TUM School of Computation, Information and Technology - Informatics
Technical University of Munich

Master’s Thesis in Informatics

A Comparison between Language Models with Transfer Learning on Medical Devices Manuals

Ein Vergleich zwischen Sprachmodellen mit Transfer Learning auf Handbüchern für medizinische Geräte

Supervisor	Prof. Dr.-Ing. habil. Alois C. Knoll
Advisor	Mohammadhossein Malmir, M.Sc.
Author	Seyed Navid Mirnouri Langeroudi
Date	June 15, 2023 in Munich

Hello world

Disclaimer

I confirm that this Master's Thesis is my own work and I have documented all sources and material used.

Munich, June 15, 2023

(Seyed Navid Mirnouri Langeroudi)

Abstract

This project is a study of a few state-of-the-art natural language processing models on a specific dataset. The thesis starts with a brief review of some classical supervised learning methods such as support vector machines and hidden Markov models, and the connections to transfer learning. In order to proceed with structural investigation of neural language models, first principles and backgrounds are studied. Artificial neurons and neural networks are reviewed. How feedforward neural networks are created from neurons is studied. Then the important concept of recurrency in neural networks is introduced. Architectures of recurrent neural networks and long short-term memory models are reconsidered along with their weaknesses and strengths. Finally, the architecture of transformers is examined. The most important building blocks of transformers which are attention and self attention are investigated. The few models that are studied in this project are all based on transformers. The strength of the models are evaluated for a question answering task by training and validation on two distinct datasets: the publicaly available SQuAD dataset, and a customized medical SQuAD-like dataset created during this thesis. The experiments show that transfer learning could be helpful to reduce the finetuning time.

Zusammenfassung

Bei diesem Projekt handelt es sich um eine Untersuchung einiger moderner Modelle zur Verarbeitung natürlicher Sprache auf einem spezifischen Datensatz. Die Arbeit beginnt mit einem kurzen Überblick über einige klassische überwachte Lernmethoden wie Support-Vektor-Maschinen und Hidden-Markov-Modelle und die Verbindungen zum Transfer-Lernen. Um mit der strukturellen Untersuchung von neuronalen Sprachmodellen fortzufahren, werden erste Prinzipien und Hintergründe untersucht. Künstliche Neuronen und neuronale Netze werden besprochen. Es wird untersucht, wie neuronale Feedforward-Netzwerke aus Neuronen entstehen. Dann wird das wichtige Konzept der Rekursion in neuronalen Netzen eingeführt. Die Architekturen rekurrenter neuronaler Netze und Modelle des Langzeitgedächtnisses werden mit ihren Schwächen und Stärken erörtert. Schließlich wird die Architektur von Transformatoren untersucht. Die wichtigsten Bausteine von Transformatoren, nämlich Aufmerksamkeit und Selbstaufmerksamkeit, werden untersucht. Die wenigen Modelle, die in diesem Projekt untersucht werden, basieren alle auf Transformern. Die Stärke der Modelle wird für eine Fragebeantwortungsaufgabe durch Training und Validierung auf zwei verschiedenen Datensätzen evaluiert: dem öffentlich zugänglichen SQuAD-Datensatz und einem angepassten medizinischen SQuAD-ähnlichen Datensatz, der im Rahmen dieser Arbeit erstellt wurde. Die Experimente zeigen, dass Transfer-Lernen hilfreich sein kann, um die Zeit für das Finetuning zu reduzieren.

Contents

Hello world

Chapter 1

Introduction

1.1 Outline

Natural Language Processing (NLP) has been an active field of research for decades. We could categorize NLP techniques in two category: old-fashioned NLP and Modern NLP. While classic approaches use techniques such as Regular expression, modern NLP lies around mainly on deep learning and machine learning and shows promising results. New-born models such as BERT [1] and GPT [2] are using deep learning to process large amount of data, represented as text. These large models have huge amount of trainable parameters and are being trained on a large amount of data from the internet. One technique they are using is called *transfer learning*. *Transfer Learning* is a technique that trains the model on huge amount of data and uses the trained model for smaller data sets.

Modern approaches in NLP mainly use *Transformers* for various tasks. Transformers' architecture and design can be found at [3]. In this work we are going to make a study on different state-of-the-art models in NLP, which are mainly based on transformers, on different tasks such as question answering (QA). For embedding there are lots of libraries such as word2vec [4]. Word2vec is a library developed by Google that uses two algorithm for embedding: continuous bag of words and continuous skip gram. Embedding is an algorithm which map a sentence into a vector of numbers. These numbers then are fed into the NLP model.

One of the main challenges in NLP is finding an apropriate dataset and annotate it. Although there are many extensively-curated existing datasets, creation of appropriate datasets for new domains ready enough to use for training is time consuming and expensive. In order to overcome this challenge we can use recent techniques such as self-supervised learning so we don't have to label the data manually.

This work is a part of Medical Device Operational Platform (MDOP) application. MDOP is an application for health system. It can be used by doctors and nurses in the hospital. They can scan a device in the app by its QR code. Then there are lots of material shown in the app regarding the scanned device including: manual of the device, quick guide, training material, chat functionality and etc. The aim of this work is to perform NLP on the manual of the medical devices. Tasks such as QA will be done in this project. On the other hand we are going to compare some of the state of the art models for different tasks such as QA on medical devices manuals. In the following sections, Transformers and their structure have been discussed. Then, an introduction to supervised learning and transfer learning has been stated. Finally, the introduction chapter finishes with an outlook over the next chapters and a summary.

1.2 Natural Language Processing

In the 1950s, the intersection of artificial intelligence and linguistics marked the beginning of NLP [5]. At that time, NLP and text information retrieval (IR) were two separate fields. IR used statistics-based techniques to index and search large volumes of text quickly and efficiently, as explained in [6] introductory paper on the topic [5]. Nevertheless, over time, NLP and IR have become more closely related [5]. Today's NLP researchers and developers draw from a wide range of disciplines, necessitating a significant expansion of their knowledge base [5]. Simple early approaches, such as Russian-to-English machine translation on a word-by-word basis, were unsuccessful due to homographs (words with multiple meanings) and metaphor, as evidenced by the infamous translation of the Biblical phrase "the spirit is willing, but the flesh is weak" as "the vodka is agreeable, but the meat is spoiled." [5].

Nadkarni et al. [5] provide a history and introduction of NLP, which a summary of it is stated in the following. Early investigations in 1956 by Chomsky's theoretical analysis of language grammars provided an assessment of the difficulty of the problem, which influenced the creation of Backus-Naur Form (BNF) notation in 1963. BNF is a means of specifying a Context-Free Grammar (CFG) and is frequently employed to represent the syntax of programming languages. A language's BNF specification consists of a collection of derivation rules that collectively serve to validate the syntax of program code. These rules are strict constraints, not heuristics employed by expert systems. In addition to identifying these rules, Chomsky also defined "regular" grammars that are even more restrictive and are the foundation of the regular expressions utilized to specify text-search patterns.

Based on [5], in 1956, Kleene defined regular expression syntax, which was first implemented in Ken Thompson's grep utility on UNIX. Later in the 1970s, lexical-analyzer (lexer) generators and parser generators like the lex/yacc combination utilized grammars. A lexer takes text and transforms it into tokens, while a parser validates a sequence of tokens. Lexer/parser generators simplify the implementation of programming languages significantly by taking as input regular expression and BNF specifications, respectively, and generating code and lookup tables that make lexing/parsing decisions.

According to [5], while CFGs are theoretically inadequate for natural language, they are commonly used in NLP practice. Programming languages are usually designed with a restrictive CFG variant, an LALR(1) grammar, which simplifies implementation. An LALR(1) parser scans text left-to-right, builds compound constructs from simpler ones, and employs a look-ahead of a single token to make parsing decisions.

The Prolog language was originally created in 1970 for NLP applications [5]. Its syntax is well-suited for writing grammars. However, in the simplest implementation mode (top-down parsing), rules must be phrased differently (i.e., right-recursively) than those intended for a yacc-style parser. Top-down parsers are easier to implement than bottom-up parsers (and do not require generators), but they are much slower [5].

When using standard parsing methods that rely purely on symbolic, hand-crafted rules, natural language's vast size, unrestrictive nature, and ambiguity lead to problems [5]. NLP must ultimately extract meaning (or "semantics") from text, but formal grammars that specify the relationship between text units (such as nouns, verbs, and adjectives) primarily address syntax. One can extend grammars to cover natural language semantics by greatly expanding sub-categorization with additional rules and constraints (such as "eat" only applying to ingestible-item nouns). However, the rules may now become unmanageably numerous and often interact unpredictably, resulting in more frequent ambiguous parses (multiple interpretations of a word sequence are possible). Handwritten rules are not effective at handling "ungrammatical" spoken prose and the highly telegraphic prose of in-hospital progress notes in medical contexts, even though these types of prose are comprehensible to humans [5].

In the 1980s, a significant shift occurred in NLP, which was characterized by Klein [7] as follows. The focus shifted to simple and robust approximations instead of deep analysis. Evaluation methods became more rigorous and probabilistic machine-learning methods became more prominent, despite initial skepticism from experts like Chomsky. Large, annotated corpora were used to train machine-learning algorithms and provided gold standards for evaluation. This change resulted in the emergence of statistical NLP. Statistical parsing, for instance, addresses the problem of parsing-rule proliferation through probabilistic CFGs. These grammars use probabilities associated with individual rules, which are determined through machine learning on annotated corpora. This approach results in fewer, broader rules that replace numerous detailed rules. Additionally, other approaches build probabilistic "rules" from annotated data, which are similar to machine-learning algorithms like C4.5. These approaches build decision trees from feature-vector data. Regardless of the method used, a statistical parser identifies the most likely parse of a sentence/phrase, which is context-dependent.

The effectiveness of statistical approaches in practice can be attributed to their ability to learn from vast amounts of real data [5]. The more representative and abundant the data, the better the results [5]. Statistical approaches also handle unfamiliar or erroneous input more gracefully. However, as this issue's articles demonstrate, handwritten-rule-based and statistical approaches are complementary [5].

Statistical and machine learning involve the creation or utilization of algorithms that enable a program to identify patterns in example data, allowing it to make predictions about new data through a process called generalization [5]. During the learning phase, the algorithm computes numerical parameters that characterize its underlying model by optimizing a numerical measure, usually through an iterative process [5].

Learning can be supervised, where each item in the training data is labeled with the correct answer, or unsupervised, where the learning process tries to recognize patterns automatically [5]. One issue with any learning approach is over-fitting, where the model fits the example data almost perfectly but makes poor predictions for new, previously unseen cases because it learns the random noise in the training data instead of its essential features [5]. Cross-validation is used to minimize over-fitting by partitioning the example data randomly into training and test sets to internally validate the model's predictions [5].

According to [5], machine-learning models can be classified as either generative or discriminative. Generative methods create rich models of probability distributions and are so-called because they can generate synthetic data. Whereas, discriminative methods estimate posterior probabilities directly based on observations. Logistic regression and conditional random fields (CRFs) are examples of discriminative methods, while Naive Bayes classifiers and hidden Markov models (HMMs) are examples of generative methods.

Several articles in this issue use common machine-learning methods in NLP tasks, including logistic regression, CRFs, Naive Bayes classifiers, and HMMs. In the following subsections, the relevance of two dominant classic approaches, support vector machines and hidden Markov models, are discussed.

1.2.1 Support Vector Machines

Support vector machines (SVMs) use a discriminative approach to classify inputs (such as words) into different categories (such as parts of speech) based on a set of features [5]. The input may be transformed mathematically using a "kernel function" to enable linear separation of data points from various categories [5]. In the simplest case with two features, a straight line would separate the data points on an X-Y plot. For N-features, an (N-1) hyperplane would separate the data. The most common kernel function used is a Gaussian function (which forms the basis of the normal distribution in statistics) [5]. During the separation process, a subset of the training data, called the "support vectors," is selected. These are data points closest to the hyperplane that best distinguish between the categories. The separating hyperplane maximizes the distance from the support vectors of each category [5].

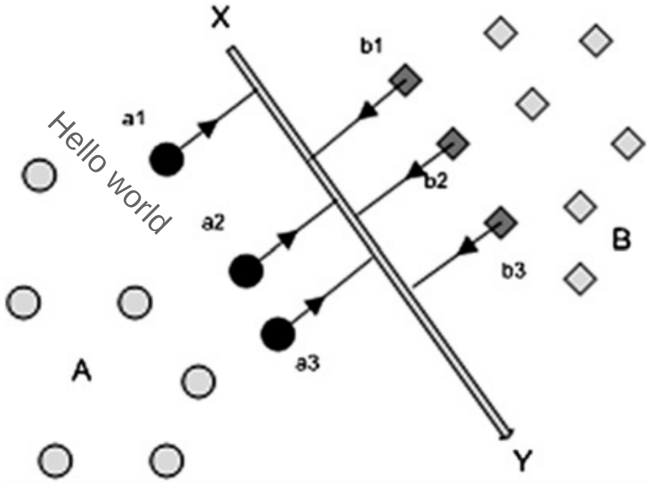


Figure 1.1: Illustration of a basic example of support vector machines [5]. The dataset contains two categories, A (represented by circles) and B (represented by diamonds), that can be separated by a straight line in a two-dimensional space (X-Y plane). The algorithm selects data points, known as "support vectors," from each category that are closest to the other category (a1, a2, a3, b1, b2, b3), and calculates the position of the line (X-Y plane) such that the margin separating the categories on both sides is maximized. In more complex, high-dimensional cases, the line is actually a hyperplane with a dimension of N-1, where N is the number of features in the dataset. Often, data must be transformed mathematically to achieve linear separation.

1.2.2 Hidden Markov Models

A Hidden Markov Model (HMM) is a system that permits a variable to transition, with varying probabilities, between multiple states, producing one of many possible output symbols with each transition, also with varying probabilities [5]. The number of possible states and unique symbols may be extensive but limited and familiar [5]. We can perceive the outputs, but the internal workings of the system (i.e., state-transition probabilities and output probabilities) are "hidden". As reported by [5], the following is the list of issues need to be addressed.

(A) Inference: Given a specific sequence of output symbols, compute the likelihoods of one or more candidate state-transition sequences.

