding grants the data a way to hold meaning about order while being processed

simultaneously.

Like word embeddings, positional encodings may either be learned or fixed. The original transformer uses fixed sinusoidal positional encodings based on sine and cosine functions:

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

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{model}})$$

where $pos$ is the position, $i$ is the dimension of the positional encoding, and $d_{model}$ is the dimension of the model.

Since sinusoidal functions are periodic, they are suitable choices for representing relative positions due to their regularity of patterns. However, in demand for more model flexibility, many pre-trained use learned positional encodings. The BERT architecture used throughout this paper uses learned positional encodings. Using learned positional encodings allows for more flexibility within the model's ability to perform a wider variety of tasks since they are able to adapt and learn according to the specific task.

The input embeddings and positional encodings are added together, so that $X = E + PE$ where $E$ is the matrix of word embeddings, and $PE$ is the matrix of positional encodings.

The input then passes through each sublayer, where the function $Sublayer(X)$ is implemented by its respective layer. Each sublayer has a residual connection, and then it is normalized. The resulting output can be described as $LayerNorm(X + Sublayer(X))$ of dimension $d_{model} = 512$.

#### 2.4.1.2 Mulit-Head Self-Attention Layer

The first sublayer the input passes through is the multi-head self-attention layer. The purpose of this layer is to capture contextual information and dependencies within sequences. It does this by calculating an attention score. This can be done through dot-product attention and additive attention. Additive attention involves using a feed forward network and a single hidden layer, which can lead to lower optimization. This paper will focus on transformer model architecture using the scaled-dot product attention.

In order to calculate the attention score, each word is first transformed into a set of three vectors of dimensionality of 64: a query, $\mathbf{q} = (q_1, ..., q_n)$, key, $\mathbf{k} = (k_1, ..., k_n)$, and value, $\mathbf{v} = (v_1, ..., v_n)$, vector. Similarly, a matrix of queries, keys, and values can be expressed as $Q$, $K$, and $V$, respectively. In practice, calculations are done in matrix form to compute attention scores. The query matrix, $Q$, holds information about the element which attends to other elements in the sequence. It helps the model decide which element to focus on. The key matrix, $K$, holds information about the element which is attended to. The value matrix, $V$, holds information from the chosen element that will be used for the weighted sum calculation based on attention scores.

More simply put, the query helps the model decide which elements are most important to focus on. The key then helps the model to understand the general characteristics and features of the elements chosen by the query. Finally, the value holds the specific characteristics or features of the elements that are most useful for the model to use.

To calculate $Q, K$ and $V$, a matrix of embeddings, $X$, is multiplied by $W^Q, W^K, W^V$, learned weight matrices for each query, key and value. Each row in $X$ represents an element in the input sequence. The weight matrices are trained along with the other learned parameters of the model. The weight matrices are expressed as such:

$$X \times W^Q = Q,$$
$$X \times W^K = K,$$
$$X \times W^V = V.$$

Next, a self-attention score is calculated for each word. The dot product of the query and key vector is taken, $q_i \cdot k_i$, for $i = 1, ..., n$ and divided by the square root of the key vector dimension, $\sqrt{d_k}$. The division here is to provide numerical stability to the score, scaling the score to prevent it from getting too large. Then, the value goes through a softmax operation which normalizes the score. Thus, the words with more contextual importance have higher values.

Each vector is then multipled by its softmaxed score and then summed up. In practice,

these steps are all done in matrix form for efficiency, and can be expressed as :

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

where $T$ is sequence length.

The original transformer, as well as transformers focused in on the paper, use multi-head attention. Multi-head attention maps each input element to $h$ different queries, keys, and vectors. By doing this, the model is able to express different patterns within each sequence, reduce redundancy, and overall allow for a deeper understanding of each word within its sequence than regular self-attention.

The original transformer uses $h = 8$ attention heads, and the number of attention heads can and do often vary from model to model. The attention heads can be executed in parallel, leading to efficient processing. Mathematically this can be expressed as:

$$X \times W_i^Q = Q,$$
$$X \times W_i^K = K,$$
$$X \times W_i^V = V,$$

where $i = 1, ..., h$ and $W_i^Q \in \mathbb{R}^{d_{model} \times d_q}$, $W_i^K \in \mathbb{R}^{d_{model} \times d_k}$, $W_i^V \in \mathbb{R}^{d_{model} \times d_v}$

Then, each set of the matrices goes through the $Attention(Q, K, V)$ function. The eight attention heads create eight matrices, $Z_1, ..., Z_8$. All eight matrices are concatenated into one large matrix, and then they are multiplied by a weights matrix, denoted as $W^O$, where $W^O \in \mathbb{R}^{d_{model} \times hd_v}$. $W^O$ is trained with the model similarly to the weight matrices in regular self-attention. The result is one matrix, $Z$. This function can be written as:

$$MultiHead(Q, K, V) = Concat(head_1, ..., head_h)W^O,$$

where $head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$

There is a residual connection around this step, which adds together the input of the multi-head self attention layer, to the output of the multi-head self attention layer, $X + Z$. This ensures that the model can pass on the original information from the input, as well as

mitigating the vanishing gradient problem by creating a direct path from output to input. Finally, the output goes through a layer normalization step to provide stability to the model. It is necessary as it creates a consistent scale across the positions of the elements.

Mathematically, this can all be summarized as :

$$LayerNorm(X + Z),$$

where $Z = Multihead(Q, K, V)$.

#### 2.4.1.3 Position-wise Feed Forward Network Layer

Next, the information passes through a position-wise feed forward network (FFN) layer. The purpose of this sublayer is to process the information from the previous step in a position-wise manner. In doing so, it transforms the output of the multi-head self-attention layer to better fit the next self-attention layer as an input. The feed forward network layer allows the model to better capture complex patterns within the data. It does this by using two linear transformations with Rectified Linear Unit (ReLU) activation.

Mathematically, this can be written as:

$$FFN_1(x) = ReLU(xW_1 + b_1)$$

where $W_1$ is a weight matrix and $b_1$ is the bias term. The weight and bias are learned during the training process through back propagation. The weight matrix captures the relationships between inputs. The bias is an offset that represents the constant terms in the relationship between inputs.

Activation functions are helpful in machine learning models because they introduce non-linearity to the model. Non-linear models are more flexible and possess higher ability to capture the complex patterns of the data, than their linear counterpart. Its flexibility manifests in many forms. Many types of data do not conform to linear relationships, and thus cannot be expressed as so. Since non-linear models hold a higher degree of of representation power, they can model a wider variety of functions. They are also able to express

non-independent variables and interactions. Linear models assume that the impact of one variable is independent to the values of other variables, and cannot properly show these relationship that do not fall under this category.

The choice of activation function depends on model requirements. Subsequent models introduced in this paper use other activation functions. The ReLU function is more computationally simple than other activation functions, as there are no complex computations involved, it simply passes its input through. The ReLU function is helpful in solving the vanishing gradient problem that other functions face, since it doesn't saturate its positive inputs. Its gradient is always 1 for positive inputs.

The ReLU function returns $x$, if $x$ is positive, and returns 0, otherwise. The piecewise linear function can be expressed as $ReLU(x) = max(0, x)$. In terms of our problem, it would be:

$$FFN_1 = ReLU(xW_1 + b_1) = max(0, xW_1 + b_1).$$

The FFN processes one vector at a time independently, thus it is able to process them in parallel. This is important because it mitigates the computationally expensive problem that traditional neural networks face when they perform similar functions sequentially.

The FFN then passes the data through a second linear transformation, such that:

$$FFN_2(FFN_1) = max(0, xW_1 + b_1)W_2 + b_2 = FFN_1(x)W_2, +b_2.$$

This introduces additional non-linearity for additional ability to capture more complex patterns.

Similar to the previous attention sublayer, a residual connection is added to the output, and then the sublayer is normalized. The output of this layer is:

$$LayerNorm(X + Z),$$

where $Z = FFN_2(FFN_1(x))$.

The output is then sent to the next encoder to go through the self-attention and feed forward network steps again, until it reaches the last encoder.

### 2.4.2 Decoder

Once the data has gone through all the encoder steps, it begins its journey through the $N$ decoders. The decoder takes two inputs: the output of the top encoder, as well as the output text shifted to the right. Then the decoder outputs a vector $y$, which is the target sequence. Depending on the task of the transformer, $y$ could be a variety of different types of sequences or values. In sentiment analysis, it is common for $y$ to be a scalar value or categorical labels. The analysis in this paper will express $y$ as a binary classification of values 0 and 1.

Similar to the encoder architecture, the output travels through a word embedding and positional encoding, followed by sublayers, each encompassed with a residual connection and followed by a layer normalization. The original decoder structure contains three sublayers: a masked multi-head self-attention layer, a multi-head self-attention layer, and a feed forward network layer. BERT, in particular, has an encoder-only architecture, and does not use a decoder. BERT specific architecture will be explained more in Chapter 3.

The two inputs of the decoder enter at different sublayers. The output text, which is shifted to the right, also goes through an embedding and positional encoding step, similar to the input in the encoder step. The output text is shifted to the right, inserting a "start of sentence", <SOS>, token in the first position, in order to make sure that predictions made at position $i$ only depend on elements in earlier positions. This sequence is then sent to the masked multi-head attention layer.

In the original transformer, a masked multi-head attention layer allows the decoder to attend to previous positions in the sequence while also preventing it from looking at future positions while it is training. This essentially keeps the model from "cheating" by looking at future positions to make its present predictions. The encoder and decoder steps are similar, however now the model applies a mask to the future attention scores, by setting them to $-\infty$. By setting the future position scores to $-\infty$, the autoregressive property of the model is retained. An autoregressive model predicts future values based solely on past values. This property allows the model to learn and predict in a realistic way. The masked multi-head

attention layer outputs a query, $Q$.

The next input of the decoder enters at the multi-head self-attention layer. The output of the top encoder is transformed into keys, $K$, and values $V$ and enters the multi-head attention layer. In a similar process as the previous encoder step, using $Q$ from the masked multi-head attention layer, the multi-head attention sublayer takes queries ($Q$), keys ($K$), and values ($V$) to compute the self-attention score:

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

Then, the attention scores are softmaxed and summed and weighted as they were in the encoder step. In the softmax steps, the masked positions get assigned zero probability. The sequence then passes through the feed forward network layer and layer normalization layers with residual connections as it does in the encoder step. The decoder outputs a vector of floats.

### 2.4.3 Linear and Softmax

In the final step, the vector of floats, $f$, is converted into the target sequence. The linear and softmax steps are important to transform the prediction into a comprehensible output structure in a form that is suitable for the problem. First, it enters a linear transformation. The linear transformation transforms the outputs into a logits vector, $l$, raw unnormalized predictions. The matrix linear transformation is in the form of:

$$L = F \cdot W_{out} + b_{out},$$

where $W_{out}$ is a learned weight matrix and $b_{out}$ is a bias term.

Then, it passes through a softmax activation function. For models which predict text, the logits vector gets transformed into a probability distribution over the target vocabulary in the softmax step. The softmax function normalizes the logits to a probability distribution

that sums to 1, and is in the form of:

$$P(y_i) = \frac{e^{l_i}}{\sum_{j=1}^{Voc} e^{l_j}},$$

where $P(y_i)$ is the probability of the ith word in the vocabulary, $Voc$ is the size of the vocabulary, and $l_i$ is the logit for the word.

