Generative Pre-Trained Transformers: Architecture, Pre-Training and Fine-Tuning

Sušac, Ivo

Josip Juraj Strossmayer University of Osijek
School of Applied Mathematics and Informatics

Undergraduate University Study in Mathematics and Computer Science

Generative Pre-Trained Transformers: Architecture, Pre-Training and Fine-Tuning

Undergraduate Thesis / Final Paper

Supervisor:
Dr. Domagoj Matijević, Associate Professor

Student:
Ivo Sušac

Osijek, 2024

Contents

1 | Introduction

Large language models (LLMs) are deep neural network models that have revolutionized the field of natural language processing (NLP) and amassed large popularity in recent years due to their exceptional ability to interpret and generate human language. Most modern LLMs rely on the *Transformer* [1] architecture which has excelled at various NLP tasks such as machine translation and document generation. A specifically popular subset of LLMs are Generative Pretrained Transformers (GPTs). First developed by OpenAI [2], GPT models (such as those powering the very successful ChatGPT) have surpassed traditional NLP models in both versatility and performance by pre-training on vast amounts of text data and fine-tuning on specific tasks. They also show great execution of zero-shot and few-shot learning tasks [3].

GPT models generate text using an advanced pipeline with multiple stages (1.1), each critical to the model's overall performance. The pipeline begins with text preprocessing, where raw textual data is projected into a vector space and prepared for further analysis. These vector representations, called *embeddings* , serve as input for the Transformer decoder blocks, which implement mechanisms for context-aware language modeling such as *self-attention*. After a few postprocessing steps, the model generates output text.

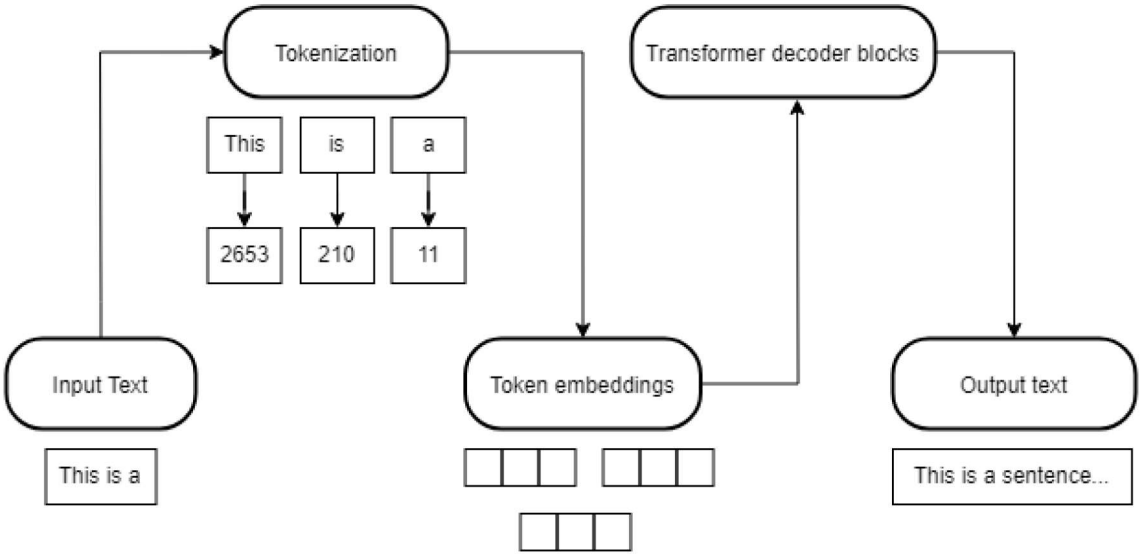

Figure 1.1: The text generation pipeline of a GPT model.

To generate coherent text, the model is subjected to pretraining on massive datasets, learning sophisticated language patterns and next word prediction by leveraging self-supervised learning. Finally, the model undergoes fine-tuning on specific datasets to specialize in particular tasks, such as text classification or question answering.

This thesis aims to explore and dissect each stage of this process, providing a comprehensive explanation of how Generative Pretrained Transformers are built and what makes them such a powerful tool in the field of NLP.

2 | Text data preprocessing

2.1 Tokenization

For machines, interpreting unstructured text data is inherently challenging. Tokenization solves this problem by converting raw text into a structured format by breaking the text down into smaller units called tokens, followed by assigning a unique ID to each token to get their numerical representation. These tokens can represent various elements of the text, such as words, subwords, or even individual characters, depending on the chosen tokenization strategy. Each token can be analyzed in the context of the tokens surrounding it, allowing for high-quality sequential text generation.

There are several tokenization techniques, each offering distinct benefits and trade-offs.

2.1.1 Word-Based Tokenization

The most commonly used tokenization technique is word-based tokenization. It simply splits a batch of text into words using a delimiter, typically a whitespace. Punctuation characters are also accounted for, being encoded as separate entries. This is an important step, as the model's vocabulary shouldn't include every possible punctuation of every word.

Word-based tokenization is straightforward to implement using Python's `re` library. An example implementation is provided below.

```
import re

def word_tokenizer(text):
    tokens = re.findall(r'\w+|[^\w\s]', text)
    return tokens
```

Running the above function on the text: `"Hello, world!  How's everything?"` gives us the resulting output:

```
['Hello', ',', 'world', '!', 'How', "'", 's', 'everything', '?']
```

To map the previously generated tokens into unique token IDs, a vocabulary is built. The vocabulary defines the mapping of each word and special character to a unique integer value. It also contains an `<|unk|>` token to handle out-of-vocabulary words and an `<|endoftext|>` token to add a padding between unrelated texts (which helps the model understand which parts of the training data are connected). This process is illustrated in the figure below.

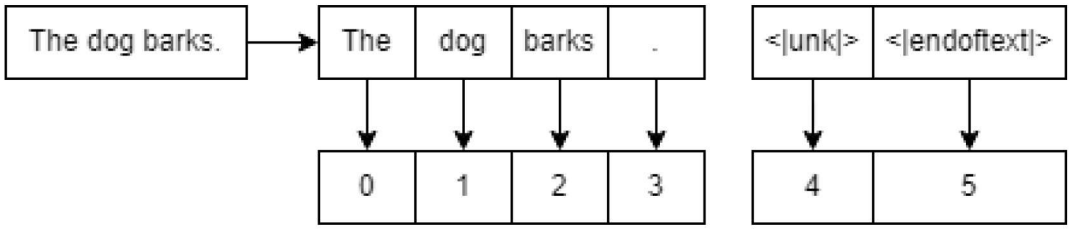

Figure 2.1: Example token IDs for a 6-word vocabulary consisting of the following tokens: *"The", "dog", "barks", ".", "<|unk|>", "<|endoftext|>"*.

While this tokenization technique is widely used, it has some drawbacks. Since every distinct word in the training data will be assigned a corresponding token, the vocabulary size can become extremely large, which will increase memory usage and may result in slower model performance. Furthermore, words not encountered during training will not be represented in the vocabulary, impacting the model's ability to generalize to unseen data or handle languages with a large number of unique words.

