CHAPTER 9

Transformers

Chapter Goals

In this chapter you will:

- Learn about the origins of GPT, a powerful decoder Transformer model for text generation.
- Learn conceptually how an attention mechanism mimics our way of attaching more importance to some words in a sentence than others.
- Delve into how the attention mechanism works from first principles, including how queries, keys, and values are created and manipulated.
- See the importance of causal masking for text generation tasks.
- Understand how attention heads can be grouped into a multihead attention layer.
- See how multihead attention layers form one part of a Transformer block that also includes layer normalization and skip connections.
- Create positional encodings that capture the position of each token as well as the word token embedding.
- Build a GPT model in Keras to generate the text contained in wine reviews.
- Analyze the output from the GPT model, including interrogating the attention scores to inspect where the model is looking.
- Learn about the different types of Transformers, including examples of the types of tasks that can be tackled by each and descriptions of the most famous state-of-the-art implementations.
- Understand how encoder-decoder architectures work, like Google's T5 model.
- Explore the training process behind OpenAI's ChatGPT.

We saw in Chapter 5 how we can build generative models on text data using recurrent neural networks (RNNs), such as LSTMs and GRUs. These autoregressive models process sequential data one token at a time, constantly updating a hidden vector that captures the current latent representation of the input. The RNN can be designed to predict the next word in a sequence by applying a dense layer and softmax activation over the hidden vector. This was considered the most sophisticated way to generatively produce text until 2017, when one paper changed the landscape of text generation forever.

Introduction

The Google Brain paper, confidently entitled "Attention Is All You Need,"[1] is famous for popularizing the concept of *attention*—a mechanism that now powers most state-of-the-art text generation models.

The authors show how it is possible to create powerful neural networks called *Transformers* for sequential modeling that do not require complex recurrent or convolutional architectures but instead only rely on attention mechanisms. This approach overcomes a key downside to the RNN approach, which is that it is challenging to parallelize, as it must process sequences one token as a time. Transformers are highly paralellizable, allowing them to be trained on massive datasets.

In this chapter, we are going to delve into how modern text generation models make use of the Transformer architecture to reach state-of-the-art performance on text generation challenges. In particular, we will explore a type of autoregressive model known as the *generative pre-trained transformer* (GPT), which powers OpenAI's GPT-4 model, widely considered to be the current state of the art for text generation.

GPT

OpenAI introduced GPT in June 2018, in the paper "Improving Language Understanding by Generative Pre-Training,"[2] almost exactly a year after the appearance of the original Transformer paper.

In this paper, the authors show how a Transformer architecture can be trained on a huge amount of text data to predict the next word in a sequence and then subsequently fine-tuned to specific downstream tasks.

The pre-training process of GPT involves training the model on a large corpus of text called BookCorpus (4.5 GB of text from 7,000 unpublished books of different genres). During pre-training, the model is trained to predict the next word in a sequence given the previous words. This process is known as *language modeling* and is used to teach the model to understand the structure and patterns of natural language.

After pre-training, the GPT model can be fine-tuned for a specific task by providing it with a smaller, task-specific dataset. Fine-tuning involves adjusting the parameters of the model to better fit the task at hand. For example, the model can be fine-tuned for tasks such as classification, similarity scoring, or question answering.

The GPT architecture has since been improved and extended by OpenAI with the release of subsequent models such as GPT-2, GPT-3, GPT-3.5, and GPT-4. These models are trained on larger datasets and have larger capacities, so they can generate more complex and coherent text. The GPT models have been widely adopted by researchers and industry practitioners and have contributed to significant advancements in natural language processing tasks.

In this chapter, we will build our own variation of the original GPT model, trained on less data, but still utilizing the same components and underlying principles.

Running the Code for This Example

The code for this example can be found in the Jupyter notebook located at *notebooks/09_transformer/01_gpt/gpt.ipynb* in the book repository.

The code is adapted from the excellent GPT tutorial created by Apoorv Nandan available on the Keras website.

The Wine Reviews Dataset

We'll be using the Wine Reviews dataset that is available through Kaggle. This is a set of over 130,000 reviews of wines, with accompanying metadata such as description and price.

You can download the dataset by running the Kaggle dataset downloader script in the book repository, as shown in Example 9-1. This will save the wine reviews and accompanying metadata locally to the */data* folder.

Example 9-1. Downloading the Wine Reviews dataset

```
bash scripts/download_kaggle_data.sh zynicide wine-reviews
```

The data preparation steps are identical to the steps used in Chapter 5 for preparing data for input into an LSTM, so we will not repeat them in detail here. The steps, as shown in Figure 9-1, are as follows:

1. Load the data and create a list of text string descriptions of each wine.
2. Pad punctuation with spaces, so that each punctuation mark is treated as a separate word.
3. Pass the strings through a `TextVectorization` layer that tokenizes the data and pads/clips each string to a fixed length.
4. Create a training set where the inputs are the tokenized text strings and the outputs to predict are the same strings shifted by one token.

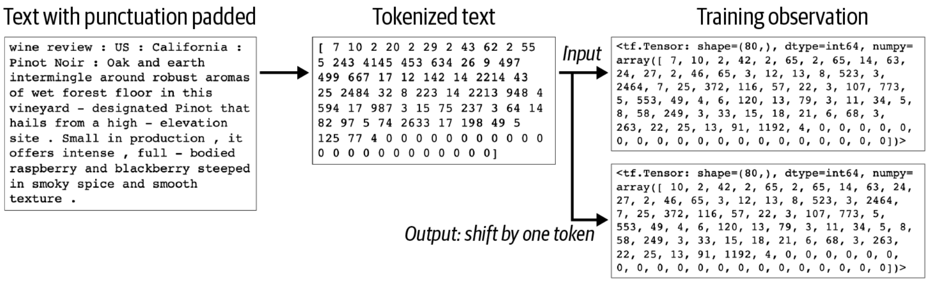

Figure 9-1. Data processing for the Transformer

Attention

The first step to understanding how GPT works is to understand how the *attention mechanism* works. This mechanism is what makes the Transformer architecture unique and distinct from recurrent approaches to language modeling. When we have developed a solid understanding of attention, we will then see how it is used within Transformer architectures such as GPT.

When you write, the choice that you make for the next word in the sentence is influenced by other words that you have already written. For example, suppose you start a sentence as follows:

```
The pink elephant tried to get into the car but it was too
```

Clearly, the next word should be something synonymous with *big*. How do we know this?

Certain other words in the sentence are important for helping us to make our decision. For example, the fact that it is an elephant, rather than a sloth, means that we prefer *big* rather than *slow*. If it were a swimming pool, rather than a car, we might choose *scared* as a possible alternative to *big*. Lastly, the action of *getting into* the car implies that size is the problem—if the elephant was trying to *squash* the car instead, we might choose *fast* as the final word, with *it* now referring to the car.

Other words in the sentence are not important at all. For example, the fact that the elephant is pink has no influence on our choice of final word. Equally, the minor words in the sentence (*the*, *but*, *it*, etc.) give the sentence grammatical form, but here aren't important to determine the required adjective.

In other words, we are *paying attention* to certain words in the sentence and largely ignoring others. Wouldn't it be great if our model could do the same thing?

An attention mechanism (also know as an *attention head*) in a Transformer is designed to do exactly this. It is able to decide where in the input it wants to pull information from, in order to efficiently extract useful information without being clouded by irrelevant details. This makes it highly adaptable to a range of circumstances, as it can decide where it wants to look for information at inference time.

In contrast, a recurrent layer tries to build up a generic hidden state that captures an overall representation of the input at each timestep. A weakness of this approach is that many of the words that have already been incorporated into the hidden vector will not be directly relevant to the immediate task at hand (e.g., predicting the next word), as we have just seen. Attention heads do not suffer from this problem, because they can pick and choose how to combine information from nearby words, depending on the context.

Queries, Keys, and Values

So how does an attention head decide where it wants to look for information? Before we get into the details, let's explore how it works at a high level, using our *pink elephant* example.

Imagine that we want to predict what follows the word *too*. To help with this task, other preceding words chime in with their opinions, but their contributions are weighted by how confident they are in their own expertise in predicting words that follow *too*. For example, the word *elephant* might confidently contribute that it is more likely to be a word related to size or loudness, whereas the word *was* doesn't have much to offer to narrow down the possibilities.

