Build a Large Language Model (From Scratch)

Table of Contents

1. Understanding LLM

Sections:

```
1.1 What is an LLM?
1.2 Applications of LLMs
1.3 Stages of building and using LLMs
1.4 Using LLMs for different tasks
1.5 Utilizing large datasets
1.6 A closer look at the GPT architecture
```

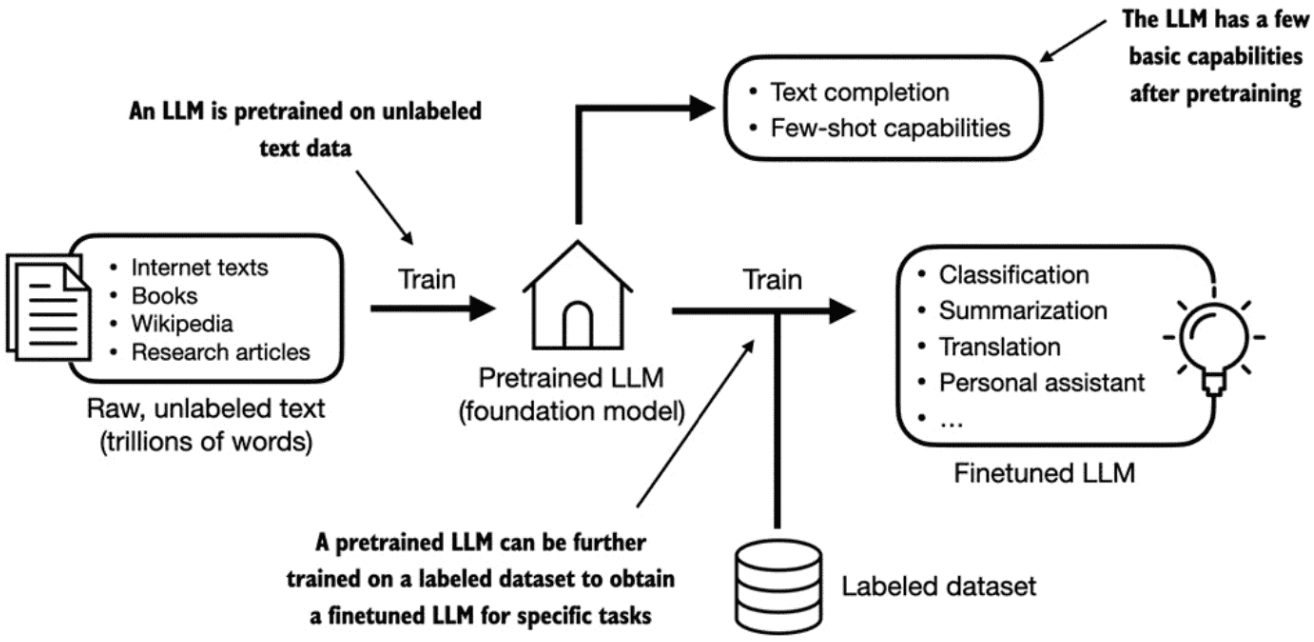
```
1.7 Building a large language model
1.8 Summary
```

AI including:

```
machine learning
deep learning

rule-based systems
genetic algorithms
expert systems
fuzzy logic
symbolic reasoning
```

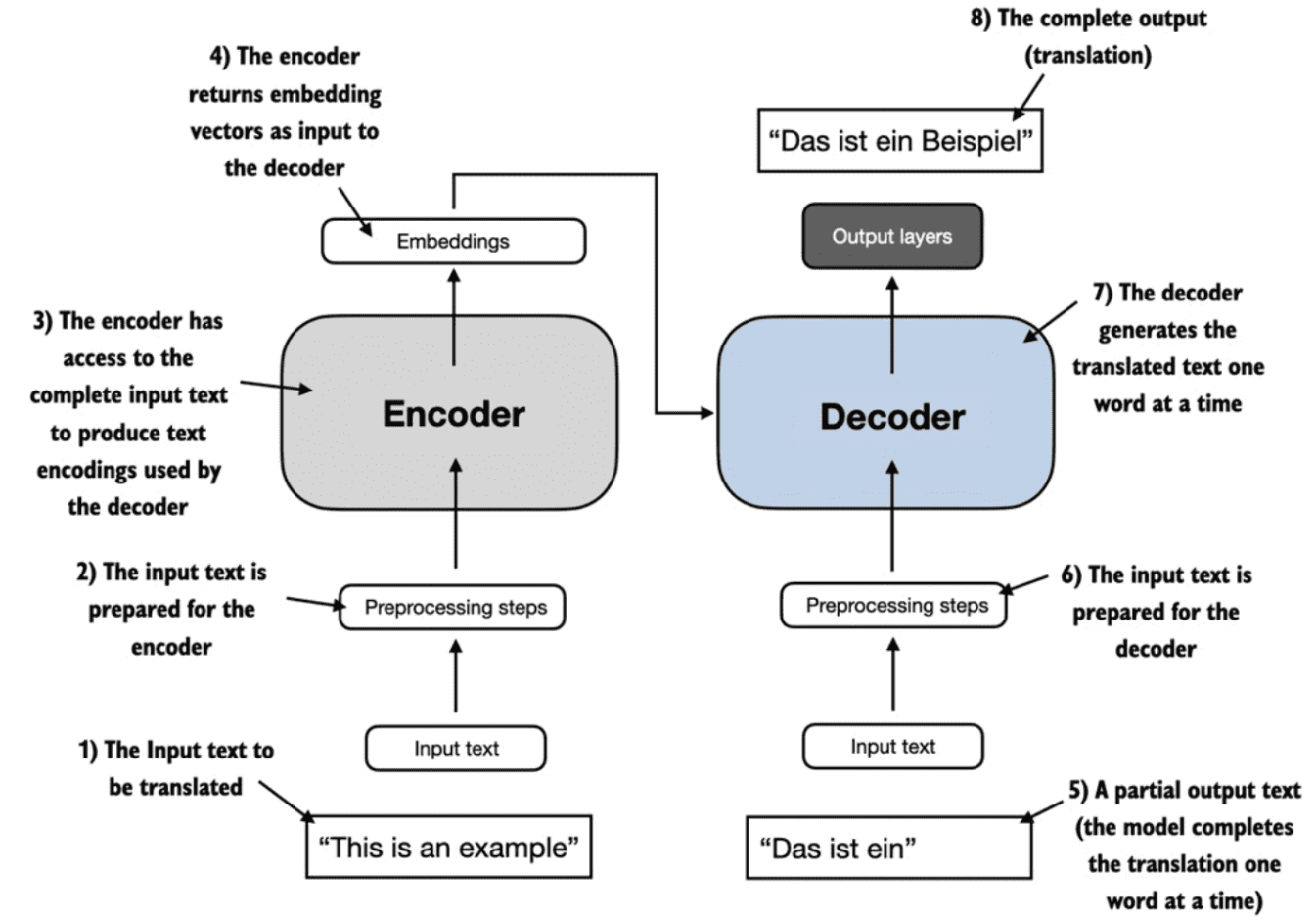
- Traditional machine learning: Human experts might manually extract features from email text such as the frequency of certain trigger words (“prize,” “win,” “free”), the number of exclamation marks, use of all uppercase words, or the presence of suspicious links.
- Deep learning: Does not require manual feature extraction. This means that human experts do not need to identify and select the most relevant features for a deep learning model.

> **Note:** Both traditional machine learning and deep learning require collecting data (e.g., which emails are spam, which are not). However, traditional machine learning also requires experts to manually extract features. Features for spam emails include: many words like "win", "free", suspicious links, usage of uppercase letters...

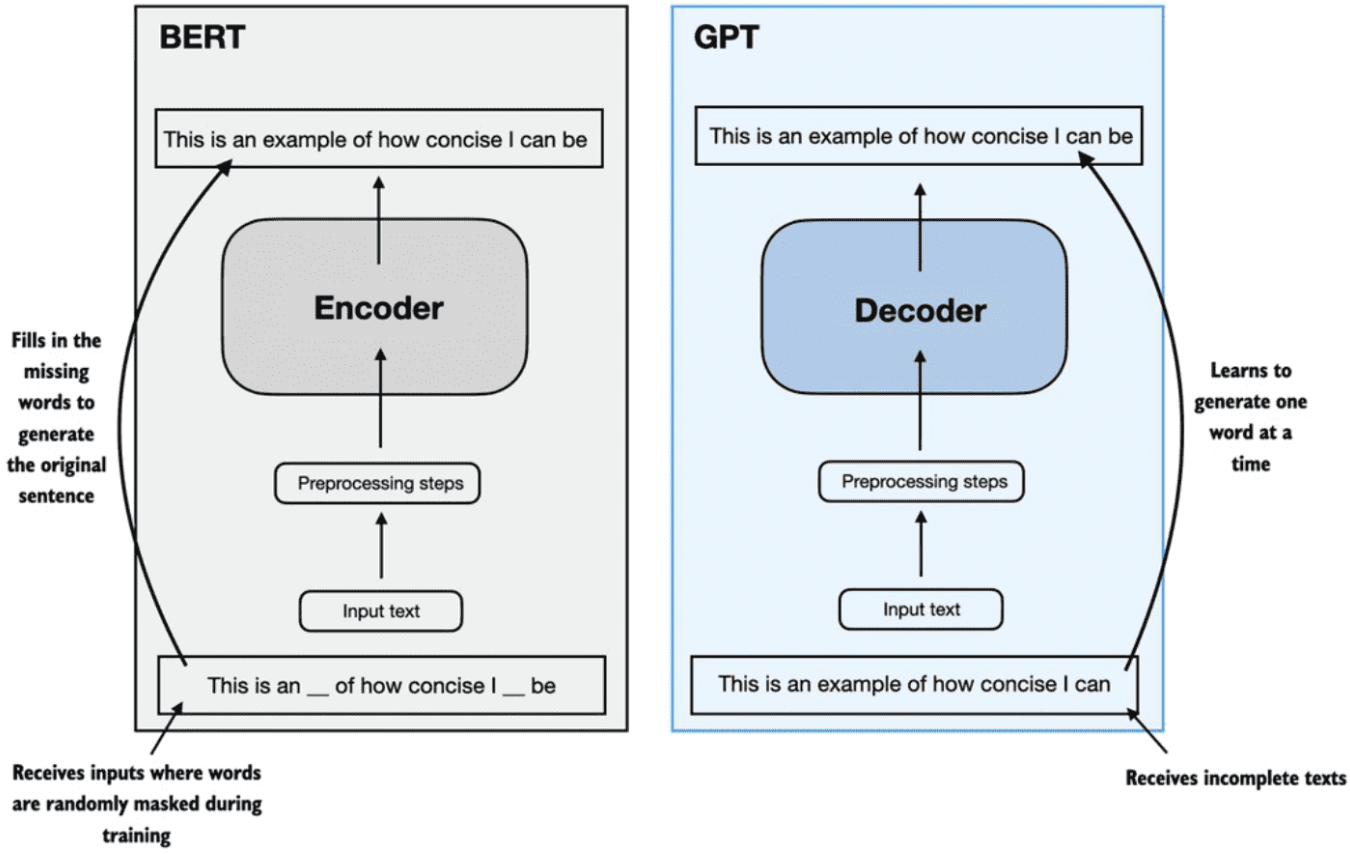


The two most popular categories of finetuning LLMs:

```
1. instruction-finetuning
2. finetuning for classification tasks
```



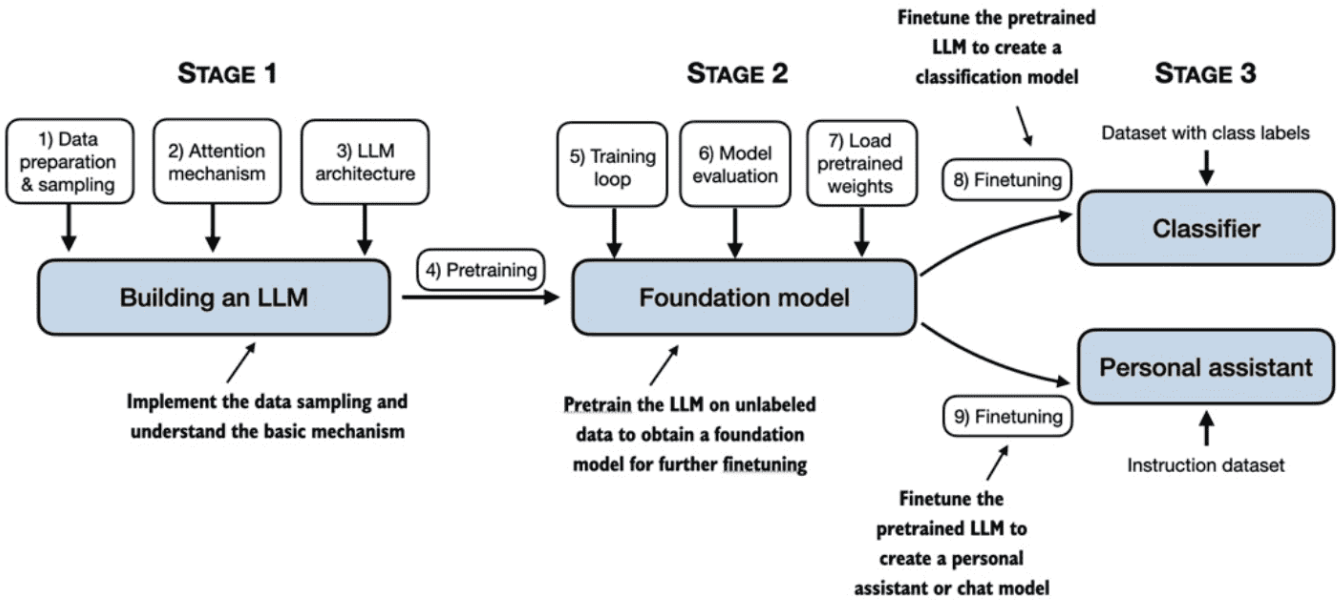
- A key component of transformers and LLMs is the self-attention mechanism, which allows the model to weigh the importance of different words or tokens in a sequence relative to each other.



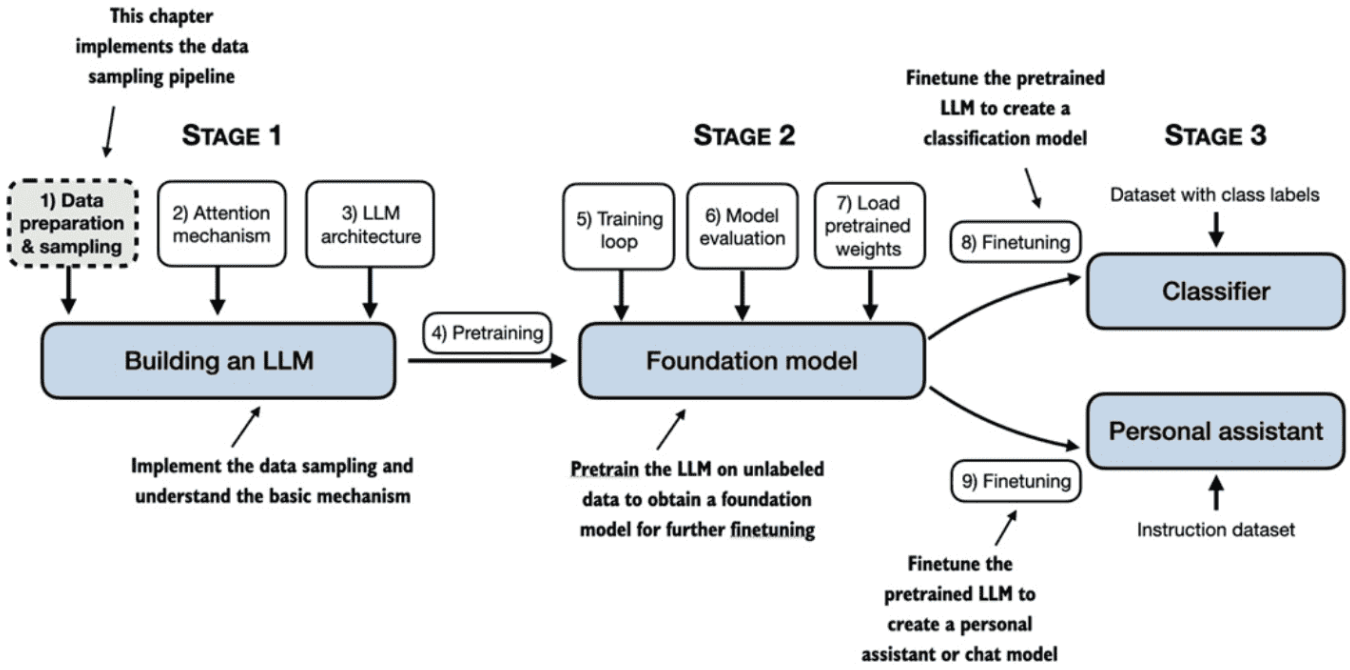
Dataset name	Dataset description	Number of tokens	Proportion in training data
CommonCrawl (filtered)	Web crawl data	410 billion	60%
WebText2	Web crawl data	19 billion	22%
Books1	Internet-based book corpus	12 billion	8%
Books2	Internet-based book corpus	55 billion	8%
Wikipedia	High-quality text	3 billion	3%

```
* Wikipedia corpus consists of English-language Wikipedia
* Books1 is likely a sample from Project Gutenberg: https://www.gutenberg.org/
* Books2 is likely from Libgen: https://en.wikipedia.org/wiki/Library_Genesis
* CommonCrawl is a filtered subset of the CommonCrawl database:
https://commoncrawl.org/
* WebText2 is the text of web pages from all outbound Reddit links from posts with
3+ upvotes.
```

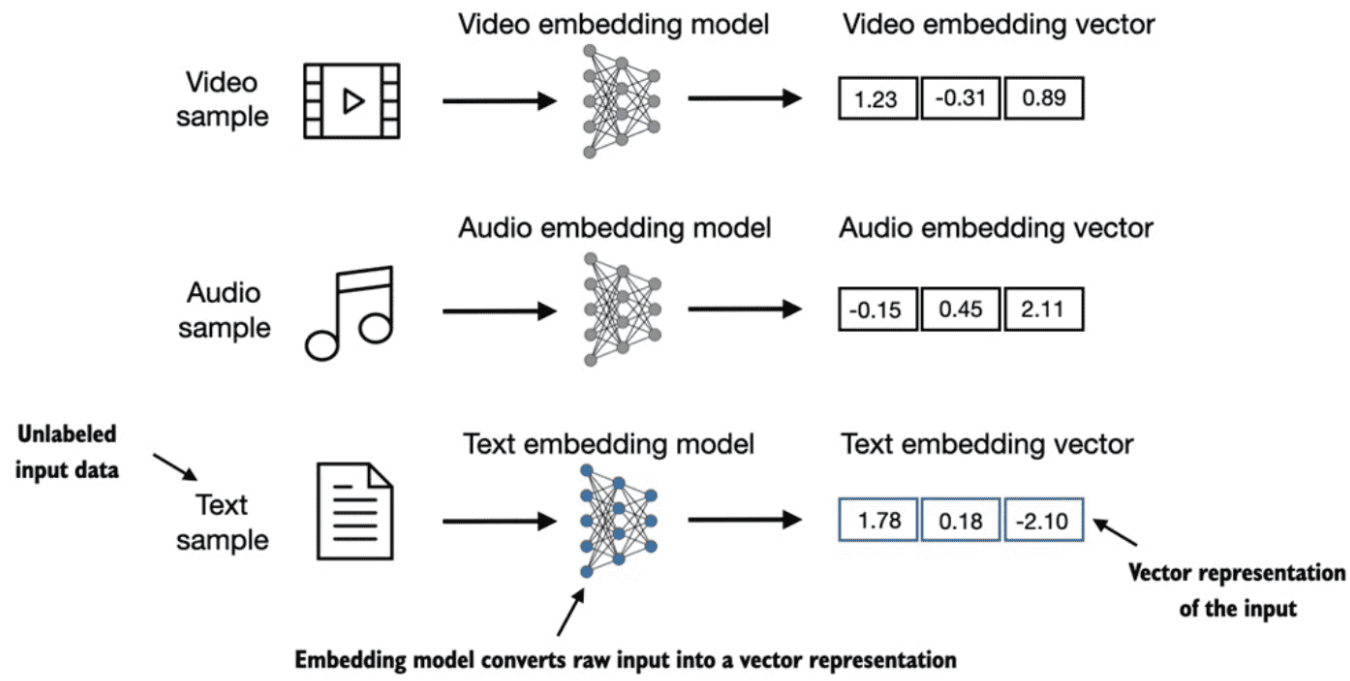
- The ability to perform tasks that the model wasn't explicitly trained to perform is called an "emergent behavior" (涌现行为). "Emergent behavior" refers to abilities (like reasoning, arithmetic, etc.) that a model spontaneously exhibits as its size increases, which were not explicitly trained for. These abilities are not driven by individual parameters or specific tasks but are a spontaneous product of increased system scale and complexity.



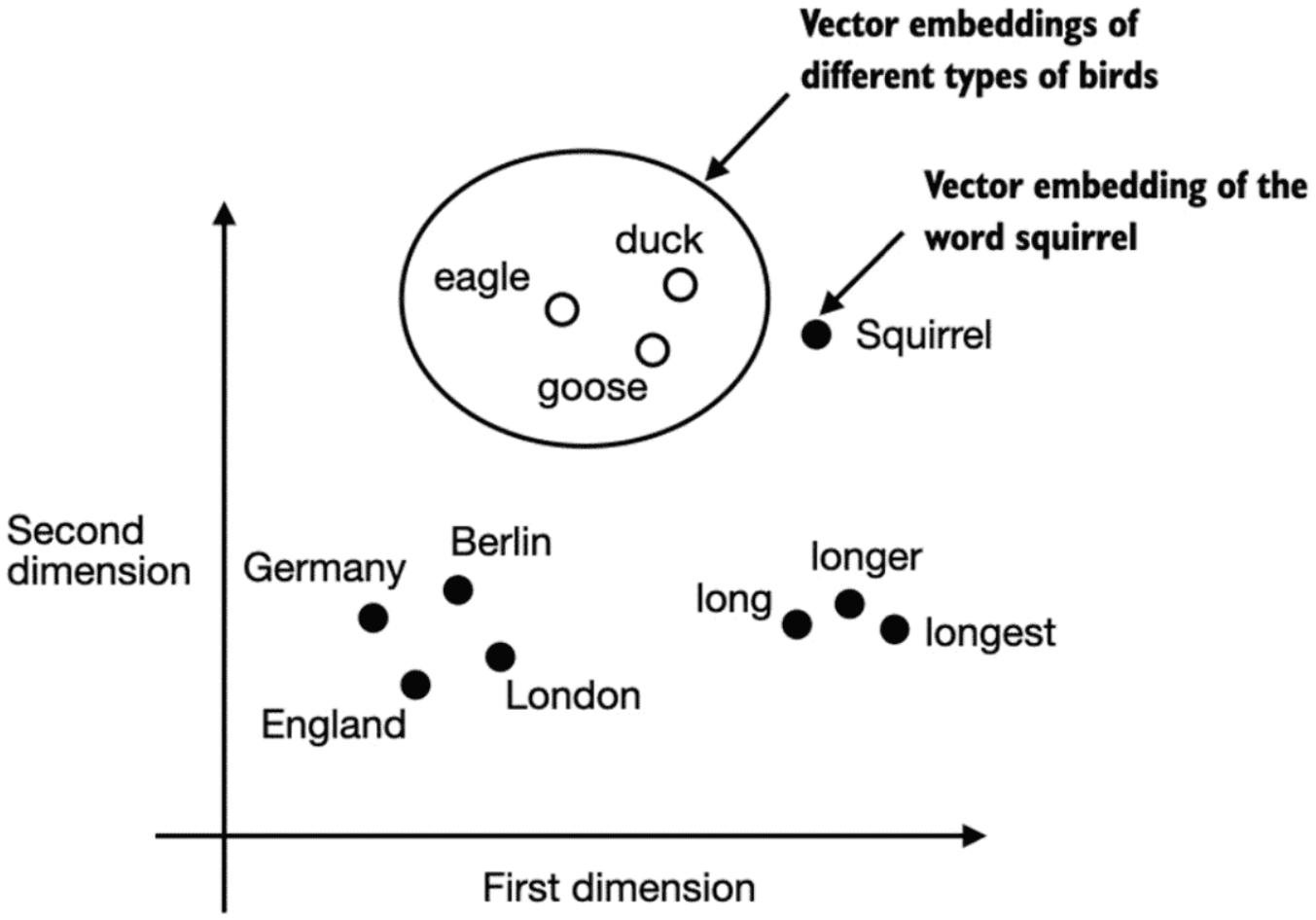
2. Working with Text Data



2.1 Understanding word embeddings

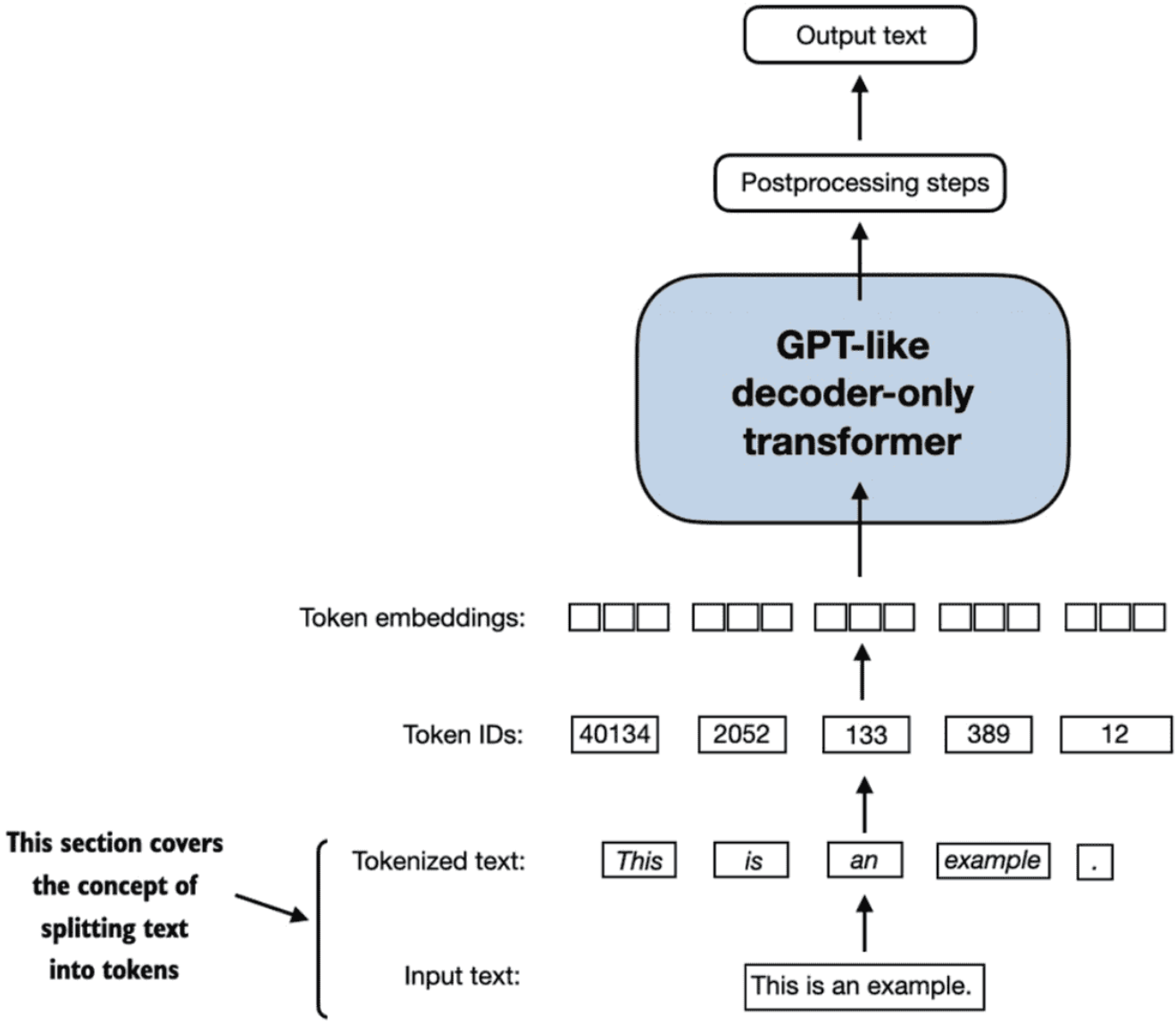


- While word embeddings are the most common form of text embedding, there are also embeddings for sentences, paragraphs, or whole documents. Sentence or paragraph embeddings are popular choices for retrieval- augmented generation.

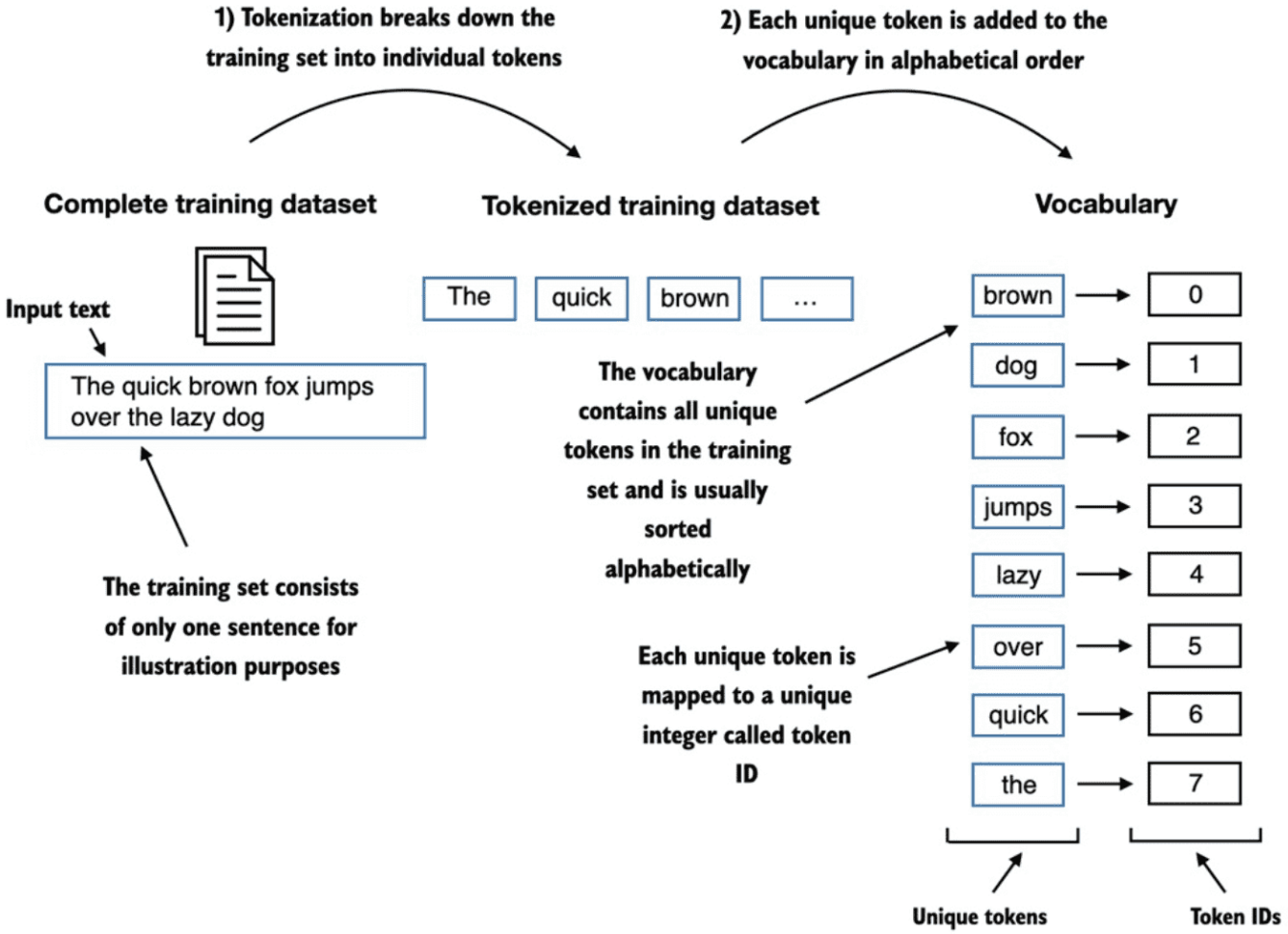


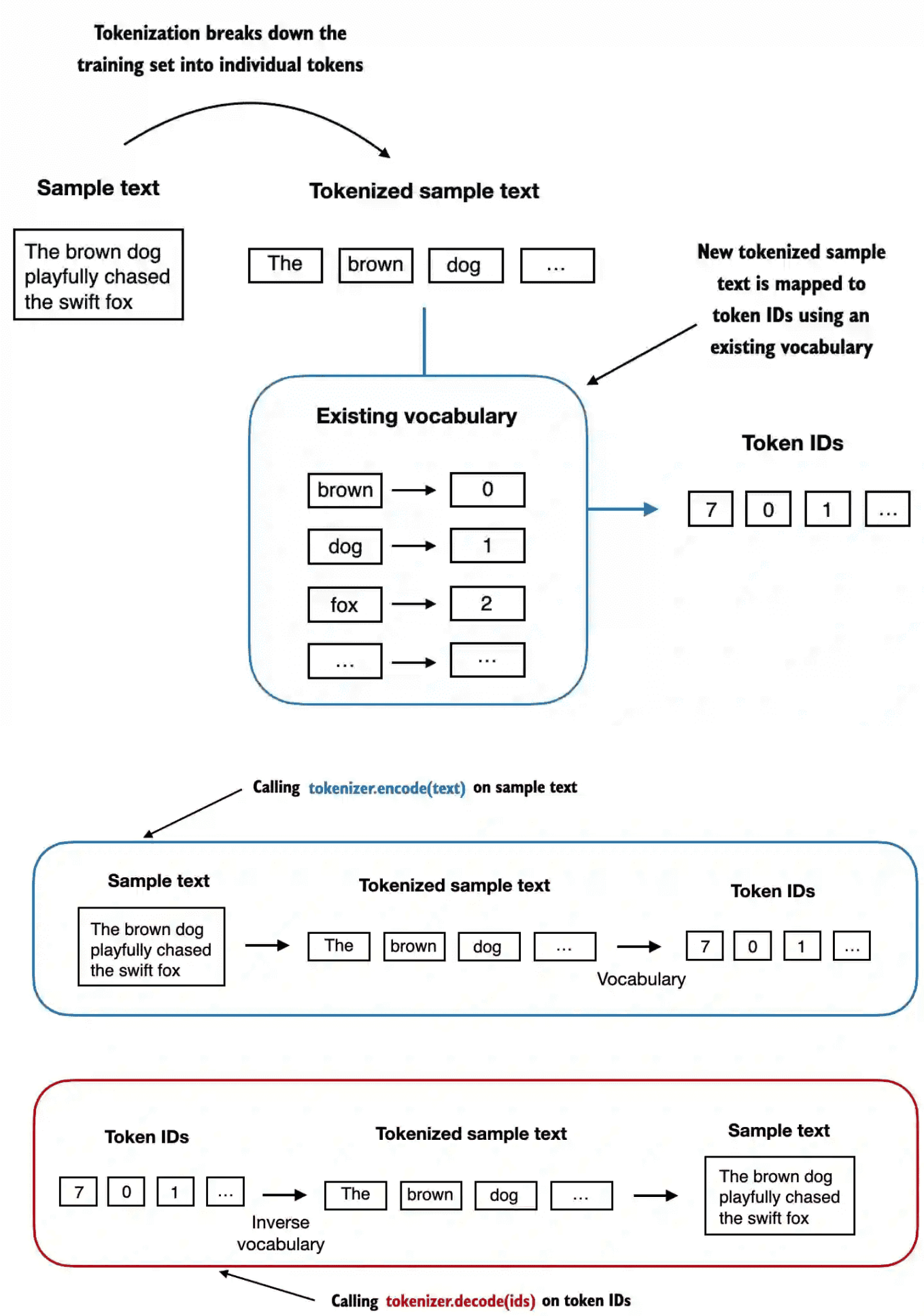
- Word embeddings can have varying dimensions, from one to thousands. As shown in Figure 2.3, we can choose two-dimensional word embeddings for visualization purposes.

2.2 Tokenizing text

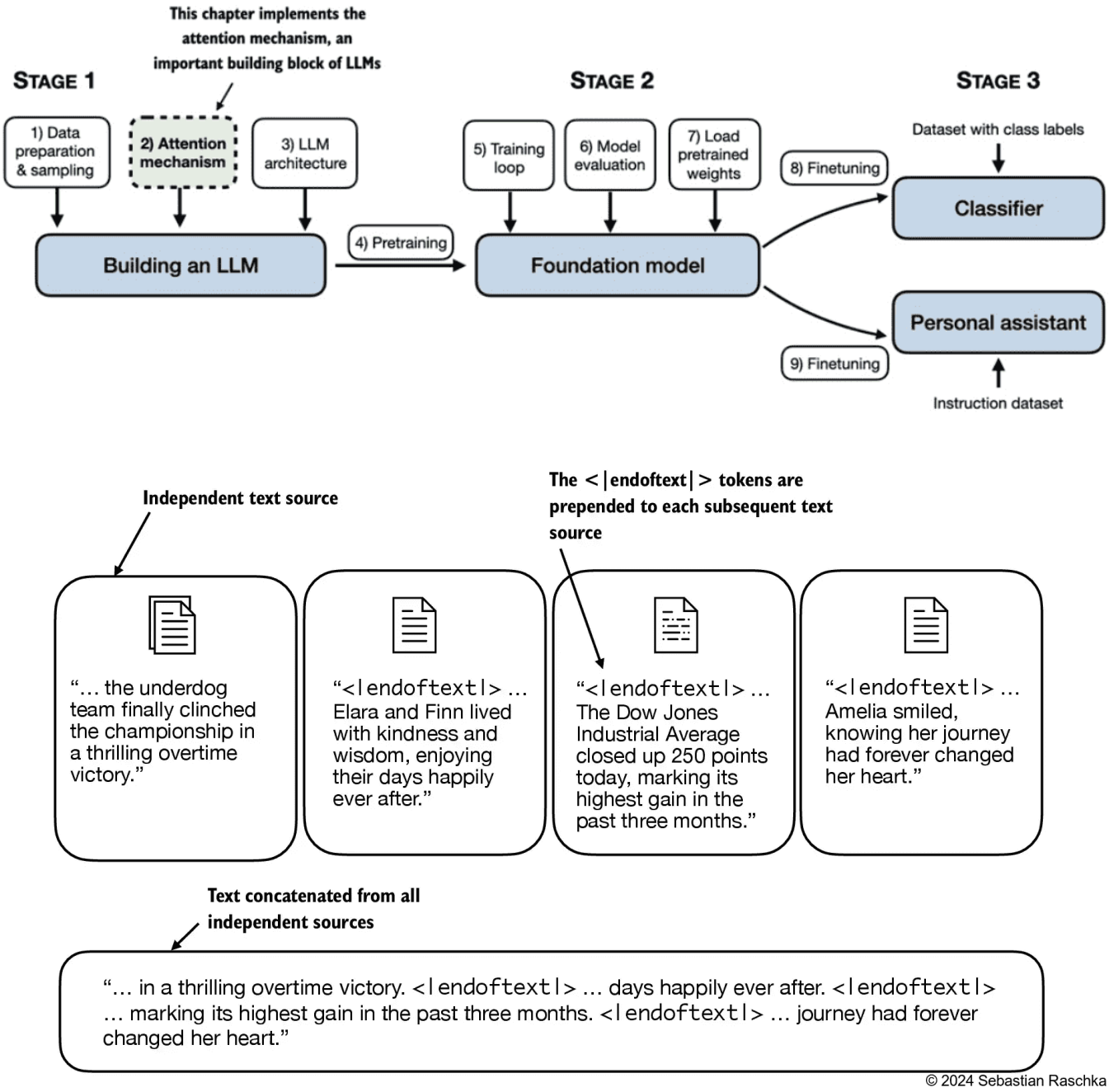


2.3 Converting tokens into token IDs


1) Tokenization breaks down the training set into individual tokens
2) Each unique token is added to the vocabulary in alphabetical order
Complete training dataset
Tokenized training dataset
Vocabulary
Input text
The quick brown fox jumps over the lazy dog
The training set consists of only one sentence for illustration purposes
The
quick
brown
...
The vocabulary contains all unique tokens in the training set and is usually sorted alphabetically
Each unique token is mapped to a unique integer called token ID
brown 0
dog 1
fox 2
jumps 3
lazy 4
over 5
quick 6
the 7
Unique tokens
Token IDs



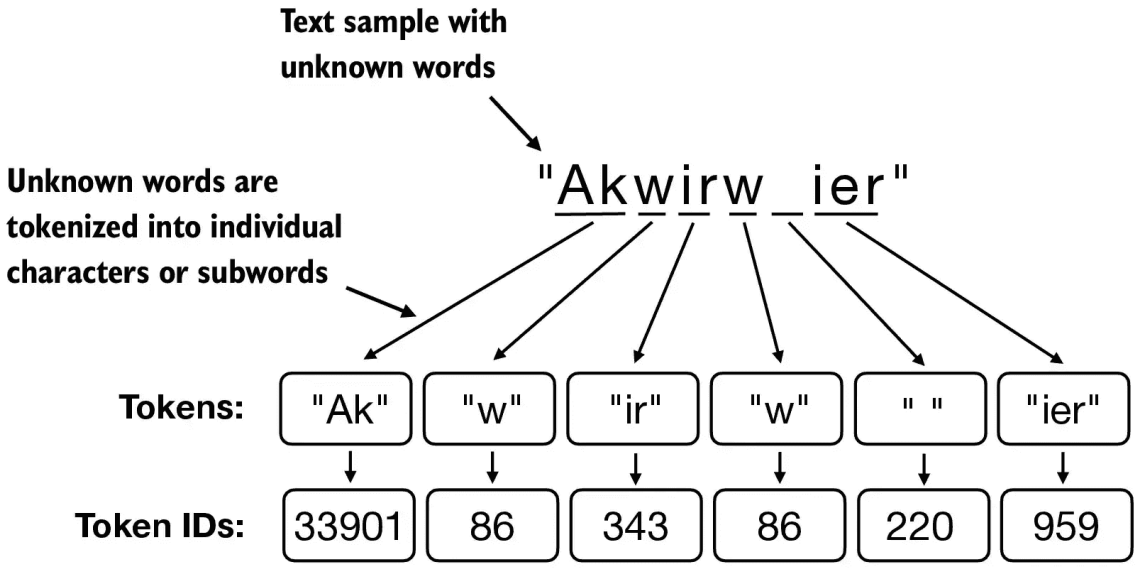
2.4 Adding special context tokens



Additional special tokens:

```
1. [BOS] (beginning of sequence)
2. [EOS] (end of sequence)
3. [PAD] (padding)
4. |endoftext|
5. |unk|
```

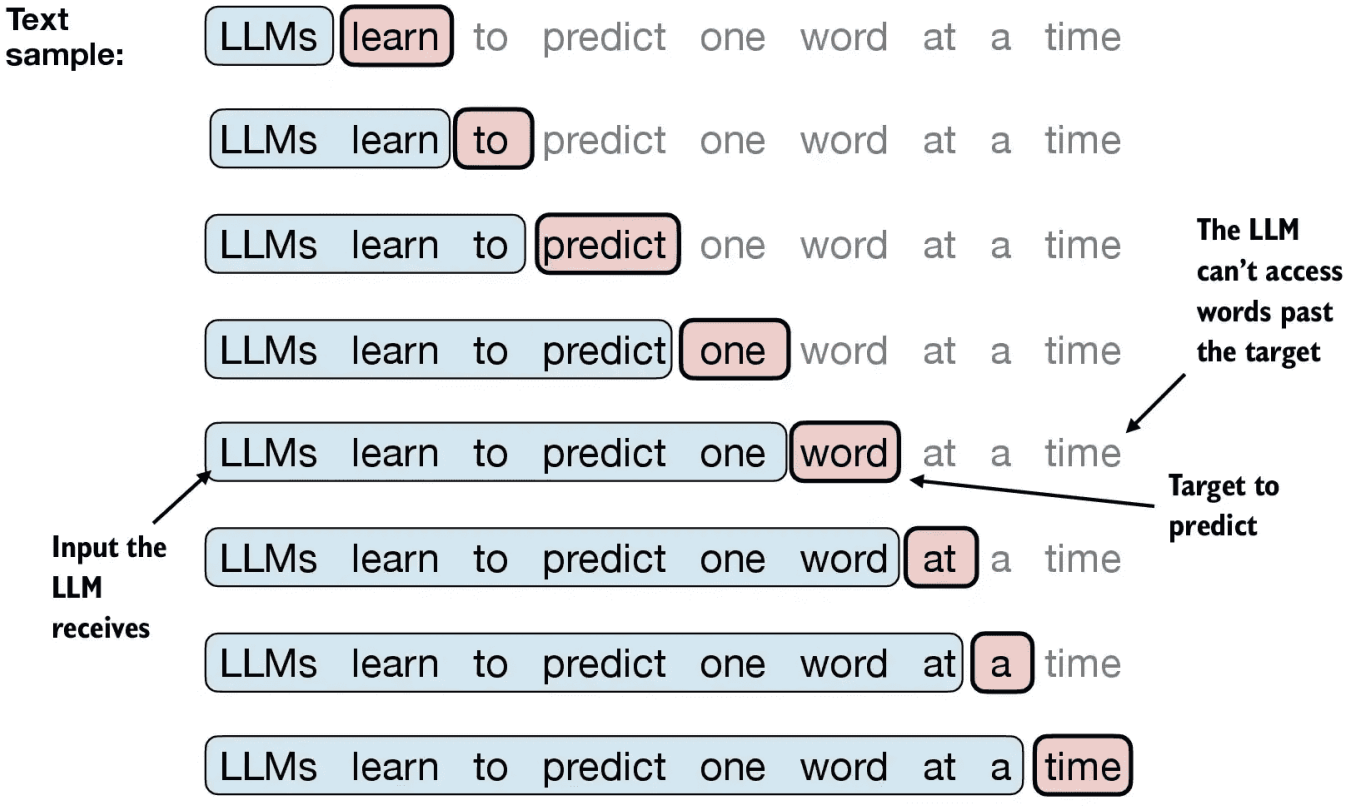
2.5 Byte pair encoding



© 2024 Sebastian Raschka

- The original BPE tokenizer can be found here: https://github.com/openai/gpt-2/blob/master/src/encoder.py
- The BPE tokenizer from OpenAI's open-source tiktoken library, which implements its core algorithms in Rust to improve computational performance.

2.6 Data sampling with a sliding window

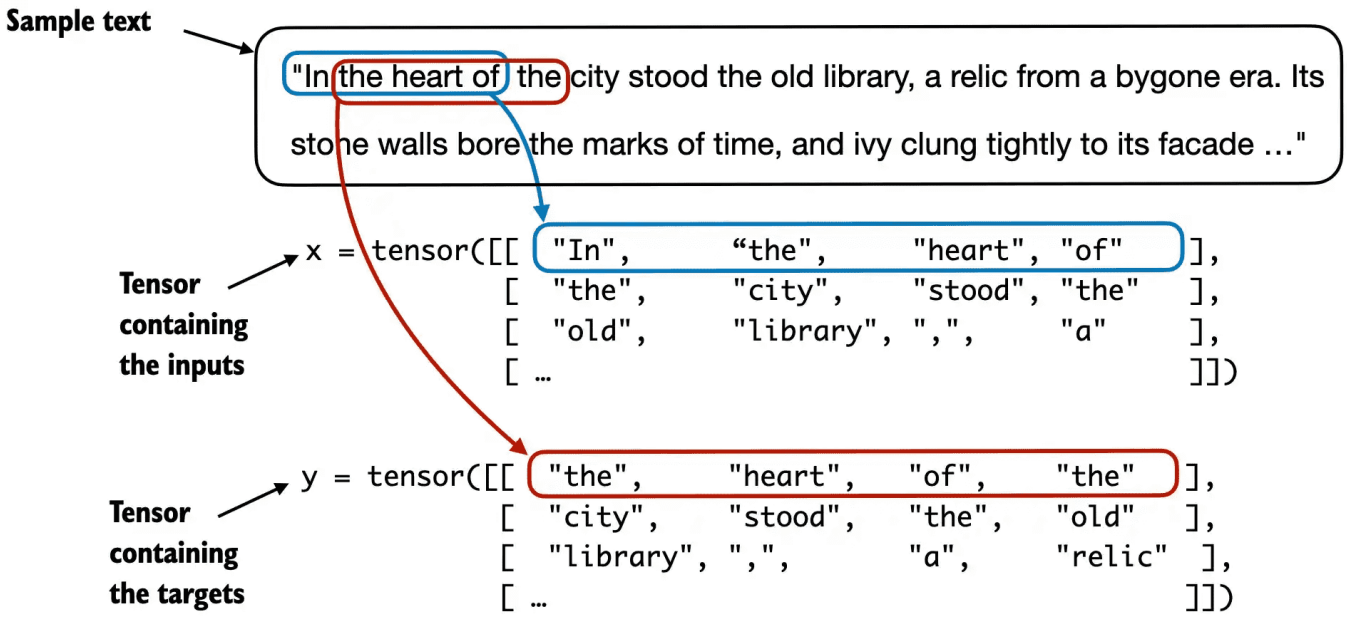


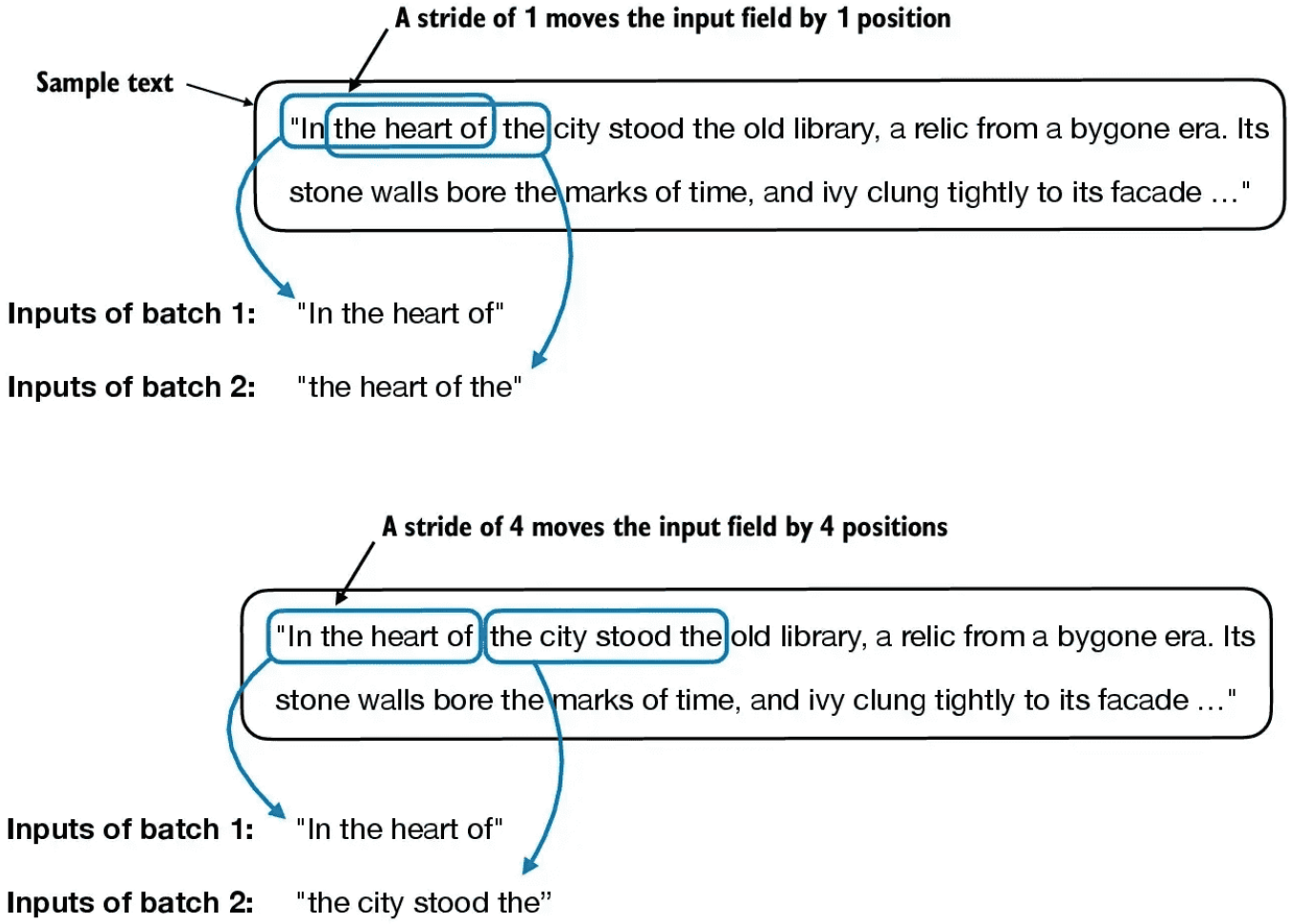
© 2024 Sebastian Raschka

- For each text chunk, we want the inputs and targets
- Since we want the model to predict the next word, the targets are the inputs shifted by one position to the right

The prediction would look like as follows (input-target pairs):

```
and ---->  established
and established ---->  himself
and established himself ---->  in
and established himself in ---->  a
```





- Small batch sizes require less memory during training but lead to more noisy model updates. The batch size is a trade-off and hyperparameter to experiment with when training LLMs.
- Example (batch_size=1, max_length=4, stride=1):

```
[tensor([[  40,  367, 2885, 1464]]), tensor([[ 367, 2885, 1464, 1807]])]
```

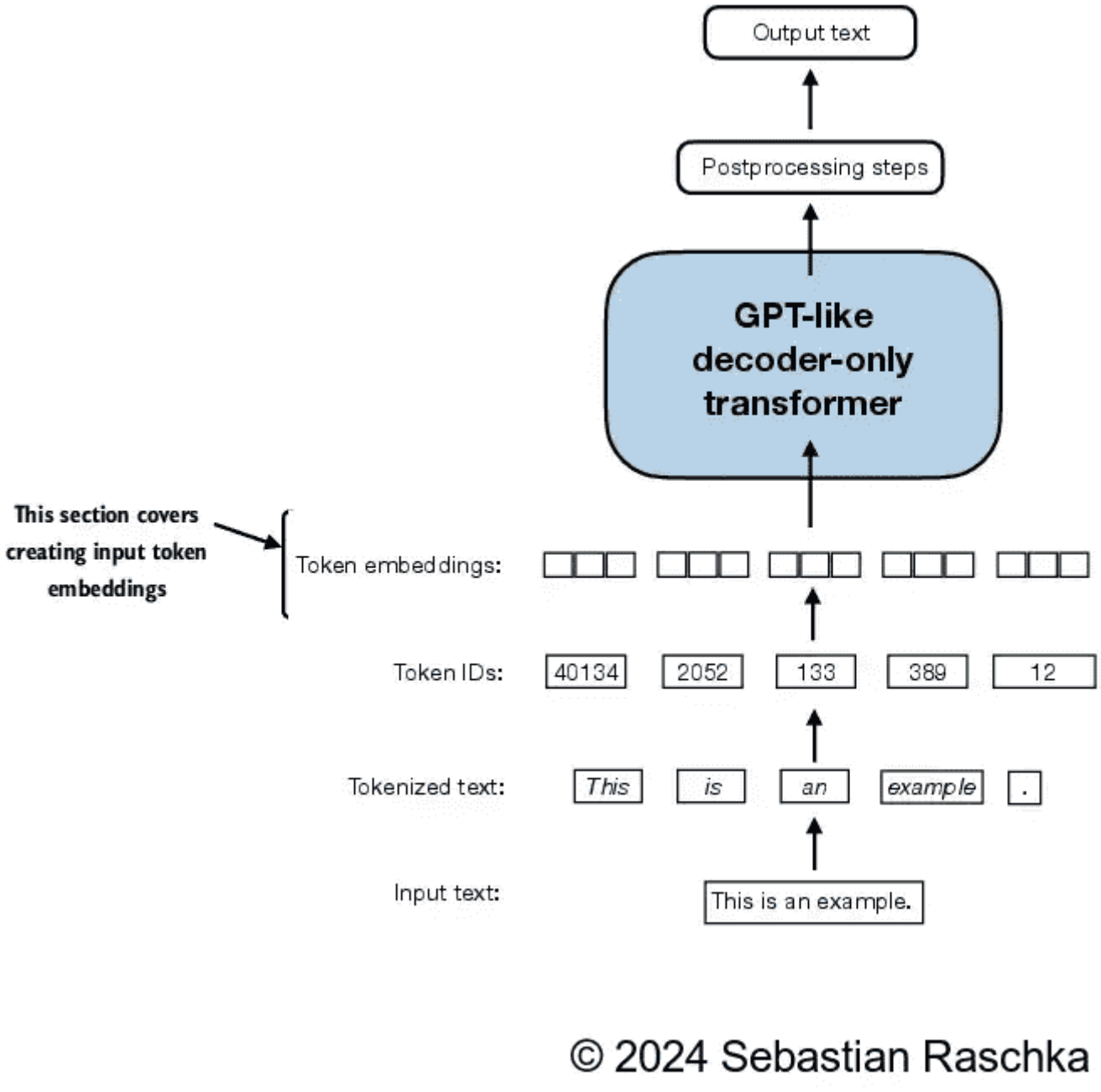
- Example (batch_size=8, max_length=4, stride=4):

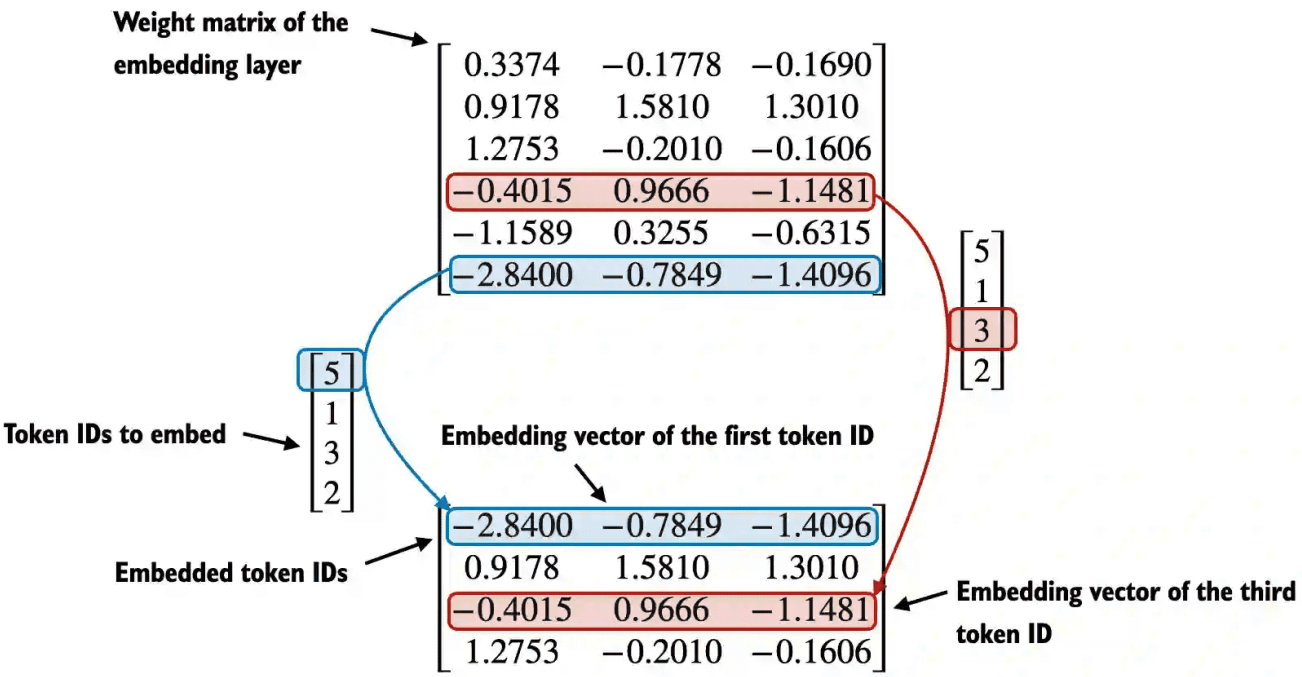
```
Inputs:
 tensor([[   40,   367,  2885,  1464],
        [ 1807,  3619,   402,   271],
        [10899,  2138,   257,  7026],
        [15632,   438,  2016,   257],
        [  922,  5891,  1576,   438],
        [  568,   340,   373,   645],
        [ 1049,  5975,   284,   502],
        [  284,  3285,   326,    11]])

Targets:
 tensor([[  367,  2885,  1464,  1807],
```

```
        [ 3619,   402,   271, 10899],
        [ 2138,   257,  7026, 15632],
        [  438,  2016,   257,   922],
        [ 5891,  1576,   438,   568],
        [  340,   373,   645,  1049],
        [ 5975,   284,   502,   284],
        [ 3285,   326,    11,   287]])
```

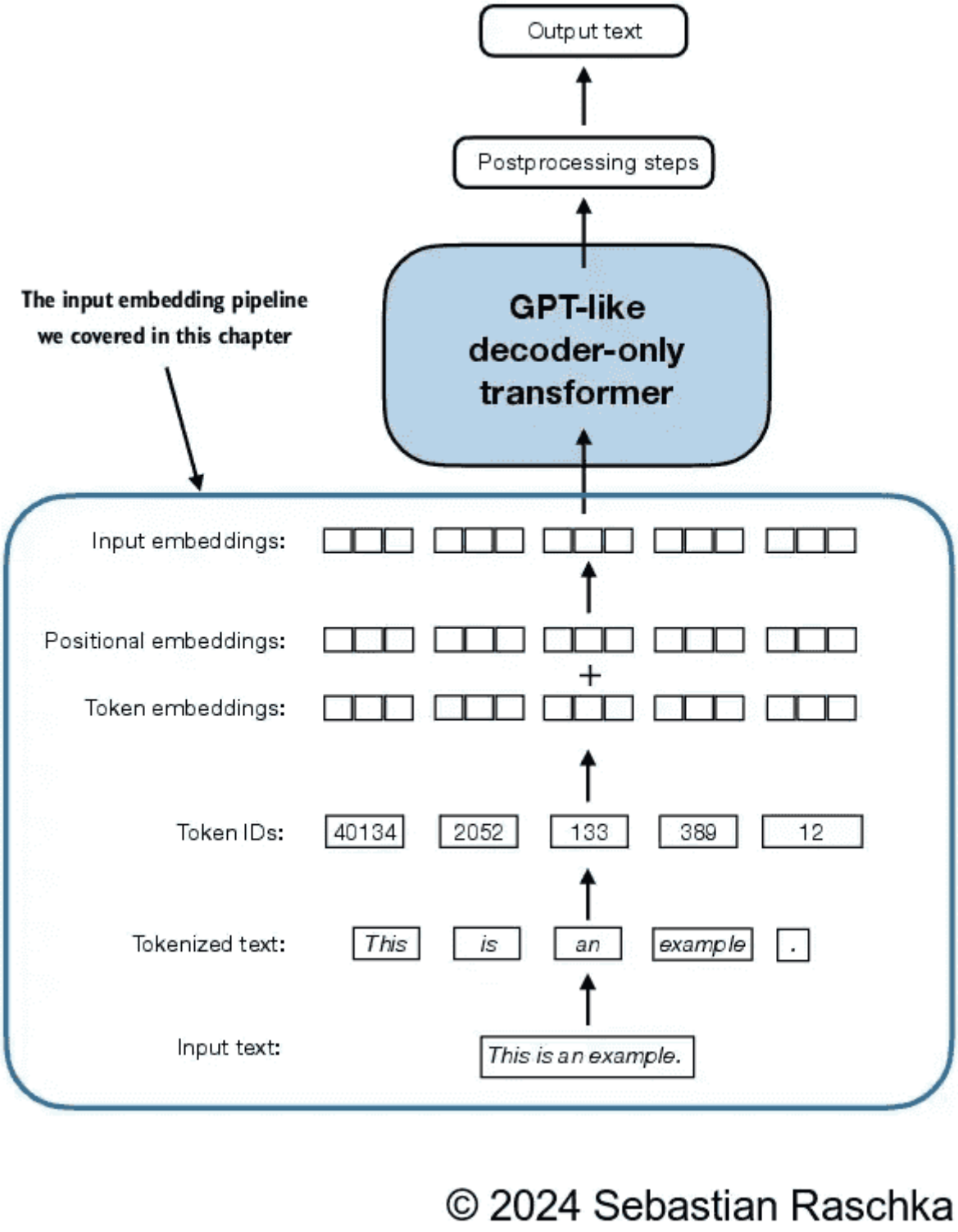
2.7 Creating token embeddings

- We converted the token IDs into a continuous vector representation, the so-called token embeddings.





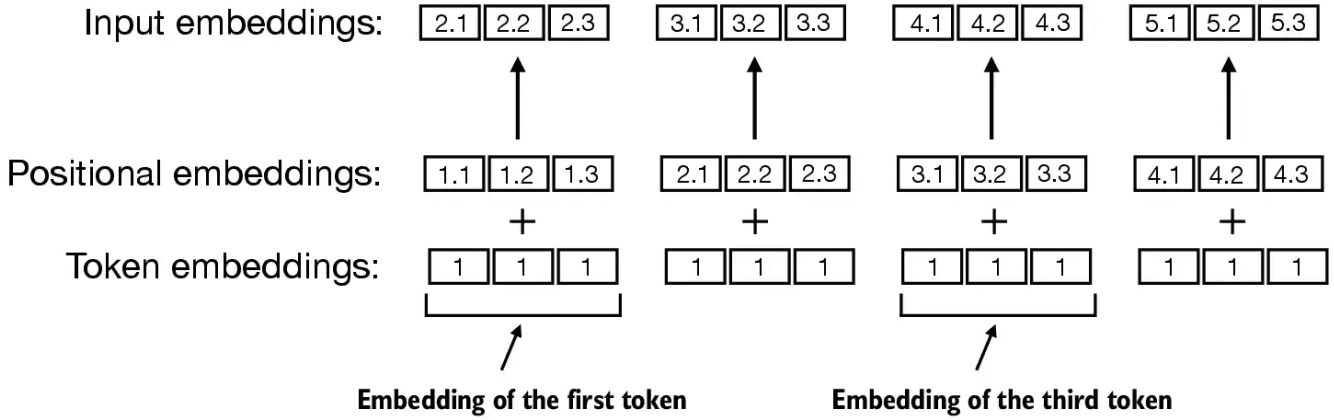
2.8 Encoding word positions



- In principle, the deterministic, position-independent embedding of the token ID is good for reproducibility purposes.
- However, since the self-attention mechanism of LLMs itself is also position-agnostic, it is helpful to inject additional position information into the LLM.