The probability distribution corresponds to a vocabulary, where the model will output the word with the highest corresponding probability. The word becomes a part of the input for the next time step for the next word, and the process repeats until an "end of sentence",<EOS>, token is received.

The model is trained to maximize the likelihood of the next word in the sequence. During training, the predicted probability is then compared to the actual distribution, a one-hot coded representation of the target word, to optimize some loss function. Common choices are cross entropy loss functions and binary cross entropy loss functions. A cross entropy function is in the form of:

$$CE(p, q) = -\sum_{i=1}^{n} p_i log(q_i),$$

where $p$ is the true probability distribution, and $q$ is the predicted probability distribution.

The target sequence of a binary sentiment analysis makes this step a little more simple, as the model only has to output the element with the highest corresponding probability, which is either 0 or 1 in the case of binary analysis.

The cross entropy function measures how well the predicted probability distribution represents the true distribution, penalizing the model for larger deviations away from the true distribution. The function employs a *log* term to penalize the model heavier for providing an incorrect prediction with a higher probability than an incorrect prediction with a lower probability.

The gradient of the loss is calculated with respect to the parameters, and then the parameters are adjusted to reduce cross entropy loss. The process is repeated for multiple epochs until the model parameters converge.

Thus, a final sequence is produced, $y$.

## 2.5 Pre-Trained Models For NLP Tasks

Using pre-trained models as opposed to training a transformer model completely from scratch can be advantageous in many ways. This paper will focus on training BERT, a pre-trained transformer model. It is important to overview the basics about pre-trained models and their purpose in machine learning.

A pre-trained model is a saved network that is already trained on a general and large dataset [QSX20]. They are valuable in machine learning because they save vast amounts of training time as training does not have to be restarted from scratch for every task. Pre-trained models allow for a good representation of weights to start from. These weights are probabilistic representations that have been derived from the pre-training process. Thus, by using our pre-trained models, our parameters can be expected to converge faster and more efficiently.

In most models, the bottom layers of a model are more basic and they are similar across a large variety of tasks. These general features do not usually need to be changed in order to cater to specific tasks. The top layers tend to be more task-specific. When these top layers are unfrozen, they can very effectively be fine-tuned to perform more distinct tasks. In the case of this paper, they are fine-tuned for classification. It is also possible to add additional layers on top of existing pre-trained layers to customize the models.

Pre-trained models are especially effective for NLP tasks, as NLP tasks generally require a large, but universal, representation of language. The bottom layers of models are able to hold information about these vocabularies and linguistic rules. Then the top layers can be unfrozen and tuned to learn more nuanced patterns of required by the specific downstream task. It is also possible to unfreeze all parameters of the model and fine-tune all the weights to cater to a specific dataset. However, this tends to be computationally expensive and unnecessary, as the model weights for bottom layers are unlikely to require intense adjustments

across different datasets.

When considering text representations for NLP tasks, there are two main types of embeddings for words. Non-contextual embeddings describe a representation of discrete language symbols. For each word $x$ in vocabulary $V$, it is mapped to a vector $e_x \in R^{D_e}$ with a lookup table $E \in R^{D_e X|V|}$, where $D_e$ is a hyperparameter indicating the dimension of the token embedding representing the word. The embedding of these words do not change based on their context. Their use falls short in some areas, since in colloquial language most words do have multiple meanings that vary depending on context of word. The more effective type of embedding, context embeddings, change depending on the context the word appears in. These emebeddings can be represented as $x_1, x_2, ..., x_t$ where $x_t \in V$ is a sub-word with $t$ different representations that corresponds to the representation of $x_t$, depending on the context in which it appears [QSX20]. Transformer models use a fully-connected self-attention model as their neural contextual encoder to create context embeddings and directly model the dependency between words in a sequence.

However, contextual embeddings require a much larger training dataset than a non-contextual embeddings. For most supervised NLP tasks, the datasets are not large enough to train an effective model. Thus, pre-trained models which have been trained on datasets large enough to properly learn contextual embeddings are very advantageous.

Masked Language Models (MLM) are a self-supervised model that combines supervised and unsupervised learning techniques to predict on inputs. Similar to supervised learning techniques, an MLM uses training data with explicit labels. However, MLMs proceed to mask certain words and make predictions on the masked words. This masking can be static or dynamic. Static models, such as BERT, mask the same words for every epoch of training. Extensions of BERT, such as roBERTa, use dynamic masking, where the model changes which words it masks during training. Dynamic masking has been shown to result in a higher accuracy since the masking strategy is different for each training epoch [LOG19]. By using different masking techniques, pre-trained models gain a deeper understanding of the contextual meanings of their training vocabulary.

Pre-trained models can also be built with a variety of number of layers and parameters. $\text{BERT}_{BASE}$ is trained with 12 layers and 110 million parameters. $\text{BERT}_{LARGE}$ is trained with 24 layers and 340 million parameters. It is able to process more complex data than $\text{BERT}_{BASE}$, but it also is much more computationally expensive. DistilBERT compresses $\text{BERT}_{BASE}$ layers by a factor of 2x. In doing so, it is able to perform with much less computational power. This paper will focus on the fine-tuning of $\text{BERT}_{BASE}$ and will subsequently be referring to $\text{BERT}_{BASE}$ and BERT interchangeably.

Different pre-trained models can be more advantageous to certain NLP tasks. The choice of pre-trained model is important to model performance. Another way to customize a pre-trained model is to perform transfer learning.

## 2.6 Overview of Fine-Tuning

The next section delves deeper in fine-tuning techniques. Fine-tuning falls under a broader category of transfer learning. A brief explanation of different types of transfer learning is provided. Then, a more in-depth explanation of the type of fine-tuning techniques will be explained. The limitations of traditional fine-tuning will be discussed as well as the methods used to circumvent the problems.

### 2.6.1 Transfer Learning

Transfer learning provides a way for a pre-trained model to be better customized to a specific task. It involves taking one model and re-purposing it for a new task. In this case, taking a pre-trained model and making it more task-specific. Transfer learning saves a significant amount of training time required, as well as on the amount of data required. The idea behind transfer learning is that the weights and biases of a model trained on one task can be utilized toward a model training to perform a different task.

Fine-tuning is a prevalent type of transfer learning. In fine-tuning, the pre-trained parameters are unfrozen and the model weights are updated according to the task which the

model is tuned to. By re-training the layers of the model on a dataset specific to the new task, the model is able to update its knowledge to the needs of the new task. For tasks such as sentiment analysis, fine-tuning is particularly advantageous since a pre-trained model may lack important domain knowledge essential to the comprehension of the language unique to the task. In this paper, both the terms fine-tuning and training are used, with the term training just referring to the re-training of the unfrozen parameters on the particular dataset and task employed for the fine-tuning process.

Hyperparameter tuning refers to the adjustment of the model settings which are not learned during the training or fine-tuning process, but are rather set before the process begins. During fine-tuning, it is beneficial to tweak the hyperparameters of the model, such as the batch size that data is processed, its learning rate, drop out rate, and the number of epochs the model is trained for. These directly affect the performance of the model, but are usually manually determined before the training process, rather than learned.

Feature extraction uses pre-trained representations to extract important features from new data. By adding new layers to a pre-existing model and training the new layer using the important knowledge from the existing model, the model is able to learn more efficiently.

### 2.6.2 Parameter-Efficient Fine-Tuning

Fine-tuning on every parameter of a model can be an incredibly computationally expensive process. Although many pre-trained models are built with the intention of eventually being fine-tuned on downstream tasks, traditional approaches of fine-tuning where every parameter is updated can be unfeasible and unnecessary. BERT has 110 million parameters, which would require an unpractical amount of computational power to fine-tune on every one of the entire model's parameters. Parameter-efficient fine-tuning (PEFT) focuses on more computationally efficient fine-tuning methods which reduce the number of trainable parameters, while still retaining the performance of models which would be fine-tuned traditionally [XXQ23]. PEFT also mitigates the overfitting risks that occur when all parameters

are updated during fine-tuning. This section will review four types of PEFT that can be used specifically with transformer models.

*Additive Fine-Tuning* incorporates new task-specific trainable parameters into the model. Adapter-based additive fine-tuning introduce an adapter that has parameters which are updated, while the rest of the pre-trained parameters are unchanged. Soft-prompt based additive fine-tuning insert trainable continuous vectors into the input or hidden states of the model. These trainable vectors are called soft-prompts because they possess a quality of adaptability. They search for prompts within a discrete token space based on the task-specific training data and thus can be optimized based on the particular task and training data. These vectors are fine-tuned while the pre-trained parameters remain frozen. Ladder side fine-tuning involves training a ladder side network with the pre-trained model using shortcut connections, called ladders, from the pre-trained model. Unlike adapter or soft-prompt based fine-tuning techniques which inject trainable parameters into the main network of the pre-trained model, ladder side fine-tuning uses a separate network on the side to learn new trainable parameters. There are a variety of variations of different additive fine-tuning techniques that can be used.

*Partial Fine-Tuning* determines the subset of pre-trained parameters which are the most consequential to the performance of the particular downstream tasks, and fine-tunes only on that subset of parameters. This minimizes the number of parameters which need to be fine-tuned. In bias update fine-tuning, the bias term in the attention layer, feed-forward layer, and layer normalization is the only term that is updated. This still leaves most of the pre-trained parameters to remain frozen. In pre-trained weight masking, a binary weight matrix mask is created in order to mask certain weights. This matrix creates a pruning criteria to select which weights will be masked, which discerns which parameters should be chosen to be updated. Delta weight masking uses optimization approximation and other pruning techniques to mask delta weights. This type of pruning usually requires an update at each iteration. A common technique includes selecting some number, $k$, pre-trained parameters to update, which are the $k$ parameters which provide the highest absolute difference in

pre-trained weights between the original model and a fully fine-tuned model.

*Reparameterized Fine-Tuning* reduces the number of trainable parameters by using a low-rank transformation. One variation of reparameterized fine-tuning uses low-rank decomposition techniques. Low-rank decomposition derives a lower-rank matrix from the original matrix that still retains the most important information from the original matrix. Low-rank adaptation (LoRA) places low-rank matrices into every layer of the transformer to be trained, while the rest of the model weights are frozen. When models are adapted to include a smaller number of task-specific parameters, the models tend to underperform in comparison to full fine-tuning. LoRA uses the low-rank matrices to represent the dense layer's change during adaption, and is able to efficiently optimizes the rank decomposition of these lower-rank matrices. LoRA derivatives encompass a group of reparameterization techniques based on LoRA [HSW21].

*Hybrid Fine-Tuning* uses a combination of the above variations of fine-tuning. Manual combination incorporates the structures and features of different methods in order to achieve higher performance or efficiency. Automatic combination seeks to employ various structure search and optimization methods to automatically find a way to incorporate the different techniques.