In other words, we can think of an attention head as a kind of information retrieval system, where a *query* ("What word follows *too*?") is made into a *key/value* store (other words in the sentence) and the resulting output is a sum of the values, weighted by the *resonance* between the query and each key.

We will now walk through the process in detail (Figure 9-2), again with reference to our *pink elephant* sentence.

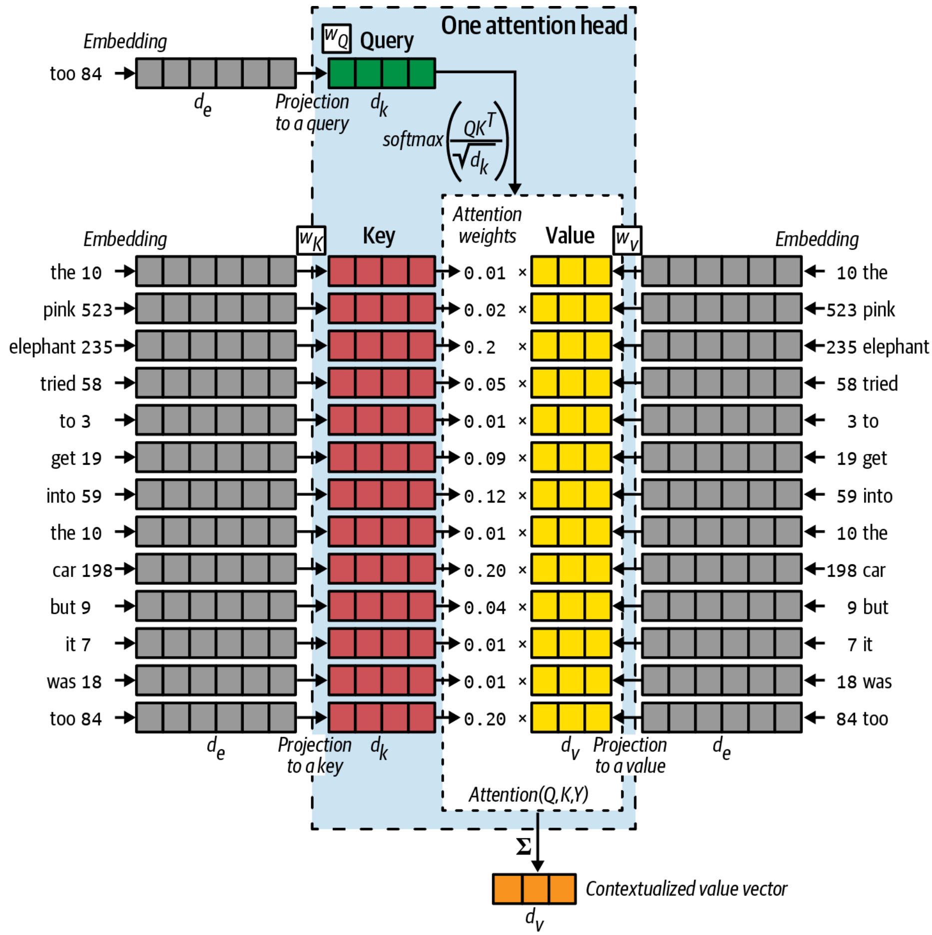

Figure 9-2. The mechanics of an attention head

The *query* (Q) can be thought of as a representation of the current task at hand (e.g., "What word follows *too*?"). In this example, it is derived from the embedding of the word *too*, by passing it through a weights matrix W_Q to change the dimensionality of the vector from d_e to d_k.

The *key* vectors (K) are representations of each word in the sentence—you can think of these as descriptions of the kinds of prediction tasks that each word can help with. They are derived in a similar fashion to the query, by passing each embedding through a weights matrix W_K to change the dimensionality of each vector from d_e to d_k. Notice that the keys and the query are the same length (d_k).

Inside the attention head, each key is compared to the query using a dot product between each pair of vectors (QK^T). This is why the keys and the query have to be the same length. The higher this number is for a particular key/query pair, the more the key resonates with the query, so it is allowed to make more of a contribution to the output of the attention head. The resulting vector is scaled by $\sqrt{d_k}$ to keep the variance of the vector sum stable (approximately equal to 1), and a softmax is applied to ensure the contributions sum to 1. This is a vector of *attention weights*.

The *value* vectors (V) are also representations of the words in the sentence—you can think of these as the unweighted contributions of each word. They are derived by passing each embedding through a weights matrix W_V to change the dimensionality of each vector from d_e to d_v. Notice that the value vectors do not necessarily have to have the same length as the keys and query (but often do, for simplicity).

The value vectors are multiplied by the attention weights to give the *attention* for a given Q, K, and V, as shown in Equation 9-1.

Equation 9-1. Attention equation

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

To obtain the final output vector from the attention head, the attention is summed to give a vector of length d_v. This *context vector* captures a blended opinion from words in the sentence on the task of predicting what word follows *too*.

Multihead Attention

There's no reason to stop at just one attention head! In Keras, we can build a `MultiHeadAttention` layer that concatenates the output from multiple attention heads, allowing each to learn a distinct attention mechanism so that the layer as a whole can learn more complex relationships.

The concatenated outputs are passed through one final weights matrix W_O to project the vector into the desired output dimension, which in our case is the same as the input dimension of the query (d_e), so that the layers can be stacked sequentially on top of each other.

Figure 9-3 shows how the output from a `MultiHeadAttention` layer is constructed. In Keras we can simply write the line shown in Example 9-2 to create such a layer.

Example 9-2. Creating a `MultiHeadAttention` layer in Keras

```
layers.MultiHeadAttention(
    num_heads = 4, ❶
    key_dim = 128, ❷
    value_dim = 64, ❸
    output_shape = 256 ❹
    )
```

❶ This multihead attention layer has four heads.

❷ The keys (and query) are vectors of length 128.

❸ The values (and therefore also the output from each head) are vectors of length 64.

❹ The output vector has length 256.

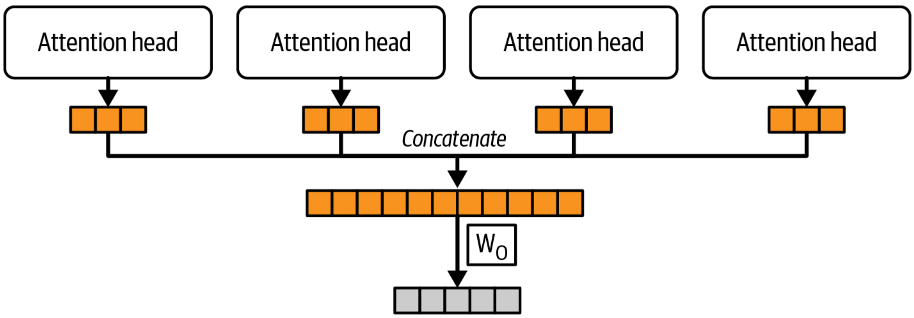

Figure 9-3. A multihead attention layer with four heads

Causal Masking

So far, we have assumed that the query input to our attention head is a single vector. However, for efficiency during training, we would ideally like the attention layer to be able to operate on every word in the input at once, predicting for each what the subsequent word will be. In other words, we want our GPT model to be able to handle a group of query vectors in parallel (i.e., a matrix).

You might think that we can just batch the vectors together into a matrix and let linear algebra handle the rest. This is true, but we need one extra step—we need to apply a mask to the query/key dot product, to avoid information from future words leaking through. This is known as *causal masking* and is shown in Figure 9-4.

