FROM SCRATCH

BUILD A

Large Language Model

Sebastian Raschka

MEAP

FROM SCRATCH

BUILD A

Large Language Model

MEAP

Build a Large Language Model (From Scratch)

welcome

Build a Large Language Model (From Scratch)

In this book

1 Understanding Large Language Models

This chapter covers

-
-
-

ó

1.1 What is an LLM?

transformer

generating

generative AI *GenAI*

Figure 1.1 As this hierarchical depiction of the relationship between the different fields suggests, LLMs represent a specific application of deep learning techniques, leveraging their ability to process and generate human-like text. Deep learning is a specialized branch of machine learning that focuses on using multi-layer neural networks. And machine learning and deep learning are fields aimed at implementing algorithms that enable computers to learn from data and perform tasks that typically require human intelligence.

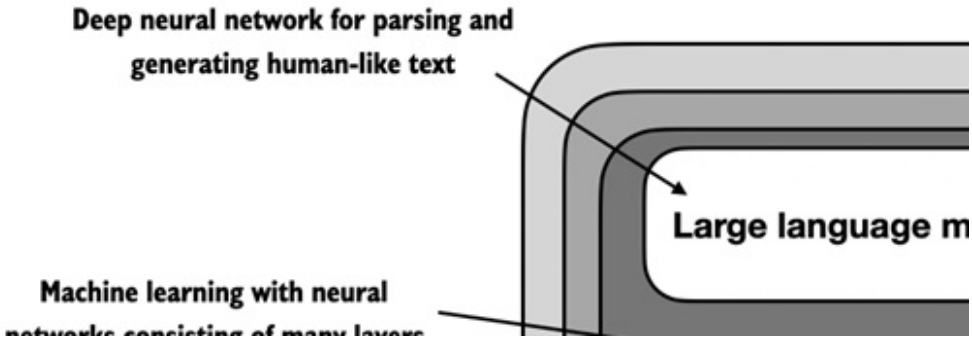
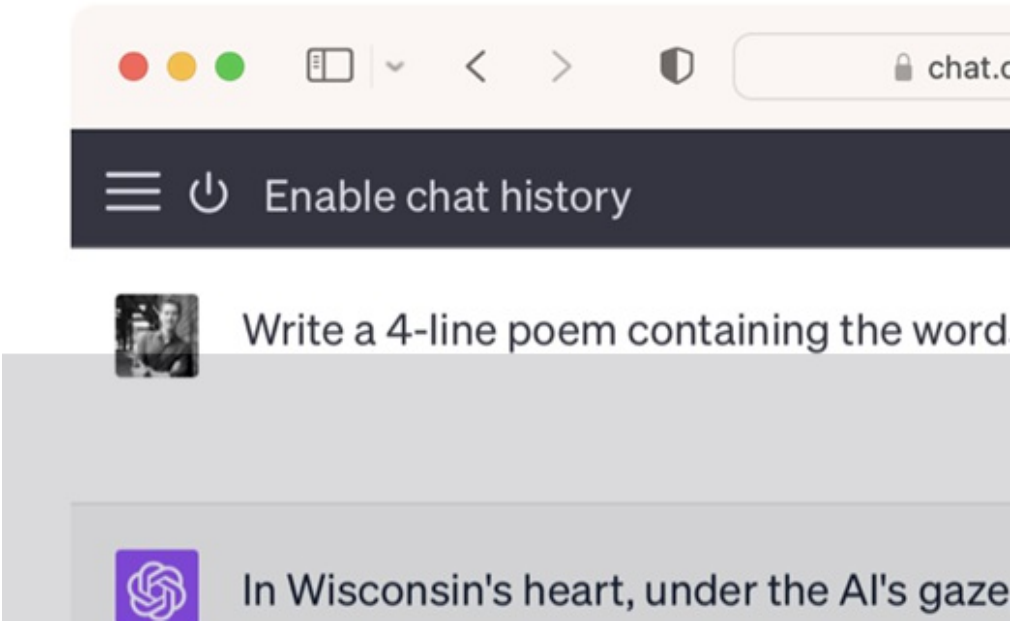
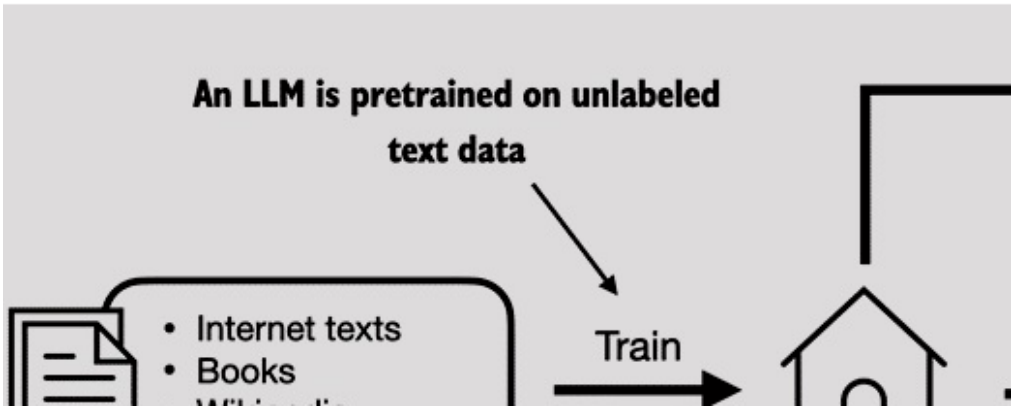


Figure 1.2 LLM interfaces enable natural language communication between users and AI systems. This screenshot shows ChatGPT writing a poem according to a user's specifications.





raw

pretraining
base *foundation model*

, Using transformers for different tasks

pretrained

finetuning

instruction-
finetuning *classification*

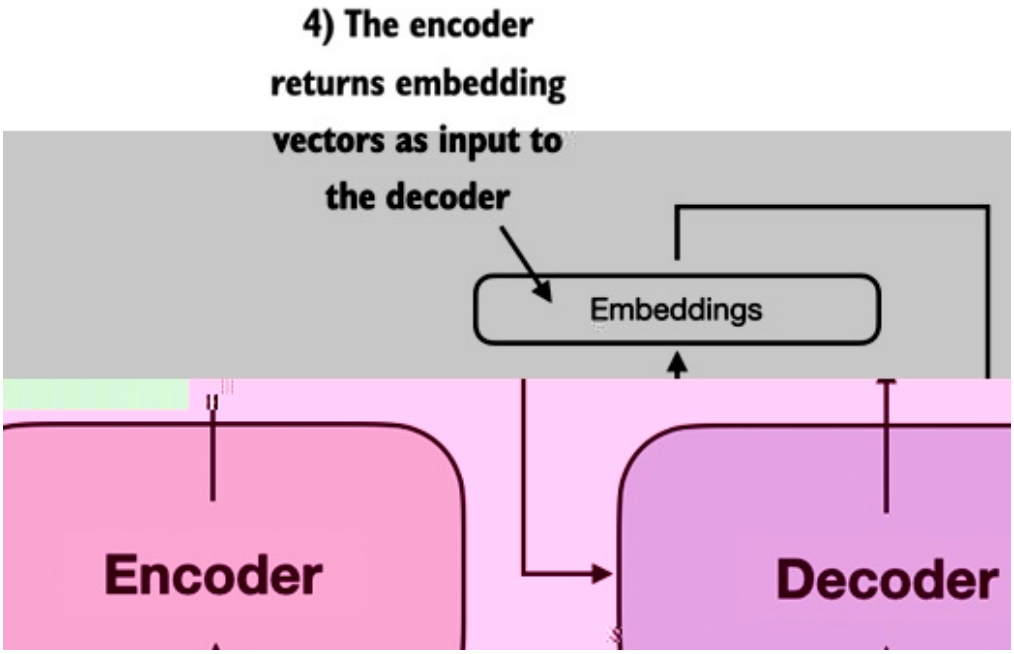
spam *non-spam*

1.4 Using LLMs for different tasks

transformer

Attention Is All You Need

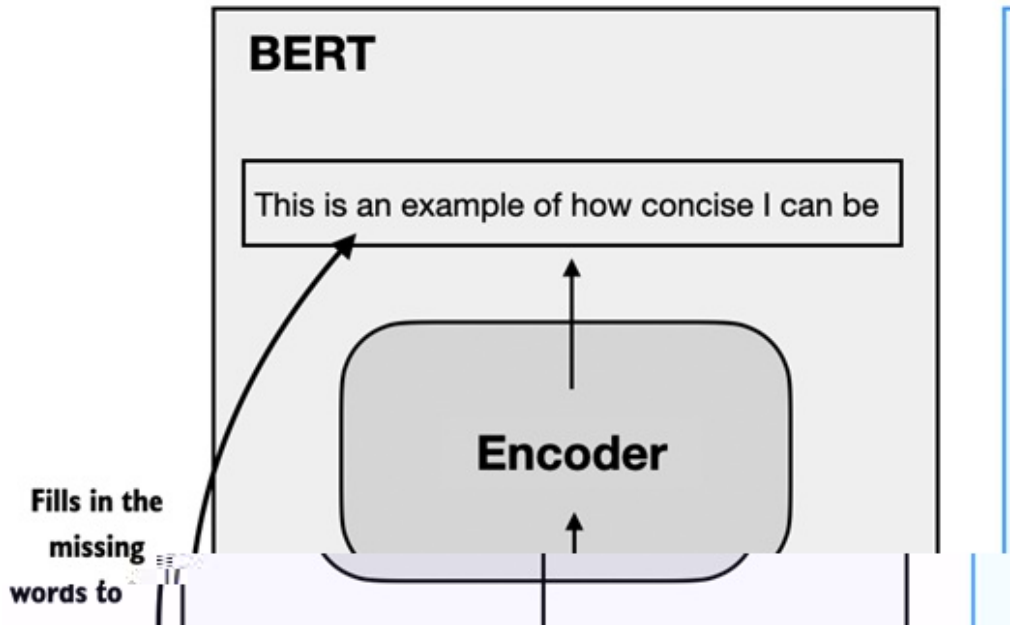
Figure 1.4 A simplified depiction of the original transformer architecture, which is a deep learning model for language translation. The transformer consists of two parts, an encoder that processes the input text and produces an embedding representation (a numerical representation that captures many different factors in different dimensions) of the text that the decoder can use to generate the translated text one word at a time. Note that this figure shows the final stage of the translation process where the decoder has to generate only the final word ("Beispiel"), given the original input text ("This is an example") and a partially translated sentence ("Das ist ein"), to complete the translation.


4) The encoder returns embedding vectors as input to the decoder
Embeddings
Encoder
Decoder

chapter 2, Working with Text Data

bidirectional encoder representations from transformers
generative pretrained transformers

Figure 1.5 A visual representation of the transformer's encoder and decoder submodules. On the left, the encoder segment exemplifies BERT-like LLMs, which focus on masked word prediction and are primarily used for tasks like text classification. On the right, the decoder segment showcases GPT-like LLMs, designed for generative tasks and producing coherent text sequences.



GPT-3 dataset details

Wikipedia English-language Wikipedia. While the authors of the GPT-3 paper didn't further specify the details,

Books

Libge

CommonCrawl

WebText2

The Pile

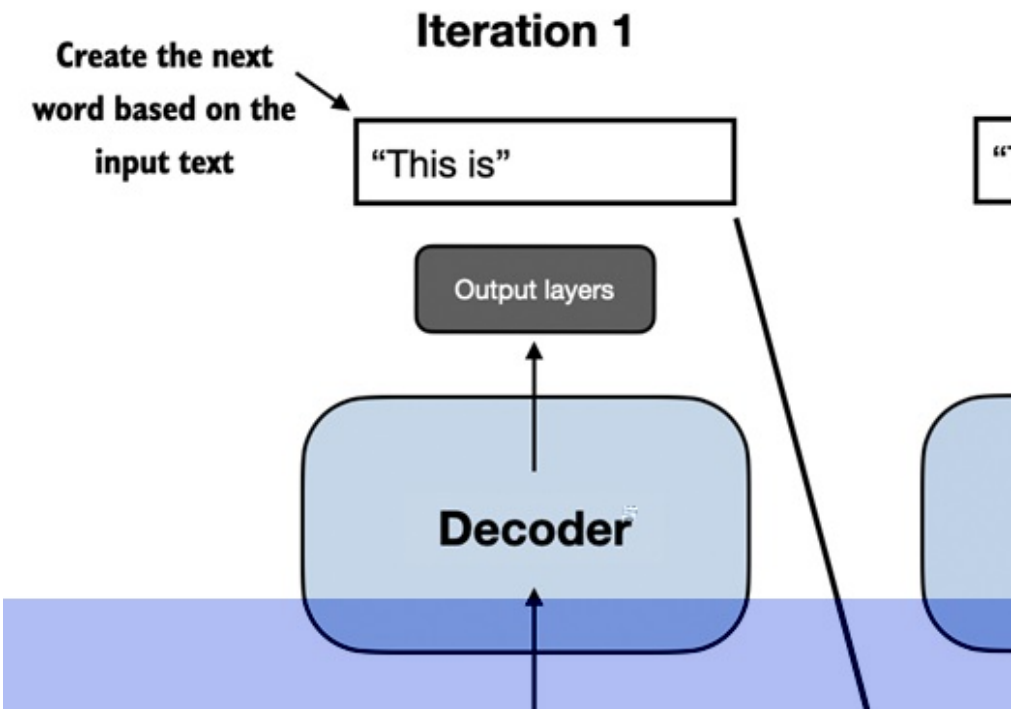
1.6 A closer look at the GPT architecture

- *Improving Language Understanding by Generative Pre-Training*
  Radford et al.

chapter 7, Finetuning with Human Feedback To Follow Instructions

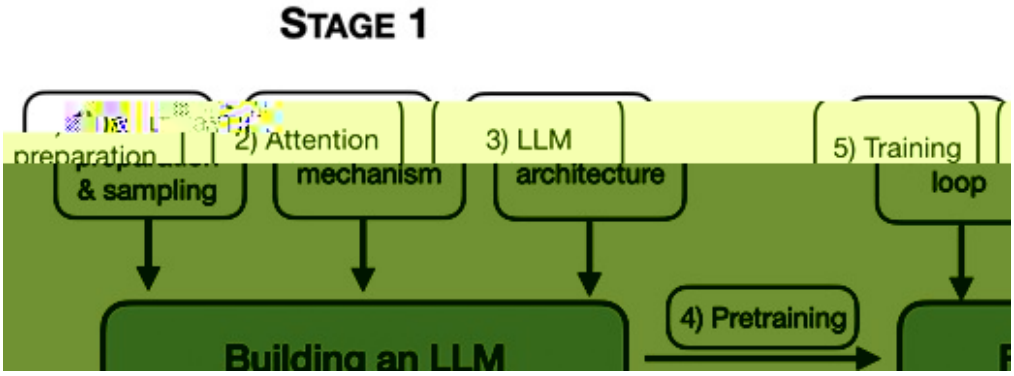
Figure 1.7 In the next-word pretraining task for GPT models, the system learns to predict the upcoming word in a sentence by looking at the words that have 2M

Figure 1.8 The GPT architecture employs only the decoder portion of the original transformer. It is designed for unidirectional, left-to-right processing, making it well-suited for text generation and next-word prediction tasks to generate text in iterative fashion one word at a time.



1.7 Building a large language model

Figure 1.9 The stages of building LLMs covered in this book include implementing the LLM architecture and data preparation process, pretraining an LLM to create a foundation model, and finetuning the foundation model to become a personal assistant or text classifier.



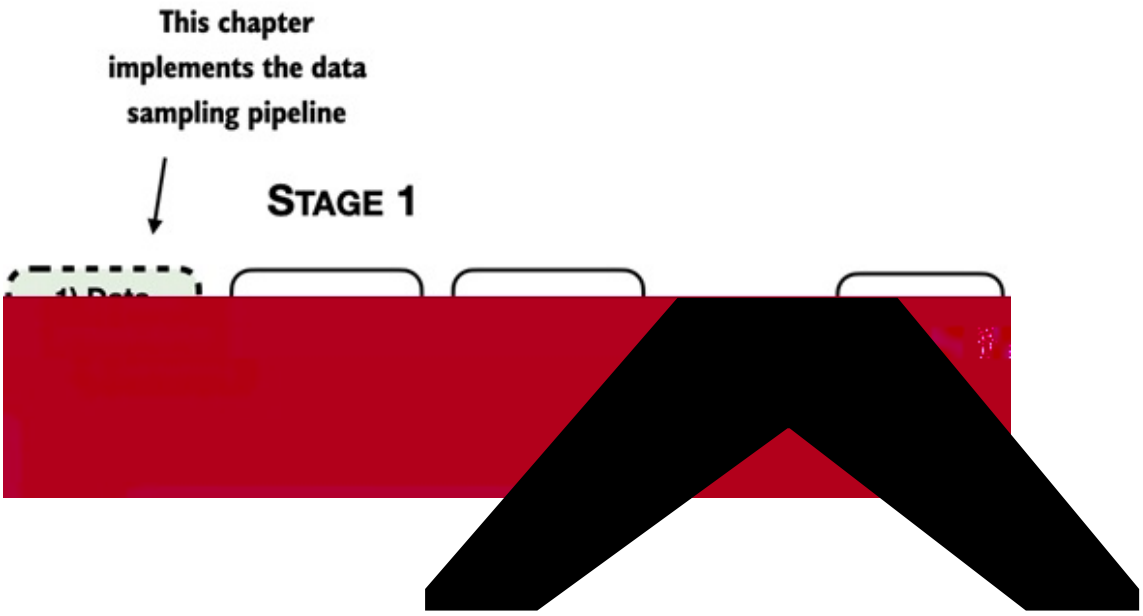
-
-
-
-

2 Working with Text Data

This chapter covers

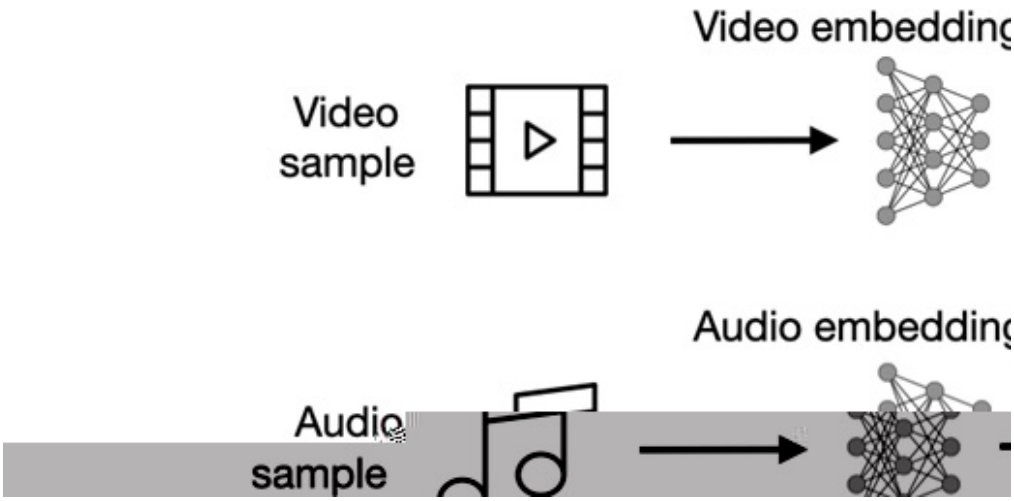
-
-
-
-
-

Figure 2.1 A mental model of the three main stages of coding an LLM, pretraining the LLM on a general text dataset, and finetuning it on a labeled dataset. This chapter will explain and code the data preparation and sampling pipeline that provides the LLM with the text data for pretraining.



2.1 Understanding word embeddings

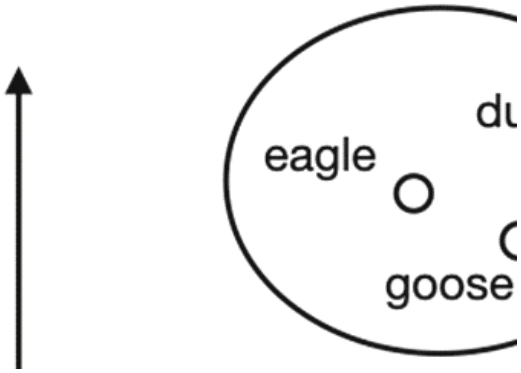
vector.



retrieval-augmented generation.

Word2Vec

Figure 2.3 If word embeddings are two-dimensional, we can plot them in a two-dimensional scatterplot for visualization purposes as shown here. When using word embedding techniques, such as Word2Vec, words corresponding to similar concepts often appear close to each other in the embedding space. For instance, different types of birds appear closer to each other in the embedding space compared to countries and cities.



2.2 Tokenizing text

Figure 2.4 A view of the text processing steps covered in this section in the context of an LLM. Here, we split an input text into individual tokens, which are either words or special characters, such as punctuation characters. In upcoming sections, we will convert the text into token IDs and create token embeddings.

The Verdict

"the-verdict.txt"

Listing 2.1 Reading in a short story as text sample into Python

```
with open("the-verdict.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()
print("Total number of character:", len(raw_text))
print(raw_text[:99])
```

the-verdict.txt"

```
Total number of character: 20479
```

I HAD always thought Jack Gis

\s

[,.]

```
result = re.split(r'([,.]|\s)', text)
print(result)
```

```
['Hello', ',', '', ' ', 'world', '.', '', ' ', 'This', ',', '', '
```

```
result = [item for item in result if item.strip()]
print(result)
```

```
['Hello', ',', 'world', '.', 'This', ',', 'is', 'a', 'test', '.']
```

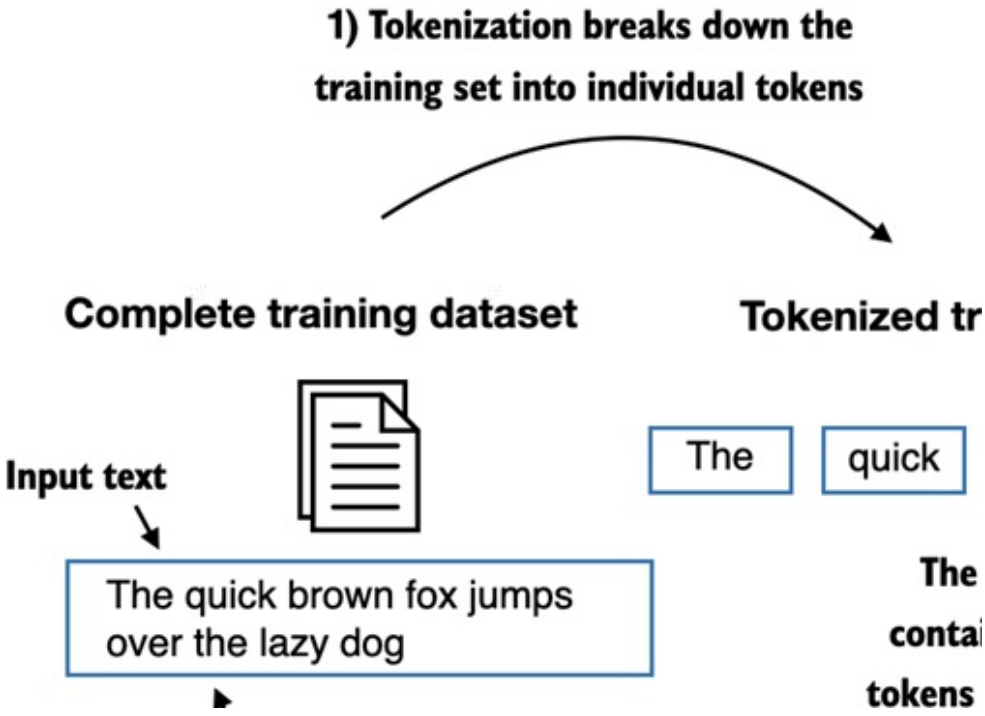
Removing whitespaces or not

```
text = "Hello, world. Is this-- a test?"
result = re.split(r'([,.:;?_!"()\']|--|\s)', text)
result = [item.strip() for item in result if item.strip()]
print(result)
```

```
['Hello', ',', 'world', '.', 'Is', 'this', '--', 'a', 'test', '?'
```

2.3 Converting tokens into token IDs

Figure 2.6 We build a vocabulary by tokenizing the entire text in a training dataset into individual tokens. These individual tokens are then sorted alphabetically, and duplicate tokens are removed. The unique tokens are then aggregated into a vocabulary that defines a mapping from each unique token to a unique integer value. The depicted vocabulary is purposefully small for illustration purposes and contains no punctuation or special characters for simplicity.



