Lecture 5: Embeddings and data sampling

CSC 352 / CSC 552 · 2026-09-10

Open in Colab → **File → Save a copy in Drive**. You cannot edit the class original.

Start-of-class quiz

Type answers in the Canvas attendance quiz (empty boxes only). The questions stay here.

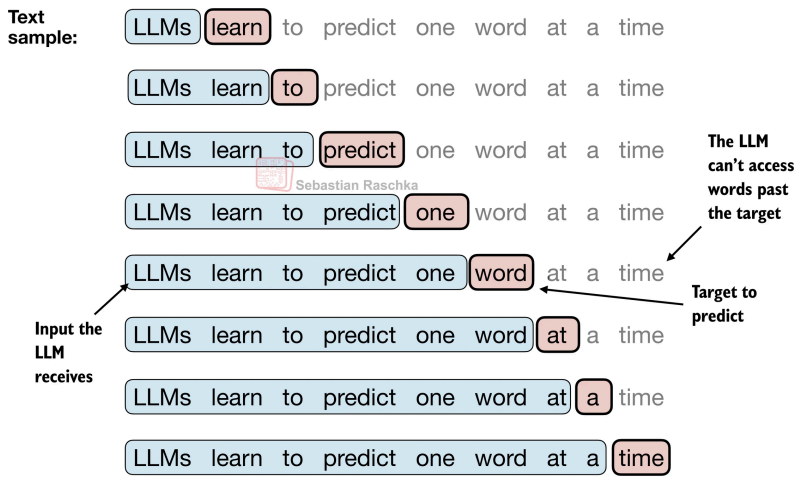
1. What does tokenize mean, in one sentence?
2. Which library did we use for GPT-2 BPE?

Continued from last class

We have token ids from BPE. Today we cut them into training windows, then look up vectors.

2.6 Data sampling with a sliding window

- We train LLMs to generate one word at a time, so we want to prepare the training data accordingly where the next word in a sequence represents the target to predict:



In [109…

```
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))
```

```
5145
```

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

In [111…
```
enc_sample = enc_text[50:]
```

`y` is `x` shifted by one: the next-token target.

In [113…
```
context_size = 4

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]
```

- One by one, the prediction would look like as follows:

In [115…
```
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
```

The next cell runs `for i in range(1, context_size+1): context = enc_sample[:i] desired = en`.

In [117…
```
for i in range(1, context_size+1):
    context = enc_sample[:i]
    desired = enc_sample[i]

    print(tokenizer.decode(context), "---->", tokenizer.decode([desired]))
```

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

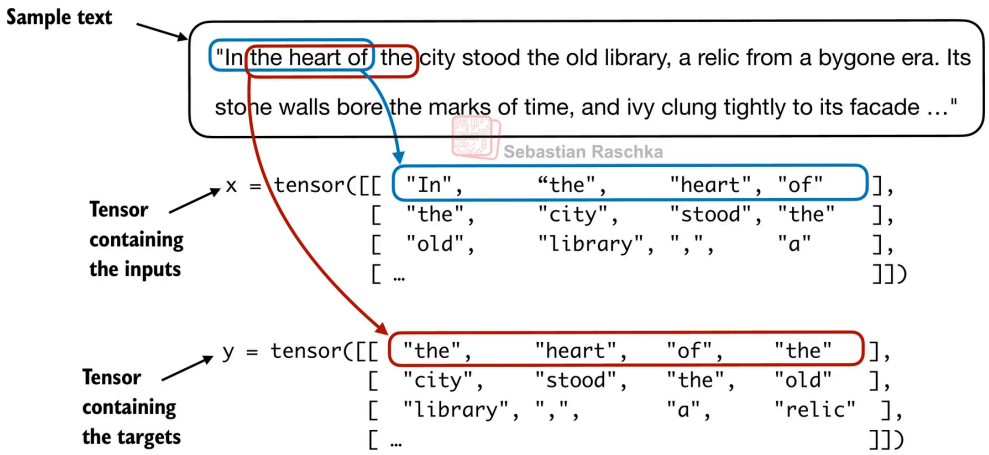
- We will take care of the next-word prediction in a later chapter after we covered the attention mechanism
- For now, we implement a simple data loader that iterates over the input dataset and returns the inputs and targets shifted by one

- Install and import PyTorch (see Appendix A for installation tips)

In [120…
```
import torch
print("PyTorch version:", torch.__version__)
```

```
PyTorch version: 2.8.0
```

- We use a sliding window approach, changing the position by +1:



- Create dataset and dataloader that extract chunks from the input text dataset

In [123…
```
from torch.utils.data import Dataset, DataLoader


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

        # Tokenize the entire text
        token_ids = tokenizer.encode(txt, allowed_special={"<|endoftext|>"})
        assert len(token_ids) > max_length, "Number of tokenized inputs must

        # Use a sliding window to chunk the book into overlapping sequences
        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):
```

```
        return len(self.input_ids)

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

A helper that wraps `GPTDatasetV1` in a `DataLoader` (same idea as Lecture 2).

In [125…
```
def create_dataloader_v1(txt, batch_size=4, max_length=256,
                         stride=128, shuffle=True, drop_last=True,
                         num_workers=0):

    # Initialize the tokenizer
    tokenizer = tiktoken.get_encoding("gpt2")

    # Create dataset
    dataset = GPTDatasetV1(txt, tokenizer, max_length, stride)

    # Create dataloader
    dataloader = DataLoader(
        dataset,
        batch_size=batch_size,
        shuffle=shuffle,
        drop_last=drop_last,
        num_workers=num_workers
    )

    return dataloader
```

- Let's test the dataloader with a batch size of 1 for an LLM with a context size of 4:

In [127…
```
with open("the-verdict.txt", "r", encoding="utf-8") as f:
    raw_text = f.read()
```

One batch: `input` and `target` tensors. Read `.shape`.

In [129…
```
dataloader = create_dataloader_v1(
    raw_text, batch_size=1, max_length=4, stride=1, shuffle=False
)

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

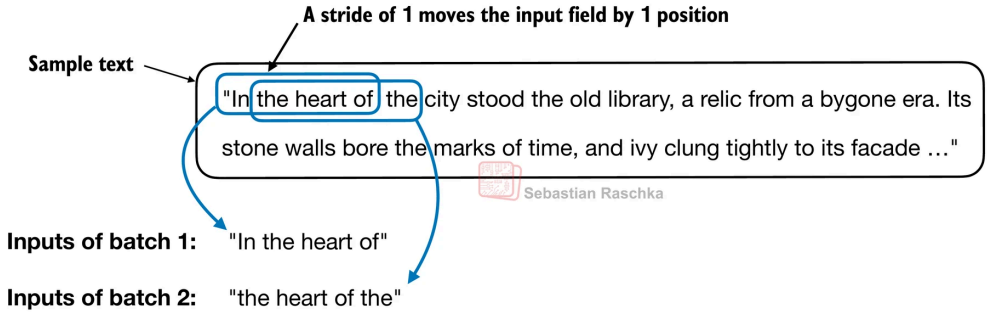
```
[tensor([[  40,  367, 2885, 1464]]), tensor([[ 367, 2885, 1464, 1807]])]
```

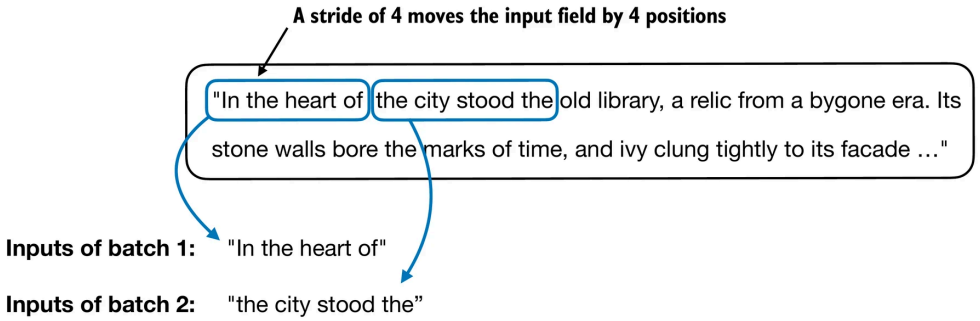
The next window. With `stride=1` it overlaps the first.

In [131…
```
second_batch = next(data_iter)
print(second_batch)
```

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

- An example using stride equal to the context length (here: 4) as shown below:





- We can also create batched outputs
- Note that we increase the stride here so that we don't have overlaps between the batches, since more overlap could lead to increased overfitting

In [135…
```
dataloader = create_dataloader_v1(raw_text, batch_size=8, max_length=4, stri

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]])
```

Exercise (2.6 Sliding window)

From `enc_sample`, take a 3-token input window and the matching next-token targets. Print both. Why is the target window shifted one step to the right?