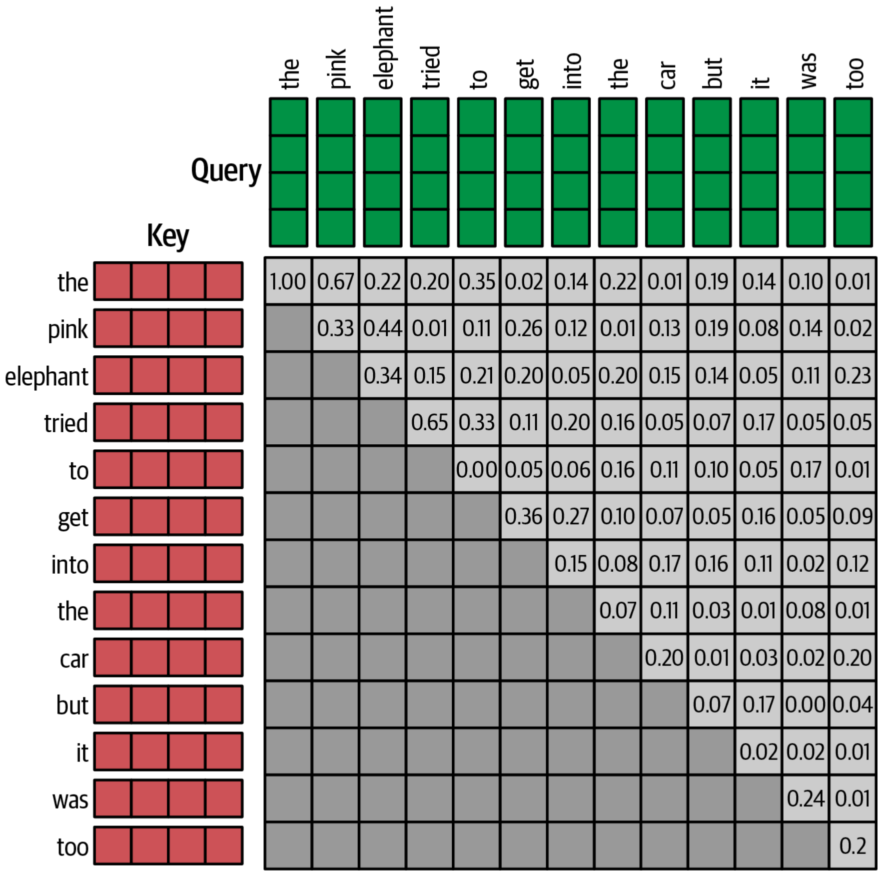

Figure 9-4. Matrix calculation of the attention scores for a batch of input queries, using a causal attention mask to hide keys that are not available to the query (because they come later in the sentence)

Without this mask, our GPT model would be able to perfectly guess the next word in the sentence, because it would be using the key from the word itself as a feature! The code for creating a causal mask is shown in Example 9-3, and the resulting `numpy` array (transposed to match the diagram) is shown in Figure 9-5.

Example 9-3. The causal mask function

```
def causal_attention_mask(batch_size, n_dest, n_src, dtype):
    i = tf.range(n_dest)[:, None]
    j = tf.range(n_src)
    m = i >= j - n_src + n_dest
    mask = tf.cast(m, dtype)
    mask = tf.reshape(mask, [1, n_dest, n_src])
```

```python
    mult = tf.concat(
        [tf.expand_dims(batch_size, -1), tf.constant([1, 1], dtype=tf.int32)], 0
    )
    return tf.tile(mask, mult)

np.transpose(causal_attention_mask(1, 10, 10, dtype = tf.int32)[0])
```

```
array([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [0, 1, 1, 1, 1, 1, 1, 1, 1, 1],
       [0, 0, 1, 1, 1, 1, 1, 1, 1, 1],
       [0, 0, 0, 1, 1, 1, 1, 1, 1, 1],
       [0, 0, 0, 0, 1, 1, 1, 1, 1, 1],
       [0, 0, 0, 0, 0, 1, 1, 1, 1, 1],
       [0, 0, 0, 0, 0, 0, 1, 1, 1, 1],
       [0, 0, 0, 0, 0, 0, 0, 1, 1, 1],
       [0, 0, 0, 0, 0, 0, 0, 0, 1, 1],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 1]], dtype=int32)
```

Figure 9-5. The causal mask as a `numpy` array—1 means unmasked and 0 means masked

> Causal masking is only required in *decoder Transformers* such as GPT, where the task is to sequentially generate tokens given previous tokens. Masking out future tokens during training is therefore essential.
>
> Other flavors of Transformer (e.g., *encoder Transformers*) do not need causal masking, because they are not trained to predict the next token. For example Google's BERT predicts masked words within a given sentence, so it can use context from both before and after the word in question.[3]
>
> We will explore the different types of Transformers in more detail at the end of the chapter.

This concludes our explanation of the multihead attention mechanism that is present in all Transformers. It is remarkable that the learnable parameters of such an influential layer consist of nothing more than three densely connected weights matrices for each attention head (W_Q, W_K, W_V) and one further weights matrix to reshape the output (W_O). There are no convolutions or recurrent mechanisms at all in a multihead attention layer!

Next, we shall take a step back and see how the multihead attention layer forms just one part of a larger component known as a *Transformer block*.

The Transformer Block

A *Transformer block* is a single component within a Transformer that applies some skip connections, feed-forward (dense) layers, and normalization around the multi-head attention layer. A diagram of a Transformer block is shown in Figure 9-6.

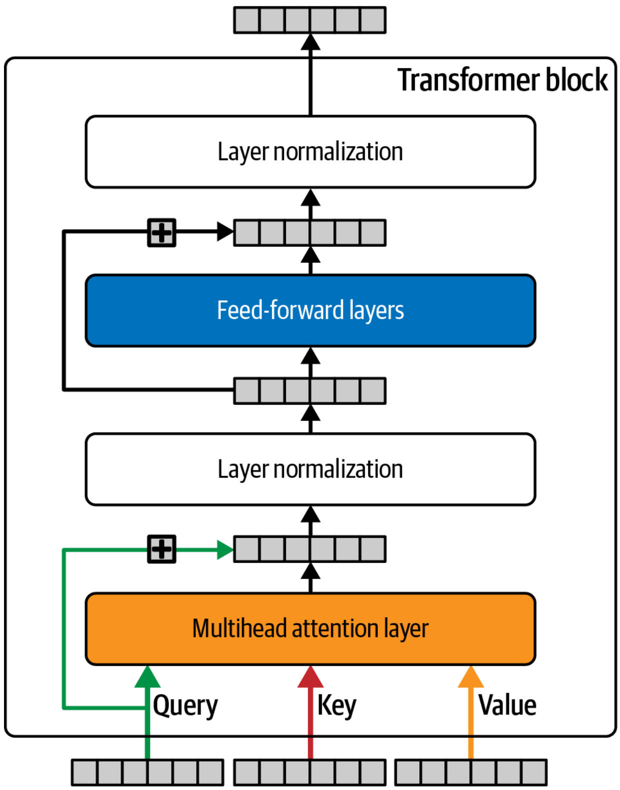

Figure 9-6. A Transformer block

Firstly, notice how the query is passed around the multihead attention layer to be added to the output—this is a skip connection and is common in modern deep learning architectures. It means we can build very deep neural networks that do not suffer as much from the vanishing gradient problem, because the skip connection provides a gradient-free *highway* that allows the network to transfer information forward uninterrupted.

Secondly, *layer normalization* is used in the Transformer block to provide stability to the training process. We have already seen the batch normalization layer in action throughout this book, where the output from each channel is normalized to have a mean of 0 and standard deviation of 1. The normalization statistics are calculated across the batch and spatial dimensions.

In contrast, layer normalization in a Transformer block normalizes each position of each sequence in the batch by calculating the normalizing statistics across the channels. It is the complete opposite of batch normalization, in terms of how the normalization statistics are calculated. A diagram showing the difference between batch normalization and layer normalization is shown in Figure 9-7.