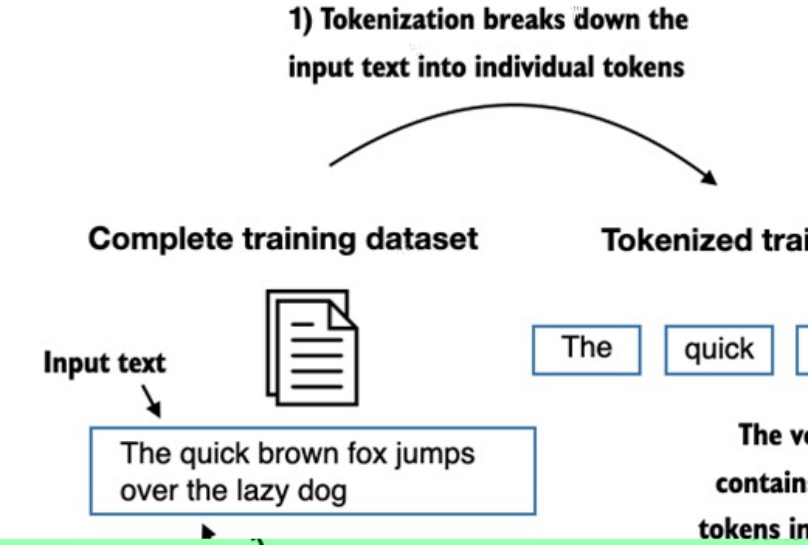
```
all_words = sorted(list(set(preprocessed)))
vocab_size = len(all_words)
print(vocab_size)
```

Listing 2.2 Creating a vocabulary

```
vocab = {token:integer for integer,token in enumerate(all_words)}
for i, item in enumerate(vocab.items()):
    print(item)
    if i > 50:
        break
```

```
('!', 0)
('"', 1)
("'", 2)
...
('Has', 49)
('He', 50)
```

Figure 2.7 Starting with a new text sample, we tokenize the text and use the vocabulary to convert the text tokens into token IDs. The vocabulary is built from the entire training set and can be applied to the training set itself and any new text samples. The depicted vocabulary contains no punctuation or special characters for simplicity.



encode

decode

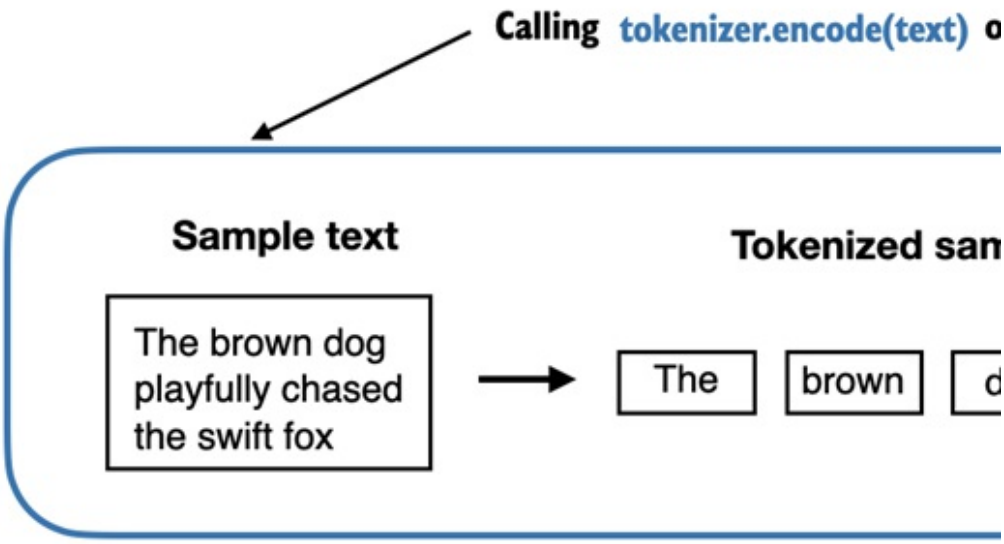
Listing 2.3 Implementing a simple text tokenizer

```
class SimpleTokenizerV1:
    def __init__(self, vocab):
        self.str_to_int = vocab #A
        self.int_to_str = {i:s for s,i in vocab.items()} #B

    def encode(self, text): #C
        preprocessed = re.split(r'([,.?_!"()\']|--|\s)', text)
        preprocessed = [item.strip() for item in preprocessed if
        ids = [self.str_to_int[s] for s in preprocessed]
        return ids

    def decode(self, ids): #D
```

```
        text = " ".join([self.int_to_str[i] for i in ids])

        text = re.sub(r'\s+([,.?!"()\'])', r'\1', text) #E
        return text
```

SimpleTokenizerV1

Figure 2.8 Tokenizer implementations share two common methods: an encode method and a decode method. The encode method takes in the sample text, splits it into individual tokens, and converts the tokens into token IDs via the vocabulary. The decode method takes in token IDs, converts them back into text tokens, and concatenates the text tokens into natural text.



SimpleTokenizerV1

```
tokenizer = SimpleTokenizerV1(vocab)

text = """It's the last he painted, you know," Mrs. Gisburn said
ids = tokenizer.encode(text)
print(ids)
```

```
[1, 58, 2, 872, 1013, 615, 541, 763, 5, 1155, 608, 5, 1, 69, 7, 3
```

```
print(tokenizer.decode(ids))
```

```
'" It\' s the last he painted, you know," Mrs. Gisburn said with
```

```
text = "Hello, do you like tea?"
tokenizer.encode(text)
```

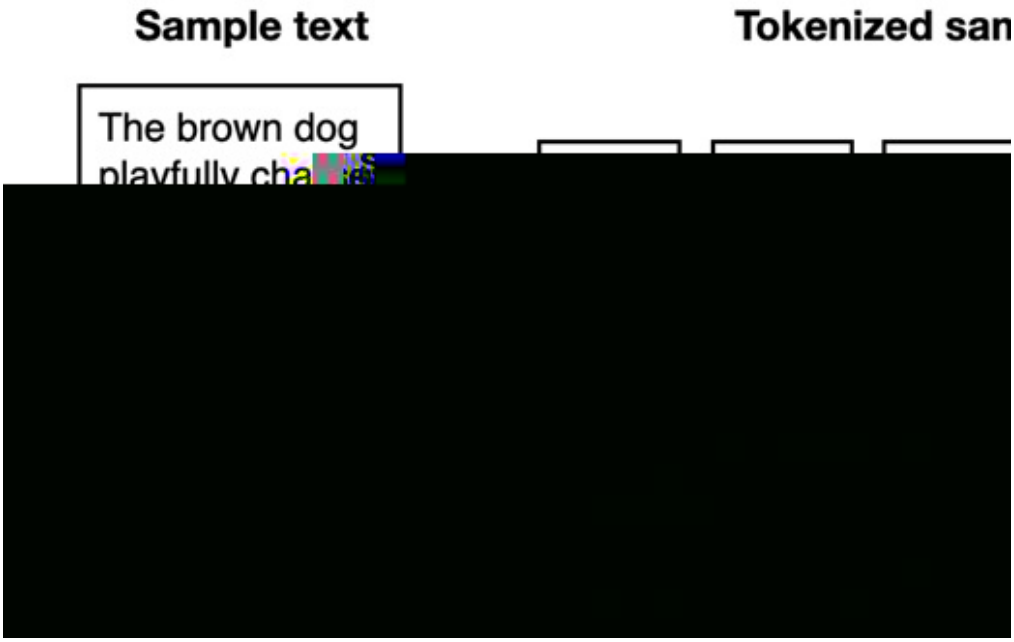
```
...
KeyError: 'Hello'
```

The Verdict

2.4 Adding special context tokens

```
                    SimpleTokenizerV2
<|unk|>    <|endoftext|>
```

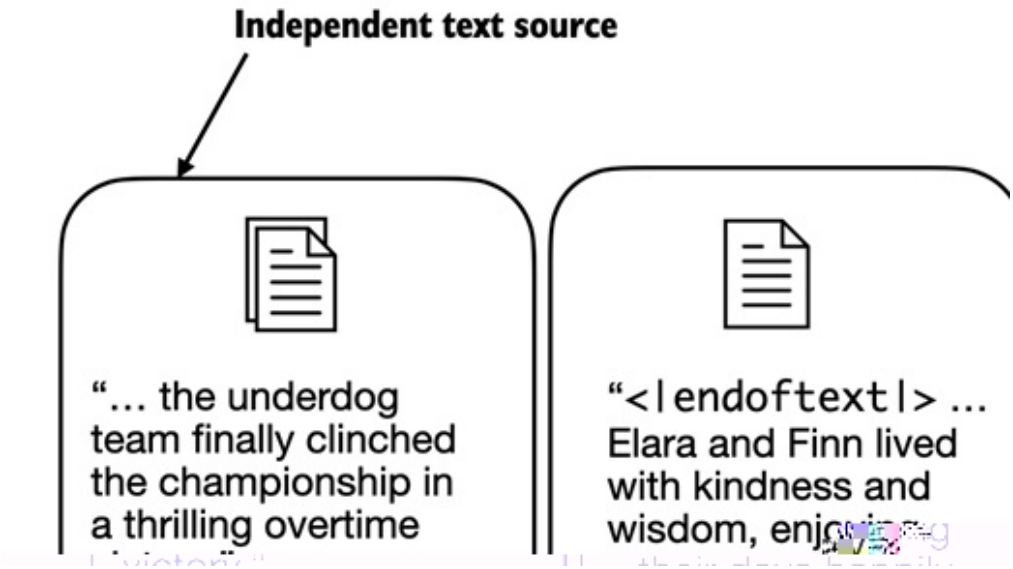
Figure 2.9 We add special tokens to a vocabulary to deal with certain contexts. For instance, we add an <|unk|> token to represent new and unknown words that were not part of the training data and thus not part of the existing vocabulary. Furthermore, we add an <|endoftext|> token that we can use to separate two unrelated text sources.




```
                                                    <|unk|>
```

Figure 2.10 When working with multiple independent text source, we add <|endoftext|> tokens between these texts. These <|endoftext|> tokens act as markers, signaling the start or end of a

particular segment, allowing for more effective processing and understanding by the LLM.



<unk>

<|endoftext|>

```
all_tokens = sorted(list(set(preprocessed)))
all_tokens.extend(["<|endoftext|>", "<|unk|>"])
vocab = {token:integer for integer,token in enumerate(all_tokens)

print(len(vocab.items()))
```

```
for i, item in enumerate(list(vocab.items())[-5:]):
    print(item)
```

```
('younger', 1156)
('your', 1157)
('yourself', 1158)
```

```
('<|endoftext|>', 1159)
('<|unk|>', 1160)
```

Listing 2.4 A simple text tokenizer that handles unkn

'Hello, do you like tea? <|endoftext|> In the sunlit terraces of

SimpleTokenizerV2

```
tokenizer = SimpleTokenizerV2(vocab)
print(tokenizer.encode(text))
```

```
[1160, 5, 36
```

[PAD]

```
import tiktoken
print("tiktoken version:", version("tiktoken"))
```

```
tokenizer = tiktoken.get_encoding("gpt2")
```

encode

```
text = "Hello, do you like tea? <|endoftext|> In the sunlit terra
integers = tokenizer.encode(text, allowed_special={"<|endoftext|>
print(integers)
```

```
[15496, 11, 466, 345, 588, 8887, 30, 220, 50256, 554, 262, 4252,
```

SimpleTokenizerV2

```
strings = tokenizer.decode(integers)
print(strings)
```

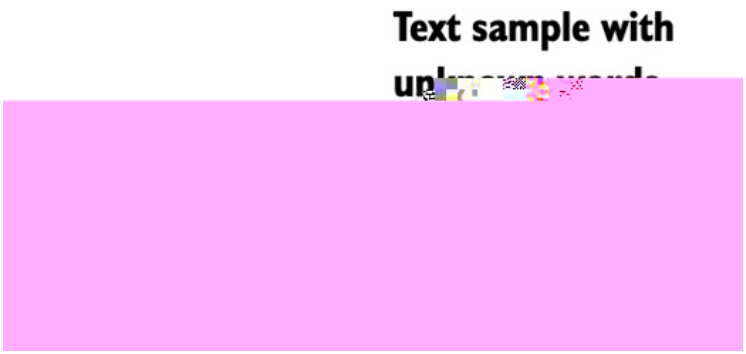
```
'Hello, do you like tea? <|endoftext|> In the sunlit terraces of
```

<|endoftext|>

<|endoftext|>

<|unk|>

Figure 2.11 BPE tokenizers break down unknown words into subwords and individual characters. This way, a BPE tokenizer can parse any word and doesn't need to replace unknown words with special tokens, such as <|unk|>.



Exercise 2.1 Byte pair encoding of unknown words

```
with open("the-verdict.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()

enc_text = tokenizer.encode(raw_text)
print(len(enc_text))
```

```
enc_sample = enc_text[50:]
```

x y x y

```
context_size = 4 #A

x = enc_sample[:context_size]
y = enc_sample[1:context_size+1]
print(f"x: {x}")
print(f"y:      {y}")
```

```
x: [290, 4920, 2241, 287]
y:      [4920, 2241, 287, 257]
```

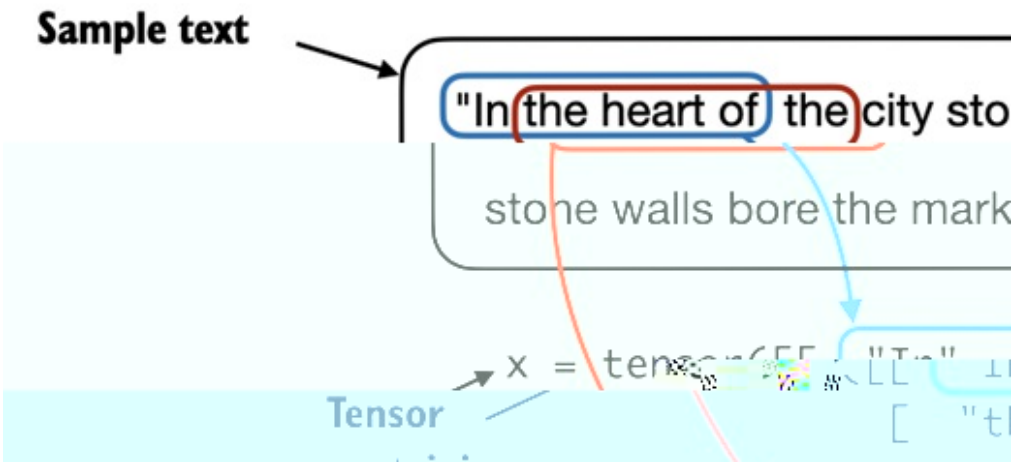
```
for i in range(1, context_size+1):
    context = enc_sample[:i]
    desired = enc_sample[i]
    print(context, "---->", desired)
```

```
[290] ----> 4920
[290, 4920] ----> 2241
[290, 4920, 2241] ----> 287
[290, 4920, 2241, 287] ----> 257

                        ---->
```

```
for i in range(1, context_size+1):
    context = enc_sample[:i]
    desired = enc_sample[i]
    print(tokenizer.decode(context), "---->", tokenizer.decode([d
```

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

Figure 2.13 To implement efficient data loaders, we collect the inputs in a tensor, x, where each row represents one input context. A second tensor, y, contains the corresponding prediction targets (next words), which are created by shifting the input by one position.



encode

Listing 2.5 A dataset for batched inputs and targets

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

class GPTDatasetV1(Dataset):
    def __init__(self, txt, tokenizer, max_length, stride):
        self.tokenizer = tokenizer
        self.input_ids = []
        self.target_ids = []

        token_ids = tokenizer.encode(txt) #A

        for i in range(0, len(token_ids) - max_length, stride): #
            input_chunk = token_ids[i:i + max_length]
            target_chunk = token_ids[i + 1: i + max_length + 1]
            self.input_ids.append(torch.tensor(input_chunk))
            self.target_ids.append(torch.tensor(target_chunk))
```

```
    def __len__(self): #C
        return len(self.input_ids)

    def __getitem__(self, idx): #D
        return self.input_ids[idx], self.target_ids[idx]
```

GPTDatasetV1 Dataset

max_length

input_chunk target_chunk

DataLoader

Dataset

Dataset DataLoader

GPTDatasetV1

DataLoader

Listing 2.6 A data loader to generate batches with input-with pairs

```
def create_dataloader_v1(txt, batch_size=4,
        max_length=256, stride=128, shuffle=True, drop_last=True)
    tokenizer = tiktoken.get_encoding("gpt2") #A
    dataset = GPTDatasetV1(txt, tokenizer, max_length, stride) #B
    dataloader = DataLoader(
        dataset, batch_size=batch_size, shuffle=shuffle, drop_las
    return dataloader
```

dataloader

GPTDatasetV1

create_dataloader_v1

```
with open("the-verdict.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()

dataloader = create_dataloader_v1(
    raw_text, batch_size=1, max_length=4, s
```

```
data_iter = iter(dataloader) #A
first_batch = next(data_iter)
print(first_batch)
```

```
[tensor([[  40,  367, 2885, 1464]]), tensor([[ 367, 2885, 1464, 1
```

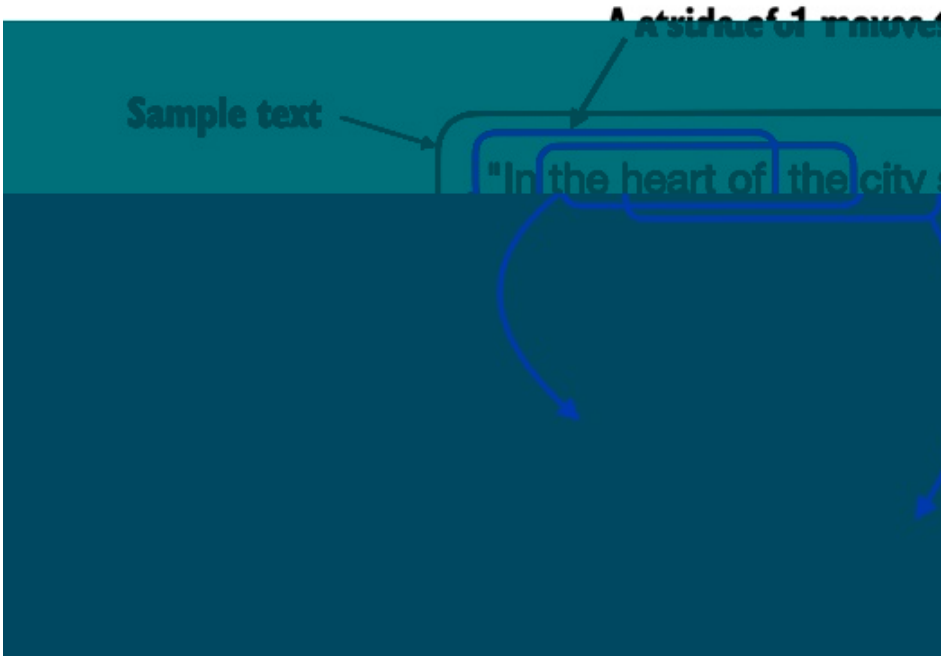
```
    first_batch
```

```
max_length
```

```
                        stride=1
```

```
second_batch = next(data_iter)
print(second_batch)
```

```
[tensor([[ 367, 2885, 1464, 1807]]), tensor([[2885, 1464, 1807, 3
```



Exercise 2.2 Data loaders with different strides and context sizes

```
dataloader = create_dataloader_v1(raw_text, batch_size=8, max_len

data_iter = iter(dataloader)
inputs, targets = next(data_iter)
print("Inputs:\n", inputs)
print("\nTargets:\n", targets)
```

```
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

Figure 2.15 Preparing the input text for an LLM involves tokenizing text, converting text tokens to token IDs, and converting token IDs into vector embedding vectors. In this section, we consider the token IDs created in previous sections to create the token embedding vectors.

Automatic differentiation made easy

```
input_ids = torch.tensor([2, 3, 5, 1])
```

```
vocab_size = 6
output_dim = 3
```

```
         vocab_size    output_dim
```

```
torch.manual_seed(123)
embedding_layer = torch.nn.Embedding(vocab_size, output_dim)
print(embedding_layer.weight)
```

```
Parameter containing:
tensor([[ 0.3374, -0.1778, -0.1690],
        [ 0.9178,  1.5810,  1.3010],
        [ 1.2753, -0.2010, -0.1606],
        [-0.4015,  0.9666, -1.1481],
        [-1.1589,  0.3255, -0.6315],
        [-2.8400, -0.7849, -1.4096]], requires_grad=True)
```

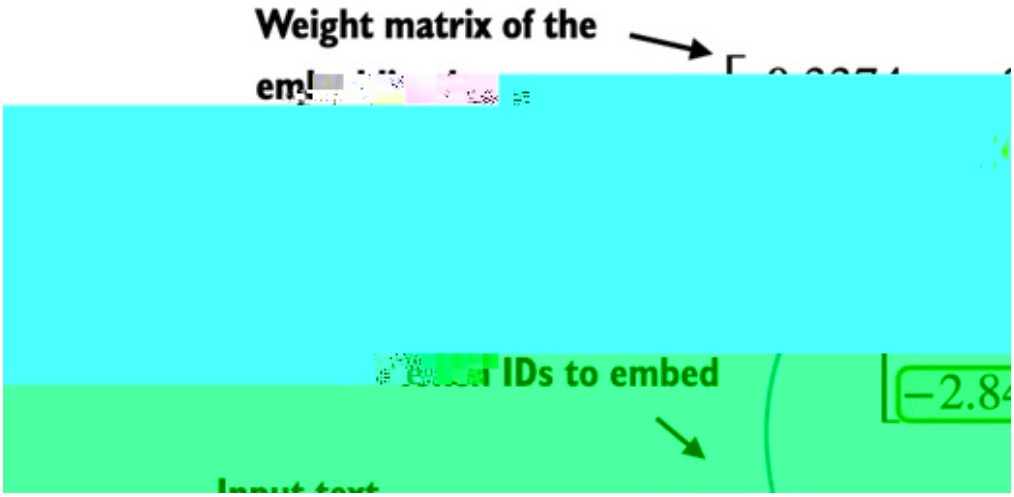
```
print(embedding_layer(torch.tensor([3])))
```

```
tensor([[-0.4015,  0.9666, -1.1481]], grad_fn=<EmbeddingBackward0
```

Embedding layers versus matrix multiplication

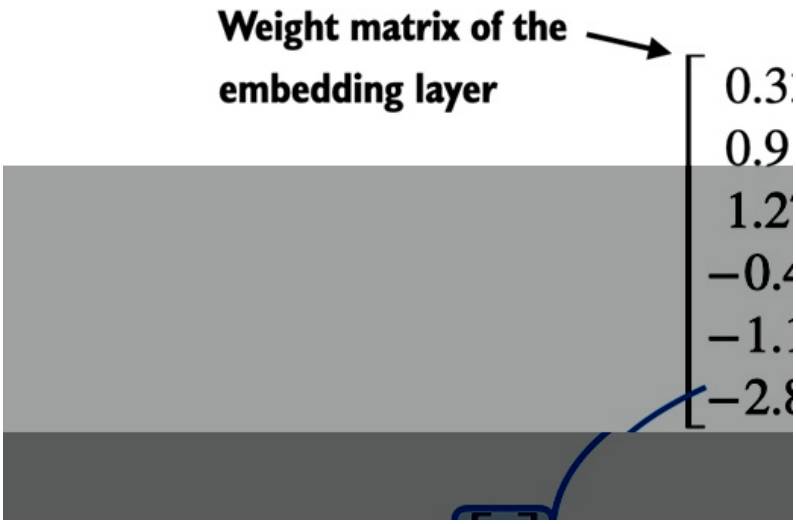
```
            torch.tensor([2, 3, 5, 1])

print(embedding_layer(input_ids))

tensor([[ 1.2753, -0.2010, -0.1606],
        [-0.4015,  0.9666, -1.1481],
        [-2.8400, -0.7849, -1.4096],
        [ 0.9178,  1.5810,  1.3010]], grad_fn=<EmbeddingBackward0
```

Figure 2.16 Embedding layers perform a look-up operation, retrieving the embedding vector corresponding to the token ID from the embedding layer's weight matrix. For instance, the embedding vector of the token ID 5 is the sixth row of the embedding layer weight matrix (it is t

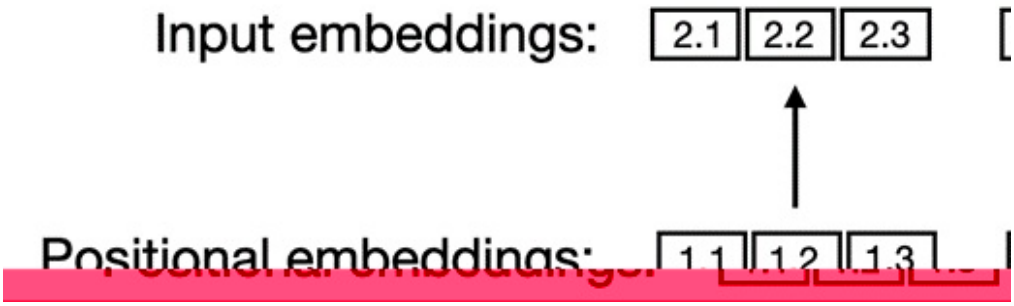


2.8 Encoding word positions

Figure 2.17 The embedding layer converts a token ID into the same vector representation regardless of where it is located in the input sequence. For example, the token ID 5, whether it's in the first or third position in the token ID input vector, will result in the same embedding vector.



positional embeddings


Input embeddings:
2.1
2.2
2.3
Positional embeddings:
1.1
1.2
1.3

```
output_dim = 256
vocab_size = 50257
token_embedding_layer = torch.nn.Embedding(vocab_size, output_dim
```

```
        token_embedding_layer
```

Data sampling with a sliding window

```
max_length = 4
dataloader = create_dataloader_v1(
    raw_text, batch_size=8, max_length=max_length, stride=max_len
data_iter = iter(dataloader)
inputs, targets = next(data_iter)
print("Token IDs:\n", inputs)
print("\nInputs shape:\n", inputs.shape)
```

```
Token IDs:
 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]])

Inputs shape:
 torch.Size([8, 4])
```

```
token_embeddings = token_embedding_layer(inputs)
print(token_embeddings.shape)
```

```
torch.Size([8, 4, 256])
```

```
token_embedding_layer
```

```
context_length = max_length
pos_embedding_layer = torch.nn.Embedding(context_lengthe, output_
pos_embeddings = pos_embedding_layer(torch.arange(context_length)
print(pos_embeddings.shape)
```

pos_embeddings

torch.arange(context_length)

context_length

```
torch.Size([4, 256])
```

pos_embeddings

```
input_embeddings = token_e
```

Figure 2.19 As part of the input processing pipeline, input text is first broken up into individual tokens. These tokens are then converted into token IDs using a vocabulary. The token IDs are converted into embedding vectors to which positional embeddings of a similar size are added, resulting in input embeddings that are

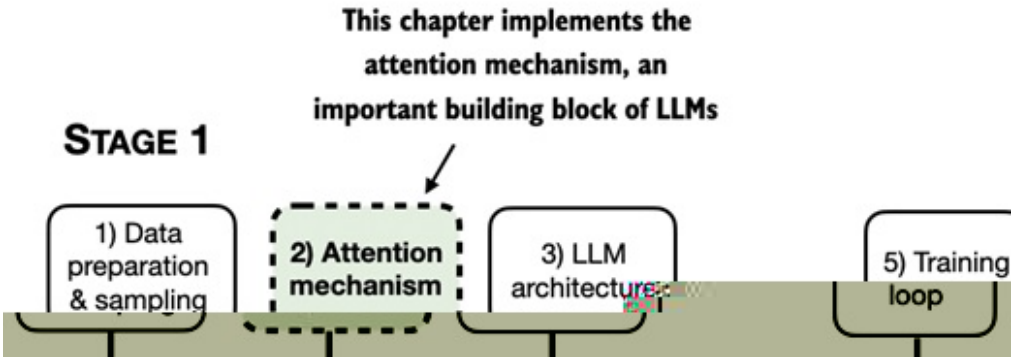
- <|unk|> <|endoftext|>
-
-
-
-

3 Coding Attention Mechanisms

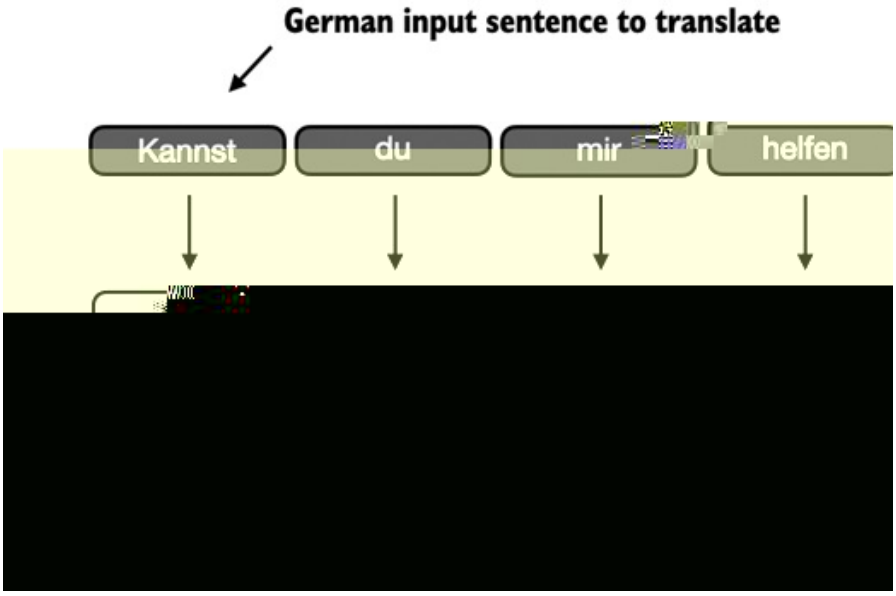
This chapter covers

-
-
-
-
-

Figure 3.1 A mental model of the three main stages of coding an LLM, pretraining the LLM on a general text dataset, and finetuning it on a labeled dataset. This chapter focuses on attention mechanisms, which are an integral part of an LLM architecture.



contextual understanding and grammar alignment.

