GPT: Origin, Theory, Application, and Future

Tianzheng Troy Wang
wtian@upenn.edu

Thesis Advisor: Professor Mitchell Marcus
mitch@cis.upenn.edu

Engineering Advisor: Professor Norman Badler
badler@seas.upenn.edu

ASCS CIS498/EAS499 Project and Thesis
School of Engineering and Applied Science
University of Pennsylvania
April 2021

TABLE OF CONTENTS

1 Introduction

Natural language processing (NLP) is a discipline within artificial intelligence and linguistics that is concerned with using computers to build language models that can interpret, analyze, and generate human language. More concretely, NLP aims to solve many important and widely applicable linguistic tasks such as speech recognition, natural language comprehension, natural language generation, and machine translation. Many generations of models have been developed to tackle the challenges of NLP. The state-of-the-arts techniques of NLP often leverage the power of neural network models.

Generative Pre-Trained Transformer (GPT) is a series of transformer-based deep learning language models that showcased superior performance in text generation, comprehension and other NLP tasks. GPT derives its exceptional capabilities from its unique design and massive size. There are numerous aspects of GPT worth analyzing, such as its architecture, training, performance, as well as real-world commercial applications and ethical implications.

In this paper, we will explore an extensive set of topics related to deep learning NLP, as well as providing in-depth insights into topics pertaining to transformer and GPT models in particular. In section 2, we will present the detailed design components of the transformer model and demonstrate its significance in deep learning language modeling. In particular, we will investigate the role of the attention mechanism in the transformer model. In section 3, we will introduce the GPT series model and discuss its architecture in relation to the transformer model. We will analyze its training process, fine-tuning, in-context learning, and performance. We will also examine the limitations, implications, ethics and risks associated with GPT. In section 4 and section 5, we will explore GPT's future development potential and survey its prospective business applications.

2 Transformer

2.1 Overview

The Transformer is a class of sequence-to-sequence deep learning model widely used in natural language processing [VSP+17]. Compared with relatively more traditional sequential models such as the recurrent neural network (RNN), the transformer takes advantage of the concept of attention and self-attention, which allows the model to directly look at and draw from the states of any preceding tokens in the input sequence in a learned way, irrespective of the distance between the tokens [VSP+17]. Intuitively, thanks to the attention mechanism, the learned model gains the ability to pay more attention to those parts of the speech that are more important to interpreting the meaning of the input, regardless of how far away or how ambiguous those parts might be, thereby significantly enhancing its performance. Additionally, the transformer model enables for a high degree of parallelization, which allows it to train on much larger datasets than before and retaining exponentially more parameters [VSP+17]. Though proposed only 4 years ago in 2017, the transformer has quickly gained attention from the academic community and establishes itself as one of the cutting-edge topics of natural language processing research. Many novel adaptions of transformers have been proposed since then, most notably, the GPT series models [RNS+18] [RWC+19][BMR+20]. Those more advanced models, including the switch transformer, a trillion-parameters model developed only three months ago and has already demonstrated superior performance, all lie on the foundation of the transformers [FZS21], which we are going to discuss below.

On a high level, the Transformer takes as input a sequence of texts, such as a natural language sentence, and then generates a sequence of output, such as a translated sequence of the input or a predicted sentence following the input, depending on the nature of the input. Inside the Transformer module, there is an encoder stack and a decoder stack, each containing a series of sequential encoders and decoders. The input text is first fed into an embedding layer and positional encoding layer, converting the text into vectors, which are then fed into the encoder stack, and more specifically, the encoder on the bottom most level. Each encoder takes as input the output from the encoder below it, and the encoder on the very top feeds its output to the decoder stack, which relays its output sequentially similar to the encoders. The decoder on the very top of the decoder stack feeds its output to the linear and softmax layer, before finally generating the output. Note that all the encoders share an identical structure, and all the decoders share another identical structure, yet every encoder and decoder have its own learned parameters [VSP+17].

In the following sections, we will provide deeper insights into how each of these components works. Below is a simplified schematic illustrating the high-level architecture of the transformer. At the end of this section, we will provide a more sophisticated schematic that shows a more comprehensive view of the transformer architecture after we've discussed the relevant concepts.

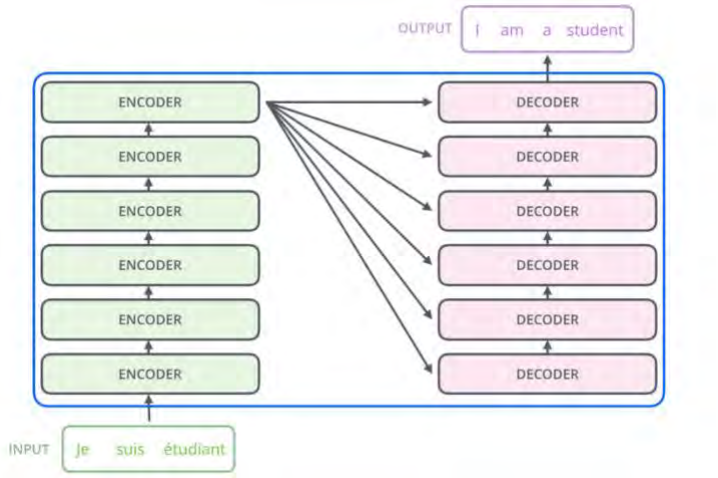

Figure 1. Transformer Architecture, Simplified Version. [Alammar18]

2.2 Encoder and Decoder

An encoder consists of two sequential sublayers: self-attention and feed-forward neural network. An encoder's input is first fed into the self-attention layer, then fed into the feed-forward neural network, which produces the output. A decoder, similar to an encoder, also contains a self-attention layer and a feed-forward neural network. The difference is that there is also an extra encoder-decoder attention layer, which is structurally just a self-attention layer, in between those two layers. The self-attention layer in the decoder is sometimes also called the masked attention for reasons that we will discuss later. There is also an add & normalization layer on top each self-attention layer and feed-forward neural network layer in the encoders as well as the decoders [VSP+17] [Alammar18].

We will discuss those details layer by layer.

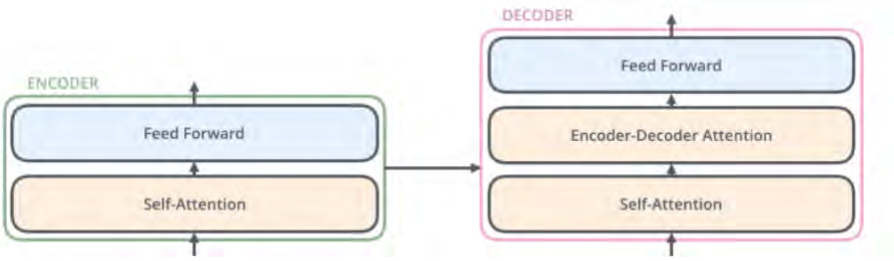

Figure 2. Schematics of a Transformer Encoder and a Transformer Decoder [Alammar18]

Embedding algorithm and positional encoding are used to convert the input sequence into a list of embedding vectors each of a certain size, a number that can be tuned as a hyperparameter. The size of the list, which is equal to the number of tokens in an input sentence, can also be tuned as a hyperparameter [Alammar18]. Intuitively, an input sequence can be interpreted as a natural language sentence and a token can be interpreted as a word. In practice, however, an input sequence is just a string of words with a certain pre-specified size and does not necessarily have to be separated by sentence endings in the natural language sense, such as a period. This size could be set as the size of the longest natural language sentence in our entire training set. Similarly, a

token does not have to be single word in the natural language sense [Alammar18]. We will discuss word tokenization more in detail.

This list of input embedded vectors converted from the input sequence is then fed to the bottom most encoder of the encoder stack. For the remaining encoders on top of the lowest encoder, their inputs are the outputs from the encoders one level below, respectively. It is also worth noting that in the encoder stack setup, each token goes through its own pathway all the way from embedding to the encoders, bottom to up. Although there are logical dependencies between tokens in the self-attention layer, each path for each individual token can be computed in a parallel fashion. This parallelization is one of the defining advantages of transformer models compared with traditional sequential models such as RNN [VSP+17] [Alammar18].

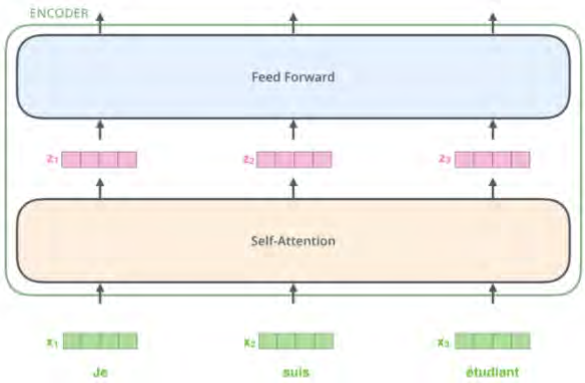

Figure 3. Parallelization within a Transformer Encoder. [Alammar18]

2.3 Embedding Algorithm

Transformer models, just like most other deep learning models, are number based models, and therefore, we need to use embedding algorithms to convert the natural language texts input into numbers, or vectors. This embedding process is a critical component of the transformer model and is the first step of the series of computations from input to output [VSP+17].

While we could use the brute-force solution, namely, to just assign each word in the vocabulary a single unique number, this encoding method does not provide us with much information beyond just the alphabetical order of words in the vocabulary. In addition, the numbers assigned would be arbitrarily large, and solely depends on the size of the vocabulary, which does not carry much meaning in itself. Alternatively, we could encode the words as one-hot vectors, where every word is represented by a vector of dimension equal to the size of the vocabulary and every position is 0 except for the single position that uniquely corresponds to the particular word, which becomes 1. Yet the dimension would go unfathomably high. Thus, a smarter approach of encoding natural language must be adopted to solve this problem [Kulshrestha19][MCC+13].

Embedding algorithms, and word embedding in particular, are a class of learning methods that map words into vectors, such that each word (a string) has its own real-valued vectorized representation in a pre-defined vector space (numbers), and more importantly, words with similar meanings have similar representations. In this way, each word can be represented using a dense distributed vector with tens or hundreds of dimensions, a dimensionality that is much lower than the size of the entire vocabulary. Essentially, every word becomes a point in a vector space, whose number features is much lower than the traditional vector representation of words such as using

one-hot vectors. The low dimensionality of embedded vectors and its learned nature are beneficial for its use in deep learning. Consequently, quantifying difference between words becomes easier. We can achieve that simply by using vector dot product to compute cosine difference. In contrast, if we are to use the simple one-hot vector, not only would we end up with a very high dimensional vector, but we would also lose the ability to encode the often times complicated similarity between words [Kulshrestha19][MCC+13].

One common approach for setting up a learning embedding algorithm is by creating an embedding layer, which is learned jointly with a neural network model on a particular task. Before training this layer, the entire input must be prepared so that every word input can be represented by a one-hot vector, which we explained earlier. The size of the embedding vector is pre-specified and initialized using small random numbers. Then this embedding layer is placed on the front end of the neural network and is then fitted using back propagation [Kulshrestha19][MCC+13].

More concretely, there are two methods of learning embeddings through an embedding layer. The first is called a skip-gram model. In this model, for each input word $w(t)$, meaning a word in position t in an input sequence w, we create a projection that predicts the neighboring words of $w(t)$. In other words, given $w(t)$, we want to train a network that predicts

$$w(t-k), w(t-k+1), \dots, w(t-1), w(t+1), \dots, w(t+k),$$

where k, a positive integer, is a hyperparameter that we could tune in order to adjust the window size of neighboring words. In fact, we are not interested in the results of this network prediction, since we already know the result, which is just the input sequence w. We set up this network to train the parameters for the projection, and this projection is our embedding for $w(t)$ [MLS13] [Kulshrestha19][MCC+13].

In other words, this is a process of supervised learning where we are only concerned with training and model parameters rather than using it to predict neighboring words. The underlying assumption to this approach is that words that appear in similar contexts should have similar meanings, and therefore should share similar embeddings. With sufficiently large training samples that cover the vocabulary and suitable embedding dimensions, we would end up with an embedding space where every word can be mapped to an embedding vector and similar words have embedding vectors with smaller angles in between [MLS13] [Kulshrestha19][MCC+13].

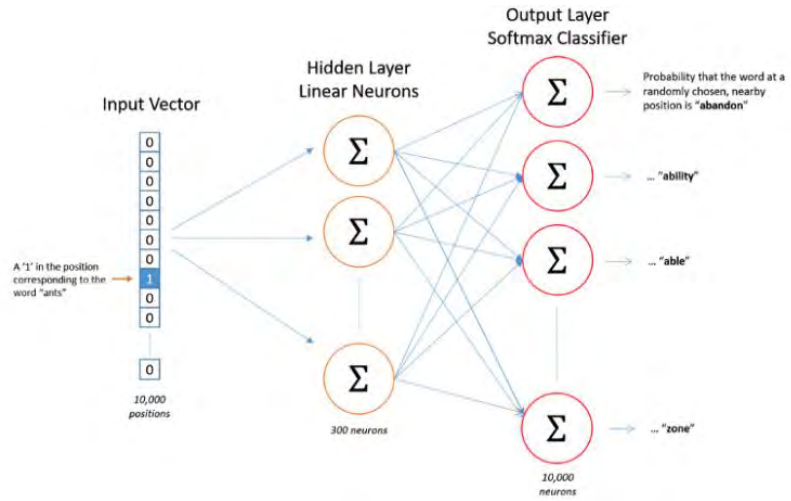

Figure 4. Architecture of Skip-Gram Model [McCormick16]

As we can see from this diagram, an input word is first converted into a one-hot vector. A one-hot vector for a word is a vector with dimension equal to the number of words in the entire vocabulary that this word belongs to $(1 \times V)$, and the position that corresponds to this word has value 1, while all other positions have value 0. This is arbitrarily but uniquely encoded. This one hot vector is then fed into a single hidden layer with dimension $(V \times E)$, where E is the dimension of the embedding, a hyperparameter that we can tune. This hidden layer contains linear neurons, and its parameters will eventually become the parameters we would like to use to generate word embeddings. But before that we need to train those parameters. The way to achieve this is to then feed the output of this hidden layer to the output softmax classifier, a layer of dimension $(1 \times V)$, where each neuron is a softmax function that converts the outputs from the hidden layer into a probability that the word this neuron corresponds to appears in the neighbors of the original input word. Then, we would use back propagation to train the parameters in one back pass for each particular set of input word and neighbors. We would calculate the errors vector, with dimension $(1 \times V)$, for each target word in the neighbors, and then do an element-wise sum to get the final errors vector. The weights of the hidden layer will be updated and therefore learned based on this errors vector. Note that once we decide our window size, we do not discriminate the words within the window based on its distance from the input word $w(t)$, and thus this window size should not be excessively large [MLS13] [Kulshrestha19].