# CHAPTER 3

# Experiment and Methodology

This chapter will explain the pre-trained models that will be used, as well as the model-specific architecture and expected results for each model. It will outline the experiment set up, including an in-depth description of the training and test data sets. It will include a brief description of how to fine-tune models for sentiment analysis specifically, and delve into an in-depth description of how each model will be altered.

## 3.1 BERT Architecture

This paper will discuss the fine-tuning of BERT, as well as a two of its variations, roBERTa and distilBERT. BERT employs a transformer encoder that is trained bidirectionally specifically for language modelling. BERT does not include a decoder. Training bidirectionally allows the model to apply its self-attention mechanism to the elements both left and right of itself rather than just in one direction, leading to a deeper model understanding of contextual meaning.

The reason for choosing BERT as opposed to other prominent pre-trained models is due to its approach to NLP tasks and the variety of different extensions from BERT to explore. BERT, in particular, is a popular model for sentiment analysis due to its ability to capture bidirectional context. As mentioned earlier, BERT uses contextualized word embeddings, allowing the representation of a word to adjust to the meaning of the sentence. As sentiment analysis takes in an input of a user-generated sentence, this property is particularly important to capture the nuances of casual language representation.

The original transformer as well as other pre-trained transformer models are unidirectional, where the model attends to tokens sequentially in only one direction. This is effective for next sentence prediction tasks where the goal is to predict the next word in a sequence. However, when fine-tuning the unidirectional models for other tasks, such as sentiment analysis, they may be sub-optimal. This is because sentiment analysis greatly relies on the context of an entire sentence to determine whether or not, as a whole, the sequence is positive or negative. It is generally difficult to determine sentiment from context from only one direction of a sentence.

The way BERT solves the unilateral directional problem is by using a MLM as introduced earlier. Unidirectional models cannot train bidirectionally, because seeing both the left and right direction would allow model to directly see the word it is trying to predict, thus making the entire process ineffective. Since MLM's mask the tokens at random, they do not suffer from the redundancy of this problem. By masking certain words, MLM's are not predicting words that they have already seen, and are able to tackle some of the limitations of unidirectional models.

As noted earlier, BERT architecture only includes an encoder, rather than the traditional encoder-decoder model. By masking certain tokens, each masked token is conditioned on the other tokens in the sentence. Because of this MLM objective, BERT does not need a decoder. Decoders are used to predict the next token in the sequence. BERT, being bidirectional, predicts on these masked tokens.

The first four lower layers of BERT are shown to contain information about linear word order [RKR21]. As the position of the layers increase, BERT begins to hold more knowledge about hierarchical sentence structure, the part of the sentence syntax which concerns the way full sentences are formed from smaller sub-phrases and words. The middle layers of BERT contain the most information about syntactic information. Syntactic information tends to be very transferable across various language tasks, as language follows similar rules regardless of task choice. While syntax, information about the structure of sentences and language, is mostly concentrated within the bottom layers of BERT, semantics, information about the

meanings of words, is spread throughout the entire model. The top final layers of BERT are the task-specific layers.

When BERT takes an input, special tokens are added to the input. The tokens <CLS> and <SEP>, classification and separation, are added during text pre-processing. The <CLS> token appears at the beginning of each input and the <SEP> token appears at the end of the input to indicate different segments of the input. This process is pre-trained using a 30,000 token vocabulary from WordPiece embeddings.

BERT is then pre-trained in two unsupervised tasks: Masked LM and Next Sentence Prediction (NSP). These are BERT's two training objectives.

In the MLM objective, tokens are randomly masked with a <MASK> token and the model is trained to predict the tokens. The model randomly selects 15% of the tokens and then masks 80% of them, keeping 10% unchanged and unmasked, and replacing 10% with a randomly chosen vocabulary token.

During the NSP objective, the model aims learn if there is a relationship between sentences, and what that relationship looks like. Given sentences A and B, the model predicts whether or not a sentence, B, is the next sentence after sentence A, or if it's just a random sentence. Sentence B can have either label, IsNext or NotNext, where 50% of the sentences B, IsNext, belong after sentence A, and 50%, NotNext, do not make sense contextually to follow A.

The whole pre-training procedure was then performed on 800M words from BookCorpus and 2,500M from English Wikipedia. These corpuses were chosen due to their document format, as opposed to single sentence formats, in order to teach the machine from longer form content.

BERT is optimized using the Adaptive Moment Estimation (Adam) algorithm with hyperparameters $\beta_1 = 0.9, \beta_2 = 0.99, \epsilon = 1e-6$ and weight decay $L_2 = 0.01$. $\beta_1$ and $\beta_2$ describe the exponential decay rates for the first and second moment estimates. $\epsilon$ is a small constant added to the denominator to prevent division by zero and helps to ensure gradient stability.

The learning rate, also known as step size or $\alpha$, describes the rate in which the model is updated. BERT uses an adaptive learning rate which warms up over the first 10,000 steps until it reaches a peak of 1e-4. Then it is linearly decayed.

BERT trains with a drop out rate of 0.1 on its layers and attention weights. Drop out helps reduce the risk of overfitting in models by randomly selecting neurons within the network to drop. When overfitting occurs, the model memorizes the patterns of the training data rather than learning from it. By dropping random neurons, neighboring neurons must adapt to learn the information.

BERT then applies a Gaussian Error Linear Unit (GELU) activation function in the form of:

$$GELU(x) = xP(X \leq x) = x\Phi(x) = x \cdot \frac{1}{2}[1 + erf(x/\sqrt{2})].$$

It is notable that the original transformer employs a ReLU function during this step. The GELU function can be preferable to the ReLU function used in the original transformer because it completely smooth, where as ReLU is not differentiable at $x = 0$.

The entire model is pre-trained for 1 million updates, using mini batches of 256 sequences, and maximum sequence length at 512 tokens. Finally, BERT is trained to contain pre-trained parameter weights, ready to be fine-tuned on.

There are different variations of BERT, including roBERTa and distillBERT that are based on the foundational structure of BERT but have updated or changed parts of the structure in order for increased model performance.

### 3.1.1 RoBERTa Architecture

The next model to be fine-tuned is roBERTa (robustly optimized BERT approach), which is built on BERT and modified on its hyperparameters and its objectives. RoBERTa has twelve layers and is trained on 125M parameters. The implementation and architecture is very similar to BERT. The basis of roBERTa lies in the assertion that BERT could perform

better if trained for longer with more data. RoBERTa proves to be an improvement from BERT, with four main training differences, roBERTa:

(1) dynamically changes the masking pattern during the training stage.

(2) removes the next sentence prediction objective of BERT.

(3) is trained with longer sequences and in bigger batches.

(4) is trained on more data for a longer time.

In order to create roBERTa, BERT models with the same configuration of $\text{BERT}_{BASE}$ were trained.

Firstly, roBERTa adjusts the way the MLM objective is approached. BERT uses static masking, masking tokens only once during pre-processing, so that the same masking patterns are used during every epoch of training. RoBERTa uses dynamic masking, which generates a masking pattern, changing the way tokens are masked each time the sequence is sent to the model.

Secondly, roBERTa proposes four other training methods other than the NSP objective the original BERT has.

- The *segment-pair and NSP* method retains NSP loss and uses a pair of segments as input. The input can have multiple natural sentences as long as the total combined length is less than 512 tokens.

- The *sentence-pair and NSP* method retains NSP loss and uses a pair of natural sentences as input. These sentences are sampled from separate parts of the same document, or two separate documents. These inputs tended to be much shorter than 512 tokens. Batch size is increased so that the total number of tokens is similar to the segment-pair method.

- The *full sentence* method does not retain NSP loss and uses inputs of full sentences sampled from one or more documents. Once one document is finished being sampled

from, a separator token in added, and inputs continue to be sampled from the next document. Inputs may cross document boundaries.

- The *doc-sentence* method does not retain NSP loss and samples its inputs in the same way as the full sentence method, except that it does not allow inputs to cross document boundaries. Batch size is dynamically increased, for inputs sampled near the end of documents which may be shorter than 512 tokens.

The results shows that models which used individual sentences under performed compared to other models, potentially due to their inability to learn long-range dependencies. Results also show that models removing NSP loss either matches or improves the performance of models including NSP loss on downstream tasks. Models which restrict input samples to a single document also perform better than models which allow input sentences from multiple different documents.

Thirdly, roBERTa is trained with larger batch sizes than BERT. BERT trains with batch size of 256 sequences. RoBERTa experiments with batch sizes of 2k and 8k sequences. Results show that larger batch sizes improve perplexity for the MLM objective. Larger batches are also easier to parallelize.

Lastly, roBERTa trains for a longer time with more data. BERT originally uses a character level BPE with a vocabulary size of 30k. A BPE (Byte-Pair Encoding) uses subword units, creating a hybrid between character and word level representations. RoBERTa is trained on a BPE with a vocabulary size of 40k. RoBERTa is trained for significantly longer than BERT, for 500k steps, as opposed to BERT's 100k steps.

In the context of this paper, roBERTa is expected to produce more accurate results than the base BERT model, however, since roBERTa does have a slightly higher number of parameters and a more complex training structure, it is expected to result in longer training times.

### 3.1.2 DistilBERT Architecture

DistilBERT (knowledge distillation on BERT) is a particularly valuable modified version of BERT for this paper, since it is a lighter and faster model. As the dataset used in this paper is very large, the experiment will experiment with a few modifications in order to adjust for high processing requirements. While BERT has 110 million parameters, distilBERT only has 66 million. DistilBERT is 60% faster than the base BERT model while retaining 95% of BERT's performance. DistilBERT accomplishes this through a process of distillation.

Distillation is a process which compresses a larger model into a smaller model, making the model computationally cheaper and faster to use. In the knowledge distillation process, the smaller model, referred to as the student, is taught to replicate the knowledge of larger model, called the teacher. In this case, the teacher, BERT, would be trained on its 110 million parameters, and then distill the knowledge to student distilBERT, which then ideally matches whatever output BERT produces.

The distillation process uses a different training data set from the one BERT is trained on as outlined in previous sections. This is called a transfer set. Both the teacher and student models are trained on the transfer set.

The teacher is trained to decrease cross entropy loss between the distribution of the actual training labels and the predicted labels, over a hard target. Hard targets refer one-hot encoded labels, represented by binary levels 0 and 1. The teacher model is able to generalize on information by giving the predicted class a high probability, and all other classes different low, but non-zero probabilities, such that the classes with slightly higher probabilities would be closer generalized toward the highest predicted class. During the distillation process, the student learns from the knowledge of the teacher to be able to train with a cross-entropy over soft targets. These soft targets use a distribution of probabilities over all classes, as a opposed to a single 0 or 1. This essentially smooths the labels, allowing more information to be passed to the student model, and allowing the student model to learn how to generalize

better. The training loss of the student can be described as:

$$L = -\sum_i t_i * log(s_i),$$

where $t_i$ is the teacher's estimated probability, and $s_i$ is the student's estimated probability. Using soft targets allow the model to hold much more knowledge during training.

A softmax-temperature is used to help generate the soft targets, defined by function:

$$p_i = \frac{exp(z_i/T)}{\sum_j exp(z_j/T)},$$

where $z_i$ is the model score for class $i$ and $z_j$ is the model score for class $j$. The temperature, $T$, controls the smoothness of the output distribution. The temperature is constant between the student and teacher during training. Using a higher $T$ produces a softer probability distribution. When $T \to 0$, the distribution is one-hot coded. When $T \to +\infty$, the distribution becomes uniform.