2.1.2 Character-Based Tokenization

In character-based tokenization, each token represents a character in the text. Since a language has many different words but only a limited, comparatively small amount of characters, this type of tokenization leads to a small vocabulary, reducing memory and time complexity. Another major benefit of character-level tokenization is that the vocabulary will include all possible characters, eliminating the out-of-vocabulary problem.

However, representing text at the character level increases the length of token sequences by a significant margin, which can lead to longer training and greater memory requirements. Also, individual characters lack semantic meaning in language, which makes it harder for models to capture context-relevant relationships effectively.

2.1.3 Byte-Pair Encoding

Byte-Pair Encoding (BPE) is a subword tokenization technique that serves as a compromise between word-based and character-based tokenization. First described in [4] as an algorithm for data compression, it has since been modified to serve as a tokenization algorithm which guarantees that frequently occurring words are represented as single tokens in the vocabulary, while less common

words are split into two or more subword tokens.

Firstly, a base vocabulary will be created, consisting only of the characters contained in the words. In real-world scenarios, this initial vocabulary would include all ASCII characters and possibly some Unicode characters. After getting the base vocabulary, new tokens are added until a set vocabulary size is reached. This step is done by learning merge rules that specify how two existing tokens can be combined into a new token. During each step of BPE, the most frequent pair of consecutive tokens in the corpus is identified and merged into a new token. This process repeats iteratively as the algorithm forms longer subword tokens.

As a simple example ([5]), let's consider a vocabulary that consists of five words: `"hug"`, `"pug"`, `"pun"`, `"bun"` and `"hugs"`. Let's assume the words have the following frequencies:

```
("hug", 10), ("pug", 5), ("pun", 12), ("bun", 4), ("hugs", 5)
```

which means that "hug" appeared 10 times in the corpus, "pug" appeared 5 times, "pun" appeared 12 times, "bun" appeared 4 times and "hugs" appeared 5 times. The first step is to split each word into individual characters:

```
("h" "u" "g", 10), ("p" "u" "g", 5), ("p" "u" "n", 12), ("b" "u"
"n", 4), ("h" "u" "g" "s", 5)
```

The algorithm then looks for the most frequent token pairs. In this case, the pair ("u", "g") appears 20 times, making it the most common pair. As a result, the first learned merge rule is ("u", "g") -> "ug", adding "ug" to the vocabulary. The corpus is also updated accordingly, replacing instances of "u" and "g" with "ug." Next, the algorithm finds the pair ("u", "n"), which appears 16 times, and merges it to form the token "un." Let's look at the state of the vocabulary and corpus after these changes:

```
Vocabulary: ["b", "g", "h", "n", "p", "s", "u", "ug", "un"]
Corpus: ("h" "ug", 10), ("p" "ug", 5), ("p" "un", 12), ("b" "un",
4), ("h" "ug" "s", 5)
```

The process continues with the most frequent pair ("h", "ug") being merged to form "hug." This leads to the creation of the first three-character token.

```
Vocabulary: ["b", "g", "h", "n", "p", "s", "u", "ug", "un", "hug"]
Corpus: ("hug", 10), ("p" "ug", 5), ("p" "un", 12), ("b" "un", 4),
("hug" "s", 5)
```

The process of identifying and merging the most frequent pairs continues iteratively until the desired vocabulary size is achieved.

Byte-Pair Encoding tokenization solves problems word-based and character-based techniques struggle with. It can represent any word by breaking them into subword tokens, removing the out-of-vocabulary problem. The vocabulary size is controllable, which balances the trade-off between word-based and character-based tokenizers. It also excels in representing semantic meaning of text since it can tokenize recurring subword fragments. Byte-Pair Encoding is used by many Transformer models, including GPT-2, GPT-3, BART, RoBERTa etc.

An example of tokenizing text with Byte-Pair Encoding can be shown using `tiktoken`, a Python open-source library which efficiently implements BPE:

```
import tiktoken

# we load in the GPT-2 BPE tokenizer with a total vocabulary
    size of 50257
tokenizer = tiktoken.get_encoding("gpt2")

encoded_txt = tokenizer.encode("Hello, world! How's everything?"
    )
print(encoded_txt)
```

The above example will give the following output:

```
[15496, 11, 995, 0, 1374, 338, 2279, 30]
```

We observe that the `encode` function performs two steps in one, splitting the text into tokens and converting them into token IDs.

2.2 Embeddings

The final step in preparing input text for training involves transforming token IDs into embedding vectors, which are essential for representing tokens in a continuous vector space.

2.2.1 Token Embeddings

Let the vocabulary size be V and let each token be represented by an index $i \in \{0, 1, ..., V-1\}$. Instead of using one-hot encodings, where each token is a vector $\mathbf{e}_i \in \mathbb{R}^V$ with all zeros except for a one on index i, we use embeddings, where each token is mapped into a typically lower-dimensional space $\mathbb{R}^d$, where $d < V$. During training, the model learns to optimize an embedding matrix $\mathbf{W} \in \mathbb{R}^{V \times d}$, where each row $\mathbf{W}_i$ represents the dense embedding vector for token i. This is done via backpropagation, where the weight matrix is updated with gradient descent to better fit the task at hand.

A key advantage embeddings hold over one-hot encodings is that they have the ability to encode semantic relationships between tokens. For instance, in an optimized embedding space, the vectors for the words "king" and "queen" would be closer to each other when using a distance metric such as cosine similarity. Given two n-dimensional vectors x and y, the cosine similarity can be computed as

$$\text{similarity}(x, y) = \frac{x \cdot y}{\|x\| \|y\|}$$

where $x \cdot y$ represents the dot product between the two vectors.

An interesting phenomenon identified in commonly used word embedding models such as Word2Vec [6] is the ability to solve analogies like “man is to king as woman is to ..?" with vector addition and subtraction (see [7]).
Optimizable embeddings can be easily implemented using `PyTorch`, a Python machine learning library.
Suppose we have a vocabulary of only 5 words, and we want to create 5-dimensional embeddings. Using our vocabulary size and vector dimension, we can instantiate an embedding layer:

```
import torch

vocab_size = 5
embedding_dim = 5

embedding_layer = torch.nn.Embedding(vocab_size, embedding_dim)
print(embedding_layer.weight)
```

The code above returns the underlying weight matrix with initial random values that are optimized during training:

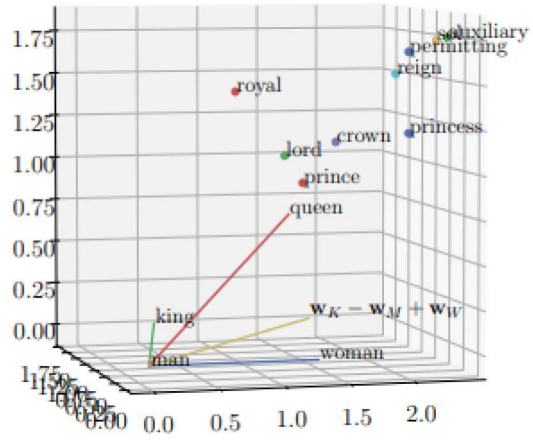

Figure 2.2: The relative locations of word embeddings in a 3-dimensional space for the analogy "man is to king as woman is to ..?". We observe that the closest embedding to the linear combination $w_K - w_M + w_W$ is that of queen. [7]

```
Parameter containing:
tensor([[ 1.2424, -0.5976,  1.1172,  0.2699, -0.3251],
        [-0.8847,  0.4047, -0.3697,  0.6438, -0.0191],
        [ 1.0972,  1.0399, -0.1295, -0.5018, -0.3435],
        [-0.2168,  0.6721,  0.7324,  0.6196,  1.6820],
        [-0.0380,  1.0718,  1.0660, -0.0657,  0.1021]],
    requires_grad=True)
```

If we apply the embedding layer to a token ID of 2:

```
print(embedding_layer(torch.tensor([2])))
```

we obtain the following vector:

```
tensor([[ 1.0972,  1.0399, -0.1295, -0.5018, -0.3435]],
       grad_fn=<EmbeddingBackward0>)
```

We notice that the vector representing token ID 2 is identical to the 3rd row of the weight matrix (indexing starts at 0), meaning that the embedding layer represents a lookup table that retrieves rows using a token ID.