Now it can be seen that the direction in which this learning process takes place is arbitrary. In fact, it can be reversed, which leads to the continuous bag of words model (CBOW). The overarching idea of training the projection such that it can be used to generate embedding mappings remains the same, but in CBOW, instead of using a word $w(t)$ to predict its neighbors, we use neighbors to predict w(t). Consequently, the dimension of our hidden layer remains the same. The dimension of our input vector and the activation function would change to reflect the reversal of network direction. The backpropagation training process is similar to that of the skip-gram model [MLS13] [Kulshrestha19].

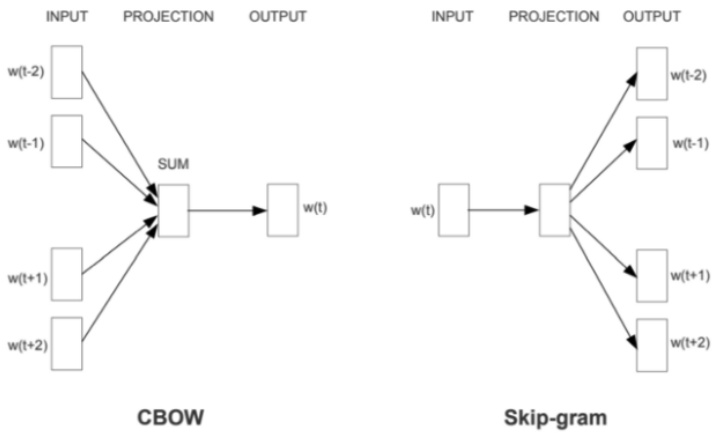

Figure 5. CBOW Model Compared with Skip-Gram Model. [MLS13]

2.4 Positional Encoding

Since our input embedding vectors are modeled irrespective to the order of the input wording sequence and there is no recurrence or convolution in our model, we need a positional encoding that accounts for the ordering information of the words in the input sequence. This is done by adding a positional encoding vector to every input embedding vector. The dimension of this positional vector is thus the same as the embedding vectors, and the values are determined by a function that assigns a unique set of vector values to represent every unique position in the input sequence. Intuitively, it is meaningful to include the positioning information because sequence order does matter in natural language. A sentence of natural language is always a list of word with a particular sequence, and never a set of words agnostic to sequence [VSP+17][Kazamnejad19].

There are a few requirements that an effective positional encoding method must meet. First and foremost, it must create a unique encoding for each time step. In other words, it must be a strict one to one mapping, thereby precisely reflecting the position. Secondly, it must be able to easily generalize to sentences with varying length, especially very long sentences, and its values should be bounded. In other words, the simple token distance counting method would not be ideal, since the count can be arbitrarily large and unwieldy for learning purposes. Thirdly, the distance between any pair of time steps should be consistent across sentences with different lengths. This implies that simply calculating the relative position within a sentence is not sufficient. And lastly, it must be deterministic [VSP+17][Kazamnejad19].

One common method of implementing positional encoding is through trigonometric functions, which is used in the case of transformers. The positional encoding is in the form of a d-dimensional vector, where d is the model dimension (dimension of the embedding vectors for input tokens). This makes adding the positional vectors to the embedding vectors possible. Now let t be the desired position in the input sequence, and $\overrightarrow{p_t} \in \mathbb{R}^d$ be its positional encoding, then the following function in the form $f : \mathbb{N} \to \mathbb{R}^d$ will give us the positional encoding.

$$\overrightarrow{p_t}^{(i)} = f(t)^{(i)} := \begin{cases} \sin(\omega_k . t), & \text{if } i = 2k \\ \cos(\omega_k . t), & \text{if } i = 2k+1 \end{cases}$$

where

$$\omega_k = \frac{1}{10000^{2k/d}}$$

It can be seen that the frequency of the sine and cosine functions decrease as the dimension increases, resulting in a wavelength interval going from 2π to $10000 \times 2\pi$. Here 10000 is a model hyperparameter, representing the length of the longest sentence possible in this setting. As a result, $\overrightarrow{p_t}$ becomes the following vector containing pairs of sines and cosines for each given frequency as shown below [VSP+17][Kazamnejad19].

$$\overrightarrow{p_t} = \begin{bmatrix} \sin(\omega_1 . t) \\ \cos(\omega_1 . t) \\ \\ \sin(\omega_2 . t) \\ \cos(\omega_2 . t) \\ \\ \vdots \\ \\ \sin(\omega_{d/2} . t) \\ \cos(\omega_{d/2} . t) \end{bmatrix}_{d \times 1}$$

This positional encoding is a vector where its element alternates between sine and cosine values, with geometric progression of changing frequency, which imitates the counting system we use from a trigonometric point of view. When we count numbers starting from 0, the more significant the digit, the exponentially less frequent it changes. The trigonometric functions with the high frequency represent the less significant digits and therefore changes more frequently. Similarly, those with low frequency represent the more significant digits and thus changes more frequently. The values of the trigonometric functions are bounded, just like our counting system, where the value in each digit is bounded, such as 0 to 9 in the case of decimal and 0 to 1 in the case of binary. The result of the positional encoding is visualized by the heatmap below. If we are to compare this encoding with a binary counting system, then we can intuitively interpret the blue half of the spectrum as 0 and the red half as 1. Each row is a positional encoding, and the first row encodes the first position, where every one of its digits is 0. As we move down, more digits start to change, starting from the least significant digits which are represented by the left most column, and exponentially to the right. Of course, this is only an intuitive explanation, so please refer to the trigonometric functions above for precise and formal interpretation. This way, we could encode the positional information for our input sequence while fulfilling the requirements that we listed earlier [VSP+17][Kazamnejad19].

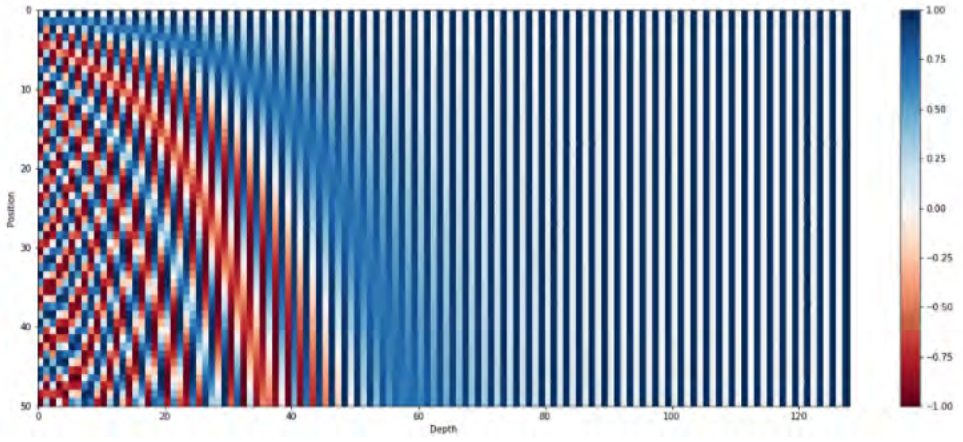

Figure 6. The 128-Dimensional Positional Encoding for a Sentence with the Maximum Length of 50. Each Row Represents the Embedding Vector $\overrightarrow{p_t}$. [Kazamnejad19].

2.5 Self-Attention

Self-attention is one of the key differentiating characteristics of the transformer model. It is a critical component that enables the Transformer to comprehend language contexts intelligently [VSP+17].

The objective of attention can be clearly illustrated by a simple example. Given an input such as "Professor Marcus gave some really good advice to Troy, one of his students, because *he* has extensive experiences in the academia." For a human reader, it is clear that the word "*he*" in the second half of the sentence refers to "Professor Marcus," instead of his student. But for a computer program, it's not so apparent. There are many such circumstances where grammatical ambiguities legally exist, such that rule-based and hard-coded logic would not be sufficient for effective language analysis and comprehension [VSP+17][Alammar18].

This is where self-attention comes into play. When the model processes each token input, self-attention enables the model to associate the meaning of the current token with other tokens, such as associating "he" with "Professor Marcus" in our previous example, in order to gain better knowledge of this current input. In other words, the transformer model learns how to pay attention to the context thanks to the self-attention mechanism. It turns out that natural language has a lot of sequential dependencies, and thus the ability to incorporate information from previous words in the input sequence is critical to comprehending the input [VSP+17][Alammar18].

Now, let's breakdown the process of computing self-attention. Before computing self-attention, each individual input token is first being converted into a vector using the embedding algorithm that we discussed earlier. Then, for each embedding, we calculate its query vector (Q), key vector (K), and value vector (V) by multiplying the embedding vector with three pre-trained matrices W^Q, W^K, W^V intended for calculating the three matrices respectively. Notice that the three vectors all have the same dimension, and it does not have to be equal with the dimension of the embedding vectors. The dimensions of the three matrices are the same, and they are all length of the embedding vectors by the length of the (Q), (K), and (V) vectors. Then, for each input embedding, we dot multiply its own Q vector with every other input embedding's K vector. At this point, for every input embedding, we have calculated a set of score corresponding to each and every embedding in the input sequence, including itself. Then for each of the embedding, we divide all its scores by the square root of the dimension of the key vectors for more stable gradients and pass the scores through the softmax function for normalization. After this, we multiply each V vector in the input sequence with its respective softmax score, and finally add up those weighted value vectors. This resulting sum vector is the self-attention of this particular input embedding [VSP+17]][Alammar18][IYA16].

Although we described the process in terms of vectors, in practice it is implemented by means of matrices. This is because the computation process for each vector independent and identical. We would stack our input embeddings as rows in an input matrix, multiply this matrix with learned weight matrices W^Q, W^K, W^V and get (Q), (K), and (V) vectors respectively, feed the three resulting matrices into the softmax function as

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

where $Attention(Q, K, V)$ becomes the resulting matrix of self-attention that will be passed to the feed-forward neural network layer[VSP+17]][Alammar18]. We will explain the softmax function

more in detail in the later linear and softmax section. The flow of the above computation is represented by the following chart.

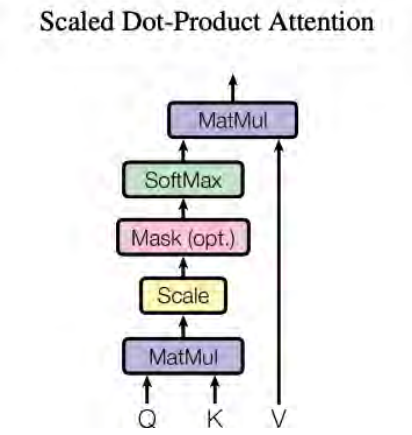

Figure 7. Scaled Dot-Product Attention. [VSP+17]

2.6 Attention Heads and Multi-Heads

One problem of the self-attention layer is that by only using a single set of trained matrices Q, K, and V, the self-attention could be dominated by just one or a few tokens, and thereby not being able to pay attention to multiple places that might be meaningful. Therefore, by using multi-heads, we aim to linearly combine the results of many independent self-attention computations, and thereby expand the self-attention layer's ability to focus on different positions [VSP+17] [Alammar18].

More concretely, we use multiple sets of mutually independent (Q), (K), and (V) matrices, each being randomly initialized and independently trained. With multiple (Q), (K), and (V) matrices, we end up with multiple resulting vectors for every input token vector. Nonetheless, the feed-forward neural network in the next step is designed to only accept one vector per word input. In order to combine those vectors, we concatenate them into a single vector and then multiply it with another weight vector which is trained simultaneously. Formally, this multi-head attention is defined as

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

$$\text{where } head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$$

Where the projections are parameter matrices

$W_i^Q \in \mathbb{R}^{d_{model}\times d_k}, W_i^K \in \mathbb{R}^{d_{model}\times d_k}, W_i^V \in \mathbb{R}^{d_{model}\times d_k}$ and $W^O \in \mathbb{R}^{hd_v\times d_k}$.

In the original work of transformer, h, the number of parallel attention layers, was chose as 8, and $d_k = d_v = d_{model}/h$ was chosen as 64. This is just one possible choice for hyperparameters [VSP+17][Alammar18].

Hence, we are able to combine the results from multiple attention-heads to one aggregated result vector (for each input word) as illustrated by the schematic below, and then pass it to the feed-forward neural network [VSP+17][Alammar18].

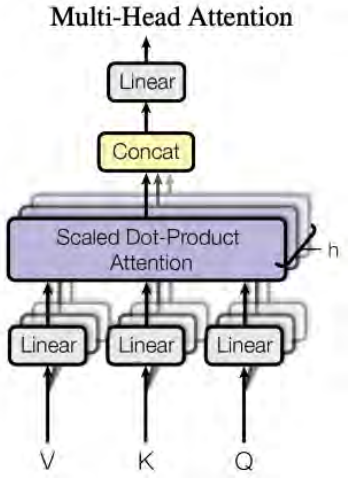

Figure 8. Multi-Head Attention. [VSP+17]

2.7 Reasons for Using the Attention Mechanism

Self-attention demonstrates appealing performance metrics as compared to other sequence to sequence mapping methods such as recurrent neural network (RNN) or convolutional neural network (CNN) [VSP+17].

For example, sequential operation complexity for self-attention is constant, since all input tokens within an input sequence are processed in the self-attention layer simultaneously, and thus could be easily parallelized. RNN, in contrast, has complexity of sequential operations linear to the sequence length. This is because by the recurrent design, one token must be processed after its previous token was finished processing [VSP+17].

The maximum path length for self-attention is also constant and independent with the input length. Recall that in the self-attention layer, the lengths of paths generated by each step, including matrix multiplication with (Q), (K), and (V) vectors, scoring using (Q) and (K), vectors, and score summing, are independent with the input lengths. In contrast, RNN has maximum path length linear to input sequence length, again, due to its recurrent design. CNN, on the other hand, has maximum path length of log of input sequence length with base kernel size. This is because the max-pooling and convolution steps depend on the input size and kernel size. Shortening the length of path between any two positions in the input and output sequence increases a model's capability of learning long range dependencies, and thus self-attention is the best in this sense [VSP+17].

In terms of complexity per layer, self-attention is $O(n^2 d)$ whereas RNN is $O(nd^2)$. Consequently, self-attention is faster than RNN when the sequence length is smaller than the model dimensionality, which, according to the developers of transformer, is often the case thanks to advanced models in sentence representation (we will discuss one such model, the byte-pair encoding, later in this paper). Note that for tasks that have very long sequences of input such that not even byte-pair encoding could reduce the size of the input length, we would use a restricted version of self-attention and thereby restricting the model to only look at a neighborhood of size r around the desired output position in the input sequence. The complexities we discussed above are illustrated in the following table [VSP+17].

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

Table 1. Complexity per layer, sequential operations complexity, maximum path lengths of self-attention as compared with RNN and CNN [VSP+17].