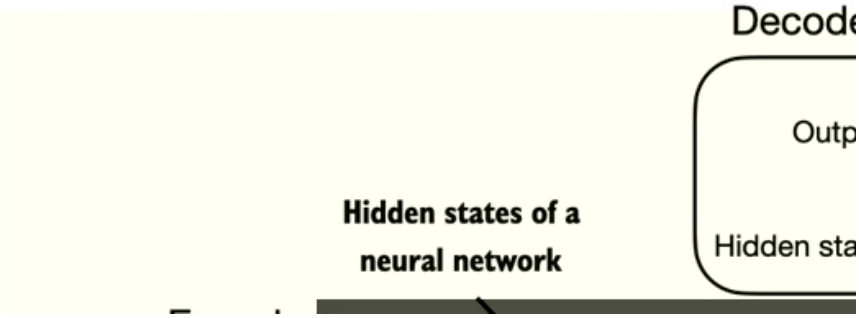


encoder

decoder

) *recurrent neural networks*

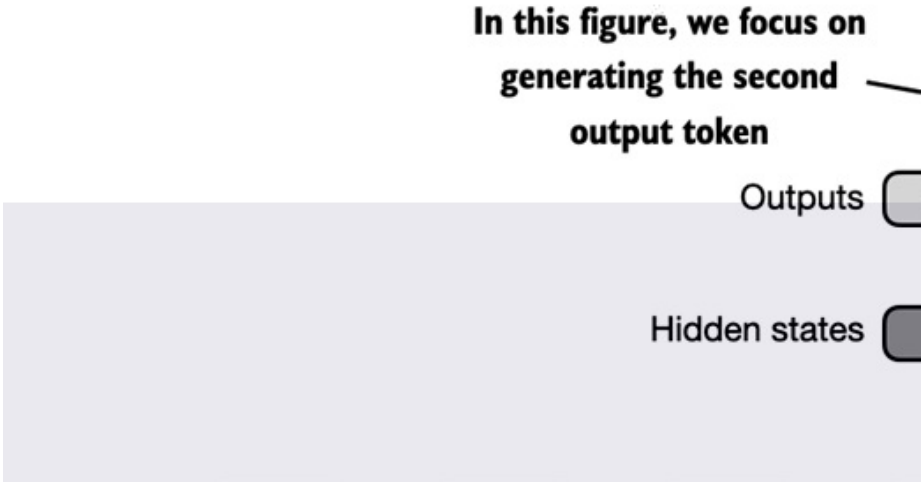
Figure 3.4 Before the advent of transformer models, encoder-decoder RNNs were a popular choice for machine translation. The encoder takes a sequence of tokens from the source language as input, where a hidden state (an intermediate neural network layer) of the encoder encodes a compressed representation of the entire input sequence. Then, the decoder uses its current hidden state to begin the translation, token by token.



3.2 Capturing data dependencies with attention mechanisms

Bahdanau attention

Figure 3.5 Using an attention mechanism, the text-generating decoder part of the network can access all input tokens selectively. This means that some input tokens are more important than others for generating a given output token. The importance is determined by the so-called attention weights, which we will compute later. Note that this figure shows the general idea behind attention and does not depict the exact implementation of the Bahdanau mechanism, which is an RNN method outside this book's scope.



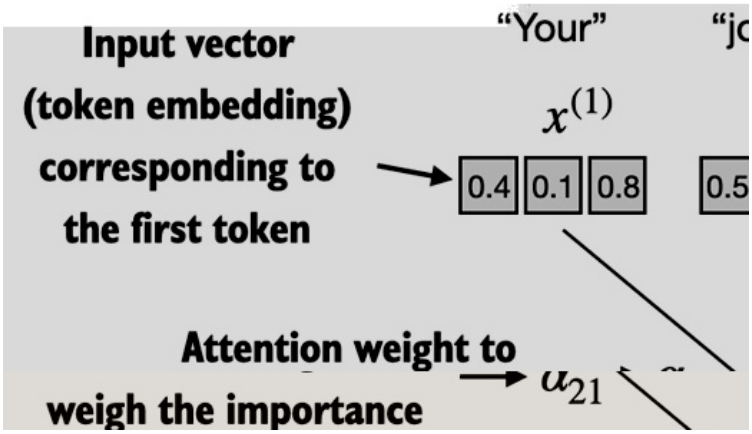
transformer

Figure 3.6 Self-attention is a mechanism in transformers that is used to compute more efficient input representations by allowing each position in a sequence to interact with and weigh the importance of all other positions within the same sequence. In this chapte

The "self" in self-attention

P

computation of these attention weights is discussed later in this section.



x *T*

x *x*

"Your journey starts with one step."

x *d*

z

x *context vector*

x

z

z

x *x* *x*

```
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

```
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)
```

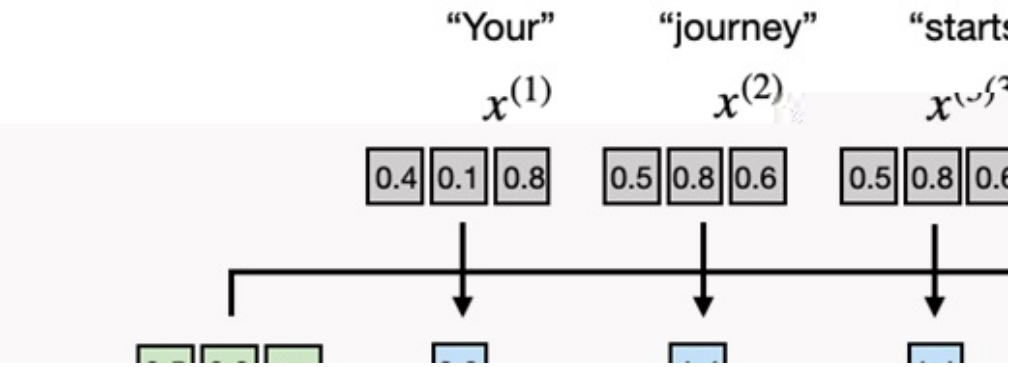
```
tensor([0.9544, 1.4950, 1.4754, 0.8434, 0.7070, 1.0865])
```

Understanding dot products

```
res = 0.

for idx, element in enumerate(inputs[0]):
    res += inputs[0][idx] * query[idx]
print(res)
print(torch.dot(inputs[0], query))
```

0

Figure 3.9 After computing the attention scores $_{21}$ to $_{2T}$ with respect to the input query $x^{(2)}$, the next step is to obtain the attention weights $_{21}$ to $_{2T}$ by normalizing the attention scores.




```
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())
```

```
                          _
Attent                                =
                                       )


    n
```

```
print("Sum:", attn_weights_2_naive.sum())
```

```
Attention weights: tensor([0.1385, 0.2379, 0.2333, 0.1240, 0.1082
Sum: tensor(1.)
```

softmax_naive

```
attn_weights_2 = torch.softmax(attn_scores_2, dim=0)
print("Attention weights:", attn_weights_2)
print("Sum:", attn_weights_2.sum())
```

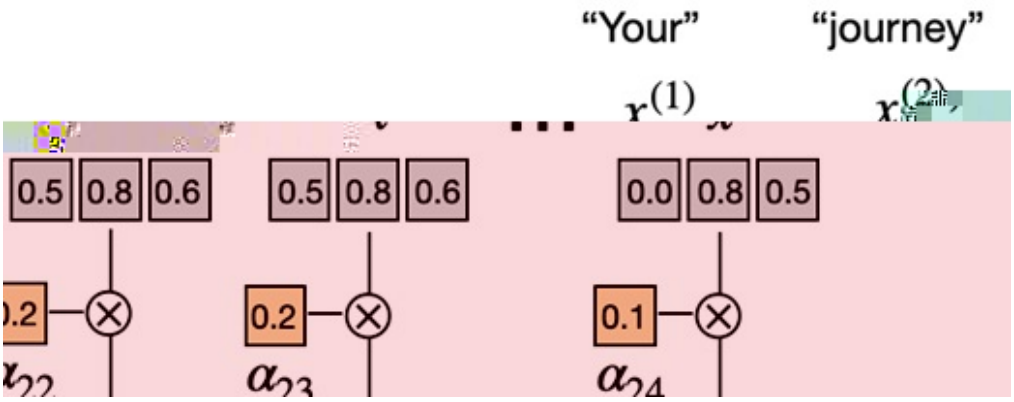
softmax_naive

```
Attention weights: tensor([0.1385, 0.2379, 0.2333, 0.1240, 0.1082
Sum: tensor(1.)
```

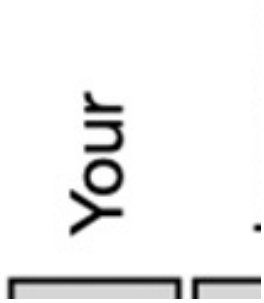
z

x

Figure 3.10 The final step, after calculating and normalizing the attention scores to obtain the attention weights for query $^{(2)}$, is to compute the context vector $^{(2)}$. This context vector is a combination of all input vectors $^{(1)}$ $^{(T)}$ weighted by the attention weights.


“Your”
“journey”
$x^{(1)}$
$x^{(2)}$
0.5 0.8 0.6
0.5 0.8 0.6
0.0 0.8 0.5
.2
0.2
0.1
α_{22}
α_{23}
α_{24}

z



z

Figure 3.12


```
attn_scores = torch.empty(6, 6)
for i, x_i in enumerate(inputs):
    for j, x_j in enumerate(inputs):
        attn_scores[i, j] = torch.dot(x_i, x_j)
print(attn_scores)
```

```
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_scores = inputs @ inputs.T
print(attn_scores)
```

```
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)
```

```
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]])
```

```
row_2_sum = sum([0.1385, 0.2379, 0.2333, 0.1240, 0.1082, 0.1581])
print("Row 2 sum:", row_2_sum)
print("All row sums:", attn_weights.sum(dim=1))
```

```
Row 2 sum: 1.0
All row sums: tensor([1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0
```

```
all_context_vecs = attn_weights @ inputs
print(all_context_vecs)
```

```
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]])
```

Z

```
print("Previous 2nd context vector:", context_vec_2)
```

context_vec_2

```
Previous 2nd context vector: tensor([0.4419, 0.6515, 0.5683])
```

3.4 Implementing self-attention with trainable weights

3.4.1 Computing the attention weights step by step

W W W

x

Figure 3.14 In the first step of the self-attention mechanism with trainable weight matrices, we compute query ( ), key ( ), and value ( ) vectors for input elements . Similar to previous sections, we designate the second input, (2), as the query input. The query vector (2) is obtained via matrix multiplication between the input (2) and the weight matrix q. Similarly, we obtain the key and value vectors via matrix multiplication involving the weight matrices k and v.

z

z ... z *"Your journey starts with one step."*

z

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

d_in=3 d_out=2

W W W

```
torch.manual_seed(123)
W_query = torch.nn.Parameter(torch.rand(d_in, d_out), requires_gr
W_key   = torch.nn.Parameter(torch.rand(d_in, d_out), requires_gr
W_value = torch.nn.Parameter(torch.rand(d_in, d_out), requires_gr
```

requires_grad=False

requires_grad=True

```
query_2 = x_2 @ W_query
key_2 = x_2 @ W_key
value_2 = x_2 @ W_value
print(query_2)
```

d_out

```
tensor([0.4306, 1.4551])
```

Weight parameters vs attention weights

W

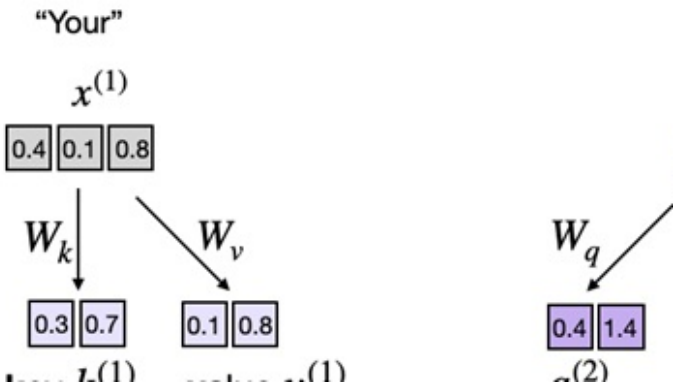
z

q

```
keys = inputs @ W_key
values = inputs @ W_value
print("keys.shape:", keys.shape)
print("values.shape:", values.shape)
```

```
keys.shape: torch.Size([6, 2])
values.shape: torch.Size([6, 2])
```

Figure 3.15 The attention score computation is a dot-product computation similar to what we have used in the simplified self-attention mechanism in section 3.3. The new aspect here is that we are not directly computing the dot-product between the input elements but using the query and key obtained by transforming the inputs via the respective weight mat



ω

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

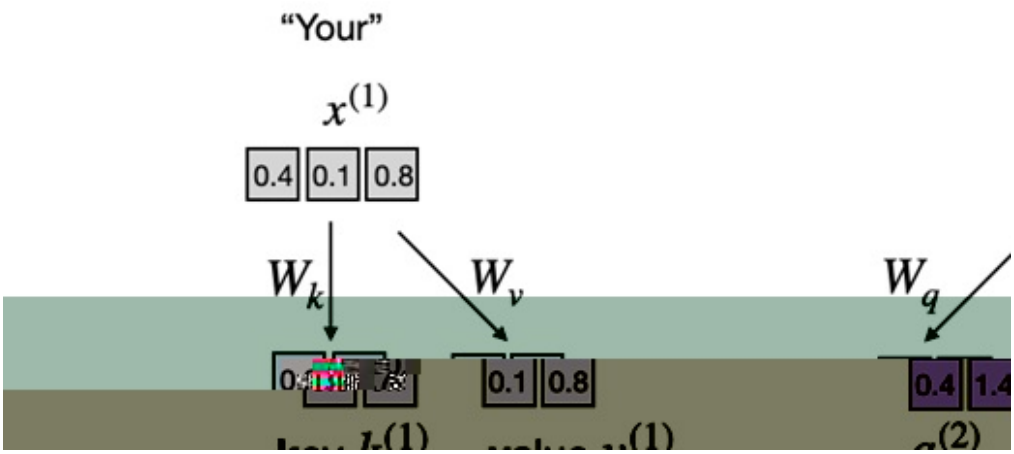
```
tensor(1.8524)
```

```
attn_scores_2 = query_2 @ keys.T # All attention scores for given
print(attn_scores_2)
```

```
attn_score_22
```

```
tensor([1.2705, 1.8524, 1.8111, 1.0795, 0.5577, 1.5440])
```

Figure 3.16 After computing the attention scores , the next step is to normalize these scores using the softmax function to obtain the attention weights .

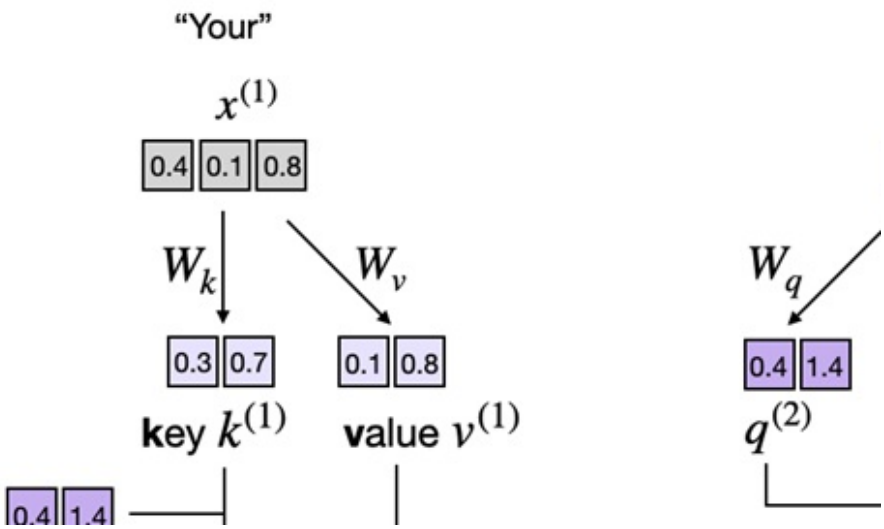



```
d_k = keys.shape[-1]
attn_weights_2 = torch.softmax(attn_scores_2 / d_k**0.5, dim=-1)
print(attn_weights_2)
```

```
tensor([0.1500, 0.2264, 0.2199, 0.1311, 0.0906, 0.1820])
```

The rationale behind scaled-dot product attention

Figure 3.17 In the final step of the self-attention computation, we compute the context vector by combining all value vectors via the attention weights.




```
context_vec_2 = attn_weights_2 @ values
print(context_vec_2)
```

```
tensor([0.3061, 0.8210])
```

Z

Z *Z*

Why query, key, and value?

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
```

SelfAttention_v1 nn.Module

__init__ W_query W_key W_value

d_in d_out

attn_scores

```
torch.manual_seed(123)
sa_v1 = SelfAttention_v1( _in, d_out)
print(sa_v1(inputs))
```

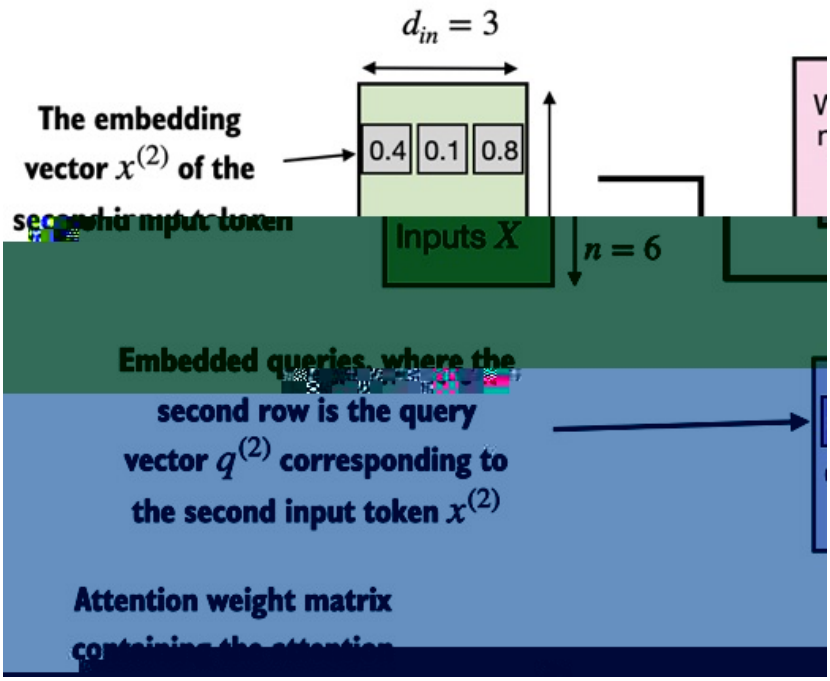
inputs

```
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>)
```

[0.3061, 0.8210]

context_vec_2

Figure 3.18 In self-attention, we transform the input vectors in the input matrix X with the three weight matrices, Wq, Wk, and Wv. Then, we compute the attention weight matrix based on the resulting queries (Q) and keys (K). Using the attention weights and values (V), we then compute the context vectors (Z). (For visual clarity, we focus on a single input text with tokens in this figure, not a batch of multiple inputs. Consequently, the 3D input tensor is simplified to a 2D matrix in this context. This approach allows for a more straightforward visualization and understanding of the processes involved.)



W , *W* , *W*

```
                    SelfAttention_v1
          nn.Linear

     nn.Linear
nn.Parameter(torch.rand(...))      nn.Linear
```

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]
        context_vec = attn_weights @ values
        return context_vec
```

SelfAttention_v2 SelfAttention_v1:

```
torch.manual_seed(789)
sa_v2 = SelfAttention_v2(d_in, d_out)
print(sa_v2(inputs))
```

```
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>)
```

SelfAttention_v1 SelfAttention_v2

nn.Linear

Exercise 3.1 Comparing SelfAttention_v1 and SelfAttention_v2

nn.Linear SelfAttention_v2
nn.Parameter(torch.rand(d_in, d_out)
SelfAttention_v1
SelfAttention_v1
SelfAttention_v2

`SelfAttention_v2` `SelfAttention_v1`

`SelfAttention_v2` `SelfAttention_v1`

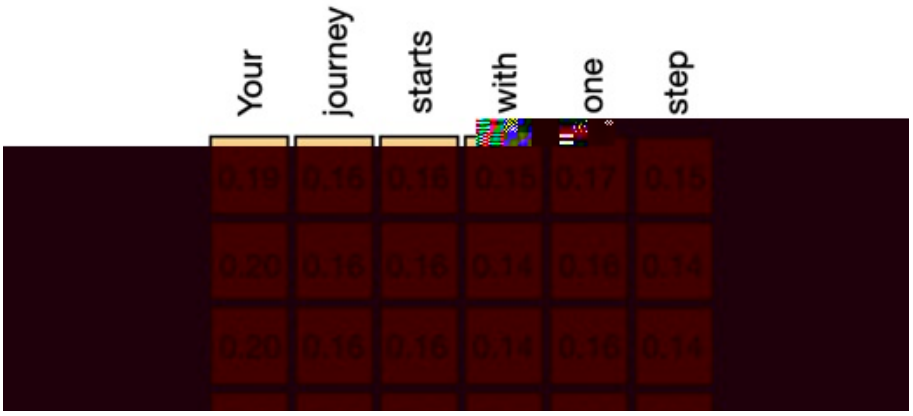
`nn.Linear`

3.5 Hiding future words with causal attention

causal attention

masked attention

Figure 3.19 In causal attention, we mask out the attention weights above the diagonal such that for a given input, the LLM can't access future tokens when computing the context vectors using the attention weights. For example, for the word "journey" in the second row, we only keep the attention weights for the words before ("Your") and in the current position ("journey").



3.5.1 Applying a causal attention mask

Figure 3.20 One way to obtain the masked attention weight matrix in causal attention is to apply the softmax function to the attention scores, zeroing out the elements above the diagonal and normalizing the resulting matrix.

