

Review section 1: Tokenization to Attention

CSC 352 / CSC 552 · 2026-09-22

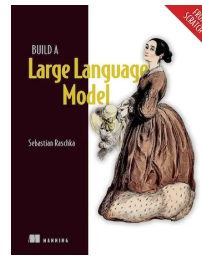
Open in Colab → **File** → **Save a copy in Drive**. You cannot edit the class original.

Start-of-class quiz

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

1. Self-attention: a query scores keys. What two steps then make the context vector?
2. Name one of the three trainable matrices in weighted self-attention.

Figures and the worked examples below follow [Build a Large Language Model From Scratch](#) by [Sebastian Raschka](#). Code repository: <https://github.com/rasbt/LLMs-from-scratch>



1. Tokens and token ids

1. Tokenize means cut text into what?
2. Each piece gets an integer. What is that integer called?
3. This is Lecture 4 SimpleTokenizerV2.encode. What do the three steps do?

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

4. This is the GPT-2 encoder from Lecture 4. What does encode do to the text?

```
tokenizer = tiktoken.get_encoding("gpt2")
ids = tokenizer.encode(text)
```

5. Lecture 4. What does this code build?

```
all_words = sorted(set(preprocessed))
vocab = {token: integer for integer, token in enumerate(all_words)}
```

6. Lecture 4 decode. What do these two lines do?

```
text = " ".join([self.int_to_str[i] for i in ids])
text = re.sub(r'\s+([, .?!"()\' ])', ' ', text)
```

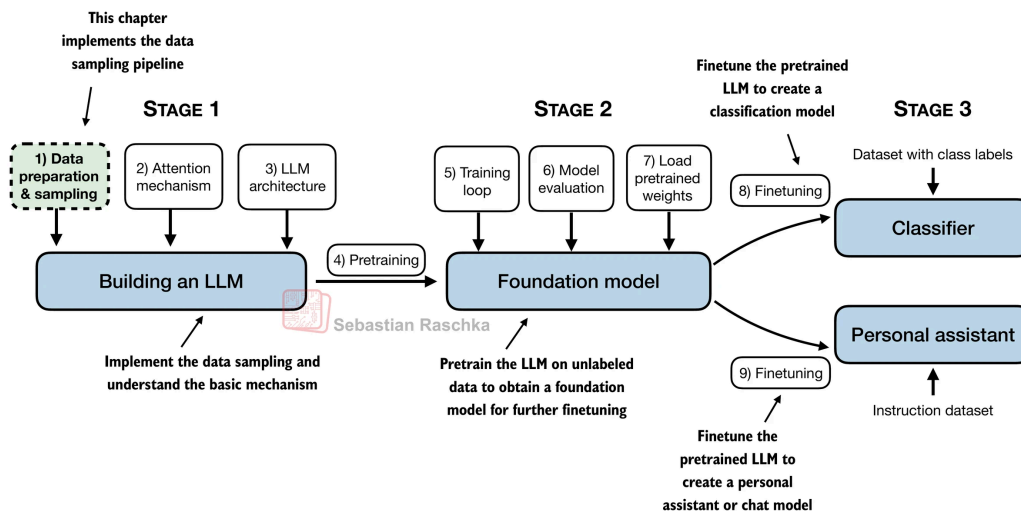
7. What is `<|endoftext|>` ?

The one picture

An LLM does not read English. It does this chain:

text → tokens → token ids → vectors (word + place) → attention → next-token scores

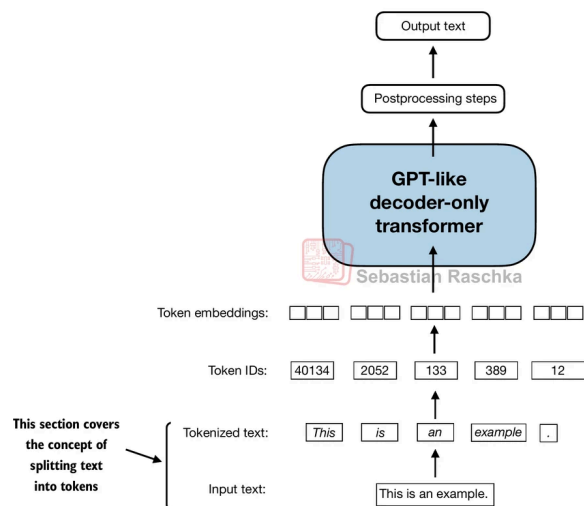
A ticket printer is a useful picture. The kitchen never sees the words "I love pizza". It sees ticket numbers. Attention is the cook looking at earlier tickets while writing the next one.