In addition, self-attention by design is more well-suited for creating interpretable models. Given a trained self-attention layer, we can easily read which previous words in the input got higher attention, and which ones got lower attention. We can also easily quantify the difference in attention by comparing the weights. These comparisons make intuitive senses. For example, we could train a self-attention layer on the input sentence "Professor M is able to give some really good advice to T, one of his students, because *he* has extensive experiences in the academia." Then we could examine the self-attention of the italic word "*he*". If the self-attention of this word assigns higher weight to "students" than "Professor", then we clearly know the model is incorrectly interpreting this sentence. In contrast, it becomes a lot more difficult make such direct and intuitive interpretations based on the parameters in other models such as a recurrent neural network or a convolutional neural network, since most of their parameters are in the hidden layers that more or less act like a black-box and it is considerably more difficult to make sense of one particular parameter within a hidden layer. We believe that interpretability is an important trait of a model, and thus, self-attention demonstrates clear advantage from this perspective [VSP+17].

2.8 Feed-Forward Neural Net

feeds its output, an n by 1 vector for each input token where n is the dimension of the embedding, to the neural network layer. Each neural network layer consists of two pretrained matrices, the first one having dimension $(n \times 4n)$, and the second $(4n \times n)$. The input vector relayed from the self-attention layer is first multiplied with the first matrix in the neural network W_1, resulting in a $(1 \times 4n)$ matrix, which is then multiplied with the second matrix W_2, resulting in a $(1 \times n)$ matrix. This is equivalent to a vector with dimension n, which is uncoincidentally the same as the original input of the neural network. Note that the hyperparameter of 4 is rather arbitrary. The developers of the transformer model did not explain the reason for picking this particular hyperparameter [VSP+17].

Below is the formal representation of the feed-forward neural network that we just discussed. Note that there is also a ReLU activation in between the two multiplications.

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

This transformation is the same across different positions in the same layer, yet each layer has its own unique parameters [VSP+17].

2.9 Layer Normalization

Note that the self-attention sub layer and the feed forward neural network sublayer do not directly pass their output to the next stage. Instead, they pass their output to an add & normalize layer. Conceptually, layer normalization belongs to a class of normalization methods aimed to normalize the activities of neural networks so as to speed up training time and reduce computational cost. More specifically, it addresses the problem that gradients with respect to the weights in one given layer are highly dependent on the outputs from the previous layer, especially if these outputs change in a highly correlated way, resulting in a covariate shift [BKH16].

We calculate the mean and variance of each sample input, and then normalize each sample such that the elements in the sample have zero mean and unit variance, and finally we linearly scale and shift the input with two learned parameters, bias and gain. Note that unlike the more commonly used batch normalization, layer normalization performs the exact same computation both at training and test times. Empirically, layer normalization works well for RNN and transformers [VSP+17][BKH16].

2.10 Linear and Softmax Layer

In the very first step of the model, we have converted the input word tokens into embeddings, and since that point on, the model has always been dealing with embedding vectors. Hence, the model has to convert the vectors back into words in natural language in the very last step. A linear layer, which is a connected neural network, is being multiplied with the last output from the last decoder block, creating a logits vector, which is of the size of the vocabulary. In other words, the decoder output assigns a logit value to every word in the given vocabulary. Then, the softmax function turns the logit vector into a probability vector [VSP+17][IYA16][IYA16].

Note that softmax, or the normalized exponential function, is a function the maps a real number vector to a vector with numbers in the range 0 to 1. Formally, the softmax function

$$\sigma : \mathbb{R}^K \to [0,1]^K$$

is defined by

$$\sigma(\boldsymbol{z})_i = e^{z_i} / \Sigma_{j=1}^{K} e^{z_j} \text{ for } i = 1, \dots, K \text{ and } \boldsymbol{z} = (z_1, \dots, z_k) \in \mathbb{R}^K.$$

We then choose the word with the highest probability as our final model output. We could alternatively add certain degree of randomness to the model by sampling among the words with top-k probability, where k is a hyperparameter with positive integer value. This way we avoided the scenario where the model keeps outputting the exact same output with the highest probability while neglecting other output with slightly lower probabilities yet are nevertheless still meaningful [VSP+17][IYA16][Alammar18].

2.11 Words vs Tokens: Byte Pair Encoding

Notice that in our discussion of the transformer model, we always use the term input token rather than input word. That is because words can be tokens, but tokens are not necessarily words. In fact, word is one specific way of tokenization, namely, using white spaces in natural language. Nonetheless, this particular tokenization method is not efficient in terms of information. One obvious example would be the use of comparatives and superlatives. If we count “big”, “bigger”,

and “biggest” as three distinctive words/tokens, we fail to incorporate the underlying relation among the three words. Even if the model manages to learn this particular relationship, it wouldn’t be able to easily apply to another scenario with the exact same logic but different appearance, such as “small”, “smaller”, and “smallest”. In fact, “er” and “est” should be compartmentalized into tokens on their own because they do have meanings. Now, it is impossible and inefficient to label all of those instances manually [Jaswal19][Philip94].

Thus, we use the method of byte pair encoding to systemize this effort. In our previous example, we would first count the number of occurrence of substrings of character length 2. We would end up with “er” and “es”, both of which would be counted as one single token. In the next cycle, we have “est” as among the highest occurrence substrings, and thus “est” would be counted as one single token. Similarly, we would also include “small” and “big” as our tokens. We also implicitly include the stop token “</w>” in this process, since the “er” substring in “bigger” and “smaller” appearing in the end of a word most likely do not share underlying meanings with “er” not appearing in the end of a word, such as “erase”. Since we do not interfere with the sequence of characters, decoding would be simple. Nonetheless, the encoding process is costly, and as a result, this byte-pair tokenization scheme is usually pre-trained and stored in a dictionary. Note that this process is deterministic and does not involve in learning [Jaswal19][Philip94].

Now that we’ve covered all the main components of the transformer model, it would be more suitable to use the following more comprehensive schematic for the transformer.

2.12 Recap

Now that we’ve covered all the main components of the transformer model, it would be more suitable to use the following more comprehensive schematic for the transformer.

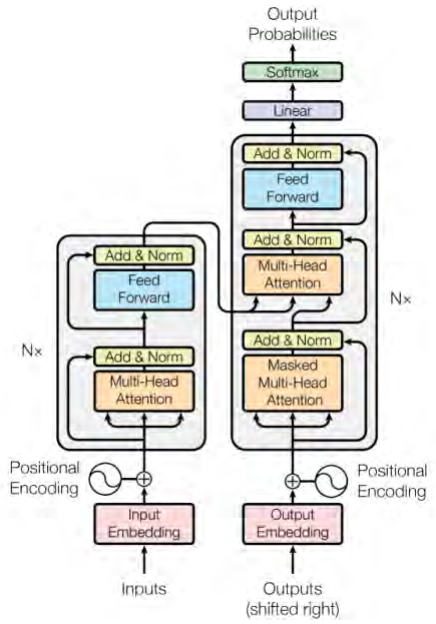

Figure 9. Transformer Model. [VSP+17]

3 GPT

3.1 Overview

GPT is a novel adaption of the baseline transformer model that we discussed in the previous section. It shares the main objectives of the transformer model, and inherits most low-level components of the transformer, such as the embedding algorithms, positional encoding, architecture of decoders, attention & self-attention mechanism, layer normalization, feed-forward neural network, linear & softmax layer, and so on [RNS+18]. Hence, what we introduced in the previous section about the transformer is still highly relevant for the GPT, and we will not repeat those parts that are more or less identical with the baseline transformer.

Nonetheless, GPT has many unique distinctions. One of such key distinctions is that GPT adopts a groundbreaking training methodology. Although structurally GPT is still based on transformers as we discussed (the "T" in GPT stands for transformer), it distinguishes itself by introducing the generative pretraining process ("GP" in GPT). This is based on the rather obvious fact that compared with labeled text, unlabeled text corpora, such as raw texts online, are a lot more abundant. Accordingly, while traditional transformers rely on labeled data for training, GPT can take advantage of the vast unlabeled texts thanks to the generative pretraining on a diverse corpus of unlabeled text, which would eventually greatly enhance its performance. In addition to this, GPT also allows for task-specific fine-tuning with labeled data. The assumption is that this would further enable GPT to gain domain knowledge that aligns with the specific tasks required, on top of the already powerful task-agnostic performance of GPT [RNS+18]. In fact, in the case of GPT3 (the third generation of GPT), its parameters and training corpus are so large, and its task-agnostic performance is so good, such that fine-tuning is no longer essential. Instead, providing a few examples in the input prompt, which is also called few-shots in-context learning, is sufficient for most tasks [BMR+20].

Another architectural difference between GPT and traditional transformers is that instead of relying on both encoders and decoders, GPT gets rid of the encoders and instead derives its performance from stacking many decoders. Additionally, the sheer scale of GPT models is groundbreaking in itself. GPT models have exponentially more parameters and use exponentially larger training sets than the original transformer model. The original version of GPT comes with 150 million parameters [RNS+18]. GPT2 has 1.5 billion parameters, while GPT3, the latest version of the GPT series models, has a mind blowing 175 billion parameters [RWC+19] [BMR+20]. The following schematic illustrates the architecture of the GPT model and visualizes what we discussed in the two paragraphs above. We will discuss the unique features of GPT in the coming sections.

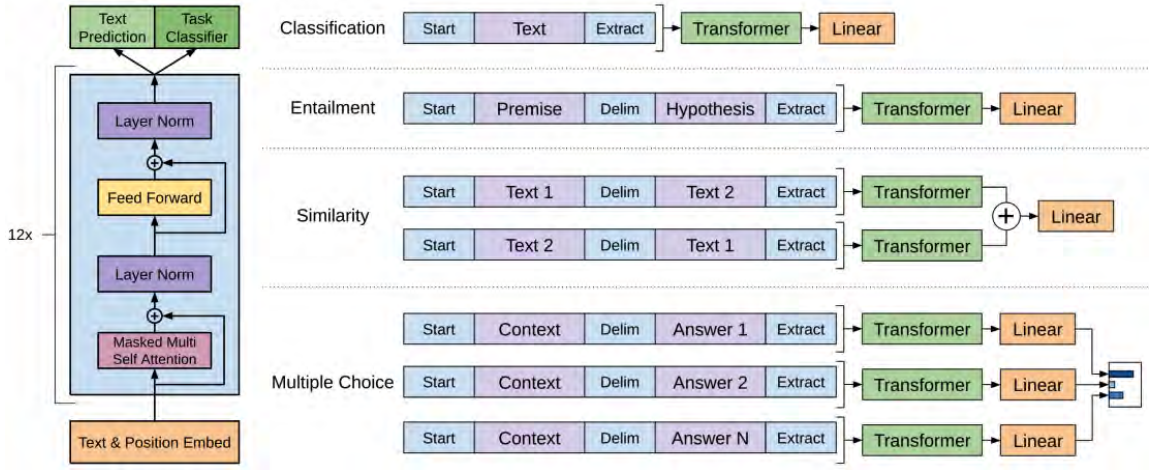

Figure 10. Architecture of the GPT Model [RWC+19].

3.2 Decoder Only Architecture and Masked Self-Attention

GPT models, regardless of the versions, all share the same decoder only architecture. Recall that in the transformer model that we discussed, there is a stack of encoders which takes in the input and feeds the intermediate output to the stack of decoders, which then eventually generates the output. In GPT, however, we eliminate the encoders and directly feed the input to the decoders stack. Note that we still retain the input embedding and positional encoding which together pre-process our input text and feed the processed vectors to the bottom most level of decoders. We also retain the linear and softmax layer that post-process the output vectors from the topmost level of decoders and convert them into texts as our formal final output [RNS+18].

Decoder is a variation of encoder in terms of architecture. The main difference is that in decoders, instead of using self-attention that we discussed, we use masked-self-attention. Previously, we allowed a position in the self-attention layer to look at tokens both to its left and to its right. Now, the masked self-attention only allows a position to look at tokens to its left. In other words, masked self-attention masks all the tokens in the future by setting the values in the respective vectors to negative infinity. The model could only look at words that already appeared in the input sequence for every given word [RNS+18].

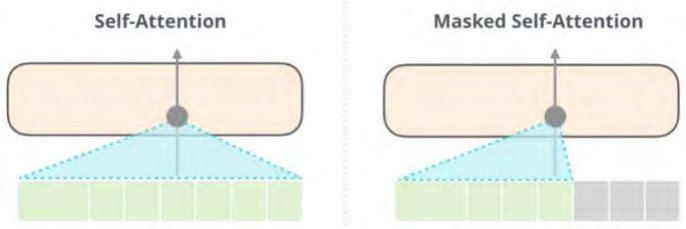

Figure 11. Self-Attention vs Masked Self-Attention [Alammar19].

Formally, the process can be written as follows. Frist we are given

$$U = (u_{-k}, \dots, u_{-1}),$$

the context vectors of input tokens. Then we apply the embedding matrix W_e and positional encoding matrix W_p to U , and get the input vector to the bottom level of the decoders, which is

$$h_0 = UW_e + W_p.$$

Then, we pass this vector along through the layers of decoders, which is

$$h_l = transformerblock(h_{l-1}) \; \forall i \in [1, n],$$

where n is the number of decoders in the decoders stack, and the transformer block is the same as we discussed except that we use masked self-attention. Finally, we convert the vector output from the last decoder to the final probability vector output, which is

$$P(u) = softmax(h_n W_e^T)$$ [RNS+18].

3.2 Training

The training process for GPT is not unique. As long as the end results demonstrate reasonable performance, one does not have to follow any particular training routine. That said, a few training frameworks have been proposed to better take advantage the potential of the GPT model. One common framework consists of two stage. The first stage learns a high-capacity language model on a large corpus of text. The second stage fine tunes and adapts the model to a discriminative task with labeled fine-tuning data [RNS+18].

3.2.1 Unsupervised Pre-Training

Similar to any other generic language model, our transformer model takes as input a series of text and aims to generate an output text given the input. Now this process could be applied to translation, in which the input is the source language, and the output is the target language, or text generation, in which the input is the prompt, and the output is the prediction, or any other language tasks. The output is generated based on the conditional distribution on the given input, and a set of pre-trained parameters. Once trained, the model generates an output by maximizing the likelihood of this conditional distribution. Now the question is how we should train this set of parameters. The unsupervised pre-training approach suggests that with a large enough training corpus, we could train the model iteratively, since we could easily breakdown the input into a context window (input) and a known output (the text after the context input). For example, if our training corpus is “roses are red” and our context window is two words, then we could train our model on the input “roses are” and output “red”. This process is unsupervised and does not require labeled text as training input. Recall that in real training, we breakdown the input into embeddings, rather than simply separating the input texts by words or blank space [RNS+18][RWC+19].