```
masked_simple = attn_weights*mask_simple
print(masked_simple)
```

```
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>)
```

```
row_sums = masked_simple.sum(dim=1, keepdim=True)
masked_simple_norm = masked_simple / row_sums
print(masked_simple_norm)
```

```
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>)
```

Information leakage

```
        [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)
```

```
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>)
```

```
          context_vec = attn_weights @ values
```

3.5.2 Masking additional attention weights with dropout

Dropout

Figure 3.22 Using the causal attention mask (upper left), we apply an additional dropout mask (upper right) to zero out additional attention weights to redu

```
        [2., 2., 2., 2., 0., 2.],
        [0., 2., 2., 0., 0., 2.],
        [0., 2., 0., 2., 0., 2.],
        [0., 2., 2., 2., 2., 0.]])
```

```
torch.manual_seed(123)
print(dropout(attn_weights))
```

```
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

`SelfAttention`

multi-head attention

CausalAttention

```
batch = torch.stack((inputs, inputs), dim=0)
print(batch.shape) #A
```

```
torch.Size([2, 6, 3])
```

CausalAttention SelfAttention

Listing 3.3 A compact causal attention class

```
class CausalAttention(nn.Module):
    def __init__(self, d_in, d_ou , context_length, dropout, qkv_
        super().__init__()
        self.d_out = d_out
        self.W_query = nn.Linear( _in, d_out, bias=qkv_bias)
        self.W_key   = nn.Linear( _in, d_out, bias=qkv_bias)
        self.W_value = nn.Linear( _in, d_out, bias=qkv_bias)
        self.dropout = nn.Dropout dropout)  #A
        self.register_buffer(
           'mask',
           torch.triu(torch.ones( ontext_length, context_length
```

```
New batch dimension b
        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  M
```

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 co l t o h cm a u

u

CausalAttention

Listing 3.4 A wrapper class to implement multi-head attention

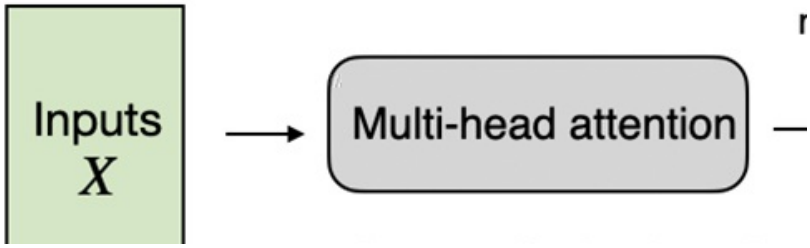
```
class MultiHeadAttentionWrapper(nn.Module):
    def __init__(self, d_in, d_out, context_length,
                 dropout, num_heads, qkv_bias=False):
        super().__init__()
        self.heads = nn.ModuleList(
            [CausalAttention(d_in, d_out, context_length, dropout
             for _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

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

num_heads=2

d_out=2

d_out*num_heads=4

Figure 3.25 Using the `MultiHeadAttentionWrapper`, we specified the number of attention heads (`num_heads`). If we set `num_heads=2`, as shown in this figure, we obtain a tensor with two sets of context vector matrices. In each context vector matrix, the rows represent the context vectors corresponding to the tokens, and the columns correspond to the embedding dimension specified via `d_out=4`. We concatenate these context vector matrices along the column dimension. Since we have 2 attention heads and an embedding dimension of 2, the final embedding dimension is 2 × 2 = 4.



MultiHeadAttentionWrapper CausalAttention

```
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,
context_vecs = mha(batch)

print(context_vecs)
print("context_vecs.shape:", context_vecs.shape)
```

```
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=<CatBack
context_vecs.shape: torch.Size([2, 6, 4])
```

context_vecs

Exercise 3.2 Returning 2-dimensional embedding vectors

MultiHeadAttentionWrapper(..., num_heads=2)

num_heads=2

```
                        [head(x) for head in self.heads]
```

3.6.2 Implementing multi-head attention with weight splits

```
                                MultiHeadAttentionWrapper


CausalAttention

                                        MultiHeadAttentionWrapper
   CausalAttention
MultiHeadAttention
MultiHeadAttentionWrapper        CausalAttention


     MultiHeadAttentionWrapper
             CausalAttention        self.heads
                      CausalAttention

                   MultiHeadAttention




                   MultiHeadAttention
```

Listing 3.5 An efficient multi-head attention class

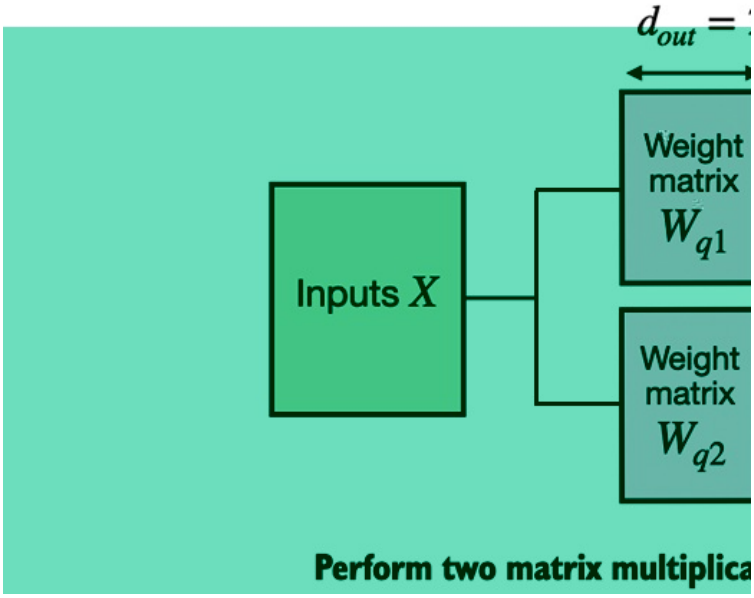
```
class MultiHeadAttention(nn.Module):
    def __init__(self, d_in, d_out,
                 context_length, dropout, num_heads, qkv_bias=Fal
        super().__init__()
        assert d_out % num_heads == 0, "d_out must be divisible b
```

```
self.d_out = d_out
self.num_heads = num_heads
self.head_dim = d_out // num_heads #A
self.W_query = nn.Linear(d_in, d_out, bias=qkv_
```

MultiHeadAttentionWrapper

MultiHeadAttention

Figure 3.26 In the `MultiheadAttentionWrapper` class with two attention heads, we initialized two weight matrices W_{q1} and W_{q2} and computed two query matrices Q_1 and Q_2 as illustrated at the top of this figure. In the `MultiheadAttention` class, we initialize one larger weight matrix W_q only perform one matrix multiplication with the inputs to obtain a query matrix Q, and then split the query matrix into Q_1 and Q_2 as shown at the bottom of this figure. We do the same for the keys and values, which are not shown to reduce visual clutter.



.view .transpose

d_out num_heads head_dim head_dim = d_out / num_heads .view (b, num_tokens, d_out) (b, num_tokens, num_heads, head_dim)

```
                                        num_heads
num_tokens                                (b, num_heads,
num_tokens, head_dim)
```

```
a = torch.tensor([[
```

```
first_res = first_head @ first_head.T
print("First head:\n", first_res)

second_head = a[0, 1, :, :]
second_res = second_head @ second_head.T
print("\nSecond head:\n", second_res)
```

```
                        print(a @ a.transpose(2, 3))
```

```
First head:
 tensor([[1.3208, 1.1631, 1.2879],
        [1.1631, 2.2150, 1.8424],
        [1.2879, 1.8424, 2.0402]])

Second head:
 tensor([[0.4391, 0.7003, 0.5903],
        [0.7003, 1.3737, 1.0620],
        [0.5903, 1.0620, 0.9912]])
```

```
        (b, num_tokens, num_heads, head_dim)
                            (b, num_tokens, d_out)
```

```
                                                    self.out_proj
  MultiHeadAttention
   CausalAttention
```

```
              MultiHeadAttention
MultiHeadAttentionWrapper
```

```
                                                    keys =
self.W_key(x)
```

MultiHeadAttention SelfAttention

MultiHeadAttention

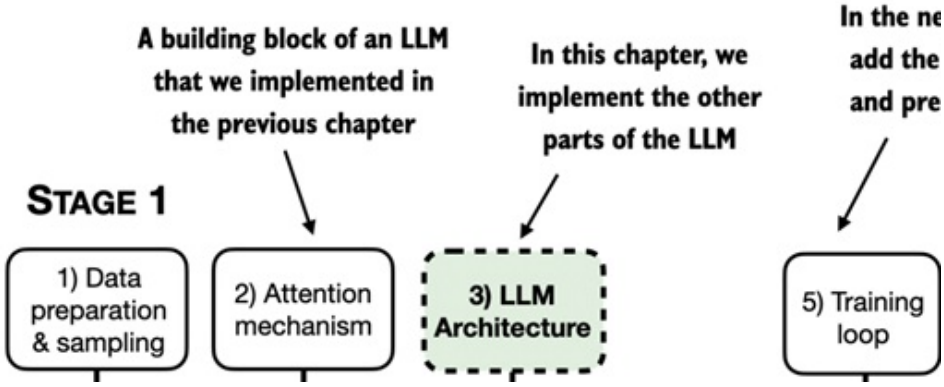
4 Implementing a GPT model from Scratch To Generate Text

This chapter covers

-
-
-
-
-

multi-head attention

Figure 4.1 A mental model of the three main stages of coding an LLM, pretraining the LLM on a general text dataset, and finetuning it on a labeled dataset. This chapter focuses on implementing the LLM architecture, which we will train in the next chapter.



l model of a GPT model. Next to the embedding layers, it consists of one or

et al.

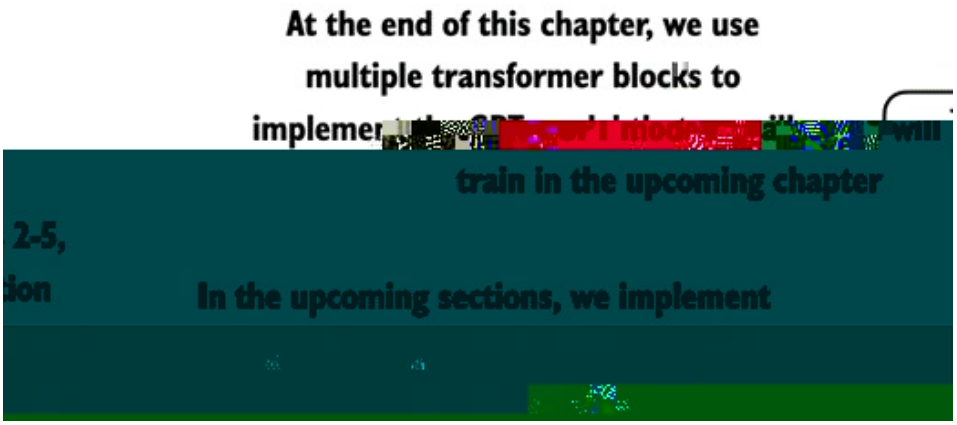
```
GPT_CONFIG_124M = {
    "vocab_size": 50257,  # Vocabulary size
    "context_length": 1024,      # Context length
    "emb_dim": 768,       # Embedding dimension
    "n_heads": 12,        # Number of attention heads
    "n_layers": 12,       # Number of layers
    "drop_rate": 0.1,     # Dropout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

GPT_CONFIG_124M

- "vocab_size"
- "context_length"
- "emb_dim"
- "n_heads"
- "n_layers"
- "drop_rate"
- "qkv_bias" Linear

DummyGPTModel

Figure 4.3 A mental model outlining the order in which we code the GPT architecture. In this chapter, we will start with the GPT backbone, a placeholder architecture, before we get to the individual core pieces and eventually assemble them in a transformer block for the final GPT architecture.



`DummyGPTModel`

Listing 4.1 A placeholder GPT model architecture class

```
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_d
        self.pos_emb = nn.Embedding(cfg["context_length"], cfg["e
        self.drop_emb = nn.Dropout(cfg["drop_rate"])
        self.trf_blocks = nn.Sequential(
            *[DummyTransformerBlock(cfg) for _ in range(cfg["n_la
        self.final_norm = DummyLayerNorm(cfg["emb_dim"]) #B
        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
        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): #C
    def __init__(self, cfg):
        super().__init__()

    def forward(self, x): #D
        return x

class DummyLayerNorm(nn.Module): #E
    def __init__(self, normalized_shape, eps=1e-5): #F
        super().__init__()

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

DummyGPTModel nn.Module

DummyGPTModel

DummyTransformerBlock DummyLayerNorm

out_head

GPT_CONFIG_124M

forward

DummyLayerNorm DummyTransformerBlock

Figure 4.4 A big-picture overview showing how the input data is tokenized, embedded, and fed to the GPT model. Note that in our `DummyGPTClass` coded earlier, the token embedding is handled inside the GPT model. In LLMs, the embedded

```
tensor([[ 6109,  3626,  6100,   345], #A
        [ 6109,  1110,  6622,   257]])
```

DummyGPTModel

batch

```
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, -0.4
         [-0.1192,  0.4539, -0.4432,  ...,  0.2392,  1.3469,  1.2
         [ 0.5307,  1.6720, -0.4695,  ...,  1.1966,  0.0111,  0.5
         [ 0.0139,  1.6755, -0.3388,  ...,  1.1586, -0.0435, -1.0

        [[-1.0908,  0.1798, -0.9484,  ..., -1.6047,  0.2439, -0.4
         [-0.7860,  0.5581, -0.0610,  ...,  0.4835, -0.0077,  1.6
         [ 0.3567,  1.2698, -0.6398,  ..., -0.0162, -0.1296,  0.3
         [-0.2407, -0.7349, -0.5102,  ...,  2.0057, -0.3694,  0.1
       grad_fn=<UnsafeViewBackward0>)
```

DummyLayerNorm

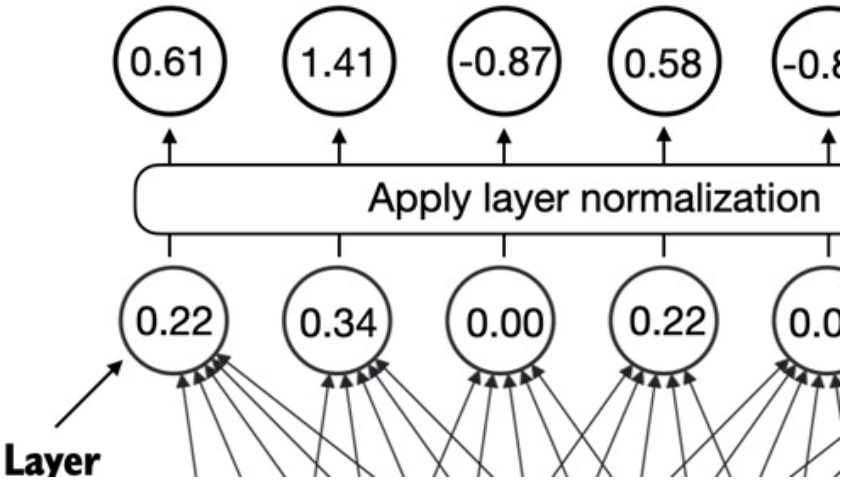
4.2 Normalizing activations with layer normalization

Section A.4, Automatic Differentiation Made Easy Appendix A: Introduction to PyTorch

layer normalization

`DummyLayerNorm`

Figure 4.5 An illustration of layer normalization where the 5 layer outputs, also called activations, are normalized such that they have a zero mean and variance of 1.


0.61
1.41
-0.87
0.58
Apply layer normalization
0.22
0.34
0.00
0.22
Layer

```
mean = out.mean(dim=-1, keepdim=True)
var = out.var(dim=-1, keepdim=True)
print("Mean:\n", mean)
print("Variance:\n", var)
```

```
Mean:
  tensor([[0.1324],
          [0.2170]], grad_fn=<MeanBackward1>)
Variance:
  tensor([[0.0231],
          [0.0398]], grad_fn=<VarBackward0>)
```

keepdim=True

dim

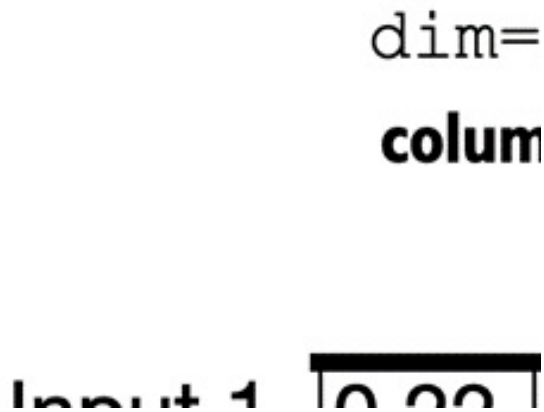
keepdim=True

[0.1324, 0.2170]

[[0.1324], [0.2170]]

dim

Figure 4.6 An illustration of the dim parameter when calculating the mean of a tensor. For instance, if we have a 2D tensor (matrix) with dimensions `[rows, columns]`, using `dim=0` will perform the operation across rows (vertically, as shown at the bottom), resulting in an output that aggregates the data for each column. Using `dim=1` or `dim=-1` will perform the operation across columns (horizontally, as shown at the top), resulting in an output aggregating the data for each row.



dim=-1

dim=1

[batch_size, num_tokens, embedding_size]

dim=-1

dim=1 dim=2

```
out_norm = (out - mean) / torch.sqrt(var)
mean = out_norm.mean(dim=-1, keepdim=True)
var = out_norm.var(dim=-1, keepdim=True)
print("Normalized layer outputs:\n", out_norm)
print("Mean:\n", mean)
print("Variance:\n", var)
```

```
Normalized layer outputs:
 tensor([[ 0.6159,  1.4126, -0.8719,  0.5872, -0.8719, -0.8719],
        [-0.0189,  0.1121, -1.0876,  1.5173,  0.5647, -1.0876]],
       grad_fn=<DivBackward0>)
Mean:
 tensor([[2.9802e-08],
        [3.9736e-08]], grad_fn=<MeanBackward1>)
Variance:
 tensor([[1.],
```

```
        [1.]], grad_fn=<VarBackward0>)
```

```
                            sci_mode

torch.set_printoptions(sci_mode=False)
print("Mean:\n", mean)
print("Variance:\n", var)
Mean:
 tensor([[    0.0000],
        [    0.0000]], grad_fn=<MeanBackward1>)
Variance:
 tensor([[1.],
        [1.]], grad_fn=<VarBackward0>)
```

Listing 4.2 A layer normalization class

```
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
```

```
 emb_dim             eps
```

```
Variance:
 tensor([[1.0000],
         [1.0000]], grad_fn=<VarBackward0>)
```

Figure 4.7 A mental model listing the different building blocks we implement in this chapter to assemble the GPT architecture.

Layer normalization versus batch normalization

4.3 Implementing a feed forward network with

GELU activations

GELU

Gaussian Error Linear Unit *Sigmoid-Weighted Linear Unit*

. . . .

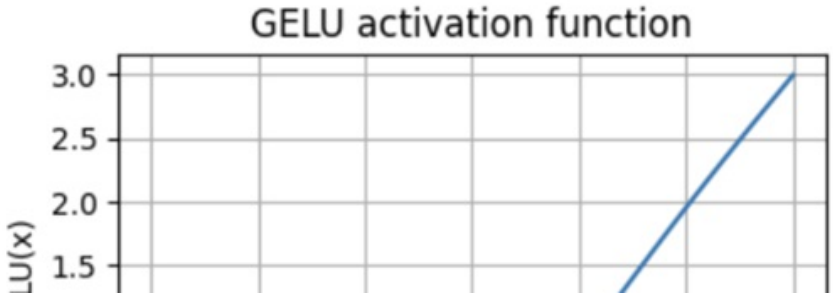
Listing 4.3 An implementation of the GELU activation function

```
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))
        ))
```

```
import matplotlib.pyplot as plt
gelu, relu = GELU(), nn.ReLU()

x = torch.linspace(-3, 3, 100) #A
y_gelu, y_relu = gelu(x), relu(x)
plt.figure(figsize=(8, 3))
for i, (y, label) in enumerate(zip([y_gelu, y_relu], ["GELU", "Re
    plt.subplot(1, 2, i)
    plt.plot(x, y)
    plt.title(f"{label} activation function")
    plt.xlabel("x")
    plt.ylabel(f"{label}(x)")
    plt.grid(True)
plt.tight_layout()
plt.show()
```

Figure 4.8 The output of the GELU and ReLU plots using matplotlib. The x-axis shows the function inputs and the y-axis shows the function outputs.



FeedForward

Listing 4.4 A feed forward neural network module

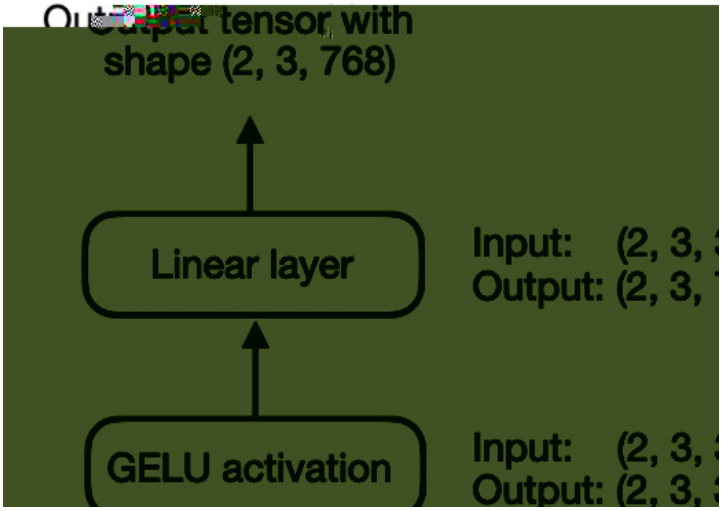
```
class FeedForward(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Linear(cfg["emb_dim"], 4 * cfg["emb_dim"]),
            GELU(),
            nn.Linear(4 * cfg["emb_dim"], cfg["emb_dim"]),
        )

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

FeedForward

Linear GELU

GPT_CONFIG_124M GPT_CONFIG_124M["emb_dim"] = 768

Figure 4.9 provides a visual overview of the connections between the layers of the feed forward neural network. It is important to note that this neural network can accommodate variable batch sizes and numbers of tokens in the input. However, the embedding size for each token is determined and fixed when initializing the weights.



FeedForward

[

```
ffn = FeedForward(GPT_CONFIG_124M)
x = torch.rand(2, 3, 768) #A
out = ffn(x)
print(out.shape)
```

```
torch.Size([2, 3, ▫
```

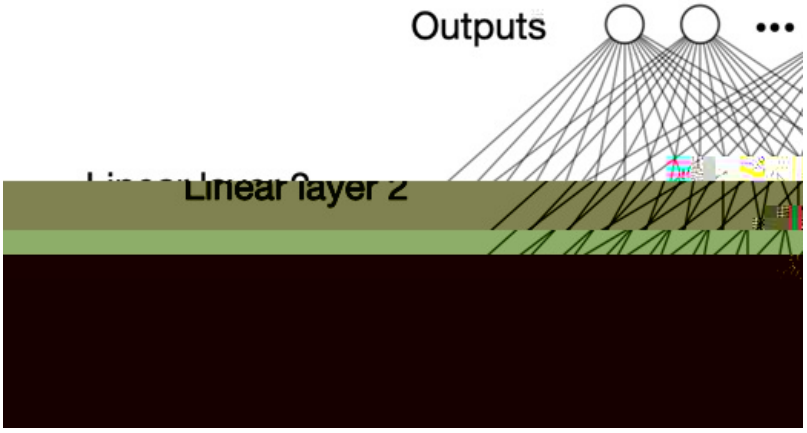
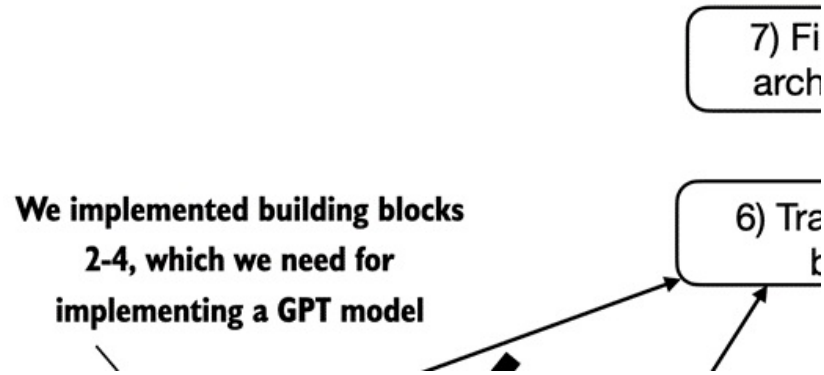


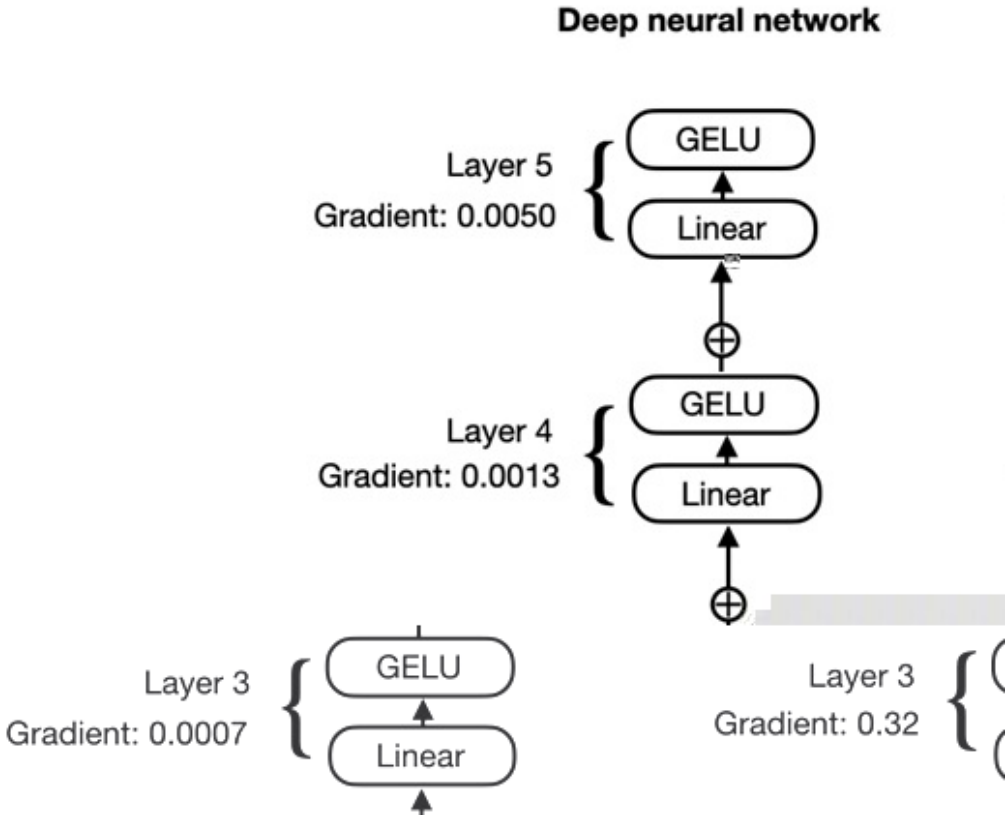
Figure 4.11 A mental model showing the topics we cover in this chapter, with the black checkmarks indicating those that we have already covered.