(B) Pattern matching: Find the state-transition sequence most likely to have generated a particular output-symbol sequence.

(C) Training: Given examples of output-symbol sequence (training) data, determine the state-transition/output probabilities (i.e., system internals) that best suit this data.

Naive Bayesian reasoning, extended to sequences, is used to address problems B and C; hence, HMMs use a generative model. To address these problems, an HMM relies on two simplifying assumptions (which are accurate for numerous real-world phenomena). HMMs are a type of system where a variable can transition between different states, producing different output symbols with each transition, each with varying probabilities [5]. These states and symbols are finite and known [5]. The internal workings of the system, such as the probabilities of transitioning between states and generating output symbols, are hidden [5]. HMMs aim to solve problems of inference, pattern matching, and training [5]. To do this, they rely on two assumptions commonly found in real-life phenomena [5]. First, the probability of transitioning to a new state or returning to the same state depends on the previous N states. In a first-order HMM, the probability is based only on the current state [5]. Second, the probability of generating a specific output in a state depends only on that state [5]. These assumptions allow the probability of a state-transition sequence and corresponding output sequence to be calculated through simple multiplication of individual probabilities [5]. Various algorithms can solve these problems, such as the Viterbi algorithm, which is useful in signal processing and cell-phone technology [5]. Although theoretically, HMMs can be extended to multivariate scenarios, the training problem can become difficult to manage [5]. Therefore, multiple-variable applications of HMMs use single, artificial variables that require more training data [5].

HMMs are commonly used in speech recognition, where the waveform of a spoken word is matched to the sequence of phonemes that likely produced it [5]. HMMs also solve bioinformatics problems such as multiple sequence alignment and gene prediction [5]. Frederick Jelinek, an HMM pioneer at IBM's Speech Recognition Group, reportedly joked that "every time a linguist leaves my group, the speech recognizer's performance improves." [5]. For a bioinformatics-oriented introduction to HMMs, Eddy [8] provides a clear overview, while Rabiner's work [9] on speech recognition provides a more detailed introduction.

1.3 Supervised Learning

According to the Deep Learning book by LeCun et al. [10], supervised learning is a process or learning algorithm that associate some input to some output given a training set input x and output set of y. The very important thing in supervised learning is "loss". Loss calculate the error between prediction and given output. Loss function can have different formula, and in pattern recognition the following formula can be used [11].

$$L(f(x), y) = 0.5 * |f(x) - y|$$

Through this short introduction of the concept of supervised learning and its formal definition by important formula of loss and error, the section continues with adressing two research areas related to application of supervised learning in NLP. Firstly, we elaborate on how the Stanford Natural Language Inference (SNLI) task is utilized to train universal sentence encoding models via the Natural Language Inference (NLI) task [12]. Next, we describe in details the investigated architectures for the sentence encoder, which encompasses an appropriate range of sentence encoders currently in use [12]. These include conventional recurrent models like LSTMs and GRUs, with a focus on mean and max-pooling over hidden representations; a self-attentive network that integrates diverse sentence perspectives; and a hierarchical convolutional network that can be perceived as a tree-based technique that blends different levels of abstraction [12].

There are two distinct methods for training models on SNLI: (i) sentence encoding-based models that distinctly encode each sentence, and (ii) joint techniques that enable the use of encoding from both sentences (to utilize cross-features or attention from one sentence to another) [12].

According to [12], the objective is to train a universal sentence encoder; therefore, we choose to use the first approach. As depicted in 1.2, a typical architecture of this type employs a shared sentence encoder that generates a representation for both the premise u and the hypothesis v [12]. After creating the sentence vectors, three matching methods are applied to extract connections between u and v: (i) concatenation of the two representations (u, v); (ii) element-wise product $u * v$; and (iii) absolute element-wise difference $|u - v|$ [12]. The resulting vector, which encapsulates information from both the premise and the hypothesis, is then inputted into a 3-class classifier that comprises multiple fully-connected layers culminating in a softmax layer [12].

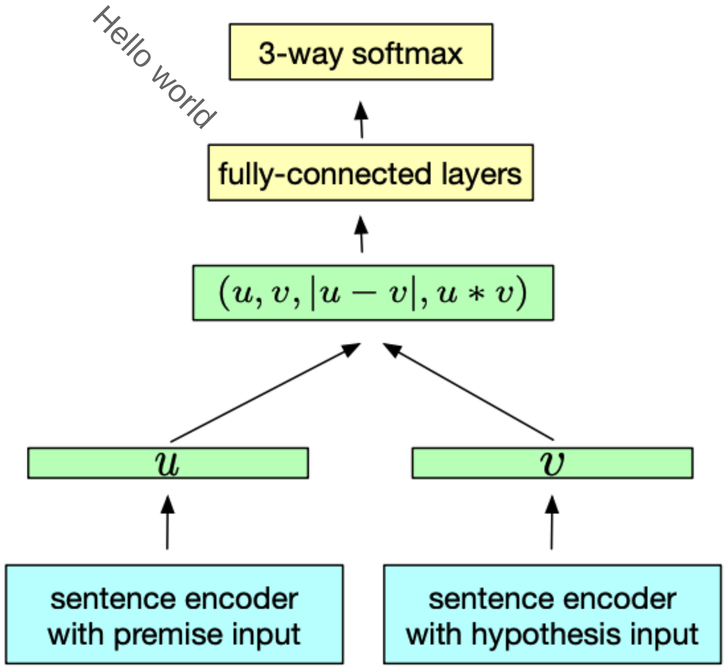


Figure 1.2: NLP schema and workflow [12]

1.4 Transfer Learning

Transfer learning is a technique used to enhance the learning of a learner in one domain by transferring knowledge from a related domain [13]. To illustrate this concept, consider the example of two individuals wanting to learn how to play the piano [13]. One person has no musical background while the other has extensive experience playing the guitar. The person with prior musical knowledge can transfer this knowledge to learning the piano, making the process more efficient. In essence, the individual can use previously acquired knowledge to learn a related task [13].

In the field of machine learning, transfer learning is often used when there is a limited supply of target training data, which could be due to the rarity, cost, or inaccessibility of the data. In such cases, transfer learning can be used to make use of existing datasets that are related to the target domain of interest [13]. For example, in the task of predicting text sentiment of product reviews, if there is an abundance of labeled data from digital camera reviews but limited data from food reviews, transfer learning can be used to improve the results of the target learner [13].

Transfer learning works by identifying the commonalities between the domains and transferring relevant knowledge from the source domain to the target domain [13]. The domains can be viewed as sub-domains of a higher-level common domain [13]. Successful applications of transfer learning include text sentiment classification, image classification, human activity classification, software defect classification, and multi-language text classification [13].

The domain D consists of two parts: a feature space X and a marginal probability distribution $P(X)$, where $x = \{x_1, ..., x_n\} \in X$. For example, if we take the machine learning application of software module defect classification, each software metric can be considered as a feature, where x_i is the i-th feature vector (instance) that corresponds to the i-th software module, n is the number of feature vectors in x, X is the space of all possible feature vectors, and x is a particular learning sample. A task T is defined by two parts: a label space Y and a predictive function $f(.)$, which is learned from the feature vector and label pairs x_i, y_i where $x_i \in X$ and $y_i \in Y$. Referring to the software module defect classification application, Y is the set of labels and contains true and false in this case, where y_i takes on a value of true or false, and $f(x)$ is the learner that predicts the label value for the software module x.

The domain is defined as $D = \{X, P(X)\}$ and the task as $T = \{Y, f(.)\}$. D_S is the source domain data and is defined as $D_S = \{x_{S_1}, y_{S_1}\}..., \{x_{S_n}, y_{S_n}\}$, where $x_{S_i} \in X_S$ is the i-th data instance of D_S and $y_{S_i} \in Y_S$ is the corresponding class label for x_{S_i}. D_T is the target domain data and is defined as $D_T = \{x_{T_1}, y_{T_1}\}..., \{x_{T_n}, y_{T_n}\}$, where $x_{T_i} \in X_T$ is the i-th data instance of D_T and $y_{T_i} \in Y_T$ is the corresponding class label for x_{T_i}. Furthermore, the source task is T_S, the target task is T_T, the source predictive function is $f_S(.)$, and the target predictive function is $f_T(.)$.

Transfer learning is the process of improving the target predictive function $f_T(.)$ by using related information from D_S and T_S, where $D_S \neq D_T$ or $T_S \neq T_T$. The condition $D_S \neq D_T$ means that $X_S \neq X_T$ and/or $P(X_S) \neq P(X_T)$. Heterogeneous transfer learning is when $X_S \neq X_T$, while homogeneous transfer learning is when $X_S = X_T$. The case $P(X_S) \neq P(X_T)$ means that the marginal distributions in the input spaces are different between the source and target domains. In a transfer learning environment, it is possible that $Y_S \neq Y_T$ and/or $P(Y_S|X_S) \neq P(Y_T|X_T)$. The case $P(Y_S|X_S) \neq P(Y_T|X_T)$ means that the conditional probability distributions between the source and target domains are different, while the case of $Y_S \neq Y_T$ refers to a mismatch in the class space.

An example of marginal distribution differences is when the source software program is written for a user interface system and the target software program is written for a DSP signaling decoder algorithm. Another possible condition of transfer learning is when $T_S \neq T_T$. In a transfer learning environment, it is possible that $Y_S \neq Y_T$ and/or $P(Y_S|X_S) \neq P(Y_T|X_T)$. The case where $P(Y_S) \neq P(Y_T)$ is caused by an unbalanced labeled data set between the source and target domains. In comparison, the case of traditional machine learning is when $D_S = D_T$ and $T_S = T_T$.

To illustrate distribution issues that can occur between the source and target domains, we can consider the example of natural language processing. In natural language processing, text instances are often modeled as a bag-of-words where a unique word represents a feature. Words that are generic and domain-independent should occur at a similar rate in both domains. However, words that are domain-specific are used more frequently in one domain because of the strong relationship

1.5 Summary

In this chapter, first the NLP was discussed and classic methods of it were investigated. Then supervised learning was discussed and some schema of supervised learning in NLP was introduced. Then there was a discussion about transfer learning. In the next chapter the background and related work will be presented. First, the structure of self attention and transformer will be discussed then other related works such as word embedding and neural network will be reviewed.

Chapter 2

Background and Related Works

In this chapter some background and related work are going to be discussed. In the first section neural networks and their structure are discussed. Then Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) will be discussed. After that transformers will be investigated. The chapter continues with introducing word embedding and language models and how transformers can behave as language models. A final discussion denotes which of the deep models are suitable for the question and answering task.

2.1 Feedforward Neural Networks

The building block of a neural network is a neuron, which simulates a simplified version of the biological neuron in human body. As a definition, a neuron value is weighted sum of its input plus a term called bias [14]

$$z = b + \Sigma w_i * x_i \ .$$

Since most of the time x and w are vectors, z would be the dot product of x and w

$$z = w.x + b \ .$$

Finally, to insert nonlinearity, an activation function is imposed on z. First of all, the sigmoid activation function is introduced for its simplicity and wide use

$$y = \sigma(z) = \frac{1}{1 + e^{-z}} \ .$$

The diagram for the sigmoid function and a neuron can be seen in 2.1 and 2.2 respectively.

Now that neuron has been introduced, it's time to talk about feedforward neural networks. Simple feedforward networks have three kinds of nodes: input units, hidden units, and output units [14]. The fundamental structure of a feedforward neural network can be seen in 2.3.

It is also worth mentioning that feedforward neural networks can be used for the task of classification [14]. The diagram for this can be seen in 2.4. As it can be seen in the picture there are three types of output (positive, negative, and neutral).

In the next section, the structure of recurrent neural networks is going to be discussed.

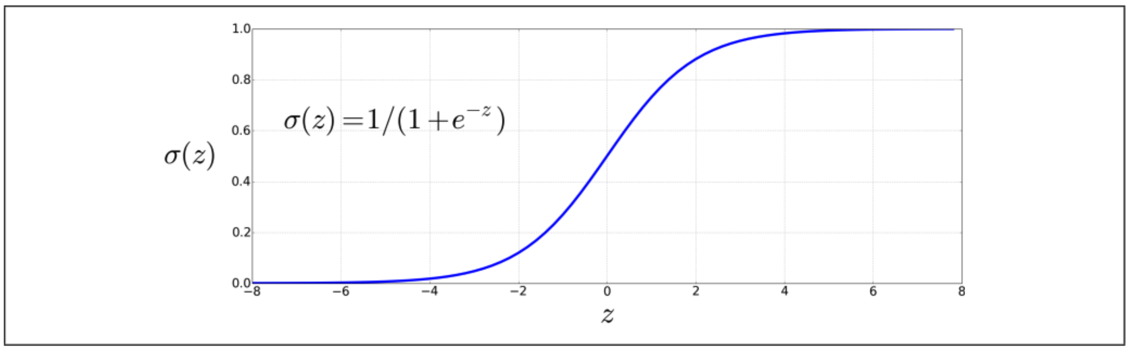


Figure 2.1: The diagram a sigmoid function [14]

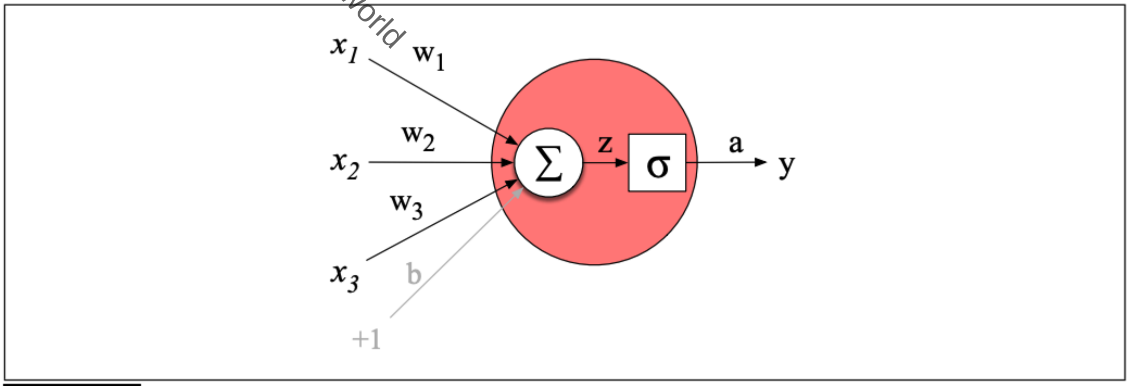


Figure 2.2: A neural unit receives three inputs: x_1, x_2, and x_3 (along with a bias b, which can be represented as a weight for an input fixed at $+1$), and produces an output y. To simplify computations, two intermediate variables are introduced: the sum output z and the sigmoid output a. However, in this case, the unit's output y is equivalent to a'. In more intricate networks, y is reserved to indicate the final output of the entire network, while a denotes the activation of an individual node. [14]

2.2 Recurrent Neural Networks

RNNs and LSTMs have important roles in sequence analysis. In 2.5 the simplest block of an RNN can be seen. As it is obvious there is a loop in the hidden layer and that is the fundamental part of an RNN.