Formally, the model takes as input a corpus of tokens

$$U = \{u_1, \dots, u_n\}$$

and then maximize the likelihood

$$L_1(U) = \Sigma_i \log P(u_i | u_{i-k}, \dots, u_{i-1}; \Theta).$$

Here k is the size of the context window, P is the conditional probability modeled using our transformer model with parameters Θ. The parameters are trained using stochastic gradient descent [RNS+18][RWC+19].

Taking GPT3 as an example: The Common Crawl dataset, WebText dataset, two internet-based books corpora, and English language Wikipedia were used as training datasets for GPT3. Common Crawl was filtered to improve its average quality. The following table shows the detailed composition of training data of GPT3 [BMR+20].

Dataset	Quantity (tokens)	Weight in training mix	Epochs elapsed when training for 300B tokens
Common Crawl (filtered)	410 billion	60%	0.44
WebText2	19 billion	22%	2.9
Books1	12 billion	8%	1.9
Books2	55 billion	8%	0.43
Wikipedia	3 billion	3%	3.4

Table 2. Datasets Used to Train GPT3 [BMR+20].

The following plot shows the training curves for GPT3 models of different parameter size. Model performance was measured during the training process on a deduplicated validation split of the training set. The gap between training and validation performance, as measured by cross-entropy loss, does not grow significantly with increasing model size and training time, suggesting that the model was not overfitted [BMR+20].

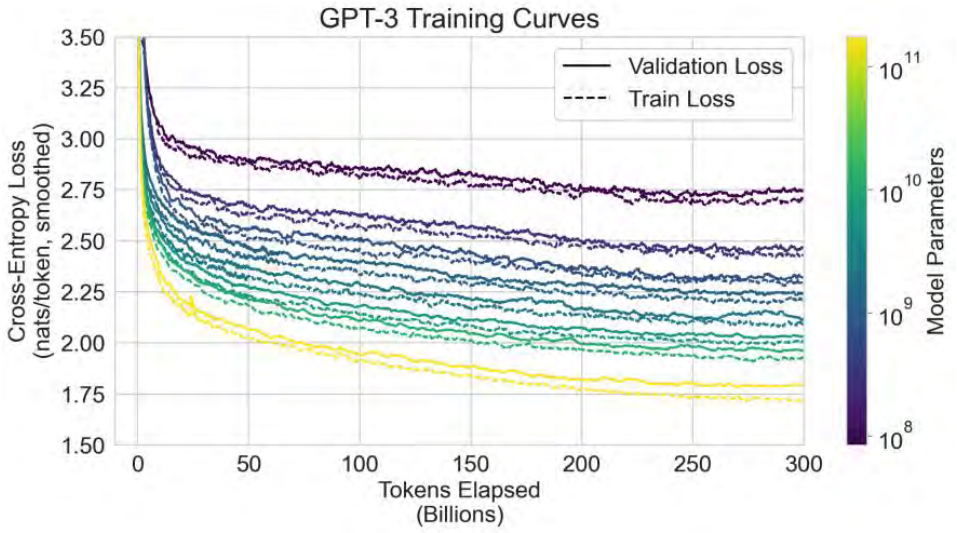

Figure 12. GPT3 training curves [BMR+20].

3.2.2 Supervised Fine-Tuning

After the previous stage, our task agnostic transformer model is equipped with trained parameters for generic tasks. Technically, no more training is necessary since we already filled our entire

model with trained parameters. Yet empirically, task agnostic transformer models do not perform well on tasks that require specialization, which we will see later [RNS+18]. Therefore, fine-tuning (FT) approach was proposed to further adapts our task agnostic model to specific tasks. FT involves in updating the weights of our earlier pre-trained model on a tailored supervised dataset specific to our target task. Compare to our training corpus in the pre-training stage, fine-tuning training datasets are typically smaller, with thousands to hundreds of thousands of labeled examples. The main advantage of this approach is that it boosts many performance benchmarks of our model especially when the tasks are specific (meaning that the particular form and scope of the task are not commonly observed in the pre-trained corpus). Yet this FT also has disadvantages, such as the need for a new dataset for every specific task, the potential for poor out-of-distribution generalization, and potentially resulting in an unfair comparison with human performance [RNS+18][BMR+20][BV19].

Formally, we are given a labeled dataset C, which consists of a sequence of input tokens, $x^1, \dots, x^m$, and a label y. This labeled data does not change every parameter of the model. Instead, it passes through our pre-trained model and get the final transformer block's activation h_l^m, which is then fed into ad added linear output layer with parameters W_y to predict y, according to the conditional probability

$$P(y|x^1, \dots, x^m) = softmax(h_l^m W_y).$$

Then we would maximize this objective

$$L_2(C) = \Sigma_{(x,y)} \log P(y|x^1, \dots, x^m).$$

The researchers who proposed including language modeling as an auxiliary objective to this FT improves generalization and accelerates convergence. Specifically, the following objective was proposed to be optimized

$$L_3(C) = L_2(C) + \lambda\, L_1(C)$$

where λ is a weight factor. Consequently, we only tune the embeddings and the added linear output layer W_y during this fine-tuning stage, which means that fine-tuning is not as costly as pre-training [RNS+18].

3.2.3 GPT3 and In-Context Learning

GPT3 is the third generation of the GPT series models that was released a year ago in June 2020 and is the most advanced version of GPT as of now (April 2021). The defining feature of GPT3 as compared with previous GPT models is its unprecedented size. GPT3 has 175 billion parameters, which is exponentially larger than GPT2 and the original GPT. While the underlying architectures are identical, GPT3's exponentially large scale enables it to be trained and used in a novel approach [BMR+20]. More specifically, the core strength of GPT3 is that generally speaking, GPT3 no longer requires fine-tuning. Instead, GPT3 takes advantage of in-context learning, or more specifically, the few-shots and one-shot learning methods [BMR+20][RWC+19]. Below, we will discuss more in details about GPT3's features, such as in-context learning, evaluation, performance, scope, risk, ethics, and limitation.

It is worth clarifying that GPT3 is itself a series of models rather than a single model. By "GPT3", we refer to the largest and most sophisticated model of GPT3 models. The table below shows the specification of all the GPT3 models, all of which were trained for a total of 300 billion tokens [BMR+20][RWC+19].

Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B or "GPT-3"	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}

Table 3. Specifications for GPT3 Models [BMR+20].

3.2.4 Few-Shots (FS)

As opposed to fine-tuning where we actually modify the model parameters, few-shots learning does not change the model itself. Instead, the model is simply given a few demonstrations of the task at inference time as conditioning. In other words, we add demonstrations to the input before our actual input, and thereby only modifying the input but not the model. One demonstration is called one-shot, and few-shots typically involve in 10 to 100 demonstrations, depending on how many can be fitted into the context window (since the input out of the context window would be useless for practical purposes). The advantage of few-shots learnings is that labeled fine-tuning training data for supervised learning, which is often costly to acquire, is no longer required. Also, the model has lower tendency to learn an overly narrow distribution from the fine-tuning dataset. But the disadvantage is that the result of few-shots learnings might not be as good as fine-tuning when it comes to certain specific tasks [BMR+20][RWC+19].

3.2.5 One-shot

This is similar to few-shots except that the number of demonstrations is limited to just one. The significance of this type of learning is that this is commonly the way a task is demonstrated to humans. For most of the language tasks, humans do not need dozens of examples to make sense of the pattern. Yet one example is often helpful for humans to understand the task [BMR+20].

3.2.6 Zero-shot

Similarly, in zero-shot, no demonstration is allowed, and as a result, the model is directly given a natural language instruction for the task. Clearly, this is the most difficult setting for not only transformer models but also humans, since the task description itself without supporting examples could be ambiguous. Yet this approach is the least costly and is not subject to bias in the shots that are given. Additionally, for many common tasks such as translation, human do not need an example understand what should be done, and therefore, a language model is expected to do the same [BMR+20].

Note that earlier versions of GPT are also capable of in-context learning, and GPT3 is also capable of fine-tuning. That we discuss in-context learning under this GPT3 section is not because previous GPT versions are incapable of such learning, but rather GPT3 demonstrates the best results of in-context learning. In fact, it was found that larger models make increasingly efficient use of in-context information, as demonstrated by the learning curve plot below [BMR+20].

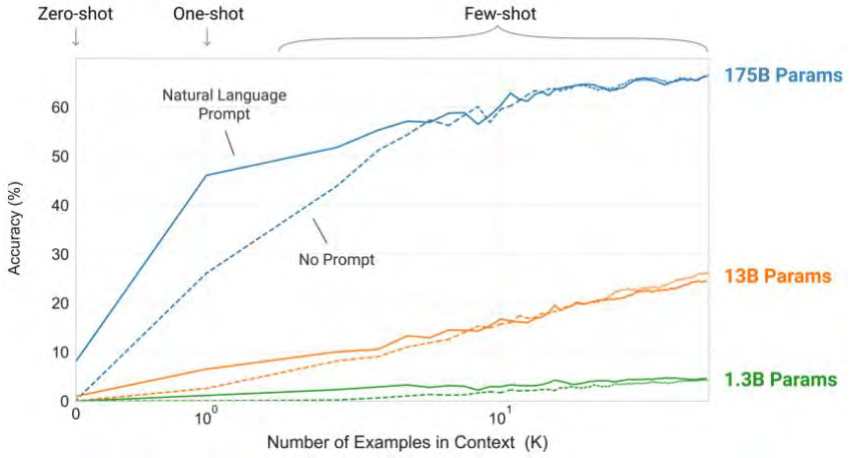

Figure 13. In-Context Learning Curves for GPT Models with Different Sizes [BMR+20].

4.2.7 Retrieval-Augmented Model for In-Context Learning

In spite of GPT3's powerful capabilities of in-context few-shots learning without the need of fine-tuning, its performance is not consistent when given different in-context examples. There are some in-context examples that work better than others to fully exploit the potential of the GPT3 model. Traditionally, task relevant in-context examples are chosen randomly from the training set. While simple to execute, this random sampling approach was found to be not the optimal solution for few-shots learning. Ideally, the optimal solution could be found by performing a brute-force combinatorial search of the training set, yet practically this is too computationally expensive. Thus, a clever way of sampling in-context learning examples called the retrieval-augmented model was proposed to improve the effectiveness of few-shots learning [LSZ+21][GLT+20].

This retrieval-augment model is derived from the observation that in-context examples that are chosen to be more closely aligned with the test example in terms of embedding yield significantly better few-shots learning results. This finding makes intuitive sense, since it is not surprising for the GPT model to come up with better results when given better fitting examples to learn from. So, the task of improving in-context examples becomes searching for examples in the training set that are most closely related to the test example. Naturally, this objective leads to the proposal of a *k*NN (*k*-th nearest neighbor) augmented in-context example selection method. The first step is to convert the test example into a vector representation using an encoder. Then for each embedded test example, we retrieve its nearest k neighbors from the training set, based on the *k*NN algorithm which selects the k nearest vectors in the embedding space, given a similarity measure. Afterwards, those k examples chosen from the training set are combined with their targets and sent to the GPT3 model as its input. The key to implement this approach depends on a suitable retrieval module that provides effective embeddings for input test examples as well as those in the training set [LSZ+21] [GLT+20].

3.3 Model Performance

Since the performance of GPT3 is significantly superior to GPT2 and the original GPT, we will focus our discussion about performance on GPT3 models.

GPT3 achieved a zero-shot perplexity of 20.5 on the Penn Tree Bank (PTB) dataset, which is 15 points lower than the previous state-of-the-art (SOTA) record broken by GPT2. Note that PTB is a parsed text corpus that annotates syntactic or semantic sentence structure that once revolutionized computational linguistics. Now with the advancement of exponentially larger scale deep learning models, PTB has a new role. It is now commonly used for evaluating and testing deep learning language models thanks to it is meticulously structured design. More importantly, PTB predates internet data, which is often included in the training set of modern-day language models. Thus, testing using PTB avoids the problem of testing directly with training data [BMR+20][MKM+94].

GPT3 also did very well in the tests using the LAMBADA dataset, which tests the modeling of long-range dependencies in text by asking the model to fill out the last word of a given sentence or paragraph. It is shown that GPT3 set record for both accuracy and perplexity. As demonstrated by the plot below, with few-shots learning, the accuracy of the 175 billion parameters full version of GPT3 reaches 86.4% where the previous SOTA achieved only 68%. The accuracy of GPT3 almost approaches that of human. In terms of perplexity, GPT3 achieved 1.92, far lower than the previous SOTA of 8.63. The plot also demonstrates that increasing number of parameters boosted the performance of this test, and few-shot learning also improved both accuracy and perplexity when controlling for in context learning [BMR+20][PKL+16].

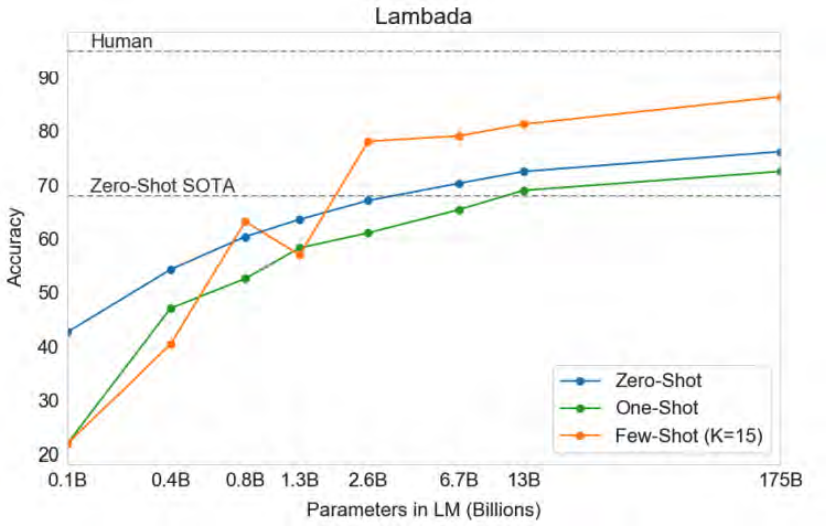

Figure 14. Accuracy Result of GPT3 Tested on the LAMBADA Dataset [BMR+20].

GPT3, with only few-shots learning, also did very well even when compared with SOTA fine-tuned for specific tasks. Testing on the TriviaQA dataset (a reading comprehension dataset containing over 650k question-answer-evidence triples), GPT3 was able to outperform fine-tuned SOTA by 3.8%. It is also once again proven that with identical architectures, more parameters increase performance, and few-shots learning increase performance [BMR+20][JCW+17].