1. Tokens and token ids

Tokenize means cut the sentence into pieces the model knows: words, punctuation, later subwords.

Those pieces are useless until each one has an integer **token id**. A **vocabulary** is the lookup table from piece to id.



Start with a split on spaces only. The comma stays glued to `Hello` .

In [1]: `import re`

```
text = "Hello, world. This, is a test."
print(re.split(r'(\s)', text))
```

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

Also cut on commas and periods, so punctuation becomes its own piece. In `I want tea.` the period is its own token. The split still leaves empty strings. Strip those.

In [2]: `result = re.split(r'([,.]|\s)', text)`
`result = [item for item in result if item.strip()]`
`print(result)`

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

The same idea covers the other marks from Lecture 4, including `--` .

In [3]: `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', '?']
```

Apply that pattern to a short paragraph. Lecture 4 used the public-domain story *The Verdict*. The steps are the same on a shorter opening.

In [4]: `raw_text = (`
 `"I had always thought Jack Gisburn rather a cheap genius--now I was sure`
 `"The canvas was blank."`
`)`
`preprocessed = re.split(r'([,.;?_!"()\']|--|\s)', raw_text)`
`preprocessed = [item.strip() for item in preprocessed if item.strip()]`
`print(preprocessed)`
`print("token count", len(preprocessed))`

```
['I', 'had', 'always', 'thought', 'Jack', 'Gisburn', 'rather', 'a', 'cheap', 'genius', '--', 'now', 'I', 'was', 'sure', '.', 'The', 'canvas', 'was', 'blank', '.']  
token count 21
```

Sort the unique pieces and number them from 0. That dictionary is the vocabulary. The integer is the token id.

```
In [5]: all_words = sorted(set(preprocessed))  
vocab = {token: integer for integer, token in enumerate(all_words)}  
for token, integer in list(vocab.items())[:8]:  
    print(integer, token)  
print("vocab size", len(vocab))
```

```
0 --  
1 .  
2 Gisburn  
3 I  
4 Jack  
5 The  
6 a  
7 always  
vocab size 18
```

Encode walks the pieces and looks each one up. Decode walks the ids and reads the pieces back.

```
In [6]: class SimpleTokenizerV1:  
    def __init__(self, vocab):  
        self.str_to_int = vocab  
        self.int_to_str = {i: s for s, i in vocab.items()}  
  
    def encode(self, text):  
        pieces = re.split(r'([.,:;?_!"()\'|\]|--|\s])', text)  
        pieces = [item.strip() for item in pieces if item.strip()]  
        return [self.str_to_int[s] for s in pieces]  
  
    def decode(self, ids):  
        text = " ".join(self.int_to_str[i] for i in ids)  
        return re.sub(r'\s+([.,:;?!])', r'\1', text)  
  
tokenizer = SimpleTokenizerV1(vocab)  
ids = tokenizer.encode("The canvas was blank.")  
print(ids)  
print(tokenizer.decode(ids))
```

```
[5, 9, 17, 8, 1]  
The canvas was blank.
```

A word with no menu row crashes this version. We do not crash. We add two special rows and stamp `<|unk|>`.

`<|endoftext|>` is the other special row. It marks the boundary between two texts that were pasted into one training string.