In [ ]:
```python
# Exercise 2.6 — your attempt
context_size_ex = 3
x_ex = None
y_ex = None
print('x', x_ex)
print('y', y_ex)
```

Solution (2.6). The target is the next token at every position, so `y` is `x` shifted by one.

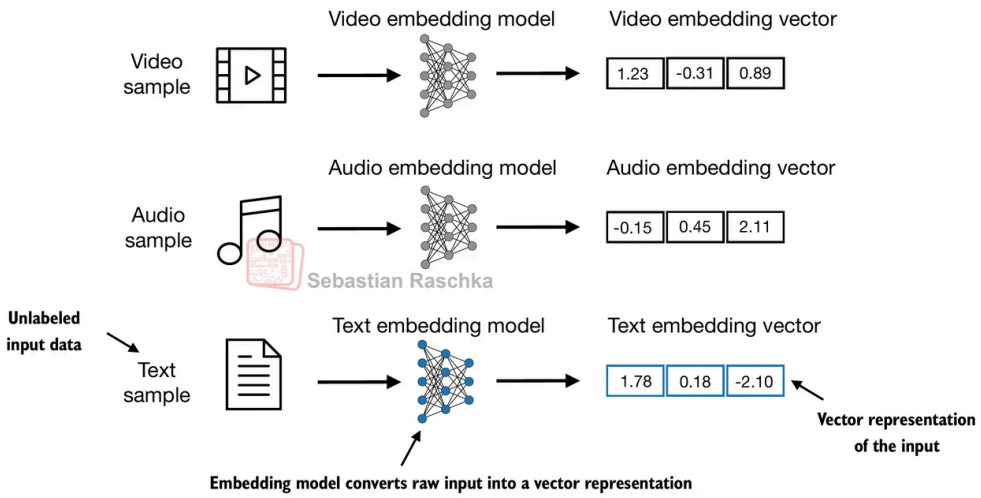
In [139…
```python
# Solution 2.6
context_size_ex = 3
x_ex = enc_sample[:context_size_ex]
y_ex = enc_sample[1:context_size_ex + 1]
print('x', x_ex)
print('y', y_ex)
```

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

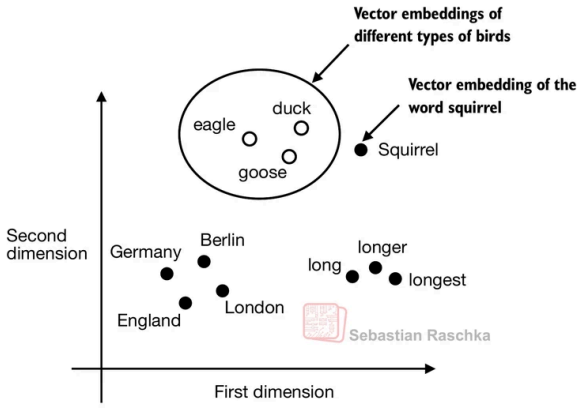
2.1 Understanding word embeddings

- No code in this section

- There are many forms of embeddings; we focus on text embeddings in this book

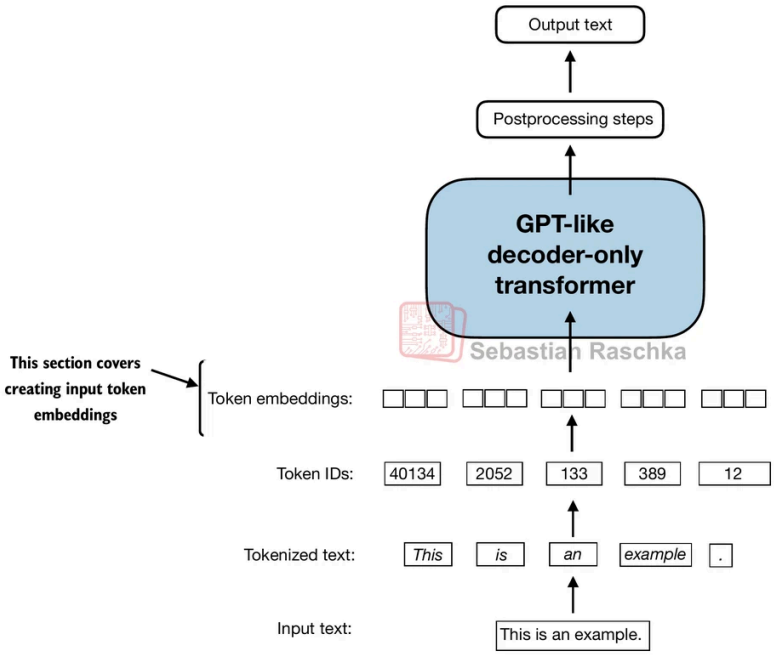


- LLMs work with embeddings in high-dimensional spaces (i.e., thousands of dimensions)
- Since we can't visualize such high-dimensional spaces (we humans think in 1, 2, or 3 dimensions), the figure below illustrates a 2-dimensional embedding space