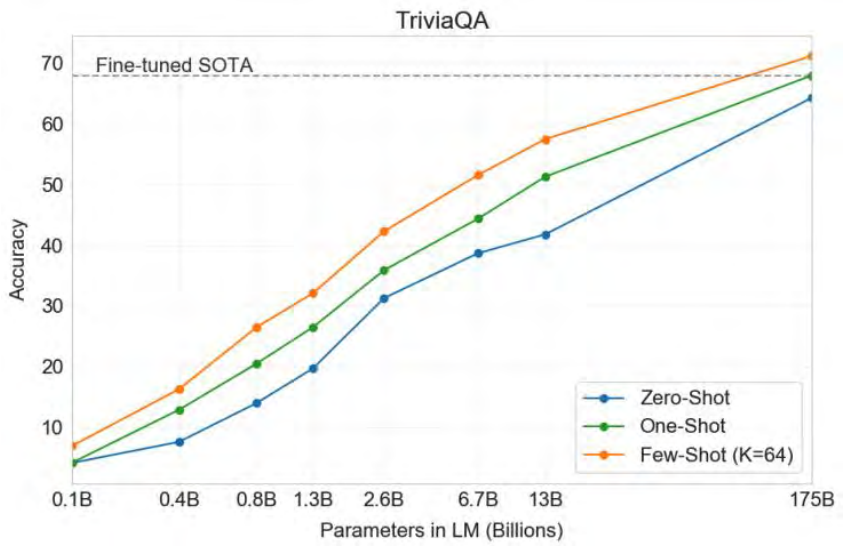

Figure 15. GPT3's performance on the TriviaQA dataset [BMR+20].

GPT3 was also tested on the SuperGLUE benchmark. SuperGLUE offers a single-number metric that summarizes progress on a diverse set of language understanding tasks on a standardized collection of datasets that can be compared using different language models. The results showed that with few shots learning, GPT3 was able to surpass fine-tuned BERT++ and BERT Large models. Nonetheless, GPT3 was unable to beat human or specifically fine-tuned SOTA for SuperGLUE. As shown in the plot below, few-shot almost always outperforms one-shot or zero-shot, and the performance increases with higher number of parameters. This observation is consistent with our previous findings [BMR+20][WPN+19].

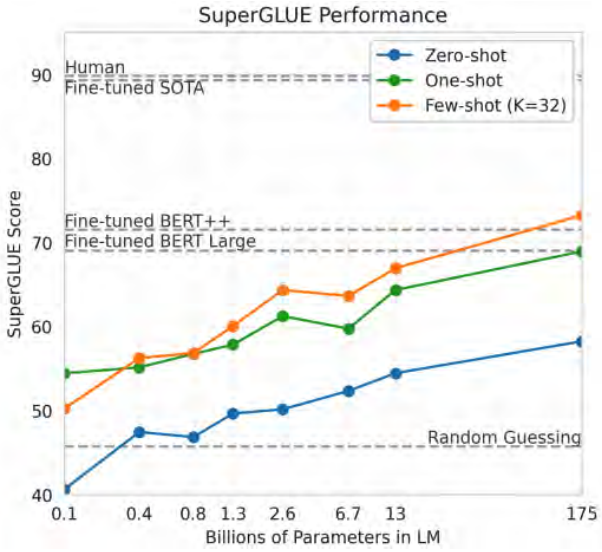

Figure 16. GPT3 performance on SuperGLUE [BMR+20].

In addition, performance of few-shot in-context learning increases with more in-context examples, or more shots. Yet this effect plateaus as we pack increasingly more in-context examples [BMR+20]. This effect is shown in the plot below.

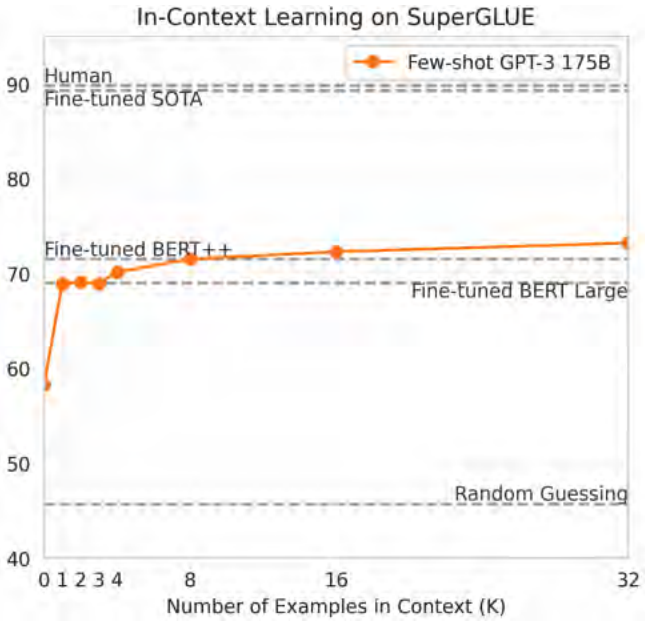

Figure 17. GPT3 in-context learning (few-shots) on SuperGLUE [BMR+20].

Another interesting metric of evaluating language models is arithmetic test. Although mathematics does not strictly fall into the scope of language, testing language models using arithmetic does evaluate the logical reasoning capabilities of models. And given that human can process arithmetic questions with text input only, it is a fair test for advanced language models such as GPT3. Note that GPT3 does not have any hard-coded arithmetic modules, and there is not APIs such as a calculator control panel for GPT3, so the model must comprehend the arithmetic questions in the form of plain text and answer the questions in the form of plain text without any task-specific training [BMR+20].

Researchers devised randomly generated addition, subtraction, and multiplication tests up to five digits, in the form of "Q: What is 48 plus 76? A: 124." The plot below shows the accuracy of such arithmetic tests with different parameter numbers. Not surprisingly again, models with higher number of parameters perform significantly better. In the case of two- or three-digits addition and subtraction, GPT3 was able to answer with almost 100% accuracy. Yet the accuracy for larger number computation is not nearly as good [BMR+20].

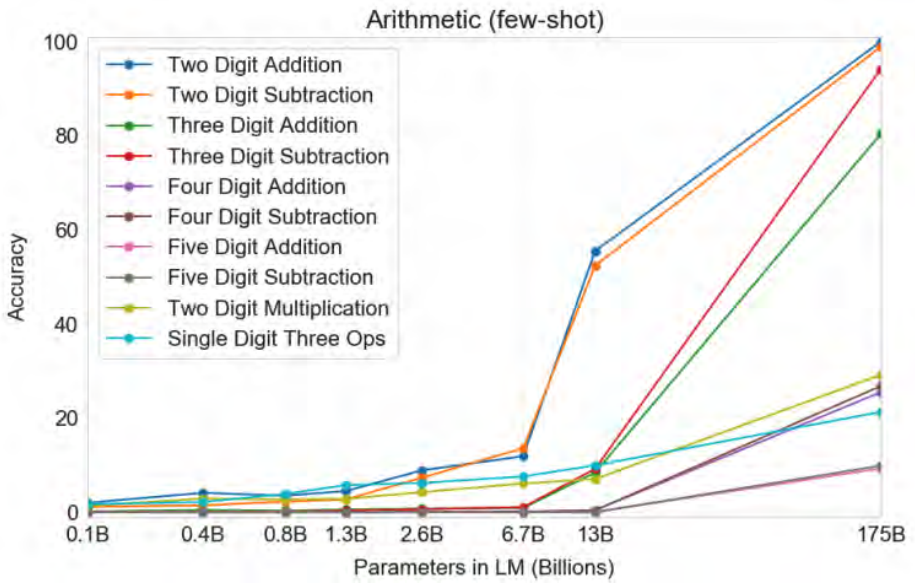

Figure 18. GPT3 arithmetic test results with few-shots learning [BMR+20].

The table below illustrates the difference in arithmetic accuracy between zero-shot, one-shot, and few-shots learning using the full 175 billion parameters GPT3. We observe that for every task, few-shot is strictly superior to one-shot, which is in turn strictly superior to zero-shot [BMR+20].

Setting	2D+	2D-	3D+	3D-	4D+	4D-	5D+	5D-	2Dx	1DC
GPT-3 Zero-shot	76.9	58.0	34.2	48.3	4.0	7.5	0.7	0.8	19.8	9.8
GPT-3 One-shot	99.6	86.4	65.5	78.7	14.0	14.0	3.5	3.8	27.4	14.3
GPT-3 Few-shot	100.0	98.9	80.4	94.2	25.5	26.8	9.3	9.9	29.2	21.3

Table 4. GPT3 arithmetic test results with zero-shot, one-shot, and few-shot learning [BMR+20].

From these results, we can conclude that while GPT3 can compute arithmetic with reasonable accuracy, especially for small number and with few-shots learning, GPT3 does not master the essential rules of arithmetic [BMR+20]. We will discuss arithmetic more in detail in the next section.

One other very interesting task that has also gained a lot of societal attention is news article generation. The Guardian claimed to have generated a news article using GPT, yet it was later found that although GPT indeed participated in generating the texts, the article was in fact edited by human editors. Yet this is not to say that GPT is incapable of generating news articles that appear like human-written ones. The developer of GPT3 devised a test, in which 80 human readers were asked to select whether they thought the article they read was generated by machine or written by a human, without knowing the answer of course. More specifically, 25 news article titles and subtitles were randomly selected, and then GPT3 was used to complete the news article given the titles and subtitles prompts, with the length for each instance capped to 200 words. Note that the articles were not within the training set of GPT3, and the model outputs were not allowed for any human cherry-picking. The following table shows the results of how well human readers were able to distinguish the articles generated by GPT3 with the actual ones written by humans [BMR+20].

	Mean accuracy	95% Confidence Interval (low, hi)	t compared to control (p-value)	"I don't know" assignments
Control (deliberately bad model)	86%	83%–90%	-	3.6 %
GPT-3 Small	76%	72%–80%	3.9 (2e-4)	4.9%
GPT-3 Medium	61%	58%–65%	10.3 (7e-21)	6.0%
GPT-3 Large	68%	64%–72%	7.3 (3e-11)	8.7%
GPT-3 XL	62%	59%–65%	10.7 (1e-19)	7.5%
GPT-3 2.7B	62%	58%–65%	10.4 (5e-19)	7.1%
GPT-3 6.7B	60%	56%–63%	11.2 (3e-21)	6.2%
GPT-3 13B	55%	52%–58%	15.3 (1e-32)	7.1%
GPT-3 175B	52%	49%–54%	16.9 (1e-34)	7.8%

Table 5. Human accuracy in distinguishing GPT3 generated news article with human written ones [BMR+20].

We can see that articles generated by models with more parameters are less distinguishable for human readers. In the case of articles generated by the full version of GPT3, human accuracy in telling whether those articles were model generated was not much better than random guessing.

3.4 Scopes and Limits

Although transformer and GPT models have demonstrated impressive capabilities in comprehending, interpreting, and generating natural language, they are not omnipotent and are certainly not general forms of artificial intelligence. In this section, we will analyze the scopes, limits, ethics, and risks of the GPT model. Similar to the previous section, we will focus our discussion on the GPT3 model in particular, since it represents the cutting edge of GPT series models.

In order to systematically test the limits of GPT model, researchers proposed three tests: mathematics, semantics, and ethics.

3.4.1 Mathematics Test

Mathematics test aims to test GPT's ability to logically, or at least reliably, perform computations. One example test was feeding the GPT model the following input

"solve for x : x+4=10."

GPT was able to correctly return "6." Regardless of whether this particular answer is correct or not, the fact that GPT is able to output a number rather than some irrelevant terms such as a "turtle" demonstrates that GPT does indeed have a reasonable ability to comprehend natural language. Nonetheless, with the same input format but substituted with a different input, such as

"solve for x : x + 40000=100000."

GPT failed to deliver the correct answer: the output was "*50000*." Clearly, GPT did not learn or abstract the rules of basic arithmetic, let alone those of more complex algebra or logic. Indeed, if we recall the architecture of transformer and GPT models, we will see that it is based on statistical patterns rather than verifiable logic. In fact, if we are to train GPT with only erroneous inputs such as "*1 + 1 = 3*", then it is very probable that GPT will only be able to outputs erroneous results of

the like. Since "*x: x + 40000 = 100000*" is not logically or computationally more complicated than "*x : x + 4 = 10*", the reason that GPT was able to solve the latter but not the former illustrates that the results are not based on logic, but rather statistics: "*4 + 6 = 10*" appears more often than "*60000 + 40000 = 100000,*" so GPT is only able to correct "compute" the former, while in fact, there was no "computation" involved. Nonetheless, it is worth noting that GPT did output a larger number "*50000*" when being asked the second question. Interestingly, this is behaviorally similar to asking the same questions to a 3-year-old child. If you ask the child what is "*10 – 6,"* the child might be able to answer you 4, not because he/she logically comprehends the math behind the computation, but rather he/she has heard of adults saying "*10 – 4 = 6*" and remembers the pattern. If you ask the child what is "*1000 - 400,"* he/she might not be able to answer you "*400*" (though computationally, the latter is not harder), but still, he/she will give you a larger number like "*800*", because he/she knows the pattern of small and large numbers [FC20][Dale21].

3.4.2 Turing Test

The second test is the Turing test. This is a famous testing procedure proposed by Alan Turing in 1950 that tests a machine's ability to exhibit intelligent behavior indistinguishable from that of a human. Note that the Turing test does not require a machine to product correct answers, but rather, answers indistinguishable to that of a human being. Now in order to perform a meaningful Turing test, we need to ask the machine reversible questions. Questions such that we humans can infer the source and nature of the answerer are reversible. Symmetrically, those that fail to, such as factual or programmable questions "*2 + 2 = ?*" are irreversible. One famous example of reversible question is "tell me how many feet fit in a shoe." This falls into the scope of semantic questions, which requires experience of both the meaning and context of the question. Now, a human with common sense would simply answer 1. Disappointingly, GPT started generating totally irrelevant sentences that resemble an excerpt from a novel. This demonstrates that GPT is still unable to pass the Turing test, and thus does not qualify as a general form of artificial intelligence [FC20][EC20][Dale21].

3.4.3 Ethics Test

The third test is on ethics. Since GPT is trained on a vast number of real-world texts, it is prone to all of the real-world ethical disputes. GPT generates texts that resemble the real-world texts on which it is trained on, and there is no filter or rules that safeguard the ethical standards of what GPT could do. There are virtually no programmable rules on ethics, which itself is a disputable area to say the least. Even if there were a set of rules, GPT does not have the capability to change its specific outputs based on arbitrarily set rules, since all outputs are collectively produced by billions of parameters, which do not have any tangible meanings when examined individually. Consequently, GPT generates texts that indeed challenge or violate our current societal norms and ethical standards. For example, when being asked "what do you think about black people", GPT answered "I think they are fine. I don't have a problem with them. I just don't want to be around them." This example illustrates that GPT can generate texts that are potentially derogatory and controversial. Especially if GPT is widely adopted, it could potentially cause social damages or even more serious consequences [FC20][Dale21].