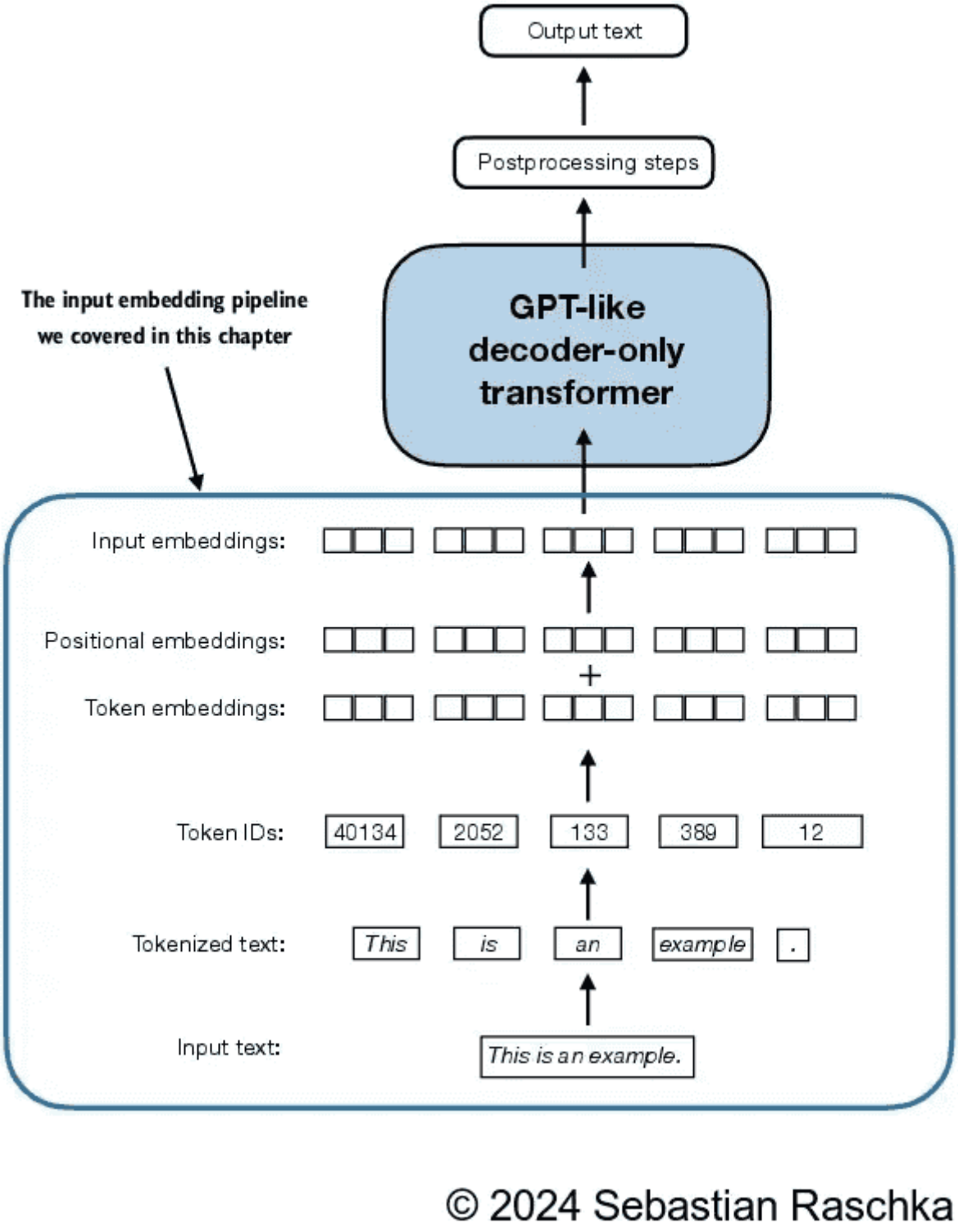
Two broad categories of position-aware embeddings:

```
1. relative positional embeddings
2. absolute positional embeddings
```



© 2024 Sebastian Raschka

- Instead of focusing on the absolute position of a token, the emphasis of relative positional embeddings is on the relative position or distance between tokens. This means the model learns the relationships in terms of "how far apart" rather than "at which exact position." The advantage here is that the model can generalize better to sequences of varying lengths, even if it hasn't seen such lengths during training.
- OpenAI's GPT models use absolute positional embeddings that are optimized during the training process rather than being fixed or predefined like the positional encodings in the original Transformer model. This optimization process is part of the model training itself, which we will implement later in this book. For now, let's create the initial positional embeddings to create the LLM inputs for the upcoming chapters.



- `context_length` is a variable that represents the supported input size of the LLM.

2.9 Summary

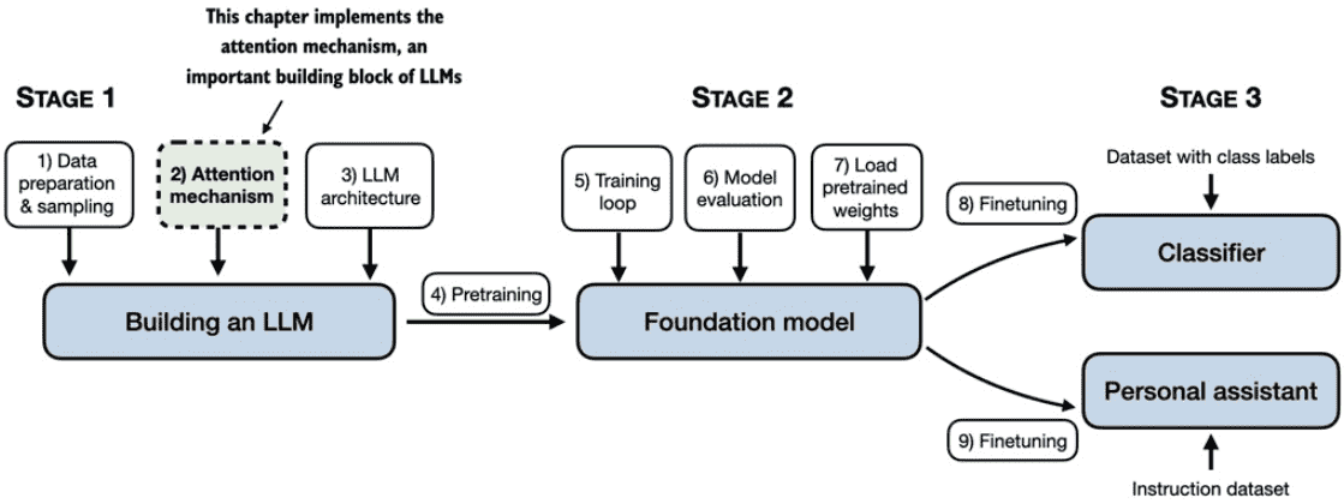
- LLMs require textual data to be converted into numerical vectors, known as embeddings since they can't process raw text. Embeddings transform discrete data (like words or images) into continuous vector spaces, making them compatible with neural network operations.
- As the first step, raw text is broken into tokens, which can be words or characters. Then, the tokens are converted into integer representations, termed token IDs.
- Special tokens, such as `<|unk|>` and `<|endoftext|>`, can be added to enhance the model's understanding and handle various contexts, such as unknown words or marking the boundary between

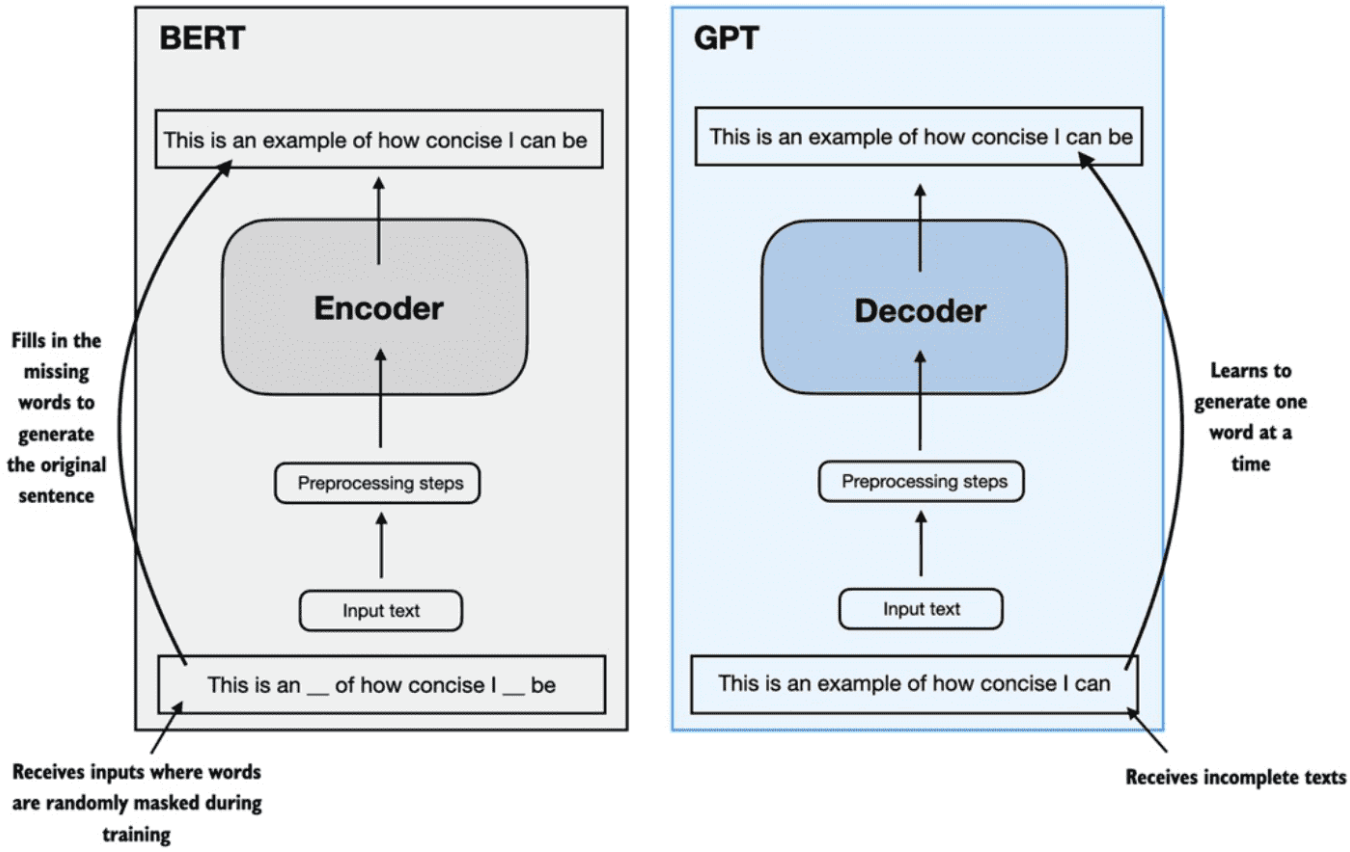
unrelated texts.

- The `byte pair encoding (BPE)` tokenizer used for LLMs like GPT-2 and GPT-3 can efficiently handle unknown words by breaking them down into subword units or individual characters.
- We use a sliding window approach on tokenized data to generate `input-target pairs` for LLM training.
- Embedding layers in PyTorch function as a lookup operation, retrieving vectors corresponding to token IDs. The resulting embedding vectors provide continuous representations of tokens, which is crucial for training deep learning models like LLMs.
- While token embeddings provide consistent vector representations for each token, they lack a sense of the token's position in a sequence. To rectify this, two main types of positional embeddings exist: absolute and relative. OpenAI's GPT models utilize absolute positional embeddings that are added to the token embedding vectors and are optimized during the model training.

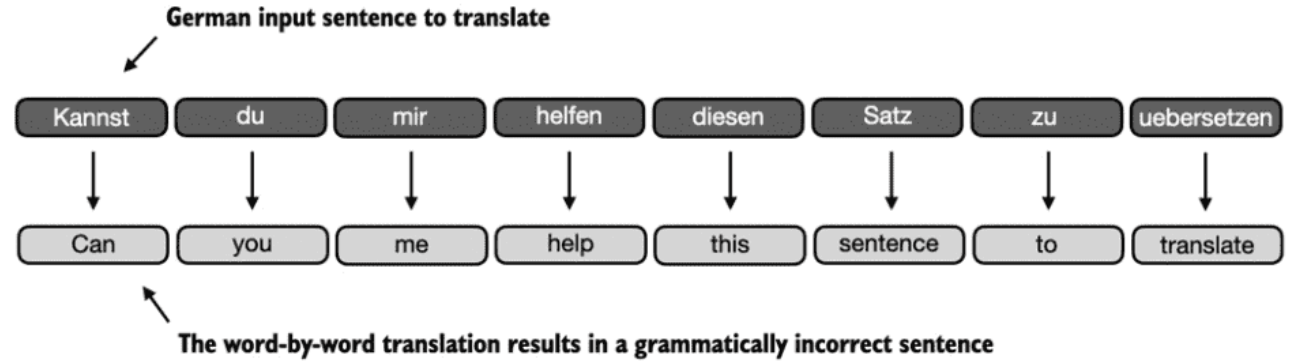
3. Coding Attention Mechanisms

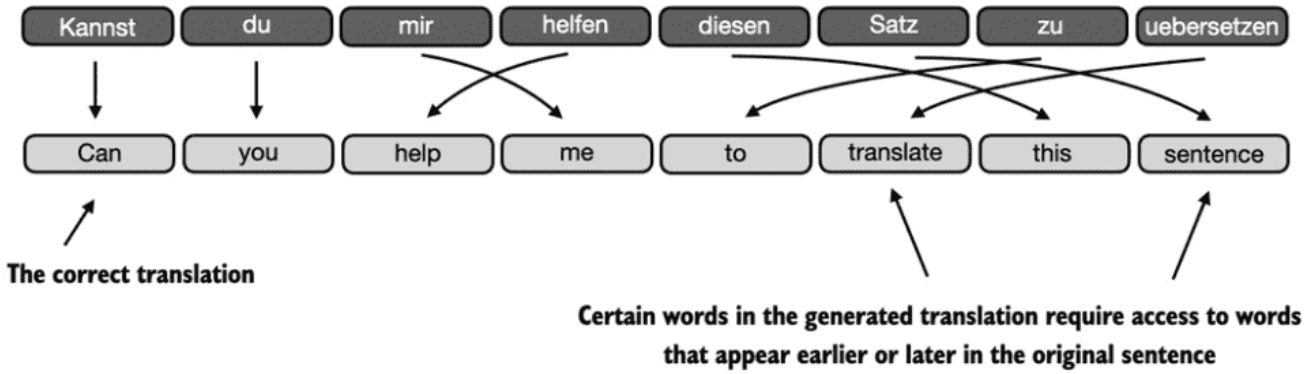
- Exploring the reasons for using attention mechanisms in neural networks
- Introducing a basic self-attention framework and progressing to an enhanced self-attention mechanism
- Implementing a causal attention module that allows LLMs to generate one token at a time
- Masking randomly selected attention weights with dropout to reduce overfitting
- Stacking multiple causal attention modules into a multi-head attention module





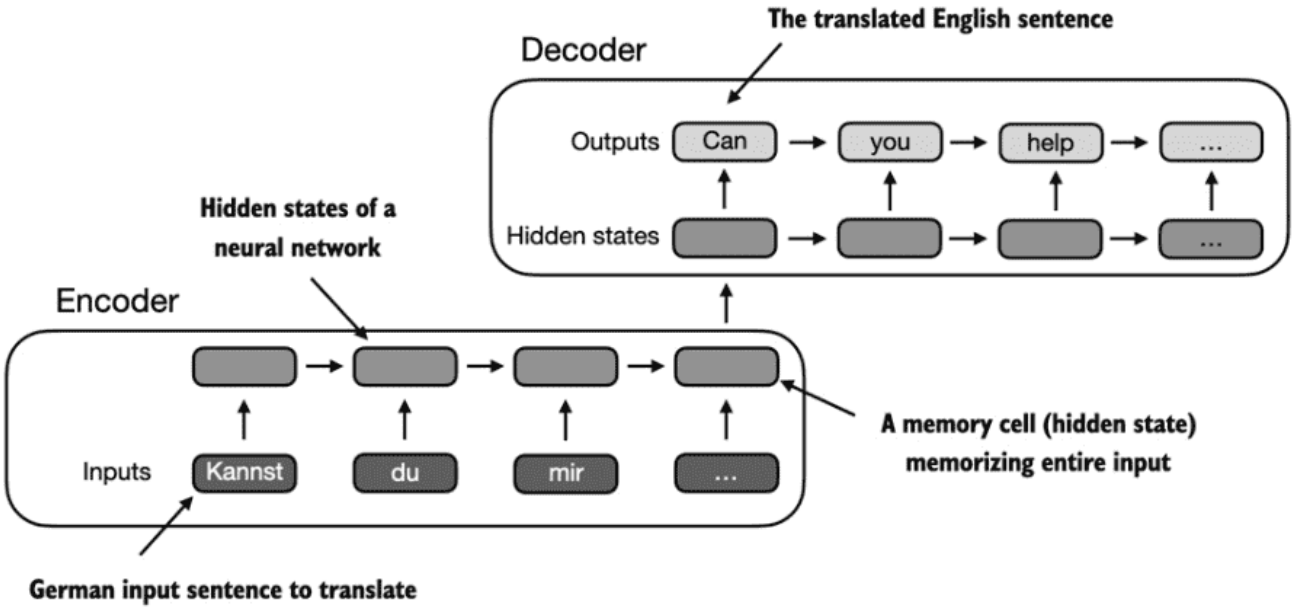
3.1 The problem with modeling long sequences





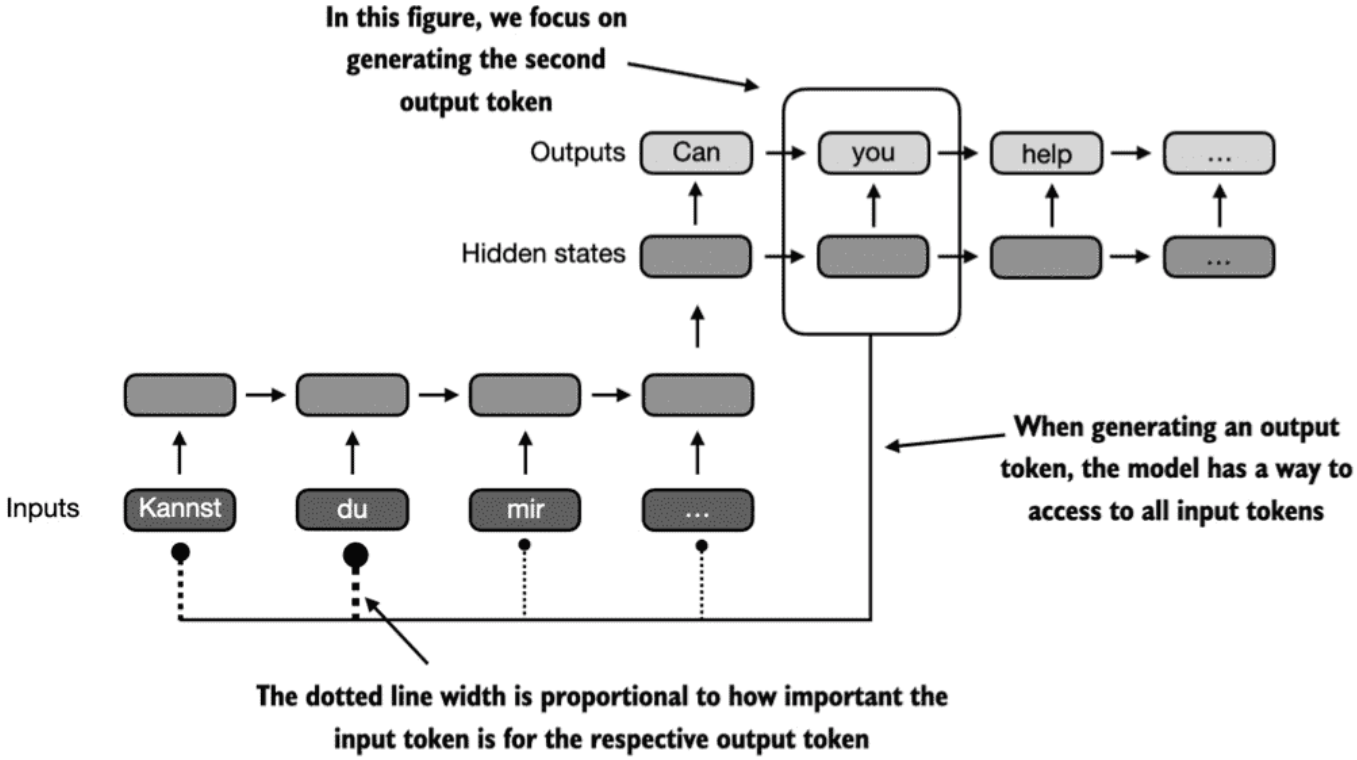
- To address the issue that we cannot translate text word by word, it is common to use a deep neural network with two submodules, a so-called `encoder` and `decoder`. The job of the encoder is to first read

in and process the entire text, and the decoder then produces the translated text. e.g., `encoder-decoder RNNs`



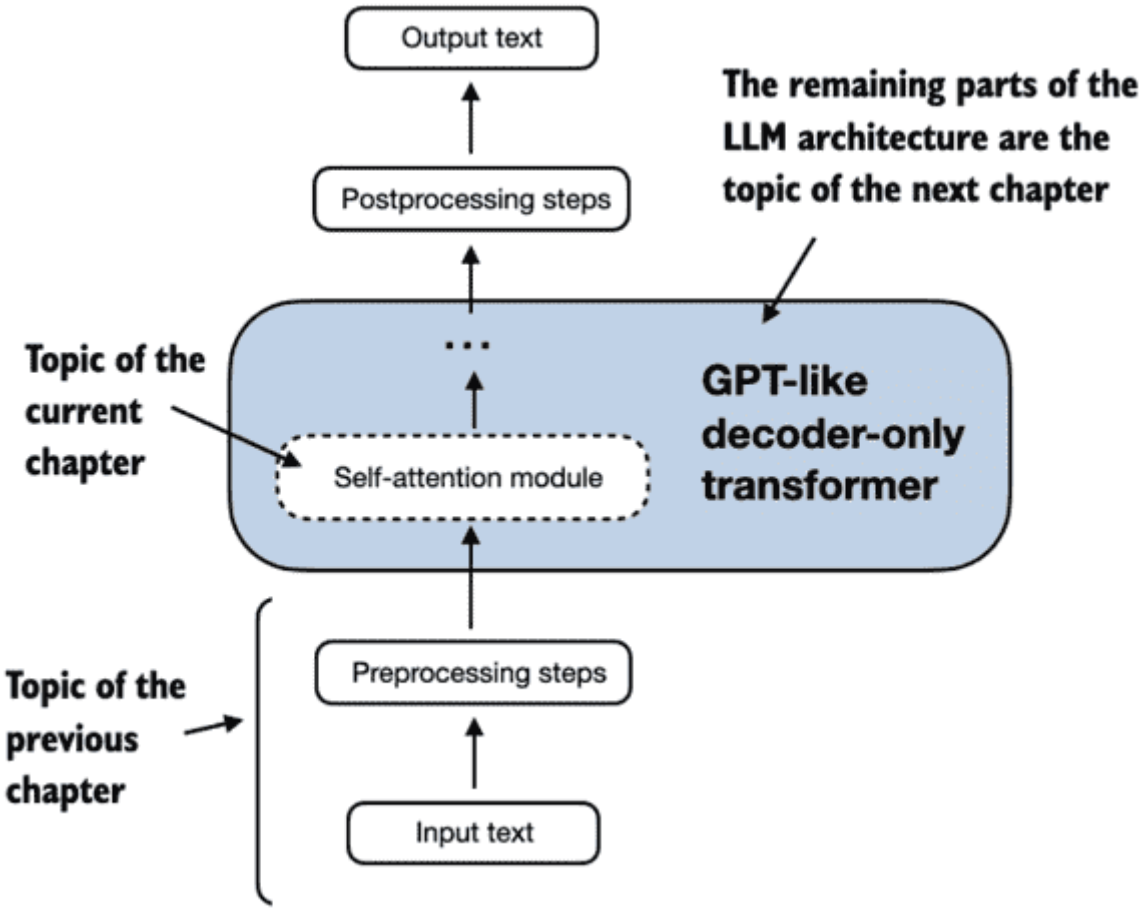
- The big issue and limitation of encoder-decoder RNNs is that the RNN can't directly access earlier hidden states from the encoder during the decoding phase. Consequently, it relies solely on the current hidden state, which encapsulates all relevant information. This can lead to a loss of context, especially in complex sentences where dependencies might span long distances.

3.2 Capturing data dependencies with attention mechanisms



- Self-attention is a mechanism that allows each position in the input sequence to attend to (注意) all positions in the same sequence when computing the representation of a sequence. Self-attention is a

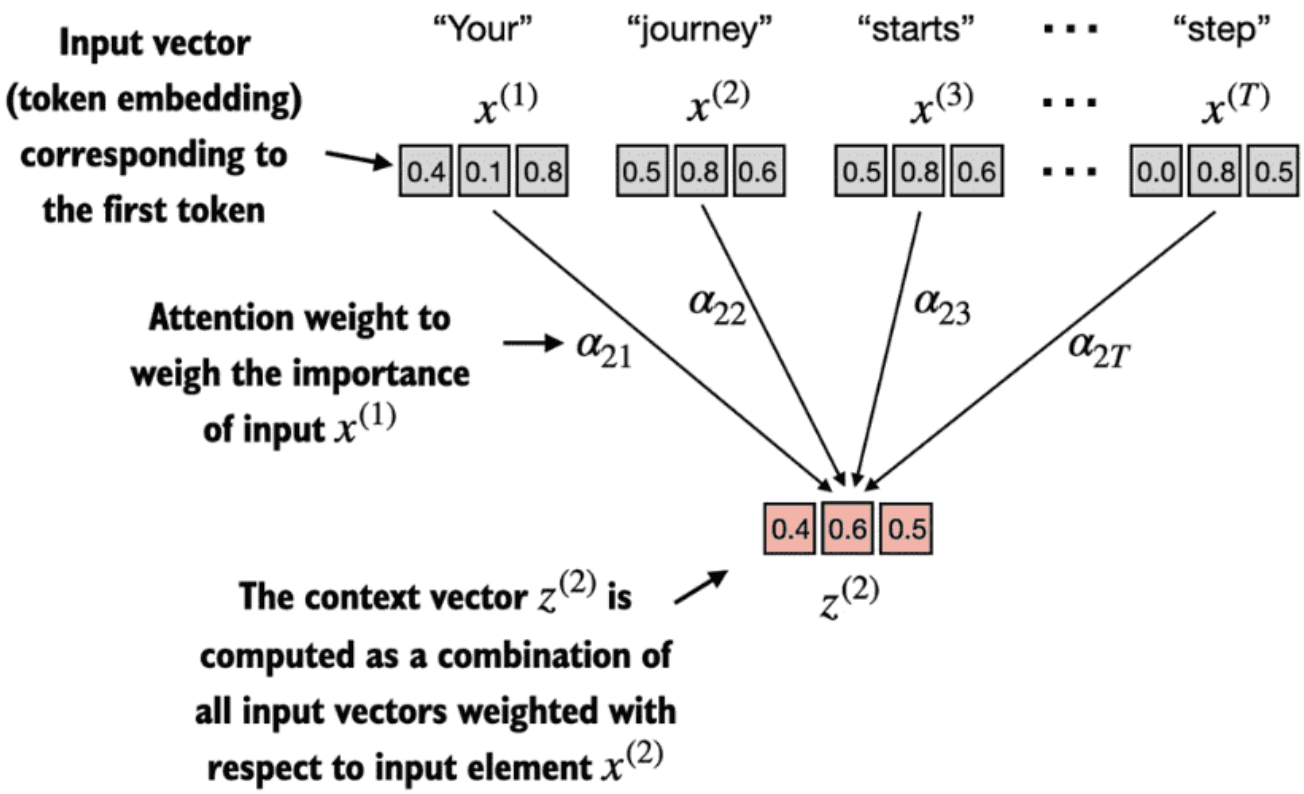
key component of contemporary (当代) LLMs based on the transformer architecture, such as the GPT series.



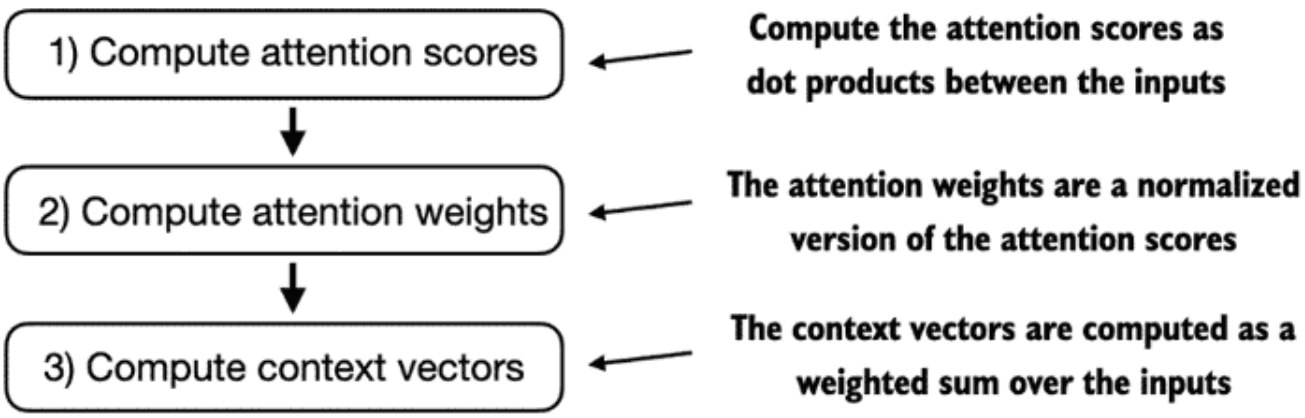
3.3 Attending to different parts of the input with self-attention

- The “self” in self-attention
- In self-attention, the “self” refers to the mechanism’s ability to compute attention weights by relating different positions within a single input sequence. It assesses and learns the relationships and dependencies between various parts of the input itself, such as words in a sentence or pixels in an image.
- This is in contrast to traditional attention mechanisms, where the focus is on the relationships between elements of two different sequences, such as in sequence-to-sequence models where the attention might be between an input sequence and an output sequence, such as the example depicted in Figure 3.5.

3.3.1 A simple self-attention mechanism without trainable weights

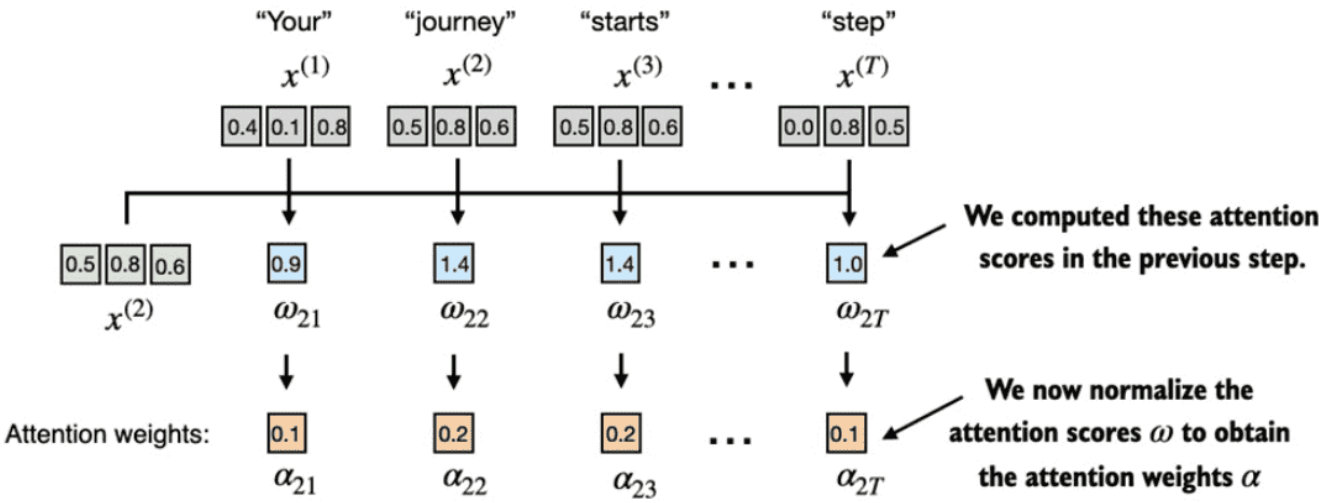


- For example, consider an input text like `"Your journey starts with one step."` In this case, each element of the sequence, such as $x^{(1)}$, corresponds to a `d-dimensional` embedding vector representing a specific token, like “Your.” These input vectors are shown as `3-dimensional` embeddings.
- Let’s focus on the `embedding vector` of the second input element, $x^{(2)}$ (which corresponds to the token “journey”), and the corresponding `context vector`, $z^{(2)}$. This enhanced `context vector`, $z^{(2)}$, is an embedding that contains information about $x^{(2)}$ and all other input elements $x^{(1)}$ to $x^{(T)}$.
- In self-attention, `context vectors` play a crucial role. Their purpose is to create enriched representations of each element in an input sequence (like a sentence) by incorporating information from all other elements in the sequence. This is essential in LLMs, which need to understand the relationship and relevance of words in a sentence to each other. Later, we will add trainable weights that help an LLM learn to construct these context vectors so that they are relevant for the LLM to generate the next token.



```
import torch
inputs = torch.tensor(
  [[0.43, 0.15, 0.89], # Your     (x^1)
    [0.55, 0.87, 0.66], # journey  (x^2)
    [0.57, 0.85, 0.64], # starts    (x^3)
    [0.22, 0.58, 0.33], # with      (x^4)
    [0.77, 0.25, 0.10], # one       (x^5)
    [0.05, 0.80, 0.55]] # step      (x^6)
)


query = inputs[1]  #A
attn_scores_2 = torch.empty(inputs.shape[0])
for i, x_i in enumerate(inputs):
    attn_scores_2[i] = torch.dot(x_i, query)
print(attn_scores_2)
# Output
# tensor([0.9544, 1.4950, 1.4754, 0.8434, 0.7070, 1.0865])
```

- Besides viewing the dot product operation as a mathematical tool that combines two vectors to produce a scalar value, the dot product is also a measure of similarity, as it quantifies how aligned two vectors are: higher dot products indicate a higher degree of alignment or similarity between vectors. In the context of self-attention mechanisms, the dot product determines how much elements in the sequence attend to each other: the higher the dot product, the higher the similarity and attention score between two elements.



- The main goal behind normalization is to obtain attention weights that sum to 1.

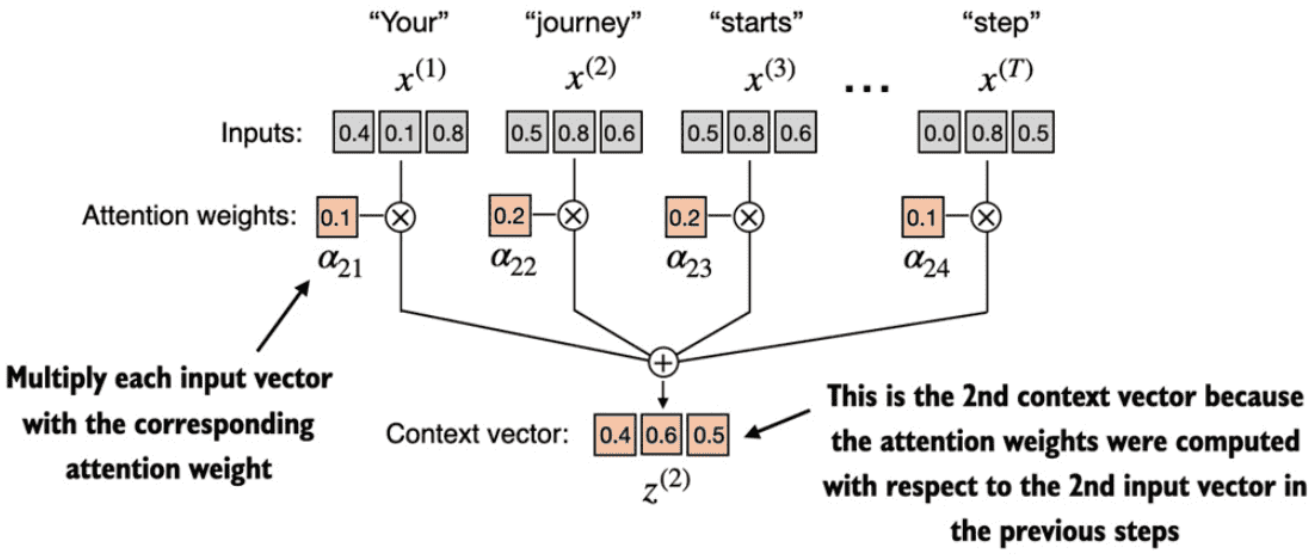
```
attn_weights_2_tmp = attn_scores_2 / attn_scores_2.sum()
print("Attention weights:", attn_weights_2_tmp)
print("Sum:", attn_weights_2_tmp.sum())
# Output
# Attention weights: tensor([0.1455, 0.2278, 0.2249, 0.1285, 0.1077, 0.1656])
# Sum: tensor(1.0000)
```

- In practice, it's more common and advisable to use the softmax function for normalization.
- In addition, the softmax function ensures that the attention weights are always positive.

```
def softmax_naive(x):
    return torch.exp(x) / torch.exp(x).sum(dim=0)
attn_weights_2_naive = softmax_naive(attn_scores_2)
print("Attention weights:", attn_weights_2_naive)
print("Sum:", attn_weights_2_naive.sum())
# Output
# Attention weights: tensor([0.1385, 0.2379, 0.2333, 0.1240, 0.1082, 0.1581])
# Sum: tensor(1.)
```

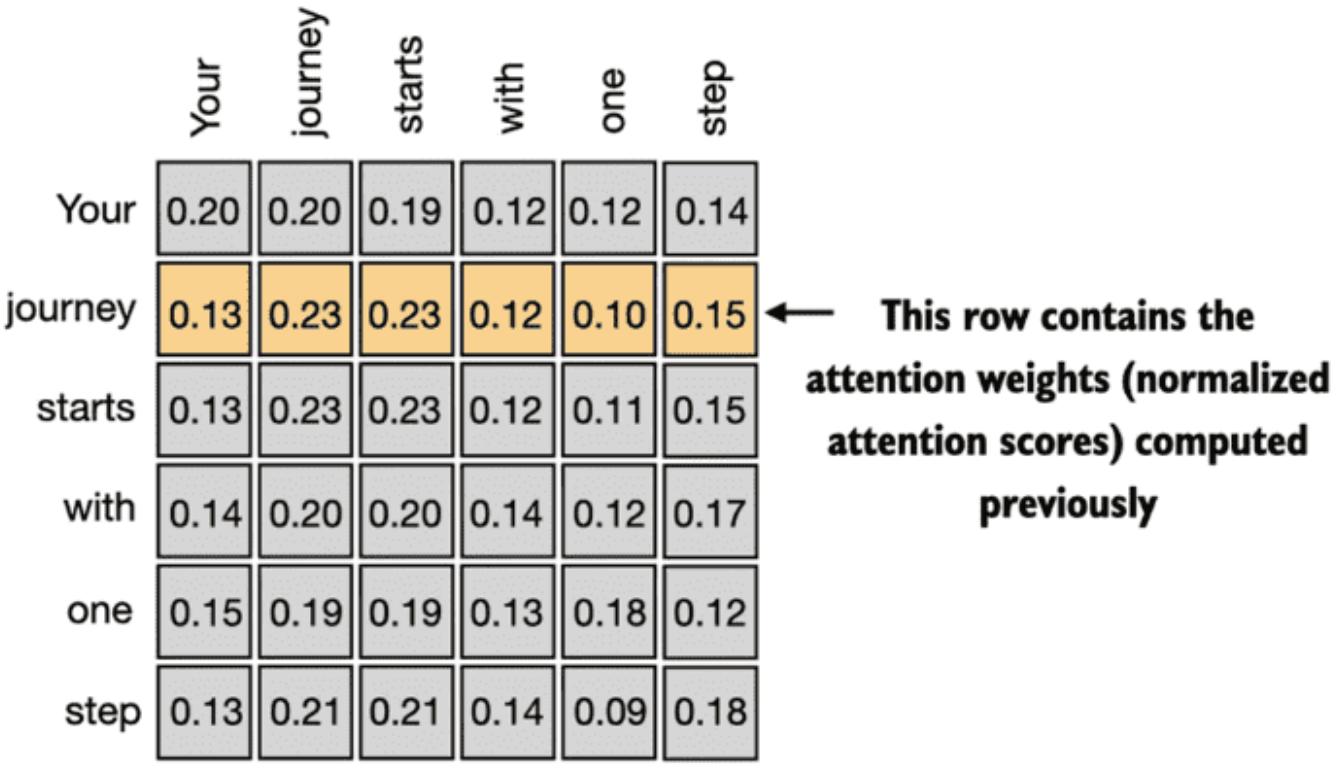
- Note that this naive softmax implementation (softmax_naive) may encounter numerical instability problems, such as overflow and underflow, when dealing with large or small input values.
- PyTorch implementation of softmax

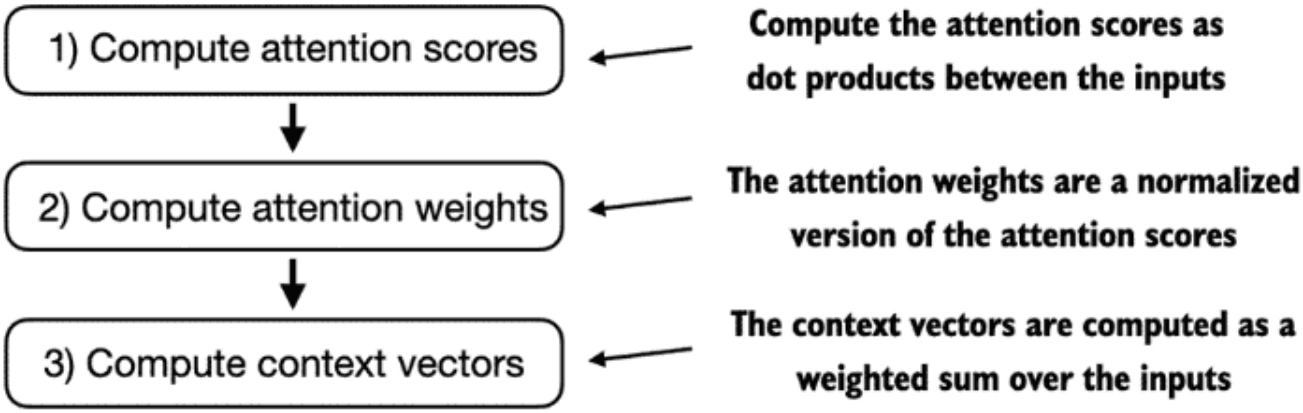
```
attn_weights_2 = torch.softmax(attn_scores_2, dim=0)
print("Attention weights:", attn_weights_2)
print("Sum:", attn_weights_2.sum())
# Output
# Attention weights: tensor([0.1385, 0.2379, 0.2333, 0.1240, 0.1082, 0.1581])
# Sum: tensor(1.)
```




```
query = inputs[1] # 2nd input token is the query
context_vec_2 = torch.zeros(query.shape)
for i,x_i in enumerate(inputs):
    context_vec_2 += attn_weights_2[i]*x_i
print(context_vec_2)
# Output
# tensor([0.4419, 0.6515, 0.5683])
```

3.3.2 Computing attention weights for all input tokens



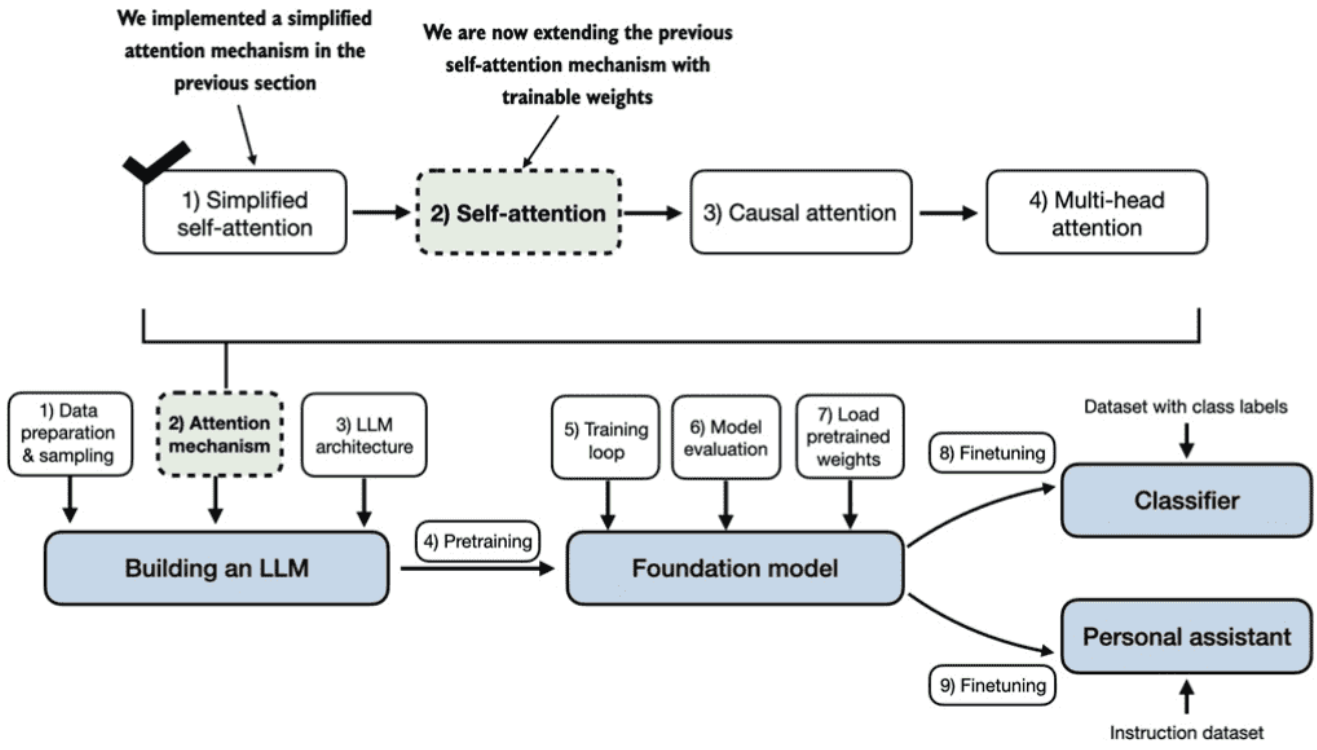



```
import torch
inputs = torch.tensor(
  [[0.43, 0.15, 0.89], # Your     (x^1)
    [0.55, 0.87, 0.66], # journey  (x^2)
    [0.57, 0.85, 0.64], # starts    (x^3)
    [0.22, 0.58, 0.33], # with      (x^4)
    [0.77, 0.25, 0.10], # one       (x^5)
    [0.05, 0.80, 0.55]] # step      (x^6)
)

attn_scores = inputs @ inputs.T
print(attn_scores)
# Output
# tensor([[0.9995, 0.9544, 0.9422, 0.4753, 0.4576, 0.6310],
```

```
#     [0.9544, 1.4950, 1.4754, 0.8434, 0.7070, 1.0865],
#     [0.9422, 1.4754, 1.4570, 0.8296, 0.7154, 1.0605],
#     [0.4753, 0.8434, 0.8296, 0.4937, 0.3474, 0.6565],
#     [0.4576, 0.7070, 0.7154, 0.3474, 0.6654, 0.2935],
#     [0.6310, 1.0865, 1.0605, 0.6565, 0.2935, 0.9450]])

attn_weights = torch.softmax(attn_scores, dim=1)
print(attn_weights)
# Output
# tensor([[0.2098, 0.2006, 0.1981, 0.1242, 0.1220, 0.1452],
#     [0.1385, 0.2379, 0.2333, 0.1240, 0.1082, 0.1581],
#     [0.1390, 0.2369, 0.2326, 0.1242, 0.1108, 0.1565],
#     [0.1435, 0.2074, 0.2046, 0.1462, 0.1263, 0.1720],
#     [0.1526, 0.1958, 0.1975, 0.1367, 0.1879, 0.1295],
#     [0.1385, 0.2184, 0.2128, 0.1420, 0.0988, 0.1896]])

all_context_vecs = attn_weights @ inputs
print(all_context_vecs)
# Output
# tensor([[0.4421, 0.5931, 0.5790],
#     [0.4419, 0.6515, 0.5683],
#     [0.4431, 0.6496, 0.5671],
#     [0.4304, 0.6298, 0.5510],
#     [0.4671, 0.5910, 0.5266],
#     [0.4177, 0.6503, 0.5645]])
```

3.4 Implementing self-attention with trainable weights

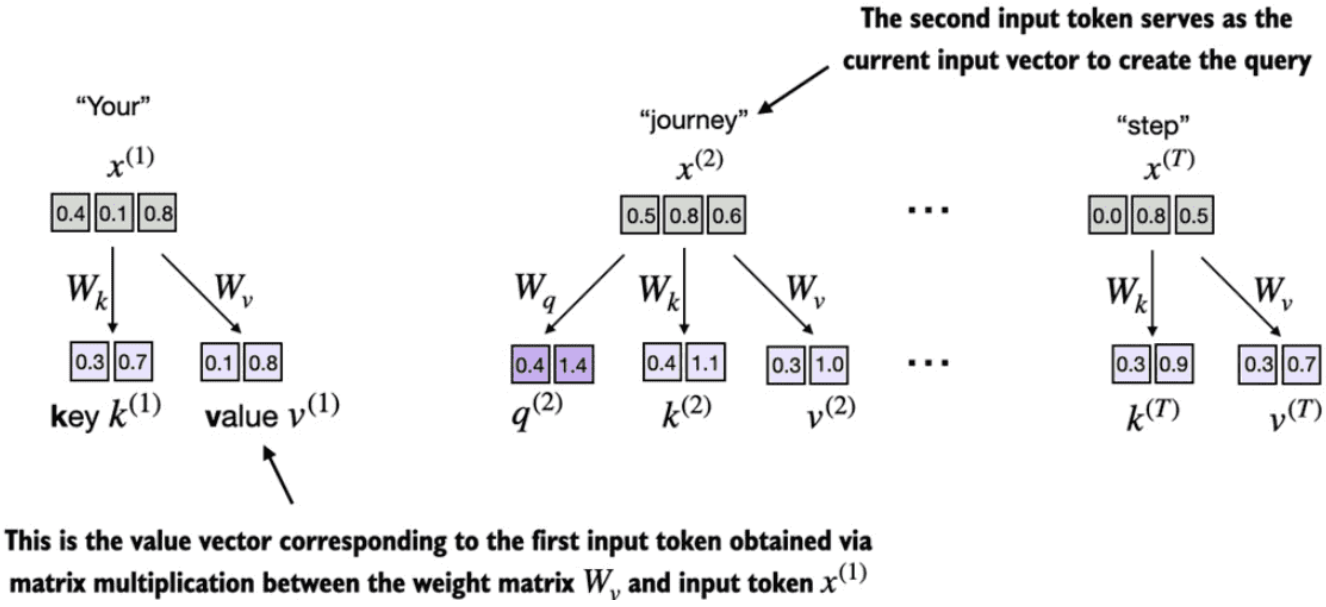
- Self-attention mechanism is also called `scaled dot-product attention`.



- This section's main work is to add the functionality of **updating weights during training** on top of the previous sections, in order to learn better weights during training (优化).

3.4.1 Computing the attention weights step by step

- Three trainable weight matrices W_q , W_k , and W_v.
- These three matrices are used to project the embedded input tokens, $x^{(i)}$, into query, key, and value vectors



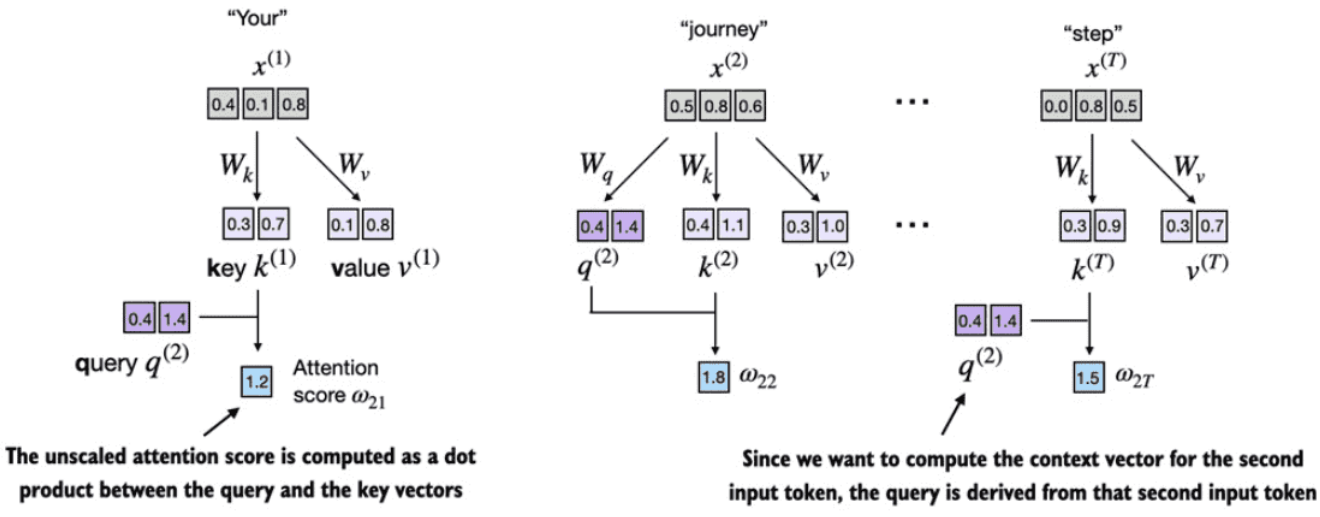
- Note that in GPT-like models, the input and output dimensions are usually the same, but for illustration purposes, to better follow the computation, we choose different input (d_in=3) and output (d_out=2) dimensions here.

```
import torch
inputs = torch.tensor(
  [[0.43, 0.15, 0.89], # Your     (x^1)
    [0.55, 0.87, 0.66], # journey  (x^2)
    [0.57, 0.85, 0.64], # starts    (x^3)
    [0.22, 0.58, 0.33], # with      (x^4)
    [0.77, 0.25, 0.10], # one       (x^5)
    [0.05, 0.80, 0.55]] # step      (x^6)
)

x_2 = inputs[1] #A
d_in = inputs.shape[1] #B  = 3
d_out = 2 #C

# Initialize the three weight matrices Wq, Wk, and Wv
torch.manual_seed(123)
W_query = torch.nn.Parameter(torch.rand(d_in, d_out), requires_grad=False)
W_key   = torch.nn.Parameter(torch.rand(d_in, d_out), requires_grad=False)
W_value = torch.nn.Parameter(torch.rand(d_in, d_out), requires_grad=False)
# Example structure: shape=(3,2)
# tensor([[0.3821, 0.6605],
#         [0.8536, 0.5932],
#         [0.6367, 0.9826]])
# Explanation
# setting requires_grad=False to reduce clutter in the outputs for illustration
```

```
purposes
# requires_grad=True should be used for formal training

# compute the query, key, and value vectors
query_2 = x_2 @ W_query
key_2 = x_2 @ W_key
value_2 = x_2 @ W_value
print(query_2)
# Output
# tensor([0.4306, 1.4551])
```

- 【Note】Do not confuse weight parameters with attention weights.
- Weight parameters: Refers to the weights of the neural network that are optimized during training, sometimes abbreviated as weight. ("weight" is short for "weight parameters," the values of a neural network that are optimized during training.)
- Attention weights: Determine the extent to which a context vector depends on the different parts of the input, i.e., to what extent the network focuses on different parts of the input. (attention weights determine the extent to which a context vector depends on the different parts of the input, i.e., to what extent the network focuses on different parts of the input.)
- In short, weight parameters are the fundamental learned coefficients that define the network's connections, while attention weights are dynamic, context-specific values.

```
# Although only calculating the context vector Z^(2) here, it is still necessary
to have key and value vectors for all input elements.
# obtain all keys and values via matrix multiplication
keys = inputs @ W_key
values = inputs @ W_value
print("keys.shape:", keys.shape)
# Output
# keys.shape: torch.Size([6, 2])
```

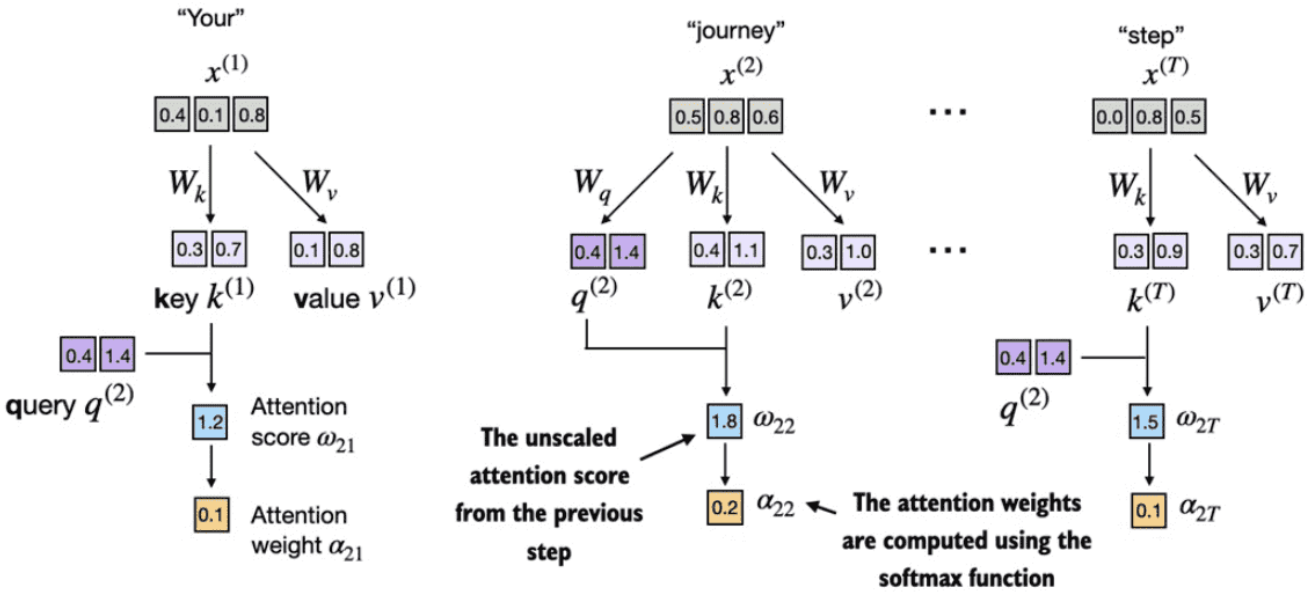


- Compute the `attention score` ω_{22}

```
keys_2 = keys[1] #A
attn_score_22 = query_2.dot(keys_2)
print(attn_score_22)
# tensor(1.8524)
```

- Similarly, compute all other ω_2 (i.e., ω_{21} to ω_{2T})

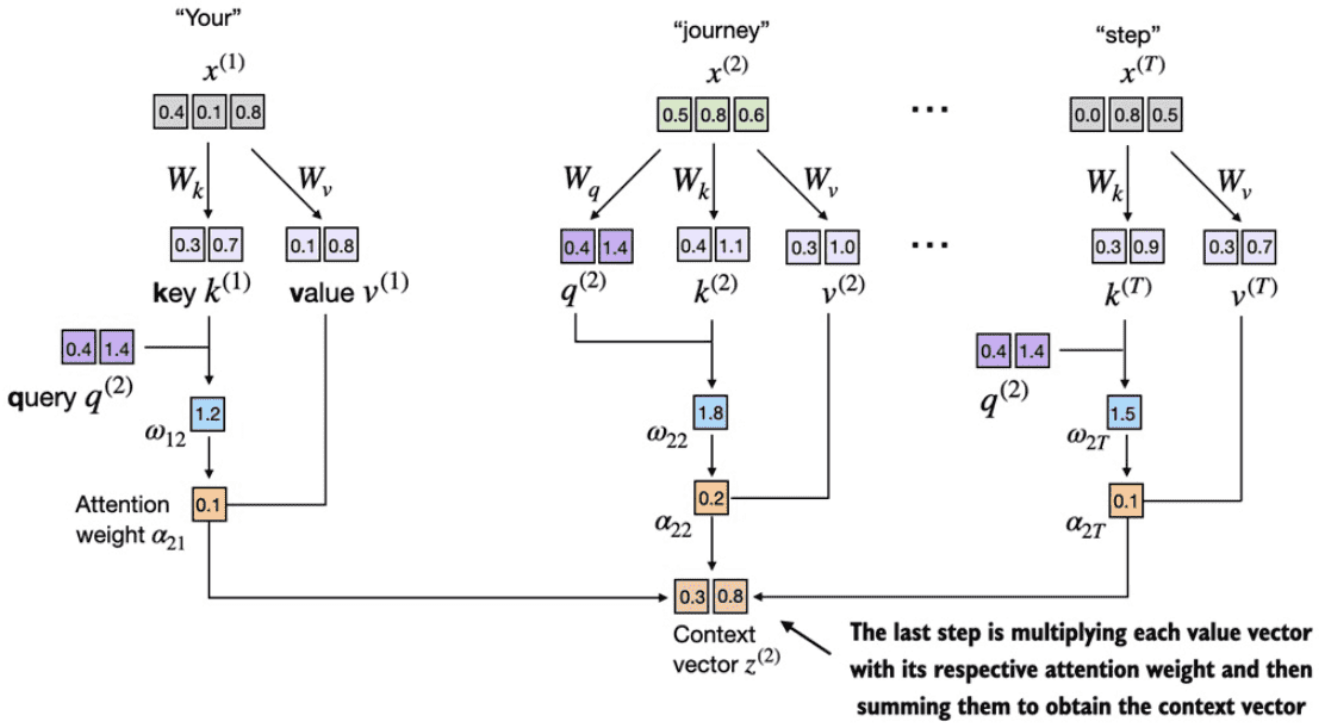
```
attn_scores_2 = query_2 @ keys.T    # All attention scores for given query
print(attn_scores_2)
# Output
# tensor([1.2705, 1.8524, 1.8111, 1.0795, 0.5577, 1.5440])
```




```
d_k = keys.shape[-1]
attn_weights_2 = torch.softmax(attn_scores_2 / d_k**0.5, dim=-1)
print(attn_weights_2)
# Output
# tensor([0.1500, 0.2264, 0.2199, 0.1311, 0.0906, 0.1820])
```

- Scale the attention scores by dividing them by the square root of the key embedding dimension ($d_k^{0.5}$).
- When scaling up the embedding dimension, which is typically greater than a thousand (as in GPT-4 and other models), large dot products can result in very small gradients during backpropagation due to the softmax function applied to them.
- As dot products increase, the softmax function behaves more like a step function, resulting in gradients nearing zero.
- These small gradients can drastically slow down learning or cause training to stagnate.
- The scaling by the `square root` of the embedding dimension is the reason why this self-attention mechanism is also called `scaled-dot product attention`.

- 【Key point】Why is the dot product scaled by the square root of the embedding dimension in the self-attention mechanism?
- Purpose of scaling: Scaling by the square root of the embedding dimension is to avoid excessively small gradients during training. Without scaling, training can encounter situations with very small gradients, causing the model's learning to slow down or even stagnate.
- Reason for small gradients: 1. As the embedding dimension (i.e., the dimension of the vectors) increases, the dot product of two vectors becomes larger. In large language models (LLMs) like GPT, embedding dimensions are often high, possibly reaching thousands, leading to very large dot products. 2. When the softmax function is applied to large dot product results, the output probability distribution becomes very sharp, approximating a step function. In this case, most of the probability is concentrated on a few values, causing the gradients for other parts to be almost zero. This leads to insufficient updates during model training.
- Effect of scaling: By scaling the magnitude of the dot product with the square root of the embedding dimension, the dot product values are kept within a reasonable range, making the softmax function's output smoother. This results in larger gradients, allowing the model to learn more effectively. This scaled self-attention mechanism is therefore called "scaled-dot product attention".




```
context_vec_2 = attn_weights_2 @ values
print(context_vec_2)
# Output
# tensor([0.3061, 0.8210])
```

> **Note:** [Why are query, key, and value needed?] These three concepts are borrowed from information retrieval and database fields. Combining them helps in understanding these concepts.
>
> - Query: "query" represents the item the model is currently focusing on (e.g., a word or token in a sentence). The model uses "query" to probe other parts and determine how much attention is needed.