Finally, the model takes a linear combination of the distillation loss, the masked language modeling loss, and a cosine embedding loss. The cosine embedding loss helps to align the direction of the hidden state vectors from the student and teacher so that the student can more closely mimic the teacher's behavior.

Architecturally, BERT and distilBERT are very similar. There are three main differences. DistilBERT removes the token-type embeddings, the pooler, and reduces the layers of BERT in half. The token-type embeddings help BERT tell the end and beginning of new segments. DistilBERT relies solely on positional encodings for positional information. The pooler helps BERT to summarize an input sequence into a fixed-size representation, while distilBERT omits this step.

DistilBERT also uses training tricks from roBERTa such as dynamic masking and removing the NSP objective of BERT. Due to the distilled process, distilBERT is expected to see a slightly lower accuracy rate for the experiments in this paper, but it is expected to see a significantly faster processing time.

## 3.2 Fine-Tuning for Sentiment Analysis

This paper uses an Amazon Review dataset containing 4 million reviews. Each review has either a positive or negative label, with positive labels referring to reviews receiving a 4 or 5 star, and negative labels corresponding to reviews receiving a 1 or 2 star. Reviews with 3 stars are considered neutral and omitted from the dataset. The test dataset contains 400,000 reviews and the training dataset contains 3,600,000 million reviews.

### 3.2.1 Pre-Processing

The first step is to pre-process the text data to ensure uniformity within the text and reduce noise. Keeping only the most relevant grammatical information is advantageous toward maximizing the model's efficiency. The text is normalized by removing punctuation and special characters, extra white spaces, removing stop words, and expanding contractions. Since this paper focuses on the uncased versions of BERT and variations of BERT, the text is also all converted to lowercase. The case of the text data is unlikely to provide much extra relevant information toward the prediction of positive and negative values. The labels are then one-hot-encoded, where 1 corresponds to positive labels, and 0 corresponds to negative labels.

The full dataset is split into subsets of size 10K, 25K, 50K, and 100K. These are the datasets that will be used for initial tests and trainings due to computational power constraints on the size of the full dataset. Each subset is split such that there is an equal number of positive and negative labels. The data is shuffled and split into a training and test set of 90/10 proportions. The training data is then split into 90/10, using 10% of the data as a validation set.

### 3.2.2 Naive Bayes

A Naive Bayes algorithm will be used to create a baseline to compare the accuracy of the fine-tuned model to. Using the baseline model, it is possible to see how a transformer

model compares to other non-attention based learning algorithms, as well as to see how the fine-tuning of a pre-trained transformer model affects the performance of the model. The baseline model also acts as a minimum standard for model performance. If any of the fine-tuned models sees a decrease in performance from the benchmark model's performance, it is an indication of either a model that is not properly implemented, or a model that is not worth further pursuing. Throughout the paper, the metric that will be used for comparison and indication of performance is accuracy, the percentage of accurate target outputs.

The Naive Bayes classifier is chosen to create a baseline accuracy benchmark. This supervised learning algorithm is a good method to compare the transformer model to because of its simplicity, speed, and its suitability toward classification tasks. The Naive Bayes classifier uses the Bayes theorem to calculate conditional probability to predict which class an input belongs to. As the algorithm only needs to calculate this probability, it is simple to implement and quick to process.

The Multinomial Naive Bayes classifier is used as the benchmark, which calculates the posterior probability that each element belongs to each class. For Multinomial Naive Bayes, the TF-IDF vectorization is used to reflect how important a term is to the document it is found in. TF-IDF is calculated as:

$$TFIDF = TF \times IDF,$$

where

$$TF = \frac{number\ of\ times\ term\ appears\ in\ document}{total\ number\ of\ words\ in\ document}$$

and

$$IDF = log(\frac{number\ of\ documents\ in\ corpus}{number\ of\ documents\ in\ corpus\ containing\ term})$$

The TF-IDF takes the log of the IDF score to lessen the effect of the score, preventing it from exploding when the number of documents in the corpus containing a certain term is very low.

The Multinomial Naive Bayes classifier is trained on the data subset of size 25K and achieves an 88% accuracy rate. Figure 3.1 shows the AUC-ROC (Area Under The Curve-Receiving Operating Characteristics) curve.

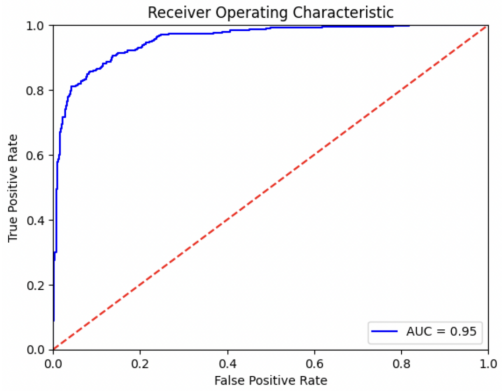


Figure 3.1: AUC-ROC curve for Multinomial Naive Bayes classifier.

The AUC-ROC graph plots the true positive vs. false positive rate. The Multinomial Naive Bayes classifier seems to predict both of the classes, positive and negative, with a similar accuracy. The Multinomial Naive Bayes classifier sees a relatively high accuracy in predicting the sentiment classes.

### 3.2.3 Implementation

The fine-tuning of BERT uncased begins by loading the tokenizer and tokenizing text. Each BERT model has its own unique tokenizer. This ensures the data is able to be processed by the particular model. During the tokenizer step, an input of the text is received, and an output of two tensors, multidimensional arrays, are created. One contains input IDs and the other contains attention masks. The text is first tokenized according to WordPiece tokenizations, and then a <CLS> and a <SEP> token is added to the beginning and end of each sentence, respectively.

The maximum token length of the entire Amazon dataset is 444 tokens. Since BERT requires a fixed input length, the sequences are padded to this length. The maximum length

that BERT can process is 512 tokens, if any sequence were to exceed this length, they would need to be truncated.

The tokens are then mapped to their IDs, a unique integer from BERT's predefined vocabulary, converting the input text to a sequence of integers. Binary attention masks are also created, where a value of 1 corresponds to tokens that contain content and will be attended to, and 0 corresponds to padding tokens that will be ignored.

Next, a custom BERT classifier is created. The one layer feed forward classifer is incorporated into the BERT model and will be trained along with the rest of the model during the fine-tuning process. This hidden layer contains four components. The classifier performs a linear transformation on the input, elementally wise applies a ReLU function to introduce non-linearity, adds an extra drop out layer to reduce overfitting, and finally linearly transforms the output again into the final logits vector. This goal of this added layer is to customize the model to the specific dataset as well as to the specific classification task of sentiment analysis.

The option to unfreeze the top layer of BERT during the fine-tuning process is also added. This is done by not requiring the parameters of BERT to calculate gradients. By doing so, the weights and parameters of the layer of the pre-trained model can also be updated when training for sentiment analysis. Since lower layers of the model are mostly used to capture more generic tasks, it is usually non-essential to unfreeze and update those layers. However, top layers of BERT capture more task-specific information and may create an impact on fine-tuning performance. In this paper, the topmost layer is unfrozen and fine-tuned on.

An optimizer is also created, which will update parameters during fine tuning in order to minimize the loss function. The cross entropy loss function is chosen as choice of loss function. The formula for cross entropy loss for binary classification is given as:

$$H(p, q) = -\sum_{x \in class} p(x) log(q(x)),$$

where $p(x)$ represents the true probability distribution and $q(x)$ represents the probability distribution predicted by the model. The goal is to minimize the difference between the

predicted and true values of the model, as $H(p, q) = 0$ would indicate a perfect model.

AdamW (Adam with Weight Decay) with a learning rate of 5e-5 is used as the optimizer. AdamW finds the minimum of a loss function by tracking averages of the gradient and the square gradient to take larger steps toward the minimum when gradients change slowly, and smaller steps when gradients change quicker. AdamW also contains a weight decay component that penalizes large model weights to prevent overfitting. The learning weight determines the step size during optimization. A smaller learning rate results in a slower but more stable training process.

Similar to the Adam learning rate described in Chapter 2, a linear learning rate with warm-up scheduler is employed to adapt the learning rate while training. Thus, the learning rate will linearly increase during a warm-up phase, and then linearly decrease after the warm-up phase has ended. By doing so, the model is able to initially learn faster while there is more uncertainty in the model, and then decrease as the model gains more experience, thus lending to more stability in the later stages of learning.

Authors suggest that batch sizes of 16 or 32 be chosen for fine-tuning [DCL19]. Batch size determines the number of training samples the model processes before updating its parameters. Smaller batch sizes are required for large datasets in order to fit into GPU memory, however larger batch sizes may train faster due to parallelism. Batch size also correlates to learning rates, larger batch sizes can handle faster learning rates, due to their stability, and vice versa. Batch size and learning rates can vary quite a bit depending on dataset and task type.

The model is trained for two epochs, authors suggest two, three, or four training epochs. When a model finishes training through all its batches, it has completed one epoch of training. More complex models and datasets may need more epochs to train through to properly capture all model intricacies. It is not always necessary to have a rigid number of epochs to train for. There are many early stopping techniques that will train the model for as many epochs necessary to achieve a certain loss value, and then stop the training process.

A similar implementation is followed for roBERTa and distilBERT. In the DistilBERT model, segments are separated with the separation token, <SEP>, since DistilBERT does not have token type IDs and does not require a user to indicate which token belongs to which segment. Similarly, roBERTa segments do not have token type IDs and are separated with a separation token. DistilBERT also does not have an option to select input positions. All other implementation is kept same as BERT.

# CHAPTER 4

# Results

This chapter will explain the results of each model.

First, the pre-trained BERT model with no fine-tuning will be tested. Then BERT and roBERTa will be fine-tuned on various hyperparameters for the data subsets of the full dataset. Finally, the hyperparameters which display the highest accuracy will be tested on distilBERT. Since distilBERT takes a much faster approach than the previous two variations of BERT, it will be fine-tuned on the full dataset.

## 4.1 Large Dataset

The size of the full dataset (1.7GB) provides a few problems with computing power limits. This section briefly mentions the challenges and solutions taken to work around large datasets. All training is performed with NVIDIA GPU A100-SXM4-40GB or Tesla GPU V100-SXM2-16GB depending on availability.

First, all one-hot encoding label types are compressed from 64int to 8int to reduce memory usage. This results in a 1.4% memory usage reduction for both the training and test sets. There is a limit of 24 hours on the amount of time code may be processed. During the pre-processing process, data is cleaned and normalized in 40 batches of 100K to mitigate any time-out issues.

The initial testing with BERT is performed on various subset sizes of the entire dataset to understand and experiment with different hyperparameters. These data subsets, 10K, 25K, 50K, and 100K, are chosen such that average training time for two epochs fall around

5 minutes, 10 minutes, 30 minutes, and 45 minutes, respectively, for the BERT model.