```
In [7]: all_tokens = sorted(set(preprocessed))
all_tokens.extend(["<|endoftext|>", "<|unk|>"])
vocab = {token: integer for integer, token in enumerate(all_tokens)}
print("unk id", vocab["<|unk|>"], "endoftext id", vocab["<|endoftext|>"])

class SimpleTokenizerV2:
    def __init__(self, vocab):
        self.str_to_int = vocab
        self.int_to_str = {i: s for s, i in vocab.items()}

    def encode(self, text):
        pieces = re.split(r'([.,:;?_!"()\'|\]|--|\s|)|', text)
        pieces = [item.strip() for item in pieces if item.strip()]
        pieces = [s if s in self.str_to_int else "<|unk|>" for s in pieces]
        return [self.str_to_int[s] for s in pieces]

    def decode(self, ids):
        text = " ".join(self.int_to_str[i] for i in ids)
        text = re.sub(r'\s+([.,:;?!])', r'\1', text)
        return text.replace("<|endoftext|>", " ")

tokenizer = SimpleTokenizerV2(vocab)
text = "The canvas was mocha. <|endoftext|> Hello world."
print(tokenizer.encode(text))
print(tokenizer.decode(tokenizer.encode(text)))
```

```
unk id 19 endoftext id 18
[5, 9, 17, 19, 1, 18, 19, 19, 1]
The canvas was <|unk|>. <|unk|> <|unk|>.
```

GPT-2 does not stamp one unknown token for a new spelling. **BPE** cuts it into leftover pieces the table already has. The result is still a list of ids.

```
In [8]: import tiktoken

gpt2 = tiktoken.get_encoding("gpt2")
for word in ["mocha", "mochalattezzz"]:
    ids = gpt2.encode(word)
    print(word, ids, [gpt2.decode([i]) for i in ids])
```

```
mocha [76, 5374, 64] ['m', 'och', 'a']
mochalattezzz [76, 5374, 282, 265, 660, 3019, 89] ['m', 'och', 'al', 'at',
'te', 'zz', 'z']
```

2. Training windows and embeddings

8.

```
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]
Lecture 5 GPTDatasetV1. token_ids = [10, 11, 12, 13, 14, 15],
max_length = 4, stride = 1. Write both chunks for i = 0 and for i = 1.
```

9. Lecture 5 create_dataloader_v1. What three objects does this build, in order?

```
tokenizer = tiktoken.get_encoding("gpt2")
dataset = GPTDatasetV1(txt, tokenizer, max_length, stride)
dataloader = DataLoader(dataset, batch_size=batch_size,
shuffle=shuffle)
```

10. Lecture 5 lookup. What does emb(x) do to these four ids?

```
emb = nn.Embedding(50257, 8)
x = torch.tensor([15, 16, 18, 17])
print(emb(x).shape)
```

11. At each position, what is the model learning to predict?

12. A list of 9 ids, frame size 4, stride 1. How many training rows?

13. Lecture 5. What does the second line do? What is lost if you skip pos_embeddings?

```
pos_embeddings = pos_embedding_layer(torch.arange(max_length))
input_embeddings = token_embeddings + pos_embeddings
```

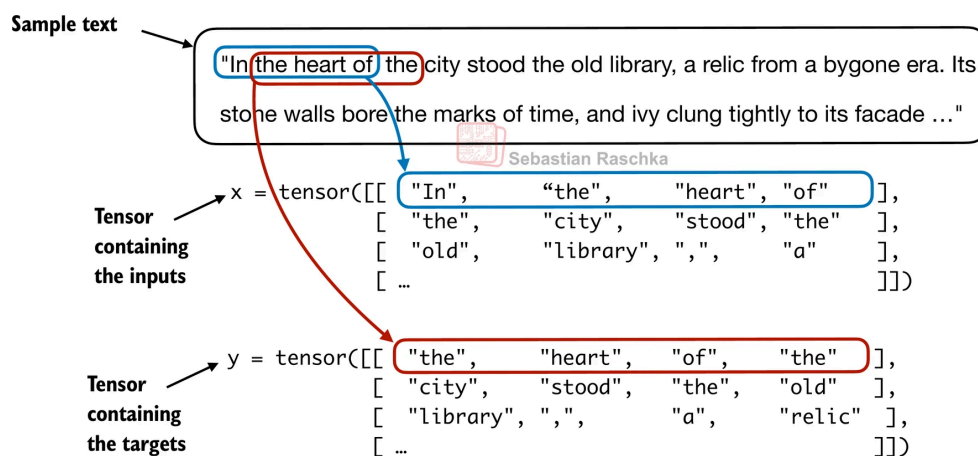
14. `nn.Embedding` looks up an id and returns what?

15. The same token id sits in two seats. Before adding position, are the two vectors equal?

16. After you add the position vectors, should those two rows stay equal?

2. Training windows and embeddings

The model learns the **next token**. For a chunk of ids `x`, the target `y` is that chunk shifted one step to the right.