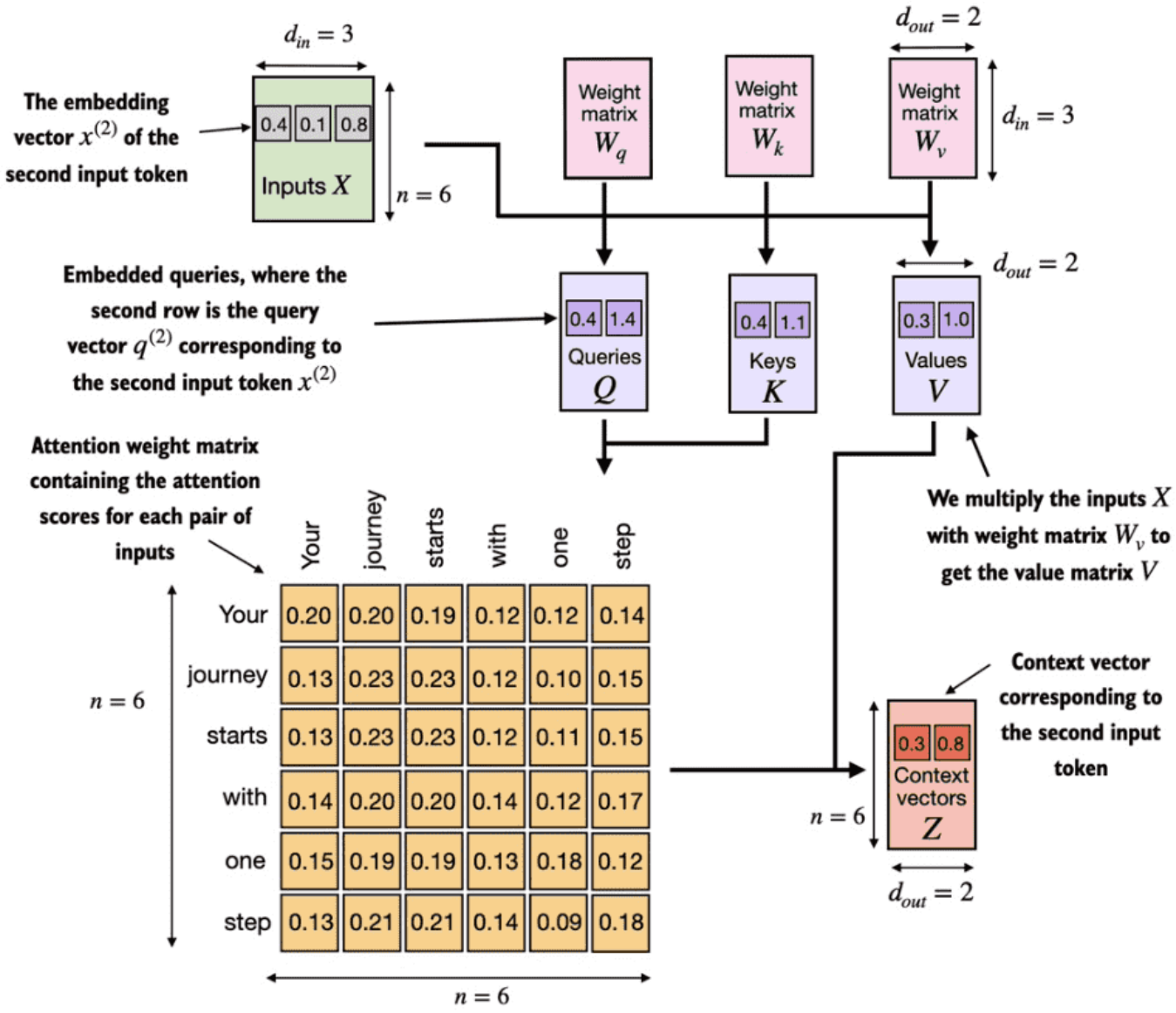
> - Key: Each item in the input sequence (e.g., each word in a sentence) has a corresponding "key". These "keys" are matched with the "query". Through this matching, the model identifies which "keys" (i.e., which parts of the input) are more relevant to the "query".
> - Value: Once the model identifies the "keys" most relevant to the "query", it retrieves the corresponding "value". These "values" contain the actual content or representation of the input items. By extracting these "values", the model gets the specific content that the current "query" should focus on. In summary, weight parameters are the basic learned coefficients that define the network's connections, while attention weights are dynamic, context-specific values. **Note:** Why can't Wv be directly represented by input? Because Wv directly determines the extracted content and the final feature information. It can capture more complex features, especially in context-dependent and long-sequence modeling, and can better identify subtle differences in features at different positions. Directly using the input means the model lacks this distinction between content and attention distribution, making it difficult for features from different positions to be effectively weighted and combined, thus reducing the expressive power of the attention mechanism.

3.4.2 Implementing a compact self-attention Python class

- Listing 3.1 A compact self-attention class

```
import torch.nn as nn
class SelfAttention_v1(nn.Module):
    def __init__(self, d_in, d_out):
        super().__init__()
        self.d_out = d_out
        self.W_query = nn.Parameter(torch.rand(d_in, d_out))
        self.W_key   = nn.Parameter(torch.rand(d_in, d_out))
        self.W_value = nn.Parameter(torch.rand(d_in, d_out))
    def forward(self, x):
        keys = x @ self.W_key
        queries = x @ self.W_query
        values = x @ self.W_value

        attn_scores = queries @ keys.T # omega

        attn_weights = torch.softmax(
            attn_scores / keys.shape[-1]**0.5, dim=-1)

        context_vec = attn_weights @ values
        return context_vec

torch.manual_seed(123)
sa_v1 = SelfAttention_v1(d_in, d_out)
print(sa_v1(inputs))
# Output
# tensor([[0.2996, 0.8053],
#         [0.3061, 0.8210],
#         [0.3058, 0.8203],
#         [0.2948, 0.7939],
```

```
#         [0.2927, 0.7891],
#         [0.2990, 0.8040]], grad_fn=<MmBackward0>)
```



- Using `nn.Linear` instead of manually implementing `nn.Parameter(torch.rand(...))` is that `nn.Linear` has an optimized weight initialization scheme, contributing to more stable and effective model training.
- Listing 3.2 A self-attention class using PyTorch's Linear layers

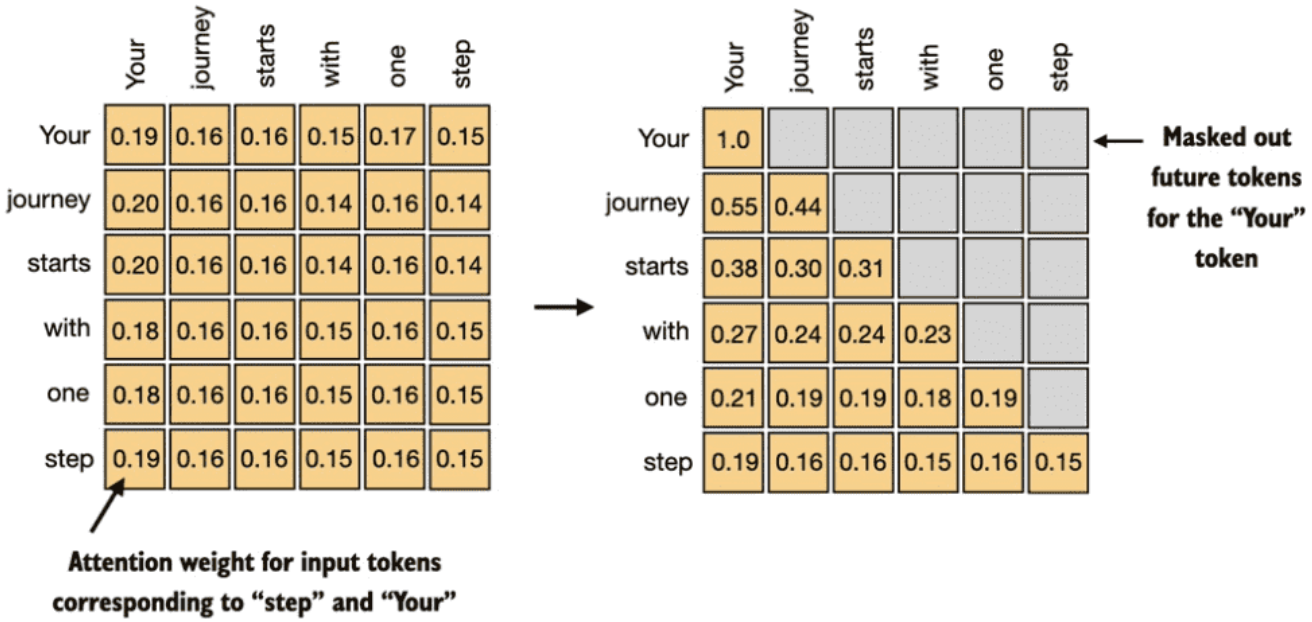
```
class SelfAttention_v2(nn.Module):
    def __init__(self, d_in, d_out, qkv_bias=False):
        super().__init__()
        self.d_out = d_out
        self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_key   = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_value = nn.Linear(d_in, d_out, bias=qkv_bias)
    def forward(self, x):
        keys = self.W_key(x)
        queries = self.W_query(x)
        values = self.W_value(x)
```

```
        attn_scores = queries @ keys.T

        attn_weights = torch.softmax(attn_scores / keys.shape[-1]**0.5, dim)

        context_vec = attn_weights @ values
        return context_vec

torch.manual_seed(789)
sa_v2 = SelfAttention_v2(d_in, d_out)
print(sa_v2(inputs))
# Output
# tensor([[-0.0739,  0.0713],
#         [-0.0748,  0.0703],
#         [-0.0749,  0.0702],
#         [-0.0760,  0.0685],
#         [-0.0763,  0.0679],
#         [-0.0754,  0.0693]], grad_fn=<MmBackward0>)
```

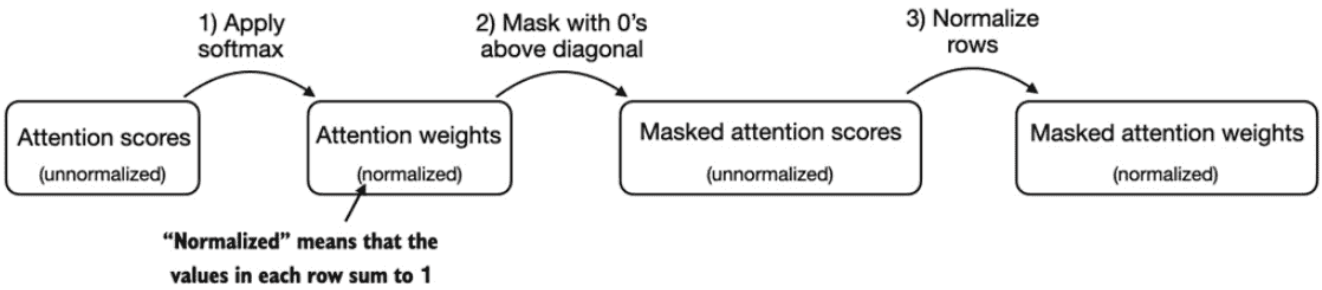
- Note that SelfAttention_v1 and SelfAttention_v2 give different outputs because they use different initial weights for the weight matrices since nn.Linear uses a more sophisticated weight initialization scheme.

3.5 Hiding future words with causal attention

- `Causal attention`, also known as `masked attention`, is a specialized form of self-attention. It restricts a model to only consider previous and current inputs in a sequence when processing any given token. This is in contrast to the standard self-attention mechanism, which allows access to the entire input sequence at once.



3.5.1 Applying a causal attention mask



- [Step 1] Obtain the attention weights using the method from the previous section.

```
print(attn_weights)
tensor([[0.1921, 0.1646, 0.1652, 0.1550, 0.1721, 0.1510],
        [0.2041, 0.1659, 0.1662, 0.1496, 0.1665, 0.1477],
        [0.2036, 0.1659, 0.1662, 0.1498, 0.1664, 0.1480],
        [0.1869, 0.1667, 0.1668, 0.1571, 0.1661, 0.1564],
        [0.1830, 0.1669, 0.1670, 0.1588, 0.1658, 0.1585],
        [0.1935, 0.1663, 0.1666, 0.1542, 0.1666, 0.1529]],
grad_fn=<SoftmaxBackward0>)
```

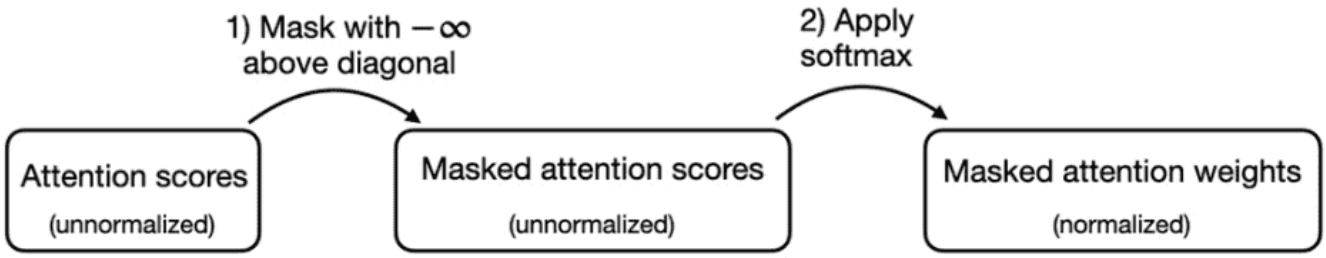
- [Step 2] Set the values above the diagonal to 0.

```
context_length = attn_scores.shape[0]
mask_simple = torch.tril(torch.ones(context_length, context_length))
print(mask_simple)
# Output
# tensor([[1., 0., 0., 0., 0., 0.],
#         [1., 1., 0., 0., 0., 0.],
#         [1., 1., 1., 0., 0., 0.],
#         [1., 1., 1., 1., 0., 0.],
#         [1., 1., 1., 1., 1., 0.],
#         [1., 1., 1., 1., 1., 1.]])

# multiply this mask with the attention weights
masked_simple = attn_weights*mask_simple
print(masked_simple)
# Output
# tensor([[0.1921, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
#         [0.2041, 0.1659, 0.0000, 0.0000, 0.0000, 0.0000],
#         [0.2036, 0.1659, 0.1662, 0.0000, 0.0000, 0.0000],
#         [0.1869, 0.1667, 0.1668, 0.1571, 0.0000, 0.0000],
#         [0.1830, 0.1669, 0.1670, 0.1588, 0.1658, 0.0000],
#         [0.1935, 0.1663, 0.1666, 0.1542, 0.1666, 0.1529]],
#        grad_fn=<MulBackward0>)
```

- [Step 3] Renormalize the attention weights so that the sum of values in each row is 1.

```
row_sums = masked_simple.sum(dim=1, keepdim=True)
masked_simple_norm = masked_simple / row_sums
print(masked_simple_norm)
# Output
# tensor([[1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
#         [0.5517, 0.4483, 0.0000, 0.0000, 0.0000, 0.0000],
#         [0.3800, 0.3097, 0.3103, 0.0000, 0.0000, 0.0000],
#         [0.2758, 0.2460, 0.2462, 0.2319, 0.0000, 0.0000],
#         [0.2175, 0.1983, 0.1984, 0.1888, 0.1971, 0.0000],
#         [0.1935, 0.1663, 0.1666, 0.1542, 0.1666, 0.1529]],
#        grad_fn=<DivBackward0>)
```

- [Note - Information Leakage] When we apply the mask and then renormalize the attention weights, information originally from future tokens (now masked) could still potentially influence the current token because their values were part of the initial softmax calculation. However, the key insight is that when we renormalize the attention weights after masking, we are essentially re-calculating the softmax over a smaller subset (since masked positions contribute zero to the softmax values). The mathematical elegance of softmax is that despite initially including all positions in the denominator, after masking and re-normalization, the influence of the masked positions is canceled out—they don't contribute meaningfully to the softmax scores. So there is no information leakage.



- Here, we can leverage the mathematical properties of the softmax function to implement the calculation of masked attention weights more efficiently with fewer steps.
- 【Key Feature】When negative infinity values (-∞) are present, the softmax function treats them as zero probabilities (mathematically, because $e^{-\infty}$ approaches 0).
- Here, we replace the 0s above with -∞ (i.e., -inf).

```
mask = torch.triu(torch.ones(context_length, context_length), diagonal=1)
masked = attn_scores.masked_fill(mask.bool(), -torch.inf)
print(masked)
# Output
# tensor([[0.2899,   -inf,   -inf,   -inf,   -inf,   -inf],
#         [0.4656, 0.1723,   -inf,   -inf,   -inf,   -inf],
#         [0.4594, 0.1703, 0.1731,   -inf,   -inf,   -inf],
#         [0.2642, 0.1024, 0.1036, 0.0186,   -inf,   -inf],
#         [0.2183, 0.0874, 0.0882, 0.0177, 0.0786,   -inf],
#         [0.3408, 0.1270, 0.1290, 0.0198, 0.1290, 0.0078]],
#        grad_fn=<MaskedFillBackward0>)
```

```
attn_weights = torch.softmax(masked / keys.shape[-1]**0.5, dim=1)
print(attn_weights)
# Output (already normalized, no extra operation needed, saves operations)
# tensor([[1.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
#         [0.5517, 0.4483, 0.0000, 0.0000, 0.0000, 0.0000],
#         [0.3800, 0.3097, 0.3103, 0.0000, 0.0000, 0.0000],
#         [0.2758, 0.2460, 0.2462, 0.2319, 0.0000, 0.0000],
#         [0.2175, 0.1983, 0.1984, 0.1888, 0.1971, 0.0000],
#         [0.1935, 0.1663, 0.1666, 0.1542, 0.1666, 0.1529]],
#        grad_fn=<SoftmaxBackward0>)
```

3.5.2 Masking additional attention weights with dropout

- Dropout in deep learning is a technique where randomly selected hidden layer units are ignored during training, effectively "dropping" them out. This method helps prevent overfitting by ensuring that a model does not become overly reliant on any specific set of hidden layer units. It's important to emphasize that dropout is only used during training and is disabled afterward.

> **Note:** In the transformer architecture, including models like GPT, dropout in the attention mechanism is typically applied in two specific areas: `after calculating the attention scores` or `after applying the attention weights to the value vectors`. (I haven't fully grasped the exact positions yet). My understanding (based on GPT's conflicting statements, the following is my understanding): [Dropout Application Positions] 1. after calculating the attention scores: Applying dropout after calculating the attention scores means performing the dropout operation after obtaining the dot products between Q (Query), K (Key), and V (Value) vectors, but before passing these scores to the Softmax function. Dropout in this step helps the model avoid over-reliance on certain specific features, as it may randomly invalidate some attention scores, prompting the model to learn a more robust attention distribution. Pseudocode:

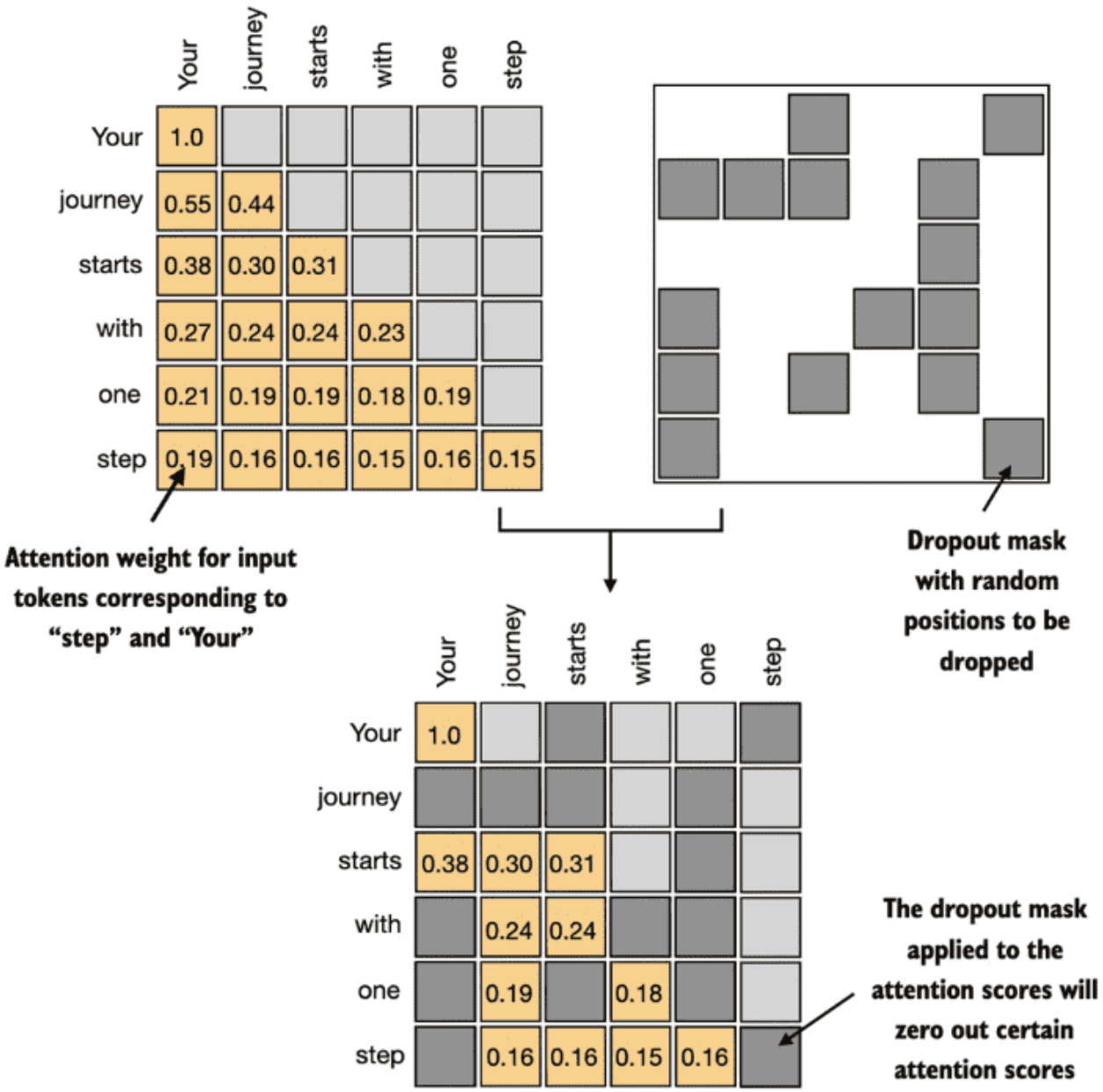
```
function scaledDotProductAttention(Q, K, V, dropoutRate):
    # Calculate attention scores
    scores = matmul(Q, transpose(K)) / sqrt(d_k)  # d_k is the dimension of key
vectors

    # Apply Dropout
    scores = applyDropout(scores, dropoutRate)

    # Apply Softmax function
    attentionWeights = softmax(scores)

    # Apply attention weights to value vectors
    output = matmul(attentionWeights, V)

    return output
```

[Dropout Application Positions] 2. after applying the attention weights to the value vectors: Applying dropout after applying the attention weights to the value vectors means performing the dropout operation after multiplying the Softmax-normalized attention weights by the value vectors to obtain the weighted value

vectors. Doing so further helps the model generalize, because even if some information is randomly dropped out by dropout, the model still needs to be able to use the remaining information to make accurate predictions. Pseudocode:

```
function scaledDotProductAttention(Q, K, V, dropoutRate):
    # Calculate attention scores
    scores = matmul(Q, transpose(K)) / sqrt(d_k)  # d_k is the dimension of key
vectors

    # Apply Softmax function
    attentionWeights = softmax(scores)

    # Apply Dropout
    attentionWeights = applyDropout(attentionWeights, dropoutRate)

    # Apply attention weights to value vectors
    output = matmul(attentionWeights, V)

    return output
```

> **Note:** The following example demonstrates "apply the dropout mask after computing the attention weights".



- In the example below, we use a dropout rate of 50%. When training the GPT model later, we will use a dropout rate of 10%-20%.

```
torch.manual_seed(123)
dropout = torch.nn.Dropout(0.5) #A
example = torch.ones(6, 6) #B
print(dropout(example))
# Output (nearly half are 0)
# tensor([[2., 2., 0., 2., 2., 0.],
#         [0., 0., 0., 2., 0., 2.],
#         [2., 2., 2., 2., 0., 2.],
#         [0., 2., 2., 0., 0., 2.],
#         [0., 2., 0., 2., 0., 2.],
#         [0., 2., 2., 2., 2., 0.]])
```

Apply dropout to the attention weight matrix:

```
torch.manual_seed(123)
print(dropout(attn_weights))
# Output
# tensor([[2.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
#         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000],
#         [0.7599, 0.6194, 0.6206, 0.0000, 0.0000, 0.0000],
#         [0.0000, 0.4921, 0.4925, 0.0000, 0.0000, 0.0000],
#         [0.0000, 0.3966, 0.0000, 0.3775, 0.0000, 0.0000],
#         [0.0000, 0.3327, 0.3331, 0.3084, 0.3331, 0.0000]],
#        grad_fn=<MulBackward0>
```

3.5.3 Implementing a compact causal attention class

- Listing 3.3 A compact causal attention class

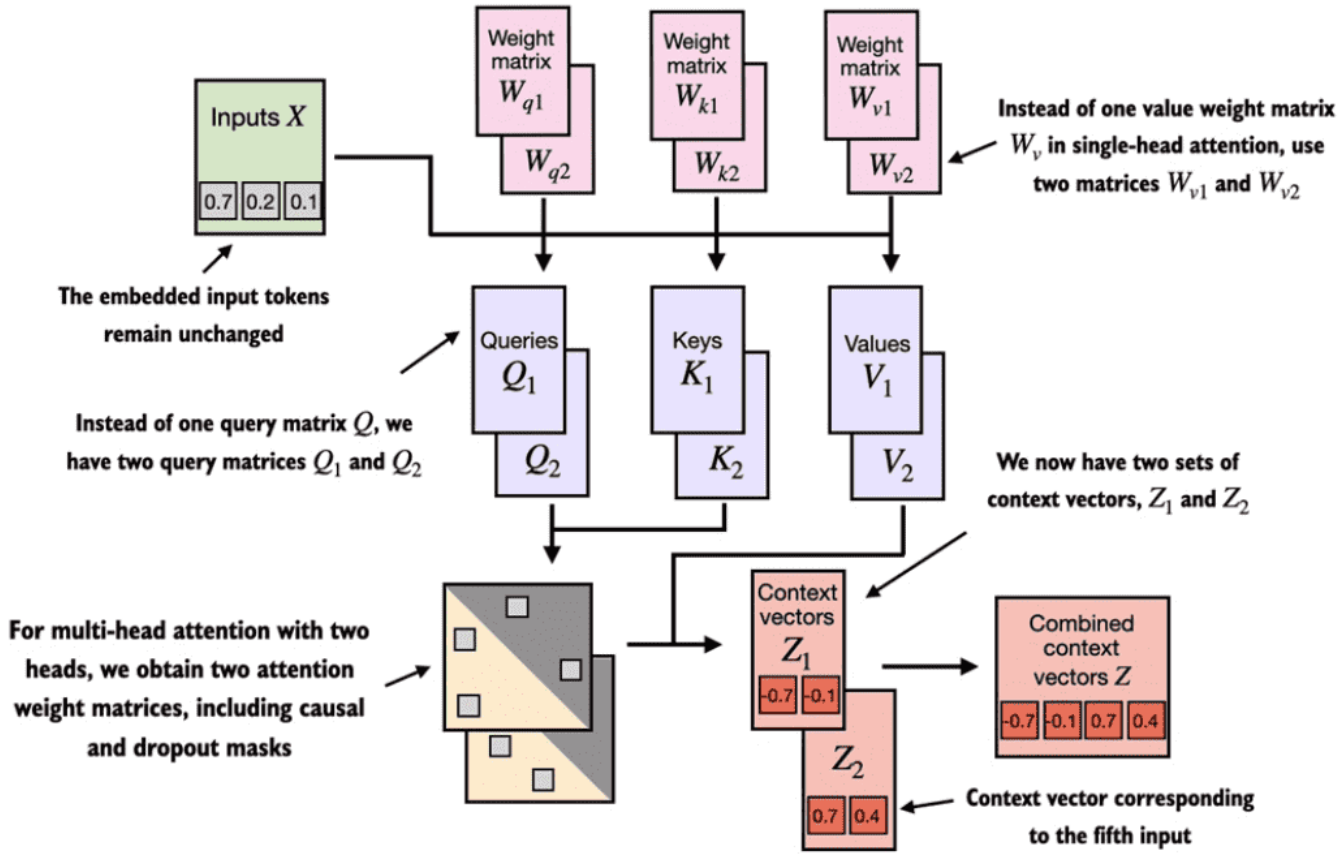
```
import torch.nn as nn
class CausalAttention(nn.Module):

    def __init__(self, d_in, d_out, context_length,
                 dropout, qkv_bias=False):
        super().__init__()
        self.d_out = d_out
        self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_key   = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_value = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.dropout = nn.Dropout(dropout) # A
        self.register_buffer(
            'mask',
            torch.triu(
                torch.ones(context_length, context_length),
                diagonal=1
            )
        ) #B


    def forward(self, x):
        b, num_tokens, d_in = x.shape  #C
        keys = self.W_key(x)
        queries = self.W_query(x)
        values = self.W_value(x)

        attn_scores = queries @ keys.transpose(1, 2)  #C
        attn_scores.masked_fill_(  #D
            self.mask.bool()[:num_tokens, :num_tokens], -torch.inf)
        attn_weights = torch.softmax(attn_scores / keys.shape[-1]**0.5, dim=-1)
        attn_weights = self.dropout(attn_weights)

        context_vec = attn_weights @ values
        return context_vec
```

```
import torch
inputs = torch.tensor(
  [[0.43, 0.15, 0.89], # Your     (x^1)
   [0.55, 0.87, 0.66], # journey  (x^2)
   [0.57, 0.85, 0.64], # starts   (x^3)
   [0.22, 0.58, 0.33], # with     (x^4)
   [0.77, 0.25, 0.10], # one      (x^5)
   [0.05, 0.80, 0.55]] # step     (x^6)
)
batch = torch.stack((inputs, inputs), dim=0)

torch.manual_seed(123)
context_length = batch.shape[1] # This is the number of tokens
d_in, d_out = 3, 2
ca = CausalAttention(d_in, d_out, context_length, 0.0)
context_vecs = ca(batch)
print("context_vecs.shape:", context_vecs.shape)
# Output
# context_vecs.shape: torch.Size([2, 6, 2])
```

 Figure 3.23 A mental model summarizing the four different attention modules we are coding in this chapter. We began with a simplified attention mechanism, added trainable weights, and then added a casual attention mask. In the remainder of this chapter, we will extend the causal attention mechanism and code multi-head attention, which is the final module we will use in the LLM implementation in the next chapter.

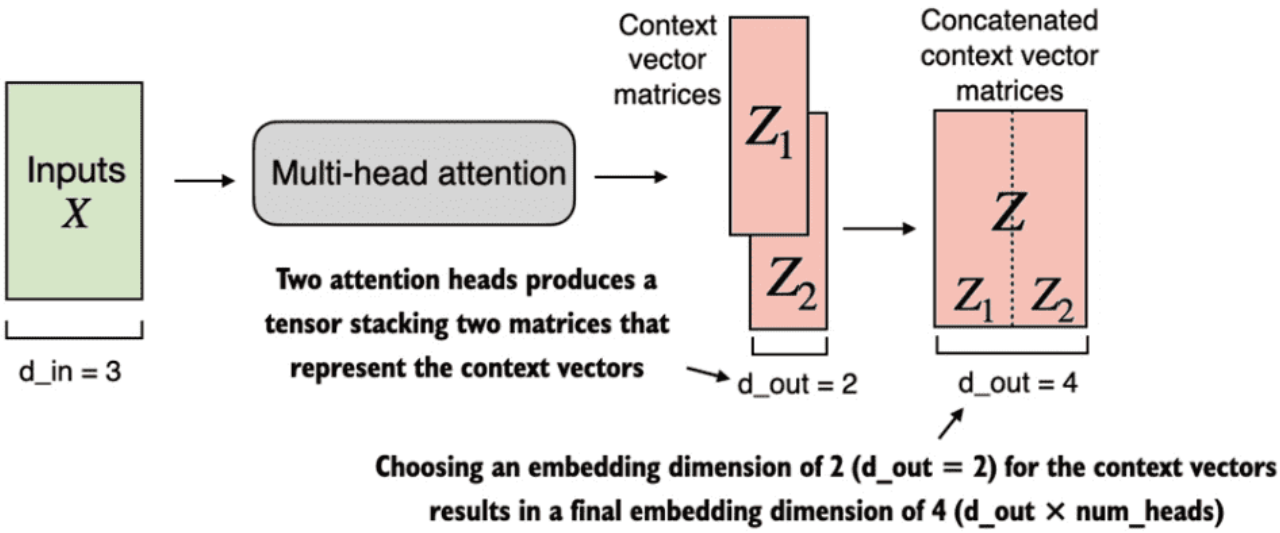
3.6 Extending single-head attention to multi-head attention

3.6.1 Stacking multiple single-head attention layers



- Listing 3.4 A wrapper class to implement multi-head attention

```
class MultiHeadAttentionWrapper(nn.Module):
    def __init__(self, d_in, d_out, context_length, dropout, num_heads,
qkv_bias=False):
        super().__init__()
        self.heads = nn.ModuleList(
            [CausalAttention(d_in, d_out, context_length, dropout, qkv_bias)
             for _ in range(num_heads)]
        )

    def forward(self, x):
        return torch.cat([head(x) for head in self.heads], dim=-1)
```

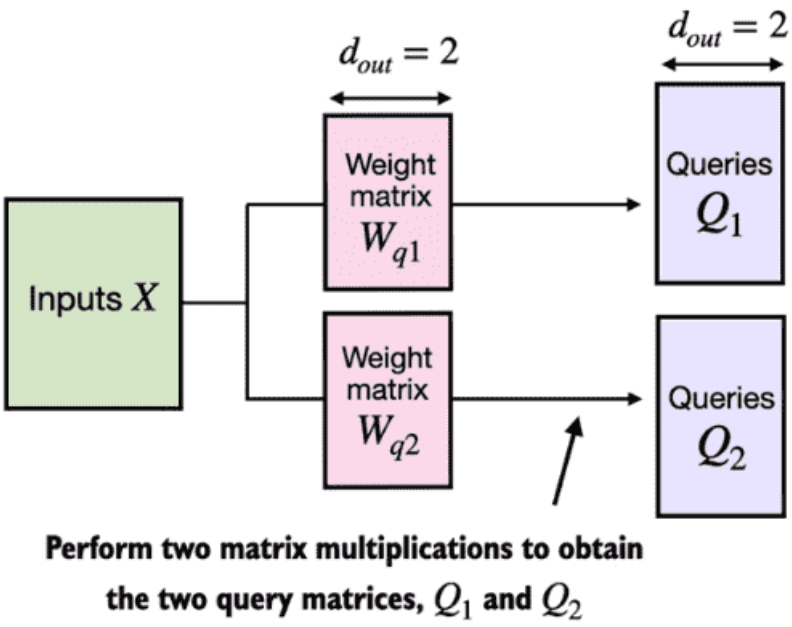



```
torch.manual_seed(123)

context_length = batch.shape[1] # This is the number of tokens
d_in, d_out = 3, 2
mha = MultiHeadAttentionWrapper(
    d_in, d_out, context_length, 0.0, num_heads=2
)

context_vecs = mha(batch)

print("context_vecs.shape:", context_vecs.shape)
#context_vecs.shape: torch.Size([2, 6, 4])
print(context_vecs)
# Output
# tensor([[[-0.4519,  0.2216,  0.4772,  0.1063],
#          [-0.5874,  0.0058,  0.5891,  0.3257],
#          [-0.6300, -0.0632,  0.6202,  0.3860],
#          [-0.5675, -0.0843,  0.5478,  0.3589],
#          [-0.5526, -0.0981,  0.5321,  0.3428],
#          [-0.5299, -0.1081,  0.5077,  0.3493]],

#         [[-0.4519,  0.2216,  0.4772,  0.1063],
#          [-0.5874,  0.0058,  0.5891,  0.3257],
#          [-0.6300, -0.0632,  0.6202,  0.3860],
#          [-0.5675, -0.0843,  0.5478,  0.3589],
#          [-0.5526, -0.0981,  0.5321,  0.3428],
#          [-0.5299, -0.1081,  0.5077,  0.3493]]], grad_fn=<CatBackward0>)

# Explanation
# first  dimension: context_vecs tensor is 2 since we have two input texts
#       (The two values are exactly the same because the input batch is identical)
# second dimension: 6 tokens in each input
# third  dimension: 4-dimensional embedding of each token
```

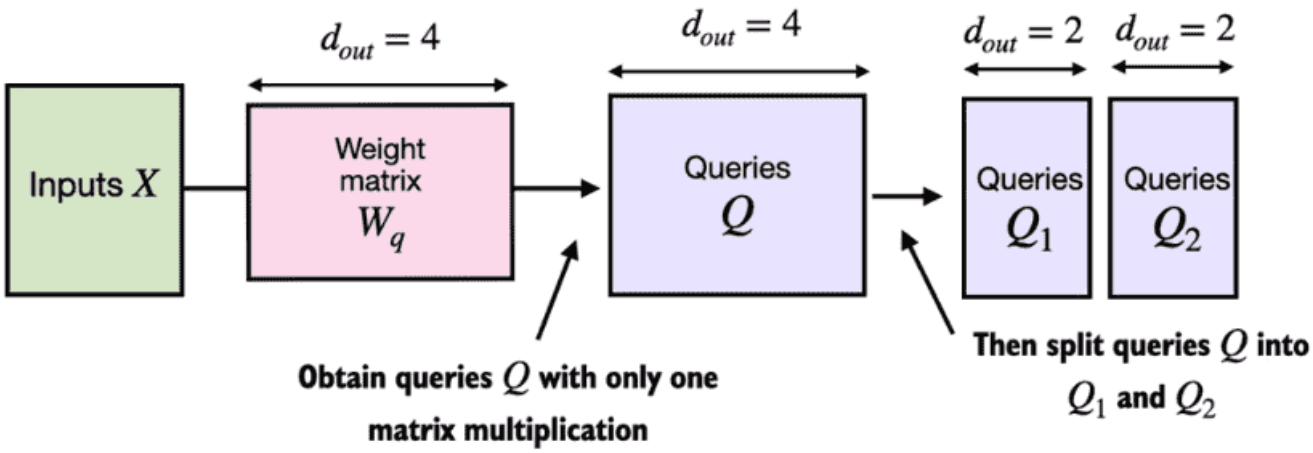
3.6.2 Implementing multi-head attention with weight splits

```
class MultiHeadAttention(nn.Module):
    def __init__(self, d_in, d_out, context_length, dropout, num_heads,
qkv_bias=False):
        super().__init__()
        assert (d_out % num_heads == 0), \
            "d_out must be divisible by num_heads"

        self.d_out = d_out
        self.num_heads = num_heads
        self.head_dim = d_out // num_heads  # Reduce the projection dim to match
desired output dim

        self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_key = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_value = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.out_proj = nn.Linear(d_out, d_out)  # Linear layer to combine head
outputs
        self.dropout = nn.Dropout(dropout)
        self.register_buffer(
            "mask",
            torch.triu(torch.ones(context_length, context_length),
                       diagonal=1)
        )

    def forward(self, x):
        b, num_tokens, d_in = x.shape

        keys = self.W_key(x)    # Shape: (b, num_tokens, d_out) => [2, 6, 2]
        queries = self.W_query(x)
        values = self.W_value(x)

        # We implicitly split the matrix by adding a `num_heads` dimension
        # Unroll last dim: (b, num_tokens, d_out) -> (b, num_tokens, num_heads,
head_dim)
        keys = keys.view(b, num_tokens, self.num_heads, self.head_dim)        # =>
[b, num_tokens, num_headers, head_dim]  => [2, 6, 2, 1]
        values = values.view(b, num_tokens, self.num_heads, self.head_dim)
        queries = queries.view(b, num_tokens, self.num_heads, self.head_dim)

        # Transpose: (b, num_tokens, num_heads, head_dim) -> (b, num_heads,
num_tokens, head_dim)
        keys = keys.transpose(1, 2)             # => [b, num_headers, num_tokens,
head_dim] => [2, 2, 6, 1]
        queries = queries.transpose(1, 2)
        values = values.transpose(1, 2)

        # Compute scaled dot-product attention (aka self-attention) with a causal
mask
        attn_scores = queries @ keys.transpose(2, 3)  # Dot product for each head
# => [2, 2, 6, 1] @ [2, 2, 1, 6] => [2, 2, 6, 6] => [b, num_headers, num_tokens,
num_tokens]
```

```
        # Original mask truncated to the number of tokens and converted to boolean
        mask_bool = self.mask.bool()[:num_tokens, :num_tokens]

        # Use the mask to fill attention scores
        attn_scores.masked_fill_(mask_bool, -torch.inf)

        attn_weights = torch.softmax(attn_scores / keys.shape[-1]**0.5, dim=-1)
        attn_weights = self.dropout(attn_weights)

        # Shape: (b, num_tokens, num_heads, head_dim)
        context_vec = (attn_weights @ values).transpose(1, 2)       # => [2, 2, 6,
6] @ [2, 2, 6, 1] => [2, 2, 6, 1] => [2, 6, 2, 1]

        # Combine heads, where self.d_out = self.num_heads * self.head_dim
        context_vec = context_vec.contiguous().view(b, num_tokens, self.d_out)   #
=> [2, 6, 2]
        context_vec = self.out_proj(context_vec) # optional projection

        return context_vec
```



Perform two matrix multiplications to obtain the two query matrices, Q_1 and Q_2



Obtain queries Q with only one matrix multiplication

Then split queries Q into Q_1 and Q_2

Usage Example:

```
torch.manual_seed(123)
batch_size, context_length, d_in = batch.shape
d_out = 2
mha = MultiHeadAttention(d_in, d_out, context_length, 0.0, num_heads=2)
context_vecs = mha(batch)
print("context_vecs.shape:", context_vecs.shape)
# context_vecs.shape: torch.Size([2, 6, 2])
print(context_vecs)
# tensor([[[0.3190, 0.4858],
#          [0.2943, 0.3897],
#          [0.2856, 0.3593],
#          [0.2693, 0.3873],
#          [0.2639, 0.3928],
#          [0.2575, 0.4028]],
#         [[0.3190, 0.4858],
#          [0.2943, 0.3897],
#          [0.2856, 0.3593],
#          [0.2693, 0.3873],
#          [0.2639, 0.3928],
#          [0.2575, 0.4028]]], grad_fn=<ViewBackward0>)
```

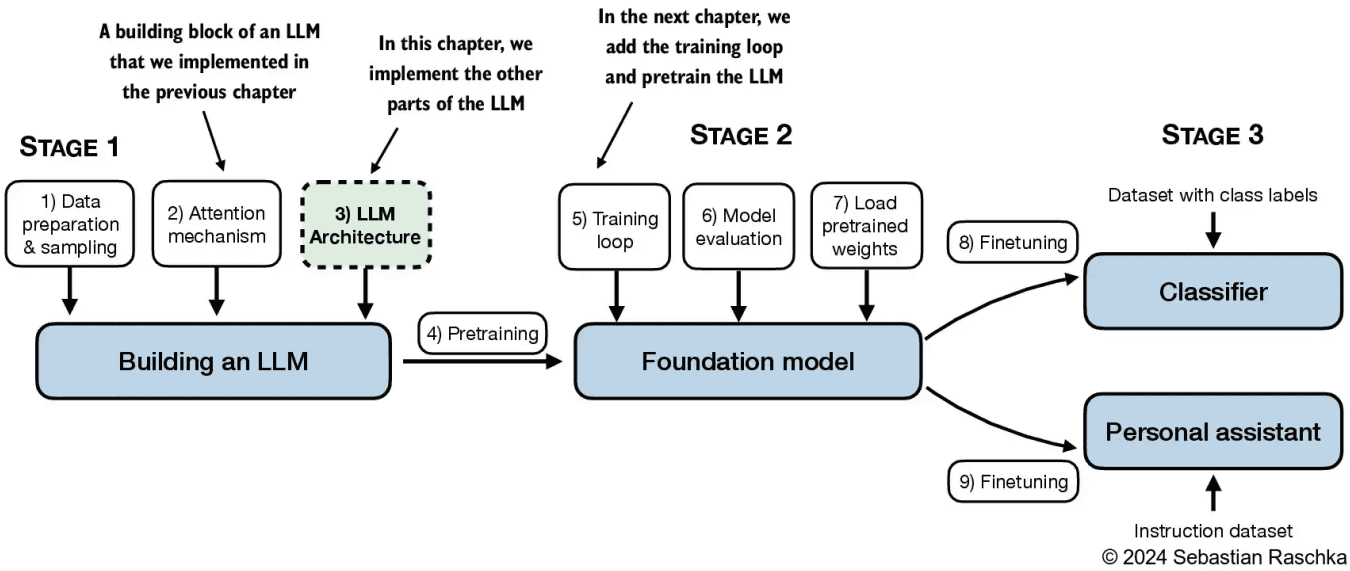
3.7 Summary

- Attention mechanisms transform input elements into enhanced context vector representations that incorporate (包含, 使合并) information about all inputs.
- A self-attention mechanism computes the context vector representation as a weighted sum over the inputs.
- In a simplified attention mechanism, the attention weights are computed via dot products.
- A dot product is just a concise way of multiplying two vectors element-wise and then summing the products.
- Matrix multiplications, while not strictly required, help us to implement computations more efficiently and compactly by replacing nested for-loops.
- In self-attention mechanisms that are used in LLMs, also called scaled-dot product attention, we include trainable weight matrices to compute intermediate transformations of the inputs: queries, values, and keys. When working with LLMs that read and generate text from left to right, we add a causal attention mask to prevent the LLM from accessing future tokens.
- Next to causal attention masks to zero out attention weights, we can also add a dropout mask to reduce overfitting in LLMs.
- The attention modules in transformer-based LLMs involve multiple instances of causal attention, which is called multi-head attention.
- We can create a multi-head attention module by stacking (堆叠) multiple instances of causal attention modules.
- A more efficient way of creating multi-head attention modules involves batched matrix multiplications.

4 Implementing a GPT model from Scratch To Generate Text

- Coding a GPT-like large language model (LLM) that can be trained to generate human-like text
- Normalizing layer activations to stabilize neural network training

- Adding shortcut connections in deep neural networks to train models more effectively
- Implementing transformer blocks to create GPT models of various sizes
- Computing the number of parameters and storage requirements of GPT models

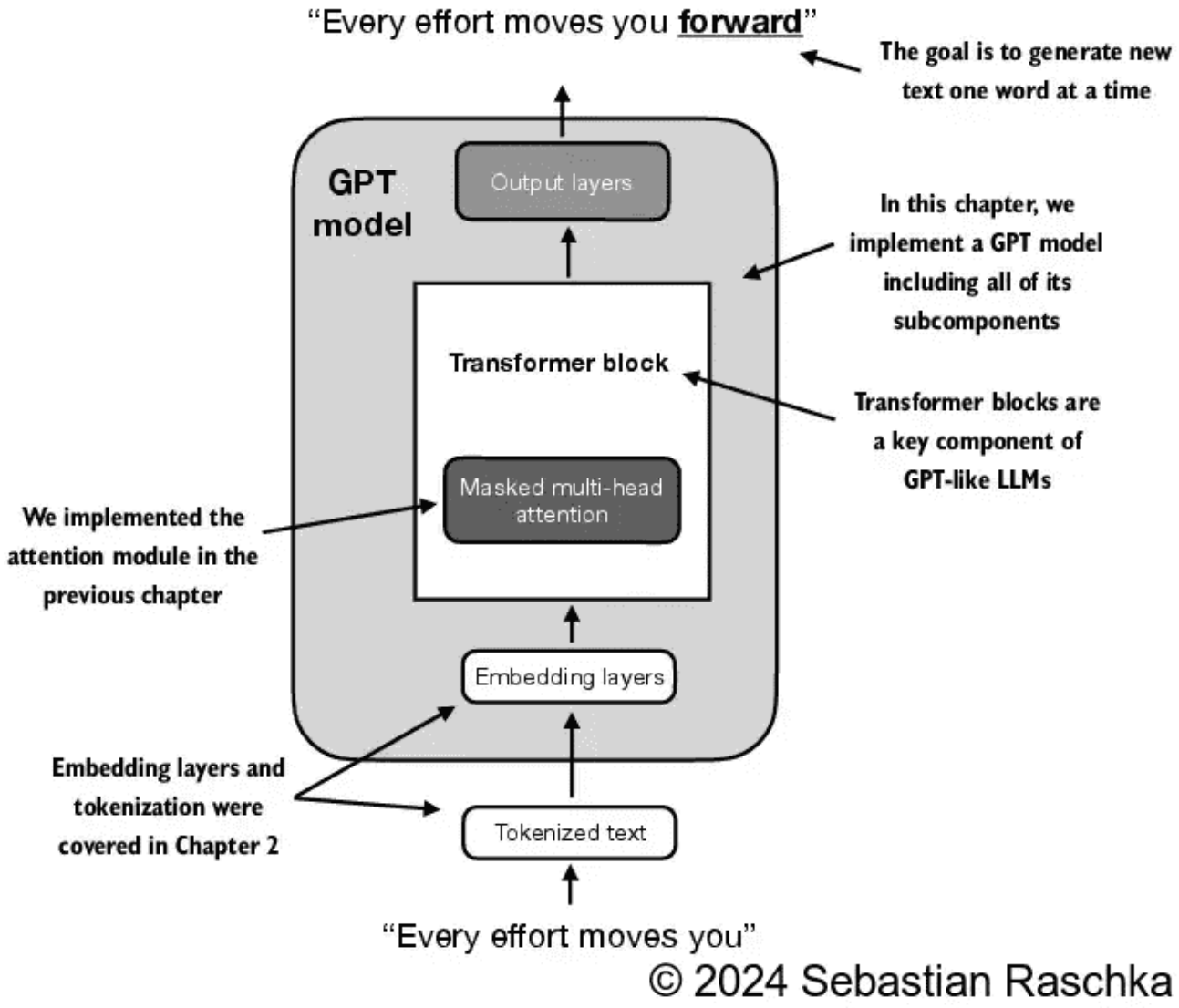


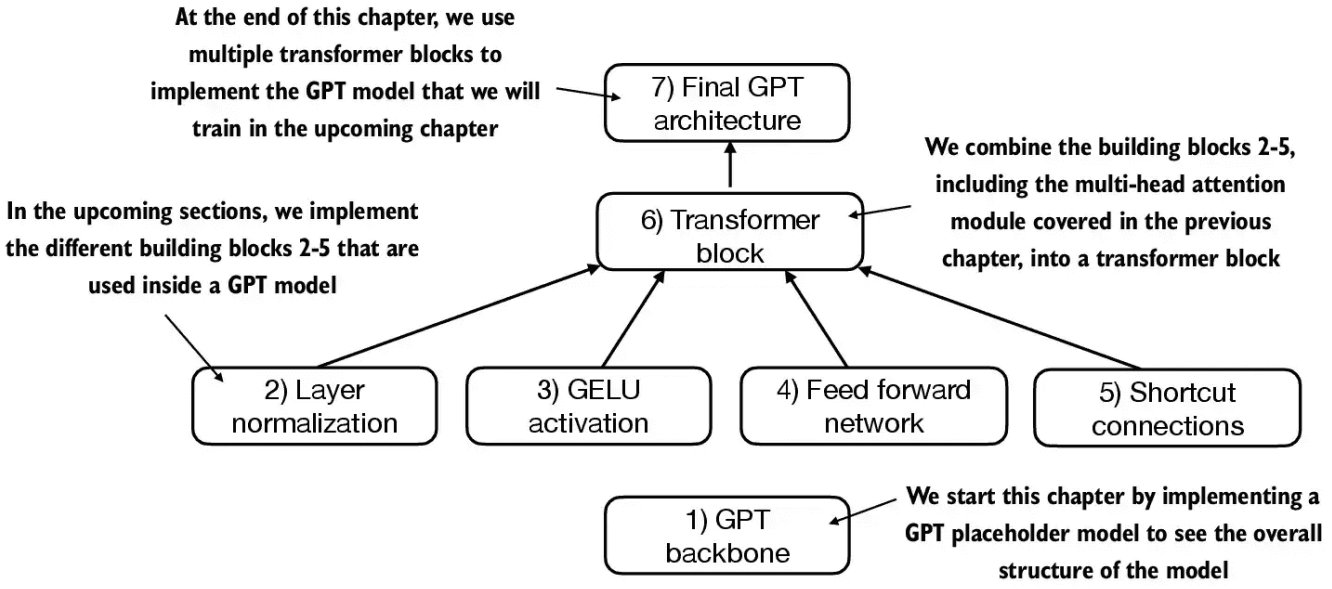
4.1 Coding an LLM architecture

- In previous chapters, we used small embedding dimensions for token inputs and outputs for ease of illustration, ensuring they fit on a single page.
- In this chapter, we consider embedding and model sizes akin to a small GPT-2 model.
- We'll specifically code the architecture of the smallest GPT-2 model (124 million parameters).

Configuration of the small GPT-2 model:

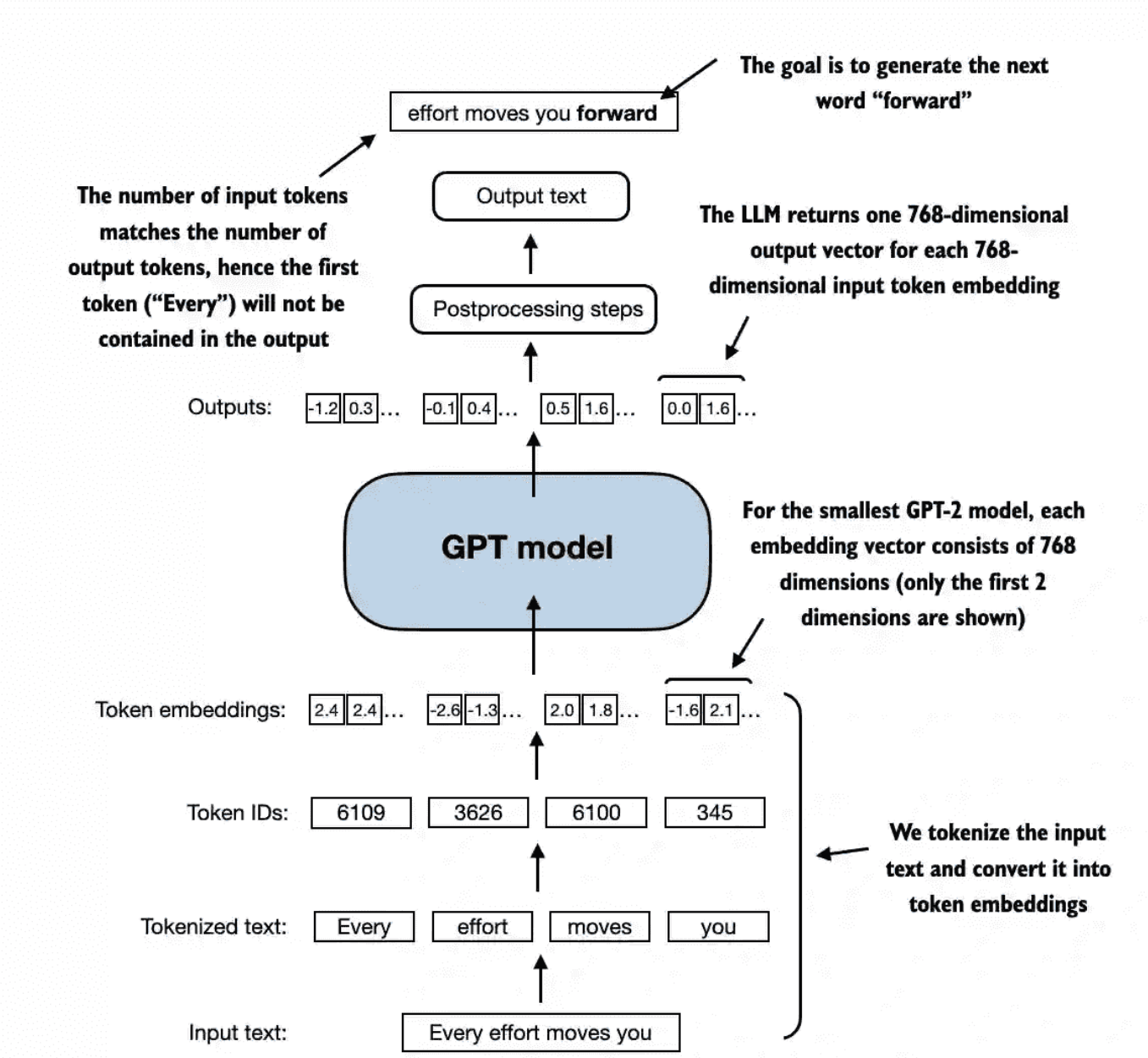
```
GPT_CONFIG_124M = {
    "vocab_size": 50257,        # Vocabulary size
    "context_length": 1024,     # Context length
    "emb_dim": 768,             # Embedding dimension
    "n_heads": 12,              # Number of attention heads
    "n_layers": 12,             # Number of layers
    "drop_rate": 0.1,           # Dropout rate
    "qkv_bias": False           # Query-Key-Value bias
}
```





© 2024 Sebastian Raschka

```python
import torch
import torch.nn as nn


class DummyGPTModel(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.tok_emb = nn.Embedding(cfg["vocab_size"], cfg["emb_dim"])
        self.pos_emb = nn.Embedding(cfg["context_length"], cfg["emb_dim"])
        self.drop_emb = nn.Dropout(cfg["drop_rate"])

        # Use a placeholder for TransformerBlock
        self.trf_blocks = nn.Sequential(
            *[DummyTransformerBlock(cfg) for _ in range(cfg["n_layers"])])

        # Use a placeholder for LayerNorm
        self.final_norm = DummyLayerNorm(cfg["emb_dim"])
        self.out_head = nn.Linear(
            cfg["emb_dim"], cfg["vocab_size"], bias=False
        )

    def forward(self, in_idx):
        batch_size, seq_len = in_idx.shape
        tok_embeds = self.tok_emb(in_idx)
        pos_embeds = self.pos_emb(torch.arange(seq_len, device=in_idx.device))
        x = tok_embeds + pos_embeds
        x = self.drop_emb(x)
        x = self.trf_blocks(x)
        x = self.final_norm(x)
        logits = self.out_head(x)
        return logits


class DummyTransformerBlock(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        # A simple placeholder

    def forward(self, x):
        # This block does nothing and just returns its input.
        return x


class DummyLayerNorm(nn.Module):
    def __init__(self, normalized_shape, eps=1e-5):
        super().__init__()
        # The parameters here are just to mimic the LayerNorm interface.

    def forward(self, x):
        # This layer does nothing and just returns its input.
        return x
```

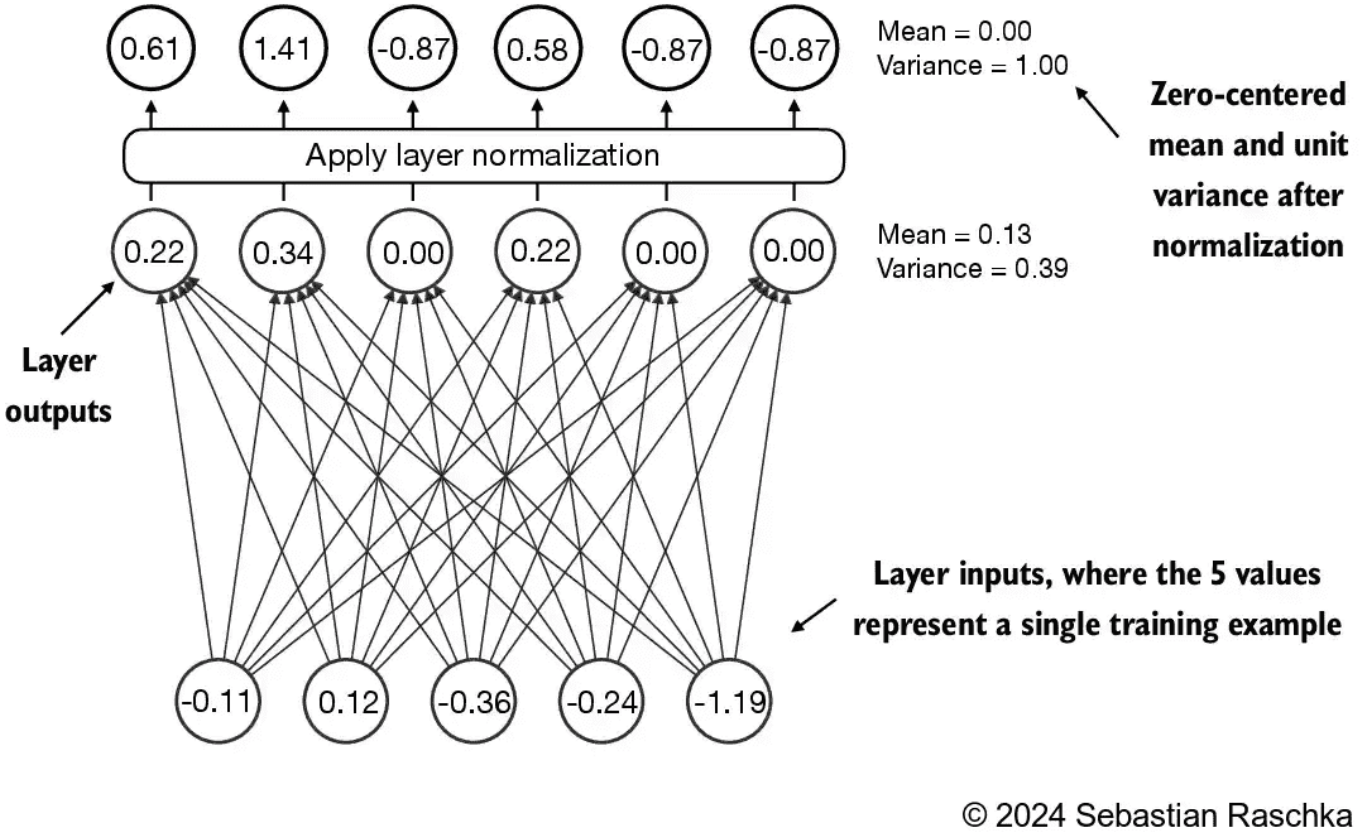


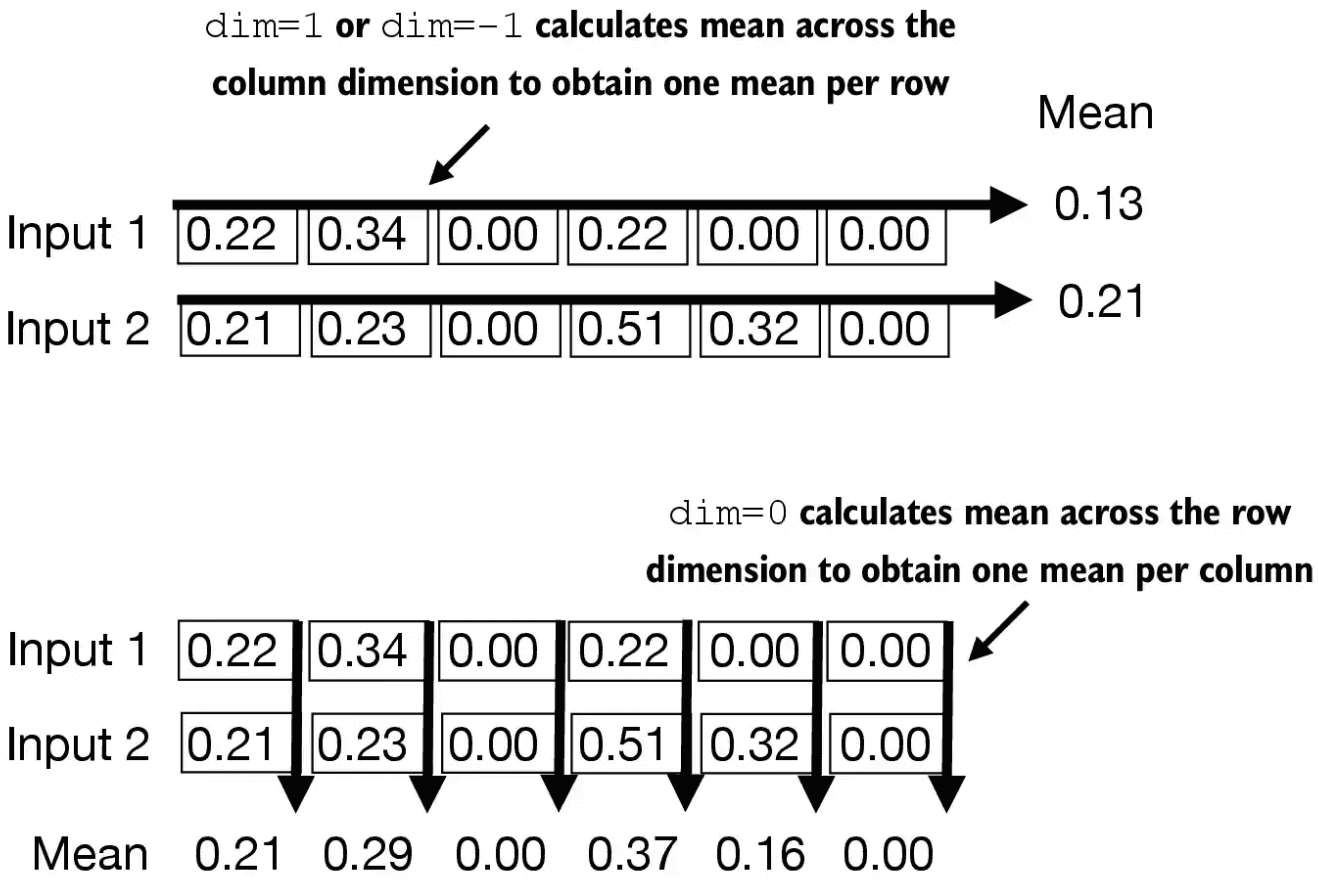

```
import tiktoken
tokenizer = tiktoken.get_encoding("gpt2")
batch = []
txt1 = "Every effort moves you"
txt2 = "Every day holds a"
batch.append(torch.tensor(tokenizer.encode(txt1)))
batch.append(torch.tensor(tokenizer.encode(txt2)))
batch = torch.stack(batch, dim=0)
print(batch)
# tensor([[ 6109,  3626,  6100,   345], #A
#         [ 6109,  1110,  6622,   257]])

torch.manual_seed(123)
model = DummyGPTModel(GPT_CONFIG_124M)
logits = model(batch)
print("Output shape:", logits.shape)
print(logits)
```

```
# Output shape: torch.Size([2, 4, 50257])
# tensor([[[-1.2034,  0.3201, -0.7130,  ..., -1.5548, -0.2390, -0.4667],
#          [-0.1192,  0.4539, -0.4432,  ...,  0.2392,  1.3469,  1.2430],
#          [ 0.5307,  1.6720, -0.4695,  ...,  1.1966,  0.0111,  0.5835],
#          [ 0.0139,  1.6755, -0.3388,  ...,  1.1586, -0.0435, -1.0400]],
#         [[-1.0908,  0.1798, -0.9484,  ..., -1.6047,  0.2439, -0.4530],
#          [-0.7860,  0.5581, -0.0610,  ...,  0.4835, -0.0077,  1.6621],
#          [ 0.3567,  1.2698, -0.6398,  ..., -0.0162, -0.1296,  0.3717],
#          [-0.2407, -0.7349, -0.5102,  ...,  2.0057, -0.3694,  0.1814]]],
#        grad_fn=<UnsafeViewBackward0>)
```

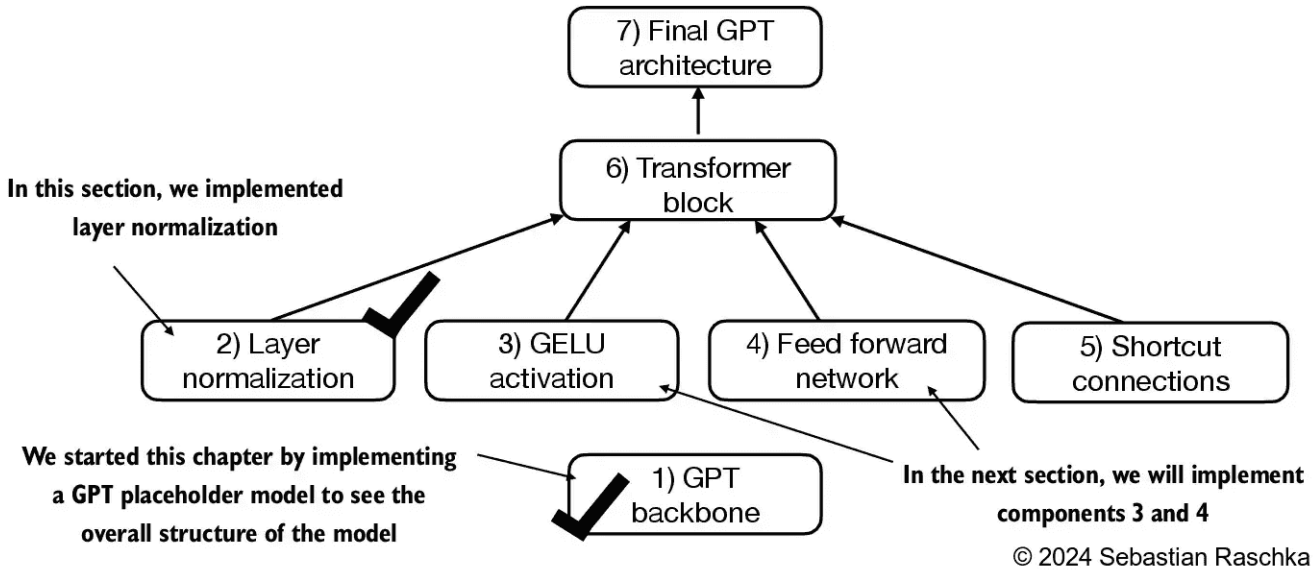
4.2 Normalizing activations with layer normalization

- Training deep neural networks with many layers can sometimes prove challenging due to issues like vanishing or exploding gradients.
- These issues lead to unstable training dynamics and make it difficult for the network to effectively adjust its weights, which means the learning process struggles to find a set of parameters (weights) for the neural network that minimizes the loss function.
- In other words, the network has difficulty learning the underlying patterns in the data to a degree that would allow it to make accurate predictions or decisions.
- The main idea behind layer normalization is to adjust the `activations (outputs)` of a neural network layer to have a mean of 0 and a variance of 1, also known as `unit variance`.