Similar to feedforward networks, an input vector, x_t, representing the current input is multiplied by a weight matrix, and then processed by a non-linear activation function to compute the values for a hidden layer of units [14]. The hidden layer is utilized to calculate a corresponding output, y_t [14]. However, in contrast to the earlier window-based approaches, the RNN processes sequences one item at a time, where x_t represents the input vector at time t [14]. The recurrent link in the figure, represented by the dashed line, augments the computation at the hidden layer by incorporating the value of the hidden layer from the previous time step [14]. This link provides a memory or context that encodes earlier processing and guides decision-making at later time points [14]. Importantly, this approach does not impose a fixed-length limit on prior context, which can extend back to the beginning of the sequence [14].

Although adding this temporal dimension to RNNs makes them appear more complex than non-recurrent architectures, the underlying feedforward calculation remains unchanged [14]. Figure 2.6 illustrates how the recurrence factorizes into the computation at the hidden layer. The main difference is the introduction of a new set of weights, U, which connects the

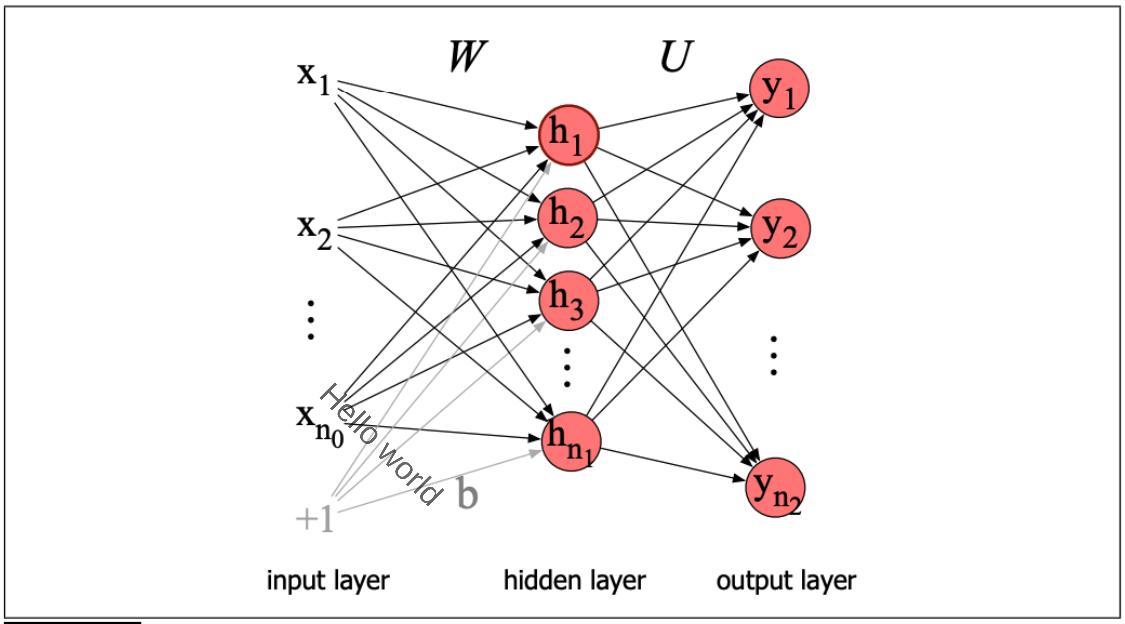


Figure 2.3: The diagram a feedforward neural network [14]

hidden layer from the previous time step to the current hidden layer [14]. These weights determine how past context is used in calculating the output for the current input [14].

Now that the structure of RNNs has been discussed, it's time to introduce LSTM. Long short-term memory architecture has some better features that can overcome the problem of long-term dependency vanishing in RNN. It adds some gate for forgetting previous history and to bypass the history to the next cell. The overall architecture of the LSTM cells can be seen in 2.7.

2.3 Transformers

Over the last few years, several transformer variants, also known as Xformers, have been proposed as a result of their success. These Xformers have improved the original transformer from various perspectives.

One of the main challenges of the transformer is its inefficiency at processing long sequences, primarily due to the computation and memory complexity of the self-attention module. To address this issue, improvement methods have been introduced, such as lightweight attention, including sparse attention variants, and divide-and-conquer methods, including recurrent and hierarchical mechanisms, which improve model efficiency.

As the transformer is a flexible architecture that makes few assumptions about the structural bias of input data, it is challenging to train on small-scale data. To improve model generalization, several methods have been introduced, including introducing structural bias or regularization, pre-training on large-scale unlabeled data, and more.

Another line of work focuses on adapting the transformer to specific downstream tasks and applications, which is known as model adaptation. In this review, the aim is to provide a comprehensive overview of the transformer and its variants. Although Xformers can be organized based on the aforementioned perspectives, many existing Xformers may address one or more issues. For instance, sparse attention variants not only reduce computational complexity but also introduce structural prior on input data to alleviate the overfitting problem on

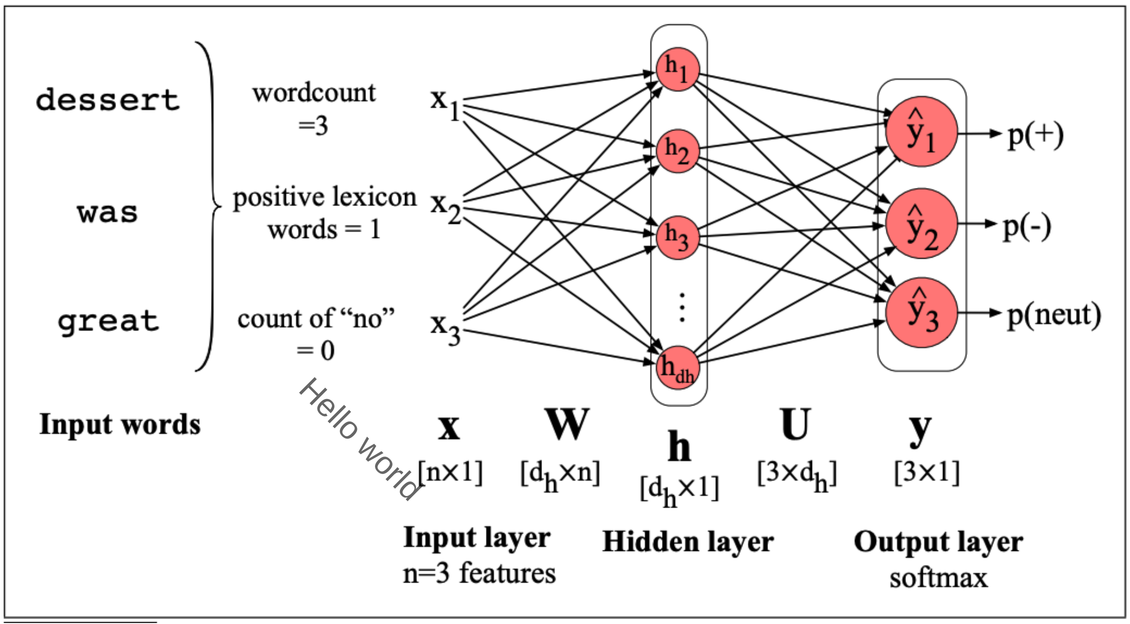


Figure 2.4: The diagram a feedforward neural network for classification [14]

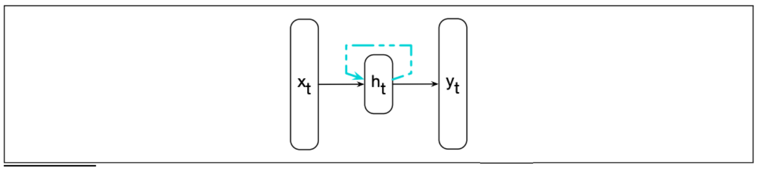


Figure 2.5: Building block of an RNN [14]

small datasets. Therefore, a new taxonomy is porposed that mainly categorizes the existing Xformers based on their ways of improving the vanilla transformer, including architecture modification, pre-training, and applications. The main focus is on the general architecture variants and including a brief discussion on the specific variants on pre-training and applications.

The sequence-to-sequence model known as the vanilla transformer [15] comprises an encoder and a decoder, both of which are stacks of identical blocks, usually denoted by L. The encoder block mainly consists of a multi-head self-attention module and a position-wise feedforward network (FFN) [15]. Entire architecture of a vanilla transformer can be seen in 2.8.

The key modules in vanilla transformer includes [15]: Attention module, Position-wise FFN, Residual connection and normalization, Position encoding. The architecture of the transformers can be categorized in (1) Encoder-Decoder, (2) Encoder-only, and (3) Decoder-only.

Based on [14], figure 2.9 displays the information flow in a single causal self-attention layer. Input sequences $(x_1, ..., x_n)$ are mapped to output sequences of the same length $(y_1, ..., y_n)$ by a self-attention layer, similar to the overall transformer. While processing each input, the model has access to all inputs up to and including the current one, but no information about inputs beyond it. Additionally, the computation performed for each item is independent of all other computations. These features make it possible to use this method to create language models for autoregressive generation and to easily parallelize both forward inference and training of such models. The fundamental aspect of an attention-based approach lies in the capacity to evaluate the significance of an item within a given context by comparing it to a

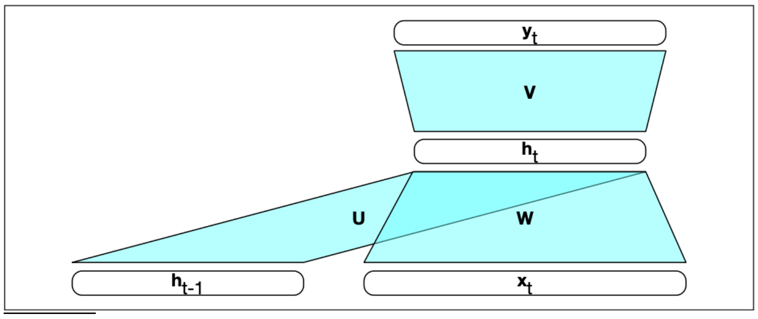


Figure 2.6: Building block of an RNN as a feedforward network [14]

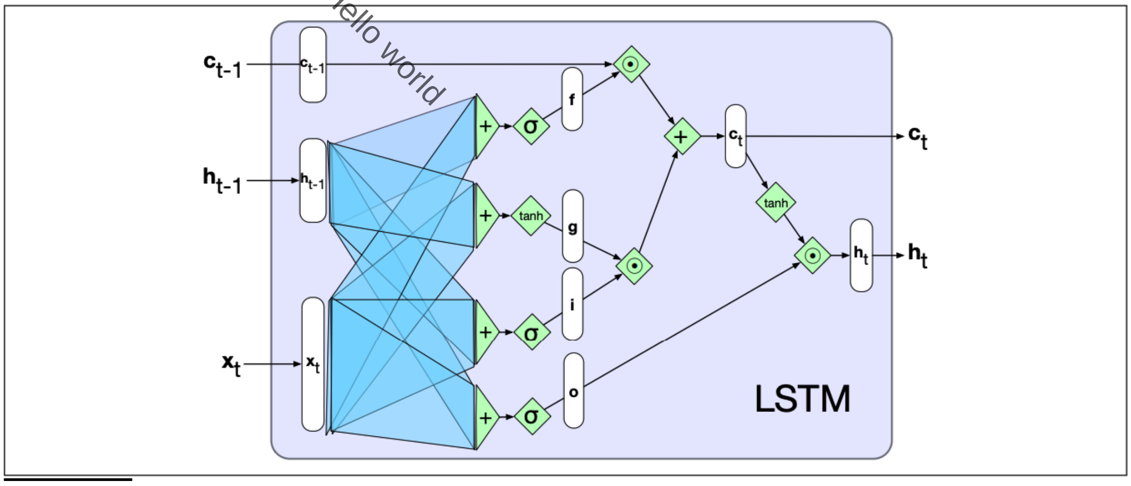


Figure 2.7: Building block of an LSTM cell [14]

set of other items.

In attention layer, each input have three roles: query, key and value. The current element under comparison with all preceding inputs is known as the query, while the preceding input being compared to the current element is called the key. The value refers to the element used to calculate the output for the current focus of attention [14]. The equations for these three values are

$$q_i = W^Q x_i \ ,$$

$$k_i = W^K x_i \ ,$$

$$v_i = W^V x_i \ ,$$

where x_i is input. For each element in attention layer there is a comparison between current input and previous inputs. This comparison is simply done by dot product operation. Since the dot product of these tensors can be large, a division by key vector d_k happens [14]. The equation for score is

$$\text{score}(x_i, x_j) = \frac{q_i . k_j}{\sqrt{d_k}} \ .$$

Finally, a softmax layer is used

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

The full operation of a single output in self-attention layer can be seen in 2.10.