2.7 Creating token embeddings

- The data is already almost ready for an LLM
- But lastly let us embed the tokens in a continuous vector representation using an embedding layer
- Usually, these embedding layers are part of the LLM itself and are updated (trained) during model training



- Suppose we have the following four input examples with input ids 2, 3, 5, and 1 (after tokenization):

In [144…
```
input_ids = torch.tensor([2, 3, 5, 1])
```

- For the sake of simplicity, suppose we have a small vocabulary of only 6 words and we want to create embeddings of size 3:

In [146…
```
vocab_size = 6
output_dim = 3

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

- This would result in a 6x3 weight matrix:

In [148…
```
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)
```

- For those who are familiar with one-hot encoding, the embedding layer approach above is essentially just a more efficient way of implementing one-hot encoding

followed by matrix multiplication in a fully-connected layer, which is described in the supplementary code in ./embedding_vs_matmul

- Because the embedding layer is just a more efficient implementation that is equivalent to the one-hot encoding and matrix-multiplication approach it can be seen as a neural network layer that can be optimized via backpropagation

- To convert a token with id 3 into a 3-dimensional vector, we do the following:

In [151…
```
print(embedding_layer(torch.tensor([3])))
```

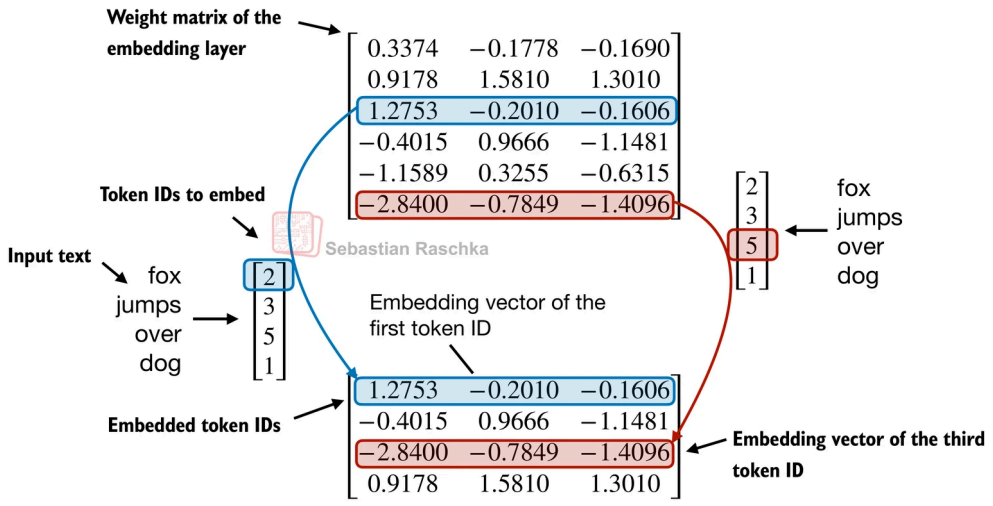
```
tensor([[-0.4015,  0.9666, -1.1481]], grad_fn=<EmbeddingBackward0>)
```

- Note that the above is the 4th row in the `embedding_layer` weight matrix
- To embed all four `input_ids` values above, we do

In [153…
```
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>)
```

- An embedding layer is essentially a look-up operation:



- **You may be interested in the bonus content comparing embedding layers with regular linear layers: ../03_bonus_embedding-vs-matmul**

Exercise (2.7 Token embeddings)

Look up token id 1 two ways: through the embedding layer, and as a row of its weight table. Print both. Are they the same vector?

In [ ]:
```
# Exercise 2.7 — your attempt
# embedding_layer is already built.
via_layer = None
via_weight = None
print(via_layer)
print(via_weight)
```

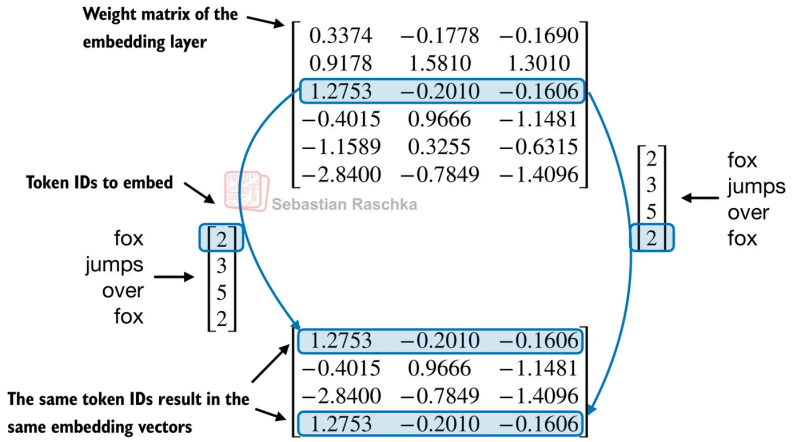
Solution (2.7). An embedding lookup **is** reading that row of the table (id 1 → row 1).

In [160…
```
# Solution 2.7
print(embedding_layer(torch.tensor([1])))
print(embedding_layer.weight[1])
```

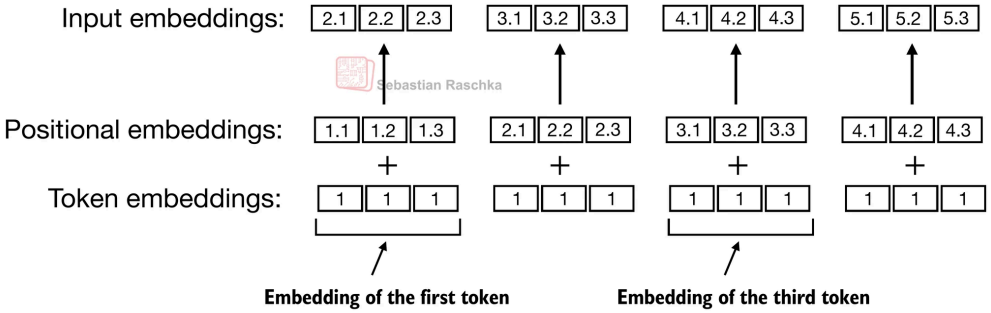
```
tensor([[0.9178, 1.5810, 1.3010]], grad_fn=<EmbeddingBackward0>)
tensor([0.9178, 1.5810, 1.3010], grad_fn=<SelectBackward0>)
```

2.8 Encoding word positions

- Embedding layer convert IDs into identical vector representations regardless of where they are located in the input sequence:



- Positional embeddings are combined with the token embedding vector to form the input embeddings for a large language model:



- The BytePair encoder has a vocabulary size of 50,257:
- Suppose we want to encode the input tokens into a 256-dimensional vector representation:

In [167…
```
vocab_size = 50257
output_dim = 256