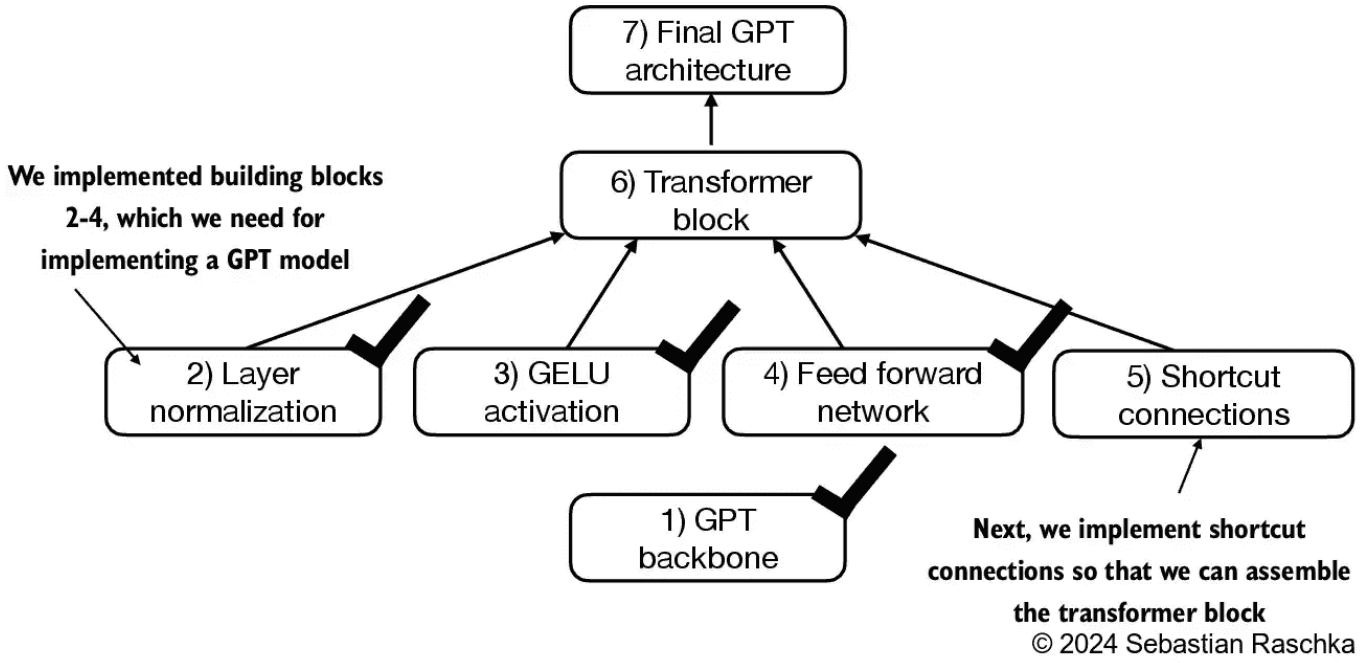



```python
# L2 Normalization (Euclidean norm normalization)
class LayerNorm(nn.Module):
    def __init__(self, emb_dim):
        super().__init__()
        self.eps = 1e-5
        self.scale = nn.Parameter(torch.ones(emb_dim))
        self.shift = nn.Parameter(torch.zeros(emb_dim))

    def forward(self, x):
        mean = x.mean(dim=-1, keepdim=True)
        var = x.var(dim=-1, keepdim=True, unbiased=False)
        norm_x = (x - mean) / torch.sqrt(var + self.eps)
        return self.scale * norm_x + self.shift
```



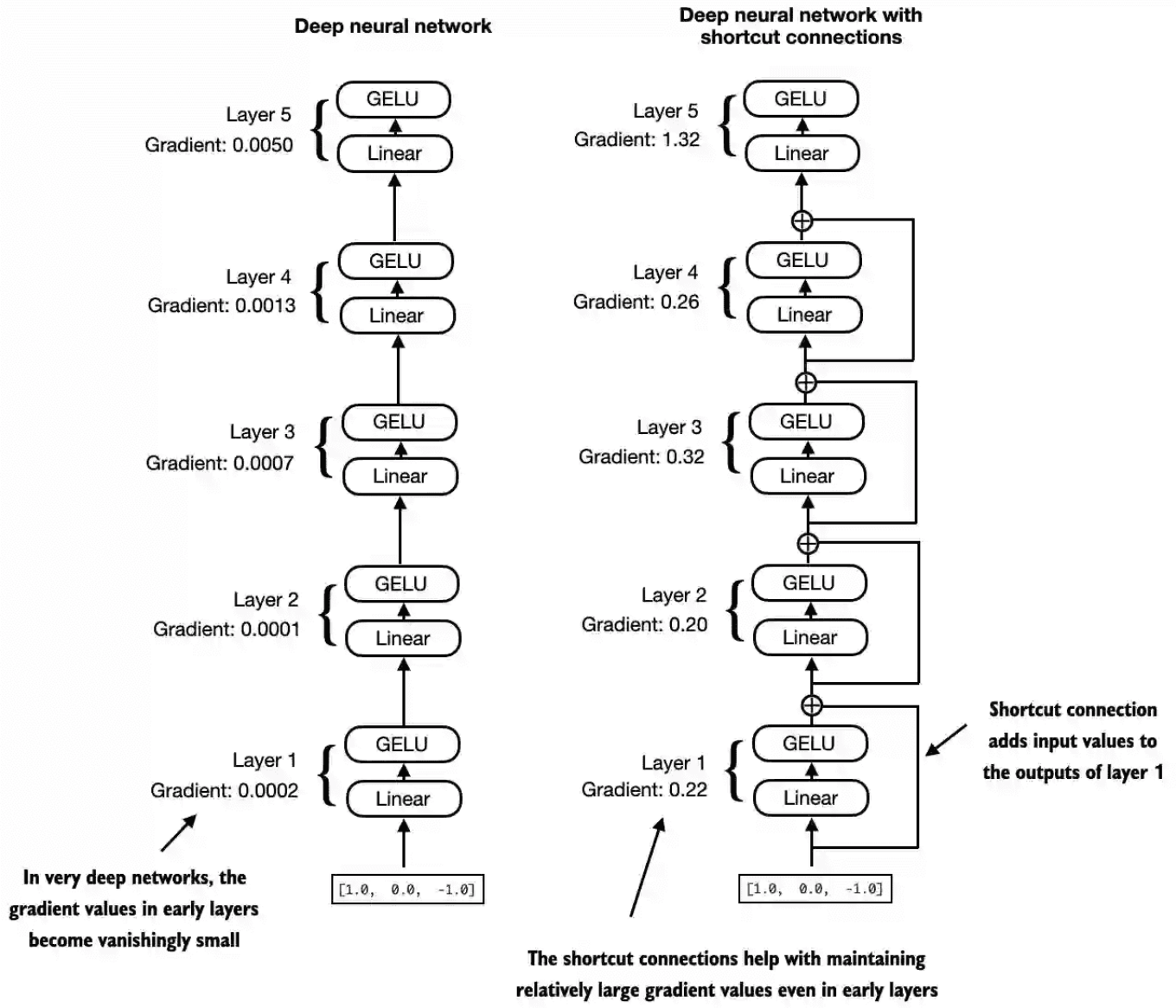
4.3 Implementing a feed forward network with GELU activations

- This section mainly explains the implementation and characteristics of GELU.



4.4 Adding shortcut connections

- `shortcut connections`, also known as `skip or residual connections`.
- Originally, `shortcut connections` were proposed for deep networks in computer vision (specifically, in residual networks) to mitigate the challenge of vanishing gradients.
- The vanishing gradient problem refers to the issue where gradients (which guide weight updates during training) become progressively smaller as they propagate backward through the layers, making it difficult to effectively train earlier layers.




```
class ExampleDeepNeuralNetwork(nn.Module):
    def __init__(self, layer_sizes, use_shortcut):
        super().__init__()
        self.use_shortcut = use_shortcut
        self.layers = nn.ModuleList([
            nn.Sequential(nn.Linear(layer_sizes[0], layer_sizes[1]), GELU()),
            nn.Sequential(nn.Linear(layer_sizes[1], layer_sizes[2]), GELU()),
            nn.Sequential(nn.Linear(layer_sizes[2], layer_sizes[3]), GELU()),
            nn.Sequential(nn.Linear(layer_sizes[3], layer_sizes[4]), GELU()),
            nn.Sequential(nn.Linear(layer_sizes[4], layer_sizes[5]), GELU())
        ])

    def forward(self, x):
        for layer in self.layers:
            # Compute the output of the current layer
            layer_output = layer(x)
            # Check if shortcut can be applied
            if self.use_shortcut and x.shape == layer_output.shape:
                x = x + layer_output
            else:
```

```
                x = layer_output
        return x

def print_gradients(model, x):
    # Forward pass
    output = model(x)
    target = torch.tensor([[0.]])

    # Calculate loss based on how close the target
    # and output are
    loss = nn.MSELoss()
    loss = loss(output, target)

    # Backward pass to calculate the gradients
    loss.backward()

    for name, param in model.named_parameters():
        if 'weight' in name:
            # Print the mean absolute gradient of the weights
            print(f"{name} has gradient mean of {param.grad.abs().mean().item()}")
```

Print the gradient values **without** shortcut connections:

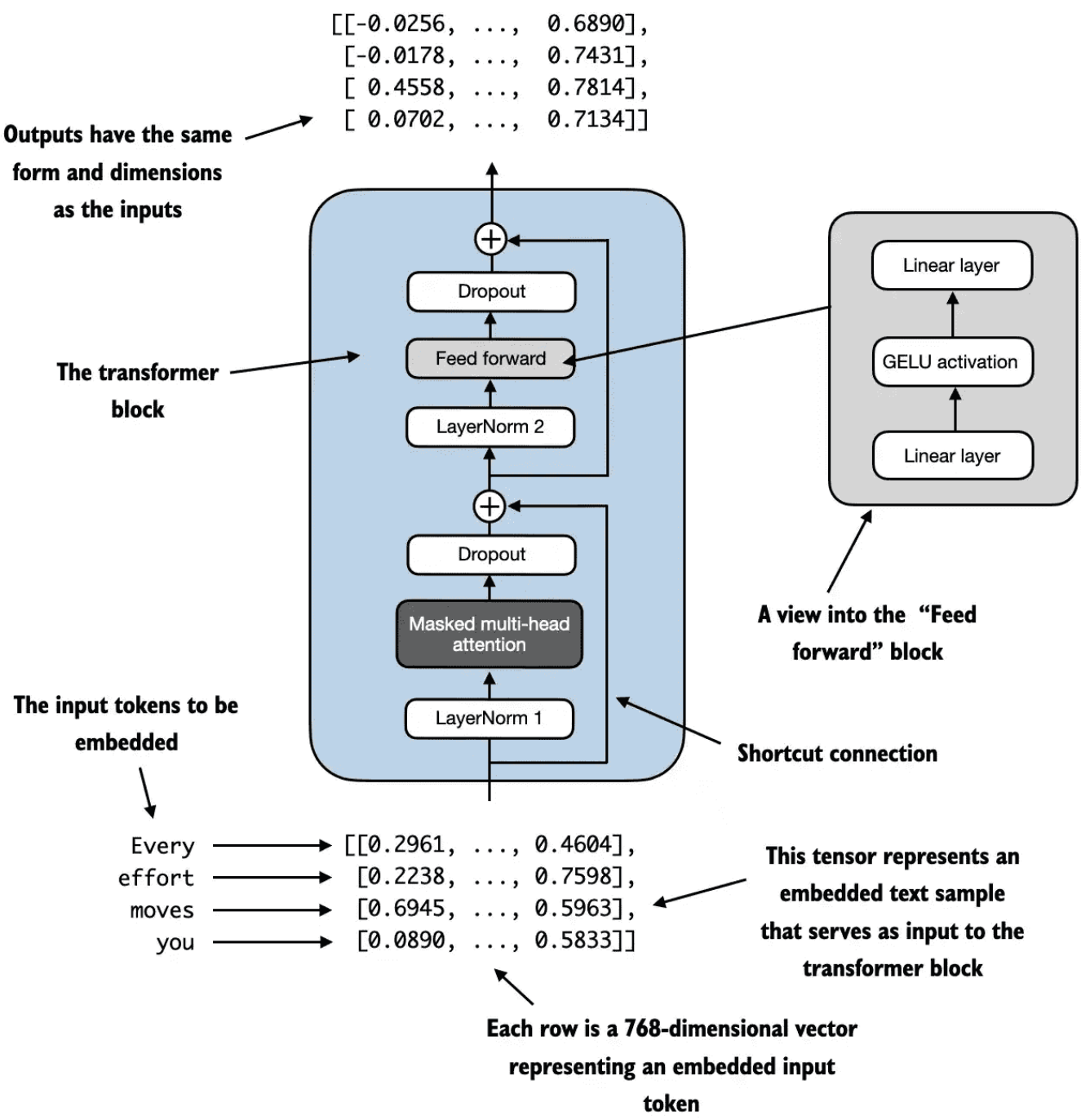
```
layer_sizes = [3, 3, 3, 3, 3, 1]

sample_input = torch.tensor([[1., 0., -1.]])

torch.manual_seed(123)
model_without_shortcut = ExampleDeepNeuralNetwork(
    layer_sizes, use_shortcut=False
)
print_gradients(model_without_shortcut, sample_input)

# Output ==>
# layers.0.0.weight has gradient mean of 0.00020173587836325169
# layers.1.0.weight has gradient mean of 0.0001201116101583466
# layers.2.0.weight has gradient mean of 0.0007152041653171182
# layers.3.0.weight has gradient mean of 0.001398873864673078
# layers.4.0.weight has gradient mean of 0.005049646366387606
```

Print the gradient values **with** shortcut connections:

```
torch.manual_seed(123)
model_with_shortcut = ExampleDeepNeuralNetwork(
    layer_sizes, use_shortcut=True
)
print_gradients(model_with_shortcut, sample_input)
# Output ==>
# layers.0.0.weight has gradient mean of 0.22169792652130127
```

```
# layers.1.0.weight has gradient mean of 0.20694105327129364
# layers.2.0.weight has gradient mean of 0.32896995544433594
# layers.3.0.weight has gradient mean of 0.2665732502937317
# layers.4.0.weight has gradient mean of 1.3258541822433472
```

4.5 Connecting attention and linear layers in a transformer block



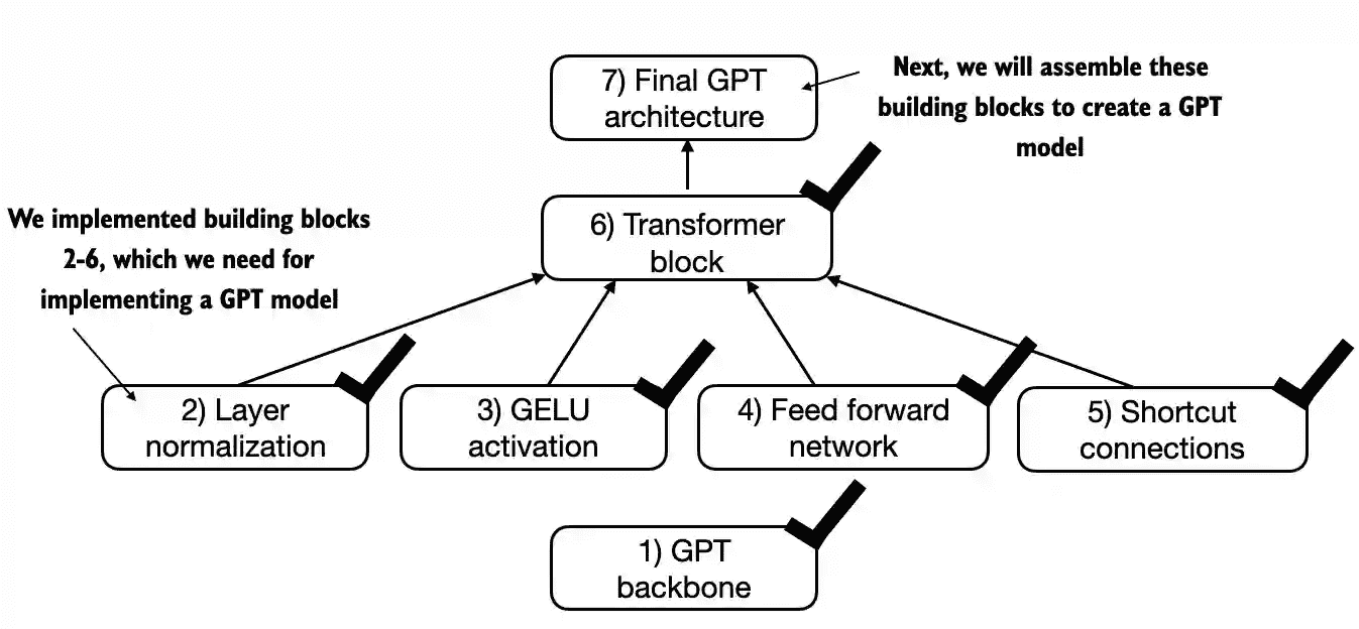
- The idea is that the self-attention mechanism in the multi-head attention block identifies and analyzes relationships between elements in the input sequence.
- In contrast, the feed forward network modifies the data individually at each position.
- This combination not only enables a more nuanced (细微差别) understanding and processing of the input but also enhances the model's overall capacity for handling complex data patterns.
- [Significance of the Combination fromGPT] Self-attention provides global information: by capturing relationships between elements in a sequence, it allows the model to understand context and structural

complexity. Feedforward network strengthens local information: performs specific nonlinear transformations on each position, enhancing its independent feature expression. Synergistic effect: This combination allows the model to both capture global patterns and process local details, thus exhibiting stronger processing capabilities when facing complex data patterns. [Example] Sentence translation task: Self-attention helps the model understand sentence structure and relationships between words; the feedforward network adjusts and optimizes specific information for each word, resulting in more accurate translations.

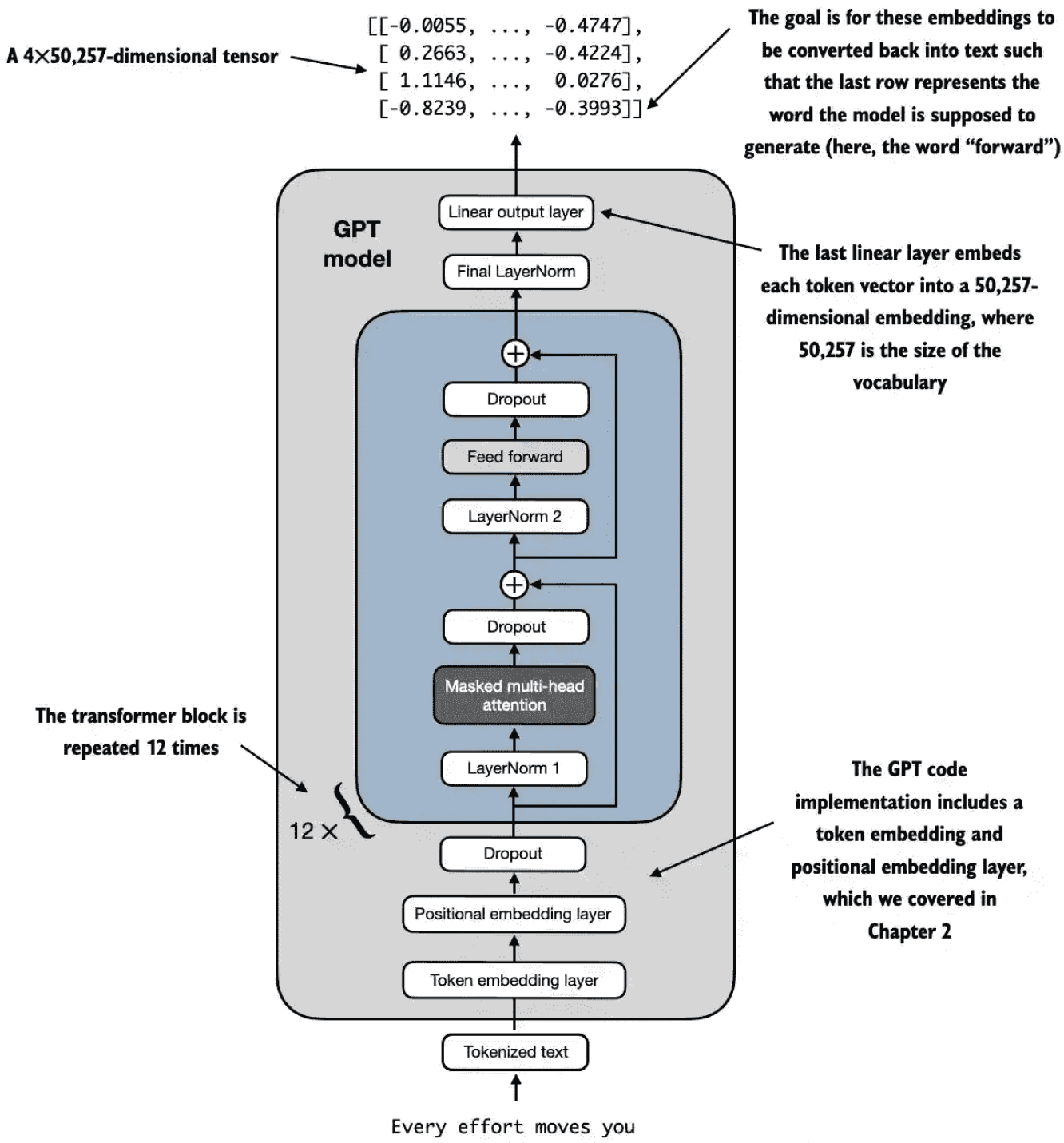
```
from previous_chapters import MultiHeadAttention

class TransformerBlock(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.att = MultiHeadAttention(
            d_in=cfg["emb_dim"],
            d_out=cfg["emb_dim"],
            context_length=cfg["context_length"],
            num_heads=cfg["n_heads"],
            dropout=cfg["drop_rate"],
            qkv_bias=cfg["qkv_bias"])
        self.ff = FeedForward(cfg)
        self.norm1 = LayerNorm(cfg["emb_dim"])
        self.norm2 = LayerNorm(cfg["emb_dim"])
        self.drop_shortcut = nn.Dropout(cfg["drop_rate"])

    def forward(self, x):
        # Shortcut connection for attention block
        shortcut = x
        x = self.norm1(x)
        x = self.att(x)  # Shape [batch_size, num_tokens, emb_size]
        x = self.drop_shortcut(x)
        x = x + shortcut  # Add the original input back

        # Shortcut connection for feed forward block
        shortcut = x
        x = self.norm2(x)
        x = self.ff(x)
        x = self.drop_shortcut(x)
        x = x + shortcut  # Add the original input back

        return x
```

- The preservation of shape throughout the transformer block architecture is not incidental but a crucial aspect of its design.
- This design enables its effective application across a wide range of sequence-to-sequence tasks, where each output vector directly corresponds to an input vector, maintaining a one-to-one relationship.
- However, the output is a context vector that encapsulates information from the entire input sequence, as we learned in chapter 3.

- This means that while the physical dimensions of the sequence (length and feature size) remain unchanged as it passes through the transformer block, the content of each output vector is re-encoded to integrate contextual information from across the entire input sequence.



4.6 Coding the GPT model



- The transformer block is repeated many times throughout a GPT model architecture.
- In the case of the 124 million parameter GPT-2 model, it's repeated 12 times.
- In the case of the largest GPT-2 model with 1,542 million parameters, this transformer block is repeated 36 times.

```
class GPTModel(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.tok_emb = nn.Embedding(cfg["vocab_size"], cfg["emb_dim"])
        self.pos_emb = nn.Embedding(cfg["context_length"], cfg["emb_dim"])
        self.drop_emb = nn.Dropout(cfg["drop_rate"])

        self.trf_blocks = nn.Sequential(
```

```
            *[TransformerBlock(cfg) for _ in range(cfg["n_layers"])])

        self.final_norm = LayerNorm(cfg["emb_dim"])
        self.out_head = nn.Linear(
            cfg["emb_dim"], cfg["vocab_size"], bias=False
        )

    def forward(self, in_idx):
        batch_size, seq_len = in_idx.shape
        tok_embeds = self.tok_emb(in_idx)
        pos_embeds = self.pos_emb(torch.arange(seq_len, device=in_idx.device))
        x = tok_embeds + pos_embeds  # Shape [batch_size, num_tokens, emb_size]
        x = self.drop_emb(x)
        x = self.trf_blocks(x)
        x = self.final_norm(x)
        logits = self.out_head(x)
        return logits
```

Request:

```
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)

out = model(batch)
print("Input batch:\n", batch)
print("\nOutput shape:", out.shape)
print(out)

# Output
# Input batch:
#  tensor([[6109, 3626, 6100,  345],
#         [6109, 1110, 6622,  257]])

# Output shape: torch.Size([2, 4, 50257])
# tensor([[[ 0.3613,  0.4222, -0.0711,  ...,  0.3483,  0.4661, -0.2838],
#          [-0.1792, -0.5660, -0.9485,  ...,  0.0477,  0.5181, -0.3168],
#          [ 0.7120,  0.0332,  0.1085,  ...,  0.1018, -0.4327, -0.2553],
#          [-1.0076,  0.3418, -0.1190,  ...,  0.7195,  0.4023,  0.0532]],

#         [[-0.2564,  0.0900,  0.0335,  ...,  0.2659,  0.4454, -0.6806],
#          [ 0.1230,  0.3653, -0.2074,  ...,  0.7705,  0.2710,  0.2246],
#          [ 1.0558,  1.0318, -0.2800,  ...,  0.6936,  0.3205, -0.3178],
#          [-0.1565,  0.3926,  0.3288,  ...,  1.2630, -0.1858,  0.0388]]],
#        grad_fn=<UnsafeViewBackward0>)
```

Total parameters calculation:

```
total_params = sum(p.numel() for p in model.parameters())
print(f"Total number of parameters: {total_params:,}")
```

```
# Output
# Total number of parameters: 163,009,536
# Question
# Why is it not 124M as stated in the GPT2 paper?
```

> **Note:** In the original GPT-2 paper, the researchers applied `weight tying`, which means that they reused the token embedding layer (`tok_emb`) as the output layer, which means setting `self.out_head.weight = self.tok_emb.weight`. The token embedding and output layers are very large due to the number of rows for the 50,257 in the tokenizer's vocabulary. Weight tying reduces the overall memory footprint and computational complexity of the model. See `WeightTying` for details.

Remove parameters reused by GPT2:

```
total_params_gpt2 =  total_params - sum(p.numel() for p in
model.out_head.parameters())
print(f"Number of trainable parameters considering weight tying:
{total_params_gpt2:,}")

# Output
# Number of trainable parameters considering weight tying: 124,412,160
```

Memory usage:

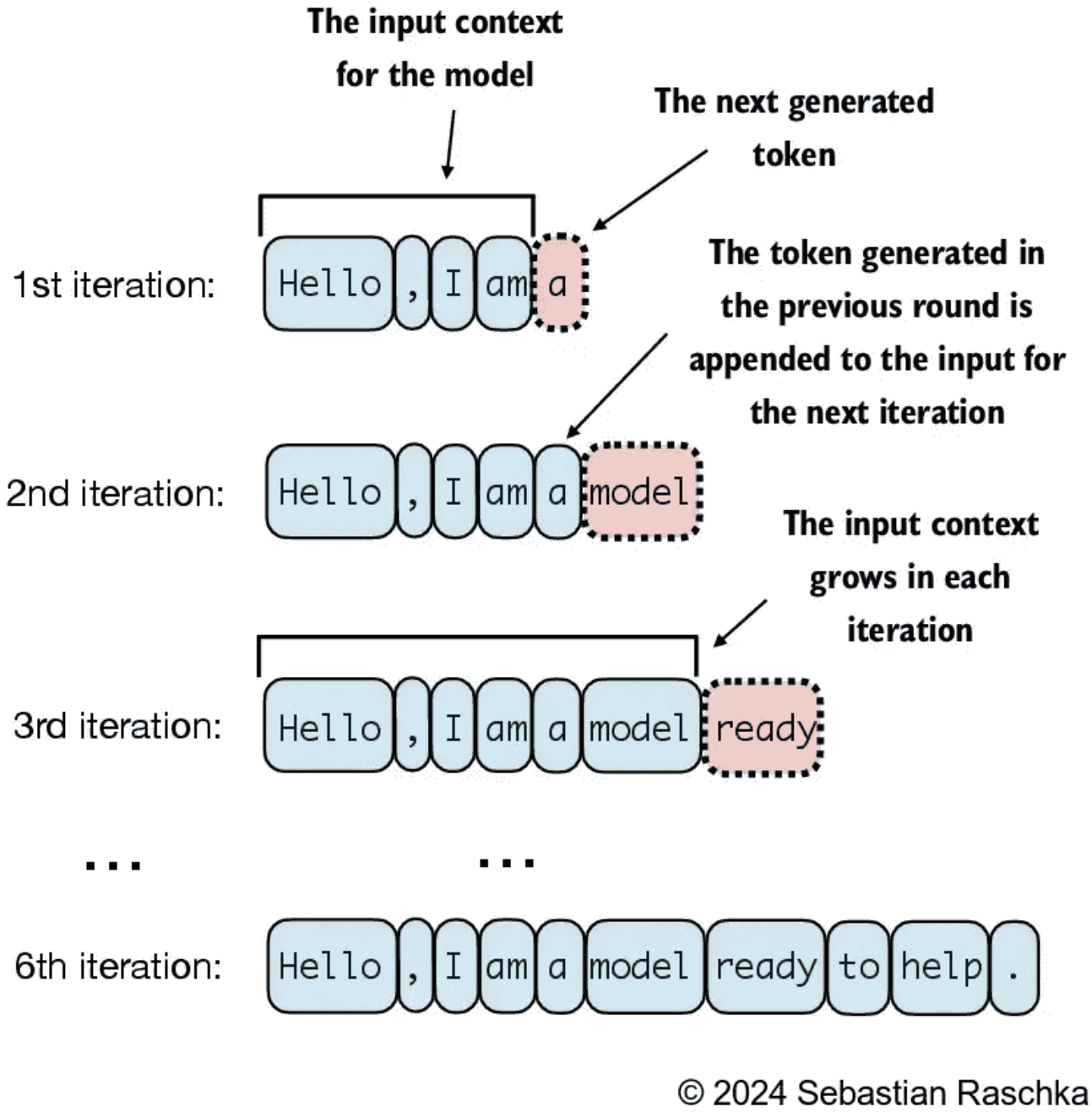
```
# Calculate the total size in bytes (assuming float32, 4 bytes per parameter)
total_size_bytes = total_params * 4

# Convert to megabytes
total_size_mb = total_size_bytes / (1024 * 1024)

print(f"Total size of the model: {total_size_mb:.2f} MB")
# Output
# Total size of the model: 621.83 MB
```

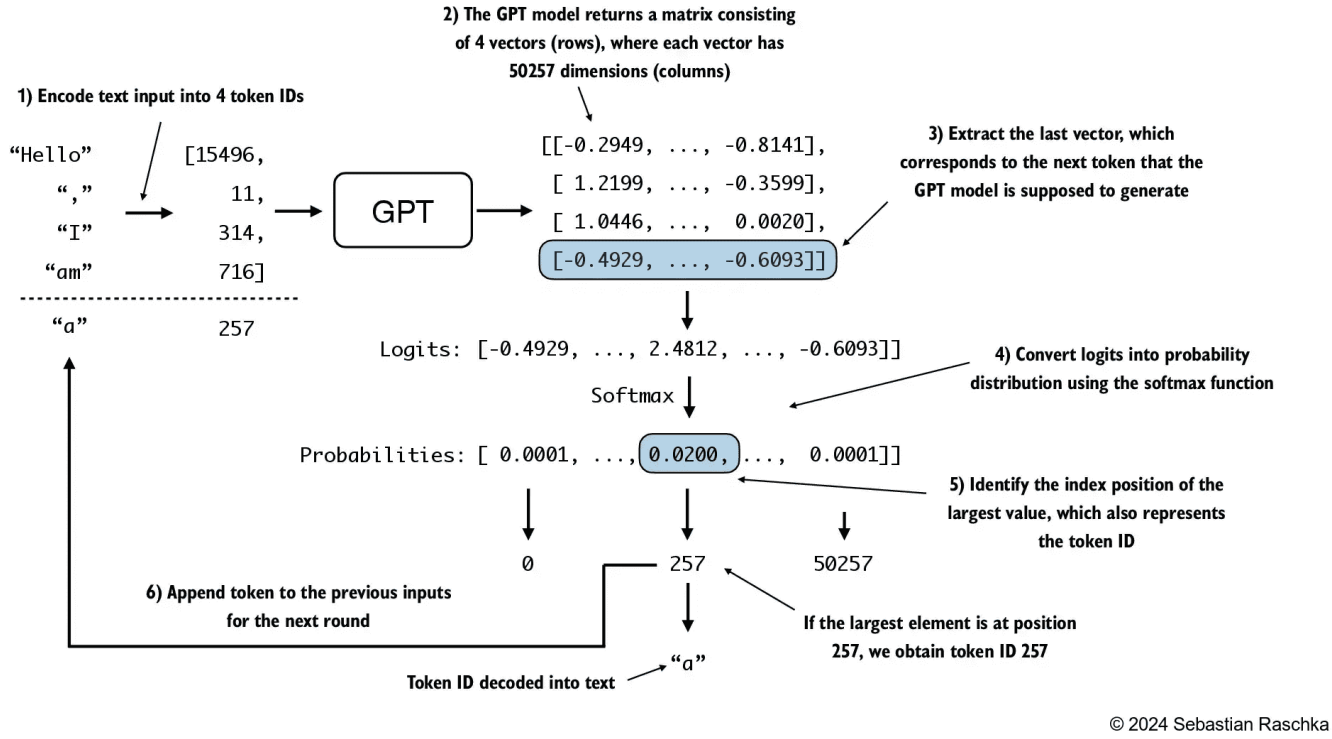
Exercise:

```
- **GPT2-small** (the 124M configuration we already implemented):
    - "emb_dim" = 768
    - "n_layers" = 12
    - "n_heads" = 12

- **GPT2-medium:**
    - "emb_dim" = 1024
    - "n_layers" = 24
    - "n_heads" = 16

- **GPT2-large:**
```

```
    - "emb_dim" = 1280
    - "n_layers" = 36
    - "n_heads" = 20

- **GPT2-XL:**
    - "emb_dim" = 1600
    - "n_layers" = 48
    - "n_heads" = 25
```

4.7 Generating text



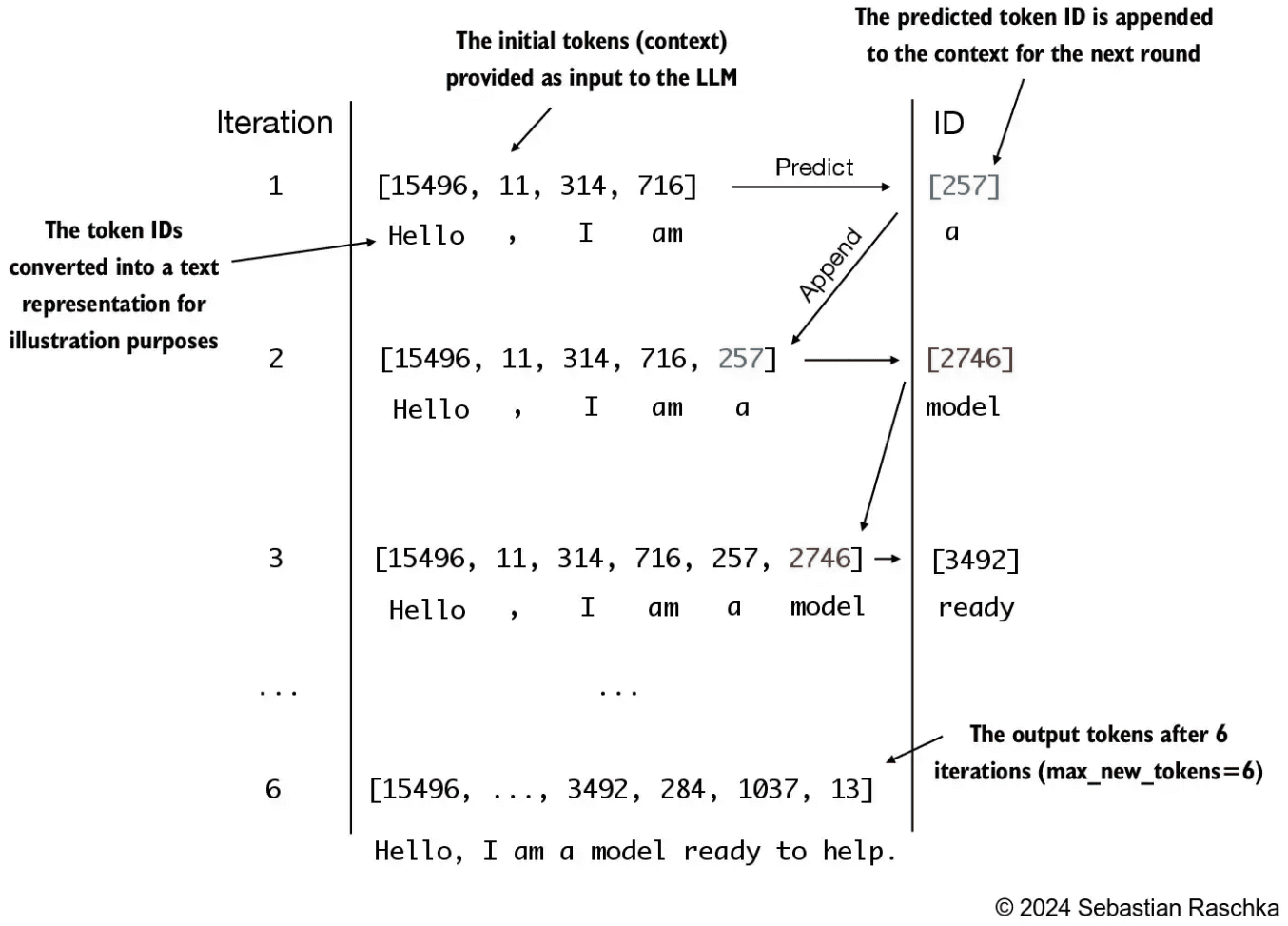


© 2024 Sebastian Raschka

```python
def generate_text_simple(model, idx, max_new_tokens, context_size):
    # idx is (batch, n_tokens) array of indices in the current context
    for _ in range(max_new_tokens):

        # Crop current context if it exceeds the supported context size
        # E.g., if LLM supports only 5 tokens, and the context size is 10
        # then only the last 5 tokens are used as context
        idx_cond = idx[:, -context_size:]

        # Get the predictions
        with torch.no_grad():
            logits = model(idx_cond)

        # Focus only on the last time step
        # (batch, n_tokens, vocab_size) becomes (batch, vocab_size)
        logits = logits[:, -1, :]

        # Apply softmax to get probabilities
        probas = torch.softmax(logits, dim=-1)  # (batch, vocab_size)

        # Get the idx of the vocab entry with the highest probability value
        idx_next = torch.argmax(probas, dim=-1, keepdim=True)  # (batch, 1)

        # Append sampled index to the running sequence
        idx = torch.cat((idx, idx_next), dim=1)  # (batch, n_tokens+1)

    return idx
```

> **Note:** Actually, there's no need to execute `torch.softmax`. Executing `torch.argmax` directly on the logits would yield the same result. We coded the conversion to illustrate the full process of

> transforming logits to probabilities, which can add additional intuition, such as that the model generates the most likely next token, which is known as `greedy decoding`. In the next chapter, when we will implement the GPT training code, we will also introduce additional `sampling techniques` where we modify the softmax outputs such that the model doesn't always select the most likely token, which introduces variability and creativity in the generated text.



© 2024 Sebastian Raschka

Prepare the data:

```
start_context = "Hello, I am"

encoded = tokenizer.encode(start_context)
print("encoded:", encoded)

encoded_tensor = torch.tensor(encoded).unsqueeze(0)
print("encoded_tensor.shape:", encoded_tensor.shape)
# Output
# encoded: [15496, 11, 314, 716]
# encoded_tensor.shape: torch.Size([1, 4])
```

Put the model into `.eval()` mode, which disables random components like dropout, which are only used during training:

```
model.eval() # disable dropout

out = generate_text_simple(
    model=model,
    idx=encoded_tensor,
    max_new_tokens=6,
    context_size=GPT_CONFIG_124M["context_length"]
)

print("Output:", out)
print("Output length:", len(out[0]))
# Output
# Output: tensor([[15496,    11,   314,   716, 27018, 24086, 47843, 30961, 42348,
7267]])
# Output length: 10
```

Remove batch dimension and convert back into text:

```
decoded_text = tokenizer.decode(out.squeeze(0).tolist())
print(decoded_text)
# Output (Untrained)
# Hello, I am Featureiman Byeswickattribute argue
```

4.8 Summary

- Layer normalization stabilizes training by ensuring that each layer's outputs have a consistent mean and variance.
- Shortcut connections are connections that skip one or more layers by feeding the output of one layer directly to a deeper layer, which helps mitigate the vanishing gradient problem when training deep neural networks, such as LLMs.
- Transformer blocks are a core structural component of GPT models, combining `masked multi-head attention modules` with `fully connected feed-forward networks` that use the GELU activation function.
- GPT models are LLMs with many repeated transformer blocks that have millions to billions of parameters.
- GPT models come in various sizes, for example, 124 million, and 1542 million parameters, which we can implement with the same GPTModel Python class.
- The text generation capability of a GPT-like LLM involves decoding output tensors into human-readable text by sequentially predicting one token at a time based on a given input context.
- Without training, a GPT model generates incoherent text, which underscores the importance of model training for coherent text generation, which is the topic of subsequent chapters.

```
import torch
import torch.nn as nn
```

```
class LayerNorm(nn.Module):
    def __init__(self, emb_dim):
        super().__init__()
        self.eps = 1e-5
        self.scale = nn.Parameter(torch.ones(emb_dim))
        self.shift = nn.Parameter(torch.zeros(emb_dim))

    def forward(self, x):
        mean = x.mean(dim=-1, keepdim=True)
        var = x.var(dim=-1, keepdim=True, unbiased=False)
        norm_x = (x - mean) / torch.sqrt(var + self.eps)
        return self.scale * norm_x + self.shift


class GELU(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, x):
        return 0.5 * x * (1 + torch.tanh(
            torch.sqrt(torch.tensor(2.0 / torch.pi)) *
            (x + 0.044715 * torch.pow(x, 3))
        ))

class FeedForward(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(cfg["emb_dim"], 4 * cfg["emb_dim"]),
            GELU(),
            nn.Linear(4 * cfg["emb_dim"], cfg["emb_dim"]),
        )

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



class MultiHeadAttention(nn.Module):
    def __init__(self, d_in, d_out, context_length, dropout, num_heads,
qkv_bias=False):
        super().__init__()
        assert d_out % num_heads == 0, "d_out must be divisible by num_heads"

        self.d_out = d_out
        self.num_heads = num_heads
        self.head_dim = d_out // num_heads  # Reduce the projection dim to match
desired output dim

        self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_key = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_value = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.out_proj = nn.Linear(d_out, d_out)  # Linear layer to combine head
outputs
```

```
        self.dropout = nn.Dropout(dropout)
        self.register_buffer('mask', torch.triu(torch.ones(context_length,
context_length), diagonal=1))

    def forward(self, x):
        b, num_tokens, d_in = x.shape

        keys = self.W_key(x)  # Shape: (b, num_tokens, d_out)
        queries = self.W_query(x)
        values = self.W_value(x)

        # We implicitly split the matrix by adding a `num_heads` dimension
        # Unroll last dim: (b, num_tokens, d_out) -> (b, num_tokens, num_heads,
head_dim)
        keys = keys.view(b, num_tokens, self.num_heads, self.head_dim)
        values = values.view(b, num_tokens, self.num_heads, self.head_dim)
        queries = queries.view(b, num_tokens, self.num_heads, self.head_dim)

        # Transpose: (b, num_tokens, num_heads, head_dim) -> (b, num_heads,
num_tokens, head_dim)
        keys = keys.transpose(1, 2)
        queries = queries.transpose(1, 2)
        values = values.transpose(1, 2)

        # Compute scaled dot-product attention (aka self-attention) with a causal
mask
        attn_scores = queries @ keys.transpose(2, 3)  # Dot product for each head

        # Original mask truncated to the number of tokens and converted to boolean
        mask_bool = self.mask.bool()[:num_tokens, :num_tokens]

        # Use the mask to fill attention scores
        attn_scores.masked_fill_(mask_bool, -torch.inf)

        attn_weights = torch.softmax(attn_scores / keys.shape[-1]**0.5, dim=-1)
        attn_weights = self.dropout(attn_weights)

        # Shape: (b, num_tokens, num_heads, head_dim)
        context_vec = (attn_weights @ values).transpose(1, 2)

        # Combine heads, where self.d_out = self.num_heads * self.head_dim
        context_vec = context_vec.contiguous().view(b, num_tokens, self.d_out)
        context_vec = self.out_proj(context_vec)  # optional projection

        return context_vec



class TransformerBlock(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.att = MultiHeadAttention(
            d_in=cfg["emb_dim"],
            d_out=cfg["emb_dim"],
```

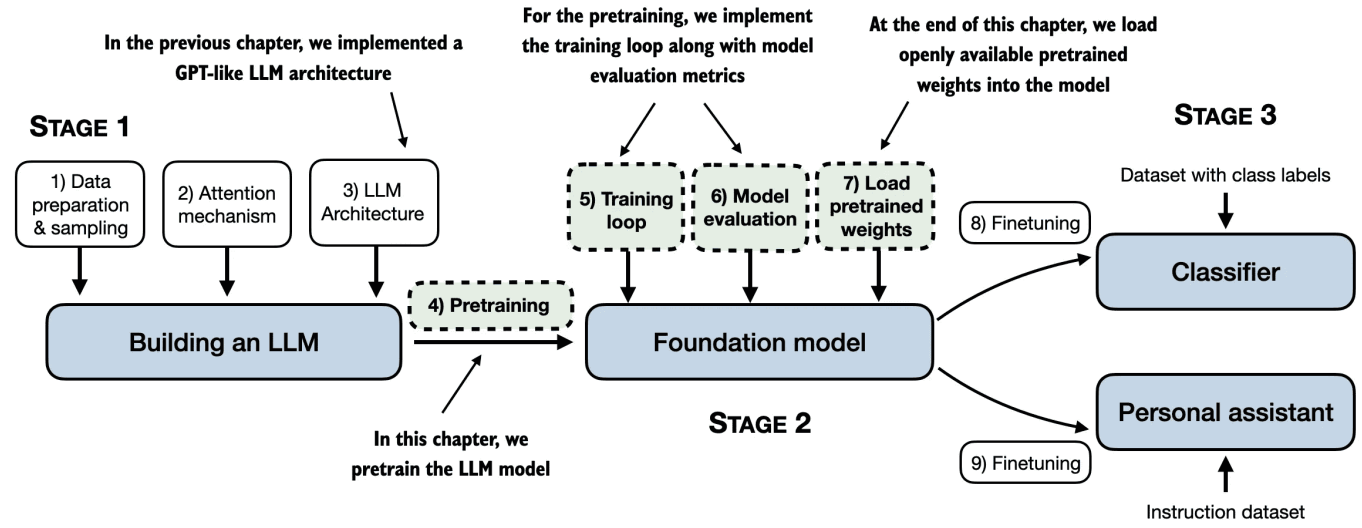
```
            context_length=cfg["context_length"],
            num_heads=cfg["n_heads"],
            dropout=cfg["drop_rate"],
            qkv_bias=cfg["qkv_bias"])
        self.ff = FeedForward(cfg)
        self.norm1 = LayerNorm(cfg["emb_dim"])
        self.norm2 = LayerNorm(cfg["emb_dim"])
        self.drop_shortcut = nn.Dropout(cfg["drop_rate"])

    def forward(self, x):
        # Shortcut connection for attention block
        shortcut = x
        x = self.norm1(x)
        x = self.att(x)  # Shape [batch_size, num_tokens, emb_size]
        x = self.drop_shortcut(x)
        x = x + shortcut  # Add the original input back

        # Shortcut connection for feed forward block
        shortcut = x
        x = self.norm2(x)
        x = self.ff(x)
        x = self.drop_shortcut(x)
        x = x + shortcut  # Add the original input back

        return x

def generate_text_simple(model, idx, max_new_tokens, context_size):
    # idx is (batch, n_tokens) array of indices in the current context
    for _ in range(max_new_tokens):

        # Crop current context if it exceeds the supported context size
        # E.g., if LLM supports only 5 tokens, and the context size is 10
        # then only the last 5 tokens are used as context
        idx_cond = idx[:, -context_size:]

        # Get the predictions
        with torch.no_grad():
            logits = model(idx_cond)

        # Focus only on the last time step
        # (batch, n_tokens, vocab_size) becomes (batch, vocab_size)
        logits = logits[:, -1, :]

        # Apply softmax to get probabilities
        probas = torch.softmax(logits, dim=-1)  # (batch, vocab_size)

        # Get the idx of the vocab entry with the highest probability value
        idx_next = torch.argmax(probas, dim=-1, keepdim=True)  # (batch, 1)

        # Append sampled index to the running sequence
        idx = torch.cat((idx, idx_next), dim=1)  # (batch, n_tokens+1)

    return idx
```

```python
class GPTModel(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.tok_emb = nn.Embedding(cfg["vocab_size"], cfg["emb_dim"])
        self.pos_emb = nn.Embedding(cfg["context_length"], cfg["emb_dim"])
        self.drop_emb = nn.Dropout(cfg["drop_rate"])

        self.trf_blocks = nn.Sequential(
            *[TransformerBlock(cfg) for _ in range(cfg["n_layers"])])

        self.final_norm = LayerNorm(cfg["emb_dim"])
        self.out_head = nn.Linear(
            cfg["emb_dim"], cfg["vocab_size"], bias=False
        )

    def forward(self, in_idx):
        batch_size, seq_len = in_idx.shape
        tok_embeds = self.tok_emb(in_idx)
        pos_embeds = self.pos_emb(torch.arange(seq_len, device=in_idx.device))
        x = tok_embeds + pos_embeds  # Shape [batch_size, num_tokens, emb_size]
        x = self.drop_emb(x)
        x = self.trf_blocks(x)
        x = self.final_norm(x)
        logits = self.out_head(x)
        return logits

GPT_CONFIG_124M = {
    "vocab_size": 50257,    # Vocabulary size
    "context_length": 1024, # Context length
    "emb_dim": 768,         # Embedding dimension
    "n_heads": 12,          # Number of attention heads
    "n_layers": 12,         # Number of layers
    "drop_rate": 0.1,       # Dropout rate
    "qkv_bias": False       # Query-Key-Value bias
}

import tiktoken

tokenizer = tiktoken.get_encoding("gpt2")

start_context = "Hello, I am"
encoded = tokenizer.encode(start_context)
encoded_tensor = torch.tensor(encoded).unsqueeze(0)

torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.eval() # disable dropout

out = generate_text_simple(
    model=model,
    idx=encoded_tensor,
    max_new_tokens=6,
    context_size=GPT_CONFIG_124M["context_length"]
)
```

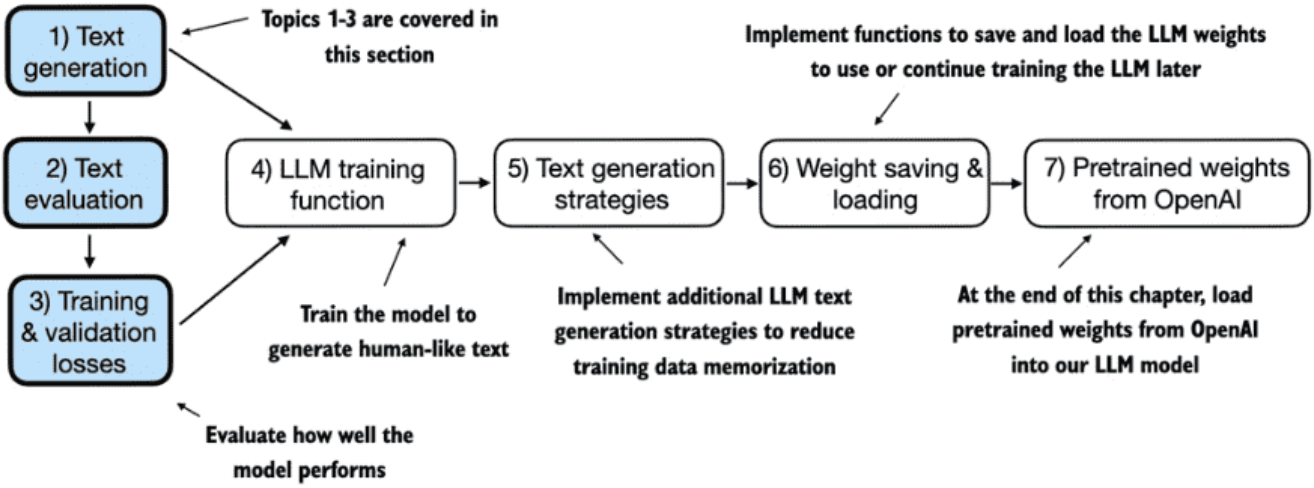
```
decoded_text = tokenizer.decode(out.squeeze(0).tolist())
print(decoded_text)
```

5 Pretraining on Unlabeled Data

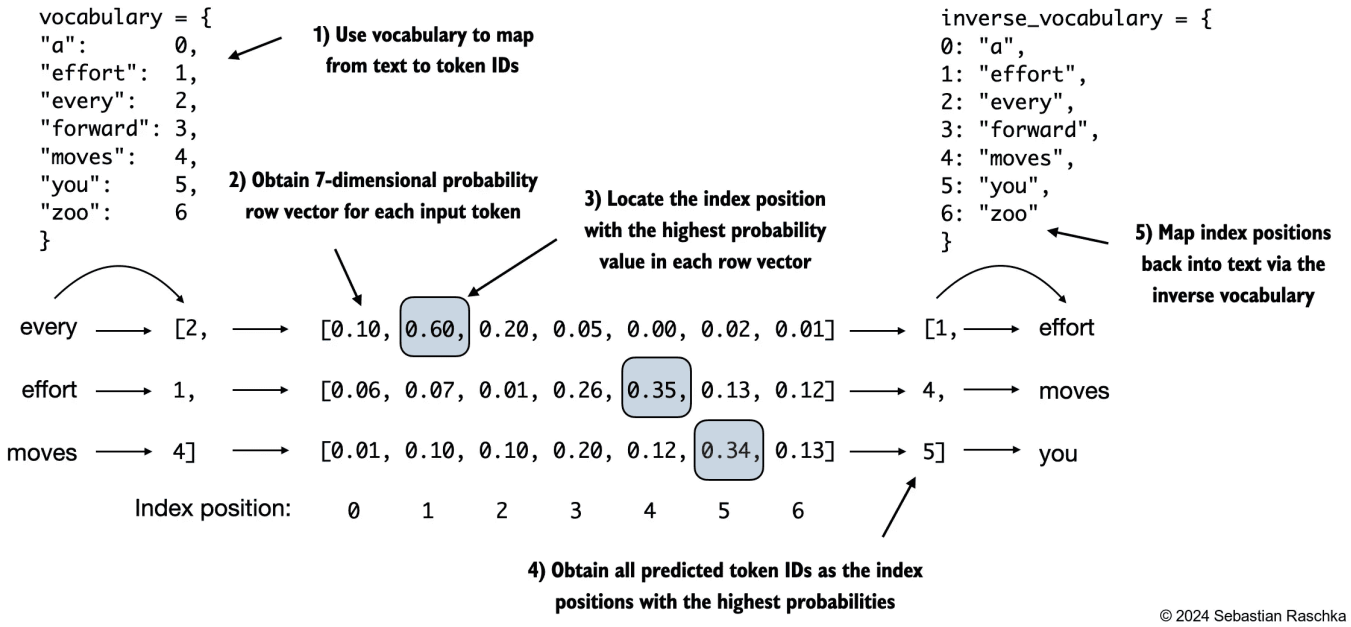
- Computing the training and validation set losses to assess the quality of LLM-generated text during training
- Implementing a training function and pretraining the LLM
- Saving and loading model weights to continue training an LLM Loading pretrained weights from OpenAI



5.1 Evaluating generative text models



5.1.1 Using GPT to generate text

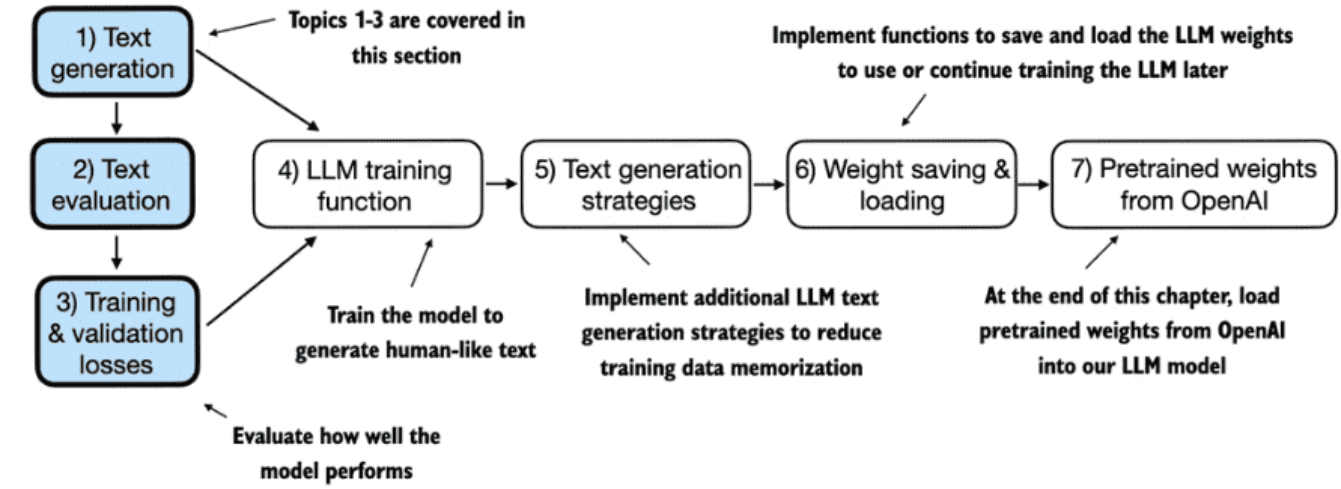


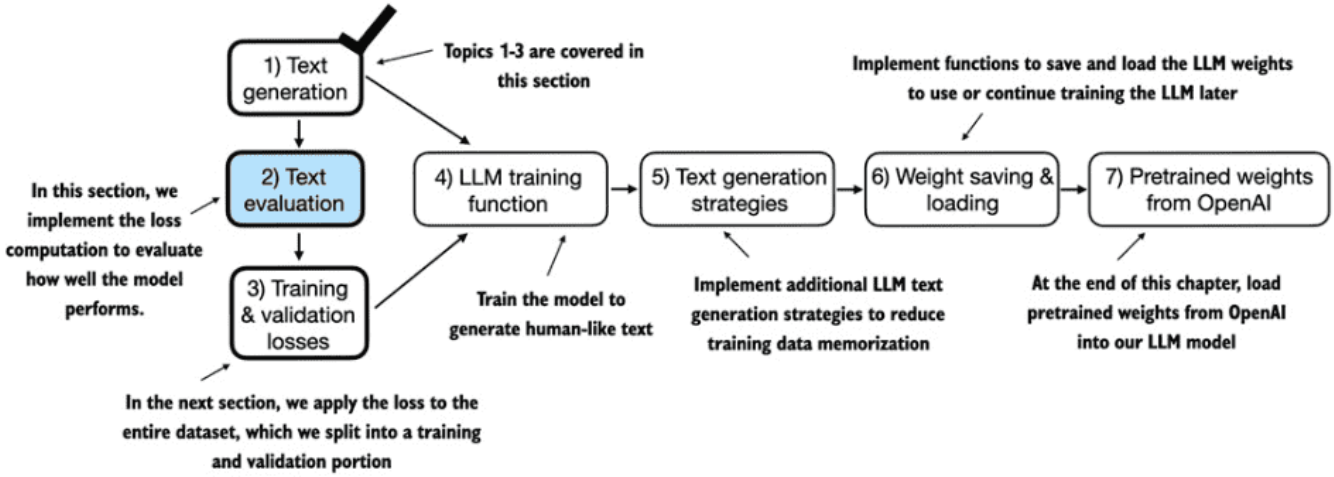
- Utility functions for text to token ID conversion

```
import tiktoken
from previous_chapters import generate_text_simple

def text_to_token_ids(text, tokenizer):
    encoded = tokenizer.encode(text, allowed_special={'<|endoftext|>'})
    encoded_tensor = torch.tensor(encoded).unsqueeze(0) # add batch dimension
    return encoded_tensor

def token_ids_to_text(token_ids, tokenizer):
    flat = token_ids.squeeze(0) # remove batch dimension
    return tokenizer.decode(flat.tolist())

start_context = "Every effort moves you"
tokenizer = tiktoken.get_encoding("gpt2")

token_ids = generate_text_simple(
    model=model,
    idx=text_to_token_ids(start_context, tokenizer),
    max_new_tokens=10,
    context_size=GPT_CONFIG_124M["context_length"]
)

print("Output text:\n", token_ids_to_text(token_ids, tokenizer))

# Output
# Output text:
#  Every effort moves you rentingetic wasn refres RexMeCHicular stren
```

5.1.2 Calculating the text generation loss

- Let's first review how the data is loaded from chapter 2 and how the text is generated via the generate_text_simple function from chapter 4.





Input & Expected Output:

```
inputs = torch.tensor([[16833, 3626, 6100], # ["every effort moves",
                       [40,    1107, 588]])   #  "I really like"]

targets = torch.tensor([[3626, 6100, 345  ],  # [" effort moves you",
                        [1107,  588, 11311]]) #  " really like chocolate"]
```

We feed the inputs into the model to calculate logit vectors for the two input examples, each comprising three tokens, and apply the softmax function to transform these logit values into probability scores:

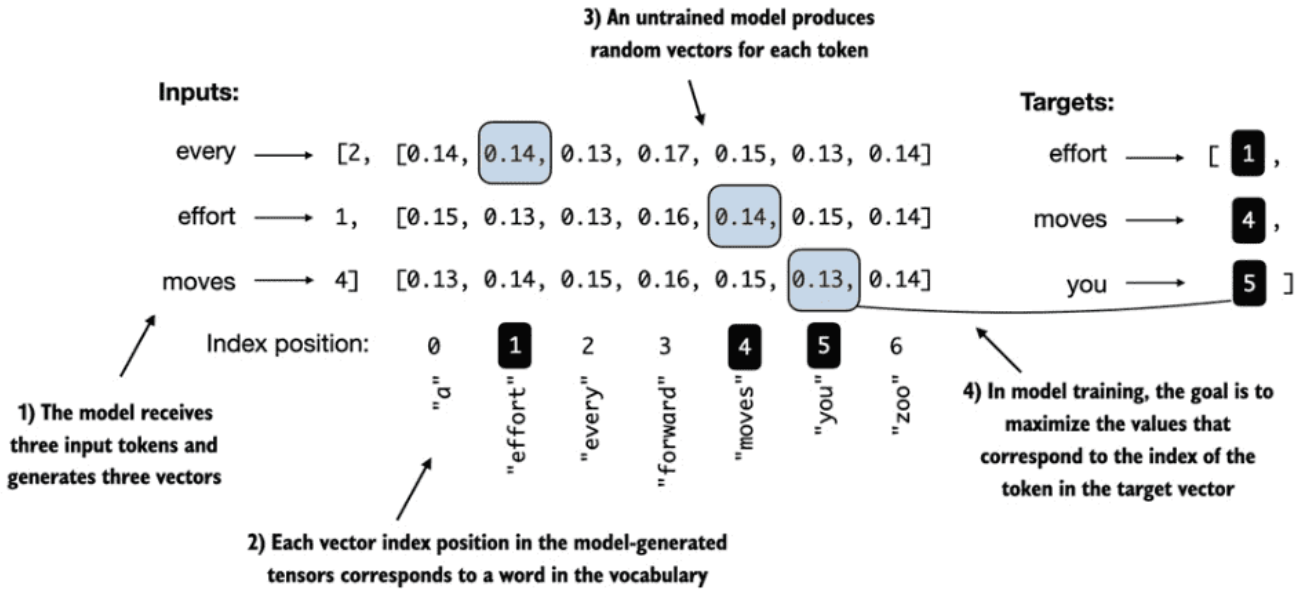
```
with torch.no_grad():
    logits = model(inputs)

probas = torch.softmax(logits, dim=-1) # Probability of each token in vocabulary
print(probas.shape) # Shape: (batch_size, num_tokens, vocab_size)
```

```
# Output
# torch.Size([2, 3, 50257])
```

By applying the argmax function to the probability scores to obtain the corresponding token IDs:

```
# This gets the model's final output and token IDs (of course, since it's not
trained yet, the result is messy)
token_ids = torch.argmax(probas, dim=-1, keepdim=True)
print("Token IDs:\n", token_ids)
# Output
# Token IDs:
# tensor([[[36397],
#          [39619],
#          [20610]],
#         [[ 8615],
#          [49289],
#          [47105]]])
```

- Part of the text evaluation process is to measure “how far” the generated tokens are from the correct predictions (targets).
- The training function will use this information to adjust the model weights to generate text that is more similar to (or ideally matches) the target text.