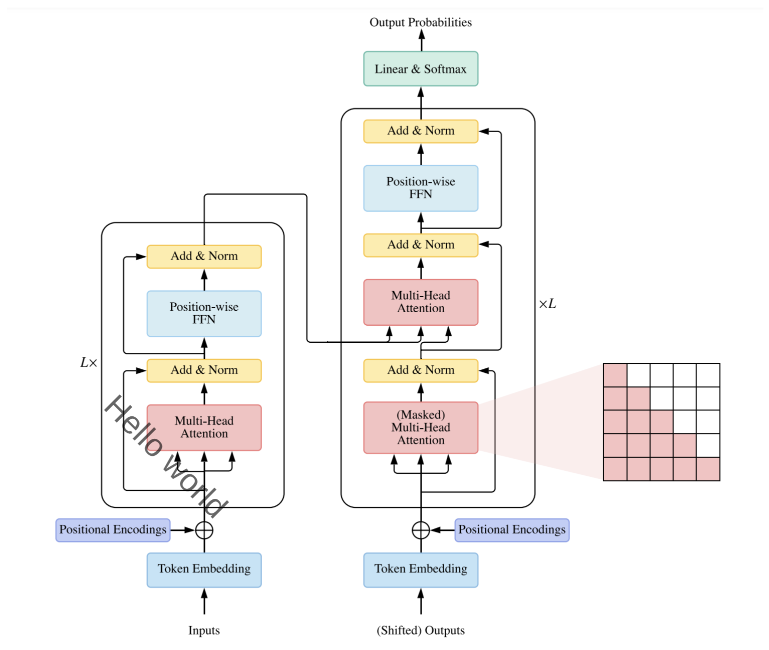


Figure 2.8: Entire architecture of a vanilla transformer [15]

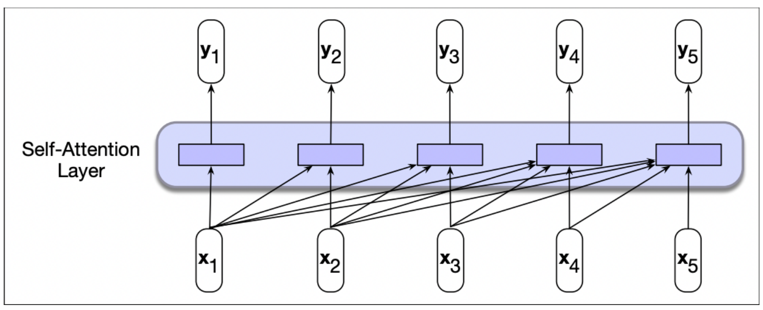


Figure 2.9: The causal (or masked) self-attention model allows information to flow by attending to all inputs up to the current element when processing each sequence element. Unlike RNNs, the model's computations at each time step are independent of all other steps and can thus be executed in parallel. [14]

Now that we have introduced self-attention layer, it is the time to talk about an important section of transformers: Transformer Blocks. The building blocks of a transformer block can be seen in 2.11. It is a sequence-to-sequence entity and the length of input and output are the same so that these blocks could be stacked together [14]. As it can be seen in the figure 2.11, the residual blocks and layer norm are important part of the transformer block.

In a sentence, various words can have multiple simultaneous relationships with each other, such as distinct syntactic, semantic, and discourse connections between verbs and their arguments. Capturing all these different types of parallel relationships among the inputs using a single transformer block would be challenging. To overcome this, transformers utilize multihead self-attention layers consisting of parallel sets of self-attention layers called heads, residing in the same depth of the model but with their unique parameters [14].

With distinct sets of parameters, each head can learn unique aspects of the relationships between inputs at the same level of abstraction. To implement this concept, every head (i) in a self-attention layer is given its own set of key, query, and value matrices (W_i^K, W_i^Q, and W_i^V). These matrices project the inputs into separate key, value, and query embeddings for each head while keeping the rest of the self-attention computation unchanged. In multi-head attention, the key and query embeddings have a dimensionality of d_k, and the value

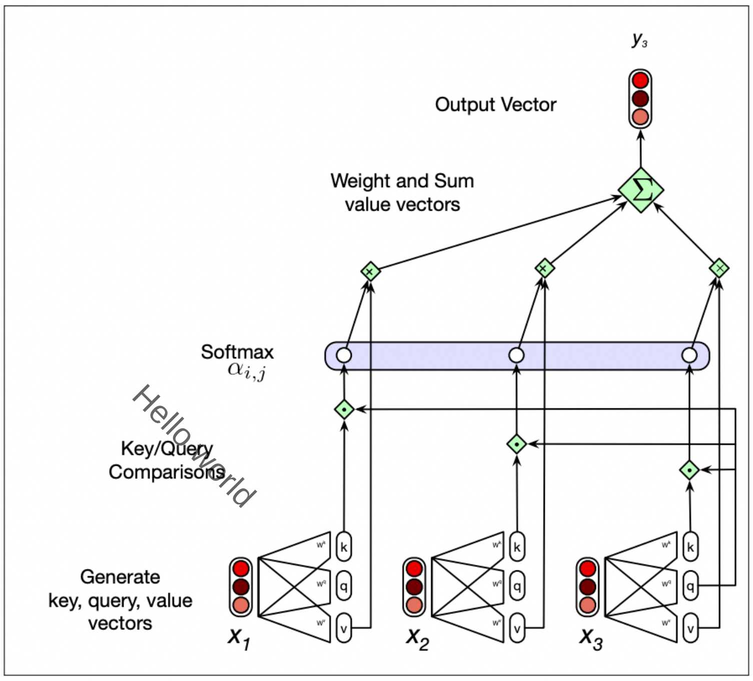


Figure 2.10: Self attention [14]

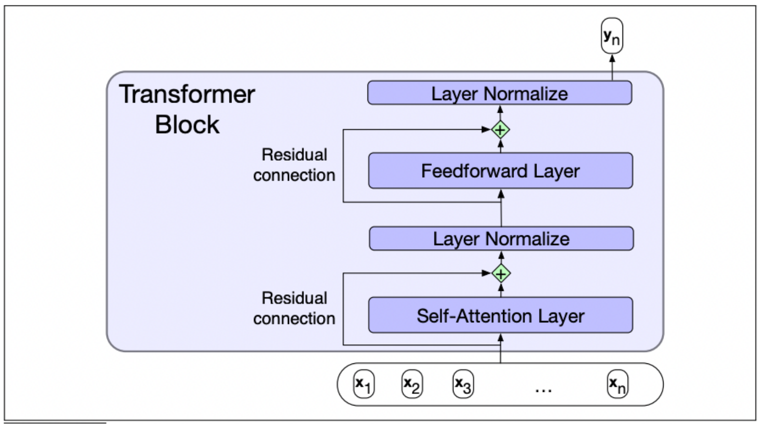
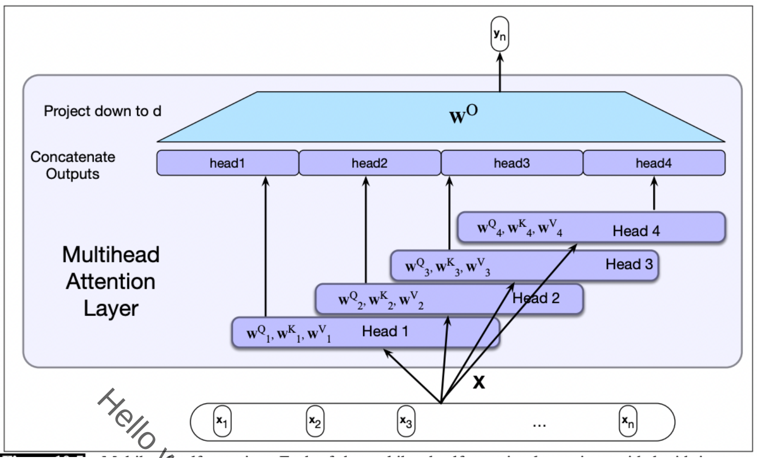


Figure 2.11: Transformer block [14]

embeddings have a dimensionality of d_v (in the original transformer paper, $d_k = d_v = 64$) instead of the model dimension d used for the input and output. Thus, for each head i, we have weight layers $W_i^Q \in R^{d \times d_k}$, $W_i^K \in R^{d \times d_k}$, and $W_i^V \in R^{d \times d_v}$, which are multiplied by the inputs packed into X to create $Q \in R^{N \times d_k}$, $K \in R^{N \times d_k}$, and $V \in R^{N \times d_v}$. The output of each of the h heads is of shape $N \times d_v$, and the output of the multi-head layer with h heads consists of h vectors of shape $N \times d_v$. To use these vectors in further processing, they are combined and then reduced to the original input dimension d by concatenating the outputs from each head and using another linear projection $W^O \in R^{hd_v \times d}$. The result is a total $N \times d$ output for each token [14]. The figure 2.12 is showing a four multiheaded attention.

In a transformer model, there is no inherent structure to encode the order of input tokens unlike RNNs. As a result, the attention computation in 2.10 produces the same result when the input tokens are shuffled. One solution to this problem is to incorporate positional embeddings into the input embeddings. These embeddings are specific to each position in the input sequence and can be initialized randomly up to a certain maximum length. During



y_n
Project down to d
W^O
Concatenate Outputs
head1
head2
head3
head4
Multihead Attention Layer
W^Q_4, W^K_4, W^V_4 Head 4
W^Q_3, W^K_3, W^V_3 Head 3
W^Q_2, W^K_2, W^V_2 Head 2
W^Q_1, W^K_1, W^V_1 Head 1
X
x_1
x_2
x_3
…
x_n

Figure 2.12: Multiheaded attention [14]

training, these positional embeddings are learned along with other parameters. To create an input embedding that incorporates positional information, a corresponding positional embedding is added to the word embedding for each input, resulting in a new vector of the same dimensionality. The positional embedding serves to convey positional information to the model, as shown in 2.13.

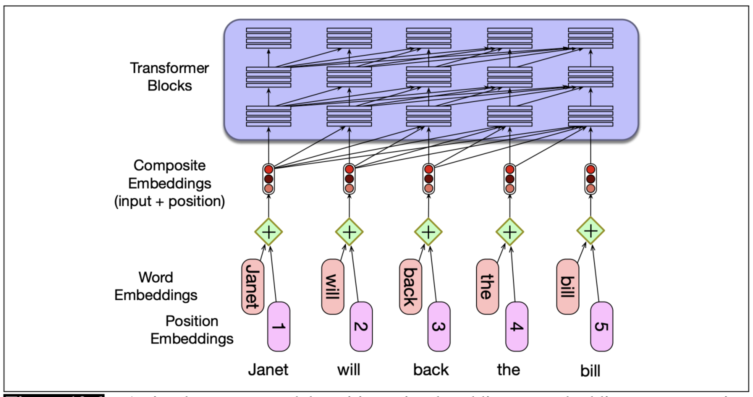
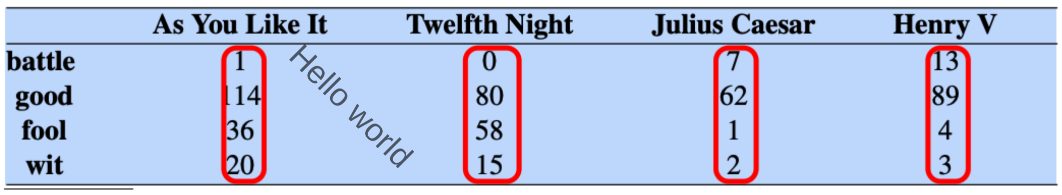


Figure 2.13: Self-attention layer [14]

One issue with using a simple absolute position embedding approach is that there may be an imbalance in the number of training examples for initial positions versus those at the outer length limits, resulting in poorly trained embeddings that may not generalize well during testing. An alternative approach to positional embeddings is to use a static function that maps integer inputs to real-valued vectors in a way that captures the relationships between positions, such as the fact that position 4 is more closely related to position 5 than to position 17. The original transformer work used a combination of sine and cosine functions with varying frequencies for this purpose. However, developing better representations for position is still an area of active research.

2.4 Word Embedding

In general the goal of word embedding is to map the words in unlabeled text data to a continuously-valued low dimensional space in order to capture the internal semantic and syntactic information [16]. One of the widely used approach for expressing the text documents is the Vector Space Model (VSM), where documents are represented as vectors. VSM was originally developed for the SMART information retrieval system [16]. Vector or distributional models of meaning are generally based on a co-occurrence matrix, a way of representing how often words co-occur [14]. In a term-document matrix each row represents a word in a document and each column is document out of collection of documents [14]. In 2.14 it can be seen that there are four words and couple of documents. Each red box can represent a vector for a document. These vectors can be seen for two words battle and fool in 2.15.



	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	1	0	7	13
good	114	80	62	89
fool	36	58	1	4
wit	20	15	2	3

Hello world

Figure 2.14: Vector Space Model [14]

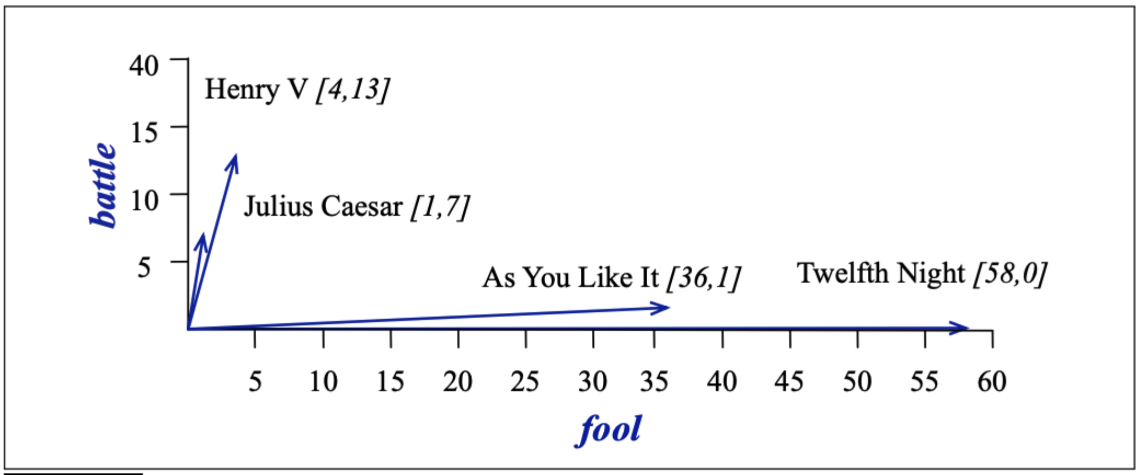
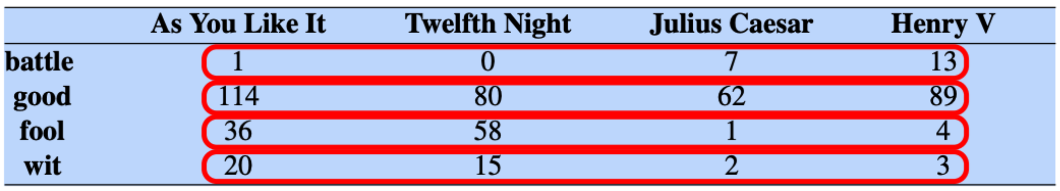


Figure 2.15: A spatial visualization of the document vectors for the four Shakespeare play documents, showing just two of the dimensions, corresponding to the words battle and fool. The comedies have high values for the fool dimension and low values for the battle dimension. [14]

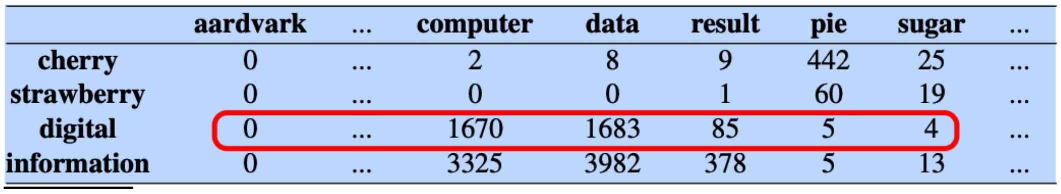
Up to now it has been seen that documents can be represented by vector. But vectors can be used to represent words. In the 2.16 it can be seen that red boxes are the vector representing each word.