It is helpful to observe how the hyperparameters interact and shift with each other, and how each adjusts as dataset size changes. Ideally the full dataset would be tested with all models. However, fine-tuning both BERT and roBERTa with the full set far exceeds the 24 hour processing limit. Thus, initial testing is done with the data subsets. Once a few high performing combinations of hyperparameters using the 100K data subset of BERT is found, they are tested on the data subsets using roBERTa. Then, the data subsets are trained on distilBERT, which is much faster than both previous models. The same subsets from roBERTa and BERT are used with distilBERT for comparison of accuracy. Finally, the entire full dataset is trained using distilBERT to look for a potential increase in accuracy.

## 4.2 Pre-Trained Transformer Model Results

### 4.2.1 BERT Results

The data is fine-tuned on the subsets of size 10K, 25K, 50K, and 100K, splitting the dataset into a 90/10 split of training and test data. Data is further split with a 90/10 split of training and validation data for each subset. The same exact training, validation, and test subsets are used for each variation of the experiments.

First, the BERT model with no fine-tuning is tested with the 25K data subset, using batch size of 32. This model achieves an accuracy rate of 79%. This is lower than the Naive Bayes benchmark of 88%. This is likely due to the nature of the two models. While Naive Bayes is structured to be a suitable model to work with classification tasks as is, the pre-trained BERT is designed to be further fine-tuned before performing downstream tasks such as sentiment analysis.

The hyperparameter tuning portion will focus mainly on three hyperparameters: drop out rate, batch size, and number of epochs. The model is fine-tuned without an extra drop out layer within the classifier layer, and then fine-tuned with drop out rates of 0.1, 0.25,

and 0.5. It is notable that removing the extra drop out layer or keeping the drop out layer, but setting drop out equal to 0, would produce the same results and can be thought of interchangeably. This paper will refer results to models that did not use an extra drop out layer as models using "None", since that is how the model was implemented. However, a model with the extra drop out layer with drop out rate set to 0, would have the same effect. Batch sizes of 16, 32, and 64 are chosen to experiment with. Since an adaptive optimizer is used, the learning rate does not need to be manually adjusted as batch size fluctuates.

Note that all of the choices of drop out rates are only experimented on the data subsets of 10K, 25K, and 50K. This is because of the processing limits on the dataset of size 100K. Since the necessity of higher drop out rates also correlates with the size of dataset, this is a suitable choice in factor to omit. As the size of the training set increases, the need to adjust drop out rate may diminish. Drop out rate addresses overfitting, particularly when there is less data available, so the size of a larger data may offset the concern for overfitting. Thus the drop out rate requirement for larger datasets is likely lower, and higher drop out rates will not be experimented with the 100K dataset. Subsequent hyperparameter tuning is performed on the 100K subsets with the drop out rates that are the most optimal for the 50K dataset.

For the subset of 10K datapoints, twelve total models are run, each with a different combination of batch size and drop out rate. Each model is trained for two epochs and the validation accuracy is recorded after each epoch of training. Finally the new model weights are used to predict on the test set, and accuracy results for the test set are recorded. Table 4.1 shows the results of the experiments for the subset of 10K datapoints in accuracy percentage. Note that the first column under Drop Out Rate, labeled "None" corresponds to the a model where an extra drop out layer was not added into the classifier layer that was added to the BERT model. Subsequent columns correspond to models where the extra drop out layer was added, with drop out weights of 0.1, 0.25, and 0.5, respectively.

**10K Results**

| Batch Size | | Drop Out Rate | | | | Test Average |
|---|---|---|---|---|---|---|
| | | None | 0.1 | 0.25 | 0.5 | |
| 16 | Epoch 1 | 91.34 | 92.11 | 91.29 | 90.90 | |
| | Epoch 2 | 91.67 | 92.58 | 92.61 | 91.67 | |
| | Test | 90.4 | 89.9 | **91.5** | 89.2 | 90.25 |
| 32 | Epoch 1 | 90.95 | 90.39 | 91.75 | 90.09 | |
| | Epoch 2 | 91.92 | 91.72 | 92.33 | 91.39 | |
| | Test | 90.5 | 90.2 | 90.7 | 90.4 | 90.45 |
| 64 | Epoch 1 | 91.67 | 91.25 | 91.23 | 90.52 | |
| | Epoch 2 | 92.50 | 92.08 | 92.12 | 91.98 | |
| | Test | 90.8 | 90.1 | 90.2 | 91.2 | 90.58 |
| Test Average | | 90.57 | 90.07 | 90.80 | 90.27 | 90.42 |

Table 4.1: Results of 10K data subset for each model in percentage of accuracy, with test accuracy of highest performing model in bold.

The total average accuracy using the 10K dataset is 90.42%. The model yielding the highest accuracy is the model which uses a drop out rate of 0.25 and batch size of 16, at 91.5% accuracy on the test set. This compares to the model with the lowest accuracy at 89.2%, using a drop out rate of 0.5 and batch size of 16. The accuracy averages across each batch size and each drop out rate is also calculated.

Taking a closer look at the drop out rate and batch sizes, Figure 4.1 shows a graph of the accuracy for each drop out rate as it changes with batch size.

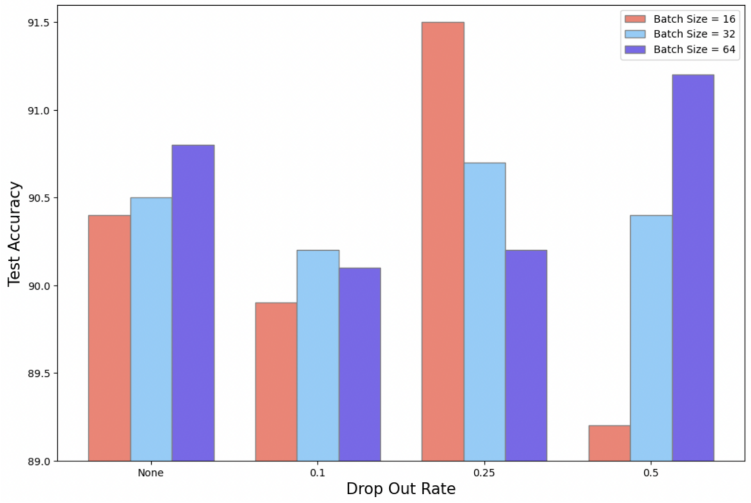


Figure 4.1: Test accuracy rates across batch sizes and drop out rates for data subset of size 10K.

There does not seem to be a consistent pattern in how batch size and drop out rate change together. As a whole, a drop out rate of 0.1 under performs or is equal to drop out rates of of 0.25 and none for all batch sizes. However, overall it seems that the use of the different hyperparameters, batch size and drop out rate, affects the model's overall performance without much consistent pattern. The highest performing model scores an 1.6% increase over the lowest performing model. This is a significant increase. As the current data subset of 10K does lean on the lower side of number of samples, the performance is likely to be more sensitive to hyperparameter fluctuations. On average, the highest performing batch size, 64, performs 0.35% better than the lowest performing batch size, 16. The highest performing drop out setting, 0.25, performs 0.73% better than the lowest, 0.1. The model likely needs a drop out rate of 0.25 to account for its smaller number of data. However, a drop out rate that is too large risks the model losing important information as too many nodes are dropped.

Next, the effect of the number of epochs the model is trained for on the accuracy of the model is studied. It is assumed that training for a second epoch will likely improve the performance of the model, but it is interesting to take a look at how greatly the second

epoch of training affects the accuracy rates. Table 4.2 shows the validation accuracy after one and two epochs of training, for each of the twelve models, numbered 1-12. It seems that by training for an extra epoch increases the accuracy by a substantial amount, with the largest increase across the 12 models in accuracy between epochs at 1.33% for the model using a batch size of 32 and a drop out rate of 0.1, and an average increase of 0.92%.

| | Drop Out Rate | | | | |
|---|---|---|---|---|---|
| Batch Size | None | 0.1 | 0.25 | 0.5 | Total Average |
| 16 | (1) 0.33 | (2) 0.47 | (3) 1.32 | (4) 0.77 | |
| 32 | (5) 0.97 | (6) 1.33 | (7) 0.58 | (8) 1.30 | |
| 64 | (9) 0.83 | (10) 0.83 | (11) 0.89 | (12) 1.46 | 0.92 |

Table 4.2: Difference in validation accuracy between epoch one and epoch two for data subset size of 10K.

Figure 4.2 visualizes the validation accuracy after epoch one and two, as well as test accuracy for each of the 12 models. For all models, the validation accuracy is after two epochs is higher than the validation accuracy after one epoch of training. It is commonly seen in practice that validation accuracy is higher than test accuracy, since the model may become more familiar with the validation set during training. For two of the models, 3 and 8, the test set accuracy is higher than the validation accuracy after training for one epoch. However, for all models, the validation accuracy after training for two epochs is the highest.

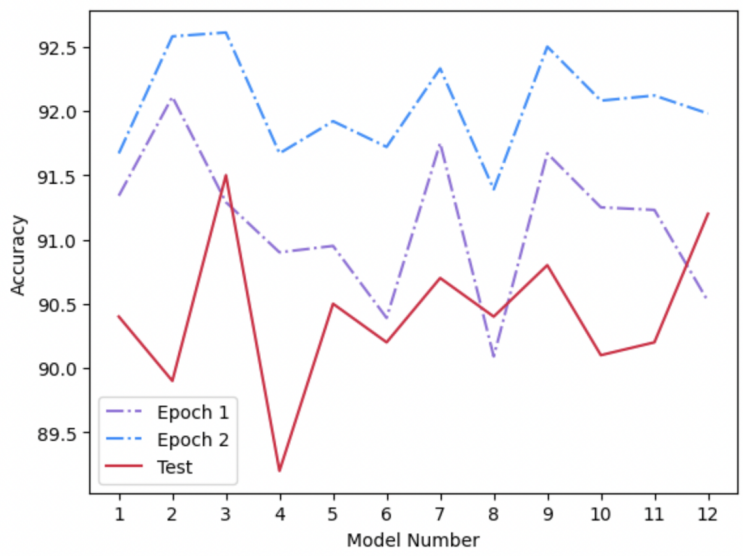


Figure 4.2: Validation accuracy after epoch one and two, and test accuracy for 10K data subset.

Next, all twelve models are fine-tuned on the data subset of 25K datapoints. Table 4.3 shows the results of the experiments.

The total average of using a 25K dataset size sees an increase of performance from the 10K set by 1.08%. The highest performing model achieves a 91.80% accuracy, while the lowest performing model achieves a 91.24%. Here, there is less of a difference between the lowest performing model, batch size 32 with drop out rate 0.25, and the highest performing model, batch size 64 with drop out rate 0.5, at a 0.56% difference in performance than seen in the previous data subset results. However, this is still a sizeable difference in performance.