4.4 Adding shortcut connections

shortcut connections

Figure 4.12 A comparison between a deep neural network consisting of 5 layers without (on the left) and with shortcut connections (on the right). Shortcut connections involve adding the inputs of a layer to its outputs, effectively creating an alternate path that bypasses certain layers. The gradient illustrated in Figure 1.1 denotes the mean absolute gradient at each layer, which we will compute in the code example that follows.



```
forward
```

Listin

```
layer_sizes = [3, 3, 3, 3, 3, 1]
sample_input = torch.tensor([[1., 0., -1.]])
torch.manual_seed(123) # specify random seed for the initial weig
model_without_shortcut = ExampleDeepNeuralNetwork(
    layer_sizes, use_shortcut=False
)



def print_gradients(model, x):
    # Forward pass
    output = model(x)
    target = torch.te
```

print_gradients

```
print_gradients(model_without_shortcut, sample_input)
```

```
layers.0.0.weight has gradient mean of 0.00020173587836325169
layers.1.0.weight has gradient mean of 0.00012011116101583466
layers.2.0.weight has gradient mean of 0.00071520416531711 82
layers.3.0.weight has gradient mean of 0.001398873864673078
layers.4.0.weight has gradient mean of 0.005049646366387606
```

print_gradients

layers.4

layers.0

```
torch.manual_seed(123)
model_with_shortcut = ExampleDeepNeuralNetwork(
    layer_sizes, use_shortcut=True
)
print_gradients(model_with_shortcut, sample_input)
```

```
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
```

(layers.4

layers.0

ry tu p q

p p q y n tp q

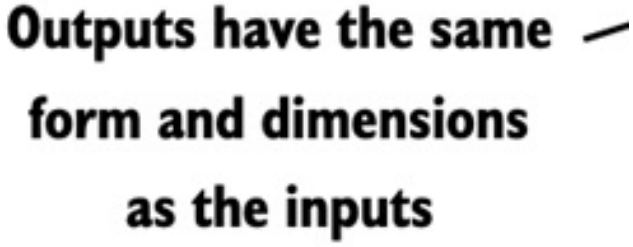
4.5 Connecting attempt c

8 > =K

Γ

\

f



`FeedForward`

`TransformerBlock`

Listing 4.6 The transformer block component of GPT

```
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"],
            block_size=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_resid = nn.Dropout(cfg["drop_rate"])

    def forward(self, x):
        #A
        shortcut = x
        x = self.norm1(x)
        x = self.att(x)
        x = self.drop_resid(x)
        x = x + shortcut  # Add the original input back

        shortcut = x #B
        x = self.norm2(x)
        x = self.ff(x)
        x = self.drop_resid(x)
        x = x + shortcut  #C
        return x
```

TransformerBlock

MultiHeadAttention

FeedForward

cfg GPT_CONFIG_124M

LayerNorm

Pre-LayerNorm

GPT_CONFIG_124M

```
torch.manual_seed(123)
x = torch.rand(2, 4, 768)  #A
block = TransformerBlock(GPT_CONFIG_124M)
output = block(x)

print("Input shape:", x.shape)
print("Output shape:", output.shape)
```

```
Input shape: torch.Size([2, 4, 768])
Output shape: torch.Size([2, 4, 768])
```

Figure 4.14 A mental model of the different concepts we have implemented in this chapter so far.

4.6 Coding the GPT model

`DummyGPTModel` `DummyGPTModel`

`DummyTransformerBlock` `DummyLayerNorm`

`DummyTransformerBlock`
`DummyLayerNorm` `TransformerBlock`
`LayerNorm`

Figure 4.15 An overview of the GPT model architecture. This figure illustrates the flow of data through the GPT model. Starting from the bottom, tokenized text is first converted into token embeddings, which are then augmented with positional embeddings. This combined information forms a tensor that is passed through a series of transformer blocks shown in the center.

Listing 4.7 The GPT model architecture implementation

```
class GPTModel(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.tok_emb = nn.Embedding(cfg["vocab_size"], cfg["emb_d
        self.pos_emb = nn.Embedding(cfg["context_length"], cfg["e
        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)
        #A
        pos_embeds = self.pos_emb(torch.arange(seq_len, device=in
        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
```

TransformerBlock

GPTModel

__init__ GPTModel

cfg

__init__ TransformerBlock
cfg
LayerNorm

GPT_CONFIG_124M

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

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

```
Input batch:
 tensor([[ 6109,  3626,  6100,   345], # token IDs of text 1
         [ 6109,  1110,  6622,   257]]) # token IDs of text 2

Output shape: torch.Size([2, 4, 50257])
tensor([[[ 0.3613,  0.4222, -0.0711,  ...,  0.3483,  0.4661, -0.2
         [-0.1792, -0.5660, -0.9485,  ...,  0.0477,  0.5181, -0.3
         [ 0.7120,  0.0332,  0.1085,  ...,  0.1018, -0.4327, -0.2
         [-1.0076,  0.3418, -0.1190,  ...,  0.7195,  0.4023,  0.0

        [[-0.2564,  0.0900,  0.0335,  ...,  0.2659,  0.4454, -0.6
         [ 0.1230,  0.3653, -0.2074,  ...,  0.7705,  0.2710,  0.2
         [ 1.0558,  1.0318, -0.2800,  ...,  0.6936,  0.3205, -0.3
```

```
  [-0.1565,  0.3926,  0.3288,  ...,  1.2630, -0.1858,  0.0
grad_fn=<UnsafeViewBackward0>)
```

[2, 4, 50257]

numel()

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

```
Total number of parameters: 163,009,536
```

model GPTModel

```
print("Token embedding layer shape:", model.tok_emb.weight.shape)
print("Output layer shape:", model.out_head.weight.shape)
```

```
Token embedding layer shape: torch.Size([50257, 768])
Output layer shape: torch.Size([50257, 768])
```

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

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

GPTModel

Exercise 4.1 Number of parameters in feed forward and attention modules

GPTModel

```
total_size_bytes = total_params * 4  #A
total_size_mb = total_size_bytes / (1024 * 1024)  #B
print(f"Total size of the model: {total_size_mb:.2f} MB")
```

```
Total size of the model: 621.83 MB
```

```
GPTModel
```

```
[batch_size, num_tokens, vocab_size]
```

Exercise 4.2 Initializing larger GPT models

4.7 Generating text

Figure 4.16 This diagram illustrates the step-by-step process by which an LLM generates text, one token at a time. Starting with an initial input context ("Hello, I am"), the model predicts a subsequent token during each iteration, appending it to the input context for the next round of prediction. As shown, the first iteration adds "a", the second "model", and the third "ready", progressively building the sentence.

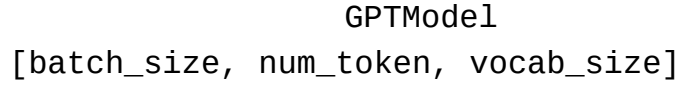
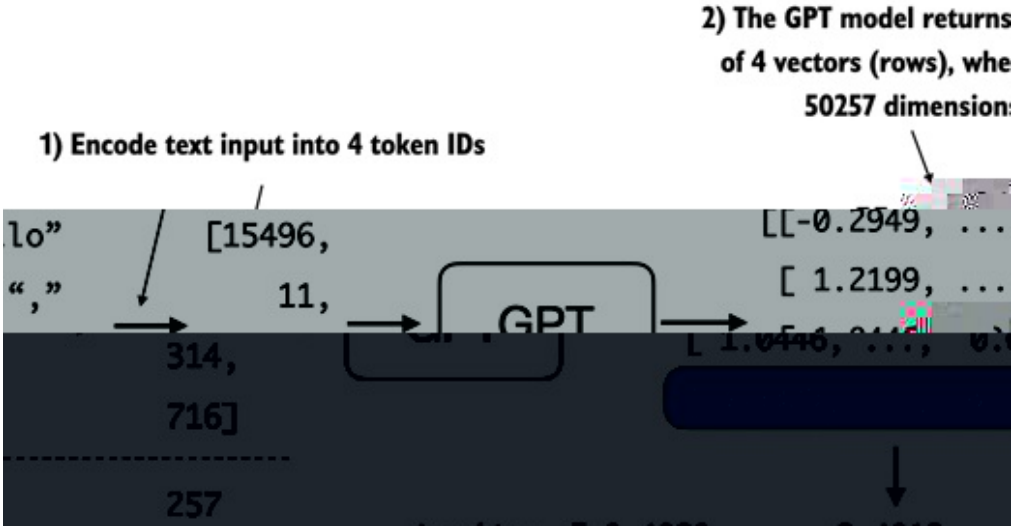


Figure 4.17 details the mechanics of text generation in a GPT model by showing a single iteration in the token generation process. The process begins by encoding the input text into token IDs, which are then fed into the GPT model. The outputs of the model are then converted back into text and appended to the original input text.



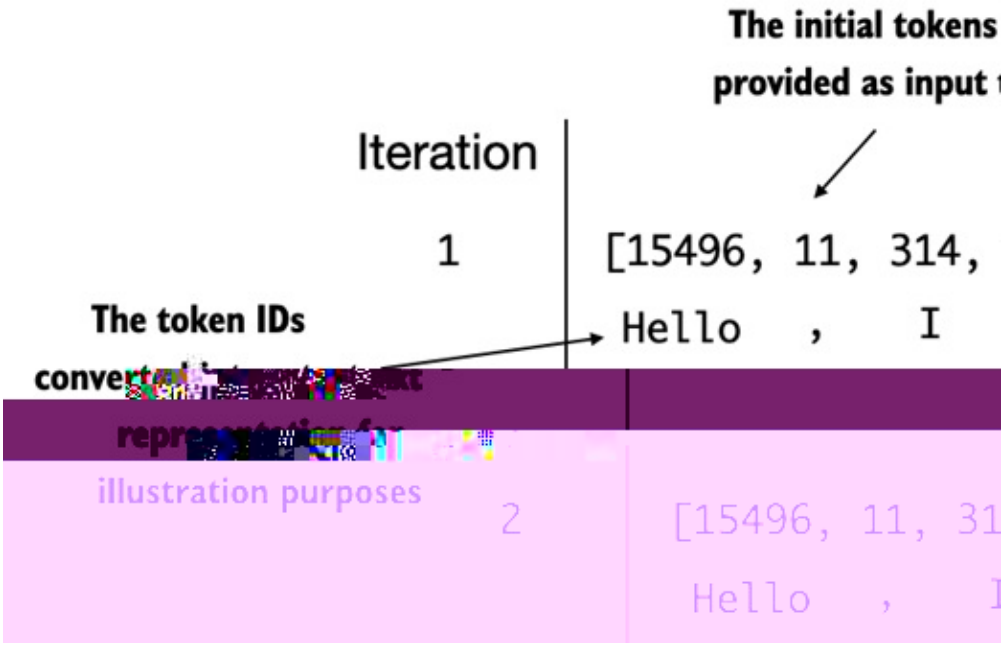
Listing 4.8 A function for the GPT model to generate text

```
def generate_text_simple(model, idx, max_new_tokens, context_size
    for _ in range(max_new_tokens):
        idx_cond = idx[:, -context_size:] #B
```

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

        logits = logits[:, -1, :] #C
        probas = torch.softmax(logits, dim=-1)  #D
        idx_next = torch.argmax(probas, dim=-1, keepdim=True) #E
        idx = torch.cat((idx, idx_next), dim=1)  #F

    return idx
```

generate_text_simple

takes a sequence of initial token IDs as input, predicts the next token, and appends this token to the input sequence for the next iteration. (The token IDs are also translated into their corresponding text for better understanding.)



generate_text_simple "Hello, I am"

start_context = "Hello

```
encoded: [15496, 11, 314, 716]
encoded_tensor.shape: torch.Size([1, 4])
```

```
                        .eval()
```

```
generate_text_simple
```

```
model.eval() #A
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: tensor([[15496,    11,   314,   716, 27018, 24086, 47843,
Output length: 10
```

```
         .decode
```

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

```
Hello, I am Featureiman Byeswickattribute argue logger Normandy C
```

Exercise 4.3 Using separate dropout parameters

`"drop_rate"`

`GPT_CONFIG_124M`

4.8 Summary

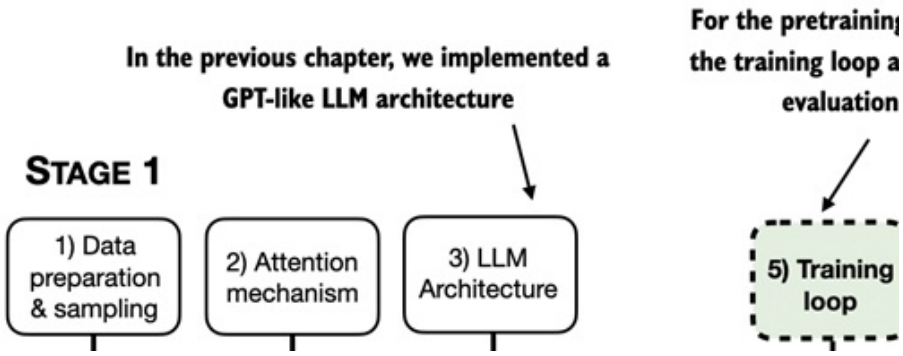
-
-

5 Pretraining on Unlabeled Data

This chapter covers

-
-
-
-

Figure 5.1 A mental model of the three main stages of coding an LLM, pretraining the LLM on a general text dataset and finetuning it on a labeled dataset. This chapter focuses on pretraining the LLM, which includes implementing the training code, evaluating the performance, and saving and loading model weights.



Weight parameters

weights

weight parameters *parameters*

`GPTModel` `new_layer =`
`torch.nn.Linear(...)` `.weight`
`new_layer.weight`

`model.parameters()`

5.1 Evaluating generative text models

Figure 5.2 An overview of the topics covered in this chapter. We begin by recapping the

GPTModel

GPT_CONFIG_124M

```
import torch
from chapter04 import GPTModel
GPT_CONFIG_124M = {
    "vocab_size": 50257,
    "context_length": 256,  #A
    "emb_dim": 768,
    "n_heads": 12,
    "n_layers": 12,
    "drop_rate": 0.1,     #B
    "qkv_bias": False
}
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.eval()
```

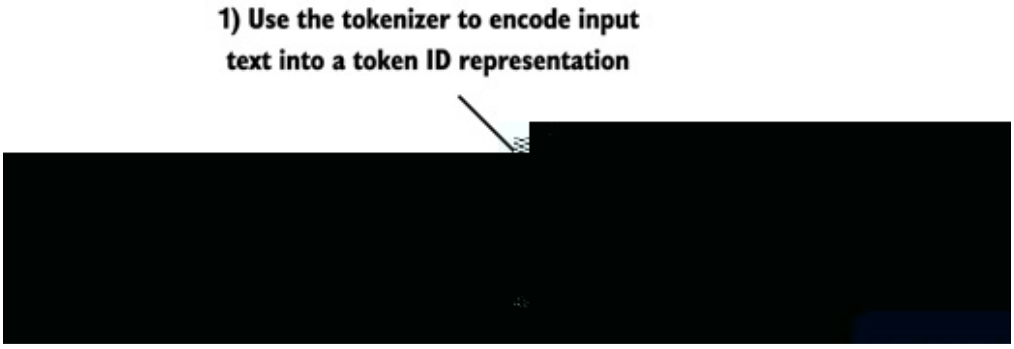
GPT_CONFIG_124M

context_length

GPTmodel generate_text_simple

text_to_token_ids token_ids_to_text

Figure 5.3 Generating text involves encoding text into token IDs that the LLM processes into logit vectors. The logit vectors are then converted back into token IDs, detokenized into a text representation.



Listing 5.1 Utility functions for text to token ID conversion

```
import tiktoken
from chapter04 import generate_text_simple

def text_to_token_ids(text, tokenizer):
    encoded = tokenizer.encode(text, allowed_special={'<|endoftex
    encoded_tensor = torch.tensor(encoded).unsqueeze(0) # add bat
    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))
```

model

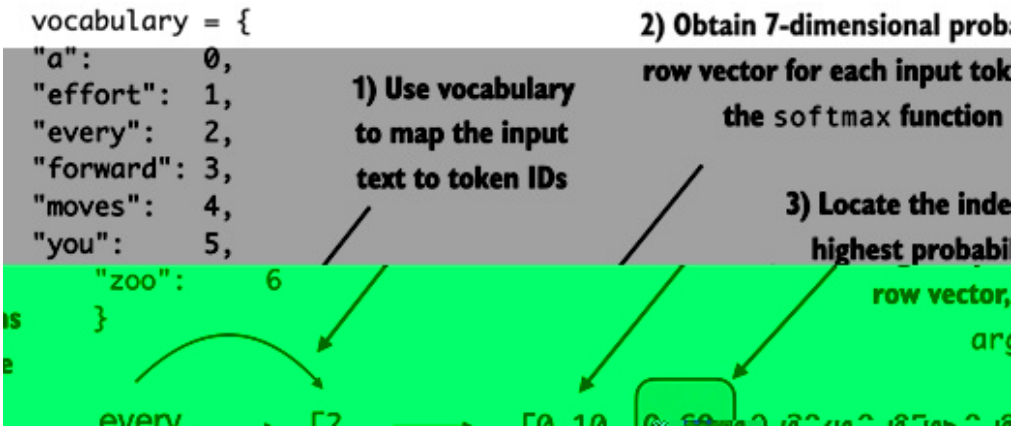
```
Output text:
 Every effort moves you rentingetic wasn  refres RexMeCHicular st
```

loss metric

5.1.2 Calculating the text generation loss

generate_text_simple

Figure 5.4 For each of the 3 input tokens, shown on the left, we compute a vector containing probability scores correspondin



generate_text_simple

GPTModel

"every effort moves"

"every effort moves" "I really like"

inputs

```
with torch.no_grad(): #A
    logits = model(inputs)
probas = torch.softmax(logits, dim=-1) # Probability of each toke
print(probas.shape)
```

probas

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

inputs

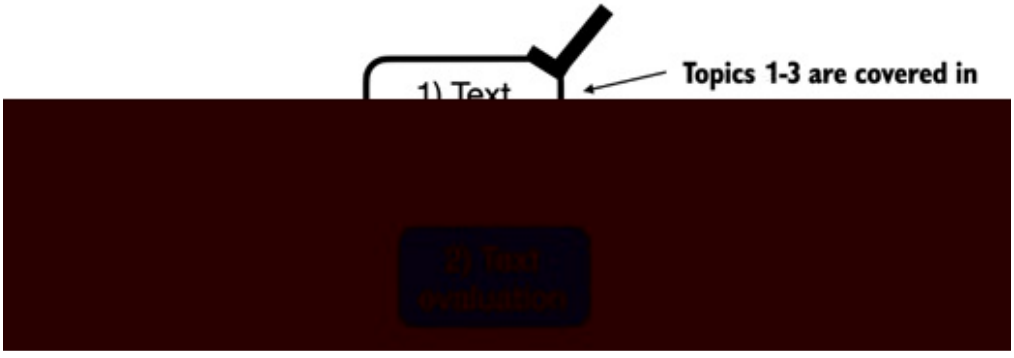
generate_text_simple

```
token_ids = torch.argmax(probas, dim=-1, keepdim=True)
print("Token IDs:\n", token_ids)
```

```
Token IDs:
 tensor([[[16657], # First batch
         [  339],
         [42826]],
        [[49906], # Second batch
         [29669],
         [41751]]])
Finally, step 5 converts the token IDs back into text:
print(f"Targets batch 1: {token_ids_to_text(targets[0], tokenizer
print(f"Outputs batch 1: {token_ids_to_text(token_ids[0].flatten(
```

```
Targets batch 1:  effort moves you
Outputs batch 1:  Armed heNetflix
```

Figure 5.5 We now implement the text evaluation function in the remainder of this section. In the next section, we apply this evaluation function to the entire dataset we use for model training.



```
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)
```

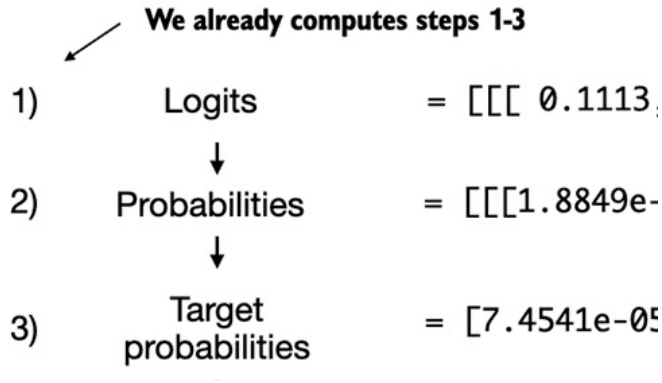
```
Text 1: tensor([7.4541e-05, 3.1061e-05, 1.1563e-05])
Text 2: tensor([3.9836e-05, 1.6783e-05, 4.7559e-06])
```

Backpropagation

backpropagation

target_probas_1 target_probas_2

Figure 5.7 Calculating the loss involves several steps. Steps 1 to 3 calculate the token probabilities corresponding to the target tensors. These probabilities are then transformed via a logarithm and averaged i




```
target_probas_1    target_probas_2
```

logarithm

```
log_probas = torch.log(torch.cat((target_probas_1, target_probas_
print(log_probas)
```

```
tensor([ -9.5042, -10.3796, -11.3677, -10.1308, -10.9951, -12.256
```

```
avg_log_probas = torch.mean(log_probas)
print(avg_log_probas)
```

```
tensor(-10.7722)
```

```
neg_avg_log_probas = avg_log_probas * -1
print(neg_avg_log_probas)
```

```
tensor(-10.7722)
```

cross entropy

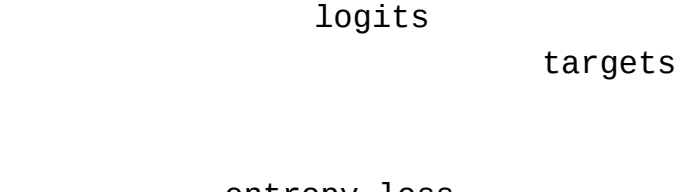
cross_entropy

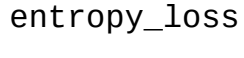
Cross entropy loss

cross_entropy

```
print("Logits shape:", logits.shape)
print("Targets shape:", targets.shape)
```

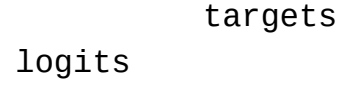
```
Logits shape: torch.Size([2, 3, 50257])
Targets shape: torch.Size([2, 3])
```

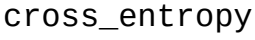





```
logits_flat = logits.flatten(0, 1)
targets_flat = targets.flatten()
print("Flattened logits:", logits_flat.shape)
print("Flattened targets:", targets_flat.shape)
```

```
Flattened logits: torch.Size([6, 50257])
Flattened targets: torch.Size([6])
```






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

```
tensor(10.7722)
```

Perplexity

Perplexity

```
                    perplexity = torch.exp(loss)
tensor(47678.8633)
```

5.1.3 Calculating the training and valid

The cost of pretraining LLMs

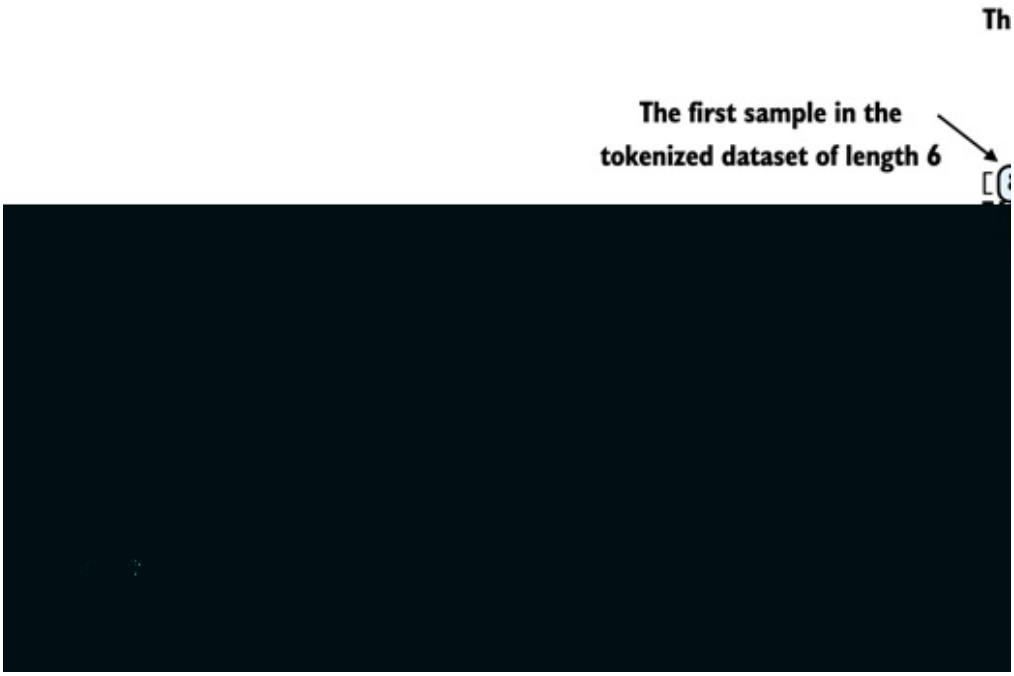
```
file_path = "the-verdict.txt"
with open(file_path, "r", encoding="utf-8") as file:
    text_data = file.read()
```