	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	1	0	7	13
good	114	80	62	89
fool	36	58	1	4
wit	20	15	2	3

Figure 2.16: Word as vector [14]

Instead of using a term-document matrix to represent words as document count vectors, a term-term matrix, also known as a word-word matrix or term-context matrix, can be utilized. In this matrix, the columns are identified by words rather than documents [14]. This matrix has a size of $|V| \times |V|$, and each cell records how many times a target word in a row and a context word in a column co-occur in some training corpus [14]. The context can be a document, and the cell represents the number of occurrences of the two words in the same document [14]. In 2.17 the co-occurrence of words can be seen as vector.



	aardvark	...	computer	data	result	pie	sugar	...
cherry	0	...	2	8	9	442	25	...
strawberry	0	...	0	0	1	60	19	...
digital	0	...	1670	1683	85	5	4	...
information	0	...	3325	3982	378	5	13	...

Figure 2.17: Word Word vector [14]

In the 2.18 spatial visualization of four words can be seen [14].

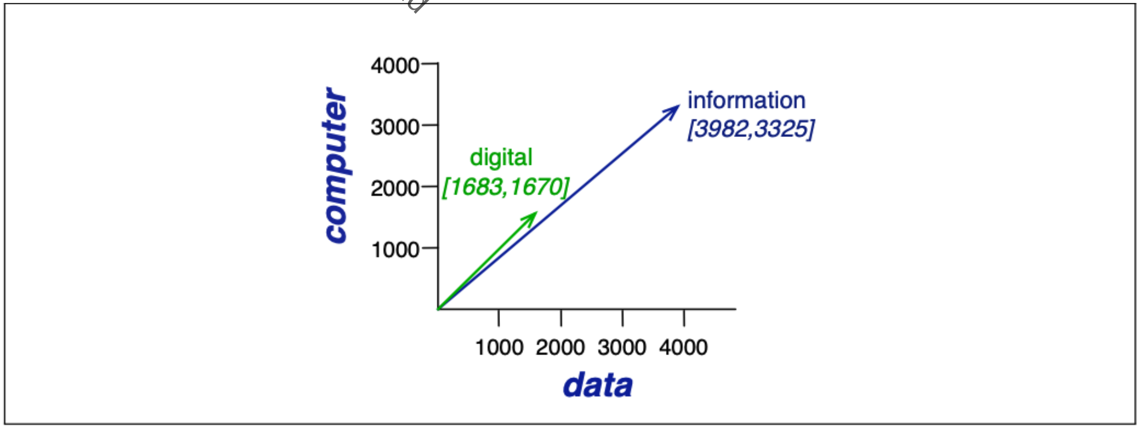


Figure 2.18: A spatial visualization of word vectors for digital and information, showing just two of the dimensions, corresponding to the words data and computer. [14]

2.5 Language Models

Models that assign probability to a sequence of words are called language models [14]. The simplest model that assigns probability to a sequence of words or sentences is called n-gram [14]. In this section we are going to talk about language models and n-grams. N-gram is sequence of n words for example 2-gram or bigram is sequence of two words and 3-gram is sequence of three words. While n-grams are much more simpler than state-of-the-art models such as RNN and transformers, they still have a fundamental impact in NLP [14]. Lets start with counting the $p(w|h)$, the probability of word w given the history of h. For this purpose we can count the sequence h in a large corpus and also count the times h is followed by w

$$\frac{c(hw)}{c(h)} \, .$$

For large corpus the above formula works, but if it is not large enough it will be tricky [14]. The joint probability of sequence $w_1 \ldots w_n$ would be

$$\prod_{k=1}^{n} P(w_k|w_{1:k-1}) \, .$$

Now with Markov assumption the formula

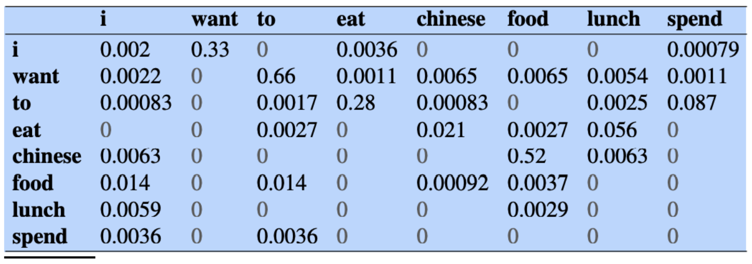
$$P(w_n : w_{1:n-1}) \approx P(w_n | w_{n-N+1:n-1})$$

is deducted. As it can be seen in the formula, we don't look too far in the history, for the n-gram just last N words is considered. Now the probability of last N words can be calculated

$$P(w_n | w_{n-N+1:n-1}) = \frac{C(w_{n-N+1:n-1} w_n)}{C(w_{n-N+1:n-1})} .$$

The above ratio is called the relative frequency. Now let's take a look at example. In 2.19 we can see the Bigram probabilities for eight words in the Berkeley Restaurant Project corpus of 9332 sentences [14]. For example we can calculate the probability of a sentence "I want chinese lunch":

$$\begin{aligned} &P(\text{<s> I want chinese lunch </s>}) = \\ &P(\text{i}|\text{<s>})P(\text{want}|\text{i})P(\text{chinese}|\text{want}) \\ &P(\text{lunch}|\text{chinese})P(\text{</s>}|\text{lunch}) \\ &= .25 \times .33 \times .0065 \times 0.0063 \times 0.68 \\ &= 0.00000229729 \end{aligned}$$



	i	want	to	eat	chinese	food	lunch	spend
i	0.002	0.33	0	0.0036	0	0	0	0.00079
want	0.0022	0	0.66	0.0011	0.0065	0.0065	0.0054	0.0011
to	0.00083	0	0.0017	0.28	0.00083	0	0.0025	0.087
eat	0	0	0.0027	0	0.021	0.0027	0.056	0
chinese	0.0063	0	0	0	0	0.52	0.0063	0
food	0.014	0	0.014	0	0.00092	0.0037	0	0
lunch	0.0059	0	0	0	0	0.0029	0	0
spend	0.0036	0	0.0036	0	0	0	0	0

Figure 2.19: Bigram [14]

2.6 Transformers as Language Models

In this section we will talk about how transformers can be used as a language model. As it was discussed in the previous section, the task in a language model is to predict the probability of next output in a sequence. There is a term called teacher forcing from [14]. In a similar scenario, a technique known as teacher forcing is employed [14]. It's important to remember that during teacher forcing, while decoding, the system is compelled to utilize the correct target token from training as the subsequent input (x_{t+1}), instead of relying on the potentially inaccurate decoder output ($\hat{y}_t$) [14]. The general training approach is depicted in 2.20. In each step, considering the preceding words, the last transformer layer generates a distribution across the entire vocabulary [14]. Throughout the training process, the probability assigned to the correct word is utilized to compute the cross-entropy loss for each item in the sequence [14]. Similar to RNNs, the loss for a training sequence is determined by taking the average of the cross-entropy losses across the entire sequence [14].

To initiate a sequence, we begin by randomly selecting a word based on its suitability as a starting point [14]. Subsequently, we continue selecting words, taking into account our

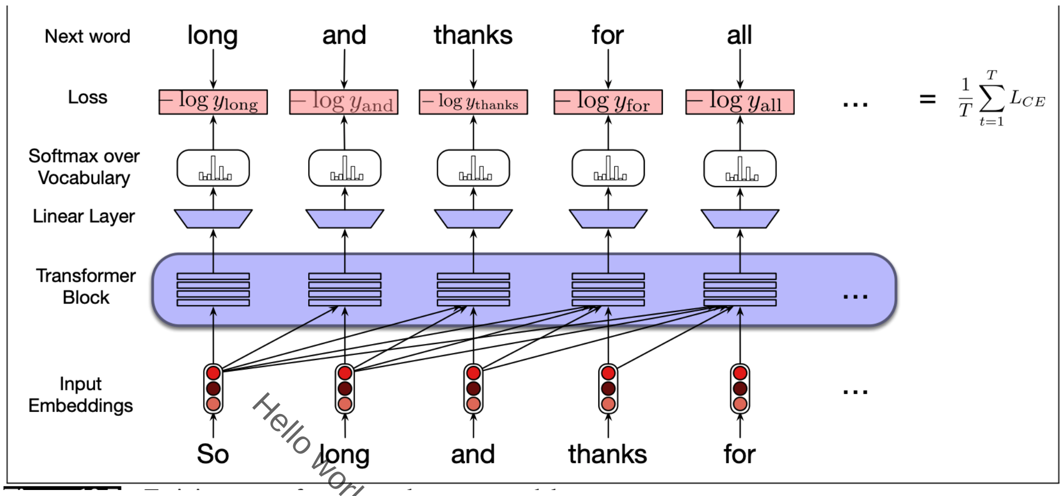


Figure 2.20: Transformers as language model [14]

previous choices, until we reach a predetermined length or encounter an end-of-sequence token [14].

The process of generating text from transformer language models closely resembles the procedure described previously, but it has been adapted to fit within a neural context [14]. We sample a word from the softmax distribution obtained by using the beginning-of-sentence marker, "<s>", as the initial input [14]. The word embedding for the first word is then used as input for the network at the next time step, and we sample the next word in a similar manner [14]. This generation process continues until either the end-of-sentence marker, "</s>", is sampled or a fixed length limit is reached [14]. Technically, an autoregressive model predicts a value at time "t" based on a linear function of the previous values at times "t-1", "t-2", and so on [14]. Although language models are not strictly linear due to their multiple layers of non-linearities, this generation technique is loosely referred as autoregressive generation since each word generated at a given time step is conditioned on the word selected by the network in the previous step [14].

The utilization of language models to generate text has significantly impacted the field of NLP [14]. Text generation, alongside image and code generation, represents a novel domain within AI often referred to as generative AI [14].

Chapter 3

Architecture of the Pretrained Models

In this chapter the BERT model and its results on two datasets: SQuAD and medical-SQuAD-like dataset are stated. At first an introduction to the BERT structure is given and then the results and diagrams on fine tuning BERT with different datasets are reported. Then, GPT-2 is described denoting its inherent differences with BERT. The chapter finishes with a short description of the recent T5 model.

3.1 SQuAD Dataset

The Stanford Question Answering Dataset (SQuAD) [17] is a popular dataset in the field of natural language processing used for training and evaluating machine learning models for question answering tasks. SQuAD was first introduced in 2016 by researchers at Stanford university and is considered one of the most challenging and comprehensive datasets for this task.

The SQuAD dataset consists of over 100,000 question-answer pairs, derived from over 500 Wikipedia articles. The dataset is split into two parts: a training set and a development set. The training set contains approximately 80,000 examples, while the development set contains around 10,000 examples. In addition, SQuAD also includes a separate test set, which is not publicly available, and is used for official evaluations of the performance of models on the task.

Each question-answer pair in SQuAD consists of a question and a corresponding answer, along with the relevant paragraph from the source article that contains the answer. The questions are designed to require complex reasoning and comprehension of natural language, making the dataset a challenging benchmark for NLP models.

SQuAD has been widely used as a benchmark dataset for evaluating the performance of various question answering models, including neural network-based models such as the BiDAF[18] model and the transformer-based models such as BERT and RoBERTa [19]. SQuAD has also been used to train models that can perform more advanced question answering tasks, such as generating answers that are not explicitly stated in the text.

In conclusion, the SQuAD dataset is a widely used benchmark in the field of NLP, used for training and evaluating models for question answering tasks. Its size and complexity make it a challenging benchmark for current state-of-the-art models, and it continues to be used for advancing the field of natural language processing.

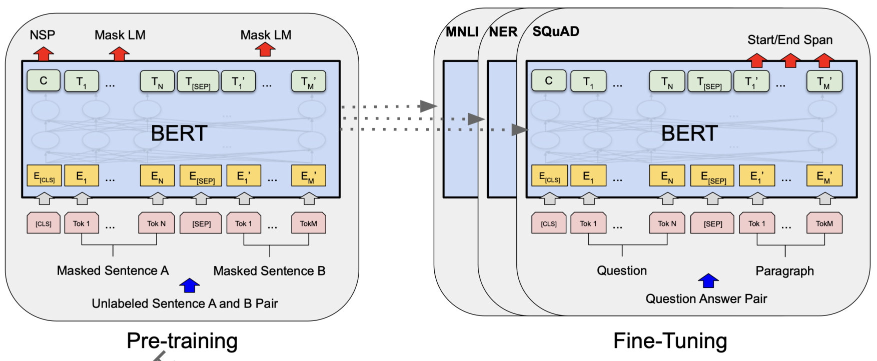


Figure 3.1: There are two steps in BERT: pretraining and finetuning [1]

3.2 BERT Structure

Bidirectional Encoder Representations from Transformers (BERT) was introduced in [1]. The overall architecture of BERT can be seen in 3.1. As it is seen in the picture there are two steps in BERT: pretraining and fine tuning [1]. BERT uses transformers which consist of Encoder and Decoder. An encoder reads the text input and a decoder create the output of the model. According to [1] all of the previous works were one directional: either left to right or right to left, while BERT is a bidirectional transformer. This makes BERT more powerful comparing to one directional transformers. BERT gets the sequence at once and can understand the relation of the words in two directions.