**25K Results**

| Batch Size | | Drop Out Rate | | | | Test Average |
|---|---|---|---|---|---|---|
| | | None | 0.1 | 0.25 | 0.5 | |
| 16 | Epoch 1 | 91.13 | 92.06 | 91.53 | 91.62 | |
| | Epoch 2 | 91.60 | 92.29 | 92.02 | 92.11 | |
| | Test | 91.36 | 91.44 | 91.60 | 91.52 | 91.48 |
| 32 | Epoch 1 | 91.77 | 90.98 | 91.73 | 92.08 | |
| | Epoch 2 | 92.34 | 92.17 | 92.61 | 92.56 | |
| | Test | 91.56 | 91.52 | 91.24 | 91.56 | 91.47 |
| 64 | Epoch 1 | 90.32 | 90.71 | 90.28 | 90.93 | |
| | Epoch 2 | 92.39 | 92.27 | 92.23 | 91.87 | |
| | Test | 91.48 | 91.56 | 91.32 | **91.80** | 91.54 |
| Test Average | | 91.47 | 91.51 | 91.39 | 91.63 | 91.50 |

Table 4.3: Results of 25K data subset for each model in percentage of accuracy, with test accuracy of highest performing model in bold.

Figure 4.3 shows the drop out rates and batch sizes in terms of accuracy. There does not seem to be too much of a pattern. This time, the batch size of 16 along with a drop out rate of 0.25 are the hyperparameters that yield the highest average accuracy. The performance between all models vary less in accuracy in this dataset subset than the previous one. The difference in average between the highest performing batch size, 64, and the lowest performing batch size, 32, is only 0.06%. The change in accuracy from batch size 32 to 16 is only a difference of 0.01%. drop out rate sees a little more variation, with the highest average performing drop out rate of 0.5 performing 0.24% higher than the lowest performing drop out rate of 0.25.

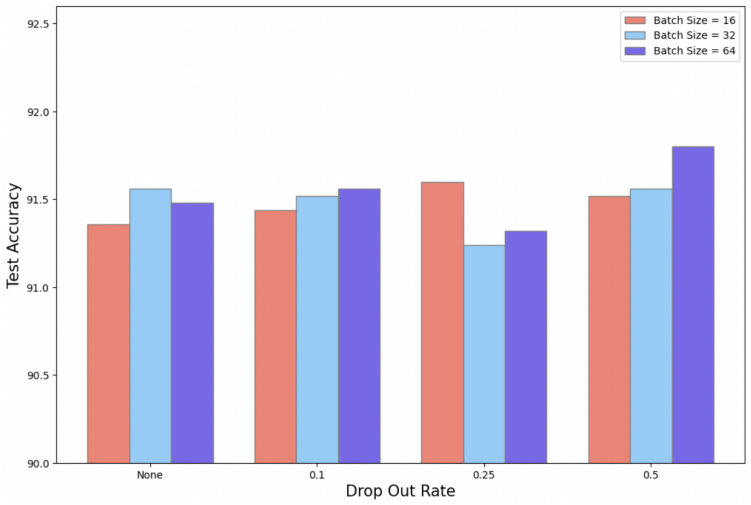


Figure 4.3: Test accuracy rates across batch sizes and drop out rates for data subset of size 25K.

Table 4.4 shows the difference in accuracy between training epochs one and two. On average, training for an extra epoch achieves a higher validation accuracy of 1.01%. This is considerable amount of increase, and also slightly higher than the results from previous data subset. For a batch size of 64, training for an extra epoch shows the greatest increase. For this dataset, it seems that for all models except one, the larger the batch size, the greater the positive impact of training for a second epoch is.

| | Drop Out Rate | | | | |
|---|---|---|---|---|---|
| Batch Size | None | 0.1 | 0.25 | 0.5 | Total Average |
| 16 | (1) 0.47 | (2) 1.23 | (3) 0.29 | (4) 0.49 | |
| 32 | (5) 0.57 | (6) 1.19 | (7) 0.88 | (8) 0.50 | |
| 64 | (9) 2.07 | (10) 1.56 | (11) 1.95 | (12) 0.94 | 1.01 |

Table 4.4: Difference in validation accuracy between epoch one and epoch two for data subset size of 25K.

Figure 4.4 visualises difference between each model's accuracy rates. Here, the test accuracy is always lower than the validation accuracy after epoch two, but model performance on the validation test after training for one epoch, and test accuracy vary. It seems that for this subset, training for an extra epoch contributes greatly to test accuracy performance on average.

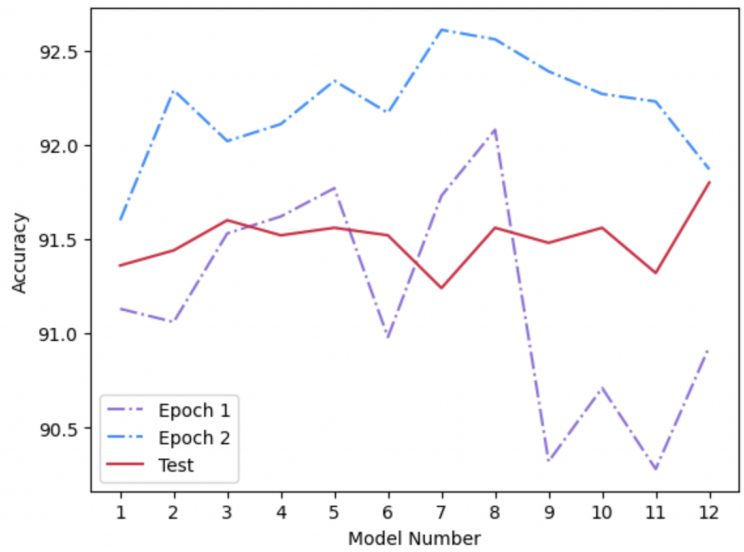


Figure 4.4: Validation accuracy after epoch one and two, and test accuracy for 25K data subset.

Table 4.5 shows accuracy rates for the 50K data subset. The total average performance for this subset is 93.21%. The top performing model uses batch size 64 with no extra drop out layer. As training dataset size increases, it is expected that overfitting is less apparent, and an extra drop out layer may not be necessary. The lowest performing model, one using batch 16 and drop out rate of 0.25 performs at 0.68% less in accuracy than the top model. On average, a batch size of 16 and 32 perform at the same accuracy rate. Again, as the dataset size increases, the differences between accuracy rates between models seem to decrease.

**50K Results**

| Batch Size | | Drop Out Rate | | | | Test Average |
|---|---|---|---|---|---|---|
| | | None | 0.1 | 0.25 | 0.5 | |
| 16 | Epoch 1 | 92.82 | 92.66 | 91.82 | 92.60 | |
| | Epoch 2 | 93.26 | 93.55 | 93.31 | 93.22 | |
| | Test | 93.16 | 93.34 | 92.98 | 93.14 | 93.16 |
| 32 | Epoch 1 | 92.81 | 93.03 | 92.54 | 92.76 | |
| | Epoch 2 | 93.46 | 93.49 | 93.70 | 93.49 | |
| | Test | 93.12 | 93.14 | 93.16 | 93.22 | 93.16 |
| 64 | Epoch 1 | 92.41 | 93.06 | 92.93 | 92.84 | |
| | Epoch 2 | 93.09 | 93.26 | 93.35 | 93.24 | |
| | Test | **93.66** | 93.16 | 93.24 | 93.22 | 93.32 |
| Test Average | | 93.31 | 93.21 | 93.13 | 92.19 | 93.21 |

Table 4.5: Results of 50K data subset for each model in percentage of accuracy, with test accuracy of highest performing model in bold.

Figure 4.5 shows the model test accuracy rates across different drop out rates and batch sizes. The differences between the groups do not seem to have an apparent pattern, however accuracy rates for the groups of drop out rate 0.1 and 0.5 seem to have very little differences between accuracy within the groups. The accuracy of batch size 32 seems to be most consistent across different drop out rates.

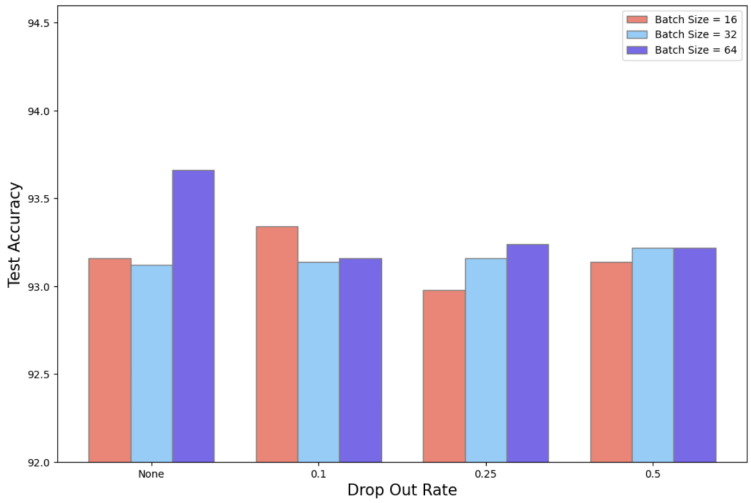


Figure 4.5: Test accuracy rates across batch sizes and drop out rates for data subset of size 50K.

The differences in training accuracy between epoch one and two are analysed. Table 4.6 shows the differences. Again, there is an increase in accuracy between epoch one and two. As dataset size increases, the need to train for an extra epoch seems to decrease, as the total average in increase between epochs is still high, but lower than the 25K subset on average.

| | Drop Out Rate | | | | |
|---|---|---|---|---|---|
| Batch Size | None | 0.1 | 0.25 | 0.5 | Total Average |
| 16 | (1) 0.44 | (2) 0.89 | (3) 1.49 | (4) 0.62 | |
| 32 | (5) 0.65 | (6) 0.46 | (7) 1.16 | (8) 0.73 | |
| 64 | (9) 0.68 | (10) 0.20 | (11) 0.42 | (12) 0.40 | 0.68 |

Table 4.6: Difference in validation accuracy between epoch one and epoch two for data subset size of 50K.

Figure 4.6 shows the accuracy plots for all twelve models. It is notable that for model nine, the test accuracy is higher than the validation accuracy after both epochs of training.

As the data subset gets larger, validation accuracy after two epochs of training gets closer to overall test accuracy. The test accuracy is also higher than the validation accuracy after training for one epoch for all models.

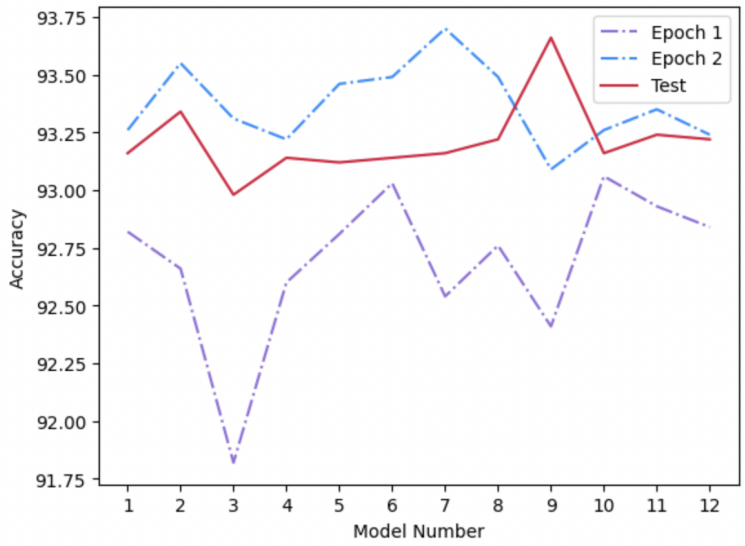


Figure 4.6: Validation accuracy after epoch one and two, and test accuracy for 50K data subset.