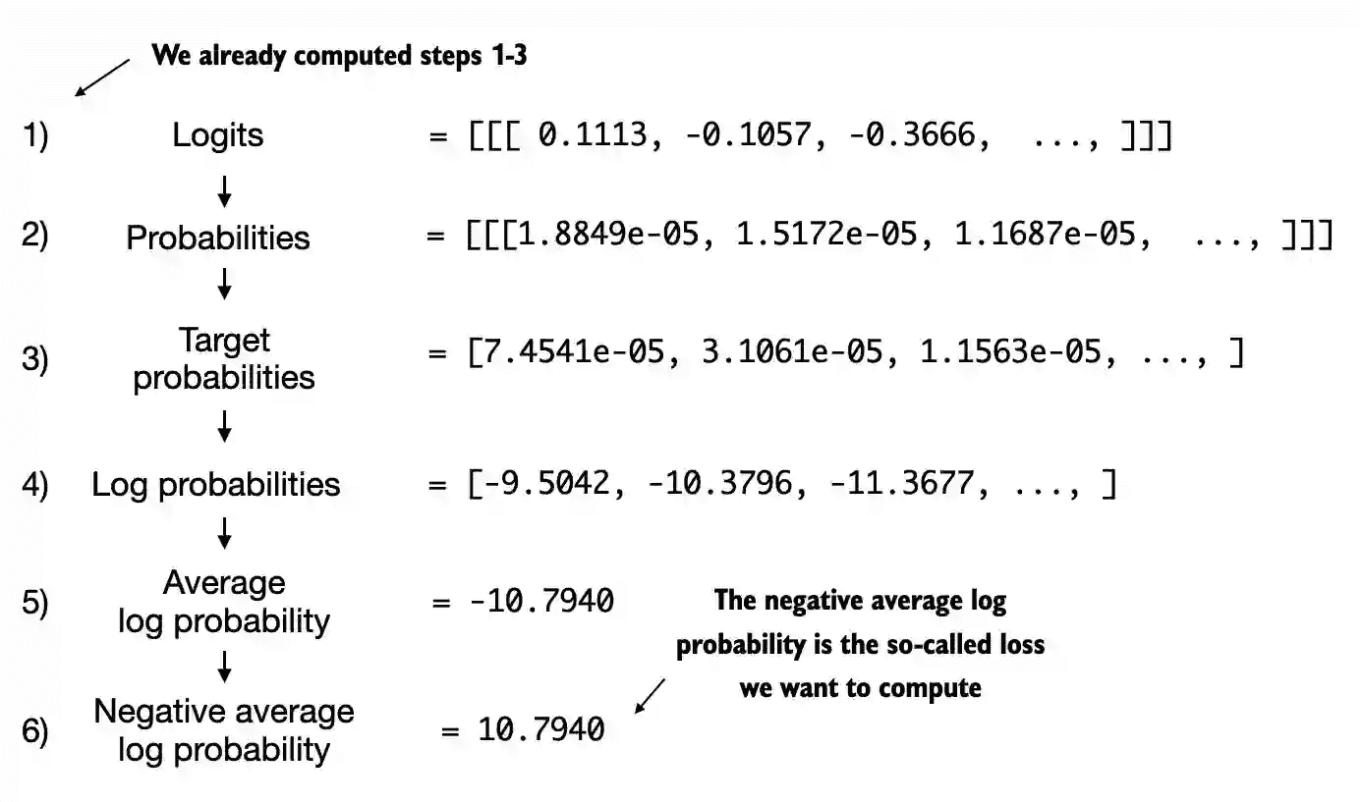
The token probabilities corresponding to the target indices are as follows:

```
# Let's look at the probabilities of the desired output now
text_idx = 0
target_probas_1 = probas[text_idx, [0, 1, 2], targets[text_idx]]
print("Text 1:", target_probas_1)

text_idx = 1
```

```
target_probas_2 = probas[text_idx, [0, 1, 2], targets[text_idx]]
print("Text 2:", target_probas_2)
# Output
# Text 1: tensor([2.3466e-05, 2.0531e-05, 1.1733e-05])
# Text 2: tensor([4.2794e-05, 1.6248e-05, 1.1586e-05])
```

> **Note:** The goal of training an LLM is to maximize these values, aiming to get them as close to a probability of 1. This way, we ensure the LLM consistently picks the target token (essentially the next word in the sentence) as the next token it generates.

Backpropagation




```
# Compute logarithm of all token probabilities
log_probas = torch.log(torch.cat((target_probas_1, target_probas_2)))
print(log_probas)
# tensor([ -9.5042, -10.3796, -11.3677, -11.4798,  -9.7764, -12.2561])

# Calculate the average probability for each token
avg_log_probas = torch.mean(log_probas)
print(avg_log_probas)
# tensor(-10.7940)

# In deep learning, the common practice is to bring the negative average log
probability down to 0.
neg_avg_log_probas = avg_log_probas * -1
print(neg_avg_log_probas)
# tensor(10.7940)
```

> **Note:** The term for this negative value, -10.7722 turning into 10.7722, is known as the cross entropy loss in deep learning.

Cross entropy loss

The shape of the logits and target tensors:

```
# Logits have shape (batch_size, num_tokens, vocab_size)
print("Logits shape:", logits.shape)
# Targets have shape (batch_size, num_tokens)
print("Targets shape:", targets.shape)
```

For the `cross_entropy` function in PyTorch, we want to flatten these tensors by combining them over the batch dimension:

```
logits_flat = logits.flatten(0, 1)
targets_flat = targets.flatten()
print("Flattened logits:", logits_flat.shape)
print("Flattened targets:", targets_flat.shape)
# Output
# Flattened logits: torch.Size([6, 50257])
# Flattened targets: torch.Size([6])
```

PyTorch's cross_entropy function:

```
loss = torch.nn.functional.cross_entropy(logits_flat, targets_flat)
print(loss)
```

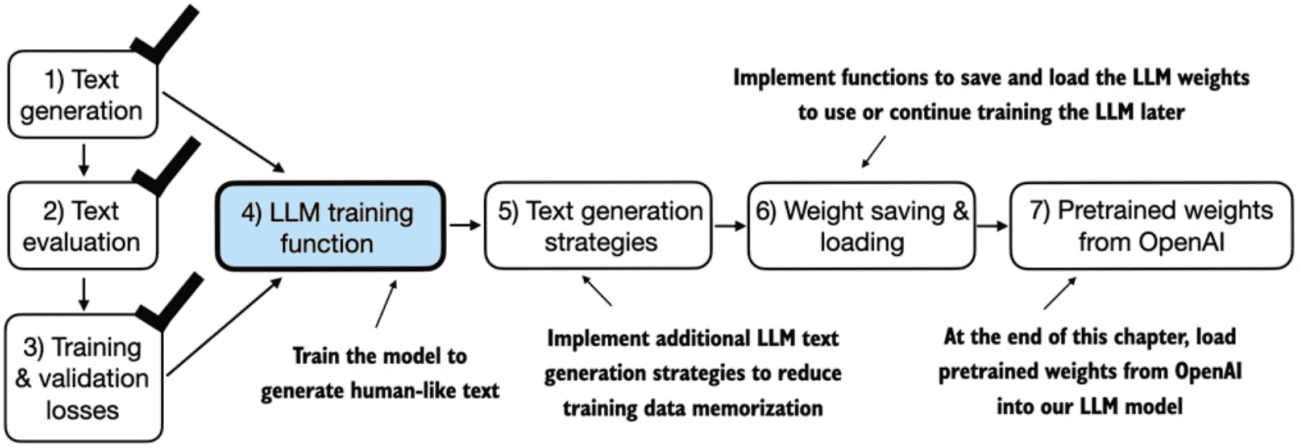
> **Note:** See definition of `_cross_entropy:` for details.

Perplexity

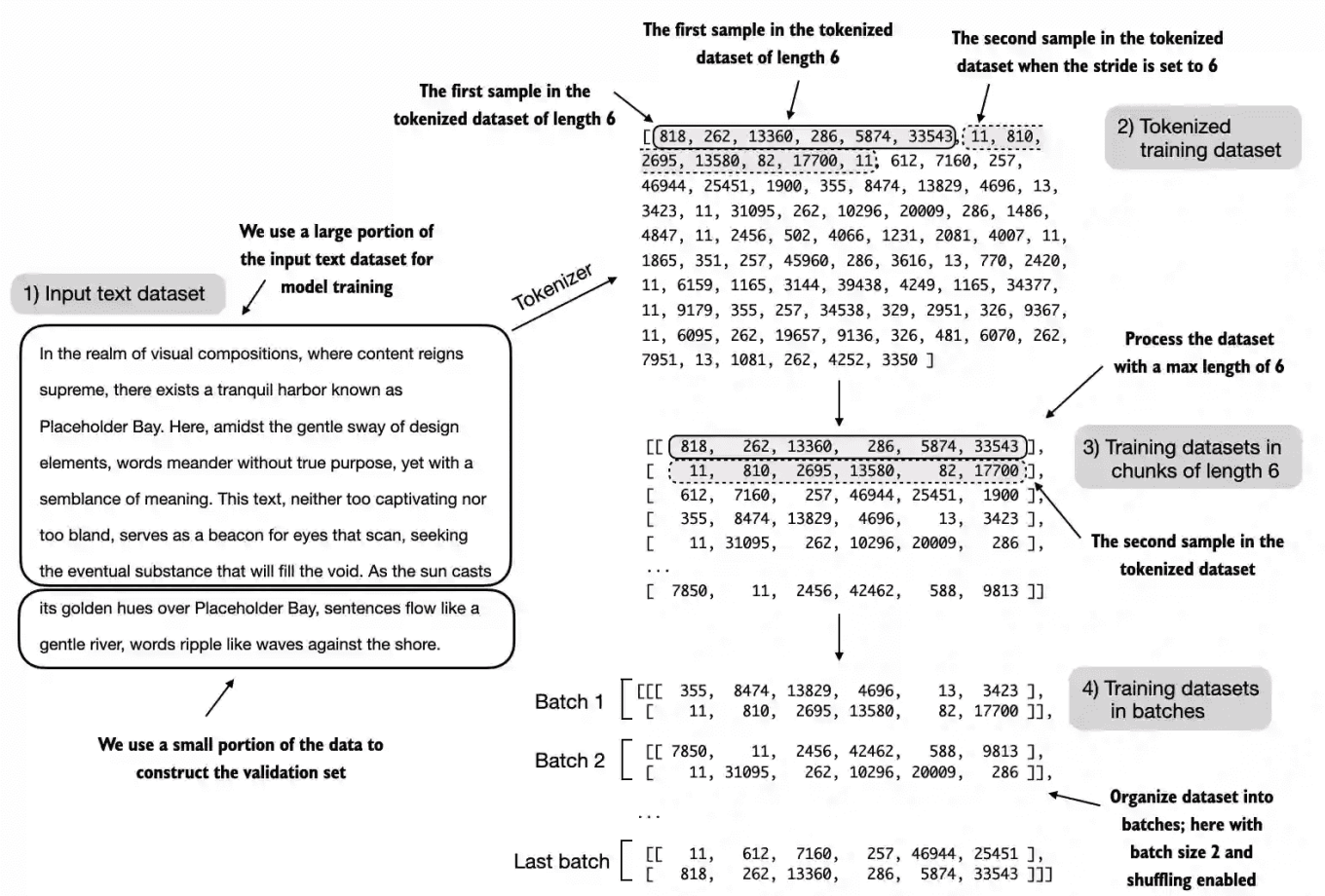
- Perplexity is a measure often used alongside cross entropy loss to evaluate the performance of models in tasks like language modeling. It can provide a more interpretable way to understand the uncertainty of a model in predicting the next token in a sequence.

> **Note:** See definition of `Perplexity` (困惑度) for details.

5.1.3 Calculating the training and validation set losses



The cost of pretraining LLMs

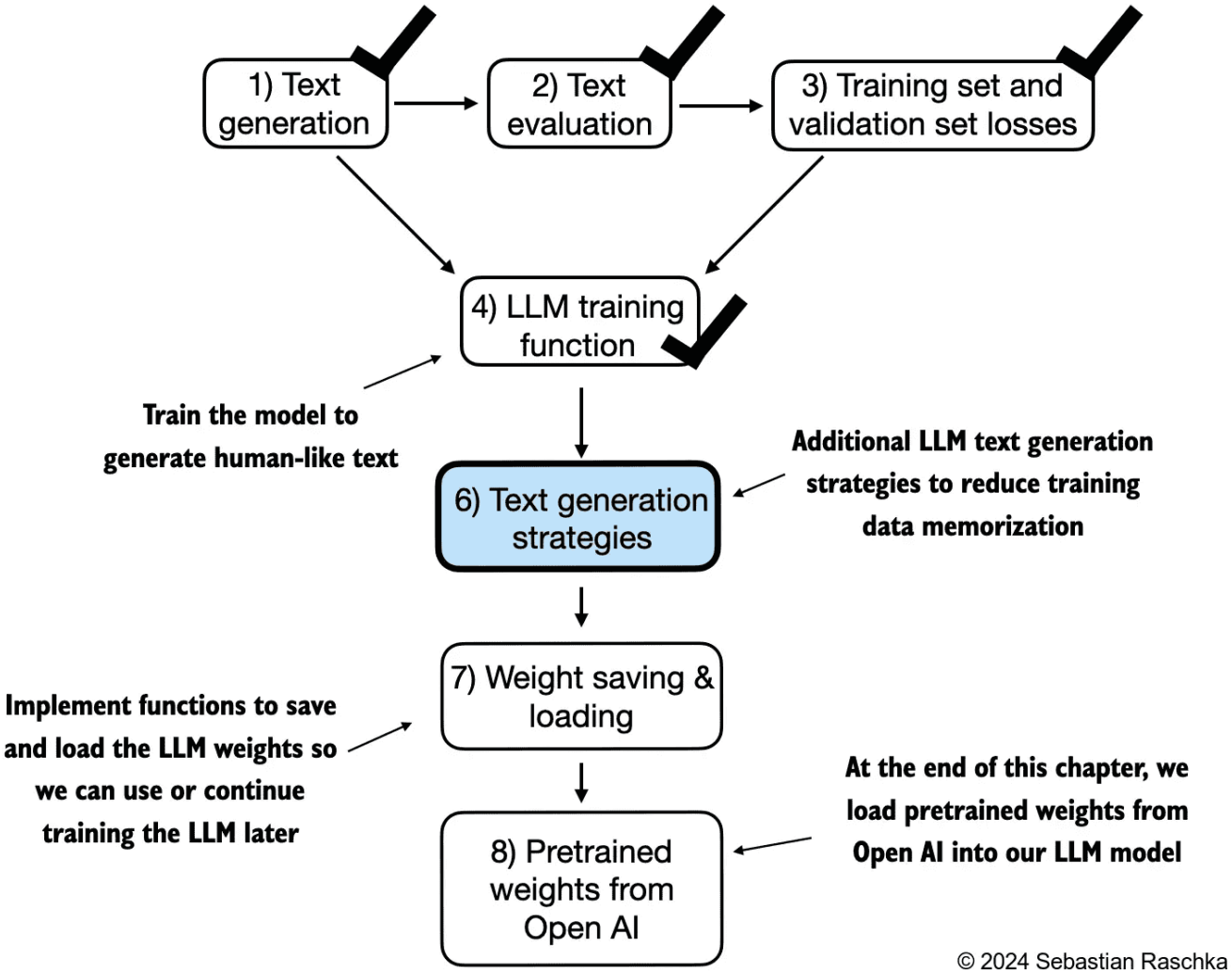


- For visualization purposes, Figure 5.9 uses a max_length=6 due to spatial constraints. However, for the actual data loaders we are implementing, we set the max_length equal to the 256-token context length that the LLM supports so that the LLM sees longer texts during training.

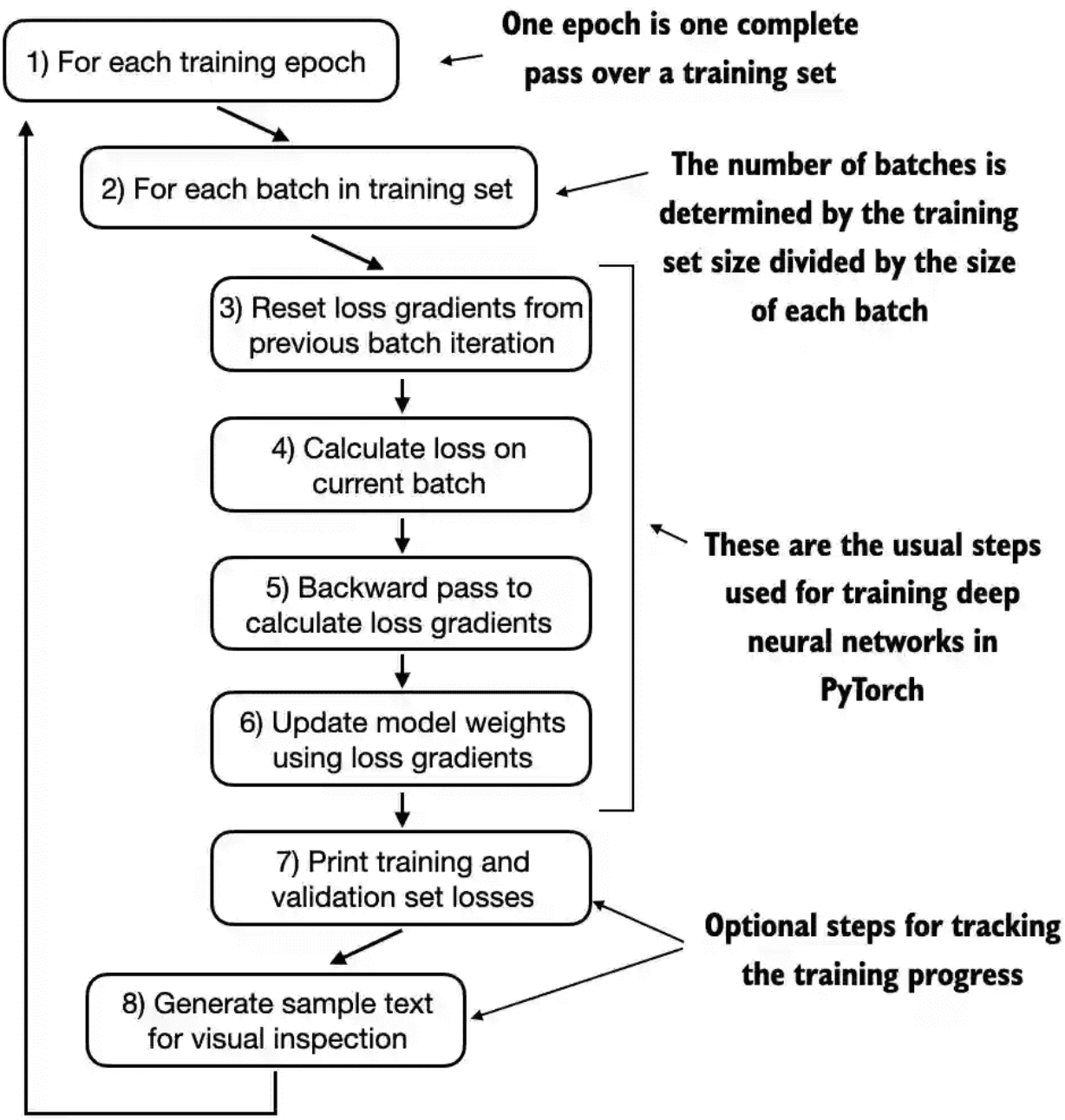
Training with variable lengths

```
def calc_loss_batch(input_batch, target_batch, model, device):
    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())
    return loss


def calc_loss_loader(data_loader, model, device, num_batches=None):
    total_loss = 0.
    if len(data_loader) == 0:
        return float("nan")
    elif num_batches is None:
        num_batches = len(data_loader)
    else:
        # Reduce the number of batches to match the total number of batches in the
data loader
        # if num_batches exceeds the number of batches in the data loader
        num_batches = min(num_batches, len(data_loader))
    for i, (input_batch, target_batch) in enumerate(data_loader):
        if i < num_batches:
            loss = calc_loss_batch(input_batch, target_batch, model, device)
            total_loss += loss.item()
        else:
            break
    return total_loss / num_batches


with torch.no_grad(): # Disable gradient tracking for efficiency because we are
not training, yet
    train_loss = calc_loss_loader(train_loader, model, device)
    val_loss = calc_loss_loader(val_loader, model, device)

print("Training loss:", train_loss)
print("Validation loss:", val_loss)
```



5.2 Training an LLM




```
def train_model_simple(model, train_loader, val_loader, optimizer, device,
num_epochs,
                       eval_freq, eval_iter, start_context, tokenizer):
    # Initialize lists to track losses and tokens seen
    train_losses, val_losses, track_tokens_seen = [], [], []
    tokens_seen, global_step = 0, -1

    # Main training loop
    for epoch in range(num_epochs):
        model.train()  # Set model to training mode

        for input_batch, target_batch in train_loader:
            optimizer.zero_grad() # Reset loss gradients from previous batch
iteration
            loss = calc_loss_batch(input_batch, target_batch, model, device)
```

```
            loss.backward() # Calculate loss gradients
            optimizer.step() # Update model weights using loss gradients
            tokens_seen += input_batch.numel()
            global_step += 1

            # Optional evaluation step
            if global_step % eval_freq == 0:
                train_loss, val_loss = evaluate_model(
                    model, train_loader, val_loader, device, eval_iter)
                train_losses.append(train_loss)
                val_losses.append(val_loss)
                track_tokens_seen.append(tokens_seen)
                print(f"Ep {epoch+1} (Step {global_step:06d}): "
                      f"Train loss {train_loss:.3f}, Val loss {val_loss:.3f}")

        # Print a sample text after each epoch
        generate_and_print_sample(
            model, tokenizer, device, start_context
        )

    return train_losses, val_losses, track_tokens_seen


def evaluate_model(model, train_loader, val_loader, device, eval_iter):
    model.eval()
    with torch.no_grad():
        train_loss = calc_loss_loader(train_loader, model, device,
num_batches=eval_iter)
        val_loss = calc_loss_loader(val_loader, model, device,
num_batches=eval_iter)
    model.train()
    return train_loss, val_loss


def generate_and_print_sample(model, tokenizer, device, start_context):
    model.eval()
    context_size = model.pos_emb.weight.shape[0]
    encoded = text_to_token_ids(start_context, tokenizer).to(device)
    with torch.no_grad():
        token_ids = generate_text_simple(
            model=model, idx=encoded,
            max_new_tokens=50, context_size=context_size
        )
    decoded_text = token_ids_to_text(token_ids, tokenizer)
    print(decoded_text.replace("\n", " "))  # Compact print format
    model.train()
```

AdamW

- Adam optimizers are a popular choice for training deep neural networks.
- However, in our training loop, we opt for the AdamW optimizer.

- AdamW is a variant of Adam that improves the weight decay approach, which aims to minimize model complexity and prevent overfitting by penalizing (处罚) larger weights.
- This adjustment allows AdamW to achieve more effective regularization and better generalization and is thus frequently used in the training of LLMs.
- Run

```
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.to(device)
optimizer = torch.optim.AdamW(model.parameters(), lr=0.0004, weight_decay=0.1)

num_epochs = 10
train_losses, val_losses, tokens_seen = train_model_simple(
    model, train_loader, val_loader, optimizer, device,
    num_epochs=num_epochs, eval_freq=5, eval_iter=5,
    start_context="Every effort moves you", tokenizer=tokenizer
)
```

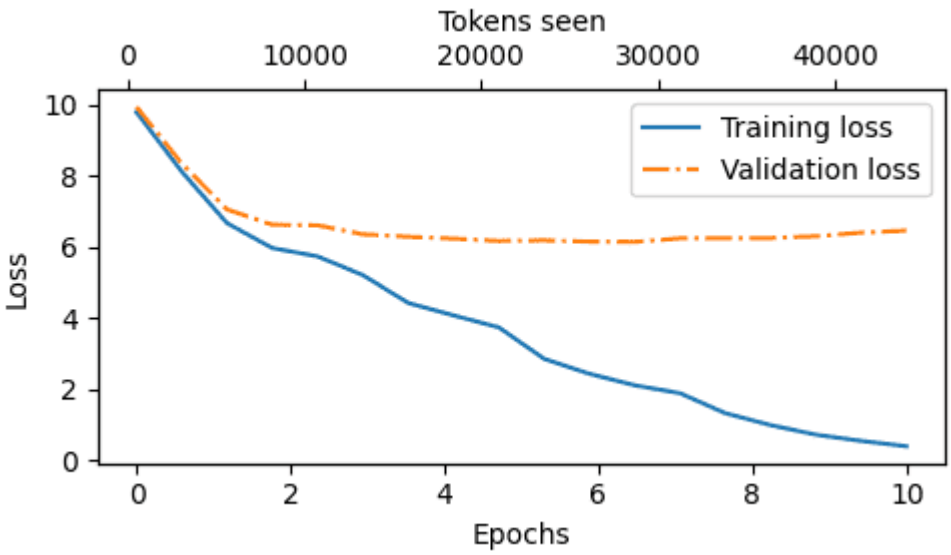
- Output:

```
Ep 1 (Step 000000): Train loss 9.781, Val loss 9.933
Ep 1 (Step 000005): Train loss 8.111, Val loss 8.339
Every effort moves you,,,,,,,,,,,,.
Ep 2 (Step 000010): Train loss 6.661, Val loss 7.048
Ep 2 (Step 000015): Train loss 5.961, Val loss 6.616
Every effort moves you, and, and, and, and, and, and, and, and, and, and, and,
and, and, and, and, and, and, and, and, and, and, and,, and, and,
... ...
Ep 9 (Step 000075): Train loss 0.717, Val loss 6.293
Ep 9 (Step 000080): Train loss 0.541, Val loss 6.393
Every effort moves you?&quot;  &quot;Yes--quite insensible to the irony. She
wanted him vindicated--and by me!&quot;  He laughed again, and threw back the
window-curtains, I had the donkey. &quot;There were days when I
Ep 10 (Step 000085): Train loss 0.391, Val loss 6.452
Every effort moves you know,&quot; was one of the axioms he laid down across the
Sevres and silver of an exquisitely appointed luncheon-table, when, on a later
day, I had again run over from Monte Carlo; and Mrs. Gis
```

simple plot

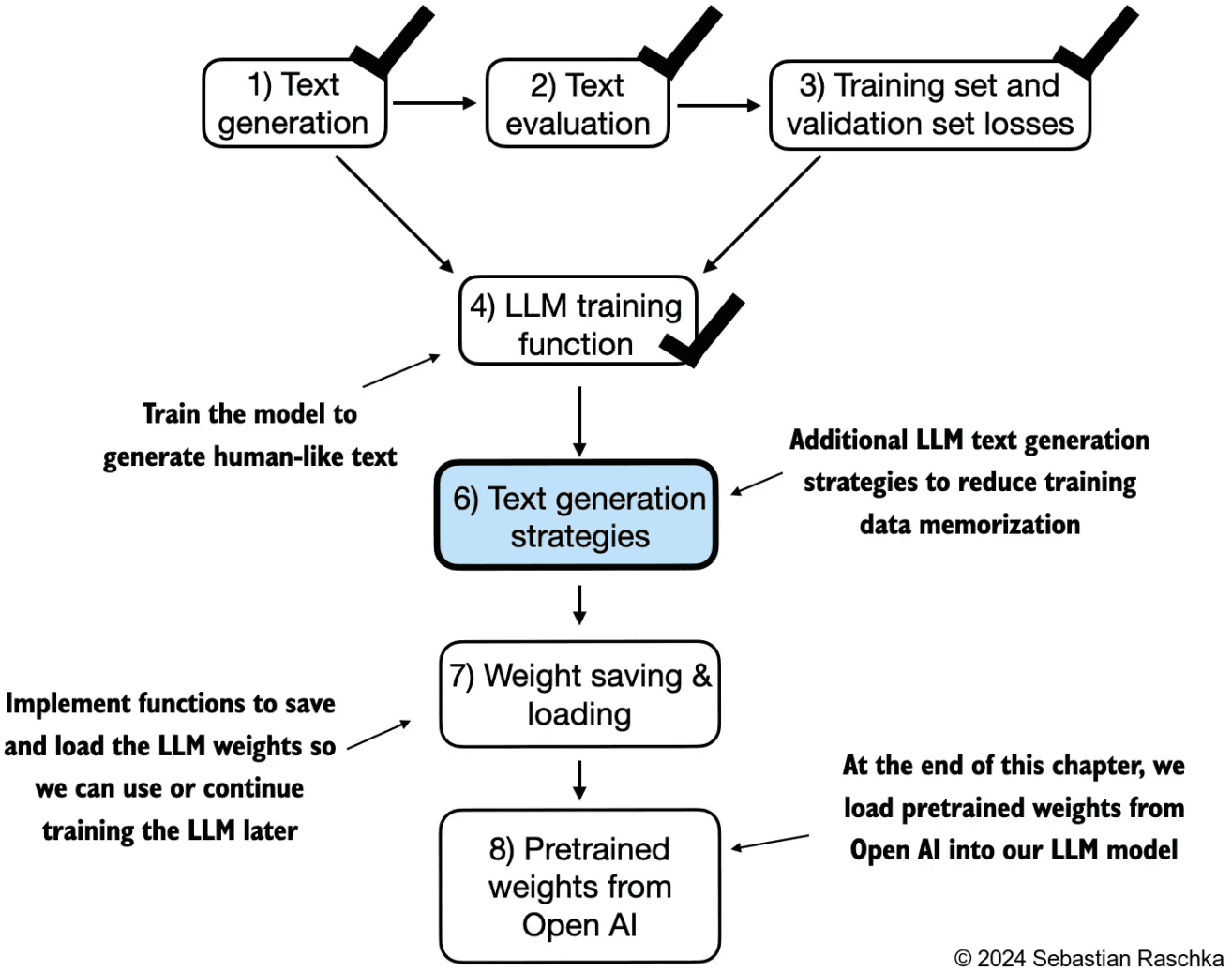
```
import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator


def plot_losses(epochs_seen, tokens_seen, train_losses, val_losses):
    fig, ax1 = plt.subplots(figsize=(5, 3))
```

```
    # Plot training and validation loss against epochs
    ax1.plot(epochs_seen, train_losses, label="Training loss")
    ax1.plot(epochs_seen, val_losses, linestyle="-.", label="Validation loss")
    ax1.set_xlabel("Epochs")
    ax1.set_ylabel("Loss")
    ax1.legend(loc="upper right")
    ax1.xaxis.set_major_locator(MaxNLocator(integer=True))  # only show integer
labels on x-axis

    # Create a second x-axis for tokens seen
    ax2 = ax1.twiny()  # Create a second x-axis that shares the same y-axis
    ax2.plot(tokens_seen, train_losses, alpha=0)  # Invisible plot for aligning
ticks
    ax2.set_xlabel("Tokens seen")

    fig.tight_layout()  # Adjust layout to make room
    plt.savefig("loss-plot.pdf")
    plt.show()

epochs_tensor = torch.linspace(0, num_epochs, len(train_losses))
plot_losses(epochs_tensor, tokens_seen, train_losses, val_losses)
```



Summary

- Looking at the results above, we can see that the model starts out generating incomprehensible strings of words, whereas towards the end, it's able to produce grammatically more or less correct sentences.
- However, based on the training and validation set losses, we can see that the model starts overfitting (the final training set loss is 0.541, while the validation set loss is 6.393).
- If we were to check a few passages it writes towards the end, we would find that they are contained in the training set verbatim (逐字翻译) – it simply memorizes the training data.
- Later, we will cover decoding strategies that can mitigate this memorization by a certain degree.
- Note that the overfitting here occurs because we have a very, very small training set, and we iterate over it so many times.



5.3 Decoding strategies to control randomness

- This section will re-implement `generate_text_simple()` in `chapter 4.7` but we will cover two techniques, temperature scaling, and top-k sampling, to improve this function.
- The previously implemented version used greedy decoding, where the returned result for each request was always the one with the highest probability, leading to identical results for repeated requests for generative tasks, resulting in monotonous output.

Small vocabulary for illustration purposes:

```
vocab = {
    "closer": 0,
    "every": 1,
    "effort": 2,
    "forward": 3,
    "inches": 4,
    "moves": 5,
    "pizza": 6,
    "toward": 7,
    "you": 8,
}
inverse_vocab = {v: k for k, v in vocab.items()}
```

Greedy decoding from the previous section:

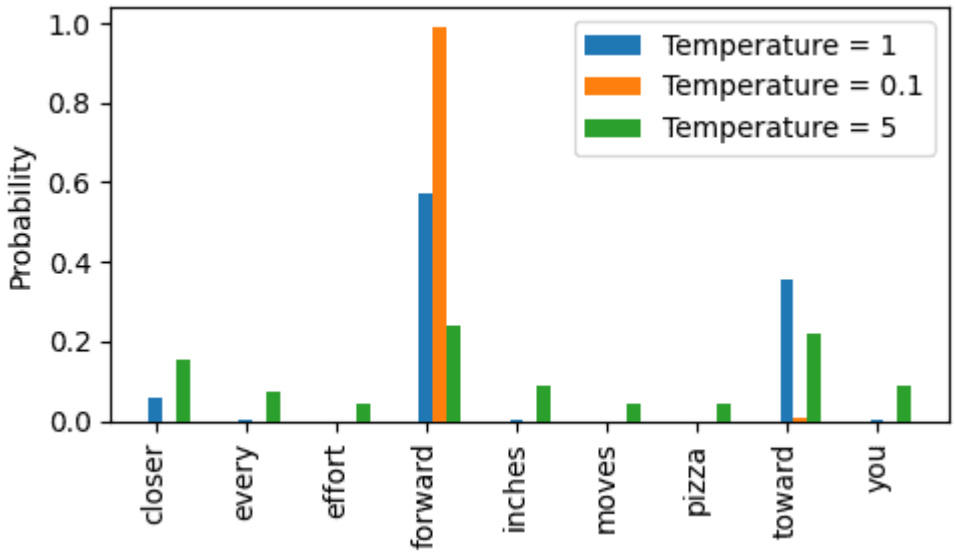
```
next_token_logits = torch.tensor(
    [4.51, 0.89, -1.90, 6.75, 1.63, -1.62, -1.89, 6.28, 1.79]
)
probas = torch.softmax(next_token_logits, dim=0)
next_token_id = torch.argmax(probas).item()
print(inverse_vocab[next_token_id])
# "forward"
# Explanation:
#   By analyzing next_token_logits, we know
#   the maximum probability is 6.75, corresponding to the max value after softmax.
#   So the index of the next token is 3, resulting in the final output: forward
```

Corresponding probabilistic sampling process:

```
# replace the argmax with the multinomial function in PyTorch
torch.manual_seed(123)
next_token_id = torch.multinomial(probas, num_samples=1).item()
print(inverse_vocab[next_token_id])
# Explanation: Regardless of the probability, each token has a chance to be
selected.
```

Effect of temperature scaling:

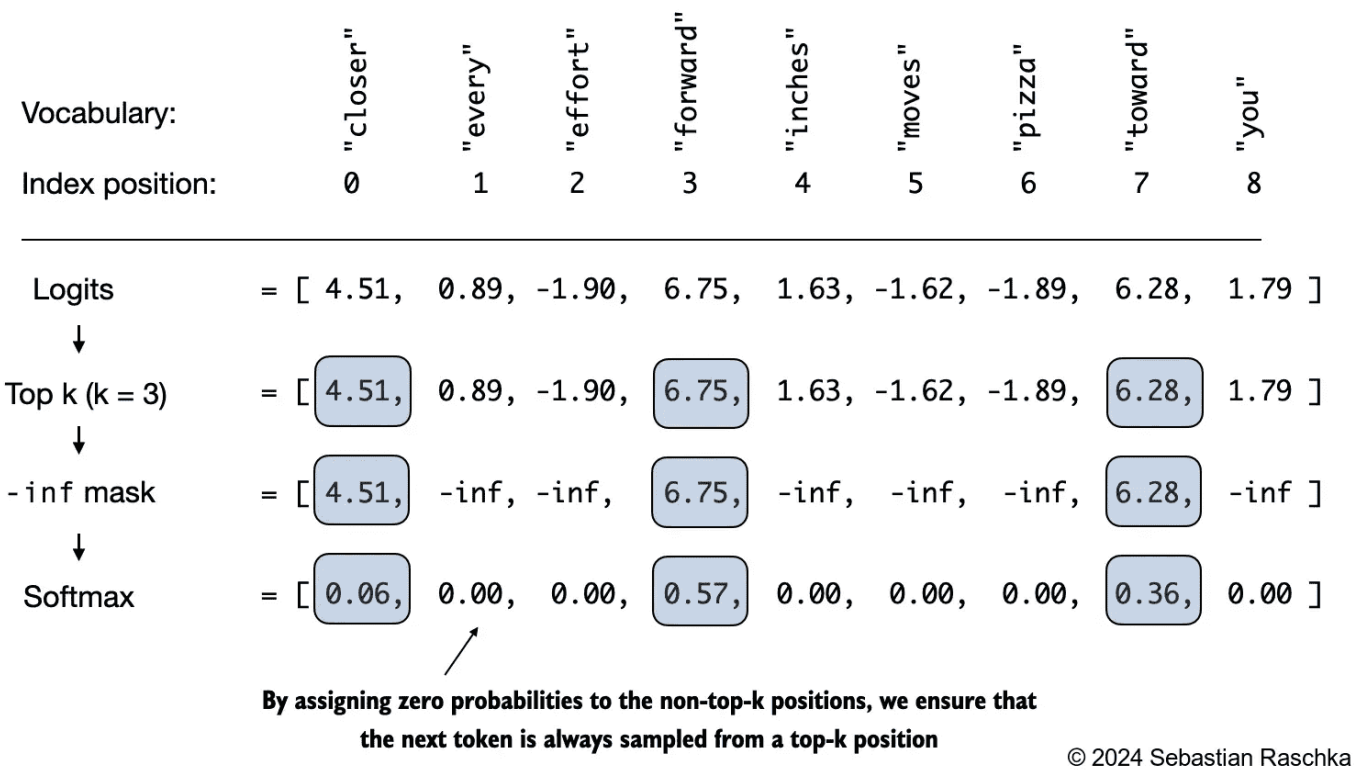
```
# Temperatures greater than 1 result in more uniformly distributed token
probabilities
# Temperatures smaller than 1 will result in more confident (sharper or more
peaky) distributions.
def softmax_with_temperature(logits, temperature):
    scaled_logits = logits / temperature
    return torch.softmax(scaled_logits, dim=0)
```



5.3.2 Top-k sampling

- The method from the previous section allows for exploring less likely but potentially more interesting and creative paths in the generation process. However, One downside of this approach is that it sometimes leads to grammatically incorrect or completely nonsensical outputs (i.e., very low probability tokens might be selected, which are typically unsuitable as next tokens, resulting in meaningless sentences).

> **Note:** In top-k sampling, we can restrict the sampled tokens to the top-k most likely tokens and exclude all other tokens from the selection process by masking their probability scores.




```
top_k = 3
top_logits, top_pos = torch.topk(next_token_logits, top_k)
new_logits = torch.where(
```

```
    condition=next_token_logits < top_logits[-1],
    input=torch.tensor(float("-inf")),
    other=next_token_logits
)
topk_probas = torch.softmax(new_logits, dim=0)

# More efficient implementation
# > new_logits = torch.full_like( # create tensor containing -inf values
# >    next_token_logits, -torch.inf
# > )
# > new_logits[top_pos] = next_token_logits[top_pos] # copy top k values into the
-inf tensor
# > topk_probas = torch.softmax(new_logits, dim=0)
```

- Top-p (also known as nuclear sampling): The algorithm first sorts all words by their probability and then accumulates these probabilities until the sum first reaches or exceeds the p value. Subsequently, it samples the next word randomly only from this set of words whose accumulated probability reached p, ignoring the words with lower probabilities. For example, if top_p=0.9 is set, the smallest set of words whose cumulative probability is at least 90% is selected, and the next word is randomly sampled from this set.

> **Note:** When both top_k and top_p are set: top_k is executed first: select the top k words with the highest probability from the model's predicted probability distribution. Then apply top_p: within the set of words filtered by top_k, further filter to the smallest set whose cumulative probability reaches or exceeds the p value. This means that even if a word is within the top_k range, it may not be considered if its cumulative probability exceeds the set p value.

5.3.3 Modifying the text generation function

- This section combines the temperature scaling and top-k sampling methods from the previous two sections into the generator function.

```
def generate(model, idx, max_new_tokens, context_size, temperature=0.0,
top_k=None, eos_id=None):
    # For-loop is the same as before: Get logits, and only focus on last time step
    for _ in range(max_new_tokens):
        idx_cond = idx[:, -context_size:]
        with torch.no_grad():
            logits = model(idx_cond)
        logits = logits[:, -1, :]

        # New: Filter logits with top_k sampling
        if top_k is not None:
            # Keep only top_k values
            top_logits, _ = torch.topk(logits, top_k)
            min_val = top_logits[:, -1]
            logits = torch.where(logits < min_val, torch.tensor(float("-
inf")).to(logits.device), logits)

        # New: Apply temperature scaling
```

```
        if temperature > 0.0:
            logits = logits / temperature

            # Apply softmax to get probabilities
            probs = torch.softmax(logits, dim=-1)  # (batch_size, context_len)

            # Sample from the distribution
            idx_next = torch.multinomial(probs, num_samples=1)  # (batch_size, 1)

        # Otherwise same as before: get idx of the vocab entry with the highest
logits value
        else:
            idx_next = torch.argmax(logits, dim=-1, keepdim=True)  # (batch_size,
1)

        if idx_next == eos_id:  # Stop generating early if end-of-sequence token
is encountered and eos_id is specified
            break

        # Same as before: append sampled index to the running sequence
        idx = torch.cat((idx, idx_next), dim=1)  # (batch_size, num_tokens+1)

    return idx
```

5.4 Loading and saving model weights in PyTorch

- It's common to train LLMs with adaptive optimizers like Adam or AdamW instead of regular SGD.
- These adaptive optimizers store additional parameters for each model weight, so it makes sense to save them as well in case we plan to continue the pretraining later:

```
torch.save({
    "model_state_dict": model.state_dict(),
    "optimizer_state_dict": optimizer.state_dict(),
    },
    "model_and_optimizer.pth"
)
```

Loading:

```
checkpoint = torch.load("model_and_optimizer.pth", weights_only=True)

model = GPTModel(GPT_CONFIG_124M)
model.load_state_dict(checkpoint["model_state_dict"])

optimizer = torch.optim.AdamW(model.parameters(), lr=0.0005, weight_decay=0.1)
optimizer.load_state_dict(checkpoint["optimizer_state_dict"])
model.train();
```

5.5 Loading pretrained weights from OpenAI

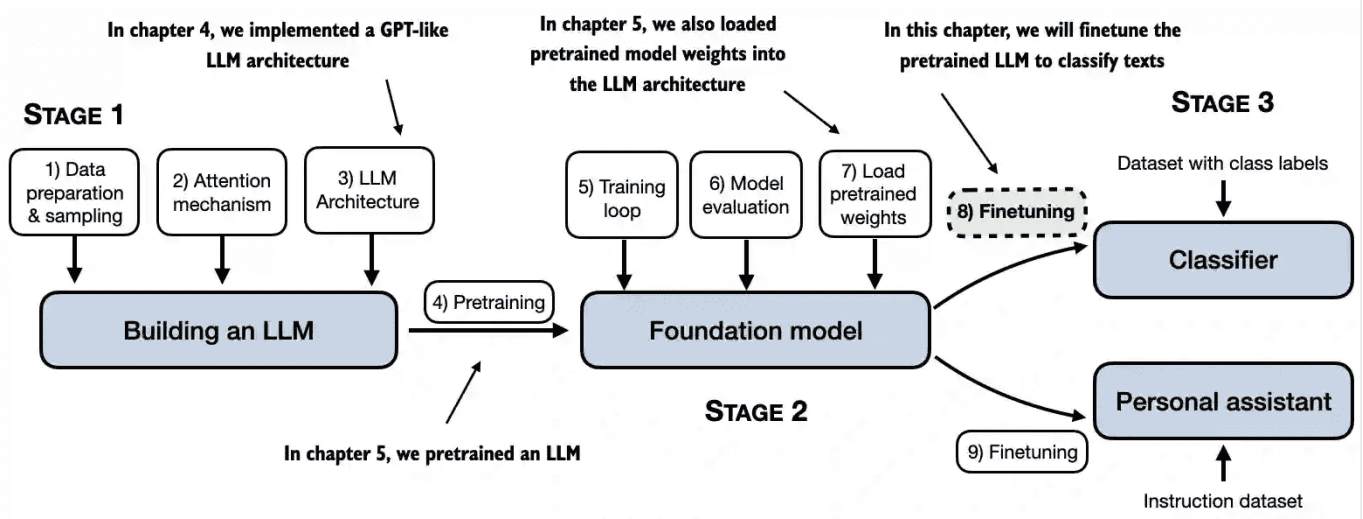
- This section mainly discusses how to load weights from GPT-2.
- Since the data structure names for GPT-2 and our definition here are not the same, custom loading is required.

5.6 Summary

- When LLMs generate text, they output one token at a time.
- By default, the next token is generated by converting the model outputs into probability scores and selecting the token from the vocabulary that corresponds to the highest probability score, which is known as “greedy decoding.”
- Using probabilistic sampling and temperature scaling, we can influence the diversity and coherence of the generated text.
- Training and validation set losses can be used to gauge (测量) the quality of text generated by LLM during training.
- Pretraining an LLM involves changing its weights to minimize the training loss.
- The training loop for LLMs itself is a standard procedure in deep learning, using a conventional `cross entropy loss` and `AdamW optimizer`. Pretraining an LLM on a large text corpus is time- and resource-intensive so we can load openly available weights from OpenAI as an alternative to pretraining the model on a large dataset ourselves.

6 Fine-tuning for classification

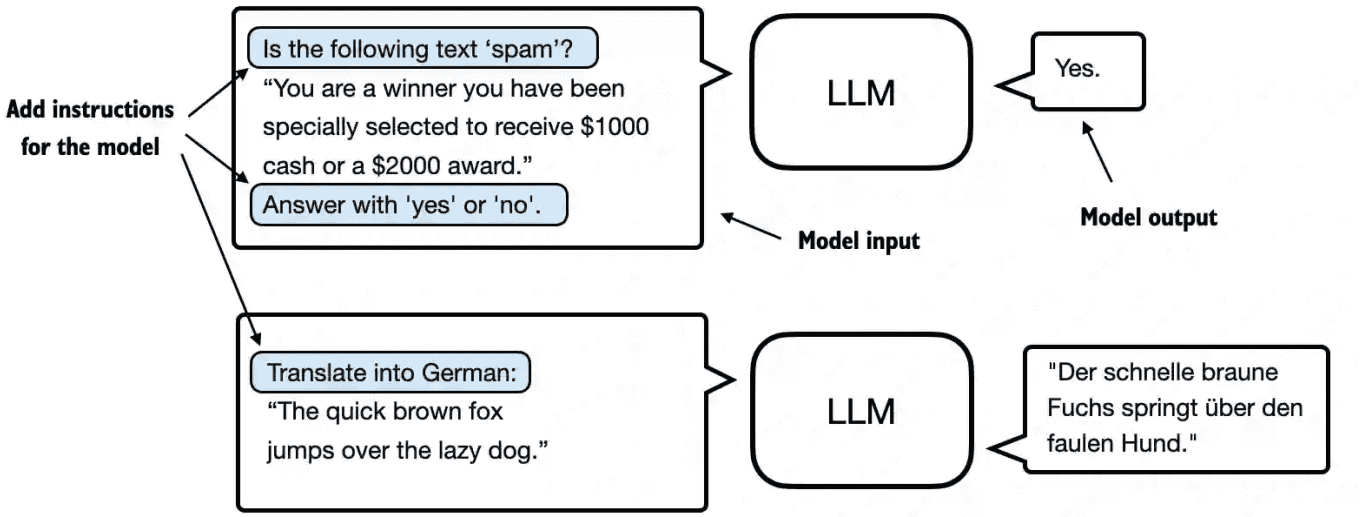
- Introducing different LLM fine-tuning approaches
- Preparing a dataset for text classification
- Modifying a pretrained LLM for fine-tuning
- Fine-tuning an LLM to identify spam messages
- Evaluating the accuracy of a fine-tuned LLM classifier
- Using a fine-tuned LLM to classify new data



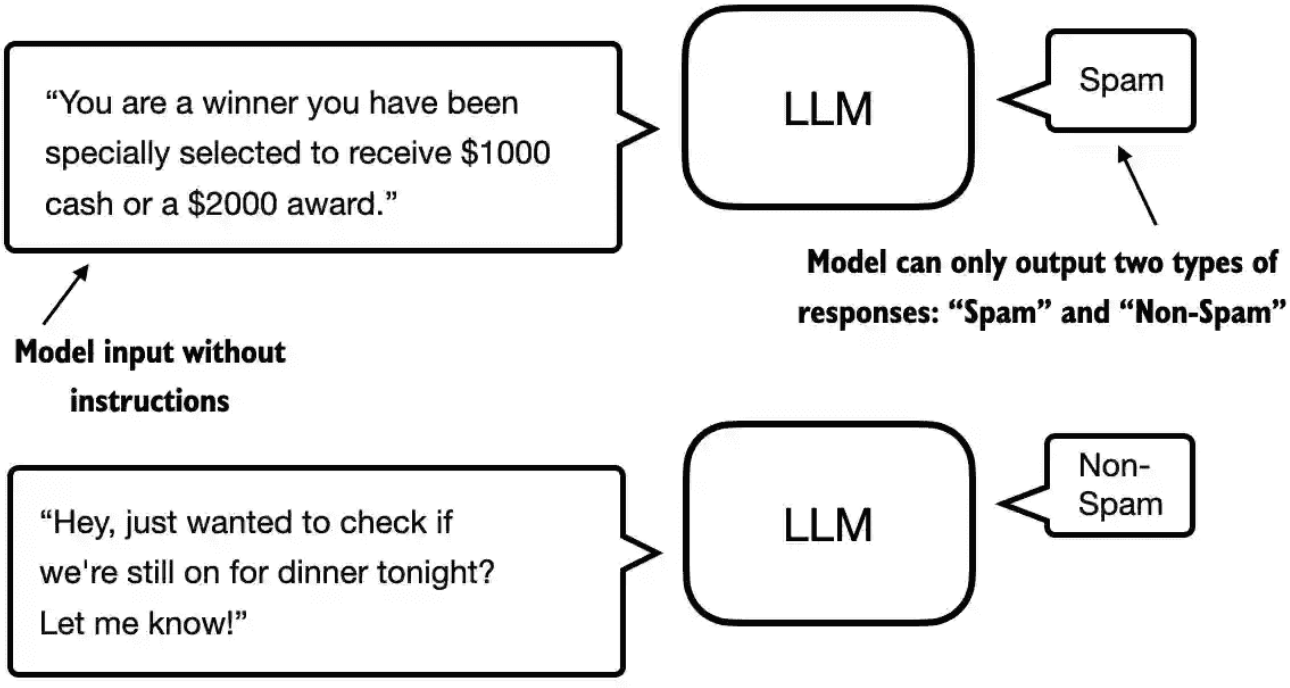
6.1 Different categories of fine-tuning

- The most common ways to fine-tune language models are `instruction fine-tuning` and `classification fine-tuning`.

- `Instruction fine-tuning` involves training a language model on a set of tasks using specific instructions to improve its ability to understand and execute tasks described in natural language prompts.



> **Note:** Instruction-finetuning, depicted below, is the topic of the next chapter. This section mainly discusses text classification.



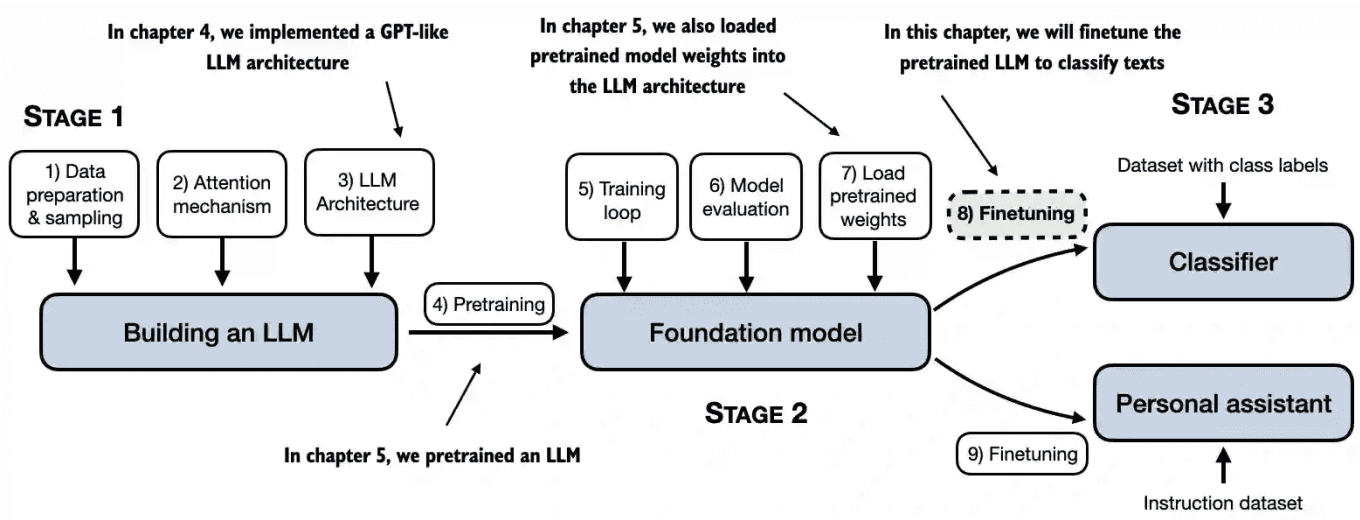
- Classification finetuning, the topic of this chapter, is a procedure you may already be familiar with if you have a background in machine learning – it's similar to training a convolutional network to classify handwritten digits, for example.
- In classification finetuning, we have a specific number of class labels (for example, "spam" and "not spam") that the model can output.
- A classification finetuned model can only predict classes it has seen during training (for example, "spam" or "not spam"), whereas an instruction-finetuned model can usually perform many tasks.

> **Note:** We can think of a classification-finetuned model as a very specialized model; in practice, it is much easier to create a specialized model than a generalist model that performs well on many different tasks.

CHOOSING THE RIGHT APPROACH

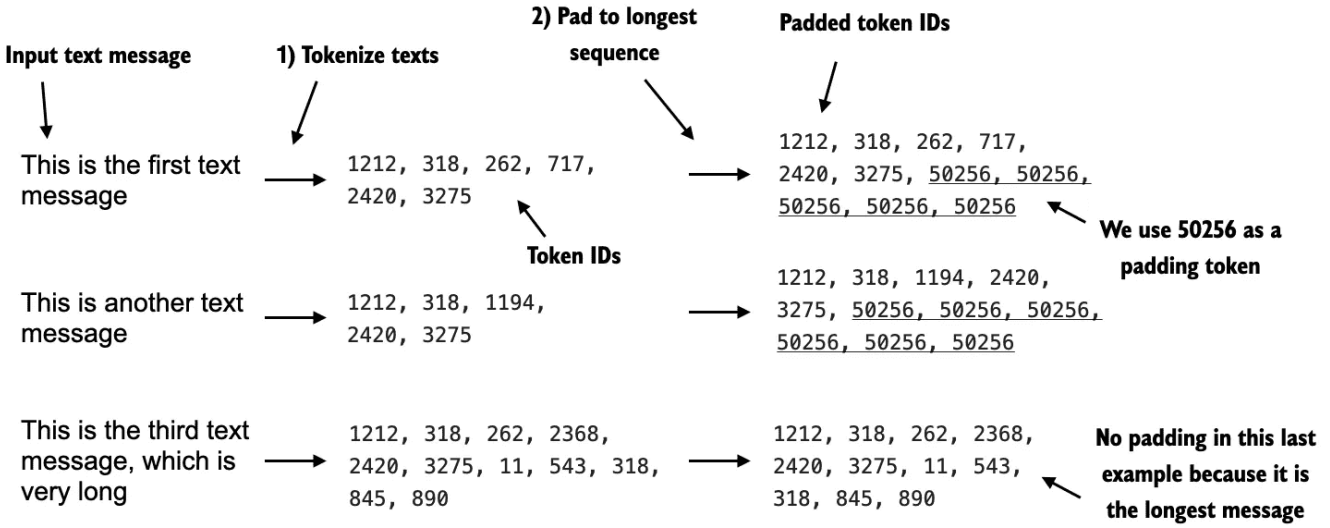
- Instruction fine-tuning improves a model's ability to understand and generate responses based on specific user instructions. Instruction fine-tuning is best suited for models that need to handle a variety of tasks based on complex user instructions, improving flexibility and interaction quality. Classification fine-tuning is ideal for projects requiring precise categorization of data into predefined classes, such as sentiment analysis or spam detection.
- While instruction fine-tuning is more versatile, it demands larger datasets and greater computational resources to develop models proficient in various tasks. In contrast, classification fine-tuning requires less data and compute power, but its use is confined to the specific classes on which the model has been trained.