Pre-training step in BERT consists of two tasks: Mask LM and Next Sentence Prediction (NSP). In order to overcome the limitations of one directional training, in BERT, they tried to use a technique called Mask LM. Mask LM is a technique which replace normally 15 percent of the sequence with a mask token and then fed the sequence to the transformer [1]. Another technique used in BERT pre-training is called NSP, which tries to understand the relations between the sentences [1].

Last stage of BERT is the fine-tuning stage, which according to [1] is inexpensive comparing to the first stage.

3.3 GPT-2 Structure

The GPT-2 model has a bit different structure of the BERT model. In 3.2 it can be seen that GPT-2 is decoder-only while BERT is encoder-only. In 3.3 the various versions of GPT-2 can be seen. In this section the details of process when the input is fed to GPT-2 are going to be discussed. As it was mentioned in the previous sections the most important part of the transformers are self-attention layers. GPT-2 has masked self-attention layers which can be seen in 3.4. For a given input at each stage, GPT-2 tries to predict the next word and then output is given to it as next input. This process is illustrated in 3.4. The details of self attention were discussed in the transformers section.

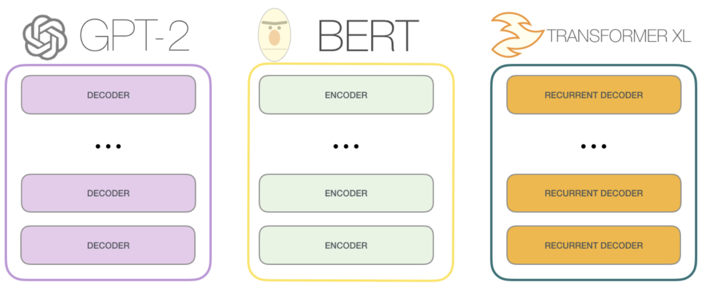


Figure 3.2: The overall structure of GPT-2, BERT and Transformer XL [20]

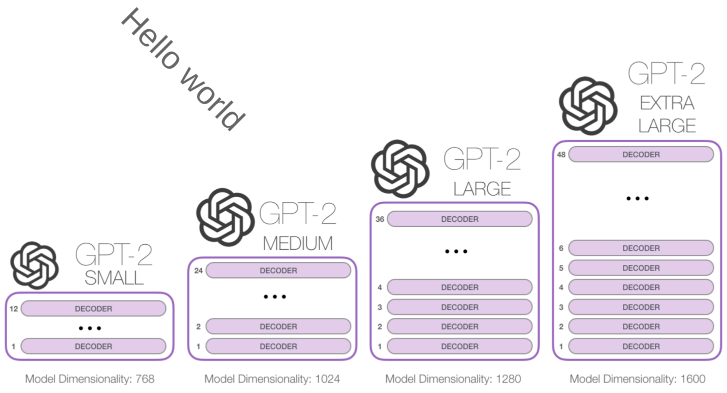

Hello world
GPT-2 SMALL
GPT-2 MEDIUM
GPT-2 LARGE
GPT-2 EXTRA LARGE
DECODER
Model Dimensionality: 768
Model Dimensionality: 1024
Model Dimensionality: 1280
Model Dimensionality: 1600

Figure 3.3: The various versions of GPT-2 [20]

3.4 GPT-2 vs BERT

The transformer architecture comprises two main components: stacked encoders and decoders, connected in a cascading manner [21]. The encoder consists of two parts: a self-attention layer and a feed-forward neural network. Self-attention enables the focus to be directed towards specific word encodings, establishing a contextual framework for each word [21]. On the other hand, the decoder includes an additional encoder-decoder layer that switches between self-attention and the feed-forward network [21]. This allows the decoder to selectively attend to particular segments of the input sequence [21].

GPT-2, unlike the original transformer architecture, eliminates the encoder blocks and retains only the decoder stack [22]. It functions as a traditional language model, capable of predicting the next word in a sequence based on preceding text [22]. Consequently, it is well-suited for general generative tasks [22]. However, as it is not specifically optimized for question generation, there is no guarantee that the generated questions will be valid or answerable [22]. In contrast, BERT, another model, is a masked language model that predicts a masked word considering both its left and right context, effectively establishing word embeddings in a context-specific and bidirectional manner [22]. BERT is primarily trained for discriminative question-answering (QA) using a specific regression head [22]. It predicts the answer text span in a given paragraph for a provided question. Moreover, BERT demonstrates exceptional versatility across various downstream tasks, surpassing the conventional transformer network and differing from GPT-2 by discarding the decoder blocks and becoming a pure encoder stack [22].

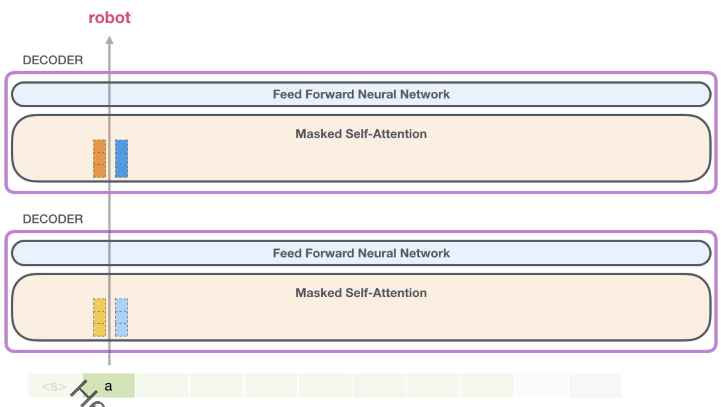


Figure 3.4: The overall structure of GPT-2 with masked self attention [20]

3.5 T5 Structure

In this section another powerful model is going to be discussed which is called Transfer learning with a unified text-to-text transformer or T5. In general, its implementation of the encoder-decoder transformer closely adheres to the original proposal by [3] [23]. Initially, a sequence of tokens representing the input is transformed into a sequence of embeddings, which is then fed into the encoder. The encoder comprises multiple "blocks," each consisting of two components: a self-attention layer followed by a small feed-forward network [23]. Layer normalization [24]is applied to the input of each component [23]. A simplified form of layer normalization is employed in [23] where the activations are rescaled without any additive bias. Additionally, a residual skip connection [25] connects the input and output of each component after layer normalization [23]. Dropout [26] is applied within the feed-forward network, on the skip connection, on the attention weights, as well as at the input and output of the entire stack to introduce regularization [23].

The decoder follows a similar structure to the encoder, with the exception that it includes a standard attention mechanism after each self-attention layer, which attends to the encoder's output [23]. The self-attention mechanism in the decoder utilizes an autoregressive or causal self-attention approach, restricting the model's attention to previous outputs only [23]. The output of the final decoder block is passed through a dense layer with a softmax activation, and the weights of this layer are shared with the input embedding matrix [23]. In the transformer, all attention mechanisms are divided into independent "heads," and their outputs are concatenated before further processing [23]. In the 3.5 some tasks that can be done by T5 can be seen.

3.6 Suitable Models for Question Answering Task

Up to our knowledge there are mainly three types of transformers: decoder-only, encoder-only and encoder-decoder models. From these three types of models decoder-only models such as GPT2 have worse performance on question answering task for datasets like SQuAD. In [2] it was seen that accuracy of GPT2 (smallest model of GPT2) does not exceed more than 1 percent [2]. The main reason for this, from our understanding, is that GPT2 is a generative model and when it comes to generating a sequence, current metrics such as exact matches on answers are not a suitable metric for evaluating the performance. On the other hand, BERT is an encoder only transformer and the task of question answering in case of BERT is similar

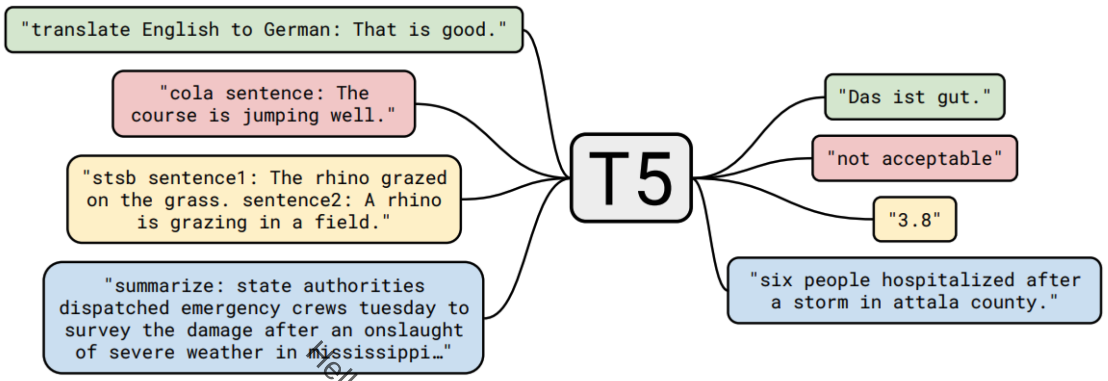

"translate English to German: That is good."
"cola sentence: The course is jumping well."
"stsb sentence1: The rhino grazed on the grass. sentence2: A rhino is grazing in a field."
"summarize: state authorities dispatched emergency crews tuesday to survey the damage after an onslaught of severe weather in mississippi…"
T5
"Das ist gut."
"not acceptable"
"3.8"
"six people hospitalized after a storm in attala county."

Figure 3.5: The text-to-text framework of T5 is represented by a diagram. Within this framework, every task, such as translation, question answering, and classification, is approached by providing textual input to the model and training it to produce the desired output text. This approach enables utilization of a consistent model, loss function, hyperparameters, and more, across a wide range of tasks. Furthermore, it establishes a standardized evaluation platform for the methods analyzed in the empirical survey. The term "T5" is used to refer to the model, as a naming for the "Text-to-Text Transfer Transformer." [23]

to text classification. The output is a sequence with start and end of the answer. Since it is not a generative model like GPT2 it has the accuracy of 80 percent for exact matches on the SQuAD dataset. There is the third type of transformers such as T5 which has a structure of encoder-decoder and it has an accuracy of 80 percent [23] on the SQuAD dataset.

Hello world

Chapter 4

Experiments and Results

In this chapter the details of experiments are going to be discussed and the libraries and repositories that were used for the experiments are going to be introduced. The chapter includes the results of the experiments for the QA task.

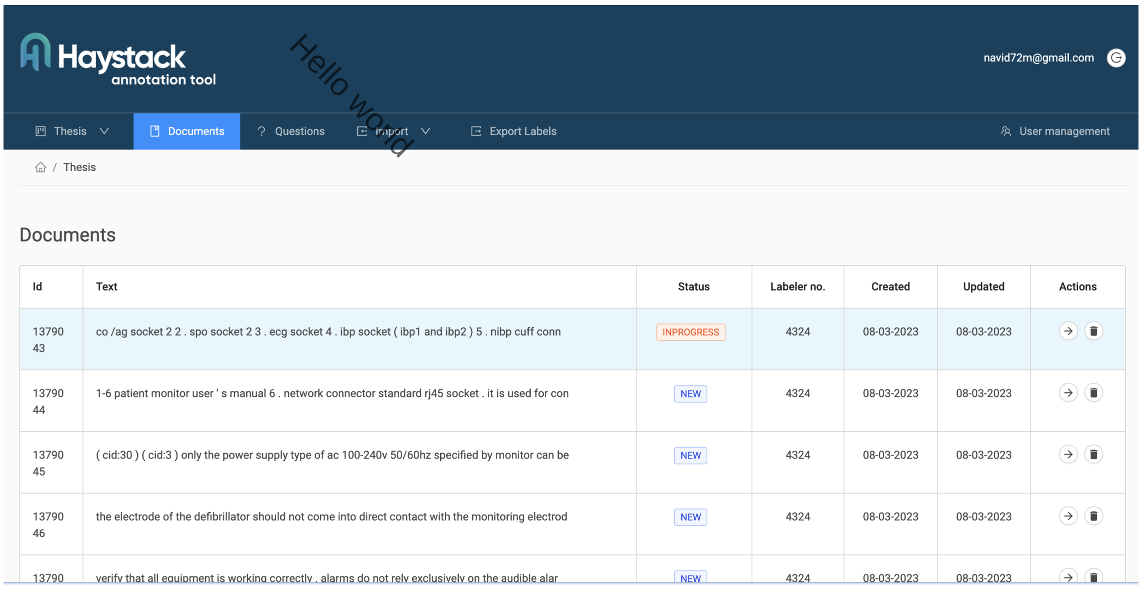
4.1 Experiments' Setup and Configuration

In this section the details of the main experiments are going to be discussed. The first performed experiment is fine-tuning of BERT on the SQuAD dataset. For this purpose, the open-source implementation of BERT in Tensorflow 1 was used. The repository for this experiment was provided in [1] by Google research. For finetuning BERT on SQuAD, there was a script *run-squad.py* in the repository. The hyperparameters for the experiments can be seen in 4.1. After running the script, the predictions of the model were saved in a desired directory. To evaluate the predictions, there was another script called *evaluate.py* in the official site of SQuAD [17]. After the experiment run, the exact match accuracy for SQuAD 1.1 on BERT was **80 percent**.