In practice, much larger embedding dimensions are used (for reference, the GPT-3 model uses an embedding dimension of 12,288) to allow the model to capture more detailed relationships between tokens.

2.2.2 Positional Encoding

The embedding layer introduced in the previous chapter always maps the same token ID to the same embedding vector, regardless of the token's positioning in the text. Unlike RNN architectures that process text sequentially and inherently capture positional information, the Transformer architecture processes an entire input sequence at once and lacks the notion of position, an important information for structuring coherent text. The authors of [1] proposed a simple technique leveraging the sine and cosine functions to encode the absolute position of tokens in a sequence:

$$\begin{aligned} PE_{(pos,2i)} &= \sin\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right) \\ PE_{(pos,2i+1)} &= \cos\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right) \end{aligned} \tag{2.1}$$

where pos is the position of the token in the sequence, i is the dimension index and d_{model} is the dimension of the model's embedding space.
These equations leverage sine and cosine functions to generate wave-like patterns that change based on the position in the sequence. By applying sine to even indices and cosine to odd indices, they create a diverse set of features that encode positional information effectively across sequences of varying lengths. The positional encoding vectors are set to the same dimension as the token embedding vectors and their values are added to form a combined representation, which carries both semantic and positional information.

An example implementation is given:

```
import torch
import torch.nn as nn
import numpy as np

class PositionalEncoding(nn.Module):
    def __init__(self, max_len, d_model):
        super(PositionalEncoding, self).__init__()

        position = torch.arange(0, max_len).unsqueeze(1).float()
        div_term = torch.exp(torch.arange(0, d_model, 2).float()
    * -(np.log(10000.0) / d_model))

        pe = torch.zeros(max_len, d_model)
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)

        self.register_buffer('pe', pe)

    def forward(self, x):
        batch_size, seq_len, _ = x.size()
        pe = self.pe[:seq_len].unsqueeze(0)
```

```
21         return x + pe
```

where the inputs to the `forward` pass are tensors of shape `(batch_size, maximum_input_length, model_dimensionality)`.
Given a random tensor of shape (16, 5000, 256), we can print out the first 10 positions of the first batch item:

```
max_len = 5000
d_model = 256
pos_enc = PositionalEncoding(max_len, d_model)

tokens = torch.randn(16, max_len, d_model)
encoded_tokens = pos_enc(tokens)

print(encoded_tokens[0, :10])
```

The above code prints out the following output:

```
tensor([[ 0.3741,  1.4068, -0.1293,  ...,  0.9279,  0.1216,
    0.6628],
        [ 2.0897,  0.2798,  0.4041,  ...,  1.5326,  0.6097,
    0.3615],
        [ 0.6906, -0.4875, -0.2741,  ...,  0.0304, -0.2491,
    -1.6278],
        ...,
        [ 1.1701,  1.5050, -0.7701,  ..., -0.3111, -0.9990,
    1.0434],
        [ 1.5591, -0.7341,  2.7129,  ...,  0.9223,  0.9805,
    -0.0644],
        [ 0.2120, -1.6400,  1.2758,  ..., -1.4612,  1.0807,
    0.7322]])
```

OpenAI's GPT models use positional embeddings that are optimized during model training rather than being predefined by fixed formulas, like 2.1. This embedding layer, along with the weight matrix can be initialized analogously to the token embedding layer introduced in the previous subchapter.

3 | The Transformer Block

The fundamental building block of GPT models is the Transformer decoder, which accepts the embedded tokens as input and relies on the self-attention mechanism and feed forward networks to capture complex dependencies in the data. This block is then repeated an arbitrary amount of times (for reference, GPT-3 uses 96 stacked Transformer layers), followed by additional post processing steps to generate the final output.

3.1 Self-Attention

Self-attention is a mechanism that allows each position in the input sequence to focus on every other position within the same sequence while generating its representation. The goal of the mechanism is to compute a context vector for each input token that aggregates the information from all other input tokens, capturing relationships between them.

3.1.1 Scaled Dot-Product Attention with Casual Masking and Dropout

As the first step of the self-attention mechanism, we compute query, key and value vectors for all input tokens. We do so by initializing trainable weight matrices $\mathbf{W_q}$, $\mathbf{W_k}$ and $\mathbf{W_v}$, and performing matrix multiplication between every input and every weight matrix to project the vectors into a lower-dimensional space. Doing so, we get a query matrix $\mathbf{Q}$, a key matrix $\mathbf{K}$ and a value matrix $\mathbf{V}$. Attention scores are context-specific values that are the result of a dot-product between the query ($\mathbf{Q}$) and the value ($\mathbf{K}$) matrix. Attention scores are then scaled by the square root of the dimension of the keys ($\sqrt{d_k}$) to avoid small gradients during training (when scaling up the embedding dimension large dot-products result in small gradients during backpropagation). The scaled scores are then passed through a softmax function to obtain normalized attention weights, which determine how much emphasis should be placed on each value vector when combining information for each query. We multiply these weights with the value matrix ($\mathbf{V}$) to get the context vectors. A compact formula for scaled dot-product attention is given [1]:

$$
\begin{aligned}
Q = XW_Q, \quad K = XW_K, \quad V = XW_V, \\
\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V
\end{aligned}
\tag{3.1}
$$

Two important techniques used in GPT models to improve the attention mechanism are casual masking and dropout. In standard self-attention, each query attends to every key, including future tokens, which is avoided since predicting the next word should only depend on the current and previous words, not future ones. To prevent this, a mask is applied to the attention weights matrix, making sure that positions corresponding to future tokens are ignored. This is achieved by masking each of the elements above the diagonal with $-\infty$ before applying the softmax function. The softmax function treats $-\infty$ values as zero probability, effectively making the attention weights for future tokens zero.