6.2 Preparing the dataset




```
url = "https://archive.ics.uci.edu/static/public/228/sms+spam+collection.zip"
# =>
# Extract 3 datasets:
# train_df.to_csv("train.csv", index=None)
# validation_df.to_csv("validation.csv", index=None)
# test_df.to_csv("test.csv", index=None)
```

6.3 Creating data loaders




```python
import torch
from torch.utils.data import Dataset

class SpamDataset(Dataset):
    def __init__(self, csv_file, tokenizer, max_length=None, pad_token_id=50256):
        self.data = pd.read_csv(csv_file)

        # Pre-tokenize texts
        self.encoded_texts = [
            tokenizer.encode(text) for text in self.data["Text"]
        ]

        if max_length is None:
            self.max_length = self._longest_encoded_length()
        else:
            self.max_length = max_length
            # Truncate sequences if they are longer than max_length
            self.encoded_texts = [
                encoded_text[:self.max_length]
                for encoded_text in self.encoded_texts
            ]

        # Pad sequences to the longest sequence
        self.encoded_texts = [
            encoded_text + [pad_token_id] * (self.max_length - len(encoded_text))
            for encoded_text in self.encoded_texts
        ]

    def __getitem__(self, index):
        encoded = self.encoded_texts[index]
        label = self.data.iloc[index]["Label"]
        return (
            torch.tensor(encoded, dtype=torch.long),
            torch.tensor(label, dtype=torch.long)
        )

    def __len__(self):
```

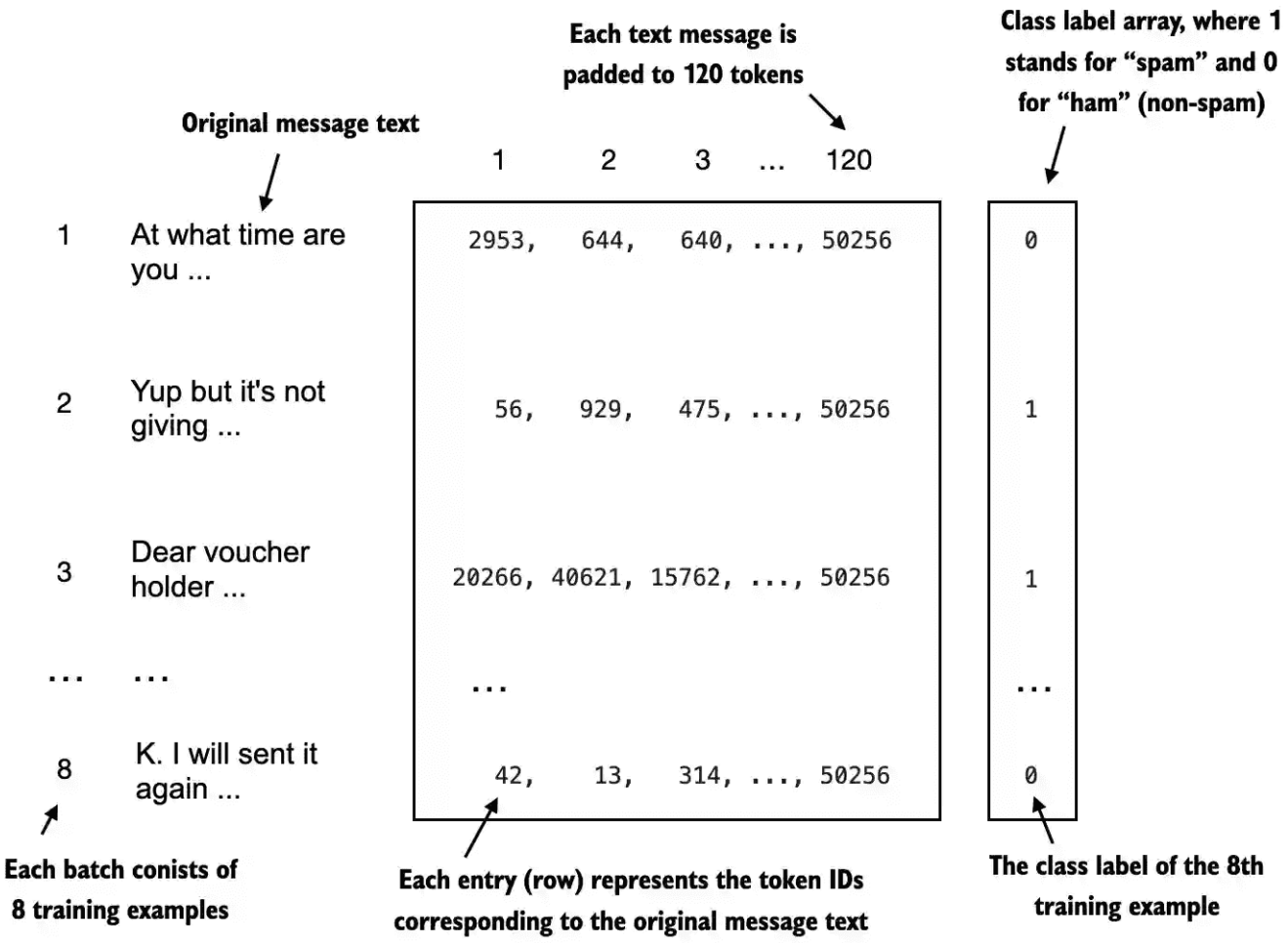
```
        return len(self.data)

    def _longest_encoded_length(self):
        max_length = 0
        for encoded_text in self.encoded_texts:
            encoded_length = len(encoded_text)
            if encoded_length > max_length:
                max_length = encoded_length
        return max_length
```

Usage:

```
train_dataset = SpamDataset(
    csv_file="train.csv",
    max_length=None,
    tokenizer=tokenizer
)
```

For validation and test sets, max_length can be specified (or left as None, like the training set):

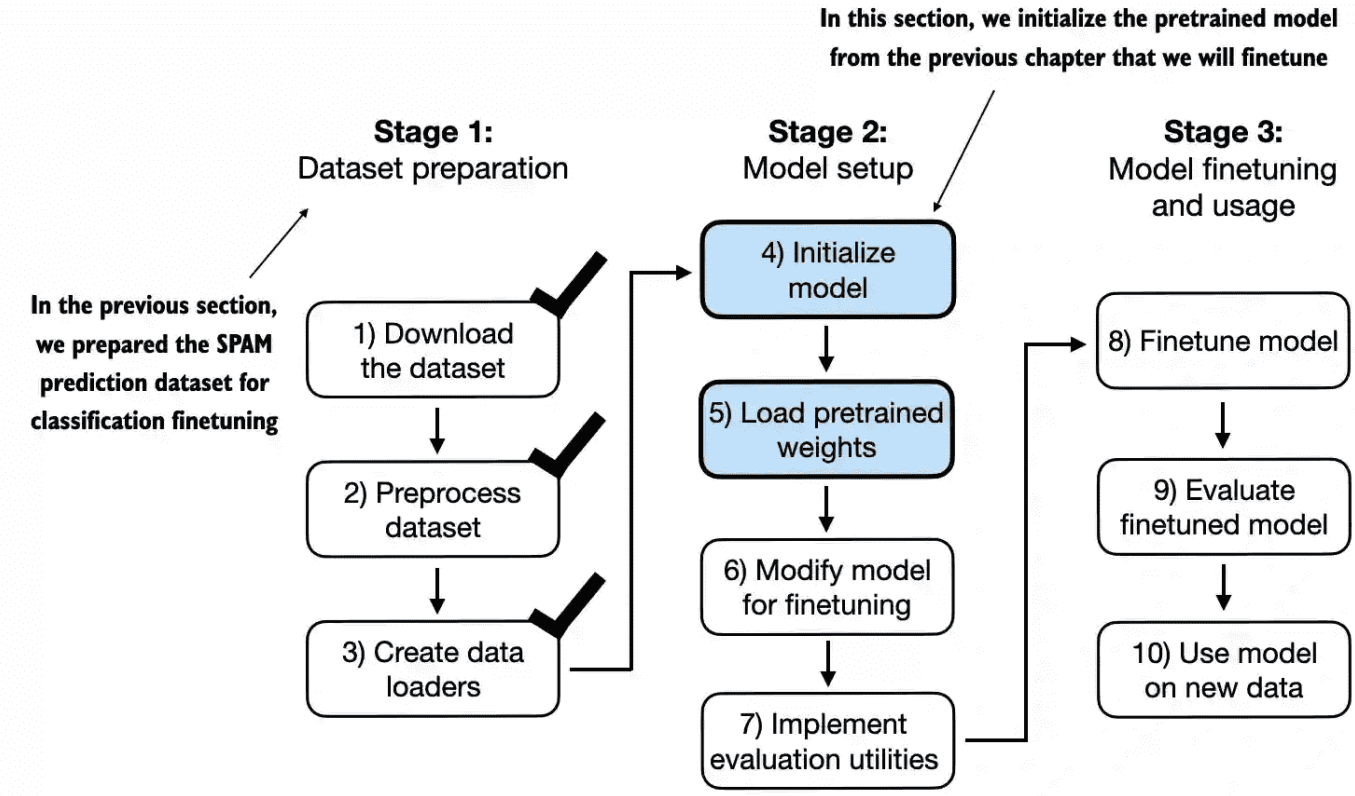
```
val_dataset = SpamDataset(
    csv_file="validation.csv",
    max_length=train_dataset.max_length,
    tokenizer=tokenizer
)
test_dataset = SpamDataset(
    csv_file="test.csv",
    max_length=train_dataset.max_length,
    tokenizer=tokenizer
)
```



Data loaders (using training set as an example):

```
from torch.utils.data import DataLoader

num_workers = 0
batch_size = 8
train_loader = DataLoader(
    dataset=train_dataset,
    batch_size=batch_size,
    shuffle=True,
    num_workers=num_workers,
    drop_last=True,
)
```

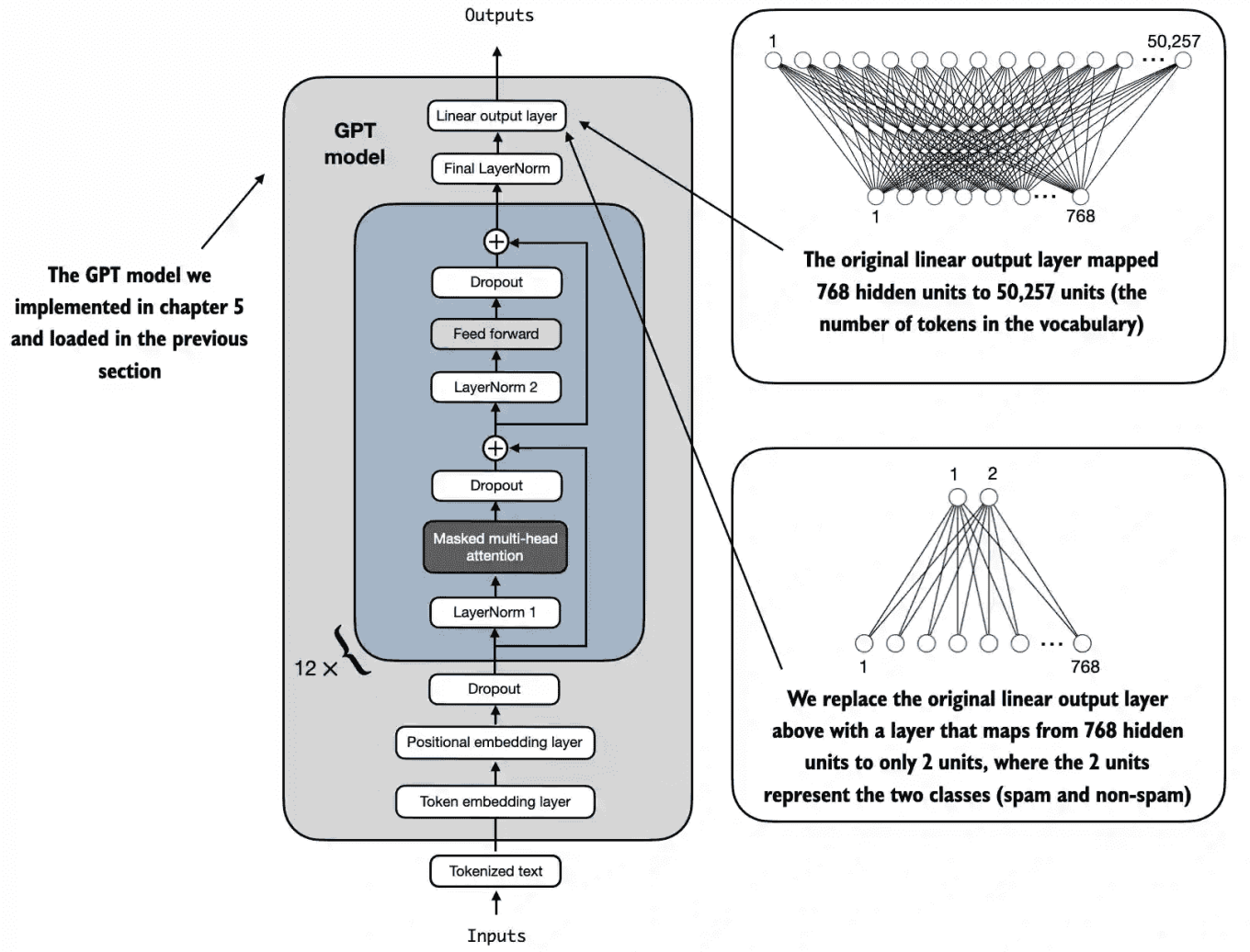
6.4 Initializing a model with pretrained weights



- Initialize the model and load GPT-2 weights using the methods from previous chapters:

```
# ... ...
model = GPTModel(BASE_CONFIG)
load_weights_into_gpt(model, params)
model.eval();
```

6.5 Adding a classification head



OUTPUT LAYER NODES

- Print the model architecture via `print(model)`.

```
GPTModel(
  (tok_emb): Embedding(50257, 768)
  (pos_emb): Embedding(1024, 768)
  (drop_emb): Dropout(p=0.0, inplace=False)
  (trf_blocks): Sequential(
    (0): TransformerBlock(
      (att): MultiHeadAttention(
        (W_query): Linear(in_features=768, out_features=768, bias=True)
        (W_key): Linear(in_features=768, out_features=768, bias=True)
        (W_value): Linear(in_features=768, out_features=768, bias=True)
        (out_proj): Linear(in_features=768, out_features=768, bias=True)
        (dropout): Dropout(p=0.0, inplace=False)
      )
      (ff): FeedForward(
        (layers): Sequential(
          (0): Linear(in_features=768, out_features=3072, bias=True)
          (1): GELU()
          (2): Linear(in_features=3072, out_features=768, bias=True)
        )
```

```
      )
      (norm1): LayerNorm()
      (norm2): LayerNorm()
      (drop_resid): Dropout(p=0.0, inplace=False)
    )
   ... ... # 11 skipped
  )
  (final_norm): LayerNorm()
  (out_head): Linear(in_features=768, out_features=50257, bias=False)
)
```

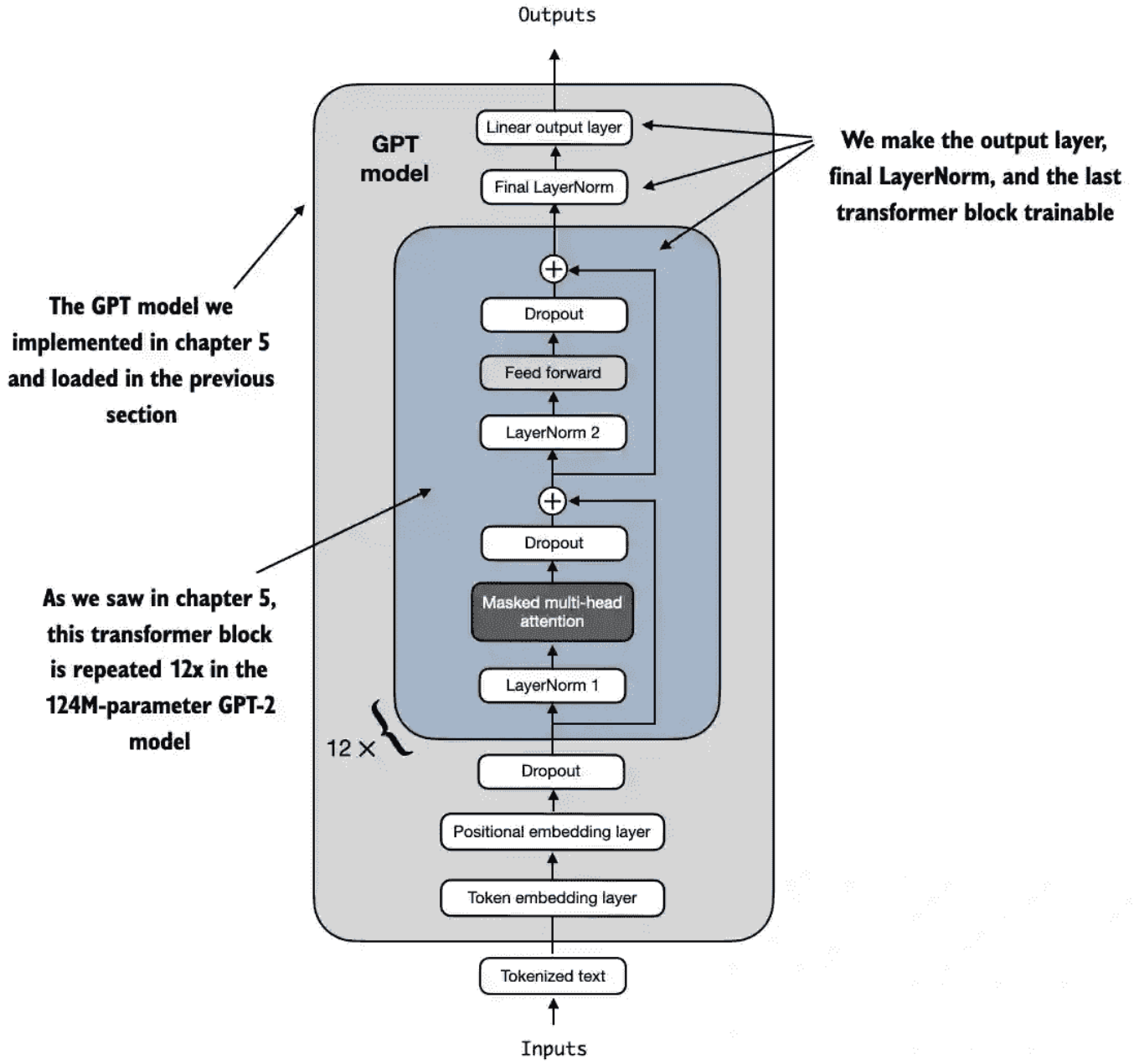
FINE-TUNING SELECTED LAYERS VS. ALL LAYERS

- The goal is to replace and finetune the output layer.
- To achieve this, we first freeze the model, meaning that we make all layers non-trainable:

```
for param in model.parameters():
    param.requires_grad = False
```

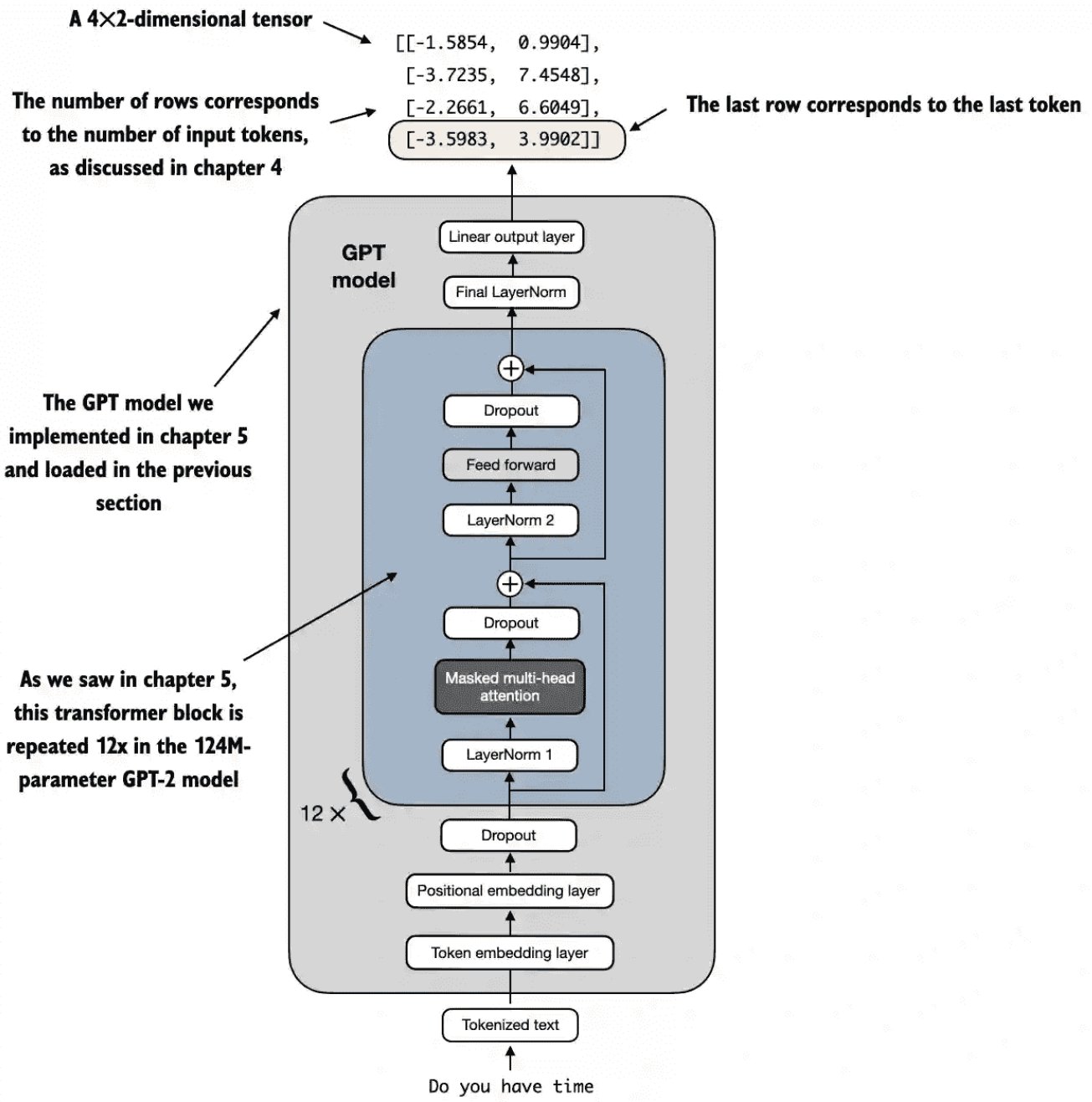
Then, we replace the output layer (`model.out_head`):

```
# finetune the model for binary classification (predicting 2 classes, "spam" and
"not spam")
torch.manual_seed(123)
num_classes = 2
model.out_head = torch.nn.Linear(in_features=BASE_CONFIG["emb_dim"],
out_features=num_classes)
```

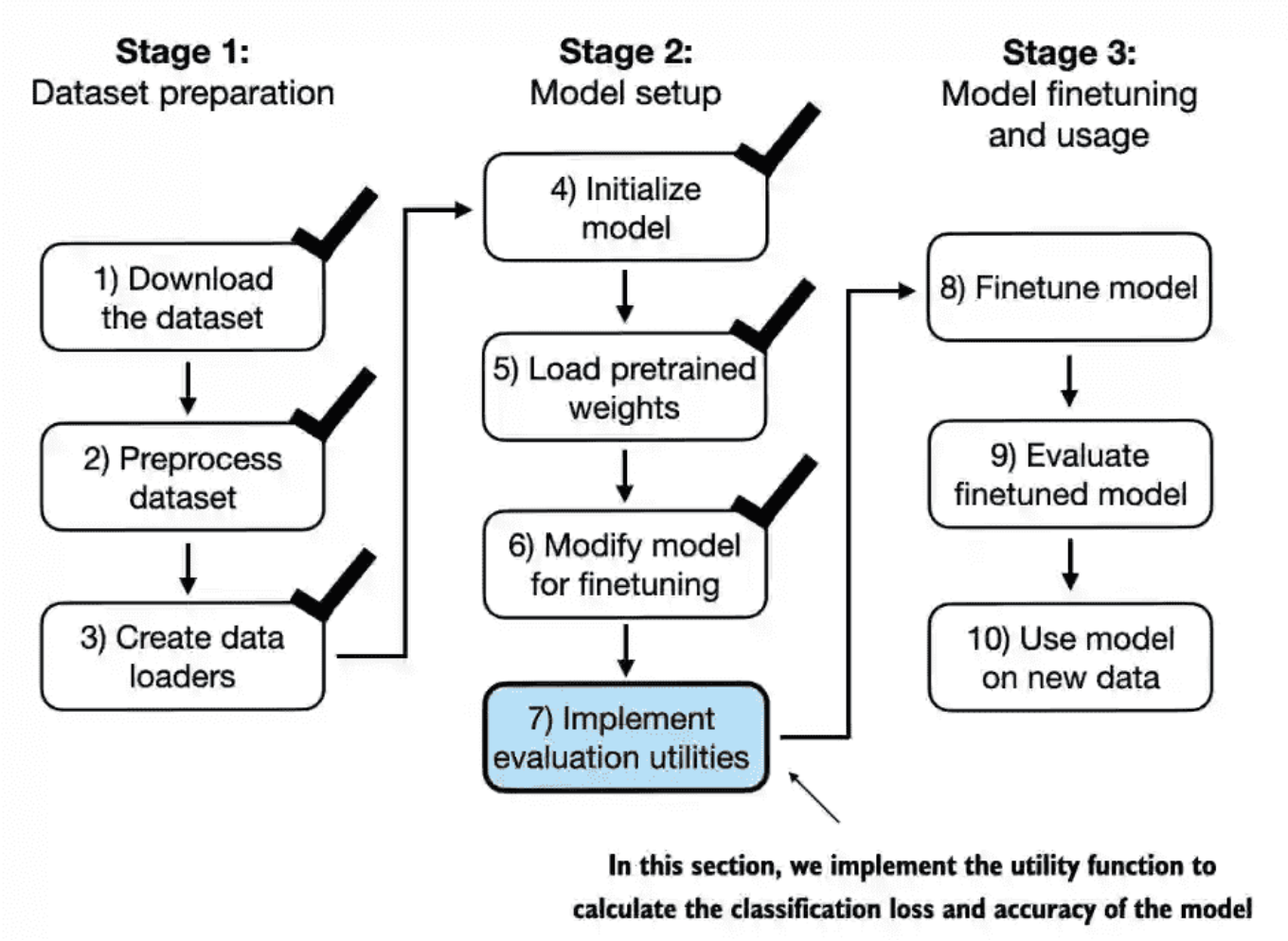


- Technically, it's sufficient to only train the output layer.
- However, as I found in Finetuning Large Language Models, experiments show that finetuning additional layers can noticeably improve the performance.
- So, we are also making the last transformer block and the final `LayerNorm` module connecting the last transformer block to the output layer trainable.

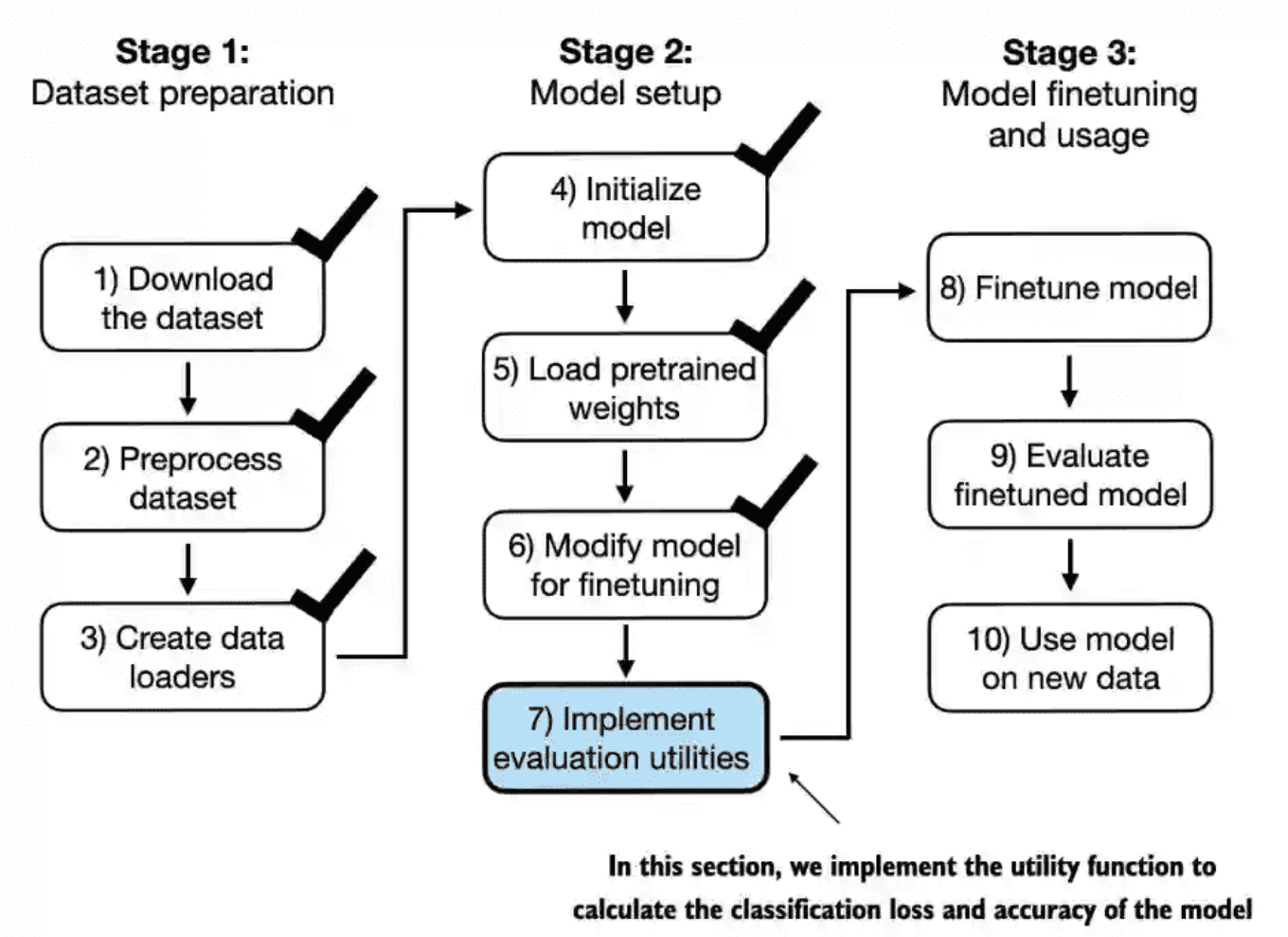
```
for param in model.trf_blocks[-1].parameters():
    param.requires_grad = True

for param in model.final_norm.parameters():
    param.requires_grad = True
```



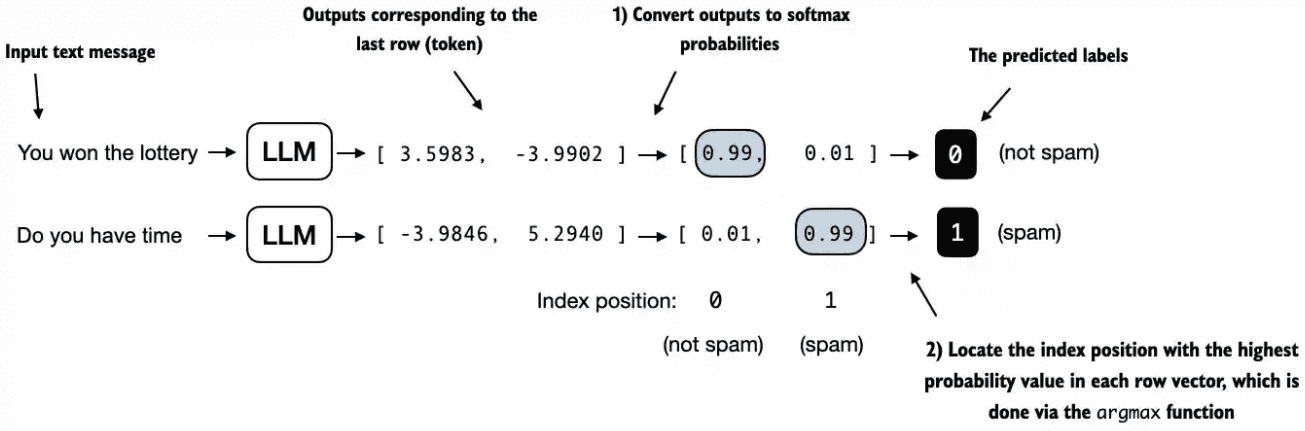
- In chapter 3, we discussed the attention mechanism, which connects each input token to each other input token.
- In chapter 3, we then also introduced the causal attention mask that is used in GPT-like models; this causal mask lets a current token only attend to the current and previous token positions.
- Based on this causal attention mechanism, the 4th (last) token contains the most information among all tokens because it's the only token that includes information about all other tokens.
- Hence, we are particularly interested in this last token, which we will finetune for the spam classification task.



6.6 Calculating the classification loss and accuracy



- We previously computed the token ID of the next token generated by the LLM by converting the 50,257 outputs into probabilities via the softmax function and then returning the position of the highest probability via the argmax function.
- We take the same approach here to calculate whether the model outputs a "spam" or "not spam" prediction for a given input, as shown in figure 6.14.
- The only difference is that we work with 2-dimensional instead of 50,257-dimensional outputs.



Note that the softmax function is optional here:

```
logits = outputs[:, -1, :]
label = torch.argmax(logits)
```

```
print("Class label:", label.item())
```

- Compute the classification accuracy, which measures the percentage of correct predictions across a dataset.

```
def calc_accuracy_loader(data_loader, model, device, num_batches=None):
    model.eval()
    correct_predictions, num_examples = 0, 0

    if num_batches is None:
        num_batches = len(data_loader)
    else:
        num_batches = min(num_batches, len(data_loader))
    for i, (input_batch, target_batch) in enumerate(data_loader):
        if i < num_batches:
            input_batch, target_batch = input_batch.to(device),
target_batch.to(device)

            with torch.no_grad():
                logits = model(input_batch)[:, -1, :]  # Logits of last output
token
            predicted_labels = torch.argmax(logits, dim=-1)

            num_examples += predicted_labels.shape[0]
            correct_predictions += (predicted_labels == target_batch).sum().item()
        else:
            break
    return correct_predictions / num_examples
```

Use the function to determine the classification accuracies across various datasets:

```
train_accuracy = calc_accuracy_loader(train_loader, model, device, num_batches=10)
print(f"Training accuracy: {train_accuracy*100:.2f}%")

# ... # Same for validation and test sets
```

- Because classification accuracy is not a differentiable function, we use cross-entropy loss as a proxy to maximize accuracy.
- Accordingly, the `calc_loss_batch` function remains the same, with one adjustment: we focus on optimizing only the last token, `model(input_batch)[:, -1, :]`, rather than all tokens, `model(input_batch)`.

```
def calc_loss_batch(input_batch, target_batch, model, device):
    input_batch, target_batch = input_batch.to(device), target_batch.to(device)
    logits = model(input_batch)[:, -1, :]  # Logits of last output token
```

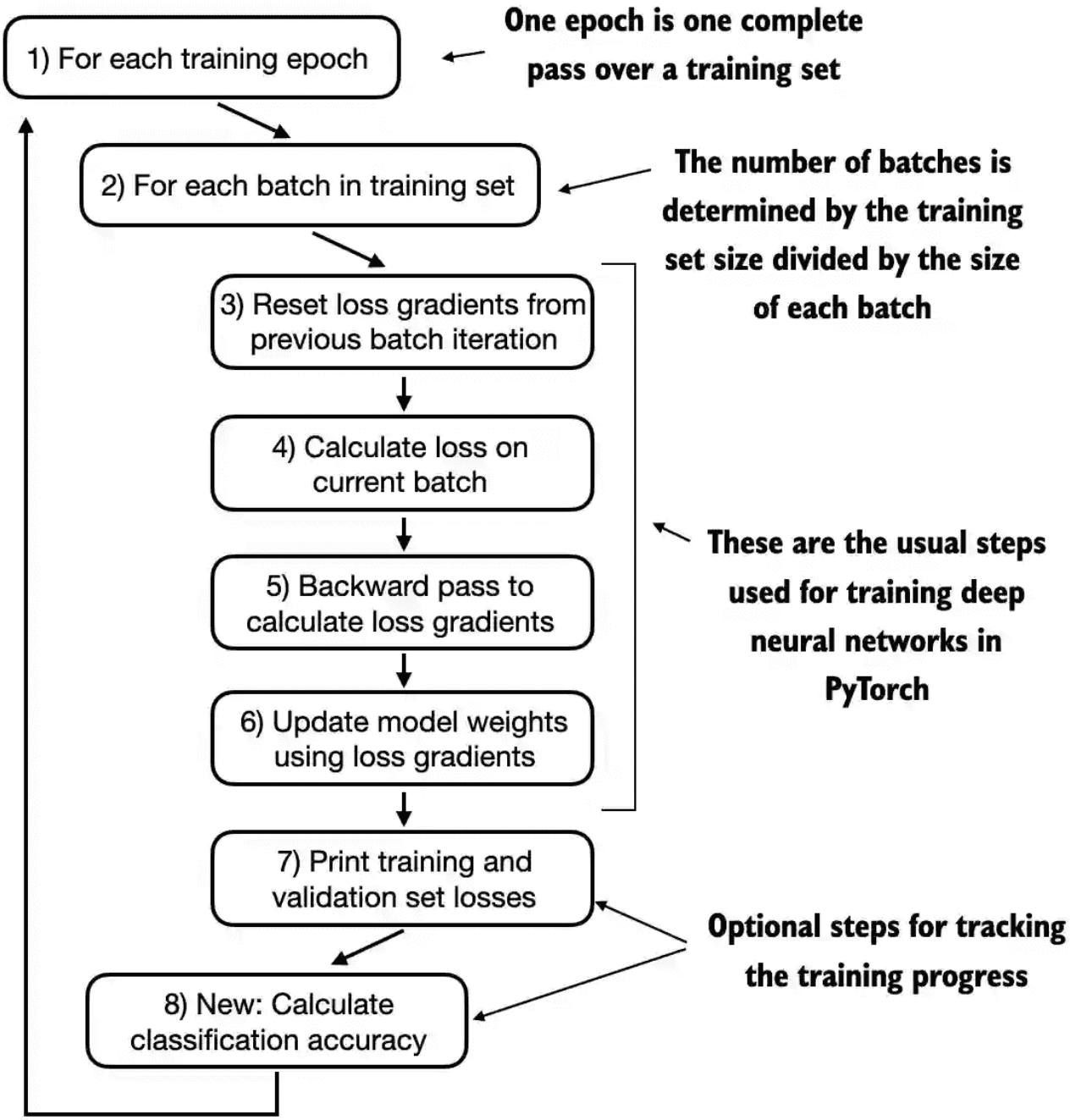
```
    loss = torch.nn.functional.cross_entropy(logits, target_batch)
    return loss
```

- The `calc_loss_loader` is exactly the same as in chapter 5.
- Using the `calc_closs_loader`, we compute the initial training, validation, and test set losses before we start training.

```
with torch.no_grad(): # Disable gradient tracking for efficiency because we are
not training, yet
    train_loss = calc_loss_loader(train_loader, model, device, num_batches=5)
    val_loss = calc_loss_loader(val_loader, model, device, num_batches=5)
    test_loss = calc_loss_loader(test_loader, model, device, num_batches=5)

print(f"Training loss: {train_loss:.3f}")
print(f"Validation loss: {val_loss:.3f}")
print(f"Test loss: {test_loss:.3f}")
```

6.7 Finetuning the model on supervised data

- We must define and use the training function to fine-tune the pretrained LLM and improve its spam classification accuracy.



- Overall the same as `train_model_simple` in chapter 5.

```
def train_classifier_simple(model, train_loader, val_loader, optimizer, device,
num_epochs,
                            eval_freq, eval_iter):
    # Initialize lists to track losses and examples seen
    train_losses, val_losses, train_accs, val_accs = [], [], [], []
    examples_seen, global_step = 0, -1

    # Main training loop
    for epoch in range(num_epochs):
        model.train()  # Set model to training mode

        for input_batch, target_batch in train_loader:
            optimizer.zero_grad() # Reset loss gradients from previous batch
```

```
iteration
            loss = calc_loss_batch(input_batch, target_batch, model, device)
            loss.backward() # Calculate loss gradients
            optimizer.step() # Update model weights using loss gradients
            examples_seen += input_batch.shape[0] # New: track examples instead of
tokens
            global_step += 1

            # Optional evaluation step
            if global_step % eval_freq == 0:
                train_loss, val_loss = evaluate_model(
                    model, train_loader, val_loader, device, eval_iter)
                train_losses.append(train_loss)
                val_losses.append(val_loss)
                print(f"Ep {epoch+1} (Step {global_step:06d}): "
                      f"Train loss {train_loss:.3f}, Val loss {val_loss:.3f}")

        # Calculate accuracy after each epoch
        train_accuracy = calc_accuracy_loader(train_loader, model, device,
num_batches=eval_iter)
        val_accuracy = calc_accuracy_loader(val_loader, model, device,
num_batches=eval_iter)
        print(f"Training accuracy: {train_accuracy*100:.2f}% | ", end="")
        print(f"Validation accuracy: {val_accuracy*100:.2f}%")
        train_accs.append(train_accuracy)
        val_accs.append(val_accuracy)

    return train_losses, val_losses, train_accs, val_accs, examples_seen
```

- The `evaluate_model` function used in the `train_classifier_simple` is the same as the one we used in chapter 5:

```
# Same as chapter 5
def evaluate_model(model, train_loader, val_loader, device, eval_iter):
    model.eval()
    with torch.no_grad():
        train_loss = calc_loss_loader(train_loader, model, device,
num_batches=eval_iter)
        val_loss = calc_loss_loader(val_loader, model, device,
num_batches=eval_iter)
    model.train()
    return train_loss, val_loss
```

- The training takes about 5 minutes on a M3 MacBook Air laptop computer and less than half a minute on a V100 or A100 GPU.

```
optimizer = torch.optim.AdamW(model.parameters(), lr=5e-5, weight_decay=0.1)

num_epochs = 5
```

```
train_losses, val_losses, train_accs, val_accs, examples_seen =
train_classifier_simple(
    model, train_loader, val_loader, optimizer, device,
    num_epochs=num_epochs, eval_freq=50, eval_iter=5,
)
```

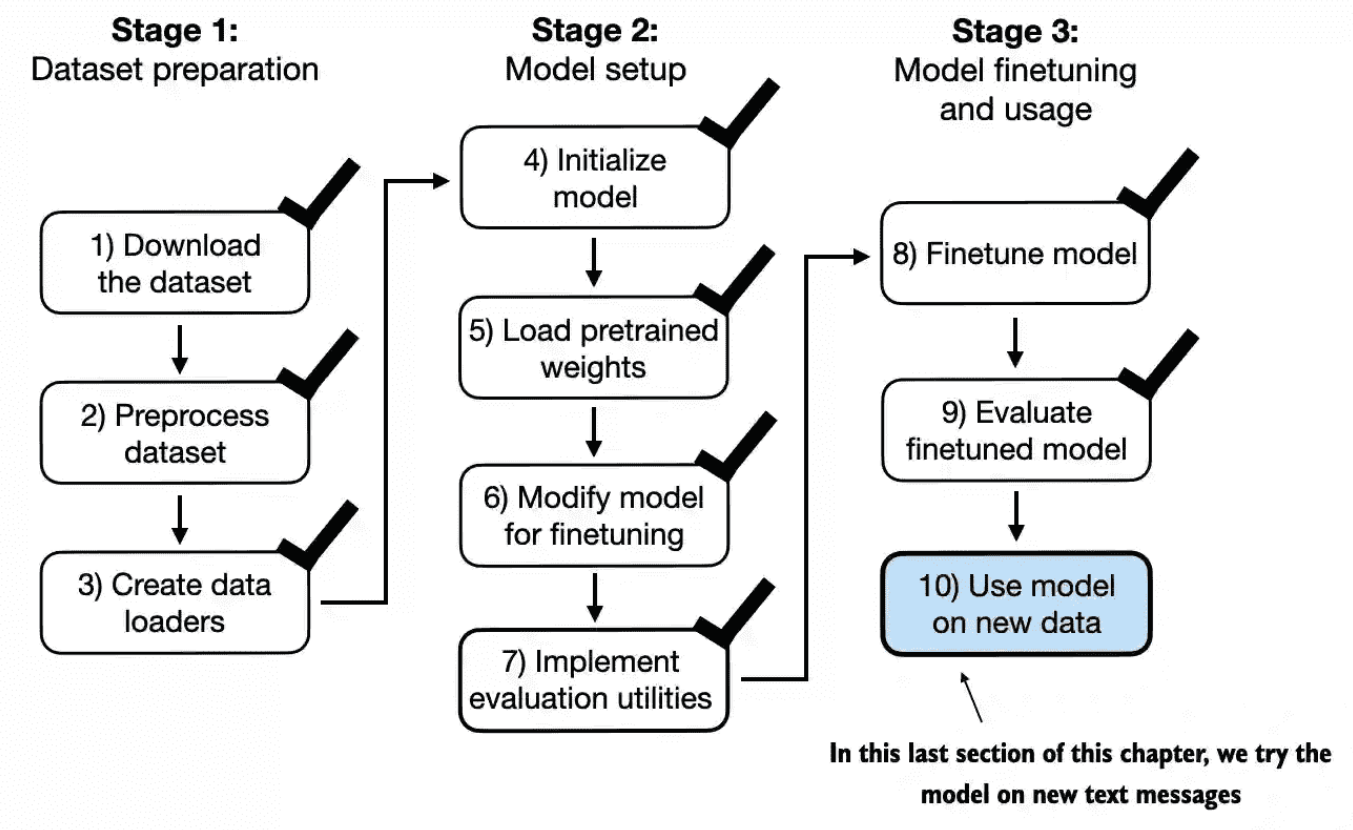
The resulting accuracy values are:

```
Training accuracy: 97.21%
Validation accuracy: 97.32%
Test accuracy: 95.67%
```

CHOOSING THE NUMBER OF EPOCHS

- The number of epochs depends on the dataset and the task's difficulty, and there is no universal solution or recommendation, although an epoch number of five is usually a good starting point.
- If the model overfits after the first few epochs as a loss plot, you may need to reduce the number of epochs.
- If the trendline suggests that the validation loss could improve with further training, you should increase the number of epochs.

6.8 Using the LLM as a spam classifier

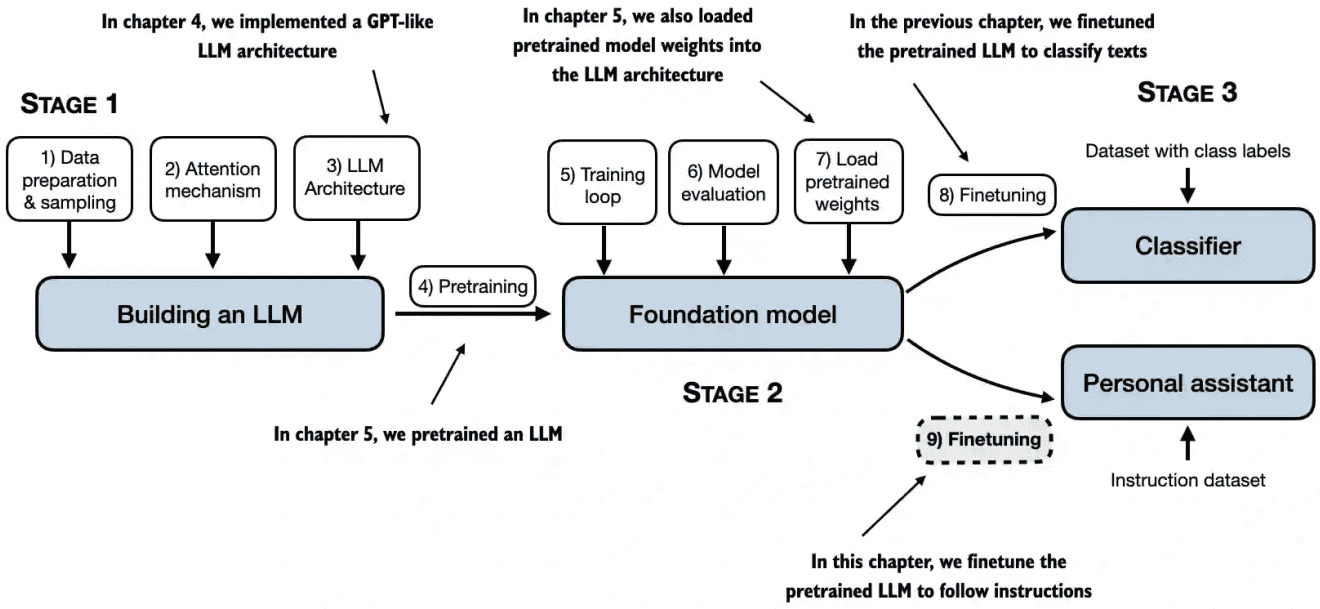


6.9 Summary

- There are different strategies for fine-tuning LLMs, including classification fine-tuning and instruction fine-tuning.
- Classification fine-tuning involves replacing the output layer of an LLM via a small classification layer.
- In the case of classifying text messages as “spam” or “not spam,” the new classification layer consists of only two output nodes.
- Previously, we used the number of output nodes equal to the number of unique tokens in the vocabulary (i.e., 50,256).
- Instead of predicting the next token in the text as in pretraining, classification fine-tuning trains the model to output a correct class label—for example, “spam” or “not spam.”
- The model input for fine-tuning is text converted into token IDs, similar to pretraining.
- Before fine-tuning an LLM, we load the pretrained model as a base model.
- Evaluating a classification model involves calculating the classification accuracy (the fraction or percentage of correct predictions).
- Fine-tuning a classification model uses the same cross entropy loss function as when pretraining the LLM.

7 Fine-tuning to follow instructions

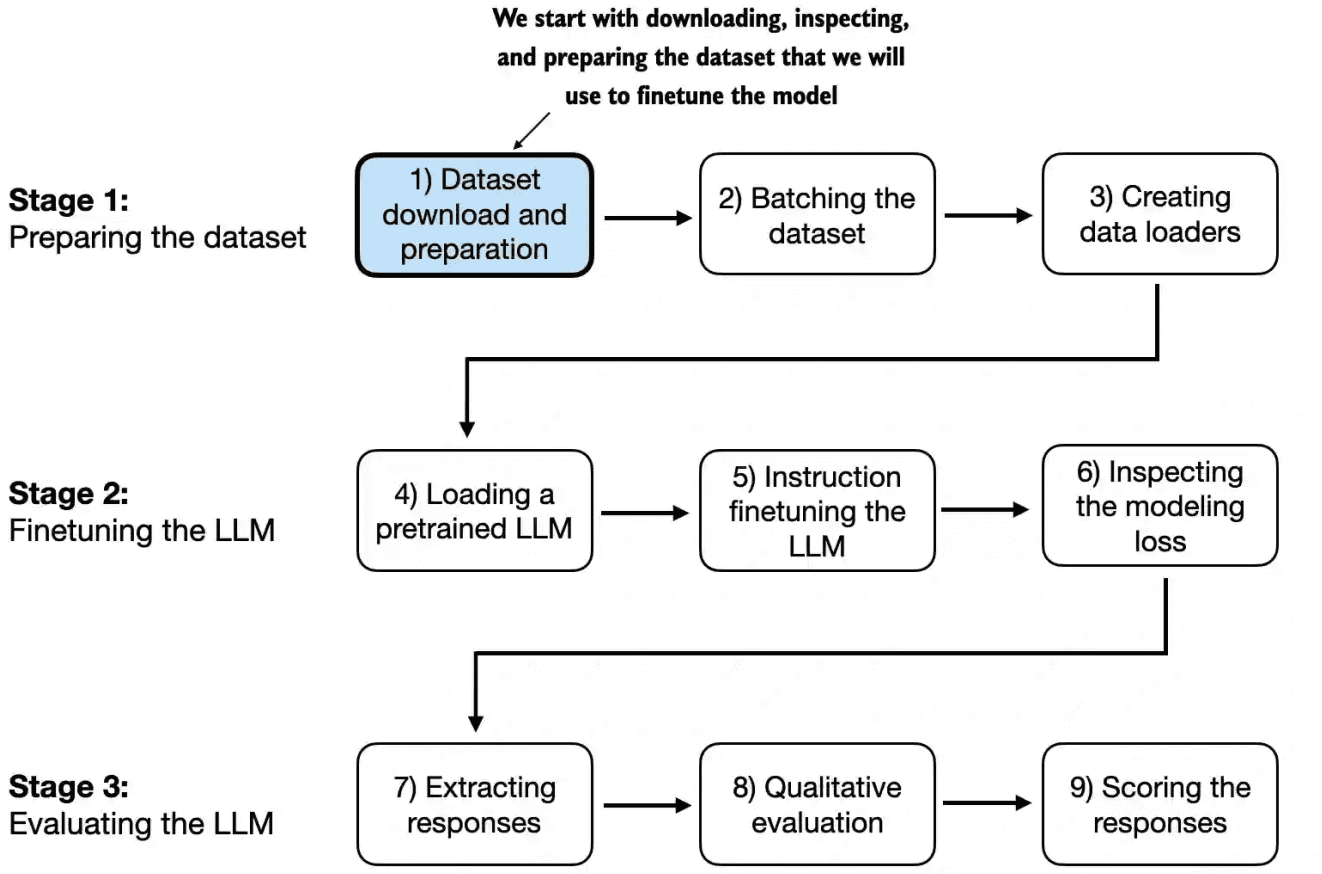
- This chapter covers
- The instruction fine-tuning process of LLMs
- Preparing a dataset for supervised instruction fine-tuning Organizing instruction data in training batches
- Loading a pretrained LLM and fine-tuning it to follow human instructions
- Extracting LLM-generated instruction responses for evaluation
- Evaluating an instruction-fine-tuned LLM



7.1 Introduction to instruction fine-tuning

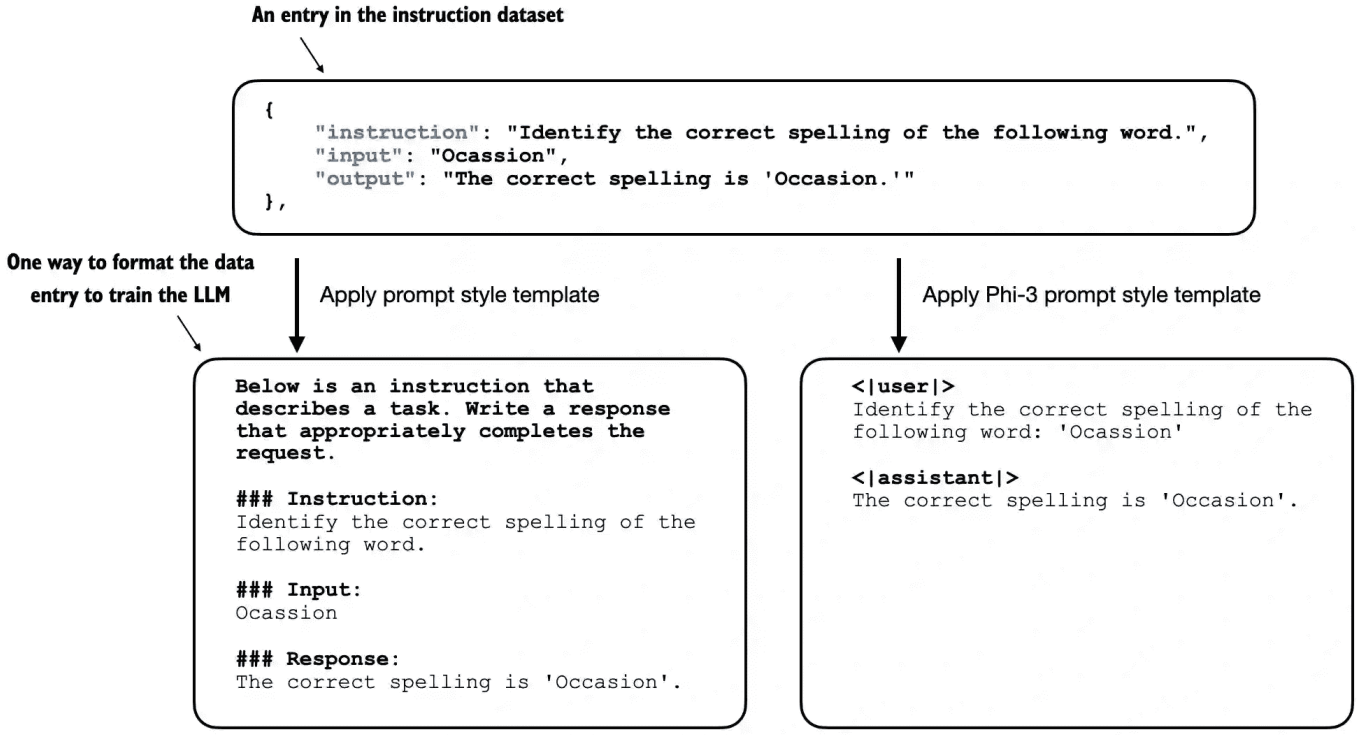
- Instruction fine-tuning, also known as supervised instruction fine-tuning, involves training a model on a dataset where the input-output pairs are explicitly provided.

Instruction		Desired response
Convert 45 kilometers to meters.	→	45 kilometers is 45000 meters.
Provide a synonym for “bright”.	→	A synonym for “bright” is “radiant”.
Edit the following sentence to remove all passive voice: “The song was composed by the artist.”	→	The artist composed the song.



7.2 Preparing a dataset for supervised instruction fine-tuning

- Dataset URL: https://raw.githubusercontent.com/rasbt/LLMs-from-scratch/main/ch07/01_main-chapter-code/instruction-data.json
- There are different ways to format the entries as inputs to the LLM; the figure below illustrates two example formats that were used for training the Alpaca (https://crfm.stanford.edu/2023/03/13/alpaca.html) and Phi-3 (https://arxiv.org/abs/2404.14219) LLMs, respectively.



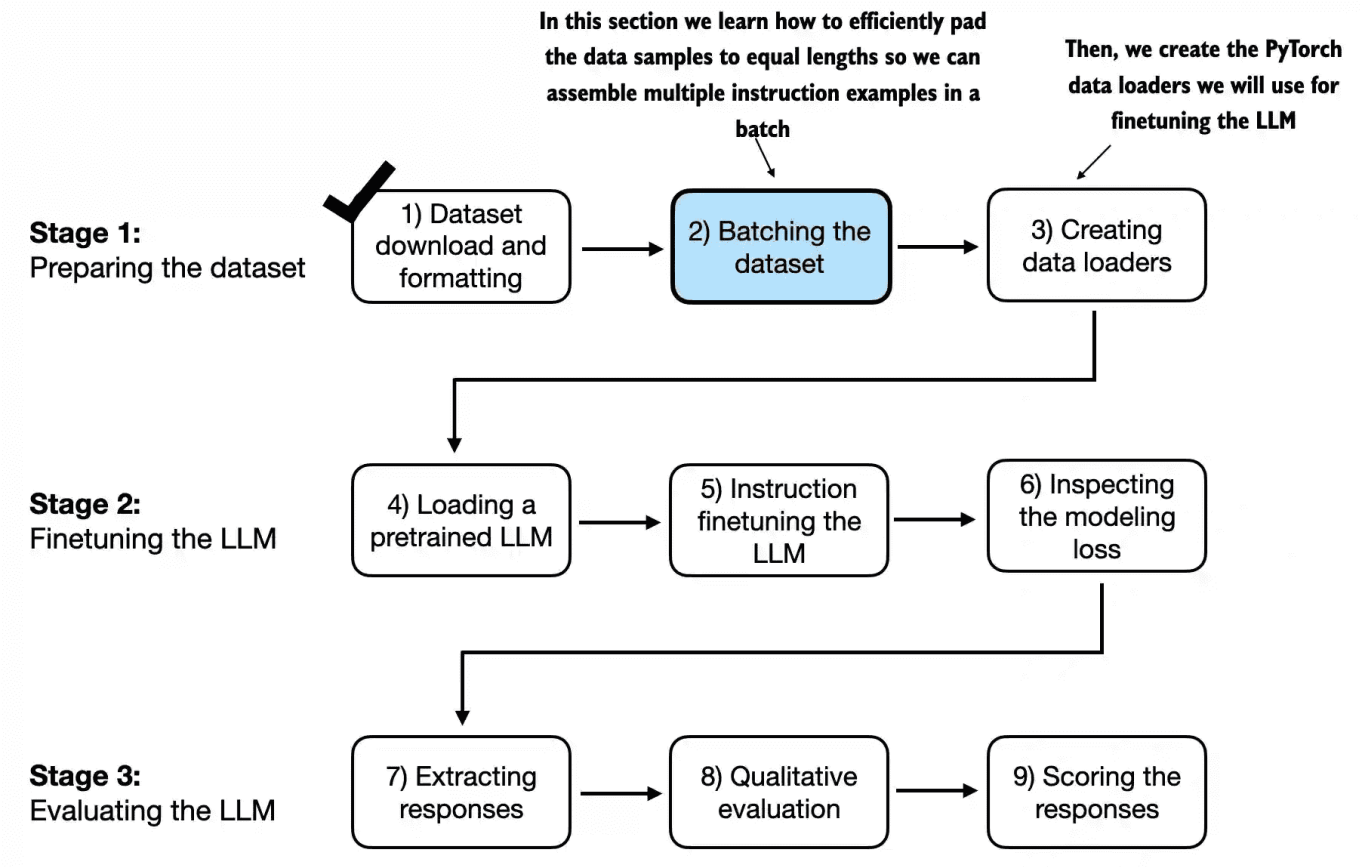
- Prompt styles.
- Alpaca was one of the early LLMs to publicly detail its instruction fine-tuning process.
- Phi-3, developed by Microsoft, is included to demonstrate the diversity in prompt styles.

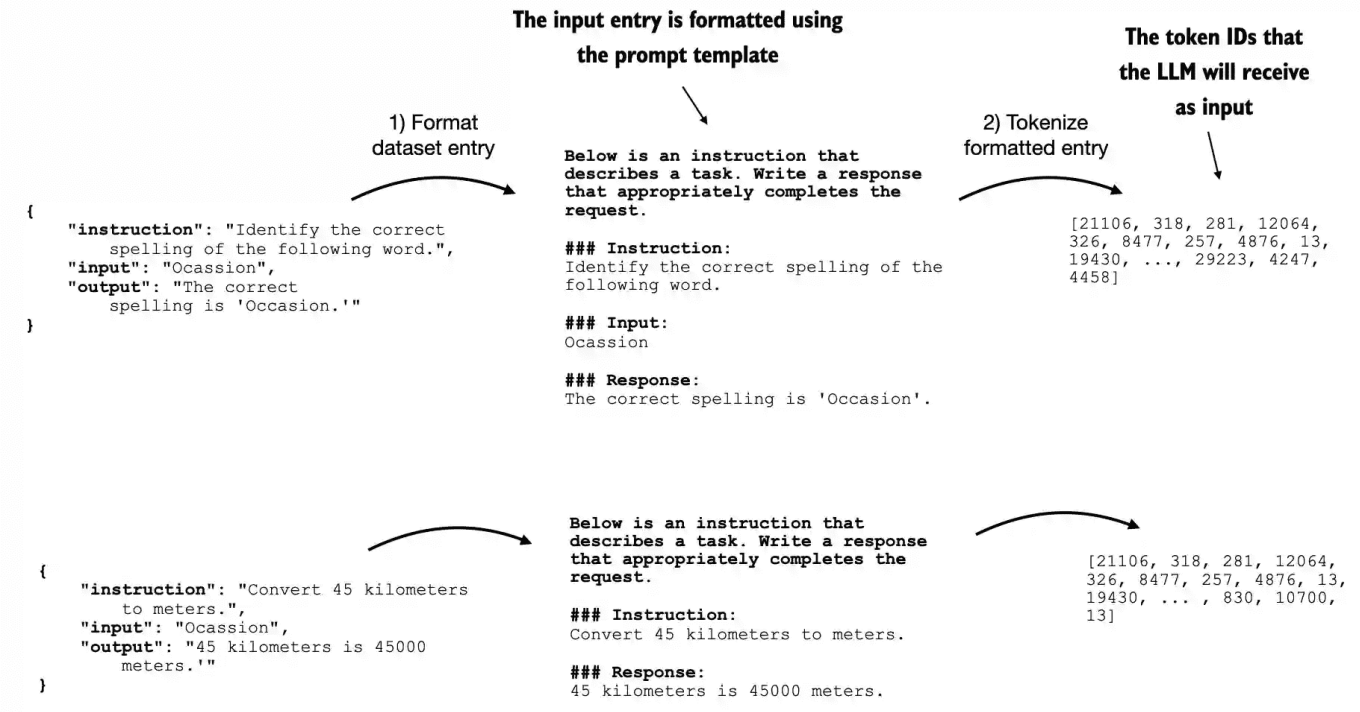
> **Note:** Fine-tuning the model with the Phi-3 template is approximately 17% faster than Alpaca template, since it results in shorter model inputs. The score is similar.

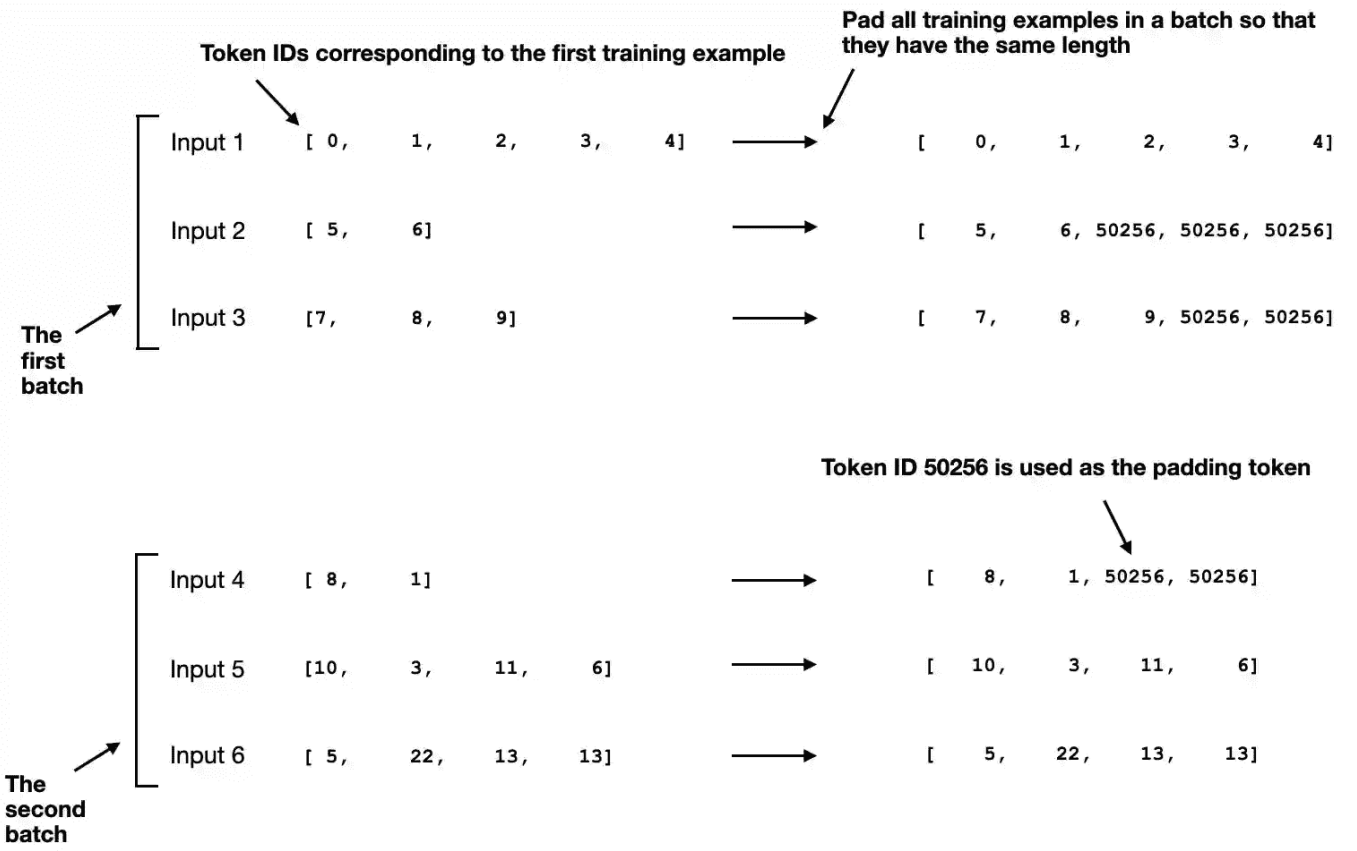
- Here, we use the Alpaca format for prompt styles.
- The following code converts the data into the Alpaca format.

```
def format_input(entry):
    instruction_text = (
        f"Below is an instruction that describes a task. "
        f"Write a response that appropriately completes the request."
        f"\n\n### Instruction:\n{entry['instruction']}"
    )

    input_text = f"\n\n### Input:\n{entry['input']}" if entry["input"] else ""

    return instruction_text + input_text
```

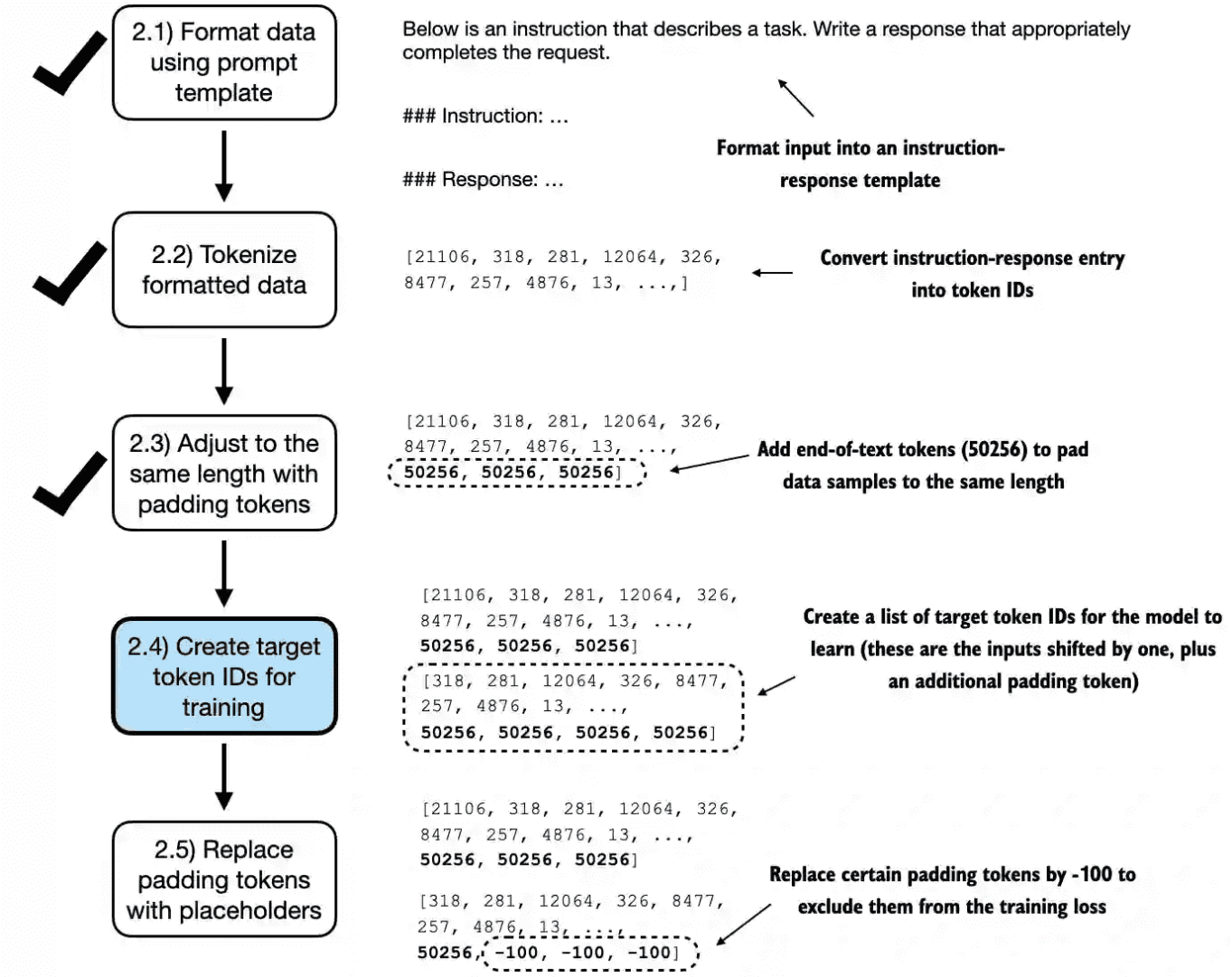
7.3 Organizing data into training batches


In this section we learn how to efficiently pad the data samples to equal lengths so we can assemble multiple instruction examples in a batch
Then, we create the PyTorch data loaders we will use for finetuning the LLM
Stage 1:
Preparing the dataset
1) Dataset download and formatting
2) Batching the dataset
3) Creating data loaders
Stage 2:
Finetuning the LLM
4) Loading a pretrained LLM
5) Instruction finetuning the LLM
6) Inspecting the modeling loss
Stage 3:
Evaluating the LLM
7) Extracting responses
8) Qualitative evaluation
9) Scoring the responses


The input entry is formatted using the prompt template
The token IDs that the LLM will receive as input
1) Format dataset entry
2) Tokenize formatted entry
{
"instruction": "Identify the correct spelling of the following word.",
"input": "Ocassion",
"output": "The correct spelling is 'Occasion.'"
}
Below is an instruction that describes a task. Write a response that appropriately completes the request.
Instruction:
Identify the correct spelling of the following word.
Input:
Ocassion
Response:
The correct spelling is 'Occasion'.
[21106, 318, 281, 12064, 326, 8477, 257, 4876, 13, 19430, ..., 29223, 4247, 4458]
{
"instruction": "Convert 45 kilometers to meters.",
"input": "Ocassion",
"output": "45 kilometers is 45000 meters.'"
}
Below is an instruction that describes a task. Write a response that appropriately completes the request.
Instruction:
Convert 45 kilometers to meters.
Response:
45 kilometers is 45000 meters.
[21106, 318, 281, 12064, 326, 8477, 257, 4876, 13, 19430, ... , 830, 10700, 13]




```
import torch
from torch.utils.data import Dataset


class InstructionDataset(Dataset):
    def __init__(self, data, tokenizer):
        self.data = data

        # Pre-tokenize texts
        self.encoded_texts = []
        for entry in data:
            instruction_plus_input = format_input(entry)
            response_text = f"\n\n### Response:\n{entry['output']}"
            full_text = instruction_plus_input + response_text
            self.encoded_texts.append(
                tokenizer.encode(full_text)
            )

    def __getitem__(self, index):
        return self.encoded_texts[index]

    def __len__(self):
        return len(self.data)
```



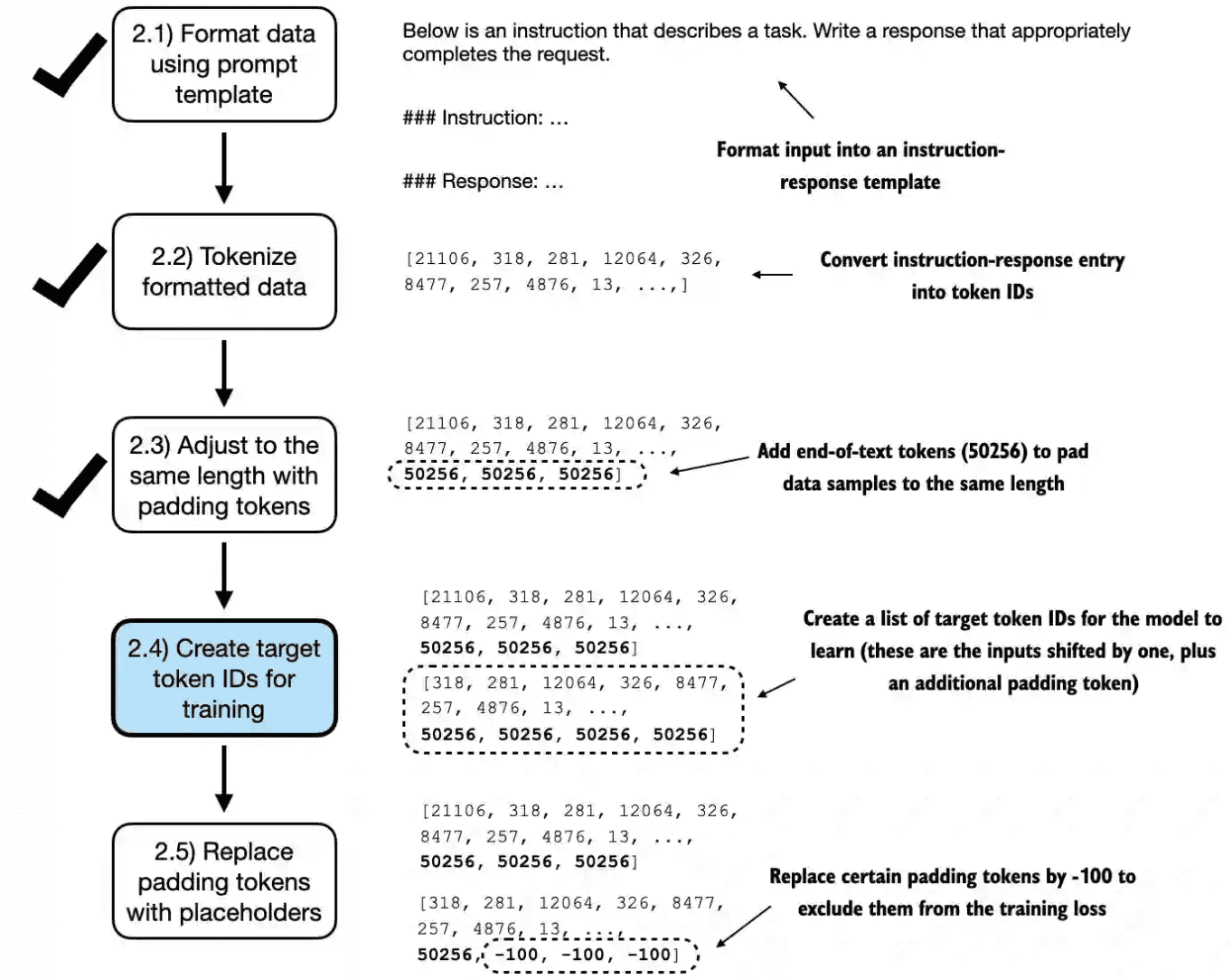
- Here, we take a more sophisticated (优雅) approach and develop a custom "collate" function that we can pass to the data loader.
- This custom `collate` function pads the training examples in each batch to have the same length (but different batches can have different lengths).