```
In [9]: enc = gpt2.encode(raw_text)
context_size = 4
x = enc[:context_size]
```

```
y = enc[1:context_size + 1]
print("x:", x)
print("y:", y)
```

```
x: [40, 550, 1464, 1807]
y: [550, 1464, 1807, 3619]
```

Read one position at a time. The context grows. The desired token is the next id.

```
In [10]: for i in range(1, context_size + 1):
          context = enc[:i]
          desired = enc[i]
          print(gpt2.decode(context), "---->", gpt2.decode([desired]))
```

```
I ----> had
I had ----> always
I had always ----> thought
I had always thought ----> Jack
```

A sliding window of frame size `max_length` starts a new row every `stride` tokens. The number of rows from a list is how many times that start fits. With stride 1, row count is `len(ids) - frame size`. A list of 9 ids and frame size 4 gives 5 rows.

```
In [11]: frame = 4
          print("rows from this paragraph", len(enc) - frame)
          toy = list(range(9))
          rows = [(toy[i:i + frame], toy[i + 1:i + frame + 1]) for i in range(len(toy))]
          print("9 ids, frame 4 ->", len(rows), "rows")
          print("last x, y", rows[-1])
```

```
rows from this paragraph 19
9 ids, frame 4 -> 5 rows
last x, y ([4, 5, 6, 7], [5, 6, 7, 8])
```

Lecture 5 wraps the same shift in a dataset. Each item is one `(x, y)` pair. `stride` is how far the next window moves. Stride 1 overlaps. Stride equal to the frame does not.

```
In [12]: import torch
          from torch.utils.data import Dataset, DataLoader

          class GPTDatasetV1(Dataset):
              def __init__(self, txt, tokenizer, max_length, stride):
                  self.input_ids = []
                  self.target_ids = []
                  token_ids = tokenizer.encode(txt)
                  for i in range(0, len(token_ids) - max_length, stride):
                      self.input_ids.append(torch.tensor(token_ids[i:i + max_length]))
                      self.target_ids.append(torch.tensor(token_ids[i + 1:i + max_length]))

              def __len__(self):
                  return len(self.input_ids)

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

```

loader = DataLoader(
    GPTDatasetV1(raw_text, gpt2, max_length=4, stride=1),
    batch_size=1, shuffle=False,
)
it = iter(loader)
print("window 0", next(it))
print("window 1", next(it))

```

window 0 [tensor([[40, 550, 1464, 1807]]), tensor([[550, 1464, 1807, 3619]])]
 window 1 [tensor([[550, 1464, 1807, 3619]]), tensor([[1464, 1807, 3619, 402]])]

Ids are still locker numbers. `nn.Embedding` is the wall of lockers. Id 3 opens row 3 and pulls out a vector. That row **is** the vector.

```

In [13]: torch.manual_seed(0)
vocab_size, dim = 6, 3
embedding = torch.nn.Embedding(vocab_size, dim)
print(embedding.weight)
print("row 3", embedding.weight[3])
print("lookup", embedding(torch.tensor([3])))

```

Parameter containing:

```

tensor([[-1.1258, -1.1524,  0.5667],
        [ 0.7935,  0.5988, -1.5551],
        [-0.3414,  1.8530,  0.4681],
        [-0.1577, -0.1734,  0.1835],
        [ 1.3894,  1.5863,  0.9463],
        [-0.8437,  0.9318,  1.2590]], requires_grad=True)
row 3 tensor([-0.1577, -0.1734,  0.1835], grad_fn=<SelectBackward0>)
lookup tensor([-0.1577, -0.1734,  0.1835], grad_fn=<EmbeddingBackward0>)