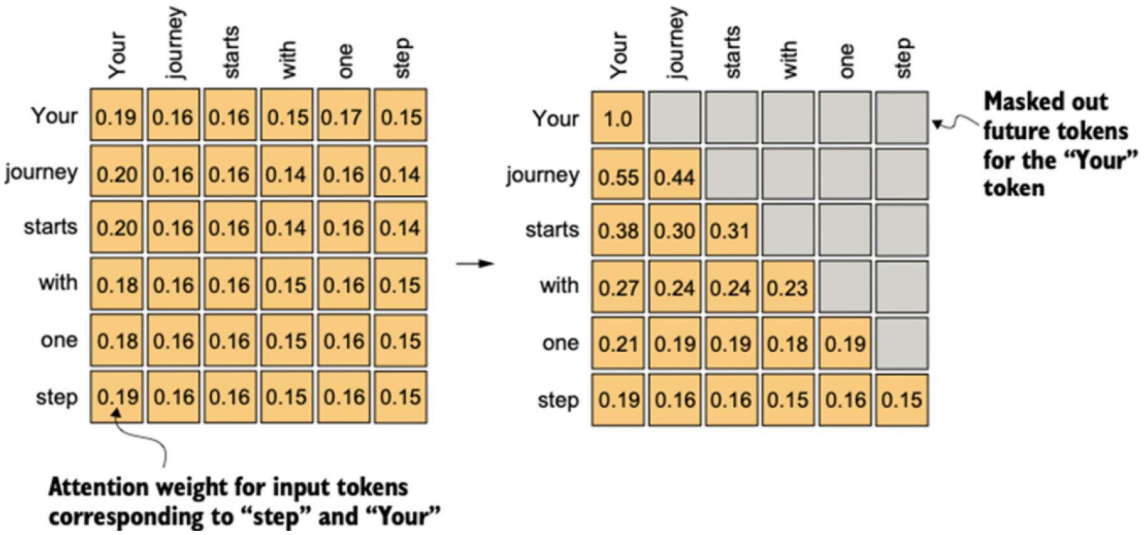

Figure 3.1: Attention weights for the sequence "Your journey starts with one step" before and after casual masking. [9]

Dropout is a regularization technique used to prevent overfitting used in multiple parts of the GPT architecture. Dropout randomly sets a subset of weights during training to zero, preventing over-reliance on a specific set of weights. It's only used during training and is disabled in inference. In terms of the attention mechanism, dropout is used after computing the masked attention weights. It's usually applied with a probability p, meaning that each attention weight has a p-chance of being zeroed out during training.

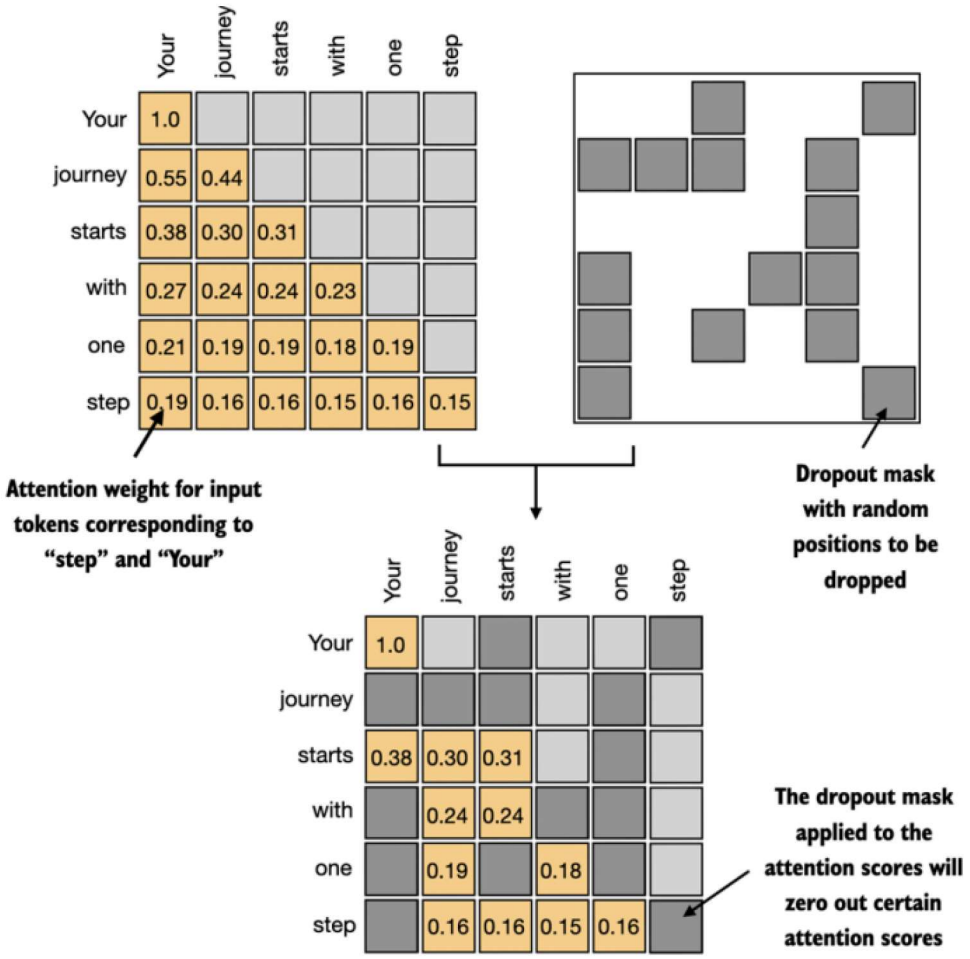

Figure 3.2: Attention weights for the sequence "Your journey starts with one step" before and after casual masking and dropout. [9]

This mechanism can be implemented in a compact Python class, using `PyTorch`'s `nn.Module`:

```
class CasualAttention(nn.Module):
    def __init__(self, dim_in, dim_out, context_size, dropout, qkv_bias=False):
        super().__init__()
        self.dim_in = dim_in
        self.dim_out = dim_out
        self.W_q = nn.Linear(dim_in, dim_out, bias=qkv_bias)
        self.W_k = nn.Linear(dim_in, dim_out, bias=qkv_bias)
        self.W_v = nn.Linear(dim_in, dim_out, bias=qkv_bias)
        self.dropout = nn.Dropout(dropout)
        # we create a mask in the constructor so that we don't have to create it every time we call the forward method
        # we use register_buffer because it's a parameter that we don't want to update during training but we can still save it in the model state dict
        # it also automatically moves the tensor to the device that the model is on so we don't have to do it manually
```

```
13          self.register_buffer("mask", torch.triu(torch.ones(
    context_size, context_size), diagonal=1))
14
15      def forward(self, x):
16          _, num_tokens, _ = x.shape # the first dimension is the
    batch size, the second is the number of tokens in the
    sequence, the third is the embedding dimension
17          q = self.W_q(x)
18          k = self.W_k(x)
19          v = self.W_v(x)
20
21          attention_scores = torch.matmul(q, k.transpose(1, 2))
22          # masked_fill is conceptually the same as
    attention_scores[self.mask == 1] = float("-inf")
23          attention_scores.masked_fill_(self.mask[:num_tokens, :
    num_tokens] == 1, -torch.inf) # we slice the mask to match
    the number of tokens in the input
24          # adding an underscore to a function in pytorch means
    that the function will modify the tensor in place
25          attention_weights = torch.softmax(attention_scores / k.
    shape[-1]**0.5, dim=-1)
26          attention_weights = self.dropout(attention_weights)
27
28          context_vectors = torch.matmul(attention_weights, v)
29          return context_vectors
```

3.1.2 Multi-Head Attention

In multi-head attention, the input embeddings projected into multiple lower-dimensional spaces, using different weight matrices for each head:

$$Q_h = XW_Q^h, \quad K_h = XW_K^h, \quad V_h = XW_V^h, \ for\ each\ head\ h \tag{3.2}$$

For each head, scaled dot-product attention is computed separately. Each head computes the attention scores between the projected queries and keys, then uses these scores to weigh the values:

$$\text{Attention}_h(Q_h, K_h, V_h) = \text{softmax}\left(\frac{Q_h K_h^T}{\sqrt{d_k}}\right) V_h \tag{3.3}$$

Since each head has different attention scores, they capture different types of relationships between tokens. Once attention has been computed for each head, the outputs from all heads are concatenated:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \text{head}_2, \ldots, \text{head}_h)W_O \tag{3.4}$$

where W_O is a learned weight matrix that transforms the concatenated output back to the original dimension of the input sequence.