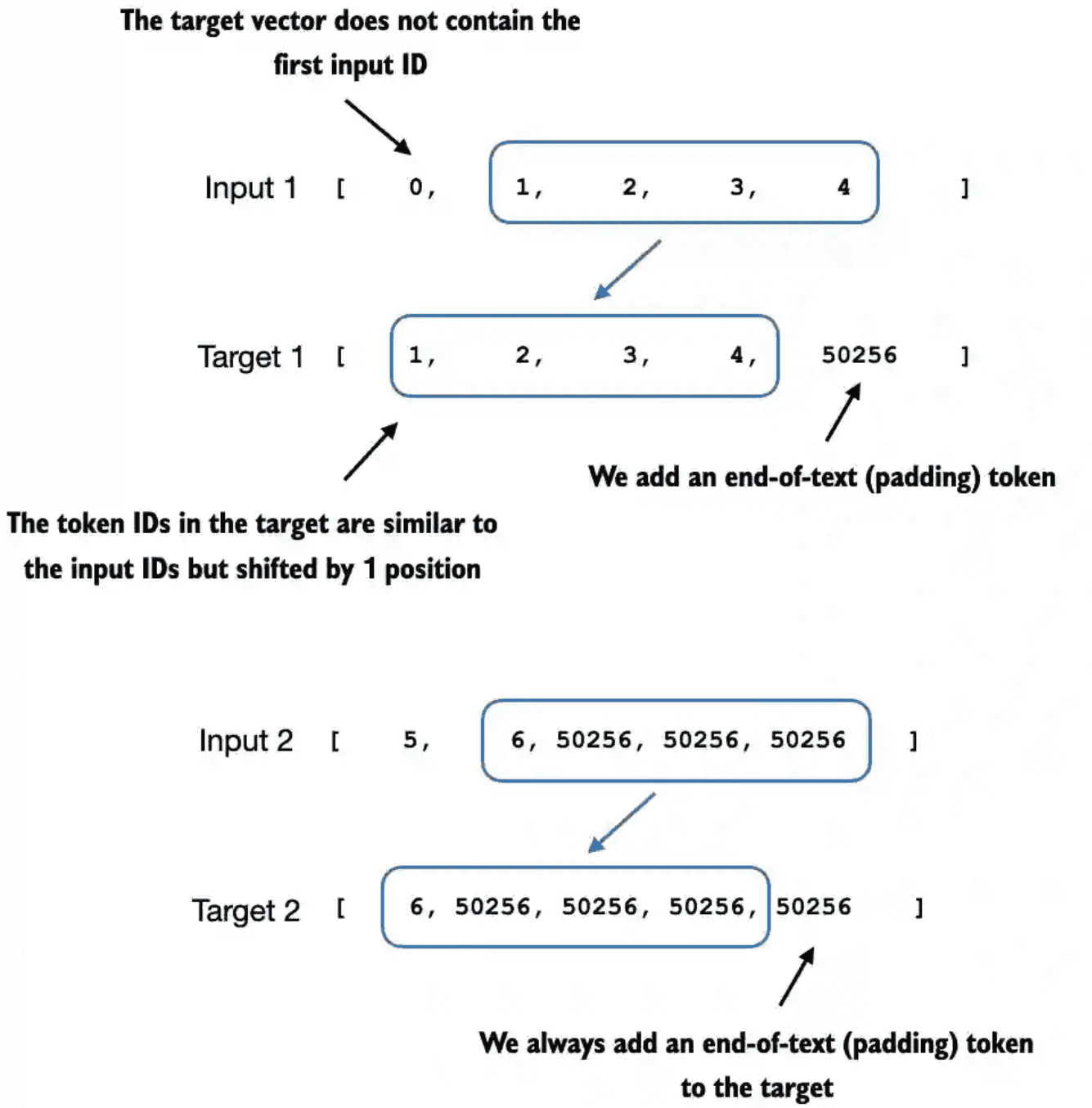
```
def custom_collate_draft_1(
    batch,
    pad_token_id=50256,
    device="cpu"
):
    # Find the longest sequence in the batch
    # and increase the max length by +1, which will add one extra
    # padding token below
    batch_max_length = max(len(item)+1 for item in batch)

    # Pad and prepare inputs
    inputs_lst = []

    for item in batch:
        new_item = item.copy()
        # Add an <|endoftext|> token
        new_item += [pad_token_id]
```

```
            # Pad sequences to batch_max_length
            padded = (
                new_item + [pad_token_id] * (batch_max_length - len(new_item))
            )
            # Via padded[:-1], we remove the extra padded token
            # that has been added via the +1 setting in batch_max_length
            # (the extra padding token will be relevant in later codes)
            inputs = torch.tensor(padded[:-1])
            inputs_lst.append(inputs)

        # Convert list of inputs to tensor and transfer to target device
        inputs_tensor = torch.stack(inputs_lst).to(device)
        return inputs_tensor
```

Usage example:

```
inputs_1 = [0, 1, 2, 3, 4]
inputs_2 = [5, 6]
inputs_3 = [7, 8, 9]

batch = (
    inputs_1,
    inputs_2,
    inputs_3
)

print(custom_collate_draft_1(batch))
# tensor([[    0,     1,     2,     3,     4],
#         [    5,     6, 50256, 50256, 50256],
#         [    7,     8,     9, 50256, 50256]])
```


2.1) Format data using prompt template
2.2) Tokenize formatted data
2.3) Adjust to the same length with padding tokens
2.4) Create target token IDs for training
2.5) Replace padding tokens with placeholders
Below is an instruction that describes a task. Write a response that appropriately completes the request.
Instruction: ...
Response: ...
Format input into an instruction-response template
[21106, 318, 281, 12064, 326, 8477, 257, 4876, 13, ...,]
Convert instruction-response entry into token IDs
[21106, 318, 281, 12064, 326, 8477, 257, 4876, 13, ..., 50256, 50256, 50256]
Add end-of-text tokens (50256) to pad data samples to the same length
[21106, 318, 281, 12064, 326, 8477, 257, 4876, 13, ..., 50256, 50256, 50256]
[318, 281, 12064, 326, 8477, 257, 4876, 13, ..., 50256, 50256, 50256, 50256]
Create a list of target token IDs for the model to learn (these are the inputs shifted by one, plus an additional padding token)
[21106, 318, 281, 12064, 326, 8477, 257, 4876, 13, ..., 50256, 50256, 50256]
[318, 281, 12064, 326, 8477, 257, 4876, 13, ..., 50256, -100, -100, -100]
Replace certain padding tokens by -100 to exclude them from the training loss



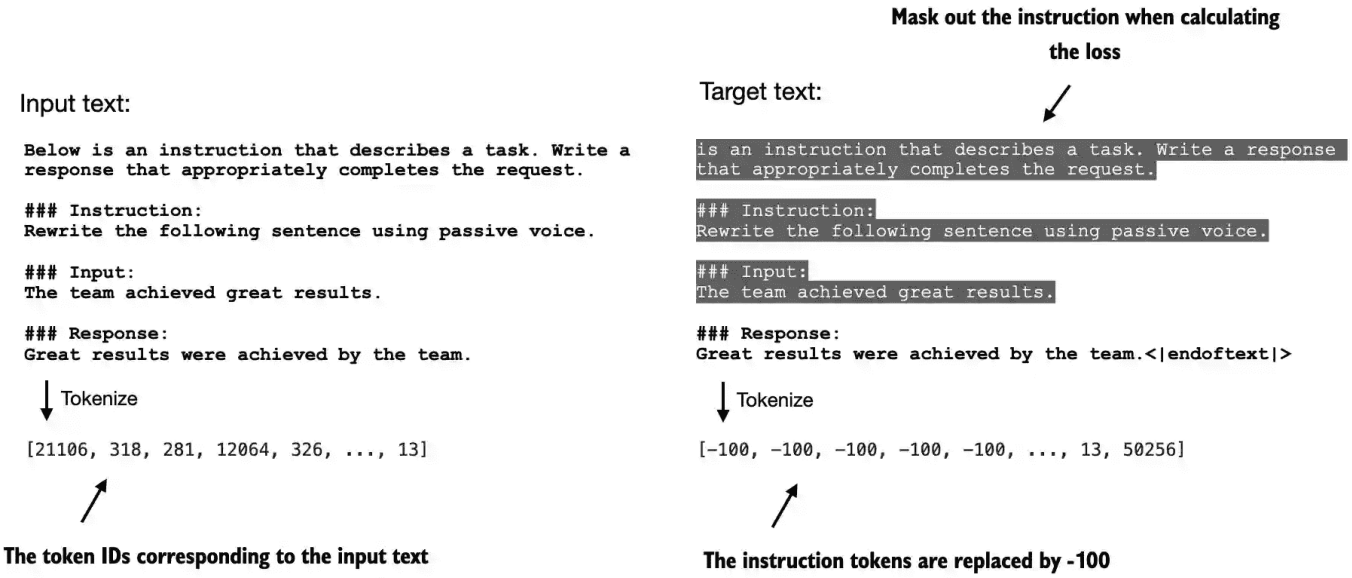

```python
def custom_collate_draft_2(
    batch,
    pad_token_id=50256,
    ignore_index=-100,
    allowed_max_length=None,
    device="cpu"
):
    # Find the longest sequence in the batch
    batch_max_length = max(len(item)+1 for item in batch)

    # Pad and prepare inputs
    inputs_lst, targets_lst = [], []
```

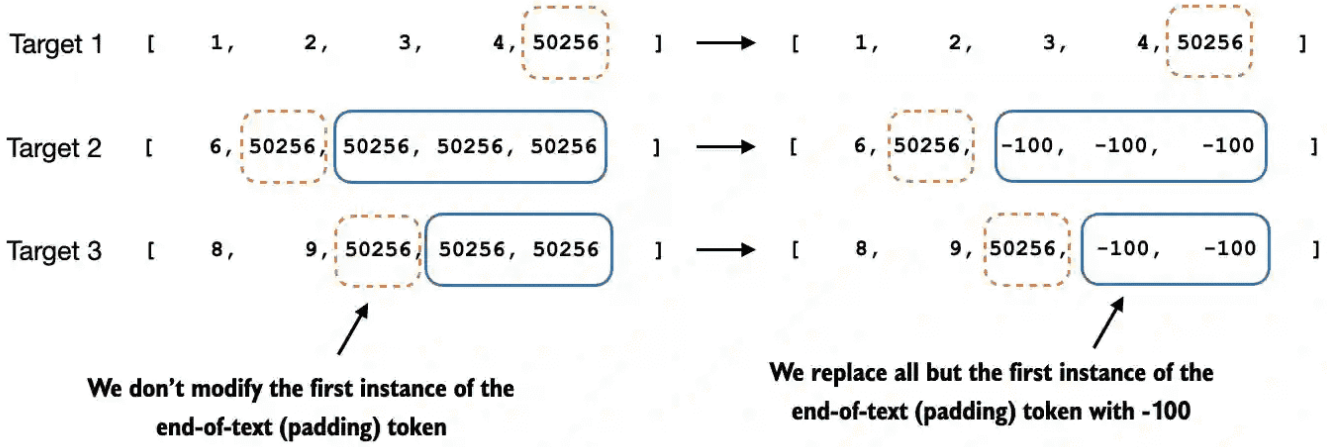
```
    for item in batch:
        new_item = item.copy()
        # Add an <|endoftext|> token
        new_item += [pad_token_id]
        # Pad sequences to max_length
        padded = (
            new_item + [pad_token_id] *
            (batch_max_length - len(new_item))
        )
        inputs = torch.tensor(padded[:-1])  # Truncate the last token for inputs
        targets = torch.tensor(padded[1:])  # Shift +1 to the right for targets

        inputs_lst.append(inputs)
        targets_lst.append(targets)

    # Convert list of inputs and targets to tensors and transfer to target device
    inputs_tensor = torch.stack(inputs_lst).to(device)
    targets_tensor = torch.stack(targets_lst).to(device)
    return inputs_tensor, targets_tensor
```

Usage example:

```
inputs, targets = custom_collate_draft_2(batch)
print(inputs)
print(targets)
# tensor([[    0,     1,     2,     3,     4],
#         [    5,     6, 50256, 50256, 50256],
#         [    7,     8,     9, 50256, 50256]])
# tensor([[    1,     2,     3,     4, 50256],
#         [    6, 50256, 50256, 50256, 50256],
#         [    8,     9, 50256, 50256, 50256]])
```

- Next, we introduce an `ignore_index` value to replace all padding token IDs with a new value; the purpose of this `ignore_index` is that we can ignore padding values in the loss function (more on that later).
- This special value allows us to exclude these padding tokens from contributing to the training loss calculation, ensuring that only meaningful data influences model learning.
- We will discuss this process in more detail after we implement this modification. (When fine-tuning for classification, we did not have to worry about this since we only trained the model based on the last output token.)






```
def custom_collate_fn(
    batch,
    pad_token_id=50256,
    ignore_index=-100,
    allowed_max_length=None,
    device="cpu"
):
    # Find the longest sequence in the batch
    batch_max_length = max(len(item)+1 for item in batch)

    # Pad and prepare inputs and targets
    inputs_lst, targets_lst = [], []

    for item in batch:
        new_item = item.copy()
        # Add an <|endoftext|> token
        new_item += [pad_token_id]
        # Pad sequences to max_length
        padded = (
            new_item + [pad_token_id] *
            (batch_max_length - len(new_item))
```

```
        )
        inputs = torch.tensor(padded[:-1])  # Truncate the last token for inputs
        targets = torch.tensor(padded[1:])  # Shift +1 to the right for targets

        # New: Replace all but the first padding tokens in targets by ignore_index
        mask = targets == pad_token_id
        indices = torch.nonzero(mask).squeeze()
        if indices.numel() > 1:
            targets[indices[1:]] = ignore_index

        # New: Optionally truncate to maximum sequence length
        if allowed_max_length is not None:
            inputs = inputs[:allowed_max_length]
            targets = targets[:allowed_max_length]

        inputs_lst.append(inputs)
        targets_lst.append(targets)

    # Convert list of inputs and targets to tensors and transfer to target device
    inputs_tensor = torch.stack(inputs_lst).to(device)
    targets_tensor = torch.stack(targets_lst).to(device)

    return inputs_tensor, targets_tensor
```

Usage example:

```
inputs, targets = custom_collate_fn(batch)
print(inputs)
print(targets)

# tensor([[    0,     1,     2,     3,     4],
#         [    5,     6, 50256, 50256, 50256],
#         [    7,     8,     9, 50256, 50256]])
# tensor([[    1,     2,     3,     4, 50256],
#         [    6, 50256,  -100,  -100,  -100],
#         [    8,     9, 50256,  -100,  -100]])
```

why replacement by -100

- Let's see what this replacement by -100 accomplishes.
- For illustration purposes, let's assume we have a small classification task with 2 class labels, 0 and 1, similar to chapter 6.
- If we have the following logits values (outputs of the last layer of the model), we calculate the following loss.

Example 1:

```
logits_1 = torch.tensor(
    [[-1.0, 1.0],  # 1st training example
     [-0.5, 1.5]]  # 2nd training example
)
targets_1 = torch.tensor([0, 1])


loss_1 = torch.nn.functional.cross_entropy(logits_1, targets_1)
print(loss_1)
# tensor(1.1269)
```

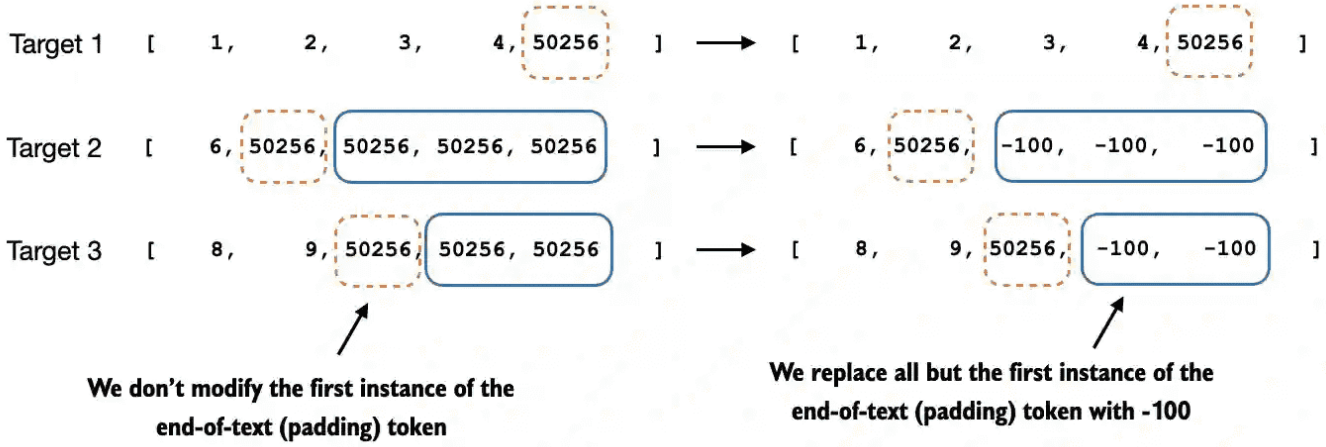
Now, adding one more training example will, as expected, influence the loss:

```
logits_2 = torch.tensor(
    [[-1.0, 1.0],
     [-0.5, 1.5],
     [-0.5, 1.5]]  # New 3rd training example
)
targets_2 = torch.tensor([0, 1, 1])

loss_2 = torch.nn.functional.cross_entropy(logits_2, targets_2)
print(loss_2)
# tensor(0.7936)
```

Let's see what happens if we replace the class label of one of the examples with -100:

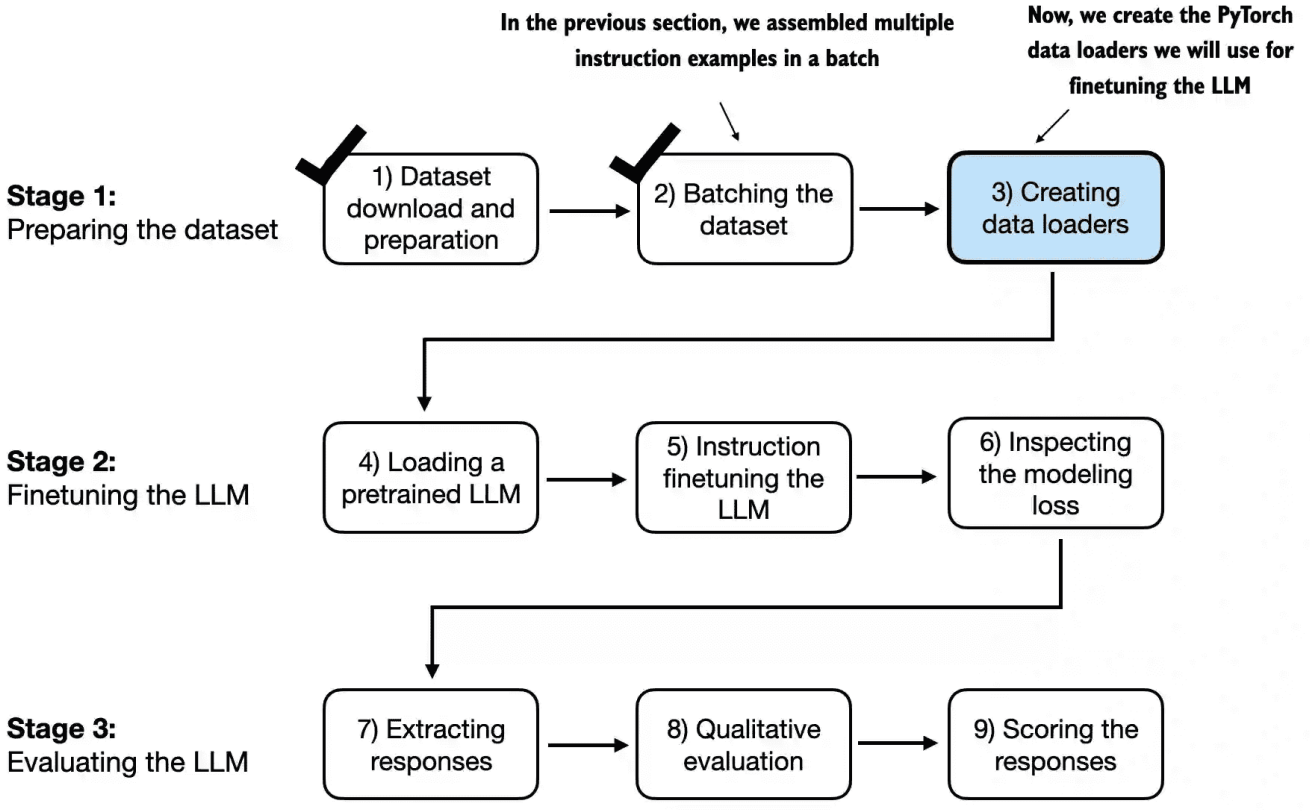
```
targets_3 = torch.tensor([0, 1, -100])

loss_3 = torch.nn.functional.cross_entropy(logits_2, targets_3)
print(loss_3)
print("loss_1 == loss_3:", loss_1 == loss_3)
# tensor(1.1269)
# loss_1 == loss_3: tensor(True)
```

- As we can see, the resulting loss on these 3 training examples is the same as the loss we calculated from the 2 training examples, which means that the cross-entropy loss function ignored the training example with the -100 label.
- By default, PyTorch has the `cross_entropy(..., ignore_index=-100)` setting to ignore examples corresponding to the label -100.
- Using this -100 `ignore_index`, we can ignore the additional end-of-text (padding) tokens in the batches that we used to pad the training examples to equal length.
- However, we don't want to ignore the first instance of the end-of-text (padding) token (50256) because it can help signal to the LLM when the response is complete.



> **Note:** As of this writing, researchers are divided on whether masking the instructions is universally beneficial during instruction fine- tuning. For instance, the 2024 paper by Shi et al., "Instruction Tuning With Loss Over Instructions" (https://arxiv.org/abs/2405.14394), demonstrated that not masking the instructions benefits the LLM performance (see appendix B for more details).

7.4 Creating data loaders for an instruction dataset



- Instantiate the data loaders similar to previous chapters, except that we now provide our own collate function for the batching process.

```
from functools import partial

customized_collate_fn = partial(
    custom_collate_fn,
    device=device,
```

```
    allowed_max_length=1024
)

from torch.utils.data import DataLoader


num_workers = 0
batch_size = 8

torch.manual_seed(123)

train_dataset = InstructionDataset(train_data, tokenizer)
train_loader = DataLoader(
    train_dataset,
    batch_size=batch_size,
    collate_fn=customized_collate_fn,
    shuffle=True,
    drop_last=True,
    num_workers=num_workers
)
# ... ... # Same for validation and test sets
```

Apply:

```
print("Train loader:")
for inputs, targets in train_loader:
    print(inputs.shape, targets.shape)
# Output
# Train loader:
#  8 represents the batch size and 61 is the number of tokens in each training
example in this batch.
# torch.Size([8, 61]) torch.Size([8, 61])   # Batch: 8, Max length in this batch:
61
# torch.Size([8, 76]) torch.Size([8, 76])   # Batch: 8, Max length in this batch:
76
# ... ...
# torch.Size([8, 66]) torch.Size([8, 66])   # ...
# torch.Size([8, 74]) torch.Size([8, 74])
# torch.Size([8, 69]) torch.Size([8, 69])
```

print(inputs[0]):

```
tensor([21106,   318,   281, 12064,   326,  8477,   257,  4876,    13, 19430,
          257,  2882,   326, 20431, 32543,   262,  2581,    13,   198,   198,
        21017, 46486,    25,   198, 30003,  6525,   262,  6827,  1262,   257,
          985,   576,    13,   198,   198, 21017, 23412,    25,   198,   464,
         5156,   318,   845, 13779,    13,   198,   198, 21017, 18261,    25,
          198,   464,  5156,   318,   355, 13779,   355,   257,  4936,    13,
```

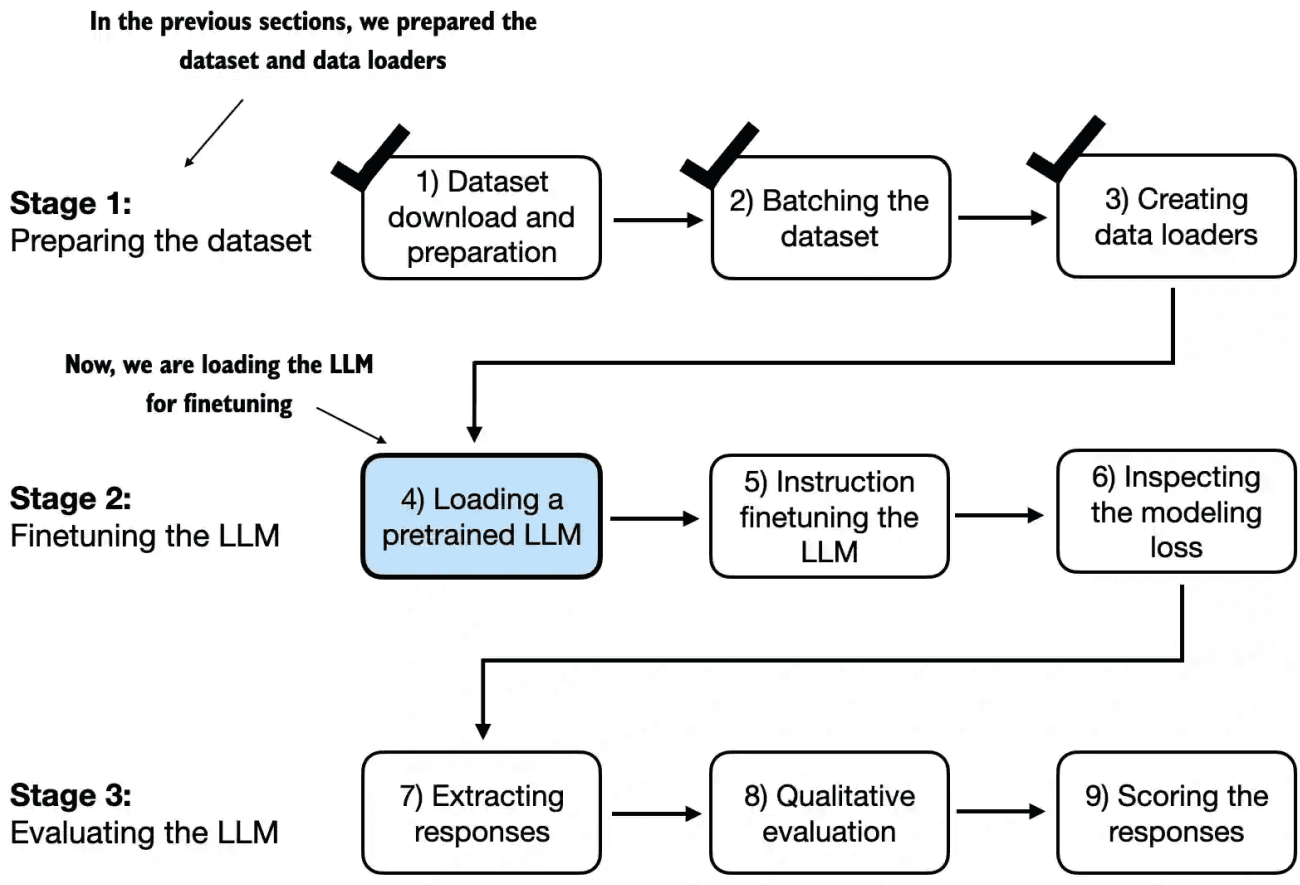
```
         50256, 50256, 50256, 50256, 50256, 50256, 50256, 50256, 50256],
        device='cuda:0')
```

print(targets[0]):

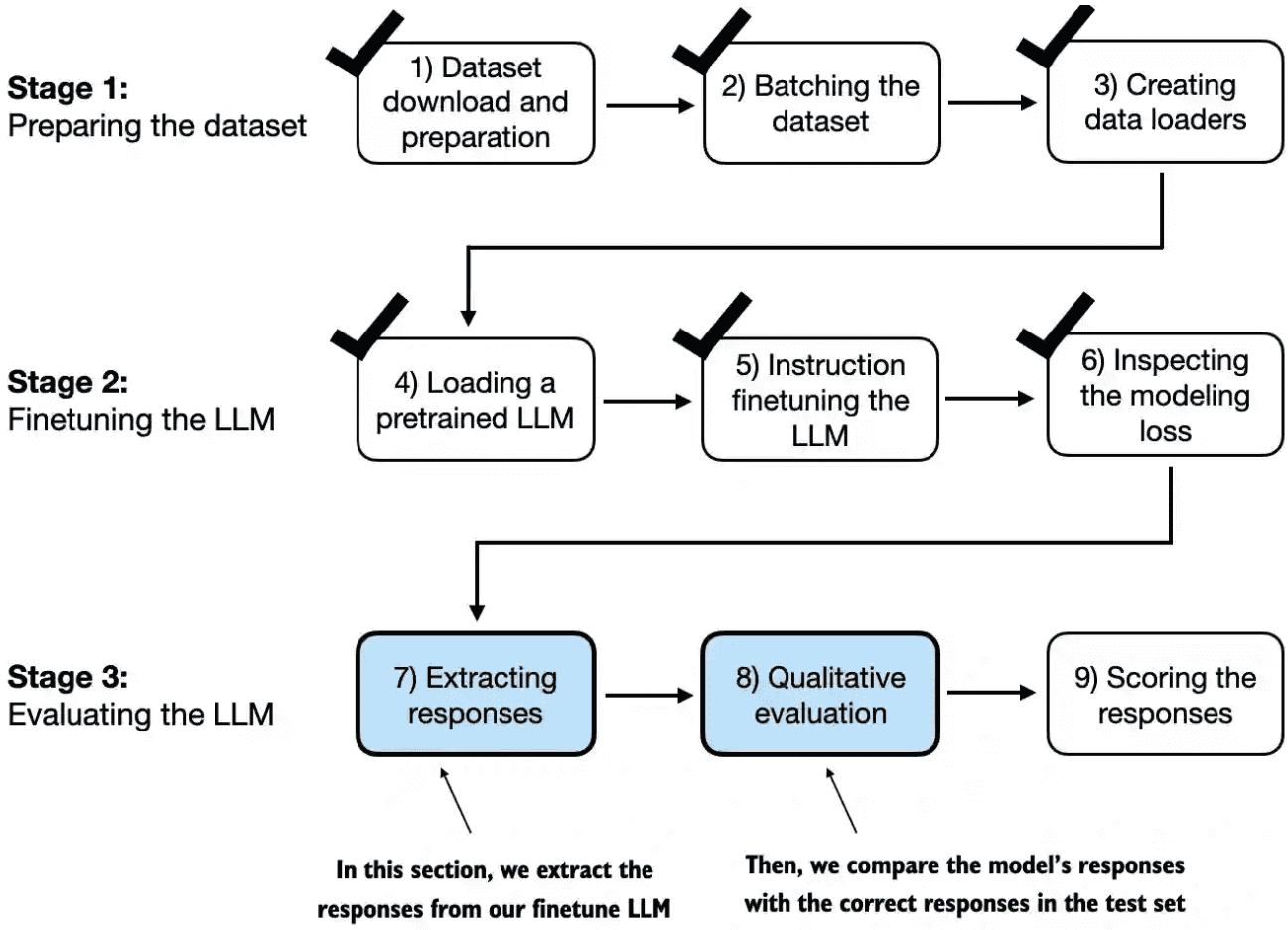
```
tensor([  318,   281, 12064,   326,  8477,   257,  4876,    13, 19430,   257,
         2882,   326, 20431, 32543,   262,  2581,    13,   198,   198, 21017,
        46486,    25,   198, 30003,  6525,   262,  6827,  1262,   257,   985,
          576,    13,   198,   198, 21017, 23412,    25,   198,   464,  5156,
          318,   845, 13779,    13,   198,   198, 21017, 18261,    25,   198,
          464,  5156,   318,   355, 13779,   355,   257,  4936,    13, 50256,
         -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100,  -100],
        device='cuda:0')
```

7.5 Loading a pretrained LLM

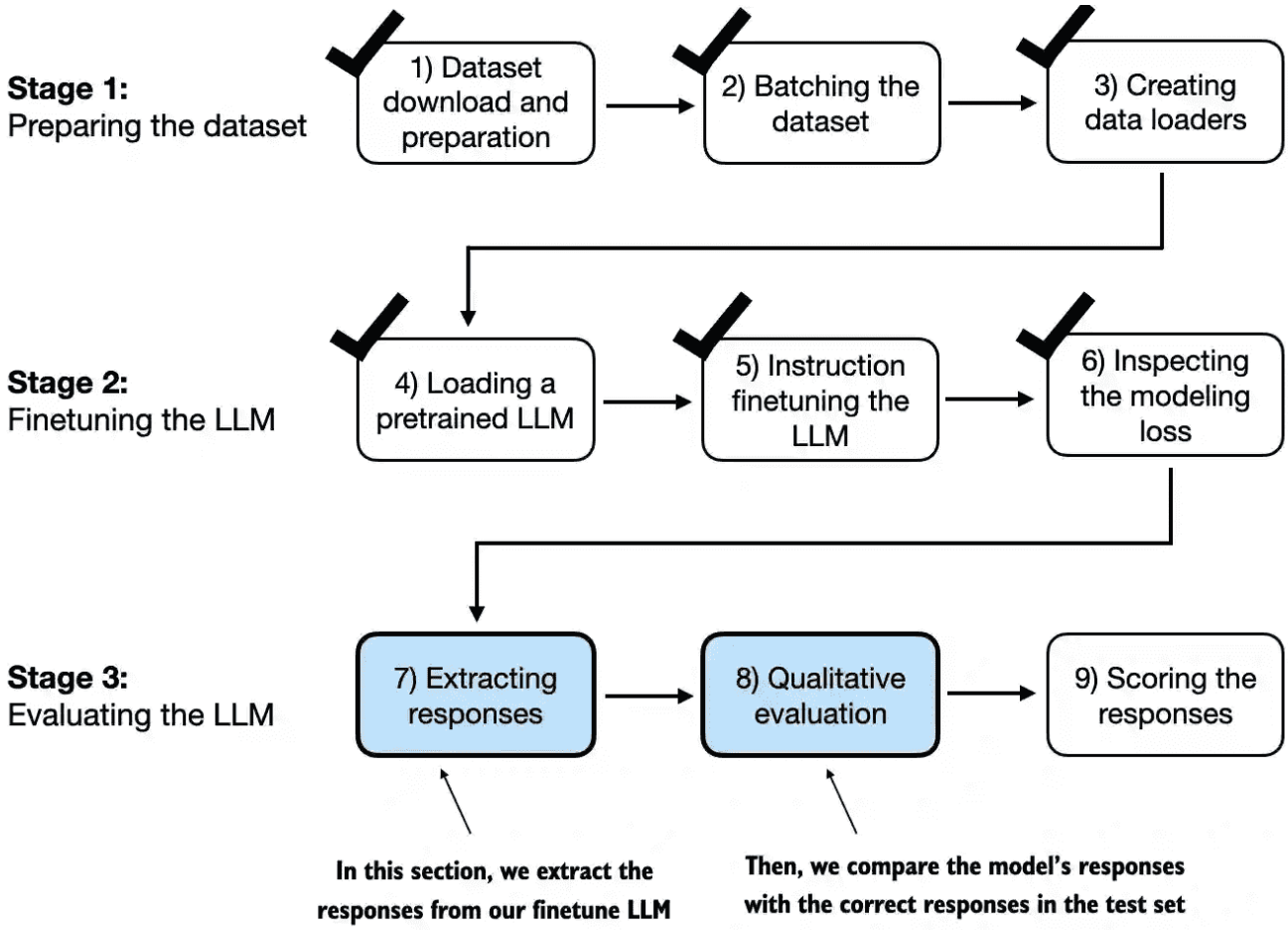
> **Note:** Instead of using the smallest 124-million-parameter model as before, we load the medium-sized model with 355 million parameters. The reason for this choice is that the 124-million- parameter model is too limited in capacity to achieve satisfactory results via instruction fine-tuning. Specifically, smaller models lack the necessary capacity to learn and retain the intricate patterns and nuanced behaviors required for high- quality instruction-following tasks.



7.6 Fine-tuning the LLM on instruction data



7.7 Extracting and saving responses

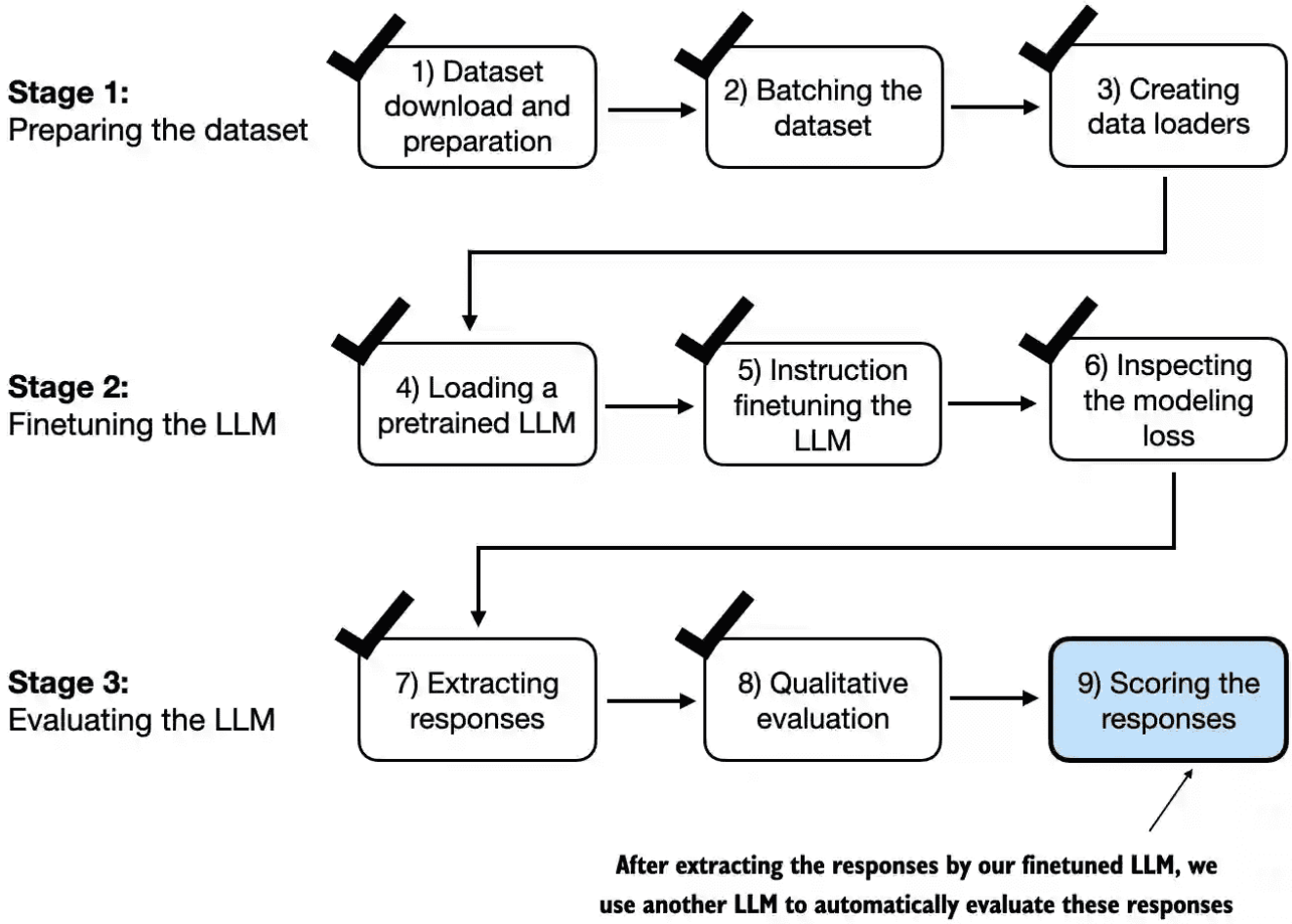


- In practice, instruction-fine-tuned LLMs such as chatbots are evaluated via multiple approaches:
  - Short-answer and multiple-choice benchmarks, such as Measuring Massive Multitask Language Understanding (MMLU; https://arxiv.org/abs/2009.03300), which test the general knowledge of a model.
  - Human preference comparison to other LLMs, such as LMSYS chatbot arena (https://arena.lmsys.org).
  - Automated conversational benchmarks, where another LLM like GPT-4 is used to evaluate the responses, such as AlpacaEval (https://tatsu-lab.github.io/alpaca_eval/).
- This book will use the third method (AlpacaEval).
- First, organize the input, expected output, and model output data.

```
from tqdm import tqdm

for i, entry in tqdm(enumerate(test_data), total=len(test_data)):

    input_text = format_input(entry)

    token_ids = generate(
        model=model,
        idx=text_to_token_ids(input_text, tokenizer).to(device),
        max_new_tokens=256,
        context_size=BASE_CONFIG["context_length"],
        eos_id=50256
    )
    generated_text = token_ids_to_text(token_ids, tokenizer).replace("
```

```
<|endoftext|>", "") # New: remove end-of-text token from output
    response_text = generated_text[len(input_text):].replace("### Response:",
"").strip()

    test_data[i]["model_response"] = response_text


with open("instruction-data-with-response.json", "w") as file:
    json.dump(test_data, file, indent=4)  # "indent" for pretty-printing
```

7.8 Evaluating the fine-tuned LLM



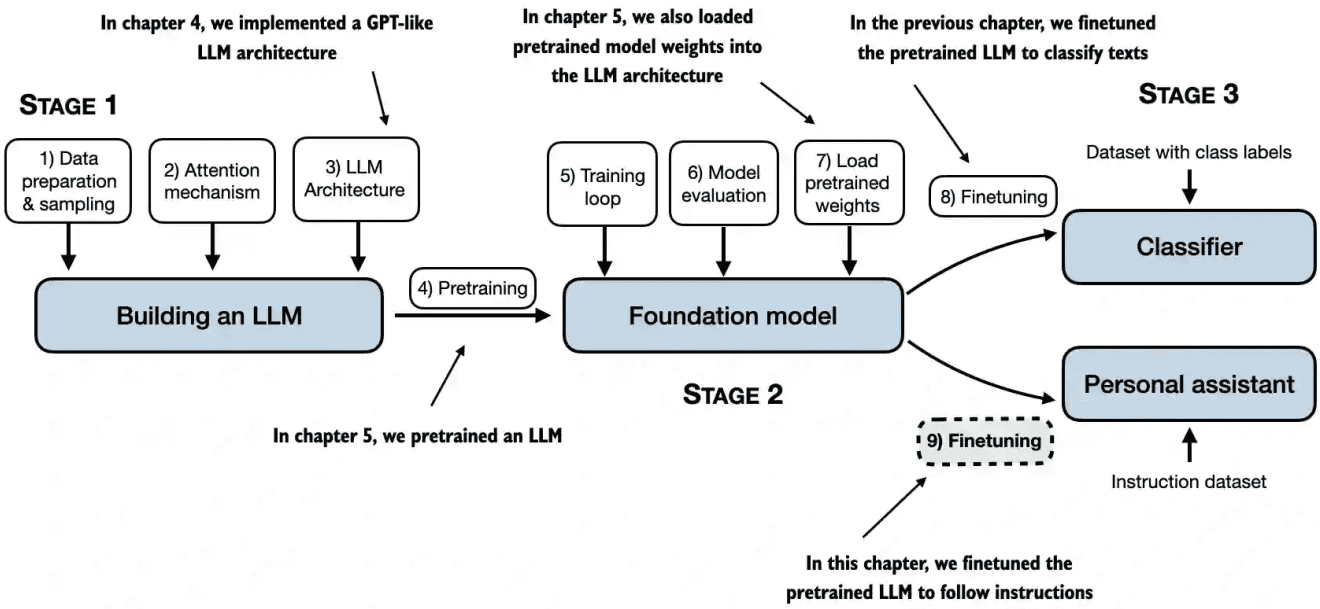

```
def format_input(entry):
    instruction_text = (
        f"Below is an instruction that describes a task. "
        f"Write a response that appropriately completes the request."
        f"\n\n### Instruction:\n{entry['instruction']}"
    )

    input_text = f"\n\n### Input:\n{entry['input']}" if entry["input"] else ""

    return instruction_text + input_text

for entry in test_data[:3]:
    prompt = (
```

```
        f"Given the input `{format_input(entry)}` "
        f"and correct output `{entry['output']}`, "
        f"score the model response `{entry['model_response']}`"
        f" on a scale from 0 to 100, where 100 is the best score. "
    )
    print("\nDataset response:")
    print(">>", entry['output'])
    print("\nModel response:")
    print(">>", entry["model_response"])
    print("\nScore:")
    print(">>", query_model(prompt))
    print("\n-------------------------")
# Output
# Dataset response:
# >> The car is as fast as lightning.

# Model response:
# >> The car is as fast as a bullet.

# Score:
# >> I'd rate the model response "The car is as fast as a bullet." an 85 out of
100.
```

Prompt to display only the percentage number:

```
prompt = (
    f"Given the input `{format_input(entry)}` "
    f"and correct output `{entry['output']}`, "
    f"score the model response `{entry[json_key]}`"
    f" on a scale from 0 to 100, where 100 is the best score. "
    f"Respond with the integer number only." # added ✳
)
```

7.9 Conclusions



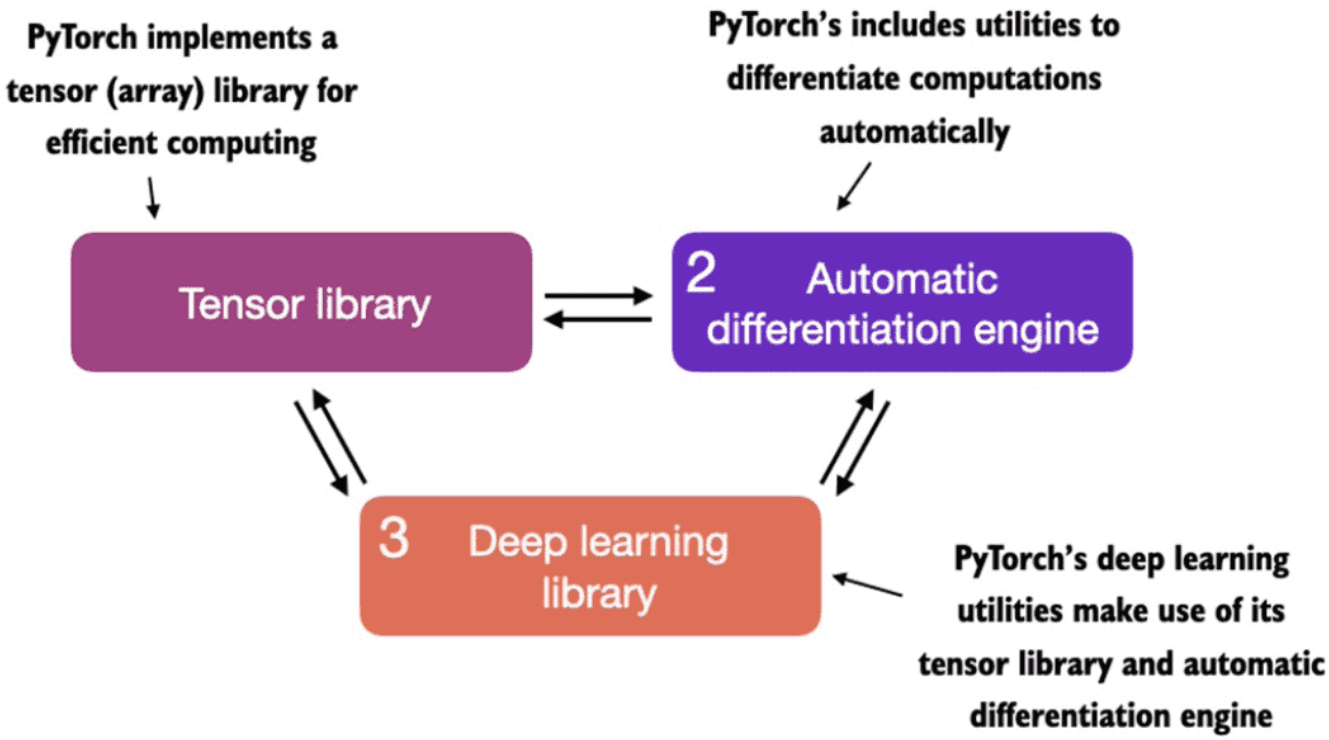
- While we covered the most essential steps, there is an optional step that can be performed after instruction fine-tuning: `preference fine-tuning`.
- `Preference fine-tuning` is particularly useful for customizing a model to better align with specific user preferences.

7.10 Summary

- The instruction-fine-tuning process adapts a pretrained LLM to follow human instructions and generate desired responses.
- Preparing the dataset involves downloading an instruction-response dataset, formatting the entries, and splitting it into train, validation, and test sets.
- Training batches are constructed using a custom collate function that pads sequences, creates target token IDs, and masks padding tokens.
- We load a pretrained GPT-2 medium model with 355 million parameters to serve as the starting point for instruction fine-tuning.
- The pretrained model is fine-tuned on the instruction dataset using a training loop similar to pretraining.
- Evaluation involves extracting model responses on a test set and scoring them (for example, using another LLM).
- The Ollama application with an 8-billion-parameter Llama model can be used to automatically score the fine-tuned model's responses on the test set, providing an average score to quantify performance.

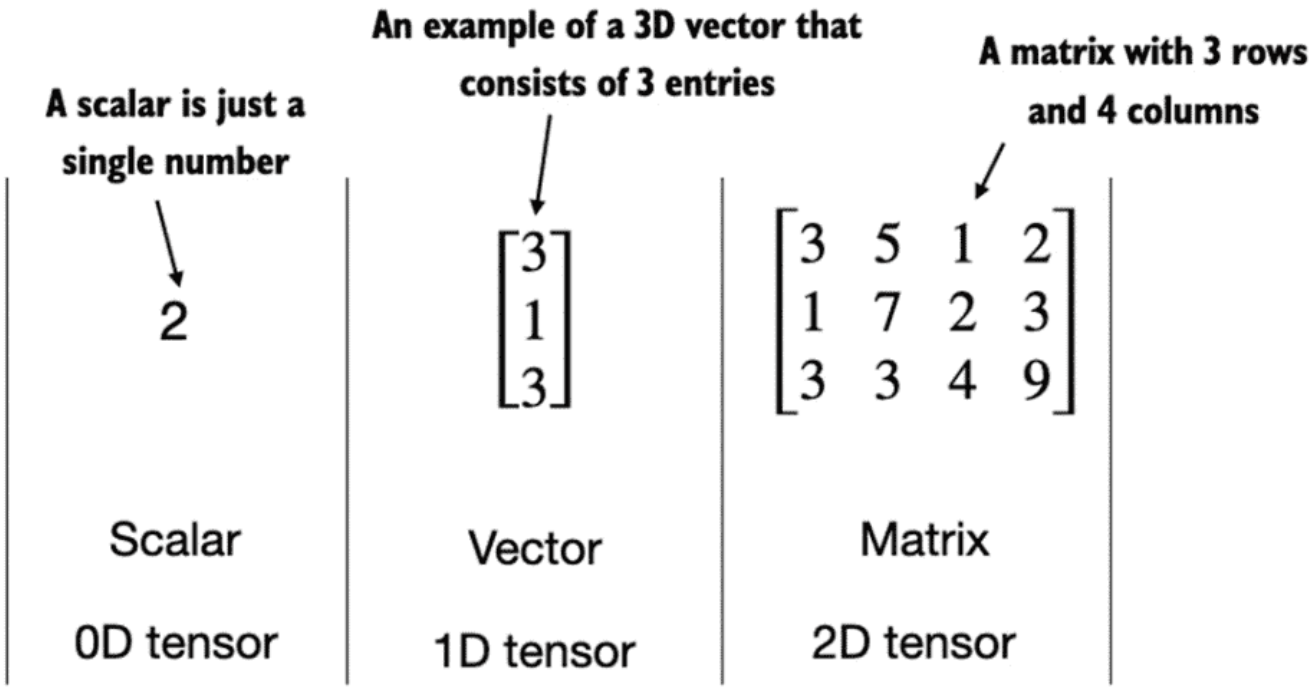
Appendix A. Introduction to PyTorch

A.1 What is PyTorch



- Firstly, PyTorch is a tensor library that extends the concept of array-oriented programming library NumPy with the additional feature of accelerated computation on GPUs, thus providing a seamless switch between CPUs and GPUs.
- Secondly, PyTorch is an automatic differentiation engine, also known as autograd, which enables the automatic computation of gradients for tensor operations, simplifying backpropagation and model optimization.
- Finally, PyTorch is a deep learning library, meaning that it offers modular, flexible, and efficient building blocks (including pre-trained models, loss functions, and optimizers) for designing and training a wide range of deep learning models, catering to both researchers and developers.

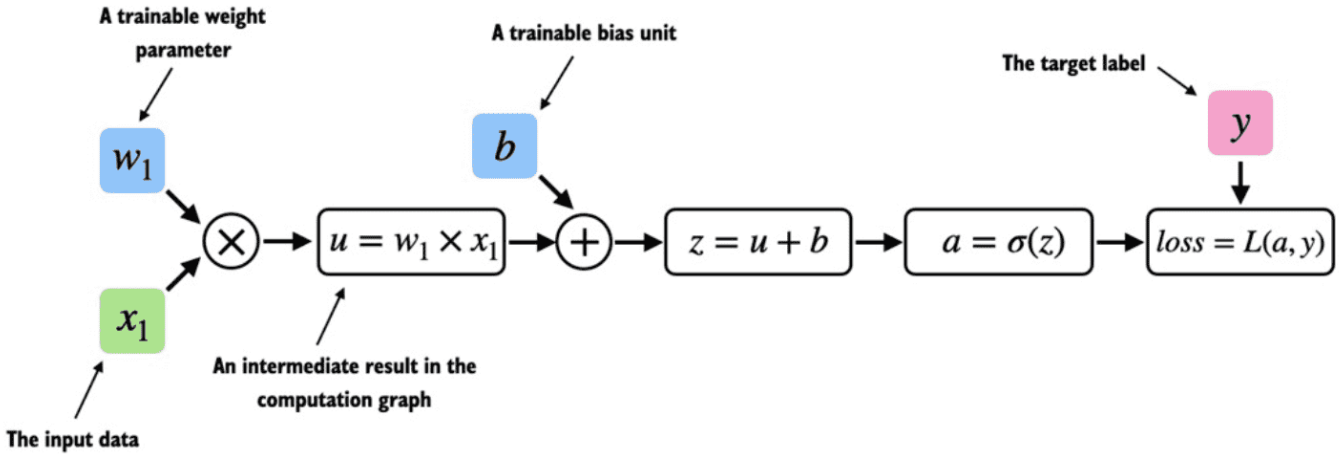
A.2 Understanding tensors



- PyTorch's has a NumPy-like API.

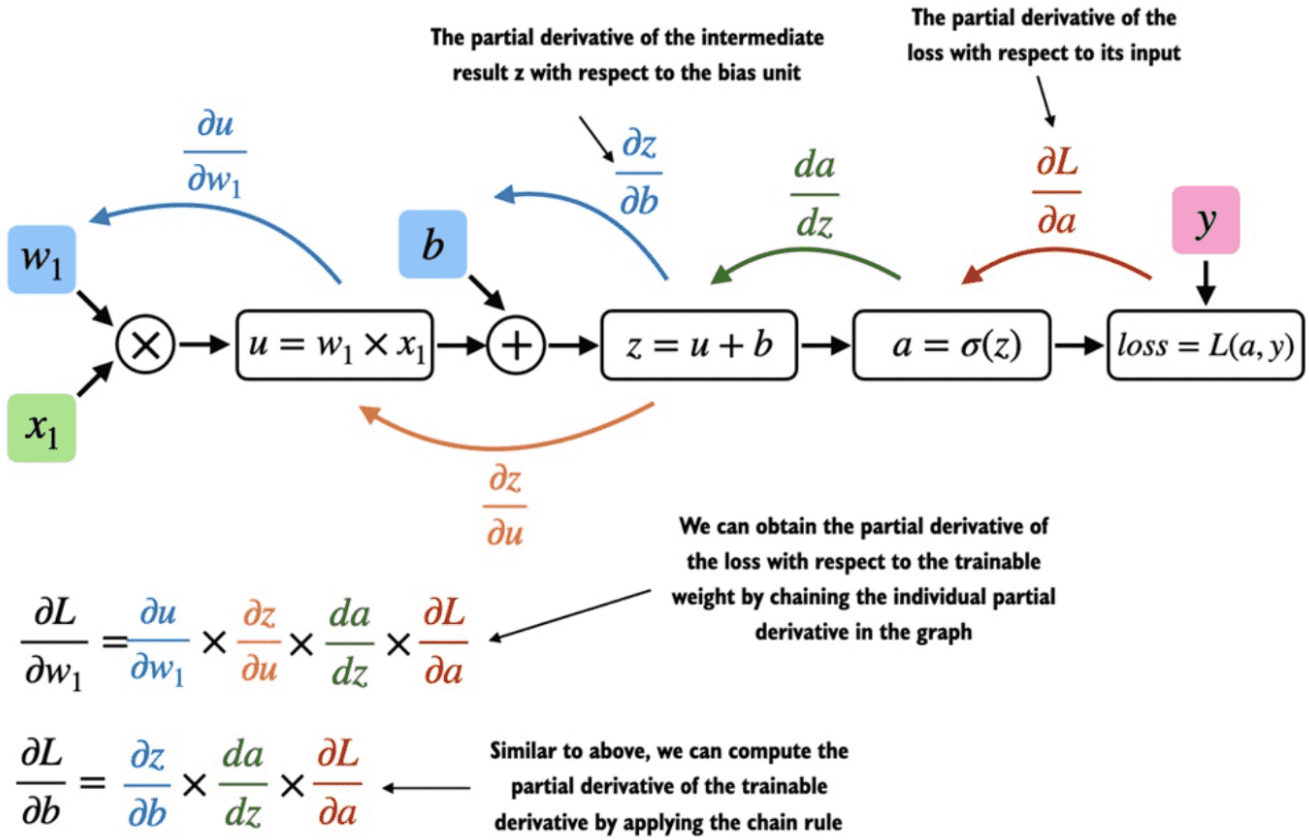
A.3 Seeing models as computation graphs

- Automatic differentiation engine, also known as autograd. PyTorch's autograd system provides functions to compute gradients in dynamic computational graphs automatically.
- Computation graph is a directed graph that allows us to express and visualize mathematical expressions.
- In the context of deep learning, a computation graph lays out the sequence of calculations needed to compute the output of a neural network – we will need this later to compute the required gradients for backpropagation, which is the main training algorithm for neural networks.



- In fact, PyTorch builds such a computation graph in the background, and we can use this to calculate gradients of a loss function with respect to the model parameters (here w1 and b) to train the model.

A.4 Automatic differentiation made easy



Gradients are required when training neural networks via the popular backpropagation algorithm, which can be thought of as an implementation of the chain rule from calculus for neural networks.