```
total_characters = len(text_data)
total_tokens = len(tokenizer.encode(text_data))
print("Characters:", total_characters)
print("Tokens:", total_tokens)
```

```
Characters: 20479
Tokens: 5145
```

GPTModel

Figure 5.9 When preparing the data loaders, we split the input text into training and validation set portions. Then, we tokenize the text (only shown for the training set portion for simplicity) and divide the tokenized text into chunks of a user-specified length (here 6). Finally, we shuffle the rows and organize the chunked text into batches (here, batch size 2), which we can use for model training.



max_length=6

max_length

Training with variable lengths

train_ratio

```
train_ratio = 0.90
split_idx = int(train_ratio * len(text_data))
train_data = text_data[:split_idx]
val_data = text_data[split_idx:]
```

train_data val_data

create_dataloader_v1

```
from chapter02 import create_dataloader_v1
torch.manual_seed(123)

train_loader = create_dataloader_v1(
    train_data,
    batch_size=2,
    max_length=GPT_CONFIG_124M["context_length"],
    stride=GPT_CONFIG_124M["context_length"],
    drop_last=True,
    shuffle=True
)
val_loader = create_dataloader_v1(
    val_data,
    batch_size=2,
    max_length=GPT_CONFIG_124M["context_length"],
    stride=GPT_CONFIG_124M["context_length"],
    drop_last=False,
    shuffle=False
)
```

```
print("Train loader:")
for x, y in train_loader:
    print(x.shape, y.shape)

print("\nValidation loader:")
for x, y in val_loader:
    print(x.shape, y.shape)
```

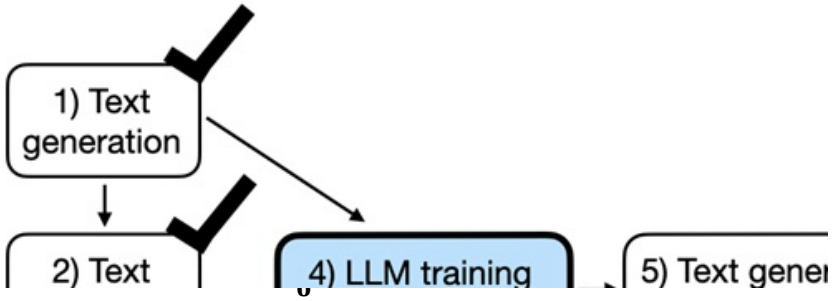
```
    loss = torch.nn.functional.cross_entropy(
        logits.flatten(0, 1), target_batch.flatten()
    )
    return loss

                calc_loss_batch
                                          calc_loss_loader
```

Listing 5.2 Function to compute the training and validation loss

```
def calc_loss_loader(data_loader, model, device, num_batches=None
    total_loss = 0.
    if num_batches is None:
        num_batches = len(data_loader) #A
    else:
        num_batches = mi          h,       d        e
```

```
Training loss: 10.98758347829183
Validation loss: 10.98110580444336
```

Figure 5.10 We have recapped the text generation process and implemented basic model evaluation techniques to mpu e th training an validation set losses. Next, we will go to the training functions and pretrain the LLM.



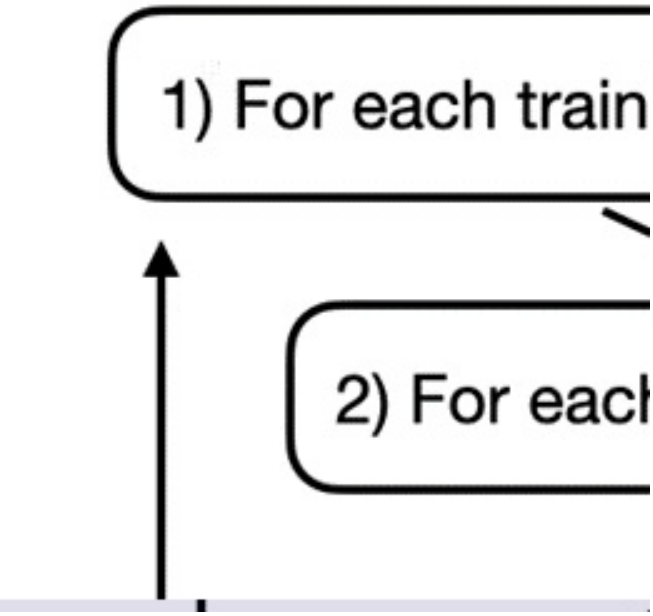
5.2 Training an LLM

`GPTModel`

earning rate warmup cosine annealing gradient clipping Appendix D, Adding Bells and Whistles to the Training Loop.

Figure 5.11 A typical training loop for training deep neural networks in PyTorch consists of several steps, iterating over the batches in the training set for several epochs. In each loop, we calculate the loss for each training set batch to determine loss gradients, which we use to update

the model weights so that the training set loss is minimized.



`train_model_simple`

Listing 5.3 The main function for pretraining LLMs

```
def train_model_simple(model, train_loader, val_loader, optimizer
                       eval_freq, eval_iter, start_context):
    train_losses, val_losses, track_tokens_seen = [], [], [] #A
    tokens_seen, global_step = 0, -1

    for epoch in range(num_epochs): #B
        model.train()
        for input_batch, target_batch in train_loader:
```

```
optimizer.zero_grad() #C
loss = calc_loss_batch(input_batch, target_batch, mod
loss.backward() #D
optimizer.step() #E
tokens_seen += input_batch.numel()
global_step += 1

if global_step % eval_freq == 0: #F
    train_loss, val_loss = evaluate_model(
        model, train_loader, val_loader, device, eval
    train_losses.append(train_loss)
    val_losses.append(val_loss)
    track_tokens_seen.append(tokens_seen)
    print(f"Ep {epoch+1} (Step {global_  Ep   d}   b
```

generate_and_print_sample

start_context

generate_text_simple

```
def generate_and_print_sample(model, tokenizer, device, start_con
    model.eval()
    context_size = model.pos_emb.weight.shape[0]
    encoded = text_to_token_ids(start_context, tokenizer).to(devi
    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 f
    model.train()
```

evaluate_model

generate_and_print_sampl

AdamW

Adam

AdamW

train_model_simple

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

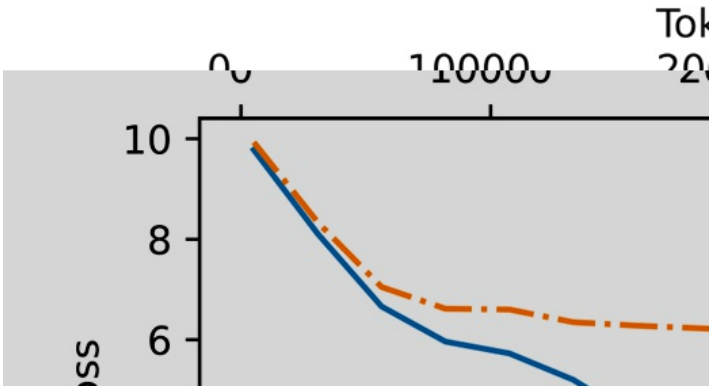
```
num_epochs = 10
train_losses, val_losses, tokens_seen = train_model_simple(
    model, train_loader, val_loader, optimizer, device,
    num_epochs=num_epochs, eval_freq=5, eval_iter=1,
    start_context="Every effort moves you"
)
```

training_model_simple

```
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, a
[...]  #A
```

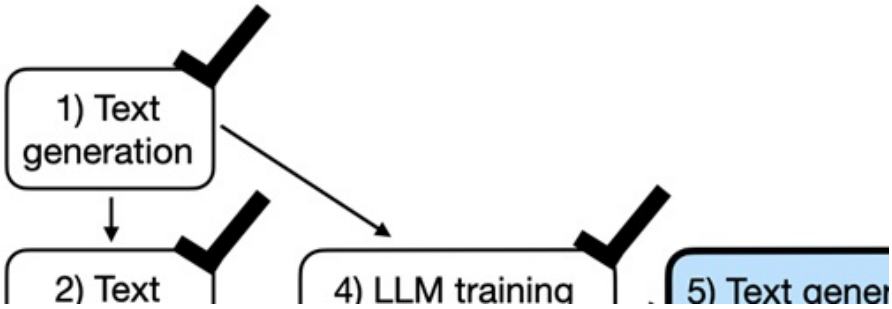
```
    ax1.set_ylabel("Loss")
    ax1.legend(loc="upper right")
    ax2 = ax1.twiny()  #A
    ax2.plot(tokens_seen, train_losses, alpha=0)  #B
    ax2.set_xlabel("Tokens seen")
    fig.tight_layout()
    plt.show()

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

Figure 5.12 At the beginning of the training, we observe that both the training and validation set losses sharply decrease, which is a sign that the model is learning. However, the training set loss continues to decrease past the second epoch, whereas the validation loss stagnates. This is a sign that the model is still learning, but it's overfitting to the training set past epoch 2.




```
                                "quite insensible to the irony"
   "The Verdict"
```

Figure 5.13 Our model can generate coherent text after implementing the training function. However, it often memorizes passages from the training set verbatim. The following section covers strategies to generate more diverse output texts.



5.3 Decoding strategies to control randomness

`generate_text_simple`

`generate_and_print_sample`

temperature scaling *top-k sampling*

```
model.to("cpu")
model.eval()
```

GPTModel model generate_text_simple

```
tokenizer = tiktoken.get_encoding("gpt2")
token_ids = generate_text_simple(
    model=model,
    idx=text_to_token_ids("Every effort moves you", tokenizer),
    max_new_tokens=25,
    context_size=GPT_CONFIG_124M["context_length"]
)
print("Output text:\n", token_ids_to_text(token_ids, tokenizer))
```

```
Output text:
Every effort moves you know," was one of the axioms he laid down
```

generate_text_simple
"Every effort moves you"

5.3.1 Temperature scaling

generate_text_simple
torch.argmax
greedy decoding

```
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()}
```

"every effort moves you"

```
next_token_logits = torch.tensor(
    [4.51, 0.89, -1.90, 6.75, 1.63, -1.62, -1.89, 6.28, 1.79]
)
```

generate_text_simple

]

]

multinomial

```
torch.manual_seed(123)
next_token_id = torch.multinomial(probas, num_samples=1).item()
print(inverse_vocab[next_token_id])
```

"forward"

multinomial

"forward"

multinomial

```
def print_sampled_tokens(probas):
    torch.manual_seed(123)
    sample = [torch.multinomial(probas, num_samples=1).item() for
    sampled_ids = torch.bincount(torch.tensor(sample))
    for i, freq in enumerate(sampled_ids):
        print(f"{freq} x {inverse_vocab[i]}")
print_sampled_tokens(probas)
```

The sampling output is as follows:

```
73 x closer
0 x every
0 x effort
582 x forward
2 x inches
0 x moves
0 x pizza
343 x toward
```

"forward"

"closer"

"inches" "toward"

argmax multinomial

generate_and_print_sample

every effort moves you toward every effort moves you inches every effort moves you closer" every effort moves you forward

temperature scaling

```
def softmax_with_temperature(logits, temperature):
    scaled_logits = logits / temperature
    return torch.softmax(scaled_logits, dim=0)
```

```
temperatures = [1, 0.1, 5]  # Original, higher, and lower tempera
scaled_probas = [softmax_with_temperature(next_token_logits, T) f
x = torch.arange(len(vocab))
bar_width = 0.15
fig, ax = plt.subplots(figsize=(5, 3))
for i, T in enumerate(temperatures):
    rects = ax.bar(x + i * bar_width, scaled_probas[i],
                   bar_width, label=f'Temperature = {T}')
ax.set_ylabel('Probability')
ax.set_xticks(x)
ax.set_xticklabels(vocab.keys(), rotation=90)
ax.legend()
plt.tight_layout()
plt.show()
```

multinomial

multinomial "forward"

"every effort moves you pizza"

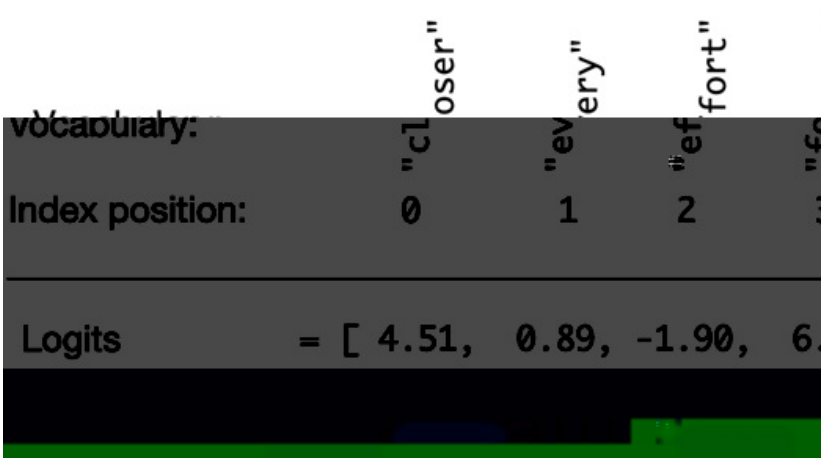
Exercise 5.1

print_sampled_tokens

"every effort moves you pizza".

top-k sampling

Figure 5.15 Using top-k sampling with k=3, we focus on the 3 tokens associated with the highest logits and mask out all other tokens with negative infinity (-inf) before applying the softmax function. This results in a probability distribution with a probability value 0 assigned to all non-top-k tokens.



-inf

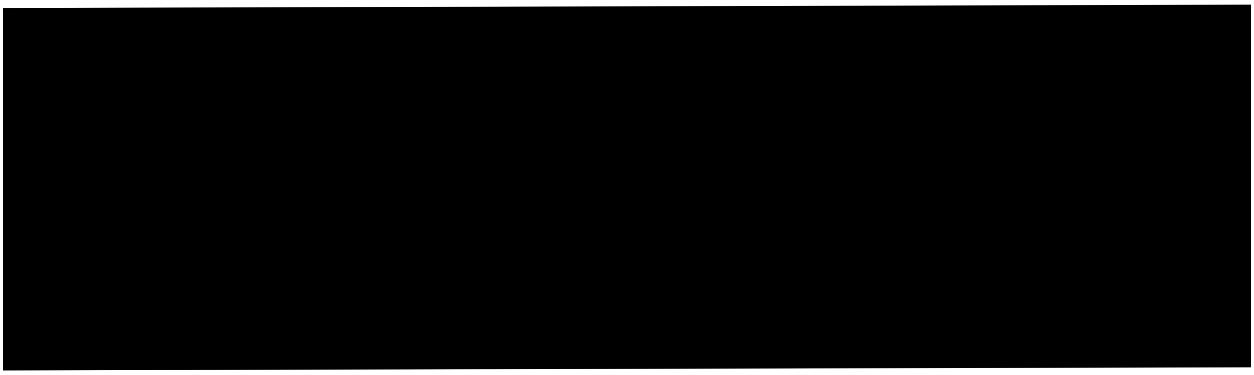
Applying a causal attention mask

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

```
print("Top logits:", top_logits)
print("Top positions:", top_pos)
```

```
Top logits: tensor([6.7500, 6.2800, 4.5100])
Top positions: tensor([3, 7, 0])
```

where



5.3.3 Modifying the text generation function

```
generate_simple
            generate
```

Listing 5.4 A modified text generation function with more diversity

```
def generate(model, idx, max_new_tokens, context_size, temperatur
    for _ in range(max_new_tokens):  #A
        idx_cond = idx[:, -context_size:]
        with torch.no_grad():
            logits = model(idx_cond)
        logits = logits[:, -1, :]
        if top_k is not None:  #B
            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
            )
        if temperature > 0.0:  #C
            logits = logits / temperature
            probs = torch.softmax(logits, dim=-1)
            idx_next = torch.multinomial(probs, num_samples=1)
        else:  #D
            idx_next = torch.argmax(logits, dim=-1, keepdim=True)
        idx = torch.cat((idx, idx_next), dim=1)
    return idx


                  generate

torch.manual_seed(123)
token_ids = generate(
    model=model,
    idx=text_to_token_ids("Every effort moves you", tokenizer),
    max_new_tokens=15,
    context_size=GPT_CONFIG_124M["context_length"],
    top_k=25,
    temperature=1.4
)
```

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

```
Output text:
 Every effort moves you stand to work on surprise, a one of us ha
```

`generate_simple`

```
 "Every effort moves you know," was one of the axioms he
laid...!"
```

Exercise 5.2

Exercise 5.3

`generate`

`generate_simple`

5.4 Loading and saving model weights in PyTorch

GPTModel

Figure 5.16 After training and inspecting the model, it is often helpful to save the model so that we can use or continue training it later, which is the topic of this section before we load the pretrained model weights from OpenAI in the final secti. ct. B

model.eval()

model

train_model_simple

5.5 Loading pretrained weights from OpenAI

weights

.weight Linear

Embedding

model.parameters()

tqdm

```
pip install tensorflow>=2.15.0  tqdm>=4.66
```

gpt_download.py

```
import urllib.request
```

```
url = (
    "https://raw.githubusercontent.com/rasbt/"
    "LLMs-from-scratch/main/ch05/"
    "01_main-chapter-code/gpt_download.py"
)
filename = url.split('/')[-1]
urllib.request.urlretrieve(url, filename)
```

```
                    download_and_load_gpt2
gpt_download.py
        settings                        params

from gpt_download import download_and_load_gpt2
settings, params = download_and_load_gpt2(model_size="124M", mode
```

```
checkpoint: 100%|████████████████████████| 77.0/77.0 [00:00<00
encoder.json: 100%|                       | 1.04M/1.04M [00:00<
hprams.json: 100%|                        | 90.0/90.0 [00:00<00
model.ckpt.data-00000-of-00001: 100%|     | 498M/498M [01:09<00
model.ckpt.index: 100%|                   | 5.21k/5.21k [00:00<
model.ckpt.meta: 100%|                    | 471k/471k [00:00<00
vocab.bpe: 100%|                          | 456k/456k [00:00<00
```

Updated download instructions

```
print("Settings:", settings)
print("Parameter dictionary keys:", params.keys())
```

```
Settings: {'n_vocab': 50257, 'n_ctx': 1024, 'n_embd': 768, 'n_hea
Parameter dictionary keys: dict_keys(['blocks', 'b', 'g', 'wpe',
```

```
    settings    params                          settings
```

```
GPT_CONFIG_124M           params
```

```
print(params)
```

```
print(params["wte"])
print("Token embedding weight tensor dimensions:", params["wte"].
```

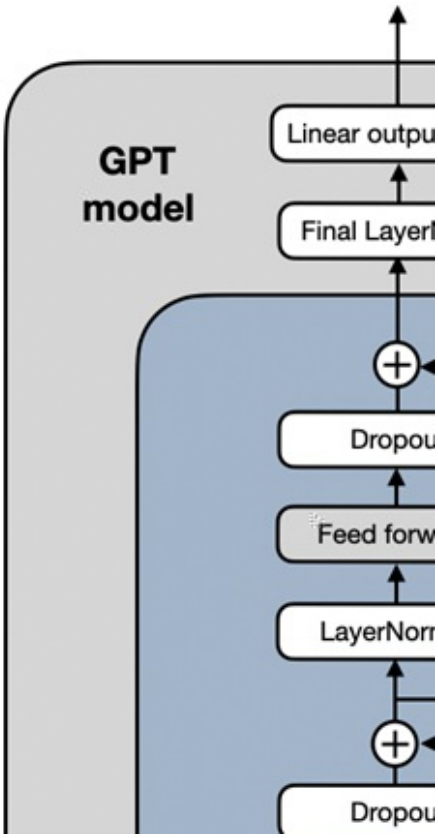
```
[[-0.11010301 ... -0.1363697   0.01506208   0.04531523]
 [ 0.04034033 ...  0.08605453  0.00253983   0.04318958]
 [-0.12746179  ...  0.08991534 -0.12972379 -0.08785918]
 ...
 [-0.04453601 ...   0.10435229  0.09783269 -0.06952604]
 [ 0.1860082  ...  -0.09625227  0.07847701 -0.02245961]
 [ 0.05135201 ...   0.00704835  0.15519823  0.12067825]]
Token embedding weight tensor dimensions: (50257, 768)
```

```
download_and_load_gpt2(model_size="124M", ...)
                                                    "355M" "774M"
   "1558M"
```

Figure 5.17 GPT-2 LLMs come in several different model sizes, ranging from 124 million to 1,558 million parameters. The core architecture is the same, with the only difference being the embedding sizes and the number of times individual components like the attention heads and transformer blocks are repeated.

Total number of parameters:

- **124 M in "gpt2-small"**
- **355 M in "gpt2-medium"**
- **774 M in "gpt2-large"**
- **1558 M in "gpt2-xl"**



Repeat this transformer block:

`settings` `params` `GPTModel`

GPT_CONFIG_124M

```
model_name = "gpt2-small (124M)"
NEW_CONFIG = GPT_CONFIG_124M.copy()
NEW_CONFIG.update(model_configs[model_name])
```

NEW_CONFIG

```
NEW_CONFIG.update({"context_length": 1024})
```

```
            load_weights_into_
    params              GPTM              gpt
```

Listing 5.5 Loading OpenA[redacted] into our GPT [redacted]ode

```
import numpy as np[redacted]

def load_weights_into_gpt(gpt, pa[redacted]
    gpt.pos_emb.weight = assign(g[redacted]s_emb.weight, params['wpe']
    gpt.tok_emb.weight = assign(g[redacted]k_emb.weight, params['wte']

    for b in range(len(params["blo[redacted]])):  #B
        q_w, k_w, v_w = np.split(
            (params["blocks"][b]["[redacted]"]["c_attn"])["w"], 3, axis
        gpt.trf_blocks[b].att.W_qu[redacted]weight = assign(
            gpt.trf_blocks[b].att[redacted]ery.weight, q_w.T)
        gpt.trf_blocks[b].att.W_ke[redacted]ight = assign(
            gpt.trf_blocks[b].att[redacted]y.weight, k_w.T)
        gpt.trf_blocks[b].att.W_v[redacted]eight = assign(
            gpt.t[redacted]ue.weight, v_w.T)

        q_b, k_b, [redacted]
            (param[redacted]"][b][redacted]["c_attn"])[redacted]'], 3, axis
        gpt.trf_blocks[b].att.W_q[redacted]ias = assign(
            gpt.trf_blocks[b].att[redacted]ry.bias, q_b)
        gpt.trf_blocks[b].att.W_k[redacted]s = assign(
            gpt.trf_blocks[b].att[redacted].bias, k_b)
        gpt.trf_blocks[b].att.W_v[redacted]ias = assign(
            gpt.trf_blocks[b].att[redacted]ue.bias, v_b)

        gpt.trf_blocks[b].att.out_proj.weight = assign(
            gpt.trf_blocks[b].att.out_proj.weight,
            params["blocks"][b]["attn"]["c_proj"]["w"].T)
        gpt.trf_blocks[b].att.out_proj.bias = assign(              a
            gpt.trf_bl
```

```
        gpt.trf_blocks[b].ff.layers[2].bias = assign(
            gpt.trf_blocks[b].ff.layers[2].bias,
            params["blocks"][b]["mlp"]["c_proj"]["b"])

        gpt.trf_blocks[b].norm1.scale = assign(
            gpt.trf_blocks[b].norm1.scale,
            params["blocks"][b]["ln_1"]["g"])
        gpt.trf_blocks[b].norm1.shift = assign(
            gpt.trf_blocks[b].norm1.shift,
            params["blocks"][b]["ln_1"]["b"])
        gpt.trf_blocks[b].norm2.scale = assign(
            gpt.trf_blocks[b].norm2.scale,
            params["blocks"][b]["ln_2"]["g"])
        gpt.trf_blocks[b].norm2.shift = assign(
            gpt.trf_blocks[b].norm2.shift,
            params["blocks"][b]["ln_2"]["b"])

    gpt.final_norm.scale = assign(gpt.final_norm.scale, params["g
    gpt.final_norm.shift = assign(gpt.final_norm.shift, params["b
    gpt.out_head.weight = assign(gpt.out_head.weight, params["wte
```

load_weights_into_gpt

GPTModel

params["blocks"][0]["attn"]["c_proj"]["w"]

gpt.trf_blocks[b].att.out_proj.weight gpt GPTModel

load_weights_into_gpt

assign

load_weights_into_gpt

GPTModel gpt

```
load_weights_into_gpt(gpt, params)
gpt.to(device)
```

generate

```
torch.manual_seed(123)
token_ids = generate(
    model=gpt,
    idx=text_to_token_ids("Every effort moves you", tokenizer),
    max_new_tokens=25,
    context_size=NEW_CONFIG["context_length"],
    top_k=50,
    temperature=1.5
)
print("Output text:\n", token_ids_to_text(token_ids, tokenizer))
```

```
Output text:
 Every effort moves you toward finding an ideal new way to practi
What makes us want to be on top of that?
```