Hyperparameter	Value
learning rate	3e-5
epochs	2
doc stride	128
max sequence length	384
batch size	2

The next experiment is finetuning the pretrained BERT on our own dataset. For this reason, the haystack platform [27] was used to create a dataset like SQuAD. In the next section more details regarding the costumized dataset will be discussed. As it was mentioned, SQuAD has a specific structure. In order to create a dataset from medical manual devices aligning with the SQuAD structure, we used haystack annotation tools which can be seen in figure 4.1. To use the annotation tool, one should first create an account in the platform and register itself with an email address. As it can be seen in the 4.1, there are multiple documents in the project. These documents are retrieved from a medical manual device "M900" and a python code was written to convert the *pdf* file to raw text and put it into different documents. There were total of 78 documents for this dataset. For each document about five question and answer were created. Finally, the whole dataset was downloaded and exported to SQuAD format.

```
import pdfplumber
f = open("manual",'w')
with pdfplumber.open(r'manual.pdf') as pdf:
    for page in pdf.pages:
        # first_page = pdf.pages[0]
        print(page.extract_text())
        f.write(page.extract_text())
    f.close()
```



Id	Text	Status	Labeler no.	Created	Updated	Actions
1379043	co /ag socket 2 2 . spo socket 2 3 . ecg socket 4 . ibp socket (ibp1 and ibp2) 5 . nibp cuff conn	INPROGRESS	4324	08-03-2023	08-03-2023	
1379044	1-6 patient monitor user ' s manual 6 . network connector standard rj45 socket . it is used for con	NEW	4324	08-03-2023	08-03-2023	
1379045	(cid:30) (cid:3) only the power supply type of ac 100-240v 50/60hz specified by monitor can be	NEW	4324	08-03-2023	08-03-2023	
1379046	the electrode of the defibrillator should not come into direct contact with the monitoring electrod	NEW	4324	08-03-2023	08-03-2023	
13790	verify that all equipment is working correctly . alarms do not rely exclusively on the audible alar	NEW	4324	08-03-2023	08-03-2023	

Figure 4.1: Haystack annotation tool

As an important part of this thesis, an experiment is done that tries to finetune the BERT-Base-uncased on SQuAD. The diagram for loss and accuracy can be seen in the following.

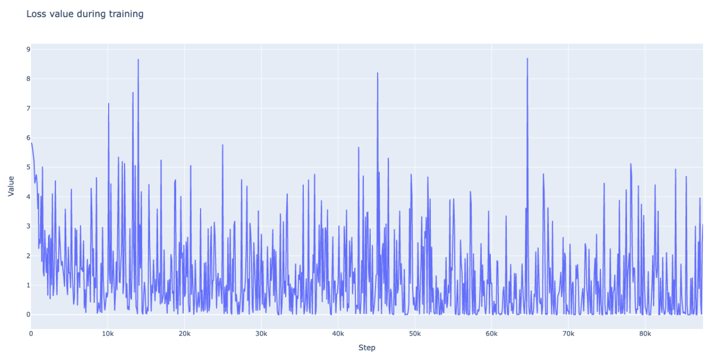


Figure 4.2: The loss diagram for finetuning BERT on SQuAD

As it can be seen, the maximum accuracy for SQuAD is **80 percent** while loss converges to the value near **1**. In contrast, in the next section we can see that the maximum accuracy for the customized dataset is **50 percent** while loss converges to **0**.

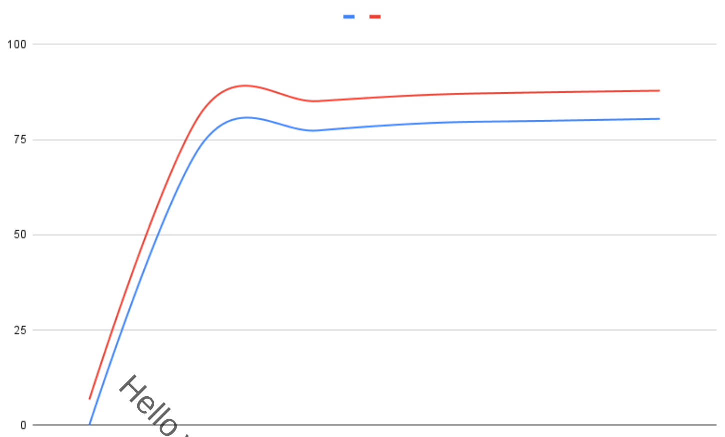


Figure 4.3: The accuracy diagram for finetuning BERT on SQuAD

4.2 Medical SQuAD-Like Dataset

The medical SQuAD-like dataset is a small dataset that has been created in this thesis to evaluate the performance of the selected models in a transferred domain. For this purpose, a manual of a medical device called M9500 Patient Monitor is used. By using haystack annotation tools, a dataset with SQuAD's structure is created. This dataset has 120 pair of questions and answers. In this experiment, BERT was finetuned with the medical SQuAD-like dataset two times: first time with SQuAD pretraining and the second time without SQuAD pretraining. The diagrams 4.4 and 4.5 show the loss of the BERT model for the first and second training respectively. As it can be seen by comparing the plots, the pretrained model finetuned on SQuAD converges faster comparing to the case that BERT finetuning with SQuAD is missing. This reflects the point that pretraining on SQuAD helps in transferring the model to a different-but-alike customized dataset. There is another point in these diagrams, the model's loss is converging to **0** but the maximum accuracy is **50 percent** which is significantly lower than the observed **80 percent** for SQuAD. The reason could be due to the limitation that the data set is not enough large.

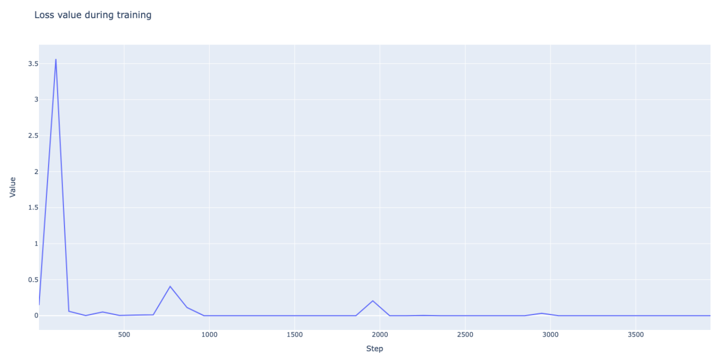


Figure 4.4: The loss diagram for the finetuned version of BERT with SQuAD on the medical SQuAD-like dataset

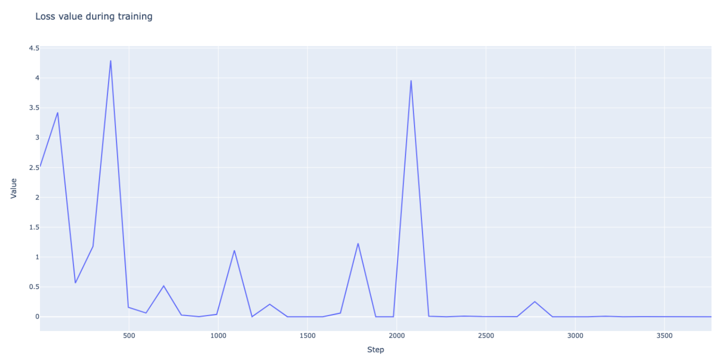


Figure 4.5: The loss diagram for BERT training on the medical SQuAD-like dataset without transfer learning or previous finetuning on SQuAD

Chapter 5

Conclusion and Summary

In this chapter the conclusion and summary of this project will be discussed. Outlook and future works will be discussed at the end of this chapter.

5.1 Summary

This project was intended to study different architecture of transformers. In the first chapter the supervised learning and general natural language processing methods were discussed. Regular expression and language models are some common NLP methods that are widely used. Furthermore, some classic methods in supervised learning such as SVM and HMMs were presented in the first chapter. Also the concept of transfer learning was described to end the chapter. The second chapter was about background and related works on this project. At the beginning, the concept of neuron and neural network was introduced. Then based on feedforward network and loop in hidden layer the structure of recurrent neural networks was discussed. After introducing RNNs, the more powerful models, LSTMs were presented and it has been summarized that LSTMs are more powerful in terms of keeping long distance history in a sequence. After RNNs and LSTMs, the architecture of transformers were discussed. The most important section of a transformer block is the self-attention layer. The details of the self-attention layer was discussed in section 2.3 and after that the whole structure of a transformer block was investigated. In the following section, the important concept of word embedding was introduced and studied. Most of the modern NLP methods use embedding as part of their process. At final section of chapter 2, the concept of language models was introduced. It was seen that one of the simplest language models are N-grams. Transformers also can be seen as language models. Third chapter was about architecture of the pretrained models. In the first section of this chapter, the structure of SQuAD dataset was discussed. Then, the architecture of BERT was studied. In the third section, GPT, another transformer model, was introduced. It was mentioned that GPT is decoder-only while BERT is encoder-only model. Finally in the last section of the third chapter, the powerful model of T5 was introduced and studied. Chapter four was about the experiments and results. By relying on an available repository written in Tensorflow, we were able to finetune the BERT model on SQuAD. After complete training, the accuracy on SQuAD 1.1 was 80 percent. However, when it comes to our own customized dataset, which includes 120 pair of question and answers, the training accuracy does not exceed more than 50 percent. Our assumption was that since the length of dataset is much shorter than the SQuAD, the accuracy does not get higher than 50 percent. We proved that assumption by designing an experiment which accumulatively increase the size of dataset and saw that the accuracy grows with size of the dataset. As it was observed during the thesis, the transformers have a large potential for different tasks.

Some future works can be: question generation and text summarization.

5.2 Final Words

This project was a research project comparing different state-of-the-art models on a specific dataset. Our dataset was retrieved from a pdf which was some how noisy because of the figures and tables and lacked certain amount of curation. There were some uncommon characters which lead to decreasing of the accuracy. So the idea for future work would be to use diffusion techniques to reduce the noise and increase the accuracy.

Appendix A

Appendix 1

A.1 Question Answering with GPT-3

In this section, the details of one experiment that was done during this project is going to be discussed. In this experiment, a question answering system was created using the OpenAI cookbook library. First, the desired libraries should be imported:

```
import pandas as pd
import openai
import numpy as np
import pickle
from transformers import GPT2TokenizerFast
import gensim
from gensim.models import Word2Vec

COMPLETIONS_MODEL = "text−davinci −002"
```

Then a simple prompt is performed and it is observed that GPT-3 is not able to answer a question before learning the context.

```
prompt = "Who won the 2020 Summer Olympics men's high jump?"

openai.Completion.create(
    prompt=prompt,
    temperature=0,
    max_tokens=300,
    top_p=1,
    frequency_penalty=0,
    presence_penalty=0,
    model=COMPLETIONS_MODEL
)["choices"][0]["text"].strip(" \n")

"The 2020 Summer Olympics men's high jump was won by
Mariusz Przybylski of Poland."
```

The first issue to tackle is that the model is hallucinating an answer rather than telling "I don't know". This is bad because it makes it hard to trust the answer that the model gives [28]. One can address this hallucination issue by being more explicit with the prompt [28]:

```
prompt = """Answer the question as truthfully as possible,
and if you're unsure of the answer, say "Sorry, I don't know".
```

```
Q: Who won the 2020 Summer Olympics men's high jump?
A:"""

openai.Completion.create(
    prompt=prompt,
    temperature=0,
    max_tokens=300,
    top_p=1,
    frequency_penalty=0,
    presence_penalty=0,
    model=COMPLETIONS_MODEL
)["choices"][0]["text"].strip(" \n")

"Sorry, I don't know."
```

To help the model answer the question, extra contextual information is provided in the prompt. When the total required context is short, this can be included in the prompt directly. For example, the following information taken from Wikipedia can be used. The initial prompt is updated to tell the model to explicitly make use of the provided text [28].

```
prompt = """Answer the question as truthfully as possible using the
provided text, and if the answer is not contained within the text
below, say "I don't know"

Context:
The men's high jump event at the 2020 Summer Olympics took place
between 30 July and 1 August 2021 at the Olympic Stadium.
33 athletes from 24 nations competed; the total possible number
depended on how many nations would use universality places to
enter athletes in addition to the 32 qualifying through mark or
ranking (no universality places were used in 2021).
Italian athlete Gianmarco Tamberi along with Qatari athlete
Mutaz Essa Barshim emerged as joint winners of the event
following a tie between both of them as they cleared 2.37m.
Both Tamberi and Barshim agreed to share the gold medal in a
rare instance where the athletes of different nations had
agreed to share the same medal in the history of Olympics.
Barshim in particular was heard to ask a competition official
"Can we have two golds?" in response to being offered a
'jump off'. Maksim Nedasekau of Belarus took bronze.
The medals were the first ever in the men's high jump for
Italy and Belarus, the first gold in the men's high jump for
Italy and Qatar, and the third consecutive medal in the men's
high jump for Qatar (all by Barshim).
Barshim became only the second man to earn three medals in
high jump, joining Patrik Sjoeberg of Sweden (1984 to 1992).

Q: Who won the 2020 Summer Olympics men's high jump?
A:"""

openai.Completion.create(
    prompt=prompt,
    temperature=0,
    max_tokens=300,
    top_p=1,
    frequency_penalty=0,
    presence_penalty=0,
    model=COMPLETIONS_MODEL
)["choices"][0]["text"].strip(" \n")
```

```
"Gianmarco Tamberi and Mutaz Essa Barshim won the 2020
Summer Olympics men's high jump."
```

Adding extra information into the prompt only works when the dataset of extra content that the model may need to know is small enough to fit in a single prompt [28]. What can be done when it is needed that the model choose relevant contextual information from within a large body of information? In the remainder of this notebook, a method for augmenting GPT-3 with a large body of additional contextual information is demonstrated by using document embeddings and retrieval [28]. This method answers queries in two steps: first it retrieves the information relevant to the query, then it writes an answer tailored to the question based on the retrieved information. The first step uses the Embedding API, the second step uses the Completions API. The steps are [28]: (1) preprocess the contextual information by splitting it into chunks and create an embedding vector for each chunk; (2) on receiving a query, embed the query in the same vector space as the context chunks and find the context embeddings, which are most similar to the query; (3) prepend the most relevant context embeddings to the query prompt; (4) submit the question along with the most relevant context to GPT, and receive an answer which makes use of the provided contextual information [28].