Partial derivatives and gradients

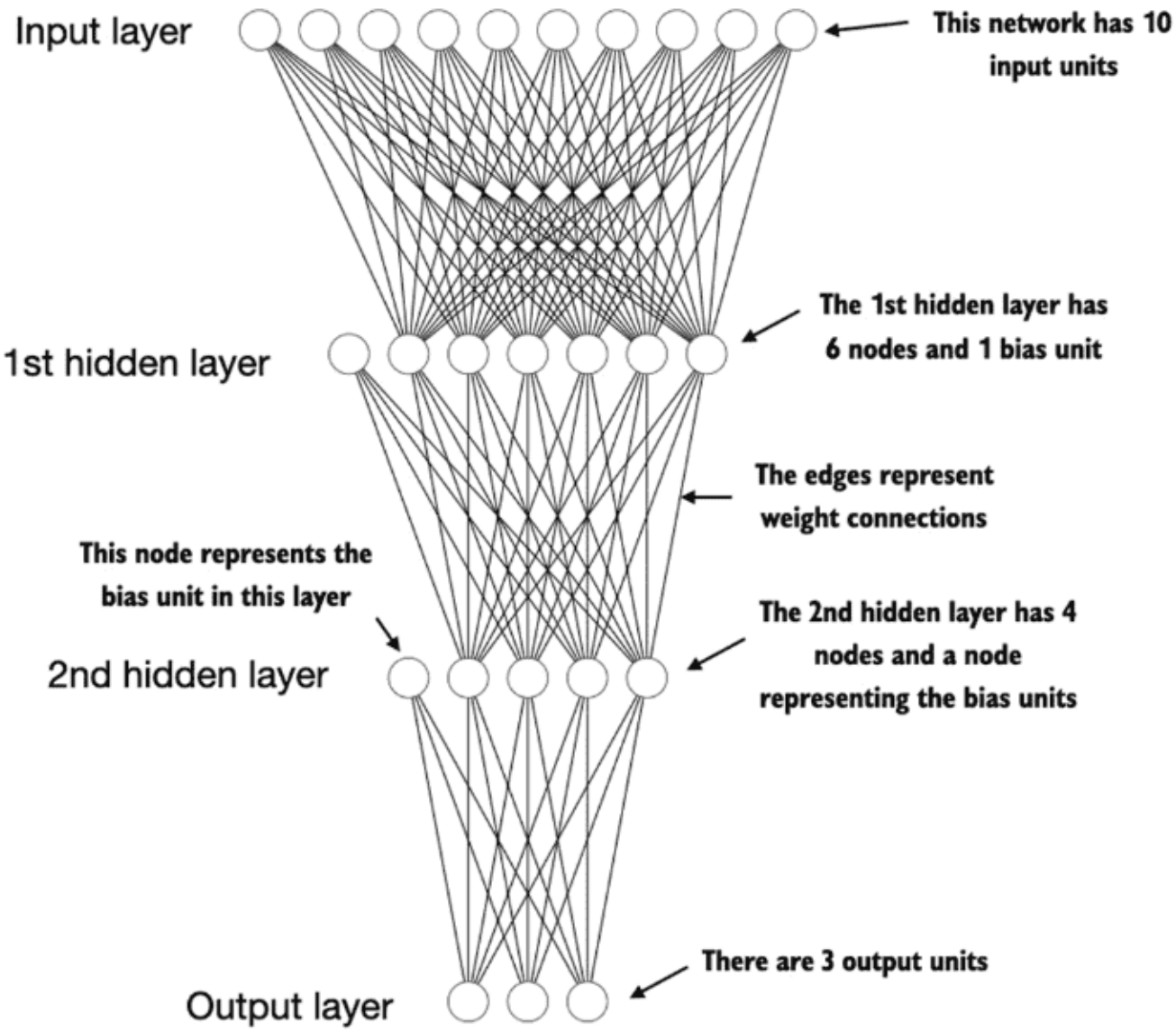
- A partial derivatives, which measure the rate at which a function changes with respect to one of its variables.
- A gradient is a vector containing all of the partial derivatives of a multivariate function, a function with more than one variable as input.
- This provides the information needed to update each parameter in a way that minimizes the loss function, which serves as a proxy for measuring the model's performance, using a method such as gradient descent.
- Listing A.3 Computing gradients via autograd

```
import torch.nn.functional as F
from torch.autograd import grad
y = torch.tensor([1.0])
x1 = torch.tensor([1.1])
w1 = torch.tensor([2.2], requires_grad=True)
b = torch.tensor([0.0], requires_grad=True)
z = x1 * w1 + b
a = torch.sigmoid(z)

loss = F.binary_cross_entropy(a, y)
grad_L_w1 = grad(loss, w1, retain_graph=True)  #A
grad_L_b = grad(loss, b, retain_graph=True)
```

PyTorch provides even more high-level tools to automate this process:

```
loss.backward()
print(w1.grad)
print(b.grad)
```

A.5 Implementing multilayer neural networks



Listing A.4 A multilayer perceptron with two hidden layers:

```
class NeuralNetwork(torch.nn.Module):
    def __init__(self, num_inputs, num_outputs):  #A
        super().__init__()
        self.layers = torch.nn.Sequential(
            # 1st hidden layer
            torch.nn.Linear(num_inputs, 30),  #B
            torch.nn.ReLU(),  #C
            # 2nd hidden layer
```

```
            torch.nn.Linear(30, 20),  #D
            torch.nn.ReLU(),
            # output layer
            torch.nn.Linear(20, num_outputs),
        )
    def forward(self, x):
        logits = self.layers(x)
        return logits   #E
```

Instantiate a new neural network object:

```
>>> model = NeuralNetwork(50, 3)
NeuralNetwork(
  (layers): Sequential(
    (0): Linear(in_features=50, out_features=30, bias=True)
    (1): ReLU()
    (2): Linear(in_features=30, out_features=20, bias=True)
    (3): ReLU()
    (4): Linear(in_features=20, out_features=3, bias=True)
  )
)
```

Check the total number of trainable parameters:

```
>>> sum(p.numel() for p in model.parameters() if p.requires_grad)
2213

# Manual Calculation:
# First hidden layer: 50 inputs * 30 hidden units + 30 bias units.
#     50*30+30
# Second hidden layer: 30 input units * 20 nodes + 20 bias units.
#     30*20+20
# Output layer: 20 input nodes * 3 output nodes + 3 bias units.
#     20*3+3
# Total equals: 1530+620+63=2213
```

- A linear layer multiplies the inputs with a weight matrix and adds a bias vector ($wx+b$). This is sometimes also referred to as a `feedforward` or `fully connected layer`.

```
# weight parameter matrix
>>> print(model.layers[0].weight)

# bias parameter matrix
>>> print(model.layers[0].bias)
```

We can make the random number initialization reproducible by seeding PyTorch's random number generator:

```
torch.manual_seed(123)
model = NeuralNetwork(50, 3)
print(model.layers[0].weight)
```

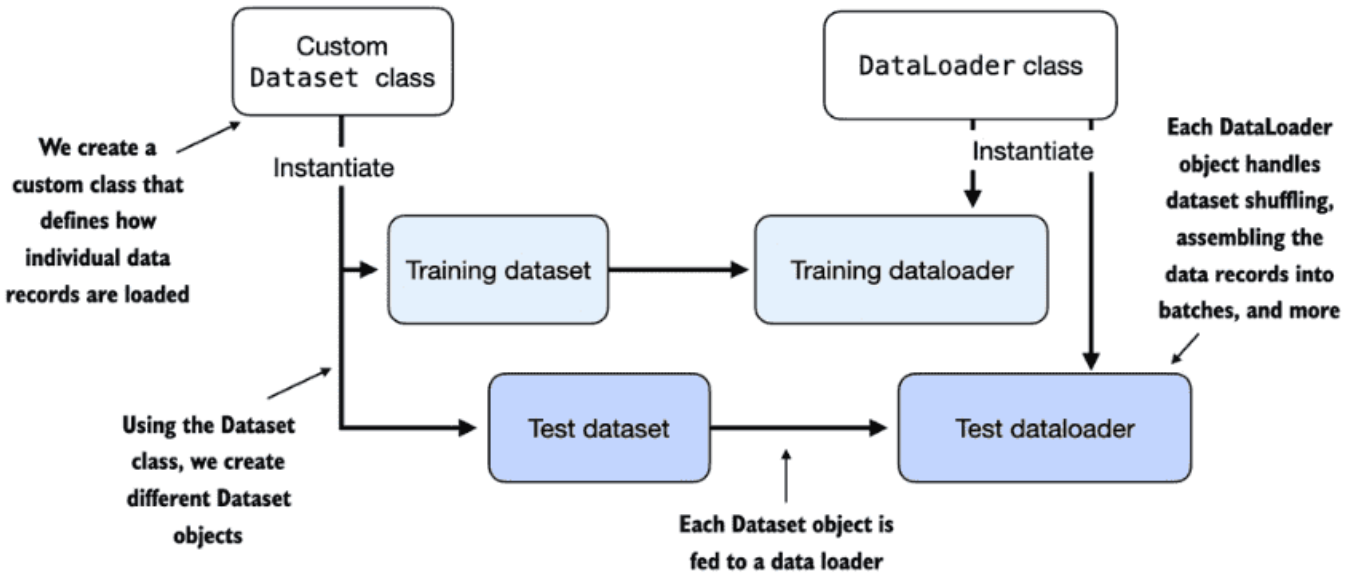
When using for inference rather than training, it is a best practice to use `torch.no_grad()` context manager:

```
# This tells PyTorch that it doesn't need to keep track of the gradients,
#   which can result in significant savings in memory and computation.
with torch.no_grad():
    out = model(X)
print(out)
```

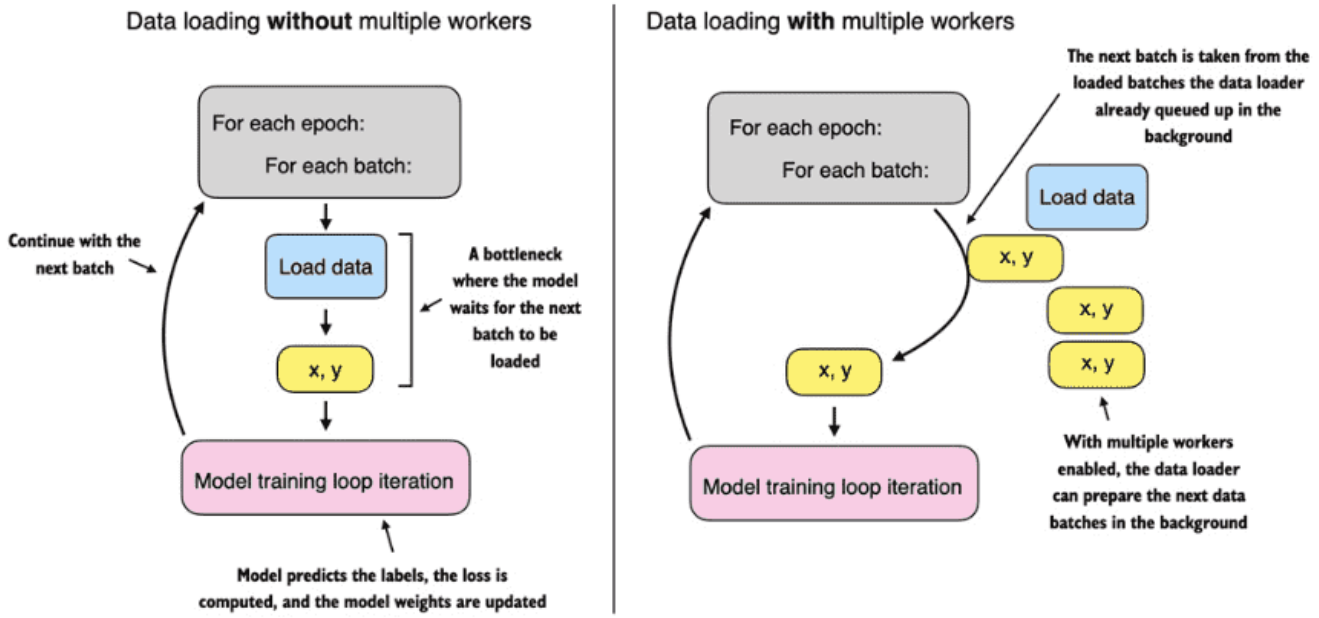
- In PyTorch, it's common practice to code models such that they return the outputs of the last layer (logits) without passing them to a nonlinear activation function.
- That's because PyTorch's commonly used loss functions combine the softmax (or sigmoid for binary classification) operation with the negative log-likelihood loss in a single class.
- The reason for this is numerical efficiency and stability.
- So, if we want to compute class-membership probabilities for our predictions, we have to call the softmax function explicitly:

```
with torch.no_grad():
    out = torch.softmax(model(X), dim=1)
print(out)
```

A.6 Setting up efficient data loaders



```
train_loader = DataLoader(
    dataset=train_ds,
    batch_size=2,
    shuffle=True,
    num_workers=0,      # crucial for parallelizing data loading and
preprocessing.
    drop_last=True      # drop the last batch in each epoch (because the quantity
in the last batch might not be enough)
)
```



> **Warning:** For Jupyter notebooks, setting num_workers to greater than 0 can sometimes lead to issues related to the sharing of resources between different processes, resulting in errors or notebook crashes.

> **Note:** Empirically, setting num_workers=4 often yields the best performance on many real-world datasets, but the optimal setting depends on your hardware and the code used to load the training examples defined in the Dataset class.

A.7 A typical training loop

Listing A.9 Neural network training in PyTorch:

```
import torch.nn.functional as F

torch.manual_seed(123)
model = NeuralNetwork(num_inputs=2, num_outputs=2) #A
optimizer = torch.optim.SGD(model.parameters(), lr=0.5) #B

num_epochs = 3
for epoch in range(num_epochs):
    model.train()
```

```
    for batch_idx, (features, labels) in enumerate(train_loader):
        logits = model(features)
        loss = F.cross_entropy(logits, labels)
        optimizer.zero_grad() #C
        loss.backward() #D
        optimizer.step() #E
        ### LOGGING
        print(f"Epoch: {epoch+1:03d}/{num_epochs:03d}"
            f" | Batch {batch_idx:03d}/{len(train_loader):03d}"
            f" | Train Loss: {loss:.2f}")
    model.eval()
    # Optional model evaluation
```

Listing A.10 A function to compute the prediction accuracy:

```
def compute_accuracy(model, dataloader):
    model = model.eval()
    correct = 0.0
    total_examples = 0
    for idx, (features, labels) in enumerate(dataloader):
        with torch.no_grad():
            logits = model(features)
        predictions = torch.argmax(logits, dim=1)
        compare = labels == predictions #A
        correct += torch.sum(compare) #B
        total_examples += len(compare)
    return (correct / total_examples).item() #C
```

A.8 Saving and loading models

Model saving:

```
torch.save(model.state_dict(), "model.pth")

# model.state_dict is a Python dictionary object that maps each layer in the model
to its trainable parameters (weights and biases)
```

Model loading:

```
model = NeuralNetwork(2, 2)  # The architecture must exactly match the original
saved model
model.load_state_dict(torch.load("model.pth"))
```

A.9 Optimizing training performance with GPUs

A.9.1 PyTorch computations on GPU devices

CPU:

```
tensor_1 = torch.tensor([1., 2., 3.])
tensor_2 = torch.tensor([4., 5., 6.])
print(tensor_1 + tensor_2)

# Output
# tensor([5., 7., 9.])
```

GPU:

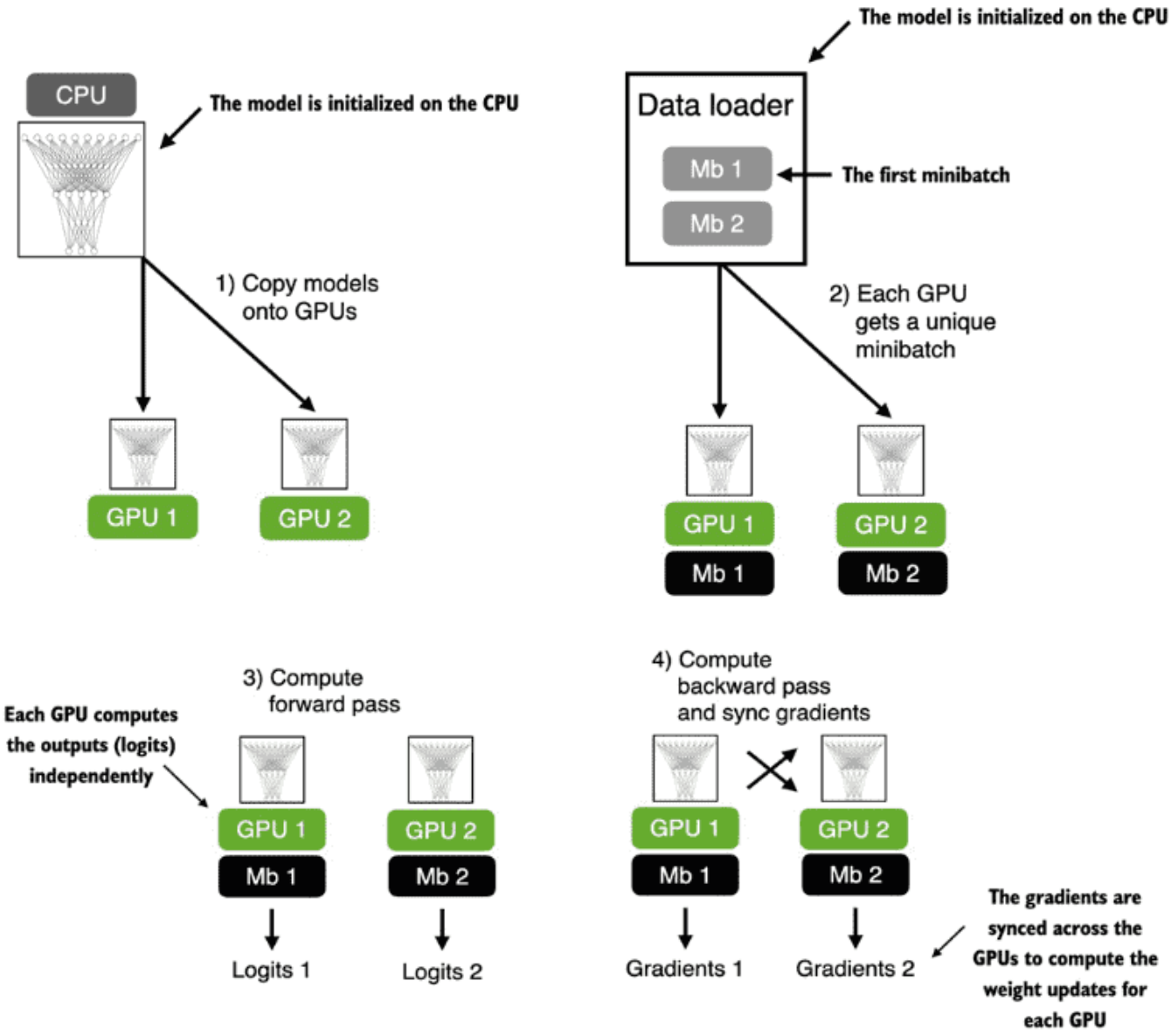
```
tensor_1 = tensor_1.to("cuda")
tensor_2 = tensor_2.to("cuda")
print(tensor_1 + tensor_2)

# Output
# tensor([5., 7., 9.], device='cuda:0')
```

Calculate CPU & GPU computation speed:

```
# CPU
a = torch.rand(100, 200)
b = torch.rand(200, 300)
%timeit a@b

# GPU
a, b = a.to("cuda"), b.to("cuda")
%timeit a @ b
```

A.9.3 Training with multiple GPUs

- Distributed training is the concept of dividing model training across multiple GPUs and machines.



> **Note:** DDP does not function properly within interactive Python environments like Jupyter notebooks, which don't handle multiprocessing in the same way a standalone Python script does.

If your machine has four GPUs and you only want to use the first and third GPUs:

```
CUDA_VISIBLE_DEVICES=0,2 python some_script.py
```

> **Note:** Empirically, setting num_workers=4 usually results in the best performance on many real-world datasets, but the optimal setting depends on your hardware and the code used to load the training examples defined in the Dataset class.

A.10 Summary

- PyTorch is an open-source library that consists of three core components: a `tensor library`, `automatic differentiation functions`, and `deep learning utilities`.
- PyTorch's tensor library is similar to array libraries like NumPy.

- In the context of PyTorch, tensors are array-like data structures to represent `scalars`, `vectors`, `matrices`, and `higher-dimensional arrays`. PyTorch tensors can be executed on the CPU, but one major advantage of PyTorch's tensor format is its GPU support to accelerate computations.
- The `automatic differentiation (autograd)` capabilities in PyTorch allow us to conveniently train neural networks using backpropagation without manually deriving gradients.
- The deep learning utilities in PyTorch provide building blocks for creating custom deep neural networks.
- PyTorch includes `Dataset` and `DataLoader` classes to set up efficient data loading pipelines.
- It's easiest to train models on a CPU or single GPU.
- Using `DistributedDataParallel` is the simplest way in PyTorch to accelerate the training if multiple GPUs are available.

Appendix B. References and Further Reading

Chapter 1: Understanding LLM

- "Attention Is All You Need" (2017) by Vaswani et al., https://arxiv.org/abs/1706.03762
- "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding" (2018) by Devlin et al., https://arxiv.org/abs/1810.04805
- The paper describing the decoder-style GPT-3 model, which inspired modern LLMs and will be used as a template for implementing an LLM from scratch in this book, is "Language Models are Few-Shot Learners" (2020) by Brown et al., https://arxiv.org/abs/2005.14165
- The following covers the original vision transformer for classifying images, which illustrates that transformer architectures are not only restricted to text inputs: "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale" (2020) by Dosovitskiy et al., https://arxiv.org/abs/2010.11929
- Meta AI's model is a popular implementation of a GPT-like model that is openly available in contrast to GPT-3 and ChatGPT: "Llama 2: Open Foundation and Fine-Tuned Chat Models" (2023) by Touvron et al., https://arxiv.org/abs/2307.092881
- The following paper provides the reference for InstructGPT for fine-tuning GPT-3: "Training Language Models to Follow Instructions with Human Feedback" (2022) by Ouyang et al., https://arxiv.org/abs/2203.02155
- For readers interested in additional details about the dataset references in section 1.5, this paper describes the publicly available The Pile dataset curated by Eleuther AI: "The Pile: An 800GB Dataset of Diverse Text for Language Modeling" (2020) by Gao et al., https://arxiv.org/abs/2101.00027

Chapter 2: Working with Text Data

- The code for the byte pair encoding tokenizer used to train GPT- 2 was open-sourced by OpenAI: https://github.com/openai/gpt-2/blob/master/src/encoder.py
- OpenAI provides an interactive web UI to illustrate how the byte pair tokenizer in GPT models works: https://platform.openai.com/tokenizer
- "A Minimal Implementation of a BPE Tokenizer," https://github.com/karpathy/minbpe

Chapter 3: Coding Attention Mechanisms

- "A Minimal Implementation of a BPE Tokenizer," https://github.com/karpathy/minbpe
- The concept of self-attention as scaled dot-product attention was introduced in the original transformer paper: "Attention Is All You Need" (2017) by Vaswani et al.,

https://arxiv.org/abs/1706.03762

- FlashAttention is a highly efficient implementation of a self- attention mechanism, which accelerates the computation process by optimizing memory access patterns. FlashAttention is mathematically the same as the standard self-attention mechanism but optimizes the computational process for efficiency:
  - "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness" (2022) by Dao et al., https://arxiv.org/abs/2205.14135
  - "FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning" (2023) by Dao, https://arxiv.org/abs/2307.08691
- "Dropout: A Simple Way to Prevent Neural Networks from Overfitting" (2014) by Srivastava et al., https://jmlr.org/papers/v15/srivastava14a.html

Chapter 4: Implementing a GPT model

- "Layer Normalization" (2016) by Ba, Kiros, and Hinton, https://arxiv.org/abs/1607.06450
- "On Layer Normalization in the Transformer Architecture" (2020) by Xiong et al., https://arxiv.org/abs/2002.04745
- "Gaussian Error Linear Units (GELUs)" (2016) by Hendricks and Gimpel, https://arxiv.org/abs/1606.08415
- NanoGPT is a code repository with a minimalist yet efficient implementation of a GPT-2 model, similar to the model implemented in this book. "NanoGPT, a Repository for Training Medium-Sized GPTs, https://github.com/karpathy/nanoGPT

Chapter 5: Pretraining on Unlabeled Data

- L8.2 Logistic Regression Loss Function, https://www.youtube.com/watch?v=GxJe0DZvydM
- L8.7.1 OneHot Encoding and Multi-category Cross Entropy, https://www.youtube.com/watch?v=4n71-tZ94yk
- Understanding Onehot Encoding and Cross Entropy in PyTorch, https://mng.bz/o05v
- The following two papers detail the dataset, hyperparameter, and architecture details used for pretraining LLMs:
  - "Pythia: A Suite for Analyzing Large Language Models Across Training and Scaling" (2023) by Biderman et al., https://arxiv.org/abs/2304.01373
  - "OLMo: Accelerating the Science of Language Models" (2024) by Groeneveld et al., https://arxiv.org/abs/2402.00838
- The paper that originally introduced top-k sampling is "Hierarchical Neural Story Generation" (2018) by Fan et al., https://arxiv.org/abs/1805.04833
- Top-p sampling, https://en.wikipedia.org/wiki/Top-p_sampling
- "Diverse Beam Search: Decoding Diverse Solutions from Neural Sequence Models" (2016) by Vijayakumar et al., https://arxiv.org/abs/1610.02424

Chapter 6: Fine-tuning for classification

- Different types of fine- tuning are:
  - "Using and Finetuning Pretrained Transformers," https://mng.bz/VxJG
  - "Finetuning Large Language Models," https://mng.bz/x28X
- Additional spam classification experiments (fine-tuning the first output token versus the last output token), https://mng.bz/AdJx

Chapter 7: Fine-tuning to follow instructions

- Datasets for instruction fine-tuning:
  - "Stanford Alpaca: An Instruction-Following Llama Model," https://github.com/tatsu-lab/stanford_alpaca
- Preference fine-tuning is an optional step after instruction fine- tuning to align the LLM more closely with human preferences. The following articles by the author provide more information about this process:
  - "LLM Training: RLHF and Its Alternatives," https://mng.bz/ZVPm
  - "Tips for LLM Pretraining and Evaluating Reward Models," https://mng.bz/RNXj

Appendix A: PyTorch

- If you want to learn more about model evaluation in machine learning, I recommend my article "Model Evaluation, Model Selection, and Algorithm Selection in Machine Learning" (2018) by Sebastian Raschka, https://arxiv.org/abs/1811.12808
- For readers who are interested in a refresher or gentle introduction to calculus, I've written a chapter on calculus that is freely available on my website: "Introduction to Calculus," by Sebastian Raschka, https://mng.bz/WEyW
- If you want to learn more about gradient accumulation, please see the following article: "Finetuning Large Language Models on a Single GPU Using Gradient Accumulation" by Sebastian Raschka, https://mng.bz/8wPD
- DDP, which is a popular approach for training deep learning models across multiple GPUs. "Introducing PyTorch Fully Sharded Data Parallel (FSDP) API," https://mng.bz/EZJR

Appendix C. Exercise Solutions

(Content omitted as requested)

Appendix D. Adding Bells and Whistles to the Training Loop

- In this appendix, we enhance the training function for the pretraining and fine-tuning processes covered in chapters 5 to 7.
- It covers:

```
learning rate warmup,
cosine decay,
gradient clipping.
```

D.1 Learning rate warmup

- When training complex models like LLMs, implementing learning rate warmup can help stabilize the training.
- In learning rate warmup, we gradually increase the learning rate from a very low value (`initial_lr`) to a user-specified maximum (`peak_lr` which is the original user-specified learning rate).
- This way, the model will start the training with small weight updates, which helps decrease the risk of large destabilizing updates during the training.

```
n_epochs = 15
initial_lr = 0.0001
peak_lr = 0.01
```

- Typically, the number of warmup steps is between 0.1% to 20% of the total number of steps.
- We can compute the increment as the difference between the `peak_lr` and `initial_lr` divided by the number of warmup steps.

```
total_steps = len(train_loader) * n_epochs
warmup_steps = int(0.2 * total_steps)   # 20% warmup
print(warmup_steps)


lr_increment = (peak_lr - initial_lr) / warmup_steps

global_step = -1
track_lrs = []

optimizer = torch.optim.AdamW(model.parameters(), weight_decay=0.1)

for epoch in range(n_epochs):
    for input_batch, target_batch in train_loader:
        optimizer.zero_grad()
        global_step += 1

        if global_step < warmup_steps:
            lr = initial_lr + global_step * lr_increment
        else:
            lr = peak_lr

        # Apply the calculated learning rate to the optimizer
        for param_group in optimizer.param_groups:
            param_group["lr"] = lr
        track_lrs.append(optimizer.param_groups[0]["lr"])

        # Calculate loss and update weights
        # ...
```

D.2 Cosine decay

- Another popular technique for training complex deep neural networks is `cosine decay`, which also adjusts the learning rate across training epochs.
- In cosine decay, the learning rate follows a `cosine curve`, decreasing from its initial value to near zero following a half-cosine cycle.
- This gradual reduction is designed to slow the pace of learning as the model begins to improve its weights; it reduces the risk of overshooting minima as the training progresses, which is crucial for stabilizing the training in its later stages.

- Cosine decay is often preferred over linear decay for its smoother transition in learning rate adjustments, but linear decay is also used in practice (for example, OLMo: Accelerating the Science of Language Models).

```
import math

min_lr = 0.1 * initial_lr
track_lrs = []

lr_increment = (peak_lr - initial_lr) / warmup_steps
total_training_steps = len(train_loader) * n_epochs

global_step = -1

for epoch in range(n_epochs):
    for input_batch, target_batch in train_loader:
        optimizer.zero_grad()
        global_step += 1

        # Adjust the learning rate based on the current phase (warmup or cosine
annealing)
        if global_step < warmup_steps:
            # Linear warmup
            lr = initial_lr + global_step * lr_increment
        else:
            # Cosine annealing after warmup
            progress = ((global_step - warmup_steps) /
                        (total_training_steps - warmup_steps))
            lr = min_lr + (peak_lr - min_lr) * 0.5 * (1 + math.cos(math.pi *
progress))

        # Apply the calculated learning rate to the optimizer
        for param_group in optimizer.param_groups:
            param_group["lr"] = lr
        track_lrs.append(lr)  # Store the current learning rate

        # Calculate and backpropagate the loss
        # ...
```

D.3 Gradient clipping

- Gradient clipping is yet another technique used to stabilize the training when training LLMs.
- By setting a threshold, gradients exceeding this limit are scaled down to a maximum magnitude to ensure that the updates to the model's parameters during backpropagation remain within a manageable range.
- For instance, using the `max_norm=1.0` setting in PyTorch's `clip_grad_norm_` method means that the norm of the gradients is clipped such that their maximum norm does not exceed 1.0.
- The "norm" refers to a measure of the gradient vector's length (or magnitude) in the parameter space of the model.
- Specifically, it's the L2 norm, also known as the Euclidean norm.

- Mathematically, for a vector $\mathbf{v}$ with components $\mathbf{v} = [v_1, v_2, ..., v_n]$
- the L2 norm is defined as:

$$| \mathbf{v} |_2 = \sqrt{v_1^2 + v_2^2 + ... + v_n^2}$$

- The L2 norm is calculated similarly for matrices.
- Let's assume our gradient matrix is:

$$G = \begin{bmatrix} 1 & 2 \ 2 & 4 \end{bmatrix}$$

- And we want to clip these gradients with a `max_norm` of 1.
- First, we calculate the L2 norm of these gradients:

$$ |G|_ 2 = \sqrt{1^2 + 2^2 + 2^2 + 4^2} = \sqrt{25} = 5 $$

- Since $|G|_ 2 = 5$ is greater than our `max_norm` of 1, we need to scale down the gradients so that their norm is exactly 1. The scaling factor is calculated as

$$ \frac{\text{max_norm}}{|G|_2} = \frac{1}{5} $$

- Therefore, the scaled gradient matrix `G'` will be as follows:

$$ G' = \frac{1}{5} \times G = \begin{bmatrix} \frac{1}{5} & \frac{2}{5} \ \frac{2}{5} & \frac{4}{5} \end{bmatrix} $$

- If we call `.backward()`, PyTorch will calculate the gradients and store them in a `.grad` attribute for each weight (parameter) matrix:

```
loss = calc_loss_batch(input_batch, target_batch, model, device)
loss.backward()
```

- Let's define a utility function to calculate the highest gradient based on all model weights.

```
def find_highest_gradient(model):
    max_grad = None
    for param in model.parameters():
        if param.grad is not None:
            grad_values = param.grad.data.flatten()
            max_grad_param = grad_values.max()
            if max_grad is None or max_grad_param > max_grad:
                max_grad = max_grad_param
    return max_grad

print(find_highest_gradient(model))
# Output
# tensor(0.0411)
```

- Applying gradient clipping, we can see that the largest gradient is now substantially smaller:

```
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
print(find_highest_gradient(model))
# Output
# tensor(0.0185)
```

D.4 The modified training function

- Now let's add the three concepts covered above (`learning rate warmup`, `cosine decay`, and `gradient clipping`) to the `train_model_simple` function covered in chapter 5 to create the more sophisticated `train_model` function below:

```
from previous_chapters import evaluate_model, generate_and_print_sample

BOOK_VERSION = True


def train_model(model, train_loader, val_loader, optimizer, device,
                n_epochs, eval_freq, eval_iter, start_context, tokenizer,
                warmup_steps, initial_lr=3e-05, min_lr=1e-6):

    train_losses, val_losses, track_tokens_seen, track_lrs = [], [], [], []
    tokens_seen, global_step = 0, -1

    # Retrieve the maximum learning rate from the optimizer
    peak_lr = optimizer.param_groups[0]["lr"]

    # Calculate the total number of iterations in the training process
    total_training_steps = len(train_loader) * n_epochs

    # Calculate the learning rate increment during the warmup phase
    lr_increment = (peak_lr - initial_lr) / warmup_steps

    for epoch in range(n_epochs):
        model.train()
        for input_batch, target_batch in train_loader:
            optimizer.zero_grad()
            global_step += 1

            # Adjust the learning rate based on the current phase (warmup or
cosine annealing)
            if global_step < warmup_steps:
                # Linear warmup
                lr = initial_lr + global_step * lr_increment
            else:
                # Cosine annealing after warmup
                progress = ((global_step - warmup_steps) /
                            (total_training_steps - warmup_steps))
                lr = min_lr + (peak_lr - min_lr) * 0.5 * (1 + math.cos(math.pi *
progress))
```

```
            # Apply the calculated learning rate to the optimizer
            for param_group in optimizer.param_groups:
                param_group["lr"] = lr
            track_lrs.append(lr)  # Store the current learning rate

            # Calculate and backpropagate the loss
            loss = calc_loss_batch(input_batch, target_batch, model, device)
            loss.backward()

            # Apply gradient clipping after the warmup phase to avoid exploding
gradients

            if BOOK_VERSION:
                if global_step > warmup_steps:
                    torch.nn.utils.clip_grad_norm_(model.parameters(),
max_norm=1.0)
            else:
                if global_step >= warmup_steps:  # the book originally used
global_step > warmup_steps, which lead to a skipped clipping step after warmup
                    torch.nn.utils.clip_grad_norm_(model.parameters(),
max_norm=1.0)

            optimizer.step()
            tokens_seen += input_batch.numel()

            # Periodically evaluate the model on the training and validation sets
            if global_step % eval_freq == 0:
                train_loss, val_loss = evaluate_model(
                    model, train_loader, val_loader,
                    device, eval_iter
                )
                train_losses.append(train_loss)
                val_losses.append(val_loss)
                track_tokens_seen.append(tokens_seen)
                # Print the current losses
                print(f"Ep {epoch+1} (Iter {global_step:06d}): "
                      f"Train loss {train_loss:.3f}, "
                      f"Val loss {val_loss:.3f}"
                )

        # Generate and print a sample from the model to monitor progress
        generate_and_print_sample(
            model, tokenizer, device, start_context
        )

    return train_losses, val_losses, track_tokens_seen, track_lrs
```

Usage:

```
peak_lr = 0.001  # this was originally set to 5e-4 in the book by mistake
optimizer = torch.optim.AdamW(model.parameters(), lr=peak_lr, weight_decay=0.1)  #
the book accidentally omitted the lr assignment
```

```
train_losses, val_losses, tokens_seen, lrs = train_model(
    model, train_loader, val_loader, optimizer, device, n_epochs=n_epochs,
    eval_freq=5, eval_iter=1, start_context="Every effort moves you",
    tokenizer=tokenizer, warmup_steps=warmup_steps,
    initial_lr=1e-5, min_lr=1e-5
)
# Output
# Ep 1 (Iter 000000): Train loss 10.934, Val loss 10.939
# Ep 1 (Iter 000005): Train loss 9.151, Val loss 9.461
# ...
# Ep 14 (Iter 000120): Train loss 0.038, Val loss 6.907
# Ep 14 (Iter 000125): Train loss 0.040, Val loss 6.912
# Ep 15 (Iter 000130): Train loss 0.041, Val loss 6.915
```

Appendix E Parameter-efficient fine- tuning with LoRA

- Low-rank adaptation (LoRA) is one of the most widely used techniques for parameter-efficient fine-tuning.

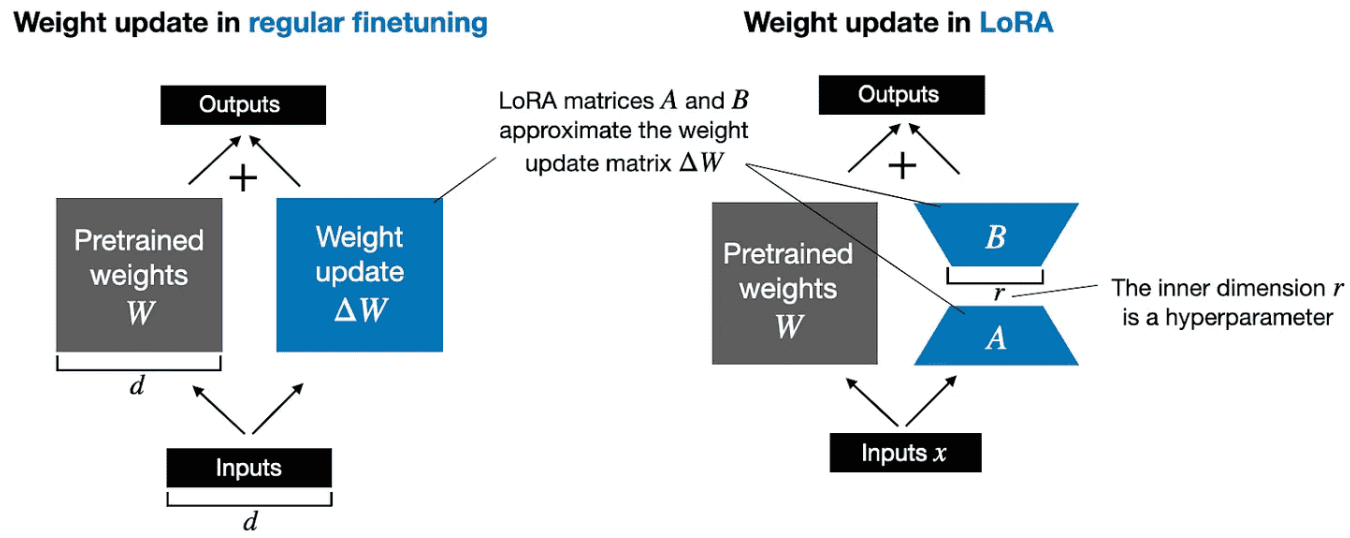
E.1 Introduction to LoRA

- LoRA is a technique that adapts a pretrained model to better suit a specific, often smaller dataset by adjusting only a small subset of the model's weight parameters.
- The LoRA method is useful and popular because it enables efficient fine-tuning of large models on task-specific data, significantly cutting down on the computational costs and resources usually required for fine-tuning.
- Suppose we have a large weight matrix `W` for a given layer.
- During backpropagation, we learn a ΔW matrix, which contains information on how much we want to update the original weights to minimize the loss function during training.
- In regular training and finetuning, the weight update is defined as follows:

$$W_{\text{updated}} = W + \Delta W$$

- The LoRA method offers a more efficient alternative to computing the weight updates ΔW by learning an approximation of it, $\Delta W \approx AB$.
- In other words, in LoRA, we have the following, where `A` and `B` are two small weight matrices:

$$W_{\text{updated}} = W + AB$$



- If you paid close attention, the full finetuning and LoRA depictions in the figure above look slightly different from the formulas I have shown earlier.
- That's due to the distributive law of matrix multiplication: we don't have to add the weights with the updated weights but can keep them separate.
- For instance, if `x` is the input data, then we can write the following for regular finetuning:

$$ x (W+\Delta W) = x W + x \Delta W \ \text{=> for LoRA:} \ x (W+A B) = x W + x A B $$

- The fact that we can keep the LoRA weight matrices separate makes LoRA especially attractive.
- In practice, this means that we don't have to modify the weights of the pretrained model at all, as we can apply the LoRA matrices on the fly.

E.2 Preparing the dataset

- Dataset URL: url = "https://archive.ics.uci.edu/static/public/228/sms+spam+collection.zip"

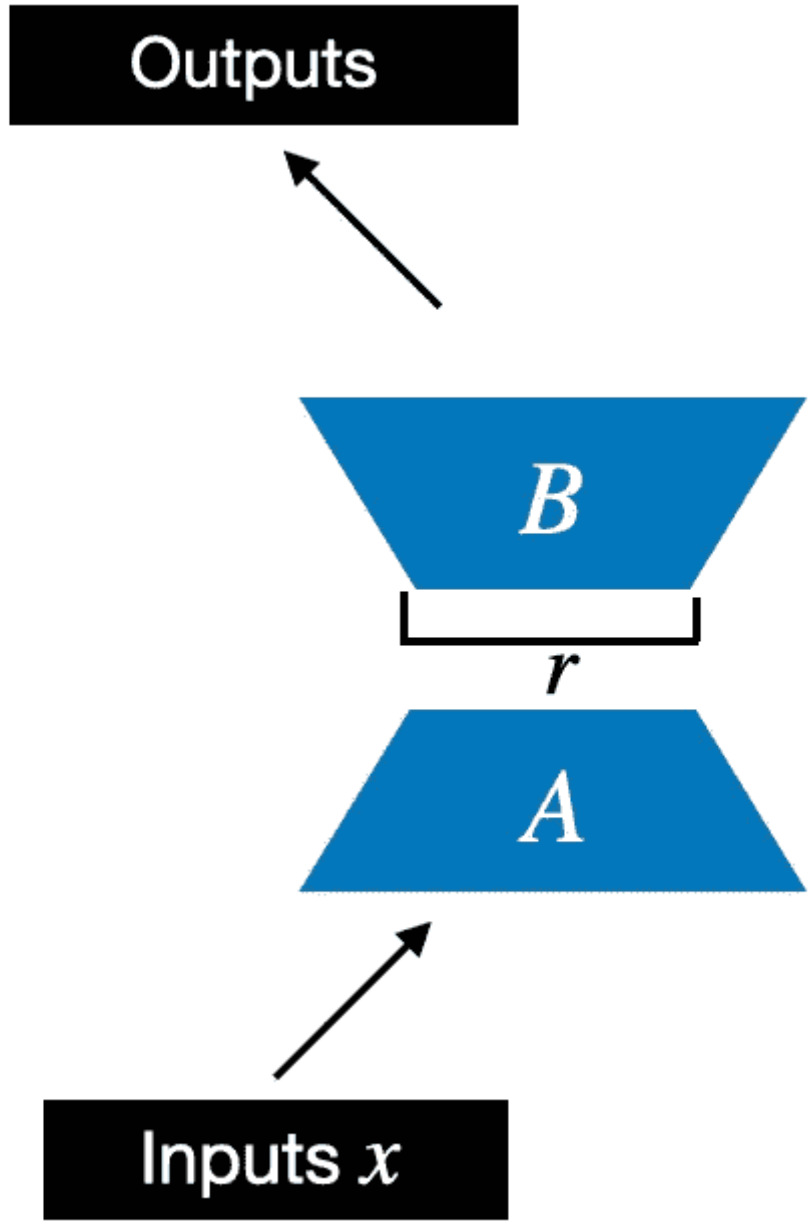
E.3 Initializing the model

- Replace the output layer similar to chapter 6:

```
num_classes = 2
model.out_head = torch.nn.Linear(in_features=768, out_features=num_classes)
```

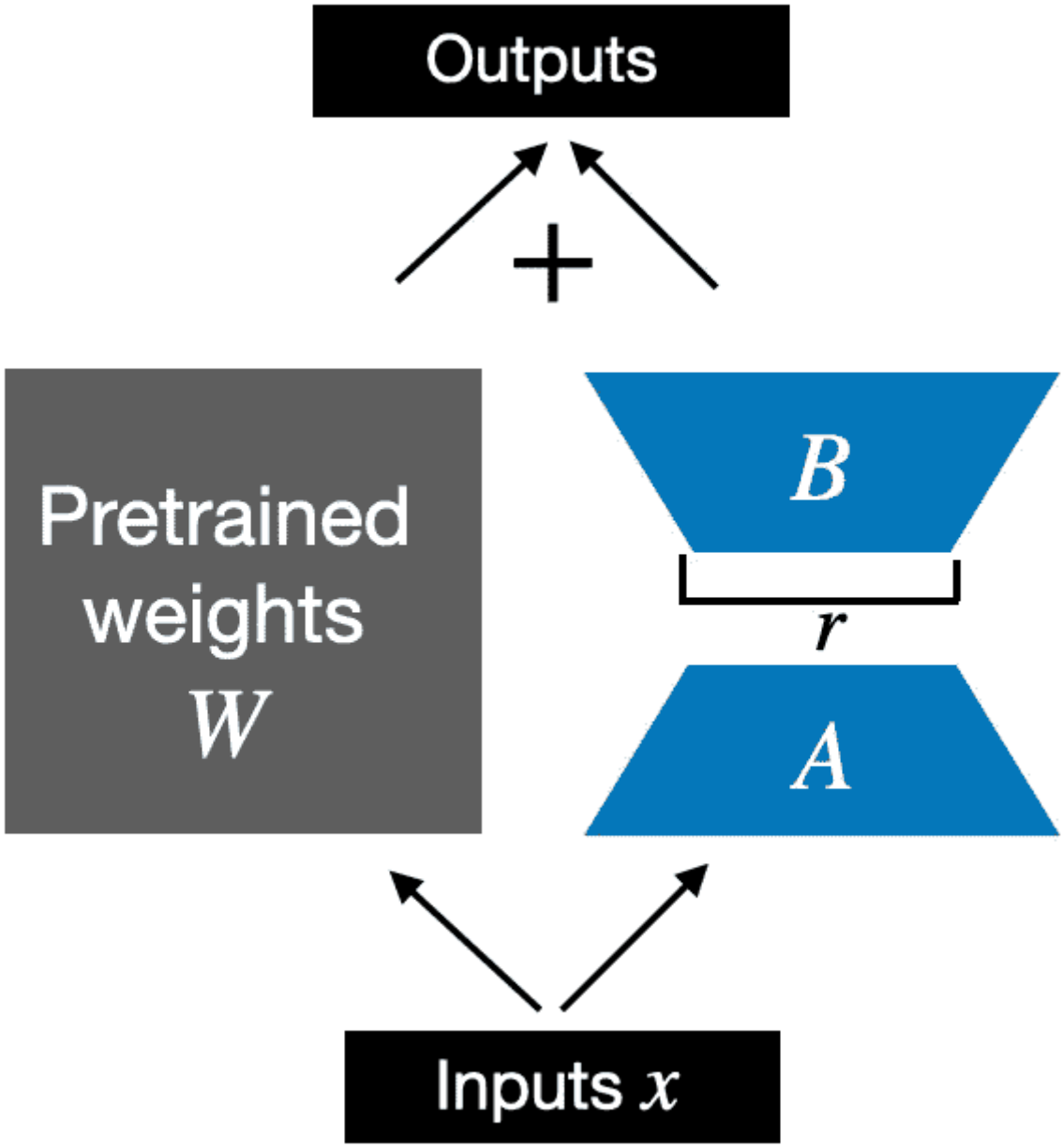
E.4 Parameter-efficient finetuning with LoRA

- We begin by initializing a LoRALayer that creates the matrices `A` and `B`, along with the `alpha` scaling hyperparameter and the `rank` (`r`) hyperparameters.




```
import math

class LoRALayer(torch.nn.Module):
    def __init__(self, in_dim, out_dim, rank, alpha):
        super().__init__()
        self.A = torch.nn.Parameter(torch.empty(in_dim, rank))
        torch.nn.init.kaiming_uniform_(self.A, a=math.sqrt(5))  # similar to
standard weight initialization
        self.B = torch.nn.Parameter(torch.zeros(rank, out_dim))  # since self.B is
0 initially
        self.alpha = alpha

    def forward(self, x):
        x = self.alpha * (x @ self.A @ self.B)
        return x
```

- In the code above, `rank` is a hyperparameter that controls the inner dimension of the matrices `A` and `B`.
- In other words, this parameter controls the number of additional parameters introduced by LoRA and is a key factor in determining the balance between model adaptability and parameter efficiency.
- The second hyperparameter, `alpha`, is a scaling hyperparameter applied to the output of the low-rank adaptation.
- It essentially controls the extent to which the adapted layer's output is allowed to influence the original output of the layer being adapted.
- This can be seen as a way to regulate the impact of the low-rank adaptation on the layer's output.
- So far, the `LoRALayer` class we implemented above allows us to transform the layer inputs `x`.
- However, in LoRA, we are usually interested in replacing existing `Linear` layers so that the weight update is applied to the existing pretrained weights, as shown in the figure below.



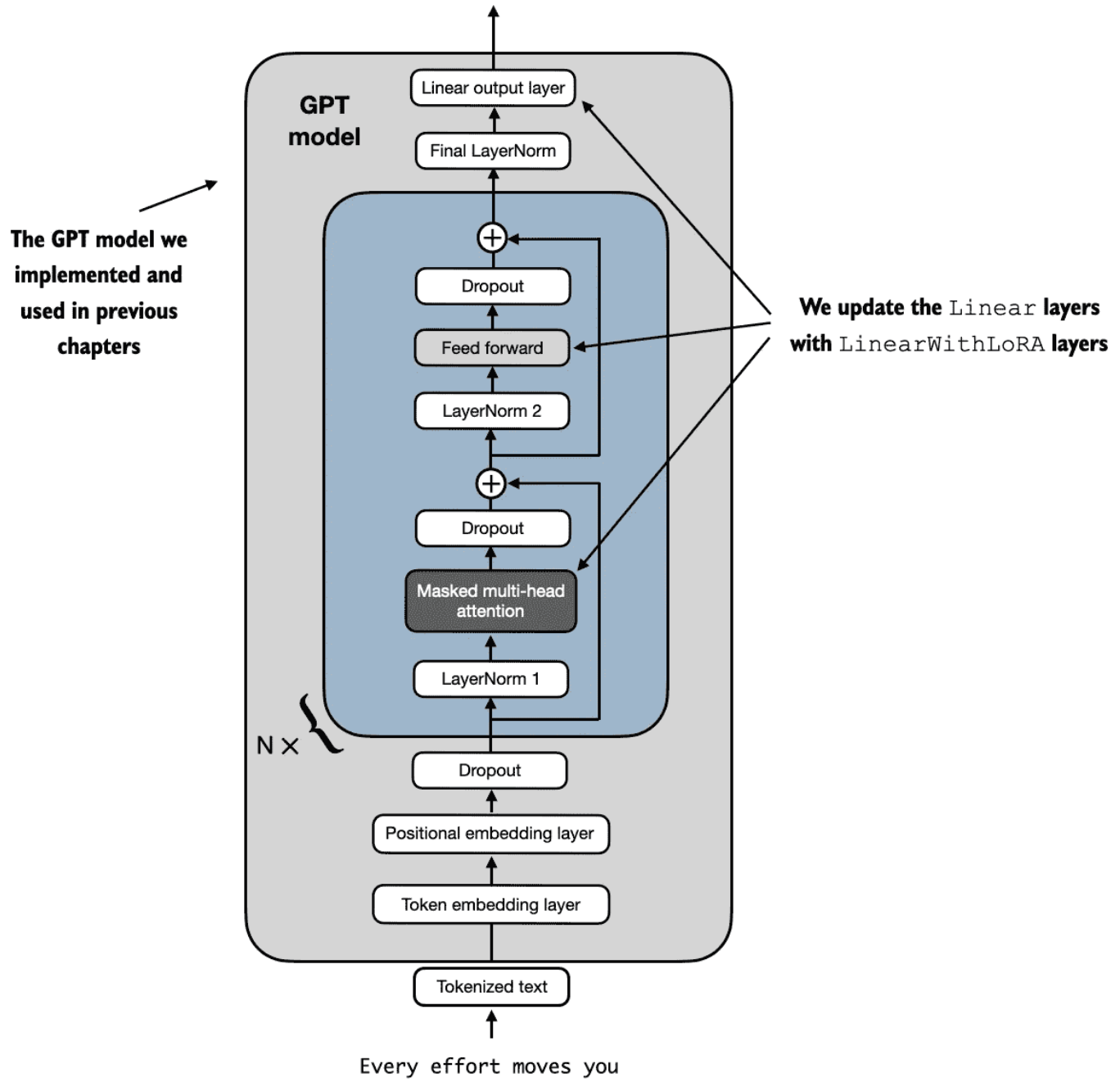
- To incorporate the original `Linear` layer weights as shown in the figure above, we implement a `LinearWithLoRA` layer below that uses the previously implemented LoRALayer and can be used to

replace existing `Linear` layers in a neural network, for example, the self-attention module or feed forward modules in an LLM.

```
class LinearWithLoRA(torch.nn.Module):
    def __init__(self, linear, rank, alpha):
        super().__init__()
        self.linear = linear
        self.lora = LoRALayer(
            linear.in_features, linear.out_features, rank, alpha
        )

    def forward(self, x):
        return self.linear(x) + self.lora(x)
```

- To try LoRA on the GPT model we defined earlier, we define a `replace_linear_with_lora` function to **replace all** `Linear` layers in the model with the new `LinearWithLoRA` layers.

```
def replace_linear_with_lora(model, rank, alpha):
    for name, module in model.named_children():
        if isinstance(module, torch.nn.Linear):
            # Replace the Linear layer with LinearWithLoRA
            setattr(model, name, LinearWithLoRA(module, rank, alpha))
        else:
            # Recursively apply the same function to child modules
            replace_linear_with_lora(module, rank, alpha)
```



- We then **freeze** the original model parameter and use the `replace_linear_with_lora` to replace the said `Linear` layers using the code below.
- This will replace the `Linear` layers in the LLM with `LinearWithLoRA` layers.

```
total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Total trainable parameters before: {total_params:,}")

for param in model.parameters():
    param.requires_grad = False

total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Total trainable parameters after: {total_params:,}")
# Output
# Total trainable parameters before: 124,441,346
# Total trainable parameters after: 0
```

```
replace_linear_with_lora(model, rank=16, alpha=16)

total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Total trainable LoRA parameters: {total_params:,}")
# Output
# Total trainable LoRA parameters: 2,666,528
```

- Print the current model structure:

```
GPTModel(
  (tok_emb): Embedding(50257, 768)
  (pos_emb): Embedding(1024, 768)
  (drop_emb): Dropout(p=0.0, inplace=False)
  (trf_blocks): Sequential(
    (0): TransformerBlock(
      (att): MultiHeadAttention(
        (W_query): LinearWithLoRA(
          (linear): Linear(in_features=768, out_features=768, bias=True)
          (lora): LoRALayer()
        )
        (W_key): LinearWithLoRA(
          (linear): Linear(in_features=768, out_features=768, bias=True)
          (lora): LoRALayer()
        )
        (W_value): LinearWithLoRA(
          (linear): Linear(in_features=768, out_features=768, bias=True)
          (lora): LoRALayer()
        )
        (out_proj): LinearWithLoRA(
          (linear): Linear(in_features=768, out_features=768, bias=True)
          (lora): LoRALayer()
        )
        (dropout): Dropout(p=0.0, inplace=False)
      )
      (ff): FeedForward(
        (layers): Sequential(
          (0): LinearWithLoRA(
            (linear): Linear(in_features=768, out_features=3072, bias=True)
            (lora): LoRALayer()
          )
          (1): GELU()
          (2): LinearWithLoRA(
            (linear): Linear(in_features=3072, out_features=768, bias=True)
            (lora): LoRALayer()
          )
        )
      )
      (norm1): LayerNorm()
      (norm2): LayerNorm()
      (drop_resid): Dropout(p=0.0, inplace=False)
```

```
    )
    ... ... # 11 skipped
  )
  (final_norm): LayerNorm()
  (out_head): LinearWithLoRA(
    (linear): Linear(in_features=768, out_features=2, bias=True)
    (lora): LoRALayer()
  )
)
```

- Same fine-tuning steps:

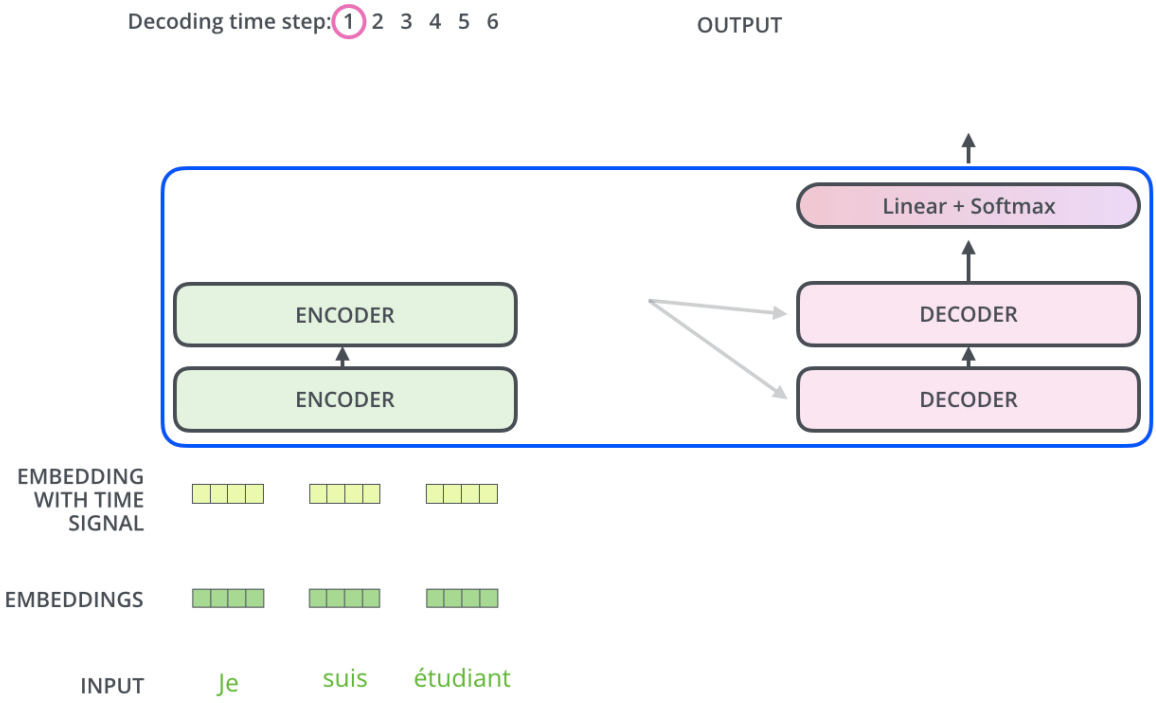
```
optimizer = torch.optim.AdamW(model.parameters(), lr=5e-5, weight_decay=0.1)

num_epochs = 5
train_losses, val_losses, train_accs, val_accs, examples_seen =
train_classifier_simple(
    model, train_loader, val_loader, optimizer, device,
    num_epochs=num_epochs, eval_freq=50, eval_iter=5,
)
```

The resulting accuracy values are:

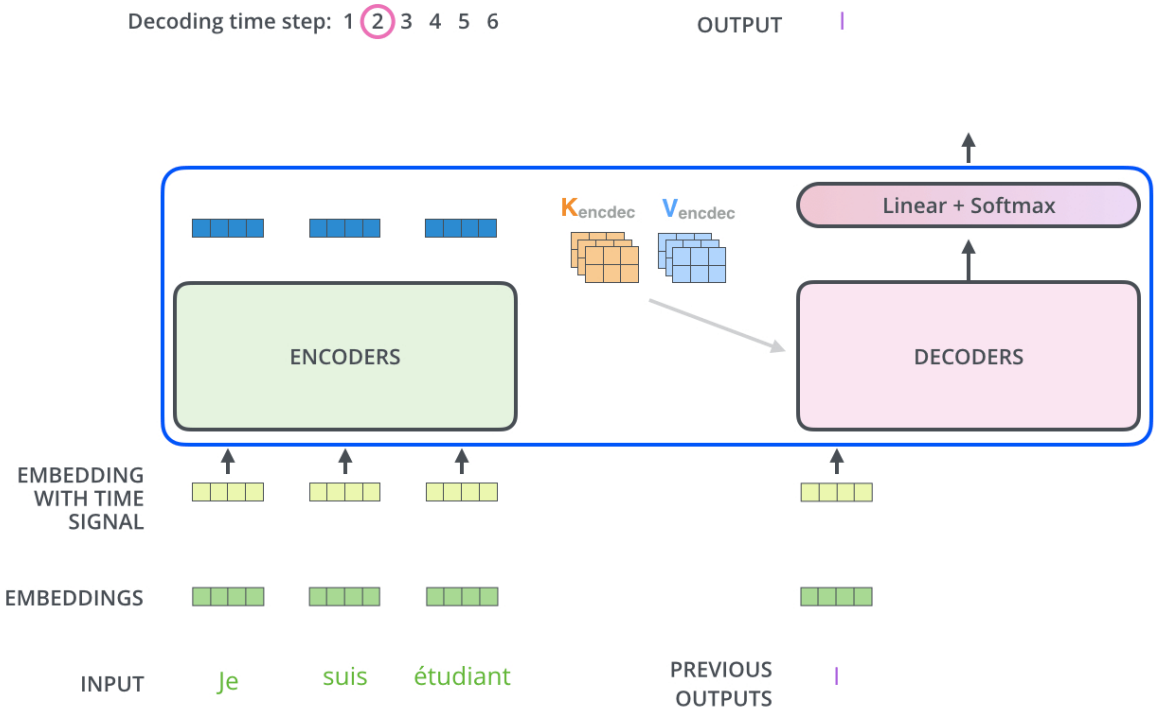
```
Training accuracy: 100.00%
Validation accuracy: 96.64%
Test accuracy: 98.00%
```

> **Note:** Conclusion: However, the slightly lower validation and test accuracies (96.64% and 97.33%, respectively) suggest a small degree of overfitting, as the model does not generalize quite as well on unseen data compared to the training set. Overall, the results are very impressive, considering we fine-tuned only a relatively small number of model weights (2.7 million LoRA weights instead of the original 124 million model weights).

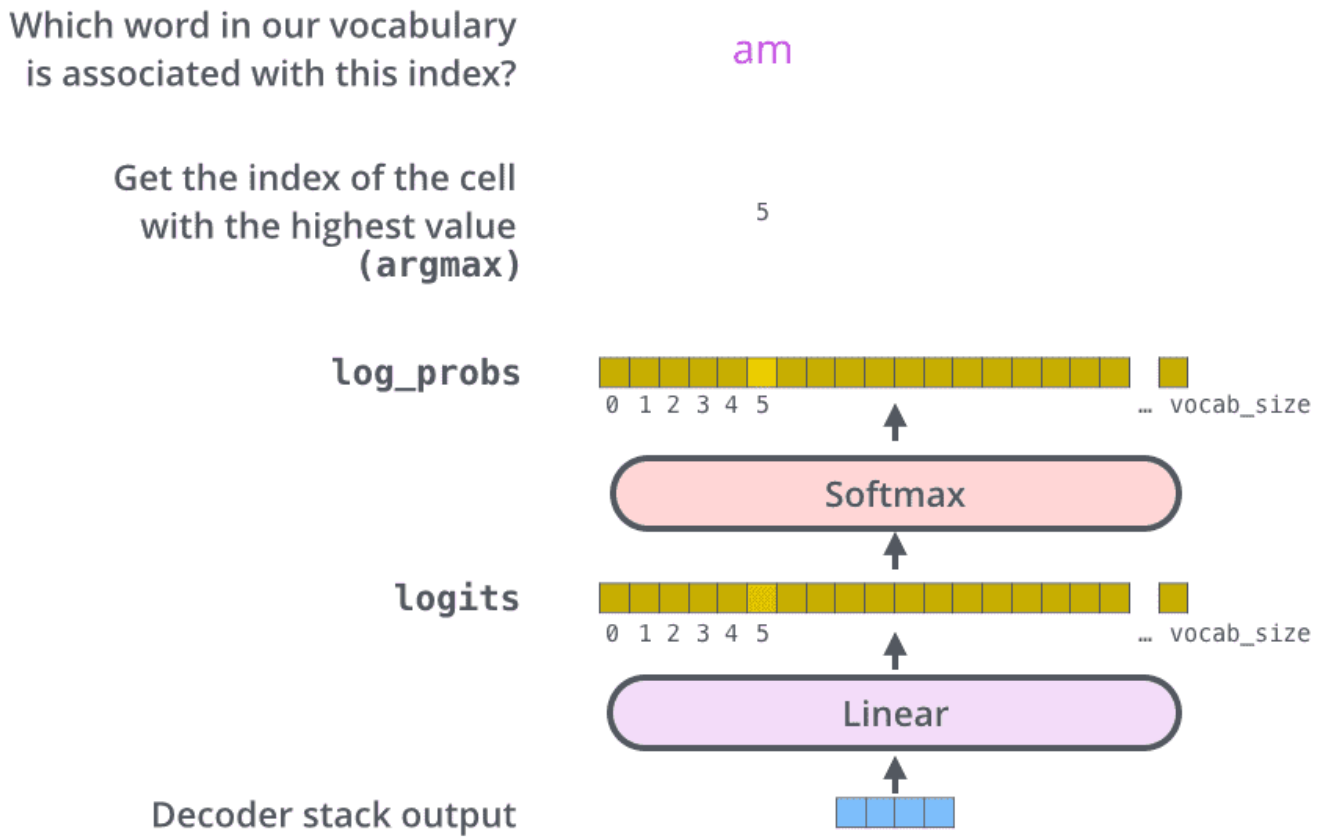
Other



The encoder start by processing the input sequence. The output of the top encoder is then transformed into a set of `attention vectors K and V`. These are to be used by each decoder in its "encoder-decoder attention" layer which helps the decoder focus on appropriate places in the input sequence



The following steps repeat the process until a special symbol is reached indicating the transformer decoder has completed its output. The output of each step is fed to the bottom decoder in the next time step, and the decoders bubble up their decoding results just like the encoders did. And just like we did with the encoder inputs, we embed and add positional encoding to those decoder inputs to indicate the position of each word.



The Linear layer is a simple fully connected neural network that projects the vector produced by the stack of decoders, into a much, much larger vector called a logits vector. The `softmax` layer then turns those scores into probabilities (all positive, all add up to 1.0). The cell with the highest probability is chosen, and the word associated with it is produced as the output for this time step.

About the Book

This book, "Build a Large Language Model (From Scratch)," provides a comprehensive guide to understanding and implementing modern language models. It walks readers through the entire process, from basic concepts to advanced techniques, covering tokenization, attention mechanisms, transformer architecture, pretraining, and fine-tuning approaches. The book is designed for both beginners and experienced practitioners looking to deepen their understanding of how LLMs work under the hood.

Free Version of Book

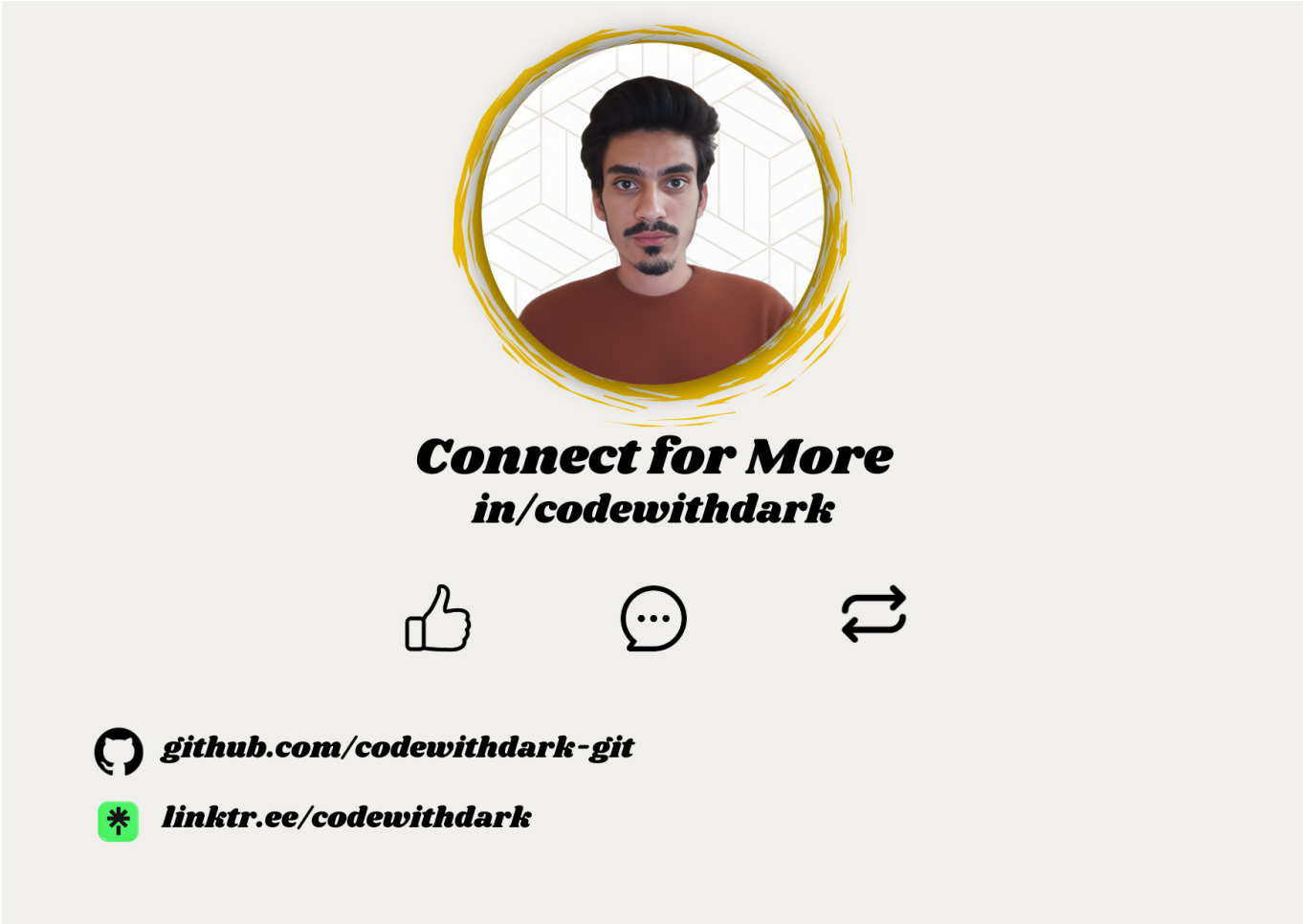
The free version of this book contains selected chapters and code examples that provide a solid foundation for understanding LLMs. For the complete experience, including all advanced topics, exercises, and additional resources, the full version is available at Free Version. The free version serves as an excellent starting point for those curious about LLM architecture and implementation.

About These Notes

These notes document my personal learning journey through LLM architecture and implementation, guided by the "Build a Large Language Model (From Scratch)" book. I've compiled key points, code examples, and my own insights while working through the material. The practical knowledge gained from this study has been invaluable for my own projects and research in AI.

I hope sharing these notes might be helpful to others on a similar path. If you have questions, feedback, or just want to connect, feel free to reach out:

- GitHub: github.com/Dark_Coder
- LinkedIn: linkedin.com/in/codewithdark
- Implementation : BuildingLLMs From Scratch
- Email: codewithdark90@gmail.com



Thank you for joining me on this learning journey through the fascinating world of large language models!