Exercise 5.5

Exercise 5.6

5.6 Summary

-

Appendix A. Introduction to PyTorch

This chapter covers

-
-
-
-
-

A.1 What is PyTorch

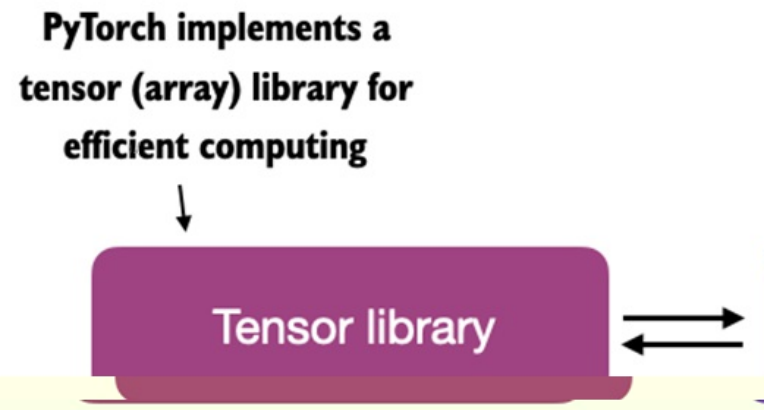
PyTorch _____

Papers With Code _____

Kaggle Data Science and Machine Learning Survey 2022 _____

A.1.1 The three core components of PyTorch

Figure A.1 PyTorch's three main components include a tensor library as a fundamental building block for computing, automatic differentiation for model optimization, and deep learning utility functions, making it easier to implement and train deep neural network models.



tensor library

automatic differentiation engine

deep learning library

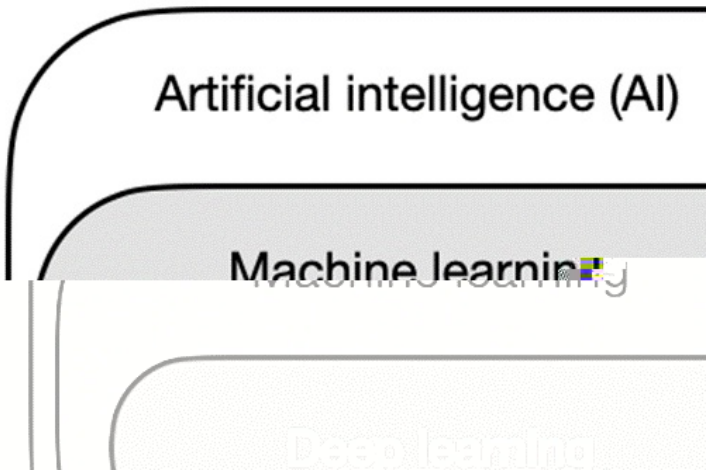
A.1.2 Defining deep learning

AI

1.1 What is an LLM?

Machine learnin

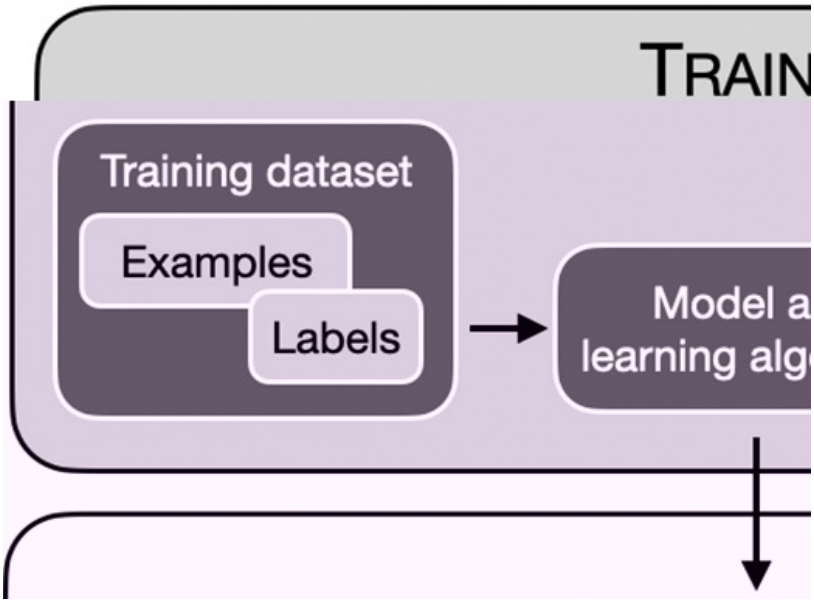
Figure A.2 Deep learning is a subcategory of machine learning that is focused on the implementation of deep neural networks. In turn, machine learning is a subcategory of AI that is concerned with algorithms that learn from data. AI is the broader concept of machines being able to perform tasks that typically require human intelligence.



Deep learning

supervised learning

Figure A.3 The supervised learning workflow for predictive modeling consists of a training stage where a model is trained on labeled examples in a training dataset. The trained model can then be used to predict the labels of new observations.



spam not-spam

spam not

spam

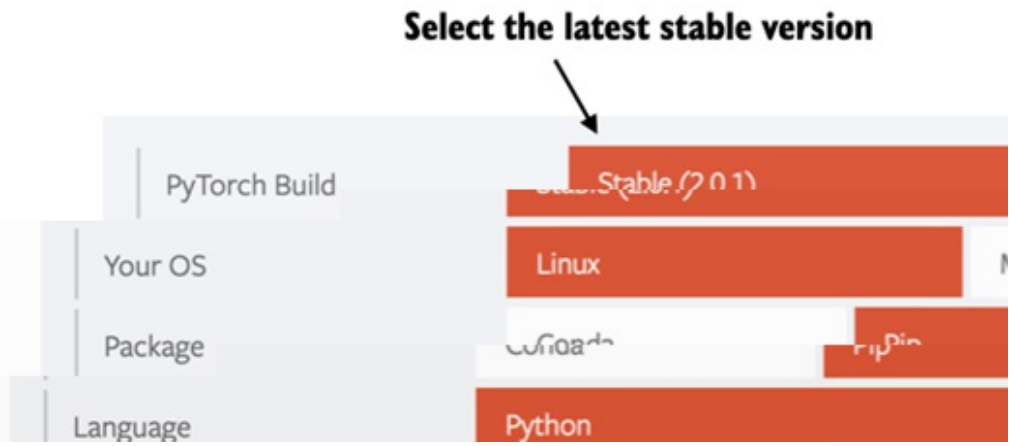
A.1.3 Installing PyTorch

Python version

```
pip install torch
```

AMD GPUs for deep learning

Figure A.4 Access the PyTorch installation recommendation on https://pytorch.org to customize and select the installation command for your system.




```
torchvision    torchaudio
```

```
pip install torch==2.0.1
```

```
torch      torch==2.0.1
```

```
import torch
torch.__version__
```

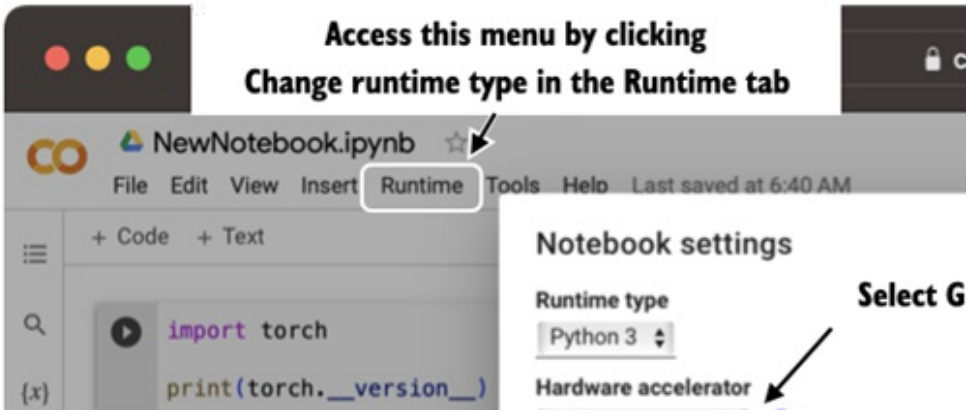
```
'2.0.1'
```

PyTorch and Torch

```
import torch
torch.cuda.is_available()
```

```
True
```

Figure A.5 Select a GPU device for Google Colab under the menu.



PyTorch on Apple Silicon

```
print(torch.backends.mps.is_available())
```

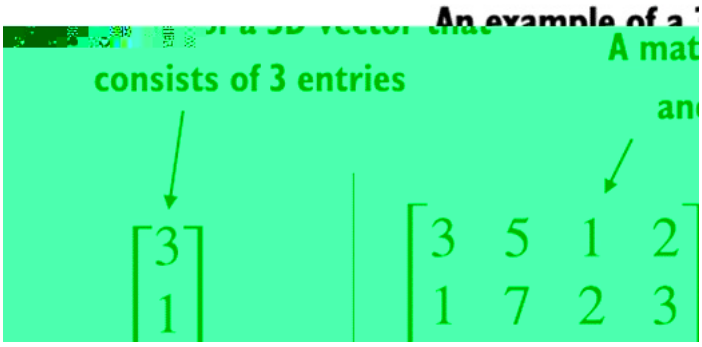
```
True
```

Exercise A.1

Exercise A.2

A.2 Understanding tensors

Figure A.6 An illustration of tensors with different ranks. Here 0D corresponds to rank 0, 1D to rank 1, and 2D to rank 2. Note that a 3D vector, which consists of 3 elements, is still a rank 1 tensor.



computing gradients

PyTorch's has a NumPy-like API

A.2.1 Scalars, vectors, matrices, and tensors

`Tensor` `torch.tensor`

Listing A.1 Creating PyTorch tensors

```
import torch

tensor0d = torch.tensor(1) #A

tensor1d = torch.tensor([1, 2, 3]) #B
```

```
tensor2d = torch.tensor([[1, 2], [3, 4]]) #C


tensor3d = torch.tensor([[[1, 2], [3, 4]], [[5, 6], [7, 8]]]) #D
```

A.2.2 Tensor data types

.dtype

```
tensor1d = torch.tensor([1, 2, 3])
print(tensor1d.dtype)
```

```
torch.int64
```

```
floatvec = torch.tensor([1.0, 2.0, 3.0])
print(floatvec.dtype)
```

```
torch.float32
```

.to

```
floatvec = tensor1d.to(torch.float32)
print(floatvec.dtype)
```

```
torch.float32
```

A.2.3 Common PyTorch tensor operations

`torch.tensor()`

```
tensor2d = torch.tensor([[1, 2, 3], [4, 5, 6]])
print(tensor2d)
```

```
tensor([[1, 2, 3],
        [4, 5, 6]])
```

`.shape`

```
print(tensor2d.shape)
```

```
torch.Size([2, 3])
```

`.shape` [2, 3]

.reshape

```
print(tensor2d.reshape(3, 2))
```

```
tensor([[1, 2],
        [3, 4],
        [5, 6]])
```

.view()

```
print(tensor2d.view(3, 2))
```

```
tensor([[1, 2],
        [3, 4],
        [5, 6]])
```

.reshape .view

.T

```
print(tensor2d.T)
```

```
tensor([[1, 4],
        [2, 5],
        [3, 6]])
```

.matmul

```
print(tensor2d.matmul(tensor2d.T))
```

```
tensor([[14, 32],
        [32, 77]])
```

```
                          @
```

```
print(tensor2d @ tensor2d.T)
```

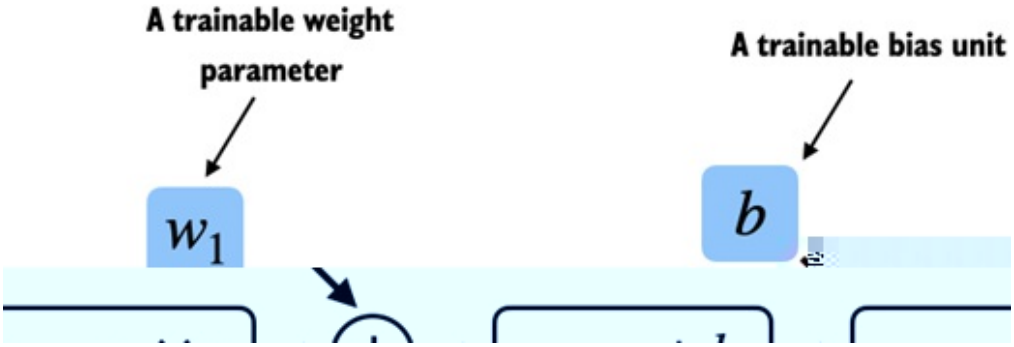
```
tensor([[14, 32],
        [32, 77]])
```

A.3 Seeing models as computation graphs

Listing A.2 A logistic regression forward pass

```
import torch.nn.functional as F   #A

y = torch.tensor([1.0])  #B
x1 = torch.tensor([1.1]) #C
w1 = torch.tensor([2.2]) #D
b = torch.tensor([0.0])  #E
z = x1 * w1 + b          #F
a = torch.sigmoid(z)     #G

loss = F.binary_cross_entropy(a, y)
```

Figure A.7 A logistic regression forward pass as a computation graph. The input feature $_1$ is multiplied by a model weight $_1$ and passed through an activation function after adding the bias. The loss is computed by comparing the model output with a given label .



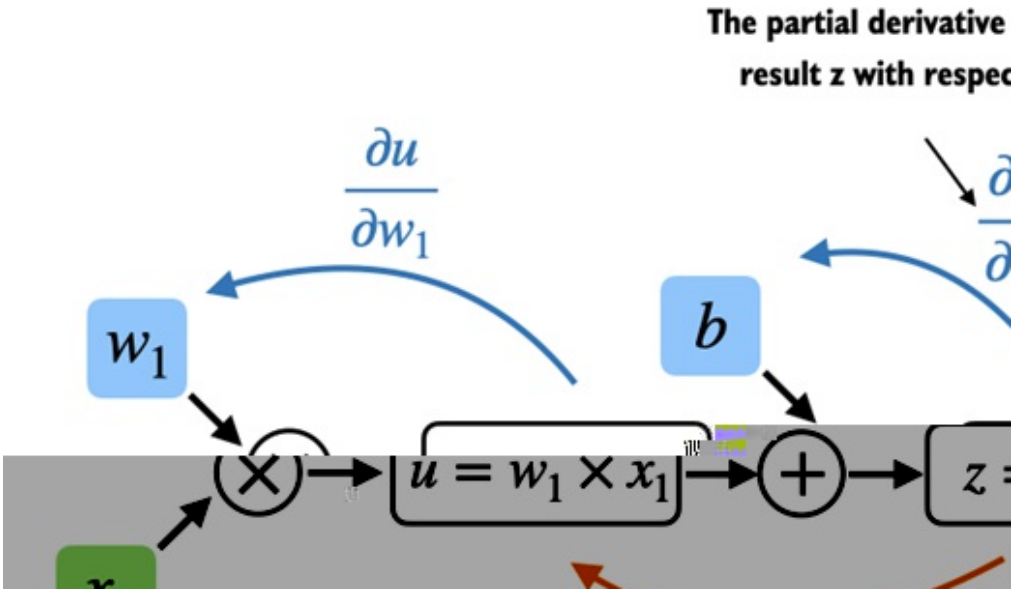
A.4 Automatic differentiation made easy

`requires_grad`

`True`

chain rule

Figure A.8 The most common way of computing the loss gradients in a computation graph involves applying the chain rule from right to left, which is also called reverse-model automatic differentiation or backpropagation. It means we start from the output layer (or the loss itself) and work backward through the network to the input layer. This is done to compute the gradient of the loss with respect to each parameter (weights and biases) in the network, which informs how we update these parameters during training.



Partial derivatives and gradients

A typical training loop

grad

w1

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)




print(grad_L_w1)
print(grad_L_b)
```

```
(tensor([-0.0898]),)
(tensor([-0.0817]),)
```

.backward

.grad

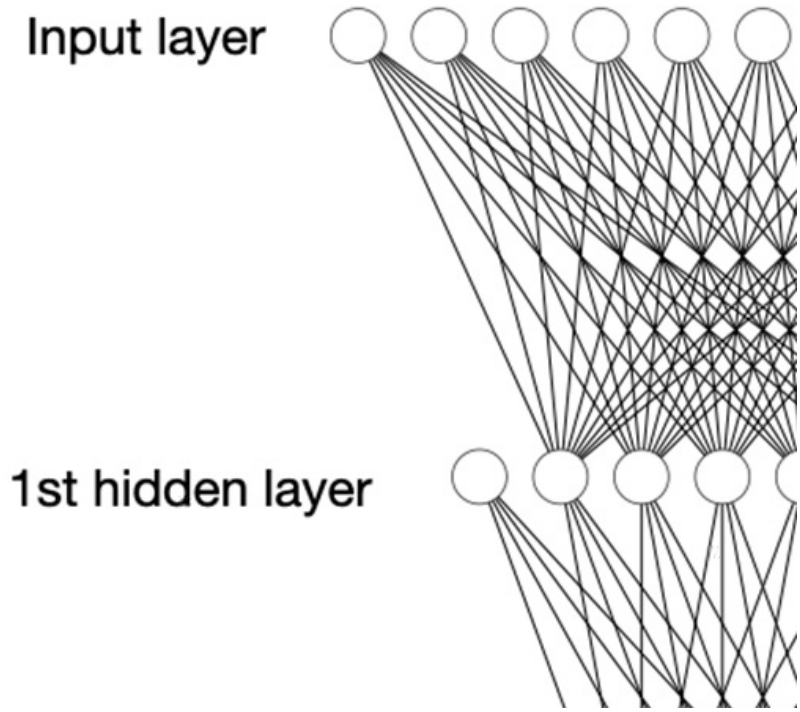
```
loss.backward()
print(w1.grad)
print(b.grad)
```

```
(tensor([-0.0898]),)
(tensor([-0.0817]),)
```

.backward

A.5 Implementing multilayer neural networks

Figure A.9 An illustration of a multilayer perceptron with 2 hidden layers. Each node represents a unit in the respective layer. Each layer has only a very small number of nodes for illustration purposes.



`torch.nn.Module`

```
class NeuralNetwork(torch.nn.Module):
    def __init__(self, num_inputs, num_outputs):  #A
        super().__init__()

        self.layers = torch.nn.Sequential(

```

NeuralNetwork

self.layers = Sequential(...) __init__ self.layers NeuralNetwork forward

```
num_params = sum(p.numel() for p in model.parameters() if p.requi
print("Total number of trainable model parameters:", num_params)
```

```
Total number of trainable model parameters: 2213
```

requires_]

```
       requires_grad=True)
```

```
.shape

print(model.layers[0].weight.shape)
```

```
torch.Size([30, 50])
```

```
                                        model.layers[0].bias
```

```
requires_grad       True
                                    torch.nn.Linear
```

```
manual_seed

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

```
Parameter containing:
tensor([[-0.0577,  0.0047, -0.0702,  ...,  0.0222,  0.1260,  0.08
        [ 0.0502,  0.0307,  0.0333,  ...,  0.0951,  0.1134, -0.02
        [ 0.1077, -0.1108,  0.0122,  ...,  0.0108, -0.1049, -0.10
```

```
        ...,
        [-0.0787,  0.1259,  0.0803,  ...,  0.1218,  0.1303, -0.13
        [ 0.1359,  0.0175, -0.0673,  ...,  0.0674,  0.0676,  0.10
        [ 0.0790,  0.1343, -0.0293,  ...,  0.0344, -0.0971, -0.05
       requires_grad=True)
```

NeuraNetwork

```
torch.manual_seed(123)
X = torch.rand((1, 50))
out = model(X)
print(out)
```

grad_fn=

```
<AddmmBackward0>)
```

X

model(x)

grad_fn

grad_fn=<AddmmBackward0>

grad_fn=

```
<AddmmBackward0>
```

```
<AddmmBackward0>
```

grad_fn=<AddmmBackward0>

Addmm Addmm

mm Add

torch.no_grad()

```
with torch.no_grad():
    out = model(X)
print(out)
```

```
tensor([[-0.1262,  0.1080, -0.1792]])
```

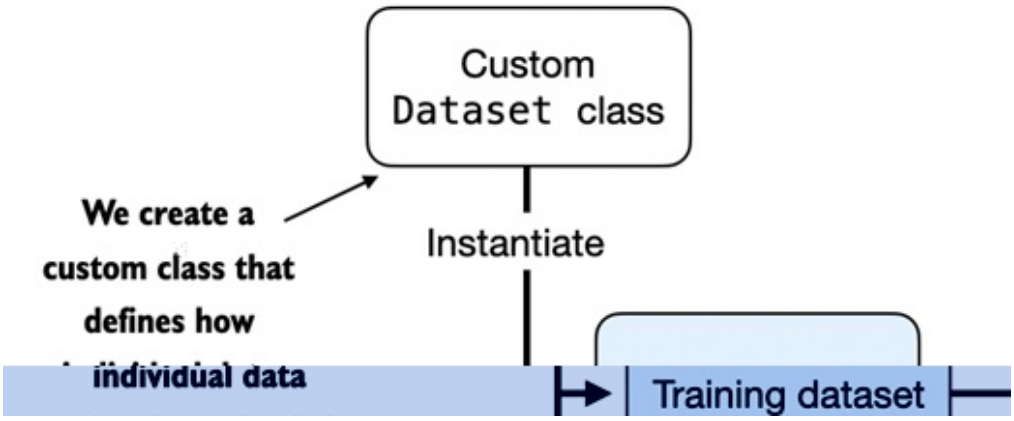
softmax

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

```
tensor([[0.3113, 0.3934, 0.2952]]))
```

A.6 Setting up efficient data loaders

Figure A.10 PyTorch implements a Dataset and a DataLoader class. The Dataset class is used to instantiate objects that define how each data record is loaded. The DataLoader handles how the data is shuffled and assembled into batches.



Listing A.5 Creating a small toy dataset

```
X_train = torch.tensor([
    [-1.2, 3.1],
    [-0.9, 2.9],
    [-0.5, 2.6],
    [2.3, -1.1],
```

```
    [2.7, -1.5]
])
y_train = torch.tensor([0, 0, 0, 1, 1])

X_test = torch.tensor([
    [-0.8, 2.8],
    [2.6, -1.6],
])
y_test = torch.tensor([0, 1])
```

Class label numbering

ToyDataset

Dataset

Listing A.6 Defining a custom Dataset class

```
from torch.utils.data import Dataset

class ToyDataset(Dataset):
    def __init__(self, X, y):
        self.features = X
        self.labels = y

    def __getitem__(self, index):     #A
        one_x = self.features[index]  #A
        one_y = self.labels[index]    #A
        return one_x, one_y           #A

    def __len__(self):
        return self.labels.shape[0]   #B

train_ds = ToyDataset(X_train, y_train)
test_ds = ToyDataset(X_test, y_test)
```

ToyDataset

DataLoader

__init__ __getitem__ __len__

__init__
__getitem__ __len__

X y

__getitem__
index

index

__len__
.shape

print(len(train_ds))

5

DataLoader

Listing A.7 Instantiating data loaders

```
from torch.utils.data import DataLoader


torch.manual_seed(123)

train_loader = DataLoader(
    dataset=train_ds,  #A
    batch_size=2,
```

```
    shuffle=True,  #B
    num_workers=0  #C
)

test_loader = DataLoader(
    dataset=test_ds,
    batch_size=2,
    shuffle=False,  #D
    num_workers=0
)
```

test_loader

```
for idx, (x, y) in enumerate(train_loader):
    print(f"Batch {idx+1}:", x, y)
```

```
Batch 1: tensor([[-1.2000,  3.1000],
                 [-0.5000,  2.6000]]) tensor([0, 0])
Batch 2: tensor([[ 2.3000, -1.1000],
                 [-0.9000,  2.9000]]) tensor([1, 0])
Batch 3: tensor([[ 2.7000, -1.5000]]) tensor([1])
```

train_loader

```
torch.manual_seed(123)
```

Listing A.8 A training loader that drops the last batch

```
train_loader = DataLoader(
    dataset=train_ds,
    batch_size=2,
    shuffle=True,
    num_workers=0,
    drop_last=True
)
```

```
for idx, (x, y) in enumerate(train_loader):
    print(f"Batch {idx+1}:", x, y)
```

```
Batch 1: tensor([[-0.9000,  2.9000],
        [ 2.3000, -1.1000]]) tensor([0, 1])
Batch 2: tensor([[ 2.7000, -1.5000],
        [-0.5000,  2.6000]]) tensor([1, 0])
```

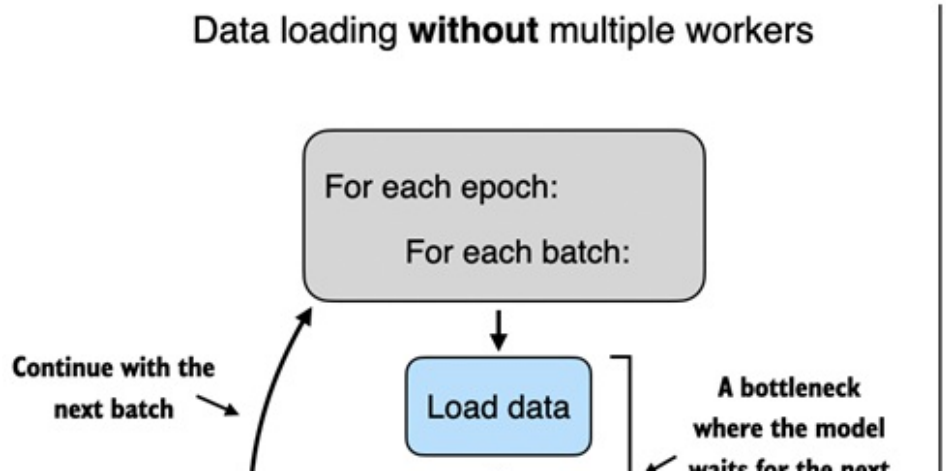
num_workers=0 DataLoader

DataLoader

num_workers

num_workers

Figure A.11 Loading data without multiple workers (setting `num_workers=0`) will create a data loading bottleneck where the model sits idle until the next batch is loaded as illustrated in the left subpanel. If multiple workers are enabled, the data loader can already queue up the next batch in the background as shown in the right subpanel.