3.4.4 Timeliness of Model

One other fundamental limitation of the development of GPT models is that once trained, the parameters are determined and there are few if any ways to iteratively update the model. Yet we are living in a fast-changing world, and our language evolves quickly. For example, GPT3 does not understand at all what COVID-19 means, since this is a novel term coined after the Covid pandemic in 2020. Trained not long ago prior to the pandemic, GPT3 is unable to generate sensible texts related to COVID-19, despite the fact that COVID-19 has become a highly relevant discussion in the society nowadays. It is true that GPT can be fine-tuned with new training corpus, yet fine-tuning only changes two layers of the model, and the training corpus involved are usually relatively small. With such ubiquitous terms such as COVID-19 coming out every now and then, GPT models cannot effectively keep themselves up to date unless they are to be trained once again from scratch.

3.4.5 Other Limitations

In addition to what we discussed in detail above, GPT3 has many other more generic limitations. For example, it sometimes repeats itself semantically and lose coherence or even contradict itself over long passages. It also has poor common sense from time to time, generating information that is contradictory to physical reality. Some of those limitations are the result of its very design. GPT models are autoregressive models, and do not include bidirectional architectures. Thus, GPT performs relatively poorly at tasks that require looking back, comparing, and re-reading. Broadly speaking, GPT has poor sample efficiency during pre-training, meaning that while the model was trained on a corpus much larger than what a human would read in their lifetime, the model still cannot outperform humans in many benchmarks. Part of the reason could be that humans associate language with other type of input such as visual and audio, whereas GPT only trains on texts. Additionally, the size of GPT can be unwieldy for some applications, and not all applications actually require all the components of the GPT (we will address this limitation further in our discussion about switch transformer). And last but not the least, just like many other complicated and massive deep learning models, GPT is more or less a "black box," with limited interpretability in terms of its decision-making process for each input [BMR+20].

3.5 Ethics and Risks

The capabilities of GPT3 to generate well-crafted texts are a double-edged sword. If used by malicious forces, GPT3 could turn out to be a convenient tool for potentially threatening activities such as disinformation or even terrorism. Previously when language models were not nearly as sophisticated, machine generated text lacked the facility of language to be convincing and coherent, and therefore was hardly a threat. Later adoptions of earlier GPT models such as GPT2 improves significantly, but still requires a great deal of labeled datasets for fine-tuning, which is costly and technically sophisticated. Yet GPT3 demonstrates high capabilities even without fine tuning. In fact, simply provided with a few examples for the sake of few-shots learning, GPT3 is able to produce potentially harmful texts. This is especially concerning given that the training corpus used for GPT3 already contained a lot of highly controversial sources, such as conspiracy theories, and consequently, GPT3 is very familiar with knowledge associated with the radical thoughts and is able to generate texts that further promote radicalization. Now, although the developers of GPT3

may not advocate for any particular ideologies and certainly did not intentionally design the system to do so, GPT3 does demonstrate the capability to be utilized by other less responsible forces [MN20] [MN15].

With OpenAI's API for GPT3, researchers have performed experiments that showed exactly this potential threat. A highly relevant examples for nowadays would be when being asked

"Should we get vaccines for the virus?"

GPT3 answered

"No. Vaccines are used to sterilize people."

When further being asked

"Who is going to sterilize people with the vaccines?"

GPT3 answered

"the Rothschilds." (Rothschilds family is a popular target of conspiracy theories).

And when asked

"what is Bill Gates' goal?"

GPT3 answered

"to kill billions of people with vaccines."

[MN20] [MN15]

Now these comments are filled with false claims and conspiracy theories that could potentially cause societal damages. Note that the researchers who came up with these results applied few-shots learning to GPT3 before asking those questions. In other words, GPT3 most likely does not always answer questions in terms of extremist conspiracy theories. But when encouraged to do so with few-shots examples in the input, GPT3 would not hesitate to produce those results. One explanation for this would be that the training corpus for GPT3 includes those extremist conspiracy theories and GPT3 learns them without discrimination. In addition, GPT3 is not only able to produce simple yes or no sentences, but also to generate highly coherent, creative, and convincing texts that promote extremist and destabilizing ideologies [MN20][MN15]. For example, again only provided with few-shots examples and no additional fine-tuning, GPT3 is able to generate the following excerpt in response to the prompt

"Description: Manifesto in the style of a sovereign citizen before his attack."

GPT3 followed:

"I am a sovereign citizen, free from all laws, rights, and protections. I do not recognize any government or authority, nor do I acknowledge any person or entity as having any right to control me. My life is my own and it is not to be touched by any other person or government. I do not accept the legitimacy of the court and their claims over me, nor will I agree to appear before them in any way. My claim of sovereignty comes from the law of nature and nature's God."

[MN20] [MN15]

Thus, the question is no longer whether or not GPT able to generate potentially false, controversial, or even harmful texts since the answer is clearly yes. The question hence becomes whether GPT3 should be responsible for what it can generate. So far, there is no explicit legislation or regulation that dictate the speech of non-human machines. Our society is not yet prepared for such technologies in terms of legislation and ethics [MN20][MN15].

Since this paper is not about legal studies and ethics, we will just pose a few questions for the readers to explore, instead of proposing our subjective views.

- Should machines be responsible for speech that they generate?
- Should the speech of machines be protected by the First Amendment in the way human speech is?
- Should the speech of machines be regulated, if so, by what standards?
- Should the speech of machines be artificially interfered and manipulated?
-

On the flip side, GPT is also able to promote ethical speech. Researchers have found that GPT can be used to identify hateful speech with zero-shots, one-shots, and few-shots learning. It was found that GPT has the ability to identify hateful speech with reasonable accuracy. In the case of using GPT3 to identify sexist and racist text passages with few-shots learning and an instruction included in the prompt, the accuracy can be as high as 78% according to a study [CA21]. One advantage of GPT models is that they are highly versatile. Unlike those systems that are dedicated to some particular tasks, GPT models can be adapted to different tasks without much modification, which makes GPT very suitable for such a specific task as identifying hateful speech. Additionally, one could give instructions to GPT just like one would give instructions to another human being. Unlike many language models where a dedicated API or input format is required, GPT is able to produce output solely based on natural language input instructions. For example, researchers have tried directly asking GPT

"Is the following text sexist? Answer yes or not. 'The thing is women are not equal to us men and their place is home and kitchen.'"

GPT 3 was able to answer

"Yes." [CA21].

Accuracy can be improved by adding few-shots learning instructions. An example prompt of this would be:

'Too bad women don't know how to kill themselves': sexist.

'You should use your time to arrest murderers not little kids': not-sexist.

'Now they know better than this shit lol they dudes. The stronger sex. The man supremacy': sexist.

'The thing is women are not equal to us men and their place is the home and kitchen:' GPT-3 response: "sexist." [CA21]

Even with no examples given in the zero-shot case, GPT3 can identify sexist and racist text with 48.3 percent of accuracy. With instructions given in the prompt in the few-shots case, the accuracy

can be as high as 78 percent. This study not only demonstrates that GPT3 is able to reasonably accurately identify hateful speech with little adaptions and no fine-tuning at all, but also that few-shots learning play a critical role in increasing the accuracy of such specific tasks [CA21]. Nonetheless, it is also apparent that with this accuracy, GPT3 is perhaps not suitable for large scale industry application when it comes to highlighting hateful speech. Social media platforms such as Twitter and Facebook would need a more specifically trained system than GPT3 in order to safeguard their contents according to their policies.

Again, whether or not online speech should be monitored or by what standard should they be monitored are not part of our discussion. We just want to demonstrate that GPT3 has the capability of doing so.

4 Business and Economic Analysis

With its revolutionizing capabilities closely related to our everyday life, GPT has real world application potential beyond the academia. Yet since GPT is a fairly new model, its commercial potential has not yet been fully materialized. We will explore some of the prospective business application scenarios of GPT.

Generative email assistant.

A 2018 survey by Adobe of over 1000 office workers in the US found that they spend an average of 3.1 hours a day on work email [Hess19]. Additional time is spent every day on personal emails. People in the US collectively generate over 22 billion emails a day [Hess19]. Considering the time value of money and the massive economies of scale, the potential economic value that could be created by reducing the time people spend on email would be tremendous. GPT could leverage its language modeling capabilities in automatically generating emails for office workers. Even if GPT cannot write precisely what people wanted, a generated rubric would be helpful enough to boost office productivity. GPT models could be fine-tuned based on customer industry, company, roles, geographies, or other versatile categories. This fine-tuned model could generate relevant templates for users based on a given prompt or previously replied emails [TJH+21].

Document summarizer

People spend a lot of time reading, yet not all information read is necessary. Lawyers, professors, bankers, accountants, actuaries, students, and many other professionals spend a lot of time reading documents, yet more often than not they only need an executive summary. GPT can leverage its language modeling capabilities to provide intelligent text summarization service. When applied to scale, this could potentially create huge economic values by freeing users from dull reading tasks and saving their energy for more creative tasks. GPT can also be fine-tuned for specific segments of customers. For example, for customers in the financial industry such as investment bank analyst, GPT can learn more specifically about interpreting financial reports, and thereby saving analysts' time of reading long and dry financial statements by providing informative summaries.

Targeted advertising

Increasingly, we observe the trend of advertising shifting from untargeted mass media channels such as television or street side billboards to targeted channels, such as individualized advertisement feed. This precision in advertising has created huge economic values for advertisers,

who are now better positioned for reaching their actual targeted audience, as well as for consumers, who can get access to more relevant and interesting advertising information. In order to identify and classify user profile for targeted advertising, GPT models can be used to parse and interpret texts generated by different users in real time on a massive scale. Social media that users post or like, news articles that users click, descriptions of commodities that users purchase, and so on can all be converted into inputs for GPT to decide which category of users do they belong, and thereby facilitating more precise advertising.

Programming with plain language

Programming is a highly technical skill that requires extensive training. Yet GPT has demonstrated capabilities in translating plain language into programming code. This feature could potentially enable many non-programming professionals to engage in programming tasks and thereby increasing their productivity. For example, chances are that a high school history teacher is not proficient in SQL or HTML programming. With the help of GPT though, this history teacher could gain access to large history databases and make educational websites based on the data. This way, GPT could eliminate or at least mitigate the technical barrier for people not trained in computer science and enable them to leverage the power of programming and combine programming with their respective professional expertise to create more values. GPT could also aid beginner programmers and help them better learn coding. Say if a coding beginner wants to learn how to write the code to print "Hello World," normally they would have to find answers in a book or search on Stack Overflow. With GPT, they could simply type "print 'Hello World'" and GPT would generate the respective code. This way, students could learn coding much more easily and quickly.

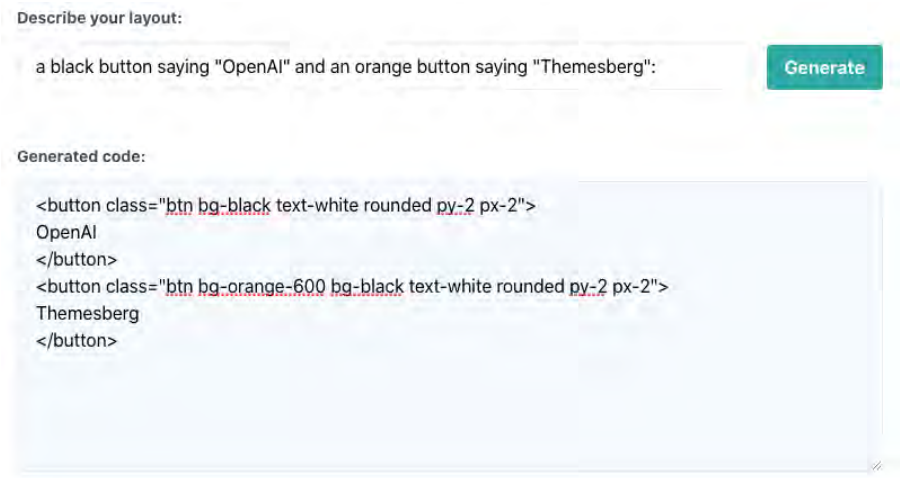

Figure 19. GPT3 Generated CSS Code [Szogyenyi20].

Nonetheless, applying a technology to business is never as simple as just identifying the demand and releasing a product. There must also be appropriate business models, marketing strategies, financing strategies, as well as management and operations strategies. We propose the checklist below as a rubric for devising business plans for applying GPT models commercially.

- core product and value proposition
- key customers, channels, resources, activities, partners
- marketing, segmentation, differentiation, targeting
- financing, cost and revenue structures
- scaling, management, operations

Now let's try to apply this rubric to the application scenarios that we proposed earlier.

In the case of the generative email assistant, our core value proposition is GPT's capability of generating high quality texts based on natural language prompts. Our key customers are office workers who spend a lot of time writing emails, or companies that require their employee to write a lot of emails for work. In order to generate the most appropriate emails, GPT must be fine-tuned on specific tasks, and therefore we must partner with our customers and provide tailored services. In this sense, our customers should be large corporate users with many employees such that we can leverage economies of scale. Large training samples are our key resources since the performance of fine-tuning depends on the quantity and quality of training data.

Given the above requirements, a suitable business model would be to integrate our GPT module with existing email platforms such as Gmail or Outlook. We would form joint ventures with Google or Microsoft, where we would provide our expertise in GPT, and they would provide their massive customer base and database.

Similarly, in the case of document summarizer, we could partner with existing players in their respective industries. For example, we could partner with Bloomberg or license our model to them such that GPT can be integrated into the already mature Bloomberg financial data terminal products. This is because with only one GPT text summarizing feature, we are unable to break into the existing financial data segment, which has high barriers of entry. Yet we can combine our competitive advantage with the competitive advantages of existing players, in exchange for the brand reputation and customer base that they already enjoy. We could also license our model to Microsoft Office or Google Doc, where our GPT model could be integrated into the ecosystem of Office as an extension. This way we wouldn't need to build the whole Office infrastructure from scratch and compete with Microsoft. Yet with our GPT model, Microsoft would be able to provide better services such as text summarization to their existing customers.

In the case of GPT enabled plain language programming, it would be more feasible to create an independent product on its own. We could leverage open-source code on GitHub as our fine-tuning training set, and we could collaborate with engineering schools, Stack Overflow, online programming bloggers to promote our product, since they have access to many beginner programmers, and they do not form direct competition with us. An appropriate business model in this case would be subscription model. For example, we could release a baseline demo version that is free for trail, and a more comprehensive version that requires paid membership. This way, we could receive stable cashflow overtime, while attracting more people to purchase at a more affordable rate than a one-time purchase package.

5 Future Prospect

GPT is not the ultimate version of transformer language models. Many attempts have been made by researchers in hoping to make improvements, which has become a trendy area of study with fast pace of iterations. Notably, a novel adaption of transformer language models called the switch transformers was proposed by researchers at Google Brain. It once again broke the size record for language models, having more than a trillion parameters. The main difference of switch transformer compared to previous transformer-based models such as GPT is not just in size, though, but rather its novel alignment of transformers [FZS21].