Multi-head attention can also be implemented in a compact class:

```
class MultiHeadAttention(nn.Module):
    def __init__(self, dim_in, dim_out, context_size, dropout, num_heads, qkv_bias=False):
        super().__init__()
        self.dim_in = dim_in
        self.dim_out = dim_out
        self.num_heads = num_heads
        # we split the output dimension by the number of heads
        self.head_dim = dim_out // num_heads
        self.W_q = nn.Linear(dim_in, dim_out, bias=qkv_bias)
        self.W_k = nn.Linear(dim_in, dim_out, bias=qkv_bias)
        self.W_v = nn.Linear(dim_in, dim_out, bias=qkv_bias)
        self.W_o = nn.Linear(dim_out, dim_out)
        self.dropout = nn.Dropout(dropout)
        self.register_buffer("mask", torch.triu(torch.ones(context_size, context_size), diagonal=1))

    def forward(self, x):
        # the context vectors will be of dimension (num_heads * dim_out)
        b, num_tokens, _ = x.shape
        q = self.W_q(x) # matrix multiplication of batched data, now the q tensor is of shape (b, num_tokens, dim_out)
        k = self.W_k(x)
        v = self.W_v(x)

        # we permute our tensors so that it's now (batch_size, num_heads, num_tokens, head_dim)
        # that makes it more intuitive and easier to process each head independently
        # for example, we have 1 batch, 2 heads each processing 3 tokens with a head dimension of 4
        q = q.view(b, num_tokens, self.num_heads, self.head_dim).permute(0, 2, 1, 3)
        k = k.view(b, num_tokens, self.num_heads, self.head_dim).permute(0, 2, 1, 3)
        v = v.view(b, num_tokens, self.num_heads, self.head_dim).permute(0, 2, 1, 3)

        # we want to do dot product between the queries and keys for each head
        # the matrix multiplication is carried out between the last two dimensions of the tensors and then repeated for all the heads
        attention_scores = torch.matmul(q, k.permute(0, 1, 3, 2))
        attention_scores.masked_fill_(self.mask[:num_tokens, :num_tokens] == 1, -torch.inf)

        attention_weights = torch.softmax(attention_scores / k.shape[-1]**0.5, dim=-1)
        attention_weights = self.dropout(attention_weights)

```

```
        # we multiply the attention weights with the values and
    then concatenate the heads
        context_vectors = torch.matmul(attention_weights, v).
    transpose(1, 2).reshape(b, num_tokens, self.dim_out)
        # we pass the concatenated heads through a linear layer
    to get the final context vectors
        context_vectors = self.W_o(context_vectors)

        return context_vectors
```

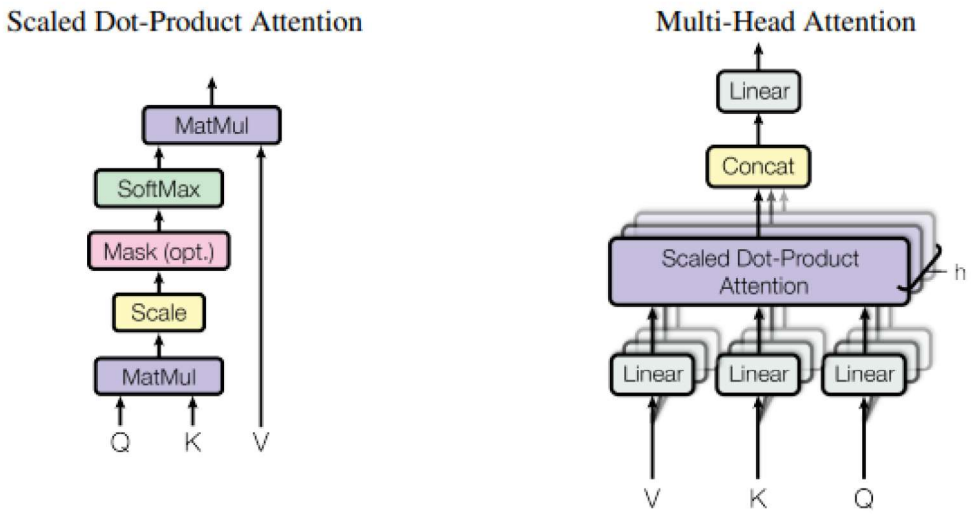

Figure 3.3: Scaled Dot-Product and Multi-Head Attention visualization from the original Transformer paper. [1]

3.2 Layer Normalization

In deep neural networks, internal covariate shift [10] occurs when the distribution of inputs to a neural network layer changes during training due to ongoing updates in the parameters of previous layers. This forces each layer to adjust to the varying input distributions, potentially making the training process slower and more complex. Let $a^{(l)}$ be the activation in layer l and $W^{(l)}$ be the weight parameters of layer l. As the distribution of $a^{(l)}$ changes, the gradient $\frac{\partial L}{\partial W^{(l)}} = \frac{\partial L}{\partial a^{(l)}} \cdot \frac{\partial a^{(l)}}{\partial W^{(l)}}$ can become unstable which can lead to vanishing or exploding gradients. Layer normalization [11] adjusts the activations of a layer to have a mean of 0 and a variance of 1, ensuring reliable training. Let $x = (x_1, ..., x_H)$ be the vector representation of an input of size H to the normalization layer. Layer normalization will re-center the input as:

$$
\begin{aligned}
h &= \gamma \cdot N(x) + \beta, \\
N(x) &= \frac{x - \mu}{\sigma}, \\
\mu &= \frac{1}{H} \sum_{i=1}^{H} x_i \\
\sigma &= \sqrt{\frac{1}{H} \sum_{i=1}^{H} (x_i - \mu)^2}
\end{aligned}
\tag{3.5}
$$

where h is the output of the normalization layer, μ and σ are the mean and standard deviation of the input, respectively. The parameters γ and β are trainable parameters that are adjusted during training to improve performance. In practice, we can also add a small fixed parameter ϵ to the variance to prevent division by zero.