`num_workers`

`num_workers`

`num_workers`

`num_workers`

`num_workers=4`

Module

Listing A.9 Neural network training in PyTorch

```
import torch.nn.functional as F


torch.manual_ eed(123)
model = Neura Network(num_inputs=2, num_outputs=2)  #A
optimizer = t rch.optim.SGD(model.parameters(), lr=0.5)  #B

num_epochs =

for epoch in  ange(num_epochs):

    model.tra n()
    for batch idx, (features, labels) in enumerate(train_loader):

        logit  = model(features)

        loss   F.cross_entropy(logits, labels)

        optim zer.zero_grad()   #C
        loss. ackward()         #D
        optim zer.step()        #E

        ### L GGING
        print f"Epoch: {epoch+1:03d}/{num_epochs:03d}"
              f" | Batch {batch_idx:03d}/{len(train_loader):03d}"
              f" | Train Loss: {loss:.2f}")

    model.eva ()
    # Optiona  model evaluation



Epoch: 001/00  | Batch 000/002 | Train Loss: 0.75
Epoch: 001/00  | Batch 001/002 | Train Loss: 0.65
Epoch: 002/00  | Batch 000/002 | Train Loss: 0.44
Epoch: 002/00  | Batch 001/002 | Trainl Loss: 0.13
Epoch: 003/00  | Batch 000/002 | Train Lon
```

`SGD`

`lr`

Exercise A.3

`model.train()` `model.eval()`

dropout *batch normalization*

```
                optimizer.step()
```

Preventing undesired gradient accumulation

```
                        optimizer.zero_grad()
```

```
model.eval()
with torch.no_grad():
    outputs = model(X_train)
print(outputs)
```

```
tensor([[ 2.8569, -4.1618],
        [ 2.5382, -3.7548],
        [ 2.0944, -3.1820],
        [-1.4814,  1.4816],
        [-1.7176,  1.7342]])
```

```
torch.set_printoptions(sci_mode=False)
probas = torch.softmax(outputs, dim=1)
print(probas)
```

```
tensor([[    0.9991,     0.0009],
        [    0.9982,     0.0018],
        [    0.9949,     0.0051],
        [    0.0491,     0.9509],
        [    0.0307,     0.9693]])
```

set_printoptions

dim=1 dim=0

```
predictions = torch.argmax(probas, dim=1)
print(predictions)
```

```
tensor([0, 0, 0, 1, 1])
```

argmax

```
predictions = torch.argmax(outputs, dim=1)
print(predictions)
```

```
tensor([0, 0, 0, 1, 1])
```

```
predictions == y_train
```

```
tensor([True, True, True, True, True])
```

torch.sum

```
torch.sum(predictions == y_train)
```

```
5
```

```
            compute_accuracy
```

Listing A.10 A function to compute the

compute_accuracy

```
print(compute_accuracy(model, train_loader))
```

```
1.0
```

```
>>> print(compute_accuracy(model, test_loader))
```

```
1.0
```

A.8 Saving and loading models

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

```
"model.pth"
```

.pth .pt

```
model = NeuralNetwork(2, 2)
```

```
model.load_state_dict(torch.load("model.pth"))

    torch.load("model.pth")                    "mo
```

```
print(torch.cuda.is_available())
```

```
True
```

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

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

.to()

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

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

```
device='cuda:0'
```

a

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

```
RuntimeError      Traceback (most recent call last)
<ipython-input-7-4ff3c4d20fc3> in <cell line: 2>()
      1 tensor_1 = tensor_1.to("cpu")
----> 2 print(tensor_1 + tensor_2)
RuntimeError: Expected all tensors to be on the same device, but
```

A.9.2 Single-GPU training

section 2.7, A typical training loop

Listing A.11 A training loop on a GPU

```
torch.manual_seed(123)
model = NeuralNetwork(num_inputs=2, num_outputs=2)

device = torch.device("cuda")   #A
model = model.to(device)   #B

optimizer = torch.optim.SGD(model.parameters(), lr=0.5)

num_epochs = 3

for epoch in range(num_epochs):

    model.train()
```

```
    for batch_idx, (features, labels) in enumerate(train_loader):

        features, labels = features.to(device), labels.to(device)
        logits = model(features)
        loss = F.cross_entropy(logits, labels) # Loss function

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        ### LOGGING
        print(f"Epoch: {epoch+1:03d}/{num_epochs:03d}"
              f" | Batch {batch_idx:03d}/{len(train_loader):03d}"
              f" | Train/Val Loss: {loss:.2f}")

    model.eval()
    # Optional model evaluation
```

```
Epoch: 001/003 | Batch 000/002 | Train/Val Loss: 0.75
Epoch: 001/003 | Batch 001/002 | Train/Val Loss: 0.65
Epoch: 002/003 | Batch 000/002 | Train/Val Loss: 0.44
Epoch: 002/003 | Batch 001/002 | Train/Val Loss: 0.13
Epoch: 003/003 | Batch 000/002 | Train/Val Loss: 0.03
Epoch: 003/003 | Batch 001/002 | Train/Val Loss: 0.00
```

```
              .to("cuda")         device = torch.device("cuda")
```

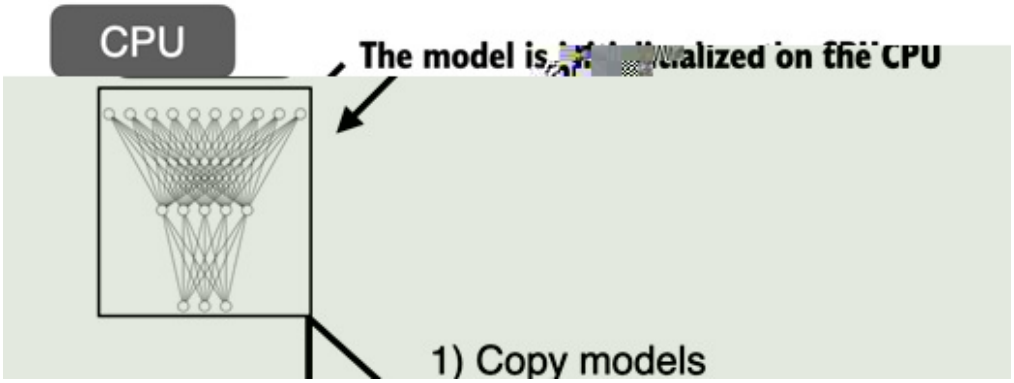
```
torch.device("cuda")
```

```
device = torch.device("cuda" if torch.cuda.is_available() else "c
```

Multi-GPU computing is optional

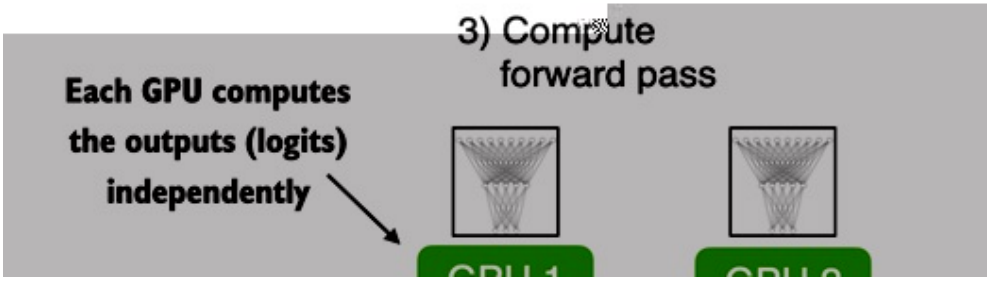
`DistributedDataParallel`

Figure A.12 The model and data transfer in DDP involves two key steps. First, we create a copy of the model on each of the GPUs. Then we divide the input data into unique minibatches that we pass on to each model copy.



`DistributedSampler`

Figure A.13 The forward and backward pass in DDP are executed independently on each GPU with its corresponding data subset. Once the forward and backward passes are completed, gradients from each model replica (on each GPU) are synchronized across all GPUs. This ensures that every model replica has the same updated weights.



Multi-GPU computing in interactive environments

Listing A.12 PyTorch utilities for distributed training

```
import torch.multiprocessing as mp
from torch.utils.data.distributed import DistributedSampler
from torch.nn.parallel import DistributedDataParallel as DDP
from torch.distributed import init_process_group, destroy_process
```

DistributedDataParallel

multiprocessing
multiprocessing.spawn

DistributedSampler

init_process_group destroy_process_group
init_process_group

destroy_process_group

NeuralNetwork

Listing A.13 Model training with DistributedDataParallel strategy

```
def ddp_setup(rank, world_size):
    os.environ["MASTER_ADDR"] = "localhost"   #A
    os.environ["MASTER_PORT"] = "12345"       #B
    init_process_group(
        backend="nccl",                       #C
        rank=rank,                            #D
        world_size=world_size                 #E
    )
    torch.cuda.set_device(rank)               #F
def prepare_dataset():
    ...
    train_loader = DataLoader(
        dataset=train_ds,
        batch_size=2,
        shuffle=False,                        #G
        pin_memory=True,                      #H
        drop_last=True,
        sampler=DistributedSampler(train_ds)  #I
    )
    return train_loader, test_loader
def main(rank, world_size, num_epochs):       #J
    ddp_setup(rank, world_size)
    train_loader, test_loader = prepare_dataset()
    model = NeuralNetwork(num_inputs=2, num_outputs=2)
    model.to(rank)
    optimizer = torch.optim.SGD(model.parameters(), lr=0.5)
    model = DDP(model, device_ids=[rank])
    for epoch in range(num_epochs):
    for features, labels in train_loader:
            features, labels = features.to(rank), labels.to(rank)
            ...
            print(f"[GPU{rank}] Epoch: {epoch+1:03d}/{num_epochs:
                  f" | Batchsize {labels.shape[0]:03d}"
                  f" | Train/Val Loss: {loss:.2f}")
    model.eval()
    train_acc = compute_accuracy(model, train_loader, device=rank
    print(f"[GPU{rank}] Training accuracy", train_acc)
    test_acc = compute_accuracy(model, test_loader, device=rank)
    print(f"[GPU{rank}] Test accuracy", test_acc)
    destroy_process_group()                    #L

if __name__ == "__main__":
    print("Number of GPUs available:", torch.cuda.device_count())
    torch.manual_seed(123)
    num_epochs = 3
    world_size = torch.cuda.device_count()
    mp.spawn(main, args=(world_size, num_epochs), nprocs=world_si
```

__name__ == "__main__"

torch.cuda.device_count()

multiprocesses.spawn spawn
nproces=world_size
spawn
main
args main rank
mp.spawn()
rank

main ddp_setup

to(rank)
DDP

destroy_process_group()

sampler=DistributedSampler(train_ds)

ddp_setup

rank

Selecting available GPUs on a multi-GPU machine

CUDA_VISIBLE_DEVICE

```
python some_script.py
```

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

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

CUDA VISIBLE_DEVICES

```
python ch02-DDP-script.py
```

```
PyTorch version: 2.0.1+cu117
CUDA available: True
Number of GPUs available: 1
[GPU0] Epoch: 001/003 | Batchsize 002 | Train/Val Loss: 0.62
[GPU0] Epoch: 001/003 | Batchsize 002 | Train/Val Loss: 0.32
[GPU0] Epoch: 002/003 | Batchsize 002 | Train/Val Loss: 0.11
[GPU0] Epoch: 002/003 | Batchsize 002 | Train/Val Loss: 0.07
[GPU0] Epoch: 003/003 | Batchsize 002 | Train/Val Loss: 0.02
[GPU0] Epoch: 003/003 | Batchsize 002 | Train/Val Loss: 0.03
[GPU0] Training accuracy 1.0
[GPU0] Test accuracy 1.0
```

```
PyTorch version: 2.0.1+cu117
CUDA available: True
Number of GPUs available: 2
[GPU1] Epoch: 001/003 | Batchsize 002 | Train/Val Loss: 0.60
[GPU0] Epoch: 001/003 | Batchsize 002 | Train/Val Loss: 0.59
[GPU0] Epoch: 002/003 | Batchsize 002 | Train/Val Loss: 0.16
[GPU1] Epoch: 002/003 | Batchsize 002 | Train/Val Loss: 0.17
[GPU0] Epoch: 003/003 | Batchsize 002 | Train/Val Loss: 0.05
[GPU1] Epoch: 003/003 | Batchsize 002 | Train/Val Loss: 0.05
[GPU1] Training accuracy 1.0
[GPU0] Training accuracy 1.0
[GPU1] Test accuracy 1.0
[GPU0] Test accuracy 1.0
```

```
 GPU0                        GPU1
```

Alternative PyTorch APIs for multi-GPU training

A.10 Summary

-
-
-
-
-
-
-
- `Dataset` `DataLoader`
-
- `DistributedDataParallel`

A.11 Further reading

- *Machine Learning with PyTorch and Scikit-Learn*
- *Deep Learning with PyTorch*

-

- *Model Evaluation, Model Selection, and Algorithm Selection in Machine Learning*

- *Introduction to Calculus*

```
optimizer.zero_grad()
```

- *Finetuning Large Language Models On A Single GPU Using Gradient Accumulation*

Fully Sharded Data Parallel

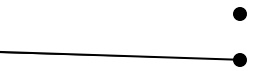
-

A.12 Exercise answers

Exercise A.3:

```
model = NeuralNetwork(2, 2)
num_params = sum(p.numel() for p in model.parameters() if p.requi
print("Total number of trainable model parameters:", num_params)
```

```
752
```



```
63.8 µs ± 8.7 µs per loop
```

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

```
13.8 µs ± 425 ns per loop
```

Appendix B. References and Further Reading

B.1 Chapter 1

- *BloombergGPT: A Large Language Model for Finance* *et al.*
- *Towards Expert-Level Medical Question Answering with Large Language Models* *et al.*
- *Attention Is All You Need* *et al.*
- *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding* *et al.*
- *Language Models are Few-Shot Learners* *et al.*

- *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale* *et al.*
- *RWKV: Reinventing RNNs for the Transformer Era* *et al.*
- *Hyena Hierarchy: Towards Larger Convolutional Language Models (2023)* *et al.,*
- *Mamba: Linear-Time Sequence Modeling with Selective State Spaces*
- *Llama 2: Open Foundation and Fine-Tuned Chat Models* *et al.*

The Pile

- *The Pile: An 800GB Dataset of Diverse Text for Language Modeling* *et al.*
- *Training Language Models to Follow Instructions with Human Feedback* *Ouyang et al.*

B.2 Chapter 2

- *Machine Learning Q and AI*

-

-

-

  `minbpe`

-

B.3 Chapter 3

- *Neural Machine Translation by Jointly Learning to Align and Translate*

- *Attention Is All You Need*

FlashAttentio

- *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*
  et al.

- *Simplifying Transformer Blocks*

B.4 Chapter 4

- *Layer Normalization*
  _____

- *On Layer Normalization in the Transformer Architecture*
  et al. _____
- *ResiDual: Transformer with Dual Residual Connections*
  et al. _____

- *Gaussian Error Linear Units (GELUs)*

- *Language Models are Unsupervised Multitask Learners*
  et al.

-

B.5 Chapter 5

-

-

- *et al.*

Appendix C. Exercise Solutions

C.1 Chapter 2

Exercise 2.1

```
print(tokenizer.encode("Ak"))
print(tokenizer.encode("w"))
# ...
```

```
[33901]
[86]
# ...
```

```
print(tokenizer.decode([33901, 86, 343, 86, 220, 959]))
```

```
'Akwirw ier'
```

Exercise 2.2

```
                              max_length=2 and stride=2

dataloader = create_dataloader(raw_text, batch_size=4, max_length
```

```
tensor([[  40,  367],
        [2885, 1464],
        [1807, 3619],
        [ 402,  271]])
```

```
                                    max_length=8 and stride=2
```

```
dataloader = create_dataloader(raw_text, batch_size=4, max_length
```

```
tensor([[   40,   367,  2885,  1464,  1807,  3619,   402,   271],
        [ 2885,  1464,  1807,  3619,   402,   271, 10899,  2138],
        [ 1807,  3619,   402,   271, 10899,  2138,   257,  7026],
        [  402,   271, 10899,  2138,   257,  7026, 15632,   438]]
```

C.2 Chapter 3

Exercise 3.1

```
sa_v1.W_query = torch.nn.Parameter(sa_v2.W_query.weight.T)
sa_v1.W_key = torch.nn.Parameter(sa_v2.W_key.weight.T)
sa_v1.W_value = torch.nn.Parameter(sa_v2.W_value.weight.T)
```

Exercise 3.2

```
                                          d_out

d_out = 1
mha = MultiHeadAttentionWrapper(d_in, d_out, block_size, 0.0, num
```

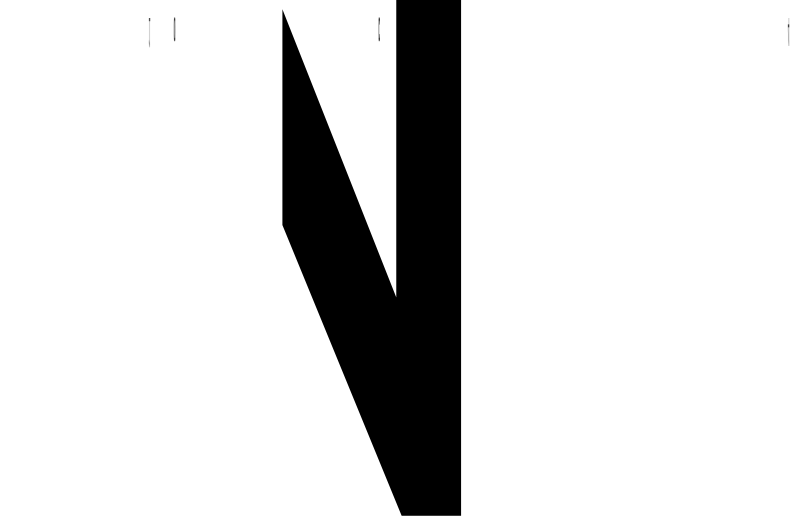
Exercise 3.3

```
block_size = 1024
d_in, d_out = 768, 768
num_heads = 12
mha = MultiHeadAttention(d_in, d_out, block_size, 0.0, num_heads)
```

C.3 Chapter 4

Exercise 4.1

```
block = TransformerBl   k(GPT_CONFIG_124M)

total_p
```



C.4 Chapter 5

Exercise 5.1

```
print_sampled_tokens
```

```
scaled_probas[2][6]
```

Exercise 5.2

Exercise 5.3

generate

top_k=None

top_k=1

Exercise 5.4

```
checkpoint = torch.load("model_and_optimizer.pth")
model = GPTModel(GPT_CONFIG_124M)
model.load_state_dict(checkpoint["model_state_dict"])
optimizer = torch.optim.AdamW(model.parameters(), lr=5e-4, weight
optimizer.load_state_dict(checkpoint["optimizer_state_dict"])
```

train_simple_function num_epochs=1

Exercise 5.5

```
train_loss = calc_loss_loader(train_loader, gpt, device)
val_loss = calc_loss_loader(val_loader, gpt, device)
```

```
Training loss: 3.754748503367106
Validation loss: 3.559617757797241
```

Exercise 5.6

```
hparams, params = download_and_load_gpt2(model_size="124M", model
model_name = "gpt2-small (124M)"
```

```
hparams, params = download_and_load_gpt2(model_size="1558M", mode
model_name = "gpt2-xl (1558M)"
```

Appendix D. Adding Bells and Whistles to the Training Loop

learning rate warmup *cosine decay* *gradient clipping*

```
if not os.path.exists(file_path):
    with urllib.request.urlopen(url) as response:
        text_data = response.read().decode('utf-8')
    with open(file_path, "w", encoding="utf-8") as file:
        file.write(text_data)
else:
    with open(file_path, "r", encoding="utf-8") as file:
        text_data = file.read()
Next, we load the text_data into the data loaders:
from previous_chapters import create_dataloader_v1

train_ratio = 0.90
split_idx = int(train_ratio * len(text_data))
torch.manual_seed(123)
train_loader = create_dataloader_v1(
    text_data[:split_idx],
    batch_size=2,
    max_length=GPT_CONFIG_124M["ctx_len"],
    stride=GPT_CONFIG_124M["ctx_len"],
    drop_last=True,
    shuffle=True
)
val_loader = create_dataloader_v1(
    text_data[split_idx:],
    batch_size=2,
    max_length=GPT_CONFIG_124M["ctx_len"],
    stride=GPT_CONFIG_124M["ctx_len"],
    drop_last=False,
    shuffle=False
)
```

D.1 Learning rate warmup

learning rate warmup

```
                  initial_lr
 peak_lr
```

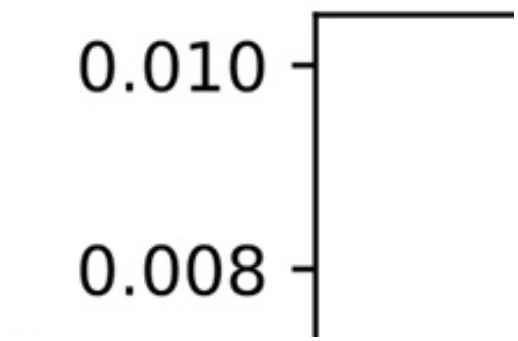
```
n_epochs = 15
initial_lr = 0.0001
peak_lr = 0.01
warmup_steps = 20
```

```
optimizer = torch.optim.AdamW(model.parameters(), weight_decay=0.
lr_increment = (peak_lr - initial_lr) / warmup_steps #A

global_step = -1
track_lrs = []

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

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

        for param_group in optimizer.param_groups: #D
            param_group["lr"] = lr
        track_lrs.append(optimizer.param_groups[0]["lr"])
        #E
```

```
import matplotlib.pyplot as plt
plt.ylabel("Learning rate")
plt.xlabel("Step")
```

```
total_training_steps = len(train_loader) * n_epochs
plt.plot(range(total_training_steps), track_lrs);
plt.show()
```

Figure D.1 The learning rate warmup increases the learning rate for the first 20 training steps. After 20 steps, the learning rate reaches the peak of 0.01 and remains constant for the rest of the training.

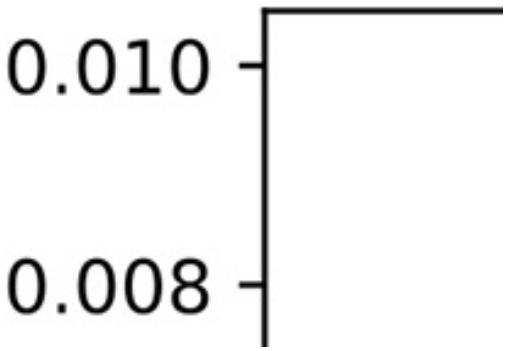


D.2 Cosine decay

cosine decay

```
import math

min_lr = 0.1 * initial_lr
track_lrs = []
lr_increment = (peak_lr - initial_lr) / warmup_steps
global_step = -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:# #B
            progress = ((global_step - warmup_steps) /
                        (total_training_steps - warmup_steps))
            lr = min_lr + (peak_lr - min_lr) * 0.5 * (1 + math.co

        for param_group in optimizer.param_groups:
            param_group["lr"] = lr
        track_lrs.append(optimizer.param_groups[0]["lr"])




plt.ylabel("Learning rate")
plt.xlabel("Step")
plt.plot(range(total_training_steps), track_lrs)
plt.show()
```

Figure D.2 The first 20 steps of linear learning rate warmup are followed by a cosine decay, which reduces the learning rate in a half-cosine cycle until it reaches its minimum point at the end of training.



D.3 Gradient clipping

gradient

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

$$|G|_2 = \sqrt{1^2 + 2^2 + 2^2 + 4^2} = \sqrt{25} = 5$$

G max_norm

max_norm **G**

G'

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

```
from previous_chapters import calc_loss_batch
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
loss = calc_loss_batch(input_batch, target_batch, model, device)
loss.backward()
```

.backward()

.grad

```
            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))
```

```
tensor(0.0373)
```

```
torch.nn.utils.clip_grad_norm_(model                ), max_norm=1.0)
print(find_highest_gradient(model))
```

```
tensor(0.0166)
```

D.4 The modified training fu

```
                                                    in_model_simple
```

```
train_model_simple

from previous_chapters import evalua                nerate_and_
```

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

peak_lr = optimizer.param_groups[0]["lr"] #A
total_training_steps = len(train_loader) * n_epochs #B
lr_increment = (peak_lr - initial_lr) / warmup_steps #C

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

        if global_step < warmup_steps: #D
            lr = initial_lr + global_step * lr_increment
        else:
            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))

        for param_group in optimizer.param_groups: #E
            param_group["lr"] = lr
        track_lrs.append(lr)
        loss = calc_loss_batch(input_batch, target_batch, mod
        loss.backward()

        if global_step > warmup_steps: #F
            torch.nn.utils.clip_grad_norm_(model.parameters()
        #G
        optimizer.step()
        tokens_seen += input_batch.numel()

        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} (Iter {global_step:06d}): "
                  f"Train loss {train_loss:.3f}, Val loss {va
```

```
        )

    return train_losses, val_losses, track_tokens_seen, track_lrs
```

train_model

train_model_simple

```
torch.manual_seed(123)
model = GPTModel(GPT_CONFIG_124M)
model.to(device)
peak_lr = 5e-4
optimizer = torch.optim.AdamW(model.parameters(), weight_decay=0.

n_epochs = 15
train_losses, val_losses, tokens_seen, lrs = train_model(
    model, train_loader, val_loader, optimizer, device, n_epochs=
    eval_freq=5, eval_iter=1, start_context="Every effort moves y
    warmup_steps=10, initial_lr=1e-5, min_lr=1e-5
)
```

```
Ep 1 (Iter 000000): Train loss 10.934, Val loss 10.939
Ep 1 (Iter 000005): Train loss 8.529, Val loss 8.843
Every effort moves you,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
Ep 2 (Iter 000010): Train loss 6.400, Val loss 6.825
Ep 2 (Iter 000015): Train loss 6.116, Val loss 6.861
Every effort moves you,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
...
the irony. She wanted him vindicated--and by me!"  He laughed aga
Ep 15 (Iter 000130): Train loss 0.101, Val loss 6.707
Every effort moves you?"  "Yes--quite insensible to the irony. Sh
```

train_model_simple