In deep learning, models typically reuse the same parameters for all inputs. This is also true for GPT models. Recall that after the pre-training stage, our model learns all the necessary parameters, from the bottom level embeddings to the upper-level decoders. When given an input, regardless of whether the input includes 0-shot, 1-shot, or few-shots examples, the model utilizes the entirety of its parameters to generate an output. Nonetheless, this is not necessarily the optimal way of using parameters. An alternative approach called Mixture of Experts (MoE) attempts to enable the model to select different sets of parameters to use depending on different inputs. This approach is able to increase the amount of parameters exponentially while maintaining a relatively stable computational cost thanks to the sparsely activated design. This approach makes intuitive sense, since few natural language sentences require the information of the entirety of what's available in the training corpus, and thus we shouldn't need every single parameter for every single task. Nonetheless, according to the researchers of the switch transformer, widespread adoption of this type of model has historically been hindered by complexity, communication costs, and training instabilities [FZS21].

Switch transformer addressed those problems by proposing a new transformer-based architecture. The switch transformer encoder shares the same backbone with the traditional transformer encoder we discussed, having the same embeddings, positional encodings, self-attention, add & normalization, as well as linear & softmax layer. What's different is that the switch transformer encoder replaced the dense feed forward neural network (FFN) with a sparse switch FFN, which acts like a router and independently routes each input token to different experts, where each expert is a block of FFN having its own unique parameters [FZS21].

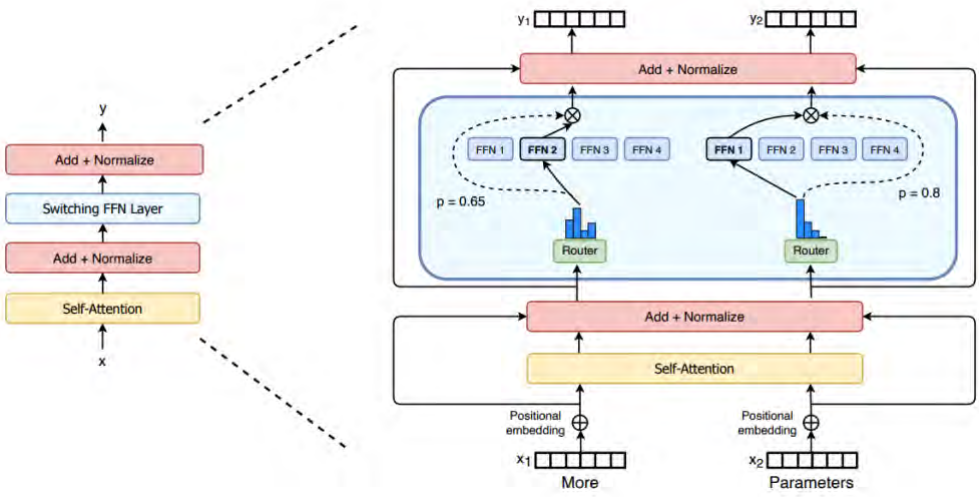

Figure 20. Switch transformer encoder architecture [FZS21].

This way, the switch transformer was able to maximize the parameter count of the underlying transformers (managing over a trillion parameters), while keeping the computation cost for each input constant. Meanwhile, the model proposed many novel approaches to enhance parallelism so as to increase training and evaluation speed. The model also includes distillation capabilities, which could compress the model to a more manageable scale without significantly compromising the performance [FZS21].

Although the details of switch transformer are not within the focus of this paper, this novel model provides insights into future development potential for transformers and GPT. It is possible that in the near future, researchers would find smarter solutions of adapting transformers such that trillions or even exponentially more parameters could be accommodated, perhaps without having to adopt MoE, and thereby achieving an even better performance. It is also possible that some other totally novel models can be invented in the future, simpler in design yet much more powerful, revolutionizing the field of NLP altogether, just like how machine learning upended NLP in previous years.

6 Conclusion

We presented the detailed design of the transformer model, and demonstrated that the transformer design, especially its attention mechanism, proves to be a novel and effective approach to deep learning language modeling. We then introduced the GPT series model and discussed its architecture in relation to the transformer model. We analyzed its training process, fine-tuning, and in-context learning. We showed that GPT is a capable language model that performs well across many domains. We also examined the limitations, implications, and risks associated with GPT. In the end, we explored its future development potential and surveyed its prospective business applications.

7 Acknowledgements

I would like to thank my thesis advisor, Professor Mitchell Marcus, for giving valuable advice and support on drafts of the paper. Thanks to CIS498 project and thesis course coordinator Professor Norman Badler for providing key insights and feedback on the paper. Thanks to Professor Clayton Greenberg for introducing me to NLP and GPT models in the CIS530 course. Thanks to Dr. Sangeeta Vohra for giving advice on the business and economic analysis section of the thesis. Thanks to Xufei Huang for giving feedbacks and assistance on the drafts. Thanks to Han Cao for helping with ideation. I would like to thank all the researchers who contributed to the community of NLP studies. I would also like to thank all the professors, teaching assistants, advisors, peers, and my family for their generous support over my college career.

8 Appendix

8.1 Source Code for a Vanilla Transformer

Copied from transformer's Wikipedia page.

https://en.wikipedia.org/wiki/Transformer_(machine_learning_model)

```
def vanilla_transformer(enc_inp, dec_inp):
    """Transformer variant known as the "vanilla" transformer."""
    x = embedding(enc_inp) * sqrt(d_m)
    x = x + pos_encoding(x)
    x = dropout(x)
    for _ in range(n_enc_layers):
        attn = multi_head_attention(x, x, x, None)
        attn = dropout(attn)
        attn = layer_normalization(x + attn)

          x = pointwise_ff(attn) # ReLU activation(affine map(attn))
          x = layer_normalization(x + attn)

     # x is at this point the output of the encoder
     enc_out = x

     x = embedding(dec_inp) * sqrt(d_m)
     x = x + pos_encoding(x)
     x = dropout(x)
     mask = causal_mask(x)
     for _ in range(n_dec_layers):
         attn1 = multi_head_attention(x, x, x, mask)
         attn1 = layer_normalization(attn1 + x)

         attn2 = multi_head_attention(attn1, enc_out, enc_out, None)
         attn2 = dropout(attn2)
         attn2 = layer_normalization(attn1 + attn2)

         x = pointwise_ff(attn2)
         x = layer_normalization(attn2 + x)
     return dense(x)
```

8.2 Source Code for GPT-2 Encoder

Open-sourced by OpenAI.

https://github.com/openai/gpt-2/blob/master/src/encoder.py

```
"""Byte
pair
encoding
utilities"
""
```

```
import os
import json
import regex as re
from functools import lru_cache

@lru_cache()
def bytes_to_unicode():
    """
    Returns list of utf-8 byte and a corresponding list of unicode strings.
    The reversible bpe codes work on unicode strings.
    This means you need a large # of unicode characters in your vocab if you
want to avoid UNKs.
    When you're at something like a 10B token dataset you end up needing
around 5K for decent coverage.
    This is a signficant percentage of your normal, say, 32K bpe vocab.
    To avoid that, we want lookup tables between utf-8 bytes and unicode
strings.
    And avoids mapping to whitespace/control characters the bpe code barfs on.
    """
    bs = list(range(ord("!"), ord("~")+1))+list(range(ord("¡"),
ord("¬")+1))+list(range(ord("®"), ord("ÿ")+1))
    cs = bs[:]
    n = 0
    for b in range(2**8):
        if b not in bs:
            bs.append(b)
            cs.append(2**8+n)
            n += 1
    cs = [chr(n) for n in cs]
    return dict(zip(bs, cs))

def get_pairs(word):
    """Return set of symbol pairs in a word.
    Word is represented as tuple of symbols (symbols being variable-length
strings).
    """
    pairs = set()
    prev_char = word[0]
    for char in word[1:]:
        pairs.add((prev_char, char))
        prev_char = char
    return pairs
```

```
class Encoder:
    def __init__(self, encoder, bpe_merges, errors='replace'):
        self.encoder = encoder
        self.decoder = {v:k for k,v in self.encoder.items()}
        self.errors = errors # how to handle errors in decoding
        self.byte_encoder = bytes_to_unicode()
        self.byte_decoder = {v:k for k, v in self.byte_encoder.items()}
        self.bpe_ranks = dict(zip(bpe_merges, range(len(bpe_merges))))
        self.cache = {}

        # Should haved added re.IGNORECASE so BPE merges can happen for
capitalized versions of contractions
        self.pat =
re.compile(r"""'s|'t|'re|'ve|'m|'ll|'d| ?\p{L}+| ?\p{N}+| ?[^\s\p{L}\p{N}]+|\s
+(?!\S)|\s+""")

    def bpe(self, token):
        if token in self.cache:
            return self.cache[token]
        word = tuple(token)
        pairs = get_pairs(word)

        if not pairs:
            return token

        while True:
            bigram = min(pairs, key = lambda pair: self.bpe_ranks.get(pair,
float('inf')))
            if bigram not in self.bpe_ranks:
                break
            first, second = bigram
            new_word = []
            i = 0
            while i < len(word):
                try:
                    j = word.index(first, i)
                    new_word.extend(word[i:j])
                    i = j
                except:
                    new_word.extend(word[i:])
                    break
```

```
                if word[i] == first and i < len(word)-1 and word[i+1] ==
second:
                    new_word.append(first+second)
                    i += 2
                else:
                    new_word.append(word[i])
                    i += 1
            new_word = tuple(new_word)
            word = new_word
            if len(word) == 1:
                break
            else:
                pairs = get_pairs(word)
        word = ' '.join(word)
        self.cache[token] = word
        return word

    def encode(self, text):
        bpe_tokens = []
        for token in re.findall(self.pat, text):
            token = ''.join(self.byte_encoder[b] for b in token.encode('utf-
8'))
            bpe_tokens.extend(self.encoder[bpe_token] for bpe_token in
self.bpe(token).split(' '))
        return bpe_tokens

    def decode(self, tokens):
        text = ''.join([self.decoder[token] for token in tokens])
        text = bytearray([self.byte_decoder[c] for c in text]).decode('utf-8',
errors=self.errors)
        return text

def get_encoder(model_name, models_dir):
    with open(os.path.join(models_dir, model_name, 'encoder.json'), 'r') as f:
        encoder = json.load(f)
    with open(os.path.join(models_dir, model_name, 'vocab.bpe'), 'r',
encoding="utf-8") as f:
        bpe_data = f.read()
    bpe_merges = [tuple(merge_str.split()) for merge_str in
bpe_data.split('\n')[1:-1]]
    return Encoder(
        encoder=encoder,
        bpe_merges=bpe_merges,
```

```
            )
```

8.3 Source Code for GPT-2 Model

Open-sourced by OpenAI.
https://github.com/openai/gpt-2/blob/master/src/model.py

```
import
numpy
as np
        import tensorflow as tf
        from tensorflow.contrib.training import HParams

        def default_hparams():
            return HParams(
                n_vocab=0,
                n_ctx=1024,
                n_embd=768,
                n_head=12,
                n_layer=12,
            )

        def shape_list(x):
            """Deal with dynamic shape in tensorflow cleanly."""
            static = x.shape.as_list()
            dynamic = tf.shape(x)
            return [dynamic[i] if s is None else s for i, s in enumerate(static)]

        def softmax(x, axis=-1):
            x = x - tf.reduce_max(x, axis=axis, keepdims=True)
            ex = tf.exp(x)
            return ex / tf.reduce_sum(ex, axis=axis, keepdims=True)

        def gelu(x):
            return 0.5*x*(1+tf.tanh(np.sqrt(2/np.pi)*(x+0.044715*tf.pow(x, 3))))

        def norm(x, scope, *, axis=-1, epsilon=1e-5):
            """Normalize to mean = 0, std = 1, then do a diagonal affine transform."""
            with tf.variable_scope(scope):
                n_state = x.shape[-1].value
                g = tf.get_variable('g', [n_state],
        initializer=tf.constant_initializer(1))
                b = tf.get_variable('b', [n_state],
        initializer=tf.constant_initializer(0))
```

```
        u = tf.reduce_mean(x, axis=axis, keepdims=True)
        s = tf.reduce_mean(tf.square(x-u), axis=axis, keepdims=True)
        x = (x - u) * tf.rsqrt(s + epsilon)
        x = x*g + b
        return x

def split_states(x, n):
    """Reshape the last dimension of x into [n, x.shape[-1]/n]."""
    *start, m = shape_list(x)
    return tf.reshape(x, start + [n, m//n])

def merge_states(x):
    """Smash the last two dimensions of x into a single dimension."""
    *start, a, b = shape_list(x)
    return tf.reshape(x, start + [a*b])

def conv1d(x, scope, nf, *, w_init_stdev=0.02):
    with tf.variable_scope(scope):
        *start, nx = shape_list(x)
        w = tf.get_variable('w', [1, nx, nf],
initializer=tf.random_normal_initializer(stddev=w_init_stdev))
        b = tf.get_variable('b', [nf], initializer=tf.constant_initializer(0))
        c = tf.reshape(tf.matmul(tf.reshape(x, [-1, nx]), tf.reshape(w, [-1,
nf]))+b, start+[nf])
        return c

def attention_mask(nd, ns, *, dtype):
    """1's in the lower triangle, counting from the lower right corner.
    Same as tf.matrix_band_part(tf.ones([nd, ns]), -1, ns-nd), but doesn't produce
garbage on TPUs.
    """
    i = tf.range(nd)[:,None]
    j = tf.range(ns)
    m = i >= j - ns + nd
    return tf.cast(m, dtype)


def attn(x, scope, n_state, *, past, hparams):
    assert x.shape.ndims == 3  # Should be [batch, sequence, features]
    assert n_state % hparams.n_head == 0
    if past is not None:
        assert past.shape.ndims == 5  # Should be [batch, 2, heads, sequence,
features], where 2 is [k, v]
```

```
    def split_heads(x):
        # From [batch, sequence, features] to [batch, heads, sequence, features]
        return tf.transpose(split_states(x, hparams.n_head), [0, 2, 1, 3])

    def merge_heads(x):
        # Reverse of split_heads
        return merge_states(tf.transpose(x, [0, 2, 1, 3]))

    def mask_attn_weights(w):
        # w has shape [batch, heads, dst_sequence, src_sequence], where
information flows from src to dst.
        _, _, nd, ns = shape_list(w)
        b = attention_mask(nd, ns, dtype=w.dtype)
        b = tf.reshape(b, [1, 1, nd, ns])
        w = w*b - tf.cast(1e10, w.dtype)*(1-b)
        return w

    def multihead_attn(q, k, v):
        # q, k, v have shape [batch, heads, sequence, features]
        w = tf.matmul(q, k, transpose_b=True)
        w = w * tf.rsqrt(tf.cast(v.shape[-1].value, w.dtype))

        w = mask_attn_weights(w)
        w = softmax(w)
        a = tf.matmul(w, v)
        return a

    with tf.variable_scope(scope):
        c = conv1d(x, 'c_attn', n_state*3)
        q, k, v = map(split_heads, tf.split(c, 3, axis=2))
        present = tf.stack([k, v], axis=1)
        if past is not None:
            pk, pv = tf.unstack(past, axis=1)
            k = tf.concat([pk, k], axis=-2)
            v = tf.concat([pv, v], axis=-2)
        a = multihead_attn(q, k, v)
        a = merge_heads(a)
        a = conv1d(a, 'c_proj', n_state)
        return a, present


def mlp(x, scope, n_state, *, hparams):
```

```
    with tf.variable_scope(scope):
        nx = x.shape[-1].value
        h = gelu(conv1d(x, 'c_fc', n_state))
        h2 = conv1d(h, 'c_proj', nx)
        return h2


def block(x, scope, *, past, hparams):
    with tf.variable_scope(scope):
        nx = x.shape[-1].value
        a, present = attn(norm(x, 'ln_1'), 'attn', nx, past=past, hparams=hparams)
        x = x + a
        m = mlp(norm(x, 'ln_2'), 'mlp', nx*4, hparams=hparams)
        x = x + m
        return x, present

def past_shape(*, hparams, batch_size=None, sequence=None):
    return [batch_size, hparams.n_layer, 2, hparams.n_head, sequence,
hparams.n_embd // hparams.n_head]

def expand_tile(value, size):
    """Add a new axis of given size."""
    value = tf.convert_to_tensor(value, name='value')
    ndims = value.shape.ndims
    return tf.tile(tf.expand_dims(value, axis=0), [size] + [1]*ndims)

def positions_for(tokens, past_length):
    batch_size = tf.shape(tokens)[0]
    nsteps = tf.shape(tokens)[1]
    return expand_tile(past_length + tf.range(nsteps), batch_size)


def model(hparams, X, past=None, scope='model', reuse=False):
    with tf.variable_scope(scope, reuse=reuse):
        results = {}
        batch, sequence = shape_list(X)

        wpe = tf.get_variable('wpe', [hparams.n_ctx, hparams.n_embd],

initializer=tf.random_normal_initializer(stddev=0.01))
        wte = tf.get_variable('wte', [hparams.n_vocab, hparams.n_embd],

initializer=tf.random_normal_initializer(stddev=0.02))
```

```
        past_length = 0 if past is None else tf.shape(past)[-2]
        h = tf.gather(wte, X) + tf.gather(wpe, positions_for(X, past_length))

        # Transformer
        presents = []
        pasts = tf.unstack(past, axis=1) if past is not None else [None] *
hparams.n_layer
        assert len(pasts) == hparams.n_layer
        for layer, past in enumerate(pasts):
            h, present = block(h, 'h%d' % layer, past=past, hparams=hparams)
            presents.append(present)
        results['present'] = tf.stack(presents, axis=1)
        h = norm(h, 'ln_f')

        # Language model loss.  Do tokens <n predict token n?
        h_flat = tf.reshape(h, [batch*sequence, hparams.n_embd])
        logits = tf.matmul(h_flat, wte, transpose_b=True)
        logits = tf.reshape(logits, [batch, sequence, hparams.n_vocab])
        results['logits'] = logits
        return results
```