It is planned to use document embeddings to fetch the most relevant parts of the document library and insert them into the prompt that it is provided to GPT-3 [28]. Therefore, it is needed to break up the document library into "sections" of context, which can be searched and retrieved separately [28]. Sections should be large enough to contain enough information to answer a question; but small enough to fit one or several into the GPT-3 prompt [28]. It has been founded that a paragraph of text is usually a good length, but one should experiment for the particular use case [28]. In this example, Wikipedia articles are already grouped into semantically related headers, so these are used to define the sections [28]. This preprocessing has already been done in this notebook, so the results are loaded and used [28].

```
df = pd.read_csv('manual.csv')
df = df.set_index(["title", "heading"])
print(f"{len(df)} rows in the data.")
df.sample(5)
```

As it can be seen in the above code the csv file is replaced with *manual.csv*, which is created by a script from a manual device file. The document sections are preprocessed by creating an embedding vector for each section. An embedding is a vector of numbers that helps to understand how semantically similar or different the texts are. The closer two embeddings are to each other, the more similar are their contents. See the documentation on OpenAI embeddings for more information. This indexing stage can be executed offline and only runs once to precompute the indexes for the dataset so that each piece of content can be retrieved later. Since this is a small example, storing and searching the embeddings are done locally. If there is a larger dataset, one should consider using a vector search engine like Pinecone or Weaviate to power the search. For the purposes of this tutorial, Curie embeddings are used, which are 4096-dimensional embeddings at a very good price and performance point. Since these embeddings will be used for retrieval, the "search" embeddings are used accordingly.

```
MODEL_NAME = "curie"

DOC_EMBEDDINGS_MODEL = f"text-search-{MODEL_NAME}-doc-001"
QUERY_EMBEDDINGS_MODEL = f"text-search-{MODEL_NAME}-query-001"

def get_embedding(text: str, model: str) -> list[float]:
#     result = openai.Embedding.create(
#       model=model,
#       input=text
#     )
    temp = gensim.models.Word2Vec(text,vector_size = 4096)
    print(temp)
    temp.wv.save_word2vec_format("example.model")
    lines = []
    result = []
    with open("example.model") as exampleEmbedding:
        line=exampleEmbedding.readline()
        print(line)
        lines = exampleEmbedding.readlines()
        for l in lines:
            splitted_line = l.split(" ")
            count = 0
            for item in splitted_line:
#                   print(item)
                if count > 0 and item!="":
                    result.append(float(item))
                count +=1
#     return result["data"][0]["embedding"]
    return result

def get_doc_embedding(text: str) -> list[float]:
    return get_embedding(text, DOC_EMBEDDINGS_MODEL)

def get_query_embedding(text: str) -> list[float]:
    return get_embedding(text, QUERY_EMBEDDINGS_MODEL)

def compute_doc_embeddings(df: pd.DataFrame) ->
dict[tuple[str, str], list[float]]:
    """
    Create an embedding for each row in the dataframe
    using the OpenAI Embeddings API.

    Return a dictionary that maps between each embedding
    vector and the index of the row that it corresponds to.
    """
    return {
        idx: get_doc_embedding(r.content.replace("\n", " "))
        for idx, r in df.iterrows()
    }


def load_embeddings(fname: str) ->
dict[tuple[str, str], list[float]]:
    """
    Read the document embeddings and their keys from a CSV.

    fname is the path to a CSV with exactly these named columns:
        "title", "heading", "0", "1", ... up to the length of
        the embedding vectors.
    """

    df = pd.read_csv(fname, header=0)
    max_dim = max([int(c) for c in df.columns
    if c != "title" and c != "heading"])
    return {
          (r.title, r.heading): [r[str(i)]
          for i in range(max_dim + 1)] for _, r in df.iterrows()
    }

document_embeddings = load_embeddings("manual_embedding.csv")

example_entry = list(document_embeddings.items())[0]
print(f"{example_entry[0]} : {example_entry[1][:5]}...
({len(example_entry[1])} entries)")

(0, 'Text') : [-0.01415598, -0.01415598, -0.021078616,
0.0022452837, 0.045371708]... (4096 entries)
```

So far, the document library is splitted into sections and encoded by creating embedding vectors that represent each chunk. Next, these embeddings are used to answer the users' questions. At the time of question-answering, to answer the user's query, the query embedding of the question are computed and used to find the most similar document sections.

```
def vector_similarity(x: list[float], y: list[float]) -> float:
    """
    We could use cosine similarity or dot product to calculate
    the similarity between vectors.
    In practice, we have found it makes little difference.
    """
#     print(np.array(y))
#     print(np.array(x[:4096]))
    dots_arr = []

#     print("hello")
    for i in range(int(len(x)/4096)):
        dots_arr.append(np.dot(np.array(x[i*4096:(i+1)*4096]),
        np.array(y)))
    return max(dots_arr)

def order_document_sections_by_query_similarity(query: str,
contexts: dict[(str, str), np.array]) -> list[(float, (str, str))]:
    """
    Find the query embedding for the supplied query, and compare it
    against all of the pre-calculated document embeddings
    to find the most relevant sections.

    Return the list of document sections, sorted by relevance in
    descending order.
    """
    query_embedding = get_query_embedding(query)

    document_similarities = sorted([
        (vector_similarity(query_embedding, doc_embedding),
        doc_index) for doc_index, doc_embedding in contexts.items()
        ], reverse=True)

    return document_similarities

question = "what should we do for continuous safe use of this
equipment?"
order_document_sections_by_query_similarity(question,
document_embeddings)[:5]

[(0.006631785467707398, (2, 'Text')),
 (0.005733592896189504, (1, 'Text')),
 (0.0056984540664277526, (5, 'Text')),
 (0.005192783225401789, (21, 'Text')),
 (0.005181982634124743, (3, 'Text'))]
```

Once the most relevant pieces of context are calculated, a prompt is constructed by simply prepending the context to the supplied query. It is helpful to use a query separator to help the model distinguish between separate pieces of text [28].

```
MAX_SECTION_LEN = 500
SEPARATOR = "\n* "

tokenizer = GPT2TokenizerFast.from_pretrained("gpt2")
separator_len = len(tokenizer.tokenize(SEPARATOR))

f"Context separator contains {separator_len} tokens"

def construct_prompt(question: str, context_embeddings: dict,
df: pd.DataFrame) -> str:
    """
    Fetch relevant
    """
    most_relevant_document_sections =
    order_document_sections_by_query_similarity(question,
    context_embeddings)

    chosen_sections = []
    chosen_sections_len = 0
    chosen_sections_indexes = []

    for _, section_index in most_relevant_document_sections:
        # Add contexts until we run out of space.
        document_section = df.loc[section_index]

        chosen_sections_len += document_section.tokens +
        separator_len
        if chosen_sections_len > MAX_SECTION_LEN:
            break

        chosen_sections.append(SEPARATOR +
        document_section.content.replace("\n", " "))
        chosen_sections_indexes.append(str(section_index))

    # Useful diagnostic information
    print(f"Selected {len(chosen_sections)} document sections:")
    print("\n".join(chosen_sections_indexes))

    header = """Answer the question as truthfully as possible
    using the provided context, and if the answer is not contained
    within the text below, say "I don't know."\n\nContext:\n"""

    return header + "".join(chosen_sections) + "\n\n Q: " +
    question + "\n A:"


prompt = construct_prompt(
    question,
    document_embeddings,
    df
)

print("===\n", prompt)
```

```
Word2Vec<vocab=5, vector_size=4096, alpha=0.025>
5 4096

Selected 1 document sections:
(2, 'Text')
===
 Answer the question as truthfully as possible using the
 provided context, and if the answer is not contained within
 the text below, say "I don't know."

Context:

* ( cid:30 ) ( cid:3 ) only the power supply type of ac 100-240v
50/60hz specified by monitor can be used . ( cid:30 ) ( cid:3 )
connect the electrical wire to a properly grounded socket . avoid
```

```
putting the socket used for it in the same loop of such devices as
the air conditioners , which regularly switch between on and off .
( cid:30 ) ( cid:3 ) avoid putting the monitor in the locations
where it easily shakes or wobbles . ( cid:30 ) ( cid:3 ) enough
space shall be left around the monitor so as to guarantee normal
ventilation . ( cid:30 ) ( cid:3 ) make sure the ambient temperature
and humidity are stable and avoid the occurrence of condensation in
the work process of the monitor . ( cid:48 ) warning : never install
the monitor in an environment where flammable anesthetic gas is
present . 2 . monitor conforms to the safety requirements of iec
60601-1:1995 . this monitor is protected against defibrillation
effects . 2-1 patient monitor user   s manual 3 . notes on signs
related to safety type cf applied part , defibrillation protected
the unit displaying this symbol contains an f-type isolated
( floating ) applied part providing a high degree of protection
against shock , and is defibrillator-proof . the type cf applied
parts provide a higher degree of protection against electric shock
than that provided by type bf applied parts . attention ! please
refer to the documents accompanying this monitor ( this manual ) !
type bf applied part , defibrillation protected the unit displaying
this symbol contains an f-type isolated ( floating ) applied part
providing a high degree of protection against shock , and is
defibrillator-proof . 4 . when a defibrillator is applied on a
patient , the monitor may have transient disorders in the display
of waveforms . if the electrodes are used and placed properly ,
the display of the monitor will be restored within 10 seconds .
during defibrillation , please note to remove the electrode of chest
lead and move the electrode of limb lead to the side of the limb .

Q: what should we do for continuous safe use of this equipment?
A:


COMPLETIONS_API_PARAMS = {
    # We use temperature of 0.0 because it gives the most
    predictable, factual answer.
    "temperature": 0.5,
    "max_tokens": 500,
    "model": COMPLETIONS_MODEL,
}


def answer_query_with_context(
    query: str,
    df: pd.DataFrame,
    document_embeddings: dict[(str, str), np.array],
    show_prompt: bool = False
) -> str:
    prompt = construct_prompt(
        query,
        document_embeddings,
        df
    )

    if show_prompt:
        print(prompt)

    response = openai.Completion.create(
                prompt=prompt,
                **COMPLETIONS_API_PARAMS
            )

    return response["choices"][0]["text"].strip(" \n")

answer_query_with_context(question, df, document_embeddings)

Word2Vec<vocab=5, vector_size=4096, alpha=0.025>
5 4096

Selected 1 document sections:
(2, 'Text')
'enough space shall be left around the monitor so as
to guarantee normal ventilation.'
```

By combining the Embeddings and Completions APIs, a question-answering model is created which thus can answer questions using a large base of additional knowledge [28]. It also understands when it doesn't know the answer. For this example, a dataset of Wikipedia articles have been used, but that dataset could be replaced with books, articles, documentation, service manuals, or much much more [28]

A.2 Fine Tuning GPT-2

Another experiment that was done during this project was fine tuning GPT-2 on the SQuAD dataset. For this purpose, the GitHub repository [29] was used. This repository uses PyTorch to finetune GPT-2 on SQuAD. Since GPT-2 is a generative model its accuracy on SQuAD found to be 0 percent [2].

Hello world

Bibliography

[1] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[2] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[3] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[4] K. W. Church, "Word2vec," *Natural Language Engineering*, vol. 23, no. 1, pp. 155–162, 2017.

[5] P. M. Nadkarni, L. Ohno-Machado, and W. W. Chapman, "Natural language processing: an introduction," *Journal of the American Medical Informatics Association*, vol. 18, no. 5, pp. 544–551, 2011.

[6] C. D. Manning, *An introduction to information retrieval*. Cambridge university press, 2009.

[7] V. Singh and G. Kaur, "The review on natural language processing (nlp) and current nlp system architecture," *Turkish Journal of Computer and Mathematics Education (TURCOMAT)*, vol. 12, no. 1, pp. 836–841, 2021.

[8] S. R. Eddy, "What is a hidden markov model?," *Nature biotechnology*, vol. 22, no. 10, pp. 1315–1316, 2004.

[9] L. R. Rabiner, "A tutorial on hidden markov models and selected applications in speech recognition," *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, 1989.

[10] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[11] P. Cunningham, M. Cord, and S. J. Delany, "Supervised learning," *Machine learning techniques for multimedia: case studies on organization and retrieval*, pp. 21–49, 2008.

[12] A. Conneau, D. Kiela, H. Schwenk, L. Barrault, and A. Bordes, "Supervised learning of universal sentence representations from natural language inference data," *arXiv preprint arXiv:1705.02364*, 2017.

[13] K. Weiss, T. M. Khoshgoftaar, and D. Wang, "A survey of transfer learning," *Journal of Big data*, vol. 3, no. 1, pp. 1–40, 2016.

[14] D. Jurafsky and J. H. Martin, "Speech and language processing (3rd ed. draft) dan jurafsky and james h. martin."

[15] T. Lin, Y. Wang, X. Liu, and X. Qiu, "A survey of transformers," *AI Open*, 2022.

[16] Y. Li and T. Yang, "Word embedding for understanding natural language: a survey," *Guide to big data applications*, pp. 83–104, 2018.

[17] P. Rajpurkar, J. Zhang, K. Lopyrev, and P. Liang, "Squad: 100,000+ questions for machine comprehension of text," *arXiv preprint arXiv:1606.05250*, 2016.

[18] M. Seo, A. Kembhavi, A. Farhadi, and H. Hajishirzi, "Bidirectional attention flow for machine comprehension," *arXiv preprint arXiv:1611.01603*, 2016.

[19] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.

[20] J. Alammar, "The illustrated gpt-2 (visualizing transformer language models)."

[21] T. Klein and M. Nabi, "Learning to answer by learning to ask: Getting the best of gpt-2 and bert worlds," *arXiv preprint arXiv:1911.02365*, 2019.

[22] X. Liu, Y. Zheng, Z. Du, M. Ding, Y. Qian, Z. Yang, and J. Tang, "Gpt understands, too," *arXiv preprint arXiv:2103.10385*, 2021.

[23] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *The Journal of Machine Learning Research*, vol. 21, no. 1, pp. 5485–5551, 2020.

[24] J. L. Ba, J. R. Kiros, and G. E. Hinton, "Layer normalization," *arXiv preprint arXiv:1607.06450*, 2016.

[25] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[26] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[27] "Annotation tools https://haystack.deepset.ai/."

[28] Openai, "Openai/openai-cookbook: Examples and guides for using the openai api."

[29] Ftarlaci, "Ftarlaci/gpt2sqa: Fine-tuning gpt-2 small for question answering."