token_embedding_layer = torch.nn.Embedding(vocab_size, output_dim)
```

- If we sample data from the dataloader, we embed the tokens in each batch into a 256-dimensional vector
- If we have a batch size of 8 with 4 tokens each, this results in a 8 x 4 x 256 tensor:

In [169…
```
max_length = 4
dataloader = create_dataloader_v1(
    raw_text, batch_size=8, max_length=max_length,
    stride=max_length, shuffle=False
)
data_iter = iter(dataloader)
inputs, targets = next(data_iter)
```

Print the token-id batch and its shape `(batch, tokens)`.

In [171…
```
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])
```

Look up a vector for each id. Shape should be `(batch, tokens, emb_dim)`.

In [173…
```
token_embeddings = token_embedding_layer(inputs)
print(token_embeddings.shape)

# uncomment & execute the following line to see how the embeddings look like
# print(token_embeddings)
```

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

- GPT-2 uses absolute position embeddings, so we just create another embedding layer:

In [175…
```
context_length = max_length
pos_embedding_layer = torch.nn.Embedding(context_length, output_dim)

# uncomment & execute the following line to see how the embedding layer weig
# print(pos_embedding_layer.weight)
```

One position vector per slot in the window, then we add it to the token vectors.

In [177…
```
pos_embeddings = pos_embedding_layer(torch.arange(max_length))
print(pos_embeddings.shape)