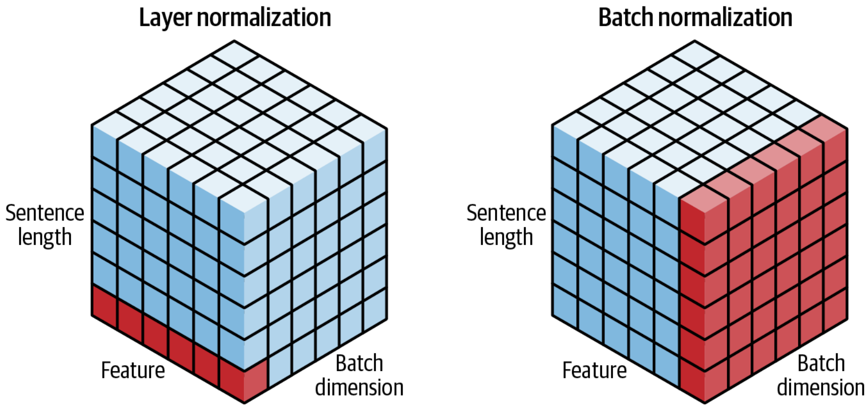

Figure 9-7. Layer normalization versus batch normalization—the normalization statistics are calculated across the blue cells (source: Sheng et al., 2020)[4]

Layer Normalization Versus Batch Normalization

Layer normalization was used in the original GPT paper and is commonly used for text-based tasks to avoid creating normalization dependencies across sequences in the batch. However, recent work such as Shen et al.*s* challenges this assumption, showing that with some tweaks a form of batch normalization can still be used within Transformers, outperforming more traditional layer normalization.

Lastly, a set of feed-forward (i.e., densely connected) layers is included in the Transformer block, to allow the component to extract higher-level features as we go deeper into the network.

A Keras implementation of a Transformer block is shown in Example 9-4.

Example 9-4. A `TransformerBlock` layer in Keras

```
class TransformerBlock(layers.Layer):
    def __init__(self, num_heads, key_dim, embed_dim, ff_dim, dropout_rate=0.1): ❶
        super(TransformerBlock, self).__init__()
        self.num_heads = num_heads
        self.key_dim = key_dim
        self.embed_dim = embed_dim
        self.ff_dim = ff_dim
        self.dropout_rate = dropout_rate
        self.attn = layers.MultiHeadAttention(
            num_heads, key_dim, output_shape = embed_dim
        )
        self.dropout_1 = layers.Dropout(self.dropout_rate)
        self.ln_1 = layers.LayerNormalization(epsilon=1e-6)
        self.ffn_1 = layers.Dense(self.ff_dim, activation="relu")
        self.ffn_2 = layers.Dense(self.embed_dim)
        self.dropout_2 = layers.Dropout(self.dropout_rate)
        self.ln_2 = layers.LayerNormalization(epsilon=1e-6)

    def call(self, inputs):
        input_shape = tf.shape(inputs)
        batch_size = input_shape[0]
        seq_len = input_shape[1]
        causal_mask = causal_attention_mask(
            batch_size, seq_len, seq_len, tf.bool
        ) ❷
        attention_output, attention_scores = self.attn(
            inputs,
            inputs,
            attention_mask=causal_mask,
            return_attention_scores=True
        ) ❸
        attention_output = self.dropout_1(attention_output)
        out1 = self.ln_1(inputs + attention_output) ❹
        ffn_1 = self.ffn_1(out1) ❺
        ffn_2 = self.ffn_2(ffn_1)
        ffn_output = self.dropout_2(ffn_2)
        return (self.ln_2(out1 + ffn_output), attention_scores) ❻
```

❶ The sublayers that make up the `TransformerBlock` layer are defined within the initialization function.

❷ The causal mask is created to hide future keys from the query.

❸ The multihead attention layer is created, with the attention masks specified.

❹ The first *add and normalization* layer.

❺ The feed-forward layers.

❻ The second *add and normalization* layer.

Positional Encoding

There is one final step to cover before we can put everything together to train our GPT model. You may have noticed that in the multihead attention layer, there is nothing that cares about the ordering of the keys. The dot product between each key and the query is calculated in parallel, not sequentially, like in a recurrent neural network. This is a strength (because of the parallelization efficiency gains) but also a problem, because we clearly need the attention layer to be able to predict different outputs for the following two sentences:

- The dog looked at the boy and … (barked?)
- The boy looked at the dog and … (smiled?)

To solve this problem, we use a technique called *positional encoding* when creating the inputs to the initial Transformer block. Instead of only encoding each token using a *token embedding*, we also encode the position of the token, using a *position embedding*.

The *token embedding* is created using a standard `Embedding` layer to convert each token into a learned vector. We can create the *positional embedding* in the same way, using a standard `Embedding` layer to convert each integer position into a learned vector.

> While GPT uses an `Embedding` layer to embed the position, the original Transformer paper used trigonometric functions—we'll cover this alternative in Chapter 11, when we explore music generation.

To construct the joint token–position encoding, the token embedding is added to the positional embedding, as shown in Figure 9-8. This way, the meaning and position of each word in the sequence are captured in a single vector.

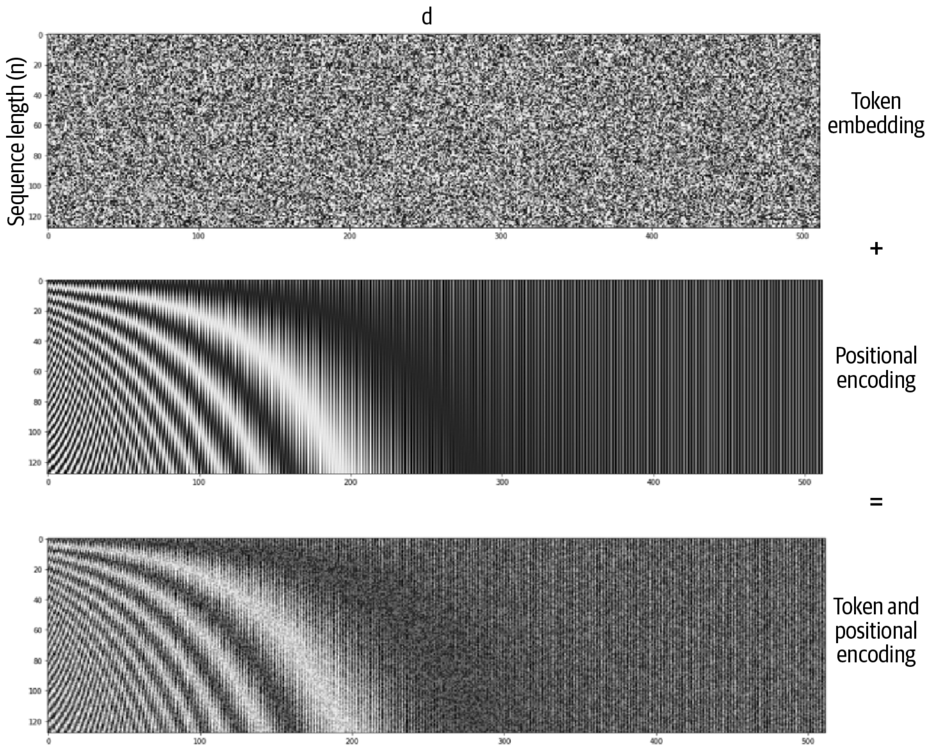

Figure 9-8. The token embeddings are added to the positional embeddings to give the token position encoding

The code that defines our `TokenAndPositionEmbedding` layer is shown in Example 9-5.

Example 9-5. The `TokenAndPositionEmbedding` layer

```
class TokenAndPositionEmbedding(layers.Layer):
    def __init__(self, maxlen, vocab_size, embed_dim):
        super(TokenAndPositionEmbedding, self).__init__()
        self.maxlen = maxlen
        self.vocab_size =vocab_size
        self.embed_dim = embed_dim
        self.token_emb = layers.Embedding(
            input_dim=vocab_size, output_dim=embed_dim
        ) ❶
        self.pos_emb = layers.Embedding(input_dim=maxlen, output_dim=embed_dim) ❷

    def call(self, x):
        maxlen = tf.shape(x)[-1]
        positions = tf.range(start=0, limit=maxlen, delta=1)
        positions = self.pos_emb(positions)
        x = self.token_emb(x)
        return x + positions ❸
```

❶ The tokens are embedded using an `Embedding` layer.

❷ The positions of the tokens are also embedded using an `Embedding` layer.

❸ The output from the layer is the sum of the token and position embeddings.

Training GPT

Now we are ready to build and train our GPT model! To put everything together, we need to pass our input text through the token and position embedding layer, then through our Transformer block. The final output of the network is a simple `Dense` layer with softmax activation over the number of words in the vocabulary.

> For simplicity, we will use just one Transformer block, rather than the 12 in the paper.