The LayerNorm class can be implemented as follows:

```
class LayerNorm(nn.Module):
    def __init__(self, emb_dim, eps=1e-6):
        super().__init__()
        # the model can learn to scale or shift the normalized values by using the gamma and beta parameters if it needs to (if it improves the performance)
        self.gamma = nn.Parameter(torch.ones(emb_dim))
        self.beta = nn.Parameter(torch.zeros(emb_dim))
        self.eps = eps # small constant added to the variance to prevent division by zero

    def forward(self, x):
        mean = x.mean(dim=-1, keepdim=True)
        var = x.var(dim=-1, keepdim=True)
        normalized = (x - mean) / torch.sqrt(var + self.eps)
        return self.gamma * normalized + self.beta
```

3.3 Fully-Connected Layer

A fully connected feed-forward network is the last component of the Transformer decoder block. It consists of two layers, one that projects the inputs into a larger space and one that shrinks the outputs to the original input dimensions. This allows for exploration of a bigger space, providing the network with greater capacity to capture complex representations. It also introduces non-linearity by applying a non-linear activation function in between the projections.
The most commonly used activation function is the Rectified Linear Unit (ReLU):

$$\text{ReLU}(x) = \max(0, x)$$

Another commonly used activation function is the Gaussian Error Linear Unit (GELU), which has been shown to slightly improve performance in comparison to ReLU [12].

$$\text{GELU}(x) = x \cdot P(X \leq x) = x \cdot \frac{1}{2}\left[1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right]$$

where $\text{erf}(x)$ is the Gaussian error function. GELU can also be approximated more efficiently by the following formula:

$$\text{GELU}(x) \approx 0.5x\left(1 + \tanh\left(\sqrt{\frac{2}{\pi}}\left(x + 0.044715x^3\right)\right)\right)$$

Unlike ReLU, which introduces sharp thresholds by zeroing out negative values, GELU applies a smooth, continuous curve which mitigates the "dead neuron" problem (for example, a large gradient flowing through a ReLU neuron could cause a weight update such that the gradient flowing through that neuron will forever be zero from that point on).

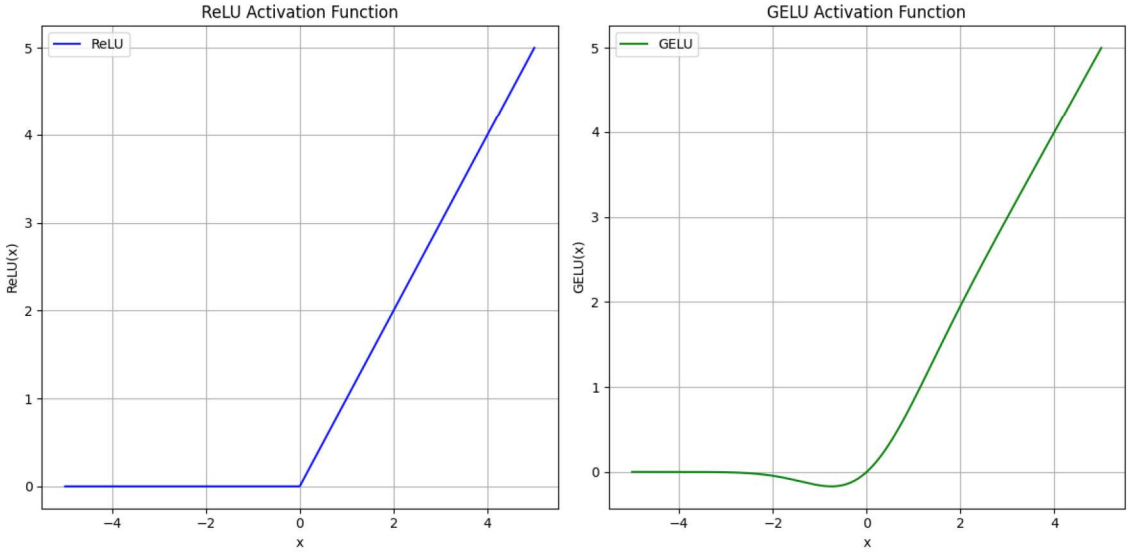

Figure 3.4: The output graph of ReLU and GELU activation functions.

The fully connected layer can be implemented as follows:

```
class FullyConnected(nn.Module):
    def __init__(self, embedding_dim):
        super().__init__()
        self.layers = nn.Sequential(
            #projecting to a 4x larger space and back
            nn.Linear(embedding_dim, 4 * embedding_dim),
            nn.GELU(),
            nn.Linear(4 * embedding_dim, embedding_dim)
        )

    def forward(self, x):
        return self.layers(x)
```

To improve our model, we add dropout layers after the attention and the feed-forward layers. Furthermore, we add residual connections [1]. A residual connection allows the input to a layer to be added directly to the output of a later layer. This is done to help mitigate the vanishing gradient problem, as it allows gradients to bypass intermediate layers that cause their vanishing.

After passing through the Transformer blocks, outputs are again normalized with another LayerNorm layer. Finally, outputs are projected into the vocabulary space of the tokenizer using a fully-connected layer and softmax is applied to represent the next token probabilities.

The full GPT architecture can be visualized below.

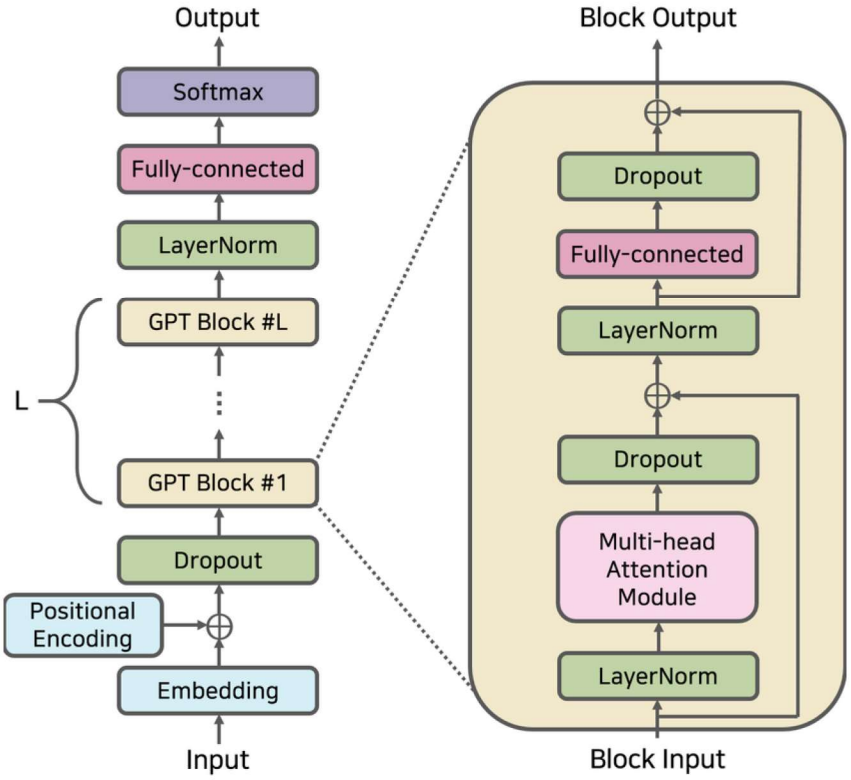

Figure 3.5: The full GPT model architecture with L repeating Transformer decoder blocks. [8]

4 | Pre-Training

Pre-training involves training the model on a large corpus of raw text data. The goal is to develop a versatile language model that can generate coherent and contextually relevant text. The model is trained to minimize the cross entropy loss function:

$$H(p,q) = -\sum p(x) \log q(x)$$

where p is the actual probability distribution (in the context of GPTs, the true probability of a word being the next in a sequence), q is the predicted probability distribution, and the sum is over all possible outcomes. The minimization is done using algorithms based on stochastic gradient descent (SGD), such as Adam.