```

Position embeddings are a second wall, indexed by the seat: 0, 1, 2, and so on. Add the two vectors. The same token id in two seats is the same token vector until you add the position vector. Skip that add, and the model loses the order.

```

In [14]: torch.manual_seed(0)
token_table = torch.nn.Embedding(8, 4)
pos_table = torch.nn.Embedding(4, 4)
ids = torch.tensor([1, 5, 1]) # same id in seat 0 and seat 2
token_vecs = token_table(ids)
pos_vecs = pos_table(torch.arange(3))
print("same token, before position", torch.equal(token_vecs[0], token_vecs[2]))
mixed = token_vecs + pos_vecs
print("after position", torch.equal(mixed[0], mixed[2]))
print(mixed.shape)

```

```

same token, before position True
after position False
torch.Size([3, 4])

```

3. Self-attention

17. Lecture 6. What number is written at row i, column j?

Six already-embedded words. The second word, *journey*, is the query. With no extra matrices, the query is that token's own vector.

```
In [15]: inputs = torch.tensor(
    [ [0.43, 0.15, 0.89], # Your
      [0.55, 0.87, 0.66], # journey
      [0.57, 0.85, 0.64], # starts
      [0.22, 0.58, 0.33], # with
      [0.77, 0.25, 0.10], # one
      [0.05, 0.80, 0.55] # step
    ]
    query = inputs[1]
    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])
```

A dot product is the element-wise products, added up. A neighbor's score is that dot product.

```
In [16]: res = 0.
    for idx, element in enumerate(inputs[0]):
        res += inputs[0][idx] * query[idx]
    print(res)
    print(torch.dot(inputs[0], query))
```

```
tensor(0.9544)
tensor(0.9544)
```

Divide by the sum and the row adds to 1. Softmax does that job in practice, and it handles large scores more safely than a hand-written `exp`.

```
In [17]: by_sum = attn_scores_2 / attn_scores_2.sum()
    print("by sum", by_sum, float(by_sum.sum()))

    def softmax_naive(x):
        return torch.exp(x) / torch.exp(x).sum(dim=0)

    print("naive", softmax_naive(attn_scores_2))
    attn_weights_2 = torch.softmax(attn_scores_2, dim=0)
    print("softmax", attn_weights_2, float(attn_weights_2.sum()))
```

```
by sum tensor([0.1455, 0.2278, 0.2249, 0.1285, 0.1077, 0.1656]) 1.0000001192
092896
naive tensor([0.1385, 0.2379, 0.2333, 0.1240, 0.1082, 0.1581])
softmax tensor([0.1385, 0.2379, 0.2333, 0.1240, 0.1082, 0.1581]) 1.0
```

The context vector is the weights times the value vectors. Here the value vectors are the same input vectors. One step is `weight * vector`, then add those up.

```
In [18]: context_vec_2 = torch.zeros(query.shape)
    for i, x_i in enumerate(inputs):
```

```
context_vec_2 += attn_weights_2[i] * x_i
print(context_vec_2)
```

```
tensor([0.4419, 0.6515, 0.5683])
```

The same dots for every query are one matrix. `inputs @ inputs.T` replaces the double loop. Softmax runs on each row. `weights @ inputs` builds every context vector at once.

```
In [19]: attn_scores = inputs @ inputs.T
attn_weights = torch.softmax(attn_scores, dim=-1)
all_context = attn_weights @ inputs
print("scores shape (queries, keys)", tuple(attn_scores.shape))
print("dim=-1 is the last axis, size", attn_scores.shape[-1])
print("query 1 scores", attn_scores[1])
print("query 1 weights after softmax dim=-1", attn_weights[1])
print("row sums", attn_weights.sum(dim=-1))
print(all_context.shape)
print("journey matches the one-query result", torch.allclose(all_context[1],
```

```
scores shape (queries, keys) (6, 6)
dim=-1 is the last axis, size 6
query 1 scores tensor([0.9544, 1.4950, 1.4754, 0.8434, 0.7070, 1.0865])
query 1 weights after softmax dim=-1 tensor([0.1385, 0.2379, 0.2333, 0.1240,
0.1082, 0.1581])
row sums tensor([1.0000, 1.0000, 1.0000, 1.0000, 1.0000, 1.0000])
torch.Size([6, 3])
journey matches the one-query result True
```

`dim` chooses the axis. Count from the left, starting at 0. Count from the right, and the last axis is `-1`.