# uncomment & execute the following line to see how the embeddings look like
# print(pos_embeddings)
```

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

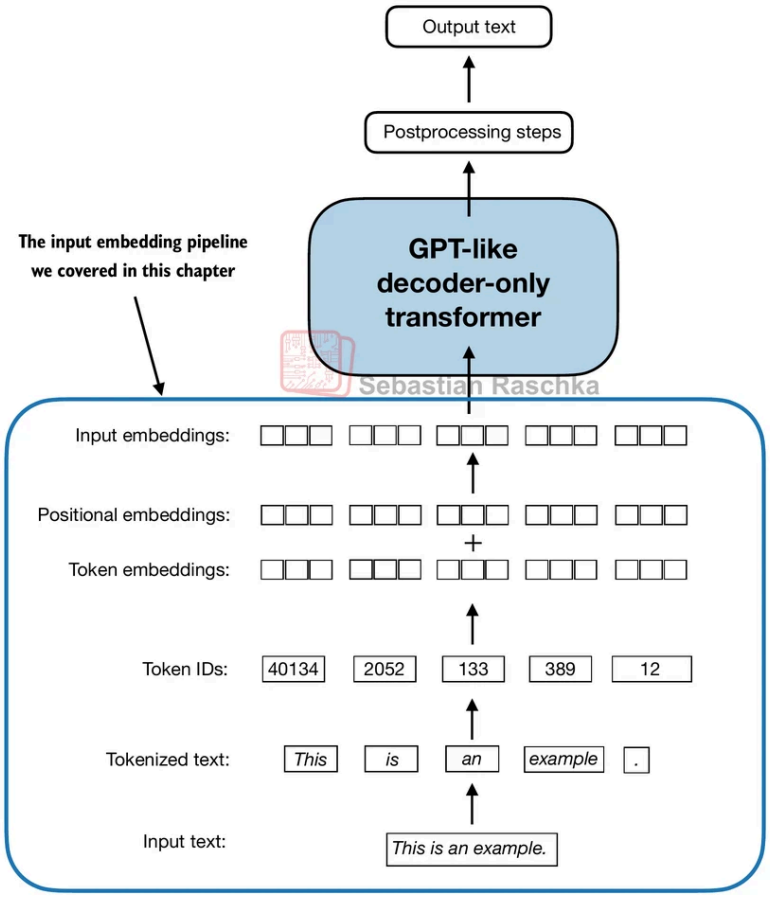
- To create the input embeddings used in an LLM, we simply add the token and the positional embeddings:

In [179…
```
input_embeddings = token_embeddings + pos_embeddings
print(input_embeddings.shape)

# uncomment & execute the following line to see how the embeddings look like
# print(input_embeddings)
```

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

- In the initial phase of the input processing workflow, the input text is segmented into separate tokens
- Following this segmentation, these tokens are transformed into token IDs based on a predefined vocabulary:



Exercise (2.8 Position embeddings)

Print the shapes of the token embeddings, the position embeddings, and the summed input embeddings. Which two tensors were added?

In [ ]:
```
# Exercise 2.8 — your attempt
# token_embeddings, pos_embeddings, and input_embeddings are already in memo
token_shape = None
pos_shape = None
input_shape = None
print('token', token_shape)
print('pos', pos_shape)
print('input', input_shape)
```

Solution (2.8). Token table is `(batch, tokens, dim)`. Position table is `(tokens, dim)` and broadcasts. We add them.

In [185…
```
# Solution 2.8
print('token', tuple(token_embeddings.shape))
print('pos', tuple(pos_embeddings.shape))
print('input', tuple(input_embeddings.shape))
```

```
token (8, 4, 256)
pos (4, 256)
input (8, 4, 256)
```

Comprehensive exercises

Exercise (comprehensive: window then vectors)

Take a 4-token input window and the matching next-token targets. Build a small embedding table (vocab size 50257, vector size 8), look up the input window, and print the resulting shape.

In [ ]:
```python
# Exercise comprehensive: your attempt
import torch
import torch.nn as nn
x = None
y = None
emb = None
print()
```

Solution. `y` is `x` shifted by one. The lookup returns `(4, 8)` for a window of 4 ids and 8-dimensional vectors.

In [ ]:
```python
# Solution comprehensive
import torch
import torch.nn as nn
ids = torch.arange(10)
x = ids[:4]
y = ids[1:5]
emb = nn.Embedding(50257, 8)
print(x, y)
print(emb(x).shape)
```

Takeaways

1. Training rows are **windows**: `x` is ids, `y` is `x` shifted by one (next-token targets).
2. `nn.Embedding` is a lookup table: id to vector.
3. **Position** embeddings are added so order is not lost.
4. Next class: attention mixes those vectors across positions.