The overall architecture is shown in Figure 9-9 and the equivalent code is provided in Example 9-6.

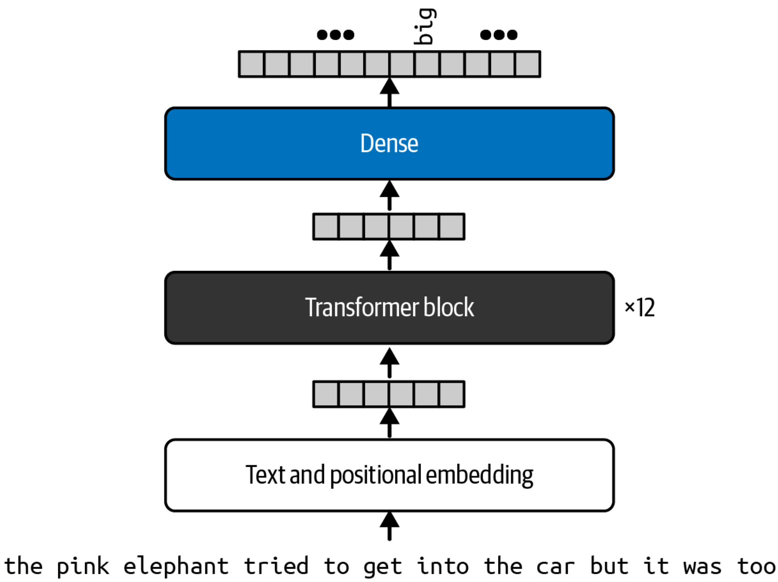

Figure 9-9. The simplified GPT model architecture

Example 9-6. A GPT model in Keras

```
MAX_LEN = 80
VOCAB_SIZE = 10000
EMBEDDING_DIM = 256
N_HEADS = 2
KEY_DIM = 256
FEED_FORWARD_DIM = 256

inputs = layers.Input(shape=(None,), dtype=tf.int32) ❶
x = TokenAndPositionEmbedding(MAX_LEN, VOCAB_SIZE, EMBEDDING_DIM)(inputs) ❷
x, attention_scores = TransformerBlock(
    N_HEADS, KEY_DIM, EMBEDDING_DIM, FEED_FORWARD_DIM
)(x) ❸
outputs = layers.Dense(VOCAB_SIZE, activation = 'softmax')(x) ❹
gpt = models.Model(inputs=inputs, outputs=[outputs, attention]) ❺
gpt.compile("adam", loss=[losses.SparseCategoricalCrossentropy(), None]) ❻
gpt.fit(train_ds, epochs=5)
```

❶ The input is padded (with zeros).

❷ The text is encoded using a `TokenAndPositionEmbedding` layer.

❸ The encoding is passed through a `TransformerBlock`.

❹ The transformed output is passed through a `Dense` layer with softmax activation to predict a distribution over the subsequent word.

❺ The `Model` takes a sequence of word tokens as input and outputs the predicted subsequent word distribution. The output from the Transformer block is also returned so that we can inspect how the model is directing its attention.

❻ The model is compiled with `SparseCategoricalCrossentropy` loss over the predicted word distribution.

Analysis of GPT

Now that we have compiled and trained our GPT model, we can start to use it to generate long strings of text. We can also interrogate the attention weights that are output from the `TransformerBlock`, to understand where the Transformer is looking for information at different points in the generation process.

Generating text

We can generate new text by applying the following process:

1. Feed the network with an existing sequence of words and ask it to predict the following word.
2. Append this word to the existing sequence and repeat.

The network will output a set of probabilities for each word that we can sample from, so we can make the text generation stochastic, rather than deterministic.

We will use the same `TextGenerator` class introduced in Chapter 5 for LSTM text generation, including the `temperature` parameter that specifies how deterministic we would like the sampling process to be. Let's take a look at this in action, at two different temperature values (Figure 9-10).

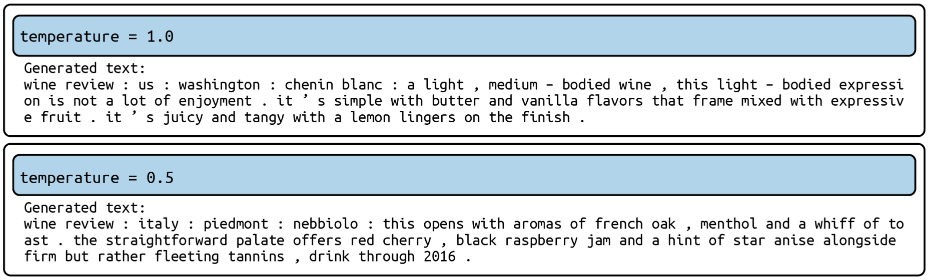

Figure 9-10. Generated outputs at `temperature = 1.0` and `temperature = 0.5`.

There are a few things to note about these two passages. First, both are stylistically similar to a wine review from the original training set. They both open with the region and type of wine, and the wine type stays consistent throughout the passage (for example, it doesn’t switch color halfway through). As we saw in Chapter 5, the generated text with temperature 1.0 is more adventurous and therefore less accurate than the example with temperature 0.5. Generating multiple samples with temperature 1.0 will therefore lead to more variety as the model is sampling from a probability distribution with greater variance.

Viewing the attention scores

We can also ask the model to tell us how much attention is being placed on each word, when deciding on the next word in the sentence. The `TransformerBlock` outputs the attention weights for each head, which are a softmax distribution over the preceding words in the sentence.

To demonstrate this, Figure 9-11 shows the top five tokens with the highest probabilities for three different input prompts, as well as the average attention across both heads, against each preceding word. The preceding words are colored according to their attention score, averaged across the two attention heads. Darker blue indicates more attention is being placed on the word.

```
wine review : germany :
 pfalz:          51.53%
 mosel:          41.21%
 rheingau:       4.27%
 rheinhessen:    2.16%
 franken:        0.44%
 --------

wine review : germany : rheingau : riesling : this is a ripe , full - bodied
 riesling:       46.56%
 ,:      27.78%
 wine:           16.88%
 and:    4.58%
 yet:    1.33%
 --------

wine review : germany : rheingau : riesling : this is a ripe , full - bodied riesling
with a touch of residual sugar . it ' s a slightly
 sweet:          94.23%
 oily:           1.25%
 viscous:        1.09%
 bitter:         0.88%
 honeyed:        0.66%
 --------
```

Figure 9-11. Distribution of word probabilities following various sequences

In the first example, the model attends closely to the country (*germany*) in order to decide on the word that relates to the region. This makes sense! To pick a region, it needs to take lots of information from the words that relate to the country, to ensure they match. It doesn't need to pay as much attention to the first two tokens (*wine review*) because they don't hold any useful information regarding the region.

In the second example, it needs to refer back to the grape (*riesling*), so it pays attention to the first time that it was mentioned. It can pull this information by directly attending to the word, no matter how far back it is in the sentence (within the upper limit of 80 words). Notice that this is very different from a recurrent neural network, which relies on a hidden state to maintain all interesting information over the length of the sequence so that it can be drawn upon if required—a much less efficient approach.

The final sequence shows an example of how our GPT model can choose an appropriate adjective based on a combination of information. Here the attention is again on the grape (*riesling*), but also on the fact that it contains *residual sugar*. As Riesling is typically a sweet wine, and sugar is already mentioned, it makes sense that it should be described as *slightly sweet* rather than *slightly earthy*, for example.

It is incredibly informative to be able to interrogate the network in this way, to understand exactly where it is pulling information from in order to make accurate decisions about each subsequent word. I highly recommend playing around with the input prompts to see if you can get the model to attend to words really far back in the sentence, to convince yourself of the power of attention-based models over more traditional recurrent models!

Other Transformers

Our GPT model is a *decoder Transformer*—it generates a text string one token at a time and uses causal masking to only attend to previous words in the input string. There are also *encoder Transformers*, which do not use causal masking—instead, they attend to the entire input string in order to extract a meaningful contextual representation of the input. For other tasks, such as language translation, there are also *encoder-decoder Transformers* that can translate from one text string to another; this type of model contains both encoder Transformer blocks and decoder Transformer blocks.