A score matrix has shape `(queries, keys)`.

- Axis `0` is the queries. `softmax(..., dim=0)` walks **down a column**.
- Axis `1` is the keys. On a matrix, that axis is also `-1`. `softmax(..., dim=-1)` walks **across one row**.

One row is one query. After `dim=-1`, that row adds to 1. Attention wants the row to add to 1. It does not want the column to add to 1.

A later batch has shape `(batch, queries, keys)`. Axis `-1` is still the keys. Axis `1` is no longer the last axis, which is why the code writes `dim=-1` and not `dim=1`.

```
In [20]: scores = torch.tensor([
    [1., 1., 1.],
    [0., 0., 10.],
])
print("shape", tuple(scores.shape))
print("axis 0 size (down the columns)", scores.shape[0])
print("axis -1 size (across the rows)", scores.shape[-1])
print("query 1 scores", scores[1].tolist())
```

```

row_w = torch.softmax(scores, dim=-1)
print("softmax dim=-1, query 1 weights")
print(row_w[1])
print("that row sums to", round(float(row_w[1].sum()), 4))
print("every row sums to", row_w.sum(dim=-1))

col_w = torch.softmax(scores, dim=0)
print("softmax dim=0 walks down the columns")
print(col_w)
print("column sums", [round(float(v), 4) for v in col_w.sum(dim=0)])

batched = scores.unsqueeze(0)
print("batch shape", tuple(batched.shape))
print("dim=-1 is still size", batched.shape[-1], "not", batched.shape[1])
print("batch row sums", torch.softmax(batched, dim=-1).sum(dim=-1))

```

```

shape (2, 3)
axis 0 size (down the columns) 2
axis -1 size (across the rows) 3
query 1 scores [0.0, 0.0, 10.0]
softmax dim=-1, query 1 weights
tensor([4.5396e-05, 4.5396e-05, 9.9991e-01])
that row sums to 1.0
every row sums to tensor([1.0000, 1.0000])
softmax dim=0 walks down the columns
tensor([[7.3106e-01, 7.3106e-01, 1.2339e-04],
        [2.6894e-01, 2.6894e-01, 9.9988e-01]])
column sums [1.0, 1.0, 1.0]
batch shape (1, 2, 3)
dim=-1 is still size 3 not 2
batch row sums tensor([[1.0000, 1.0000]])

```

Same three words, two versions. The row is *on*. A larger weight is darker.

Without matrices, Q, K, and V are the raw vectors. With matrices, `W_key` turns each vector before the scores are computed, so the largest weight can move.

```

In [21]: names = ["cat", "on", "mat"]
trio = torch.tensor([[1.0, 0.0], [0.2, 0.8], [0.0, 1.0]])
raw = torch.softmax(trio @ trio.T, dim=-1)
W_key = torch.tensor([[0.0, 1.0], [1.0, 0.0]])
keys = trio @ W_key
learned = torch.softmax((trio @ keys.T) / keys.shape[-1]**0.5, dim=-1)
print("unweighted, row on", raw[1])
print("weighted, row on", learned[1])
print("unweighted max", names[int(raw[1].argmax())])
print("weighted max", names[int(learned[1].argmax())])
print("score matrix shape", tuple((trio @ trio.T).shape))
print("last axis size", (trio @ trio.T).shape[-1])
print("row sums, dim=-1", raw.sum(dim=-1).tolist())

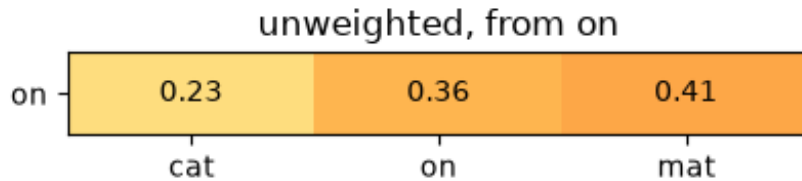
```

```

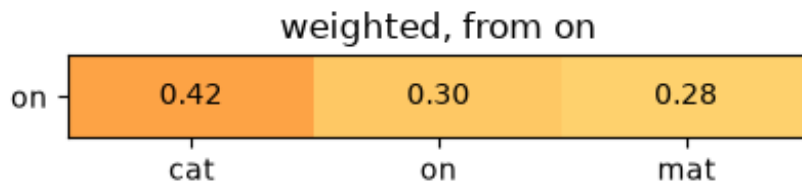
unweighted, row on tensor([0.2253, 0.3641, 0.4106])
weighted, row on tensor([0.4226, 0.3010, 0.2765])
unweighted max mat
weighted max cat
score matrix shape (3, 3)
last axis size 3
row sums, dim=-1 [1.0, 0.9999999403953552, 1.0]