From the subsets of 25K and 50K, it is clear that average test accuracy is the same or very close to the same for batch sizes of 32 and 64. It is notable that for the subset of 50K samples, the models without the extra drop out layers perform slightly better. Due to computational limits, only four variations of the models are trained using the 100K dataset. The experiment includes the variations of models using no extra drop out layer and drop out rate of 0.1. The experiment also only uses batch sizes of 32 and 64. Table 4.7 shows the results from the models.

**100K Results**

| Batch Size | | No Drop Out | 0.1 | Test Average |
|---|---|---|---|---|
| 32 | Epoch 1 | 92.83 | 92.99 | |
| | Epoch 2 | 93.26 | 93.12 | |
| | Test | 93.71 | **94.05** | 93.88 |
| 64 | Epoch 1 | 93.19 | 93.13 | |
| | Epoch 2 | 93.12 | 93.06 | |
| | Test | 93.99 | 94.01 | 94.00 |
| Test Average | | 93.85 | 94.03 | 93.94 |

Table 4.7: Results of 100K data subset for each model in percentage of accuracy, with test accuracy of highest performing model in bold.

For a batch size of 64 and drop out rate of 0.1, the model has the highest overall test performance of all models so far, at an accuracy of 94.05%. Here, both models with drop out rate of 0.1 perform better than models with no extra drop out layer. Overall, a batch size of 64 performs better than a batch size of 32. For all data subsets, except for the 10K subset, the highest performing model uses a batch size of 64.

Figure 4.7 shows the accuracy rates with various batch sizes and drop out. For a drop out rate of 0.1, both a batch size of 32 and 64 perform at similar rates. The difference in performance in between batch sizes is more apparent within the none group of drop out rate.

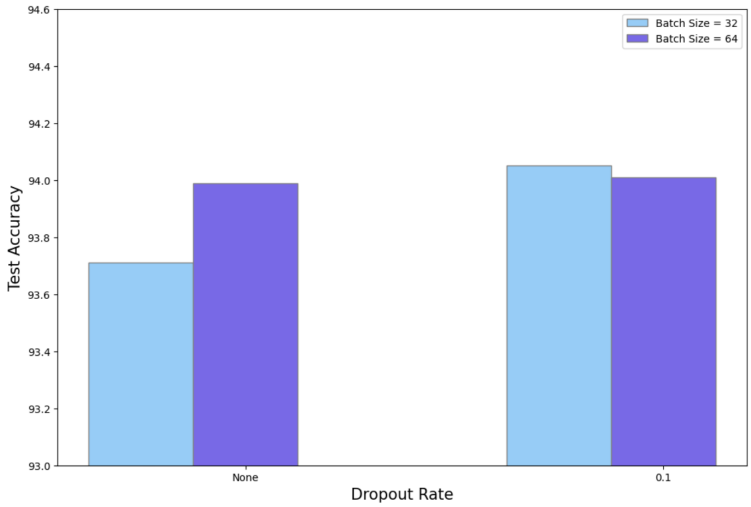


Figure 4.7: Test accuracy rates across batch sizes and drop out rates for data subset of size 100K.

Table 4.8 shows the difference in accuracy rates between epochs. Notably, there is a decrease in accuracy between epoch one and two for a batch size of 64. The decrease is slightly lower in the model with a drop out rate of 0.1. A decrease in accuracy between an epoch and a subsequent epoch can indicate the occurrence of overfitting. A drop out rate may help with this. However, despite the decrease in validation accuracy between epochs, these two models still perform well in test accuracy.

| Batch Size | No Drop Out | 0.1 | Total Average |
|---|---|---|---|
| 32 | $^{(1)}$ 0.43 | $^{(2)}$ 0.13 | |
| 64 | $^{(3)}-0.07$ | $^{(4)}-0.05$ | 0.11 |

Table 4.8: Difference in validation accuracy between epoch one and epoch two for data subset size of 100K.

Figure 4.8 shows the accuracy rates for all the models. It is notable that for this data subset, for every model, the test accuracy is higher than validation accuracy after both

models. In general, validation accuracy tends to be higher than test accuracy. However, the test accuracy is higher by quite a significant amount for this subset.

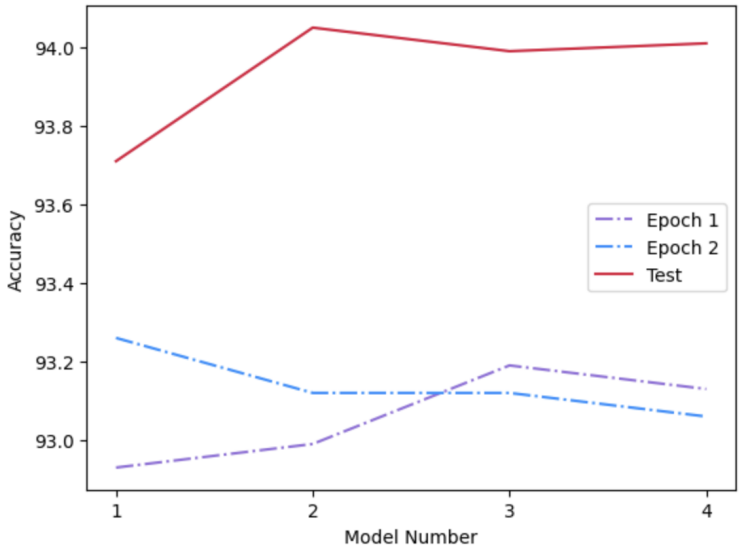


Figure 4.8: Validation accuracy after epoch one and two, and test accuracy for 100K data subset.

Despite the inconsistency in validation accuracy between epochs for batch size 64, both these models perform much better in their test set accuracy than validation accuracy for both epochs.

Overall, the subset of data of size 100K sees an average test accuracy rate of 93.94%. This is significantly more accurate than the models using a subset of 10K datapoints, at 90.24%. It is clear that the size of the dataset has more effect on the accuracy of the fine-tuning process than the particular hyperparameter tuning.

### 4.2.2 RoBERTa Results

RoBERTa is more computationally demanding than BERT, so only the two variations of models are fine-tuned, using the analysis of hyperparameters from the BERT results. Since a batch size of 64 is less computationally demanding and also performs the highest, on average from the larger data subsets on the BERT experiments, a batch size of 64 will be

used to fine-tune with roBERTa. Then a drop out rate of 0.1 and no extra drop out rate will be experimented with. Table 4.9 shows the results of the roBERTa models.

**RoBERTa 100K Results**

| Batch Size | | No Drop Out | 0.1 | Test Average |
|---|---|---|---|---|
| 64 | Epoch 1 | 92.40 | 93.30 | |
| | Epoch 2 | 93.27 | 93.59 | |
| | Test | 94.29 | 94.44 | 94.37 |

Table 4.9: Accuracy of roBERTa models on a data subset of size 100K.

RoBERTa sees a considerable increase in test accuracy from BERT. On average, roBERTa performs 0.37% better than the same models on BERT. RoBERTa performs best with a drop out rate of 0.1, outperforming the model without a drop out rate by 0.15%.

RoBERTa also seems to be more stable, as there is no issue with decrease in validation accuracy between epochs. It is expected that roBERTa outperforms BERT, due to its enhanced training regimen. Since roBERTa is trained for much longer and with more data, it performs better with the same amount of fine-tuning as BERT. RoBERTa sees considerable advantages to BERT, however due to its increase in number of parameters, as well as complexity to BERT, roBERTa is difficult to implement with limited compute powers. It may be advantageous to use a simpler model with more data.

### 4.2.3 DistilBERT Results

DistilBERT is the final model fine-tuned on. First, it is fine-tuned on a subset of 100K to explore if there is a considerable drop in accuracy between the same model variations using roBERTa and BERT. Then distilBERT is fine-tuned on the full dataset of 3.6 million training data, and tested on the 400K test set. Although training for an additional epoch

will double the training time, from previous results, training for an additional epoch also contributes to a considerable increase in accuracy. The full dataset will thus be trained for two epochs.

Table 4.10 shows the results from training distilBERT on a dataset of 100K with batch size of 64 with no drop out layer and a drop out layer with rate of 0.1. The better performing distilBERT model uses a batch size of 64 and no extra drop out layer.

**DistilBERT 100K Results**

| Batch Size | | No Drop Out | 0.1 | Test Average |
|---|---|---|---|---|
| 64 | Epoch 1 | 92.4 | 92.20 | |
| | Epoch 2 | 92.75 | 92.85 | |
| | Test | 93.6 | 93.45 | 93.53 |

Table 4.10: Accuracy of distilBERT models on a data subset of size 100K.

From the results of this experiment, DistilBERT trains at around 200% faster than BERT, and retains 99.6% of its accuracy, only underperforming the BERT version of this variation of model by 0.39%. DistilBERT is able to retain 99.1% of its accuracy from roBERTa, underperforming the roBERTa version by 0.68%. It is notable, however, that as the training dataset size increases, the speed at which distilBERT trains decreases per batch.

Figure 4.9 shows the side by side comparison of accuracy rates between the three models using a batch size of 64. The difference in accuracy between model 1 (no drop out layer) and model 2 (drop out layer with rate 0.1) vary between model. RoBERTa always performs the best, then BERT, and finally distilBERT.

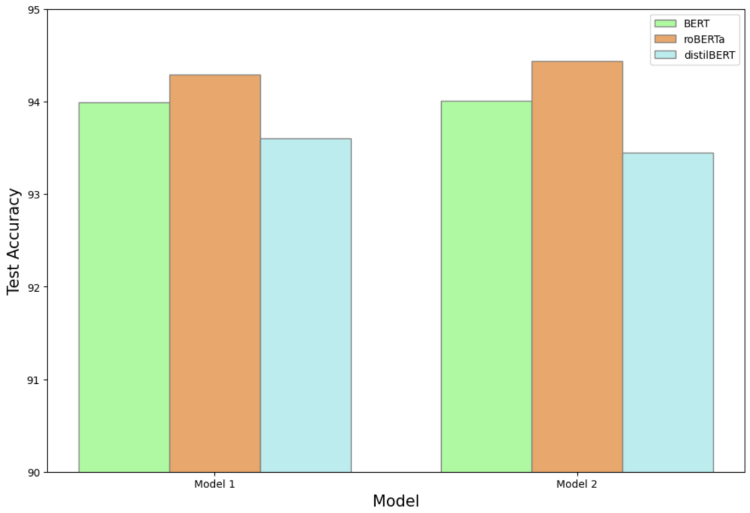


Figure 4.9: Test accuracy for model 1 and model 2 from BERT, roBERTa, and distilBERT.

Ultimately, for the distilBERT, excluding the extra drop out layer increases model performance by 0.15%, so the full data set will be fine-tuned on a model without the extra layer. The model is trained for one epoch, and the validation accuracy is recorded. The model is then evaluated on the test set and the result is recorded. The model is then trained for a subsequent epoch and finally evaluated on the test set. The entire process runs for around 18 hours on GPU.

Table 4.11 shows the results of the full dataset.