4.1 Adam Optimizer

Adam [13] is an adaptive learning rate algorithm designed to help deep neural networks achieve convergence quickly. In the standard gradient descent algorithm:

$$\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}(\theta)$$

where θ represents the parameters of the model, α is the learning rate and $\nabla_\theta \mathcal{L}(\theta)$ is the gradient of the loss function with respect to θ,
the learning rate is fixed. The inherent problem with having a fixed learning rate is that a lower initial learning rate would sometimes lead to very slow convergence, while a very high rate at the start might miss the minima. Adam solves this problem by adapting the learning rate for each parameter separately, combining momentum-based SGD with RMSProp (Root Mean Square Propagation).

For each time iteration t, Adam computes the gradient g_t. Then, it updates the first-moment vector m, responsible for storing the moving average of the gradients:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) \nabla_\theta \mathcal{L}(\theta)$$

where m_t is the first-moment vector at time t and β_1 is the decay rate for the first moment estimate.
Similarly, the second-moment estimate vector v, which stores squared gradients, is updated:

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\nabla_\theta \mathcal{L}(\theta))^2$$

where v_t is the second-moment vector at time t and β_2 is the decay rate for the second moment estimate.
Since m and v are initially set to 0 (which leads to a bias towards zero), a correction of bias in the moments is performed:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Combining the above formulas, the overall parameter update rule in Adam is:

$$\theta_t \leftarrow \theta_{t-1} - \frac{\alpha \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

where ϵ is added to prevent division by zero and maintain numerical stability.

4.2 Training Loop

Assuming a well implemented `DataLoader` that samples data using a sliding window technique, model training can be implemented as a simple `PyTorch` training loop:

```
def train_model(model, train_loader, optimizer, num_epochs, device):
    for epoch in range(num_epochs):
        model.train()
        for input_batch, target_batch in train_loader:
            global_step += 1
            optimizer.zero_grad()
            input_batch, target_batch = input_batch.to(device), target_batch.to(device)
            logits = model(input_batch)
            loss = torch.nn.functional.cross_entropy(logits.flatten(0, 1), target_batch.flatten())
            loss.backward()
            optimizer.step()
```

in which the model optimizes parameters considering all of the training data for multiple epochs. `PyTorch` also supports model saving, which is essential for preserving the trained model and resuming training or performing inference at a later time. This is often done whenever the loss on the validation set decreases.

Training can further be optimized by implementing different heuristics such as learning rate warmup and cosine decay [14].

5 | Fine-Tuning

Fine-tuning represents the process of adjusting parameters of the pre-trained model to better suit a specific task. It can be viewed as further optimizing the model's weights starting from a pre-trained state rather than random initialization. Each fine-tuning step updates the pre-trained weights based on the new task's loss (5.1), gradually adapting the model to the requirements of the fine-tuning task without losing any general knowledge obtained during pre-training. This is done to allow a model to perform a task different from just next-word prediction, the task it was pre-trained on, and is called *transfer learning*.

The most used and straightforward fine-tuning approach is supervised fine-tuning, where the model is trained on a labeled dataset fitting a specific task like classification or question answering. Fine-tuning for question answering enhances a model's capability to understand instructions and generate outputs based on them. On the other hand, classification fine-tuning is used in projects requiring accurate data categorization into predetermined groups, such as sentiment analysis or spam detection. Although fine-tuning for question answering is more versatile, it typically requires larger datasets and greater computational resources than classification fine-tuning.

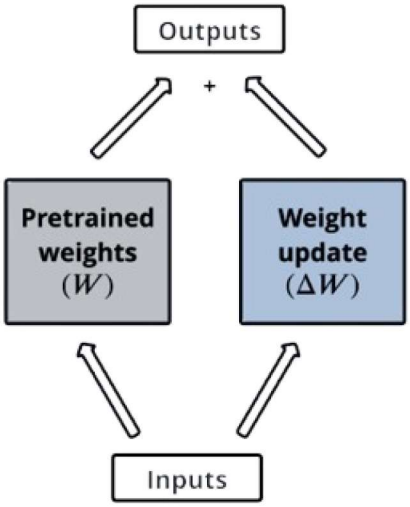

Figure 5.1: Fine tuning the model is nothing more than adding new weight updates to pre-trained weights. [15]

5.1 Parameter Efficient Fine-Tuning

In general, fine-tuning large language models requires substantial computational resources, making it both memory-demanding and time-consuming. To mitigate these challenges, researchers ([16], [17], [18]) have introduced Parameter Efficient Fine-Tuning (PEFT) methods, which optimize the process by updating only a small subset of model parameters.

5.1.1 LoRA Fine-Tuning

LoRA (Low-Rank Adaptation) [16] fine-tuning is a technique designed to reduce the number of trainable weights by learning low-rank updates to the pre-trained model's weight matrices, rather than fine-tuning all parameters.

We assume that the generalized weight matrix of a pre-trained model is $W_0 \in \mathbb{R}^{d \times k}$, where d is the input dimension and k is the output dimension. Fine-tuning without LoRA would involve updating all elements of this matrix. Instead of directly updating W_0 with a weight matrix ΔW, LoRA learns to approximate ΔW by decomposing it into two lower-rank matrices $A \in \mathbb{R}^{d \times r}$ and $B \in \mathbb{R}^{r \times k}$, where $r \ll d$ and $r \ll k$. During training, the pre-trained weights are frozen and only the parameters in matrices A and B are learned. This means that the total number of parameters learned during training is much smaller than original, which makes LoRA a highly parameter-efficient fine-tuning method.

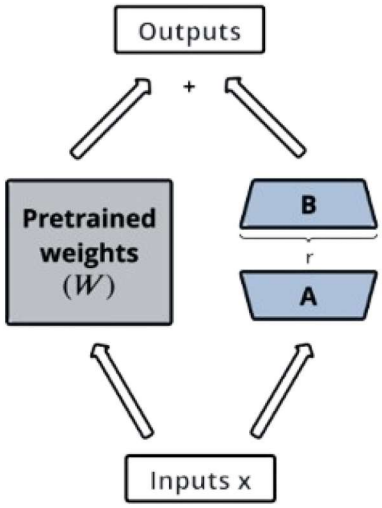

Figure 5.2: LoRA fine tuning splits the weight update matrix into two matrices of lower rank. [15]

Another important aspect of LoRA fine-tuning is that it doesn't lower the model's accuracy [16]. It can also be enhanced with QLoRA [17], a technique that maintains frozen model parameters in 4-bit quantized precision, further reducing memory usage.