Table 9-1 summarizes the three types of Transformers, with the best examples of each architecture and typical use cases.

Table 9-1. The three Transformer architectures

Type	Examples	Use cases
Encoder	BERT (Google)	Sentence classification, named entity recognition, extractive question answering
Encoder-decoder	T5 (Google)	Summarization, translation, question answering
Decoder	GPT-3 (OpenAI)	Text generation

A well-known example of an encoder Transformer is the *Bidirectional Encoder Representations from Transformers* (BERT) model, developed by Google (Devlin et al., 2018) that predicts missing words from a sentence, given context from both before and after the missing word in all layers.

Encoder Transformers

Encoder Transformers are typically used for tasks that require an understanding of the input as a whole, such as sentence classification, named entity recognition, and extractive question answering. They are not used for text generation tasks, so we will not explore them in detail in this book—see Lewis Tunstall et al.'s *Natural Language Processing with Transformers* (O'Reilly) for more information.

In the following sections we will explore how encoder-decoder transformers work and discuss extensions of the original GPT model architecture released by OpenAI, including ChatGPT, which has been specifically designed for conversational applications.

T5

An example of a modern Transformer that uses the encoder-decoder structure is the T5 model from Google.[5] This model reframes a range of tasks into a text-to-text framework, including translation, linguistic acceptability, sentence similarity, and document summarization, as shown in Figure 9-12.

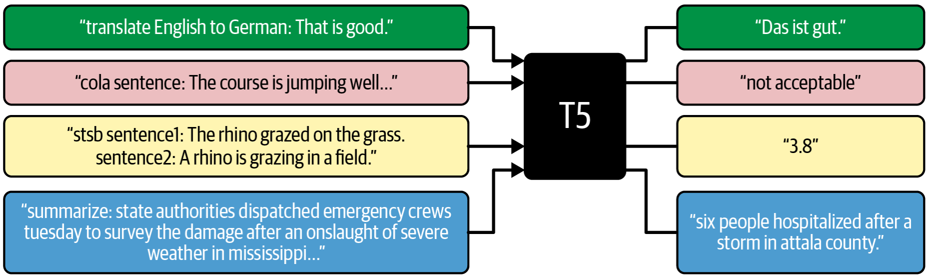

Figure 9-12. Examples of how T5 reframes a range of tasks into a text-to-text framework, including translation, linguistic acceptability, sentence similarity, and document summarization (source: Raffel et al., 2019)

The T5 model architecture closely matches the encoder-decoder architecture used in the original Transformer paper, shown in Figure 9-13. The key difference is that T5 is trained on an enormous 750 GB corpus of text (the Colossal Clean Crawled Corpus, or C4), whereas the original Transformer paper was focused only on language translation, so it was trained on 1.4 GB of English–German sentence pairs.

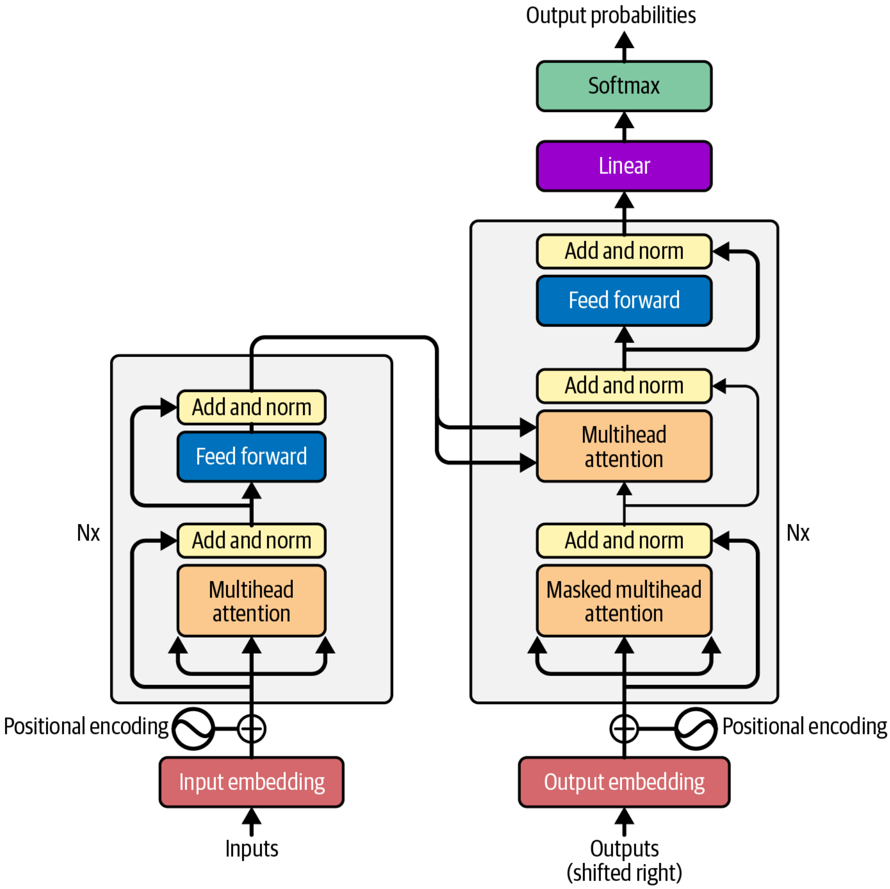

Figure 9-13. An encoder-decoder Transformer model: each gray box is a Transformer block (source: Vaswani et al., 2017)

Much of this diagram is already familiar to us—we can see the Transformer blocks being repeated and positional embedding being used to capture the ordering of the input sequences. The two key differences between this model and the GPT model that we built earlier in the chapter are as follows:

- On the lefthand side, a set of *encoder* Transformer blocks encode the sequence to be translated. Notice that there is no causal masking on the attention layer. This is because we are not generating further text to extend the sequence to be translated; we just want to learn a good representation of the sequence as a whole that can be fed to the decoder. Therefore, the attention layers in the encoder can be completely unmasked to capture all the cross-dependencies between words, no matter the order.

- On the righthand side, a set of *decoder* Transformer blocks generate the translated text. The initial attention layer is *self-referential* (i.e., the key, value, and query come from the same input) and causal masking is used to ensure information from future tokens is not leaked to the current word to be predicted. However, we can then see that the subsequent attention layer pulls the key and value from the encoder, leaving only the query passed through from the decoder itself. This is called *cross-referential* attention and means that the decoder can attend to the encoder representation of the input sequence to be translated. This is how the decoder knows what meaning the translation needs to convey!

Figure 9-14 shows an example of cross-referential attention. Two attention heads of the decoder layer are able to work together to provide the correct German translation for the word *the*, when used in the context of *the street*. In German, there are three definite articles (*der, die, das*) depending on the gender of the noun, but the Transformer knows to choose *die* because one attention head is able to attend to the word *street* (a feminine word in German), while another attends to the word to translate (*the*).

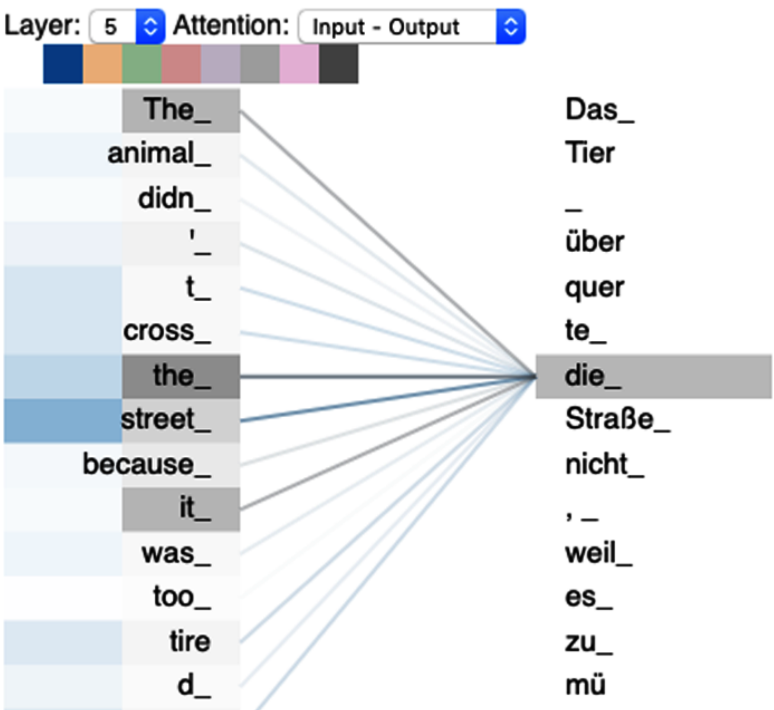

Figure 9-14. An example of how one attention head attends to the word "the" and another attends to the word "street" in order to correctly translate the word "the" to the German word "die" as the feminine definite article of "Straße"

This example is from the Tensor2Tensor GitHub repository, which contains a Colab notebook that allows you to play around with a trained encoder-decoder Transformer model and see how the attention mechanisms of the encoder and decoder impact the translation of a given sentence into German.