**Full Dataset Results**

| Batch Size | | No Drop Out |
|---|---|---|
| 64 | Epoch 1 | 95.52 |
| | Test | 95.45 |
| | Epoch 2 | 95.77 |
| | Test | 95.76 |

Table 4.11: Results of fine-tuning using distilBERT on full dataset.

The final accuracy of the model is 95.75%. By training the model for an extra epoch, an extra 0.25% in accuracy is achieved on the validation set and an extra 0.31% in accuracy is achieved on the test set. This is a significant increase from the rest of the models. The fine-tuning process on distilBERT on a full dataset in batches of 64 with no extra drop out layer performs the best.

It seems that the biggest factor toward increased performance is an increase in data. The large increase of data used in fine-tuning the distilBERT model offsets the decrease in performance due to its distillation process. It seems that although structure and training process to increase accuracy of a particular variation of model is important, the structure to increase the speed and processing power of a model is equally as important. Although distilBERT is not the model that displays overall highest performance given a set size of data, it surpasses the other models in its ability to efficiently train large amounts of data.

# CHAPTER 5

# Conclusion

In this paper, a pre-trained transformer model was fine-tuned on a downstream task in order to achieve a higher accuracy rates. In particular, BERT from Google AI was fine-tuned in order to perform sentiment analysis on a dataset of Amazon reviews. Parameter-efficient fine-tuning was employed in order to effectively fine-tune, by unfreezing and tuning only the topmost layer of the transformer. A variety of BERT-based transformer models, including $\text{BERT}_{BASE}$, $\text{roBERTa}_{BASE}$, and distilBERT were fine-tuned. Different hyperparameters for each model were further explored in order to identify an optimal model, one which yields the highest accuracy rates.

The Amazon reviews were organized as either positive or negative, based on the number of stars the review included. The dataset contained a large amount of data, requiring high memory usage. Thus, the full dataset was subset into smaller batches of 10K, 25K, 50K, and 100K in order to perform initial testing and exploration. Using a baseline Naive Bayes model, a baseline performance rate of 88% was established.

The first model that was fine-tuned was BERT. Twelve different variations of the BERT model were established, each with a different combination of hyperparameters drop out rate and batch size. The models were trained for two epochs on the subsets. BERT was able to achieve a peak accuracy of 94.05% on the data subset of 100K data points, with the model processing data in batches of size 32, and with an extra drop out layer with drop out rate of 0.1. Results showed that there was no distinct pattern in how batch size or drop out rate affected the test accuracy for the samples. Results did show that training the model for an extra epoch for the most part increased validation accuracy during training.

The next models that were fine-tuned were based on the variation of BERT, roBERTa. RoBERTa was fine-tuned on the data subset of 100K and found to achieve an impressive 94.44% test accuracy on the model which performed best, using a drop out rate of 0.1 and batch size of 64. RoBERTa was expected to perform higher than BERT, due to its advantageous adjustments made to BERT during its training. RoBERTa is trained on more data for a longer time than BERT, as well as applying different training objectives, which makes it higher-performing in comparison to BERT.

After roBERTa, the same subset of 100K data was fine-tuned using distilBERT. DistilBERT sees considerable advantages over both BERT and roBERTa due to its ability to train faster. Although there is a trade-off between roBERTa and distilBERT in performance in accuracy, distilBERT still displays competitive results, achieving an accuracy rate of 93.6% on its 100K data subset, using hyperparameters of batch size 64 and no drop out layer. DistilBERT is the only model which is able to fine-tune using the entire dataset for two epochs under computing limits ($<$ 24 hours processing time). Using this method, a test accuracy of 95.76% is achieved. Although hyperparameter tuning, as well as choice of model are important considerations toward performance, having more data to train on seems to be the most important factor.

Processing times and memory limits proved to be one of the predominant limitations faced throughout this paper. The size of the large dataset greatly contributed to the accuracy of the model, but also limited the types of models that were able to be implemented under the processing constraints. Using the full dataset and distilBERT provided impressive results. However, future research directions which would be able to carry out an experiment with the full dataset using more accurate models, such as roBERTa could see improved results.

It would also be interesting to explore other bidirectional models with the same dataset. Since the introduction of BERT, many other bidirectional models have followed in its footsteps and displayed competitive results. There are limitations which lie within the BERT and BERT variations as well. Further directions may explore other types of transformer and bidirectional models to experiment with. While BERT is shown to possess adequate syntac-

tic and semantic knowledge, BERT struggles reasoning when world knowledge is required [RKR21]. Altogether, BERT proves to be a valuable and suitable model for sentiment analysis, however there are other models which can be studied which may provide alternative types of benefit toward the problem.

Furthermore, it has been clear throughout the paper that more data contributes to better performance. It would be interesting to explore the point, if any, at which increased amounts of data only provide marginal increases in performance. Exploring other types of reviews from other platforms and analyzing their addition to the performance of the model would be on interest, as well.

# References


[AVM20] Adithya Avvaru, Sanath Vobilisetty, and Radhika Mamidi. "Detecting Sarcasm in Conversation Context Using Transformer-Based Models." In Beata Beigman Klebanov, Ekaterina Shutova, Patricia Lichtenstein, Smaranda Muresan, Chee Wee, Anna Feldman, and Debanjan Ghosh, editors, *Proceedings of the Second Workshop on Figurative Language Processing*, pp. 98–103, Online, July 2020. Association for Computational Linguistics.

[BCB16] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. "Neural Machine Translation by Jointly Learning to Align and Translate.", 2016.

[Bit] Adam Bittlingmayer. "Amazon Reviews for Sentiment Analysis." https://www.kaggle.com/datasets/bittlingmayer/amazonreviews/data.

[DC21] Ashok Kumar Durairaj and Anandan Chinnalagu. "Transformer based Contextual Model for Sentiment Analysis of Customer Reviews: A Fine-tuned BERT." 2021.

[DCL19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.", 2019.

[DT21] D. Deepa and A. Tamilarasi. "Bidirectional Encoder Representations from Transformers (BERT) Language Model for Sentiment Analysis task: Review." *Turkish Journal of Computer and Mathematics Education*, **12**(7):1708–1721, 2021. Copyright - © 2021. This work is published under https://creativecommons.org/licenses/by/4.0 (the "License"). Notwithstanding the ProQuest Terms and Conditions, you may use this content in accordance with the terms of the License; Last updated - 2023-12-05.

[GC] Swathi Gangaraju and Andrew Cheng. "Fine-Tuning BERT for Sentiment Analysis, Paraphrase Detection and Semantic Text Similarity NLP Tasks.".

[Gun23] Karthick Prasad Gunasekaran. "Exploring Sentiment Analysis Techniques in Natural Language Processing: A Comprehensive Review.", 05 2023.

[HG23] Dan Hendrycks and Kevin Gimpel. "Gaussian Error Linear Units (GELUs).", 2023.

[HSW21] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. "LoRA: Low-Rank Adaptation of Large Language Models.", 2021.

[KCH19] Shigeki Karita, Nanxin Chen, Tomoki Hayashi, Takaaki Hori, Hirofumi Inaguma, Ziyan Jiang, Masao Someki, Nelson Enrique Yalta Soplin, Ryuichi Yamamoto, Xiaofei Wang, Shinji Watanabe, Takenori Yoshimura, and Wangyou Zhang. "A

Comparative Study on Transformer vs RNN in Speech Applications." In *2019 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU)*. IEEE, December 2019.

[KNH22] Salman Khan, Muzammal Naseer, Munawar Hayat, Syed Waqas Zamir, Fahad Shahbaz Khan, and Mubarak Shah. "Transformers in Vision: A Survey." *ACM Computing Surveys*, **54**(10s):1–41, January 2022.

[KRG23] David M. Knigge, David W. Romero, Albert Gu, Efstratios Gavves, Erik J. Bekkers, Jakub M. Tomczak, Mark Hoogendoorn, and Jan-Jakob Sonke. "Modelling Long Range Dependencies in $N$D: From Task-Specific to a General Purpose CNN.", 2023.

[KRR19] Olga Kovaleva, Alexey Romanov, Anna Rogers, and Anna Rumshisky. "Revealing the Dark Secrets of BERT.", 2019.

[LOG19] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. "RoBERTa: A Robustly Optimized BERT Pretraining Approach.", 2019.

[LYL23] Kai Lv, Yuqing Yang, Tengxiao Liu, Qinghui Gao, Qipeng Guo, and Xipeng Qiu. "Full Parameter Fine-tuning for Large Language Models with Limited Resources.", 2023.

[Nar21] Gagan Reddy Narayanaswamy. *Exploiting BERT and RoBERTa to Improve Performance for Aspect Based Sentiment Analysis*. PhD thesis, Technological University Dublin, 2021.

[ON15] Keiron O'Shea and Ryan Nash. "An Introduction to Convolutional Neural Networks.", 2015.

[QSX20] XiPeng Qiu, TianXiang Sun, YiGe Xu, YunFan Shao, Ning Dai, and XuanJing Huang. "Pre-trained models for natural language processing: A survey." *Science China Technological Sciences*, **63**(10):1872–1897, September 2020.

[RKR21] Anna Rogers, Olga Kovaleva, and Anna Rumshisky. "A Primer in BERTology: What We Know About How BERT Works." *Transactions of the Association for Computational Linguistics*, **8**:842–866, 01 2021.

[Rud18] Sebastian Ruder. "A Review of the Neural History of Natural Language Processing." http://ruder.io/a-review-of-the-recent-history-of-nlp/, 2018.

[Saz06] Murat Sazli. "A brief review of feed-forward neural networks." *Communications Faculty Of Science University of Ankara*, **50**:11–17, 01 2006.

[Sch19] Robin M. Schmidt. "Recurrent Neural Networks (RNNs): A gentle Introduction and Overview.", 2019.

[SDC20] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. "DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter.", 2020.

[SQX20] Chi Sun, Xipeng Qiu, Yige Xu, and Xuanjing Huang. "How to Fine-Tune BERT for Text Classification?", 2020.

[Tra] Chris Tran. "Tutorial: Fine tuning BERT for Sentiment Analysis." https://skimai.com/fine-tuning-bert-for-sentiment-analysis/. Accessed: 2024-02-26.

[TUR50] A. M. TURING. "I.—COMPUTING MACHINERY AND INTELLIGENCE." *Mind*, **LIX**(236):433–460, 10 1950.

[VSP17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. "Attention Is All You Need.", 2017.

[WDS20] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. "HuggingFace's Transformers: State-of-the-art Natural Language Processing.", 2020.

[XXQ23] Lingling Xu, Haoran Xie, Si-Zhao Joe Qin, Xiaohui Tao, and Fu Lee Wang. "Parameter-Efficient Fine-Tuning Methods for Pretrained Language Models: A Critical Review and Assessment.", 2023.

[VSP17] [QSX20] [DCL19] [GC] [SQX20] [AVM20] [SDC20] [LOG19] [HG23] [BCB16] [HSW21]

[XXQ23] [RKR21] [KRR19] [Tra] [Bit] [WDS20] [TUR50] [Sch19] [Saz06] [KCH19] [Nar21]

[DT21] [DC21] [LYL23] [KNH22] [Gun23] [ON15] [KRG23] [Rud18]