5.1.2 Prompt Tuning

Prompt tuning [18] is a method used to fine-tune models by learning task-specific prompts, rather than modifying the model's original weights. The weights of the pre-trained model remain frozen and a trainable tensor is prepended to the model's input embeddings, creating a soft prompt to condition frozen language models to perform specific downstream tasks. These soft prompts are learned through backpropagation and can be further fine-tuned.

A shortcoming of model tuning is that it requires making a task-specific copy of the entire pre-trained model for each downstream task and doesn't support mixed-task inference. Prompt tuning optimizes this, only requiring the storage of a small task-specific prompt for each task, enabling mixed-task inference using the original pre-trained model.

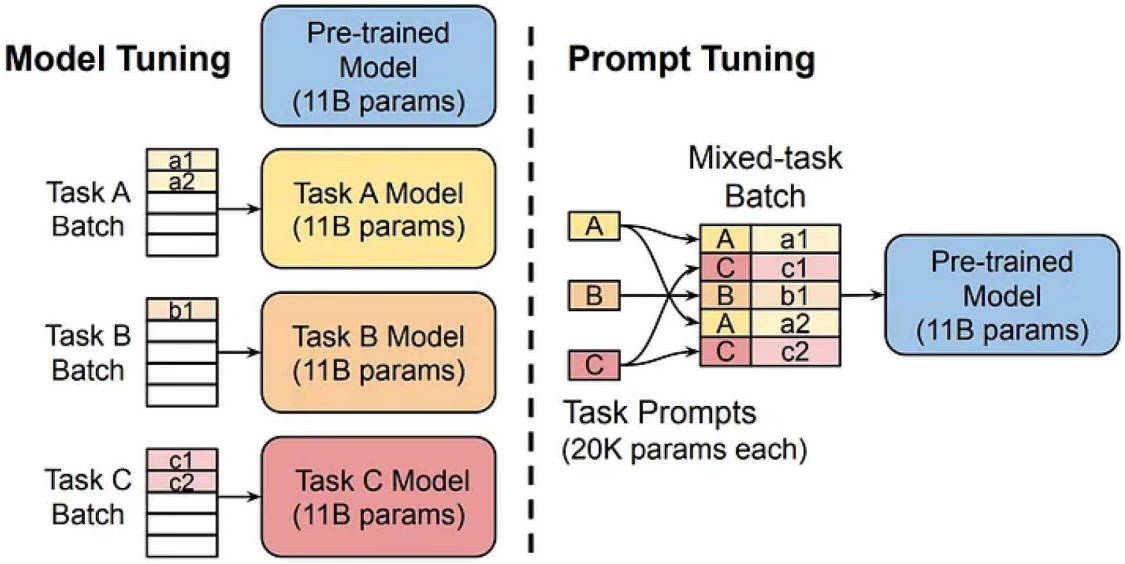

Figure 5.3: Prompt tuning supports mixed-task inference. [18]

With a T5 "XXL" model, each copy of the tuned model requires 11 billion parameters. By contrast, the tuned prompts in [18] would only require 20,480 parameters per task.

On the SuperGLUE benchmark, prompt-tuning task performance rivals that of traditional model tuning, with the gap vanishing as model size increases.

References

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, I. Polosukhin, *Attention is all you need*, arXiv:1706.03762 (2017)

[2] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever, *Improving Language Understanding by Generative Pre-Training*, OpenAI (2018)

[3] T. B. Brown et al., *Language Models are Few-Shot Learners*, arXiv:2005.14165 (2020)

[4] P. Gage, *A New Algorithm for Data Compression*, The C Users Journal (1994)

[5] *HuggingFace: Byte-Pair Encoding tokenization*, `https://huggingface.co/learn/nlp-course/en/chapter6/5`

[6] T. Mikolov, K. Chen, G. Corrado, J. Dean, *Efficient Estimation of Word Representations in Vector Space*, arXiv:1301.3781 (2013)

[7] C. Allen, T. Hospedales, *Analogies Explained: Towards Understanding Word Embeddings*, arXiv:1301.3781 (2019)

[8] *A Mathematical Investigation of Hallucination and Creativity in GPT Models - Scientific Figure on ResearchGate*, `https://www.researchgate.net/figure/Conceptual-architecture-of-a-GPT-model_fig1_370853178`

[9] S. Raschka, *Build a Large Language Model (From Scratch)*, Manning Publications (2024)

[10] S. Ioffe, C. Szegedy, *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, arXiv:1502.03167 (2015)

[11] J. Ba, J. Kiros, G. Hinton, *Layer Normalization*, arXiv:1607.06450 (2016)

[12] D. Hendrycks, K. Gimpel, *Gaussian Error Linear Units (GELUs)*, arXiv:1606.08415 (2016)

[13] D. Kingma, J. Ba, *Adam: A Method for Stochastic Optimization*, arXiv:1412.6980 (2014)

[14] I. Loshchilov, F. Hutter, *SGDR: Stochastic Gradient Descent with Warm Restarts*, arXiv:1608.03983 (2016)

[15] V. Parthasarathy, A. Zafar, A. Khan, A. Shahid, *The Ultimate Guide to Fine-Tuning LLMs from Basics to Breakthroughs: An Exhaustive Review of Technologies, Research, Best Practices, Applied Research Challenges and Opportunities*, `https://arxiv.org/html/2408.13296v1` (2024)

[16] E. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, W. Chen, *LoRA: Low-Rank Adaptation of Large Language Models*, arXiv:2106.09685 (2021)

[17] T. Dettmers, A. Pagnoni, A. Holtzman, L. Zettlemoyer, *TQLoRA: Efficient Finetuning of Quantized LLMs*, arXiv:2305.14314 (2023)

[18] B. Lester, R. Al-Rfou, N. Constant, *The Power of Scale for Parameter-Efficient Prompt Tuning*, arXiv:2104.08691 (2021)

Generative Pre-Trained Transformers: Architecture, Pre-Training and Fine-Tuning

Summary

With an emphasis on GPT models, this thesis explores the design, training, and optimization of large language models (LLMs). It starts by examining preprocessing approaches for text data, such as tokenization techniques (word-based, character-based, Byte-Pair Encoding) and embeddings. Subsequently, the Transformer block was presented, highlighting the mechanisms that constitute the basis of the model: self-attention and fully-connected layers.

The processes involved in optimizing model parameters were highlighted by the examination of the pre-training phase followed by fine-tuning, which demonstrated how models can be adapted efficiently to new tasks, particularly when using parameter-efficient techniques like LoRA and prompt tuning.

Keywords

GPT model, tokenization, embeddings, transformer architecture, self-attention, pre-training, Adam optimizer, fine-tuning, LoRA fine-tuning, prompt tuning

About the author

I am currently an undergraduate student in the Mathematics and Computer Science program at Josip Juraj Strossmayer University of Osijek, School of Applied Mathematics and Informatics. Throughout my years of study, I notably served as a teaching assistant for courses Introduction to Computer Science, Object Oriented Programming and Machine Learning. I've received commendations issued by the School of Applied Mathematics and Informatics for Success in Study, Extracurricular Activities and the Dean's Award. In 2023, I did practical training at Mono d.o.o where I specialized in creating AI solutions.