GPT-3 and GPT-4

Since the original 2018 publication of GPT, OpenAI has released multiple updated versions that improve upon the original model, as shown in Table 9-2.

Table 9-2. The evolution of OpenAI's GPT collection of models

Model	Date	Layers	Attention heads	Word embedding size	Context window	# parameters	Training data
GPT	Jun 2018	12	12	768	512	120,000,000	BookCorpus: 4.5 GB of text from unpublished books
GPT-2	Feb 2019	48	48	1,600	1,024	1,500,000,000	WebText: 40 GB of text from outbound Reddit links
GPT-3	May 2020	96	96	12,888	2,048	175,000,000,000	CommonCrawl, WebText, English Wikipedia, book corpora and others: 570 GB
GPT-4	Mar 2023	-	-	-	-	-	-

The model architecture of GPT-3 is fairly similar to the original GPT model, except it is much larger and trained on much more data. At the time of writing, GPT-4 is in limited beta—OpenAI has not publicly released details of the model's structure and size, though we do know that it is able to accept images as input, so crosses over into being a multimodal model for the first time. The model weights of GPT-3 and GPT-4 are not open source, though the models are available through a commercial tool and API.

GPT-3 can also be fine-tuned to your own training data—this allows you to provide multiple examples of how it should react to a given style of prompt by physically updating the weights of the network. In many cases this may not be necessary, as GPT-3 can be told how to react to a given style of prompt simply by providing a few examples in the prompt itself (this is known as *few-shot learning*). The benefit of fine-tuning is that you do not need to provide these examples as part of every single input prompt, saving costs in the long run.

An example of the output from GPT-3, given a system prompt sentence, is shown in Figure 9-15.

Write me a short story about a unicorn who defeats an evil dragon.

Luna was out for a leisurely stroll through the forest when she heard a loud, ferocious roar. She immediately knew it was a dragon and ran as fast as she could in the opposite direction.

As she ran, she could feel the dragon's hot breath on her back and knew she wouldn't be able to outrun it for long. She had to think of a plan, and fast.

Just as the dragon was about to snatch her up in its teeth, Luna turned around and stabbed it in the eye with her horn. The dragon screeched in pain and flew into a rage, but Luna was ready for it.

She dodged its fire breath and claws until she finally had an opportunity to strike. She impaled the dragon with her horn and it fell to the ground with a thud, dead.

Luna breathed a sigh of relief and trotted away, knowing that she had saved the forest from the evil dragon.

Figure 9-15. An example of how GPT-3 can extend a given system prompt

Language models such as GPT benefit hugely from scaling—both in terms of number of model weights and dataset size. The ceiling of large language model capability has yet to be reached, with researchers continuing to push the boundaries of what is possible with increasingly larger models and datasets.

ChatGPT

A few months before the beta release of GPT-4, OpenAI announced *ChatGPT*—a tool that allows users to interact with their suite of large language models through a conversational interface. The original release in November 2022 was powered by *GPT-3.5*, a version of the model that was more powerful that GPT-3 and was fine-tuned to conversational responses.

Example dialogue is shown in Figure 9-16. Notice how the agent is able to maintain state between inputs, understanding that the *attention* mentioned in the second question refers to attention in the context of Transformers, rather than a person's ability to focus.

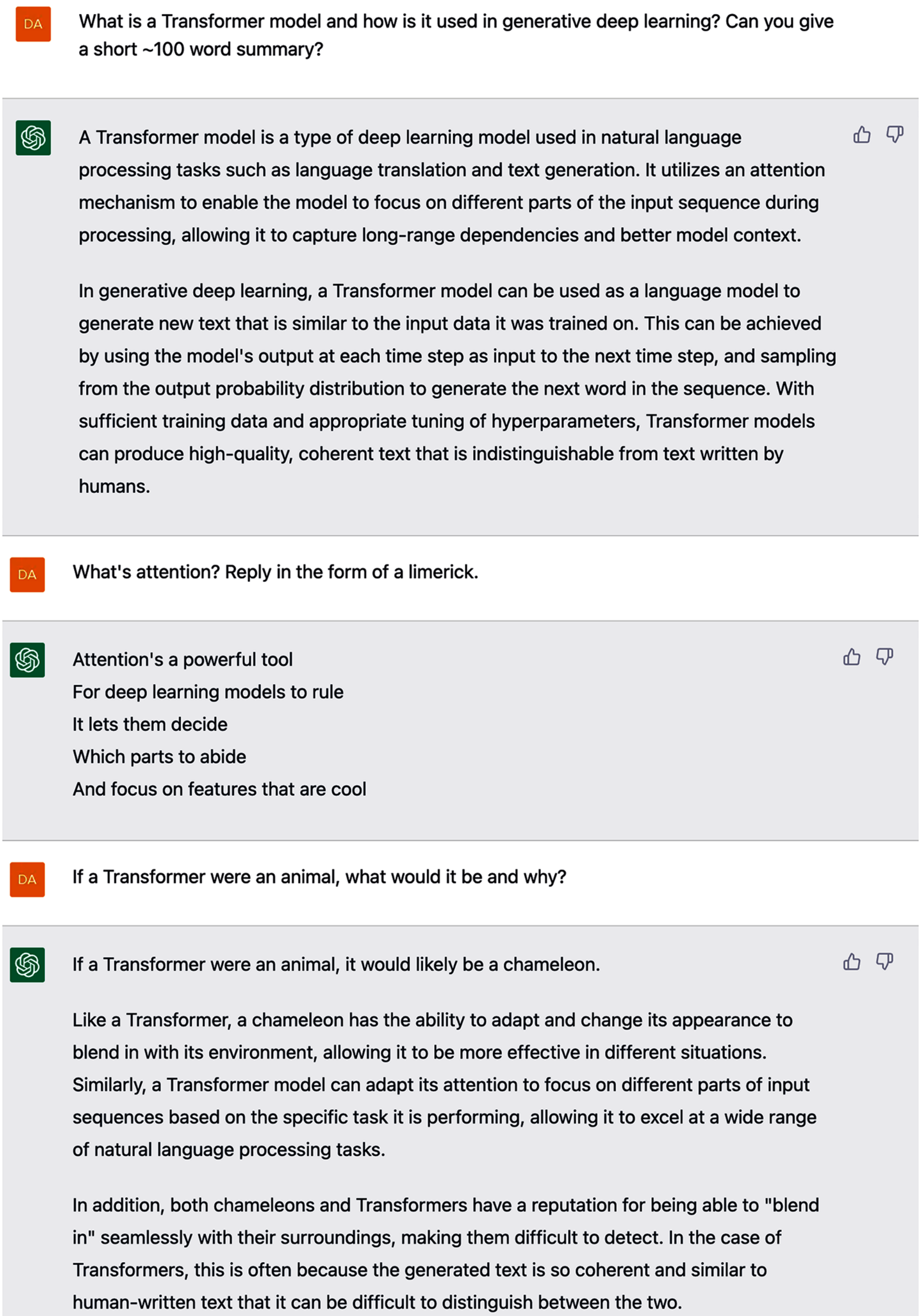

Figure 9-16. An example of ChatGPT answering questions about Transformers

At the time of writing, there is no official paper that describes how ChatGPT works in detail, but from the official blog post we know that it uses a technique called *reinforcement learning from human feedback* (RLHF) to fine-tune the GPT-3.5 model. This technique was also used in the ChatGPT group's earlier paper[6] that introduced the *InstructGPT* model, a fine-tuned GPT-3 model that is specifically designed to more accurately follow written instructions.