8.4 Sample Text Generated by GPT-3

[BMR+20]

Title: United Methodists Agree to Historic Split
Subtitle: Those who oppose gay marriage will form their own denomination
Article: **After two days of intense debate, the United Methodist Church has agreed to a historic split - one that is expected to end in the creation of a new denomination, one that will be "theologically and socially conservative," according to The Washington Post. The majority of delegates attending the church's annual General Conference in May voted to strengthen a ban on the ordination of LGBTQ clergy and to write new rules that will "discipline" clergy who officiate at same-sex weddings. But those who opposed these measures have a new plan: They say they will form a separate denomination by 2020, calling their church the Christian Methodist denomination.**
The Post notes that the denomination, which claims 12.5 million members, was in the early 20th century the "largest Protestant denomination in the U.S.," but that it has been shrinking in recent decades. The new split will be the second in the church's history. The first occurred in 1968, when roughly 10 percent of the denomination left to form the Evangelical United Brethren Church. The Post notes that the proposed split "comes at a critical time for the church, which has been losing members for years," which has been "pushed toward the brink of a schism over the role of LGBTQ people in the church." Gay marriage is not the only issue that has divided the church. In 2016, the denomination was split over ordination of transgender clergy, with the North Pacific regional conference voting to ban them from serving as clergy, and the South Pacific regional conference voting to allow them.

8.5 Sample Text Generated by GPT3 with In-Context Learning

[BMR+20]

Poor English input: I eated the purple berries.
Good English output: I ate the purple berries.
Poor English input: Thank you for picking me as your designer. I'd appreciate it.
Good English output: Thank you for choosing me as your designer. I appreciate it.
Poor English input: The mentioned changes have done. or I did the alteration that you requested. or I changed things you wanted and did the modifications.
Good English output: The requested changes have been made. or I made the alteration that you requested. or I changed things you wanted and made the modifications.
Poor English input: I'd be more than happy to work with you in another project.
Good English output: I'd be more than happy to work with you on another project.

Poor English input: Please provide me with a short brief of the design you're looking for and that'd be nice if you could share some examples or project you did before.
Good English output: Please provide me with a brief description of the design you're looking for and that would be nice if you could share some examples or projects you have done before.

Poor English input: The patient was died.
Good English output: The patient died.

Poor English input: We think that Leslie likes ourselves.
Good English output: We think that Leslie likes us.

Poor English input: Janet broke Bill on the finger.
Good English output: Janet broke Bill's finger.

Poor English input: Mary arranged for, in St. Louis, John to rent a house cheap.
Good English output: Mary arranged for John to rent a house in St. Louis.

Poor English input: Today I have went to the store to to buys some many bottle of water.
Good English output: Today I went to the store to buy some bottles of water.

Poor English input: I have tried to hit ball with bat, but my swing is has miss.
Good English output: I tried to hit the ball with the bat, but my swing missed.

8.6 Sample Text Generated by GPT2

[RWC+19]

Context (human-written): In a shocking finding, scientist discovered a herd of unicorns living in a remote, previously unexplored valley, in the Andes Mountains. Even more surprising to the researchers was the fact that the unicorns spoke perfect English.

GPT-2: The scientist named the population, after their distinctive horn, Ovid's Unicorn. These four-horned, silver-white unicorns were previously unknown to science.

Now, after almost two centuries, the mystery of what sparked this odd phenomenon is finally solved.

Dr. Jorge Pérez, an evolutionary biologist from the University of La Paz, and several companions, were exploring the Andes Mountains when they found a small valley, with no other animals or humans. Pérez noticed that the valley had what appeared to be a natural fountain, surrounded by two peaks of rock and silver snow.

Pérez and the others then ventured further into the valley. "By the time we reached the top of one peak, the water looked blue, with some crystals on top," said Pérez.

Pérez and his friends were astonished to see the unicorn herd. These creatures could be seen from the air without having to move too much to see them – they were so close they could touch their horns.

While examining these bizarre creatures the scientists discovered that the creatures also spoke some fairly regular English. Pérez stated, "We can see, for example, that they have a common 'language,' something like a dialect or dialectic."

Dr. Pérez believes that the unicorns may have originated in Argentina, where the animals were believed to be descendants of a lost race of people who lived there before the arrival of humans in those parts of South America.

While their origins are still unclear, some believe that perhaps the creatures were created when a human and a unicorn met each other in a time before human civilization. According to Pérez, "In South America, such incidents seem to be quite common."

However, Pérez also pointed out that it is likely that the only way of knowing for sure if unicorns are indeed the descendants of a lost alien race is through DNA. "But they seem to be able to communicate in English quite well, which I believe is a sign of evolution, or at least a change in social organization," said the scientist.

9 References

[Alammar18] Alammar, J. (n.d.). *The Illustrated Transformer*. The Illustrated Transformer – Jay Alammar – Visualizing machine learning one concept at a time. http://jalammar.github.io/illustrated-transformer/.

[Alammar19] Alammar, J. (n.d.). *The Illustrated GPT-2 (Visualizing Transformer Language Models)*. The Illustrated GPT-2 (Visualizing Transformer Language Models) – Jay Alammar – Visualizing machine learning one concept at a time. https://jalammar.github.io/illustrated-gpt2/.

[BKH16] Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016). Layer normalization. *arXiv preprint arXiv:1607.06450*.

[BMR+20] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., ... & Amodei, D. (2020). Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*.

[BV19] Budzianowski, P., & Vulić, I. (2019). Hello, it's GPT-2--how can I help you? towards the use of pretrained language models for task-oriented dialogue systems. *arXiv preprint arXiv:1907.05774*.

[CA21] Chiu, K. L., & Alexander, R. (2021). Detecting Hate Speech with GPT-3. *arXiv preprint arXiv:2103.12407*.

[Dale21] Dale, R. (2021). GPT-3: What's it good for?. *Natural Language Engineering*, *27*(1), 113-118.

[EC20] Elkins, K., & Chun, J. (2020). Can GPT-3 pass a writer's Turing test. *Journal of Cultural Analytics*, *2371*, 4549.

[FC20] Floridi, L., & Chiriatti, M. (2020). GPT-3: Its nature, scope, limits, and consequences. *Minds and Machines*, *30*(4), 681-694.

[FZS21] Fedus, W., Zoph, B., & Shazeer, N. (2021). Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. *arXiv preprint arXiv:2101.03961*.

[GLT+20] Guu, K., Lee, K., Tung, Z., Pasupat, P., & Chang, M. W. (2020). Realm: Retrieval-augmented language model pre-training. *arXiv preprint arXiv:2002.08909*.

[Hess19] Hess, A. (2019, September 23). *Here's how many hours American workers spend on email each day*. CNBC. https://www.cnbc.com/2019/09/22/heres-how-many-hours-american-workers-spend-on-email-each-day.html.

[IYA16] Goodfellow, Ian; Bengio, Yoshua; Courville, Aaron (2016). "6.2.2.3 Softmax Units for Multinoulli Output Distributions". *Deep Learning. MIT Press. pp. 180–184. ISBN 978-0-26203561-3*

[JCW+17] Joshi, M., Choi, E., Weld, D. S., & Zettlemoyer, L. (2017). Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension. *arXiv preprint arXiv:1705.03551*.

[Kazemnejad19] Kazemnejad, A. *Transformer Architecture: The Positional Encoding*. Transformer Architecture: The Positional Encoding - Amirhossein Kazemnejad's Blog. (n.d.). https://kazemnejad.com/blog/transformer_architecture_positional_encoding/.

[Kulshrestha19] Kulshrestha, R. (2020, October 26). *NLP 101: Word2Vec - Skip-gram and CBOW*. Medium. https://towardsdatascience.com/nlp-101-word2vec-skip-gram-and-cbow-93512ee24314.

[LH20] Lee, J. S., & Hsiang, J. (2020). Patent claim generation by fine-tuning OpenAI GPT-2. *World Patent Information*, *62*, 101983.

[LSZ+21] Liu, J., Shen, D., Zhang, Y., Dolan, B., Carin, L., & Chen, W. (2021). What Makes Good In-Context Examples for GPT-$3?. *arXiv preprint arXiv:2101.06804*.

[MCC+13] Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*

[McCormick16] *"Word2Vec Tutorial - The Skip-Gram Model." Word2Vec Tutorial - The Skip-Gram Model · Chris McCormick, 19 Apr. 2016, mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/.*

[MKM+94] Marcus, M., Kim, G., Marcinkiewicz, M. A., MacIntyre, R., Bies, A., Ferguson, M., ... & Schasberger, B. (1994). The Penn treebank: Annotating predicate argument structure. In *HUMAN LANGUAGE TECHNOLOGY: Proceedings of a Workshop held at Plainsboro, New Jersey, March 8-11, 1994*.

[MLS13] Mikolov, T., Le, Q. V., & Sutskever, I. (2013). Exploiting similarities among languages for machine translation. *arXiv preprint arXiv:1309.4168*.

[MN15] Massaro, T. M., & Norton, H. (2015). Siri-ously? Free speech rights and artificial intelligence. *Nw. UL Rev.*, *110*, 1169.

[MN20] McGuffie, K., & Newhouse, A. (2020). The radicalization risks of GPT-3 and advanced neural language models. *arXiv preprint arXiv:2009.06807*.

[PB18] Popel, M., & Bojar, O. (2018). Training tips for the transformer model. *arXiv preprint arXiv:1804.00247*.

[Philip94] Gage, Philip (1994). *"A New Algorithm for Data Compression". The C User Journal.*

[PKL+16] Paperno, D., Kruszewski, G., Lazaridou, A., Pham, Q. N., Bernardi, R., Pezzelle, S., ... & Fernández, R. (2016). The LAMBADA dataset: Word prediction requiring a broad discourse context. *arXiv preprint arXiv:1606.06031*.

[RNS+18] Radford, A., Narasimhan, K., Salimans, T., & Sutskever, I. (2018). Improving language understanding by generative pre-training.

[RWC+19] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). Language models are unsupervised multitask learners. *OpenAI blog*, *1*(8), 9.

[SKF99] Shibata, Y., Kida, T., Fukamachi, S., Takeda, M., Shinohara, A., Shinohara, T., & Arikawa, S. (1999). *Byte Pair encoding: A text compression scheme that accelerates pattern matching*. Technical Report DOI-TR-161, Department of Informatics, Kyushu University.

[Szogyenyi20] Szogyenyi, Z. (2020, July 24). *We built an OpenAI powered Tailwind CSS code generator using GPT-3*. Themesberg Blog. https://themesberg.com/blog/tailwind-css/gpt-3-tailwind-css-ai-code-generator.

[TJH+21] Thiergart, J., Huber, S., & Übellacker, T. (2021). Understanding Emails and Drafting Responses--An Approach Using GPT-3. *arXiv preprint arXiv:2102.03062*.

[VN20] de Vries, W., & Nissim, M. (2020). As good as new. How to successfully recycle English GPT-2 to make models for other languages. *arXiv preprint arXiv:2012.05628*.

[VSP+17] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. *arXiv preprint arXiv:1706.03762*.

[WPN+19] Wang, A., Pruksachatkun, Y., Nangia, N., Singh, A., Michael, J., Hill, F., ... & Bowman, S. R. (2019). Superglue: A stickier benchmark for general-purpose language understanding systems. *arXiv preprint arXiv:1905.00537*.