The training process for ChatGPT is as follows:

1. *Supervised fine-tuning*: Collect a demonstration dataset of conversational inputs (prompts) and desired outputs that have been written by humans. This is used to fine-tune the underlying language model (GPT-3.5) using supervised learning.
2. *Reward modeling*: Present a human labeler with examples of prompts and several sampled model outputs and ask them to rank the outputs from best to worst. Train a reward model that predicts the score given to each output, given the conversation history.
3. *Reinforcement learning*: Treat the conversation as a reinforcement learning environment where the *policy* is the underlying language model, initialized to the fine-tuned model from step 1. Given the current *state* (the conversation history) the policy outputs an *action* (a sequence of tokens), which is scored by the reward model trained in step 2. A reinforcement learning algorithm—proximal policy optimization (PPO)—can then be trained to maximize the reward, by adjusting the weights of the language model.

Reinforcement Learning

For an introduction to reinforcement learning see Chapter 12, where we explore how generative models can be used in a reinforcement learning setting.

The RLHF process is shown in Figure 9-17.

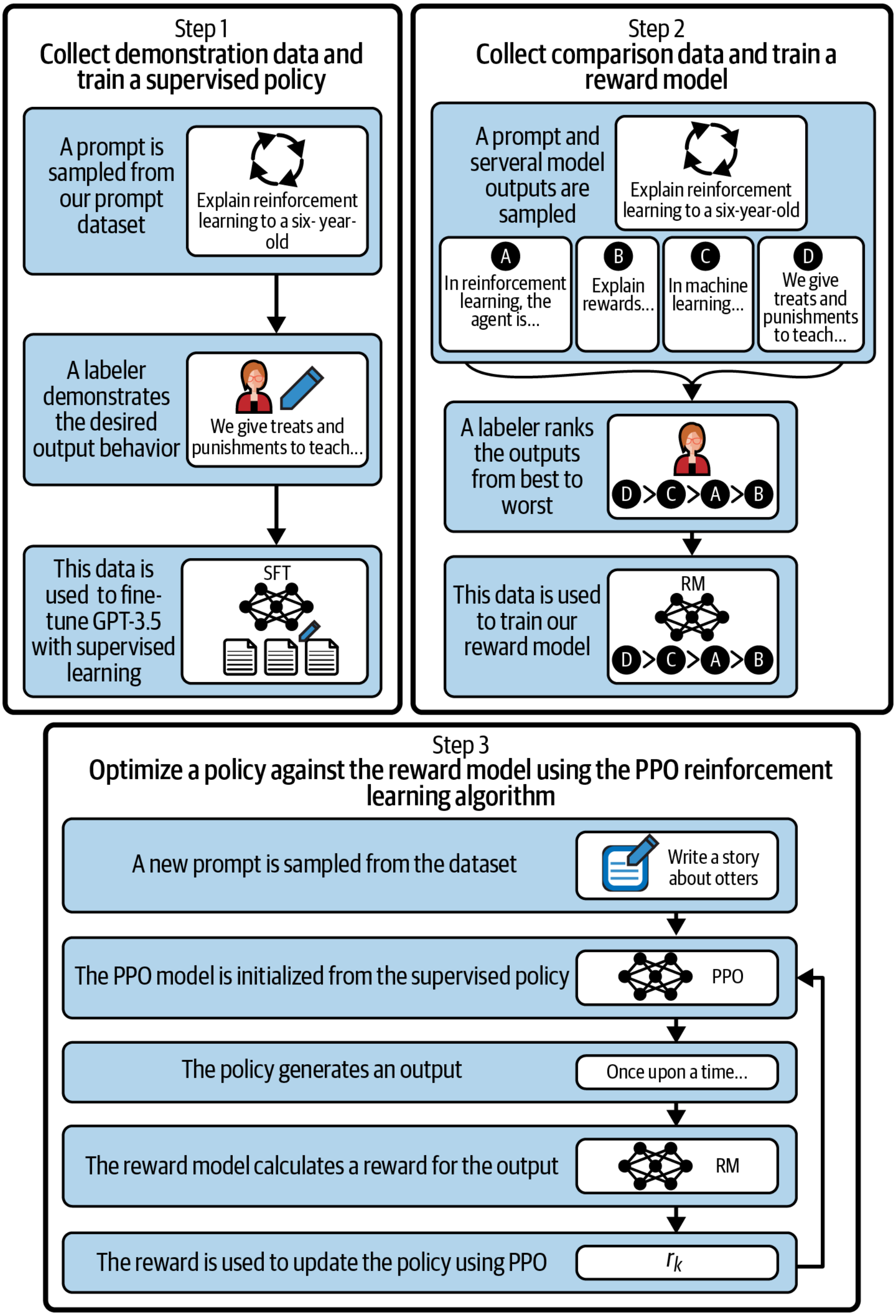

Figure 9-17. The reinforcement learning from human feedback fine-tuning process used in ChatGPT (source: OpenAI)

While ChatGPT still has many limitations (such as sometimes "hallucinating" factually incorrect information), it is a powerful example of how Transformers can be used to build generative models that can produce complex, long-ranging, and novel output that is often indistinguishable from human-generated text. The progress made thus far by models like ChatGPT serves as a testament to the potential of AI and its transformative impact on the world.

Moreover, it is evident that AI-driven communication and interaction will continue to rapidly evolve in the future. Projects like *Visual ChatGPT*[7] are now combining the linguistic power of ChatGPT with visual foundation models such as Stable Diffusion, enabling users to interact with ChatGPT not only through text, but also images. The fusion of linguistic and visual capabilities in projects like Visual ChatGPT and GPT-4 have the potential to herald a new era in human–computer interaction.

Summary

In this chapter, we explored the Transformer model architecture and built a version of GPT—a model for state-of-the-art text generation.

GPT makes use of a mechanism known as attention, which removes the need for recurrent layers (e.g., LSTMs). It works like an information retrieval system, utilizing queries, keys, and values to decide how much information it wants to extract from each input token.

Attention heads can be grouped together to form what is known as a multihead attention layer. These are then wrapped up inside a Transformer block, which includes layer normalization and skip connections around the attention layer. Transformer blocks can be stacked to create very deep neural networks.

Causal masking is used to ensure that GPT cannot leak information from downstream tokens into the current prediction. Also, a technique known as positional encoding is used to ensure that the ordering of the input sequence is not lost, but instead is baked into the input alongside the traditional word embedding.

When analyzing the output from GPT, we saw it was possible not only to generate new text passages, but also to interrogate the attention layer of the network to understand where in the sentence it is looking to gather information to improve its prediction. GPT can access information at a distance without loss of signal, because the attention scores are calculated in parallel and do not rely on a hidden state that is carried through the network sequentially, as is the case with recurrent neural networks.

We saw how there are three families of Transformers (encoder, decoder, and encoder-decoder) and the different tasks that can be accomplished with each. Finally, we explored the structure and training process of other large language models such as Google's T5 and OpenAI's ChatGPT.

References

1. Ashish Vaswani et al., "Attention Is All You Need," June 12, 2017, *https://arxiv.org/abs/1706.03762*.

2. Alec Radford et al., "Improving Language Understanding by Generative Pre-Training," June 11, 2018, *https://openai.com/research/language-unsupervised*.

3. Jacob Devlin et al., "BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding," October 11, 2018, *https://arxiv.org/abs/1810.04805*.

4. Sheng Shen et al., "PowerNorm: Rethinking Batch Normalization in Transformers," June 28, 2020, *https://arxiv.org/abs/2003.07845*.

5. Colin Raffel et al., "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer," October 23, 2019, *https://arxiv.org/abs/1910.10683*.

6. Long Ouyang et al., "Training Language Models to Follow Instructions with Human Feedback," March 4, 2022, *https://arxiv.org/abs/2203.02155*.

7. Chenfei Wu et al., "Visual ChatGPT: Talking, Drawing and Editing with Visual Foundation Models," March 8, 2023, *https://arxiv.org/abs/2303.04671*.