```

Unweighted. Q, K, and V are the raw vectors.



Weighted. The same tokens after `W_key`.



4. Q, K, and V

23. In one short phrase, what is Q?

24. In one short phrase, what is K?

25. In one short phrase, what is V?

26. Lecture 6 SelfAttention_v1.forward. What do the three matrices produce, why divide before softmax, and what is returned?

```

def forward(self, x):
    keys = x @ self.W_key
    queries = x @ self.W_query
    values = x @ self.W_value
    attn_scores = queries @ keys.T
    attn_weights = torch.softmax(
        attn_scores / keys.shape[-1]**0.5, dim=-1
    )
    context_vec = attn_weights @ values
    return context_vec

```

27. Lecture 6 SelfAttention_v2. Same three jobs as v1. What replaced `x @ W_key`?

```

keys = self.W_key(x)
queries = self.W_query(x)
values = self.W_value(x)

```

28. This is the division inside forward. What does dividing by $\sqrt{d_k}$ do before softmax?

```
attn_scores = torch.tensor([2., 4.])
d_k = 4
print(attn_scores / d_k**0.5)
```

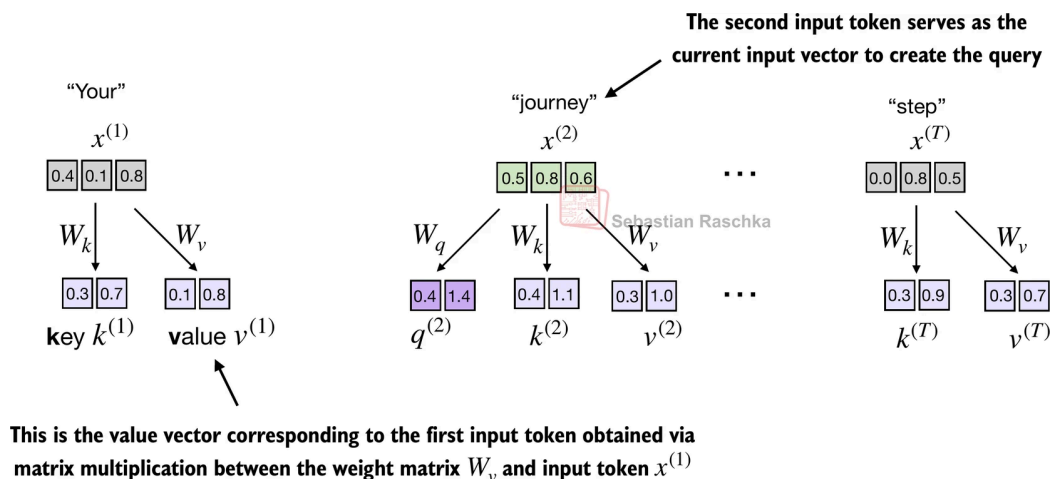
4. Q, K, and V

Unweighted attention uses the raw vectors as query, key, and value.

Trainable attention first multiplies the same tokens by three matrices. Their names are `W_query`, `W_key`, and `W_value`.

- **Q** = what I am asking
- **K** = what I offer as a label
- **V** = what I pass along if you pick me

Same spelling, different question. After training, "bank" next to "river" and "bank" next to "money" no longer share one frozen direction.



Two tokens, no batch

Same shapes as the code below, with the batch dimension left off. Two tokens. Each value is 2 numbers wide.

`attn_weights` has shape (T, T) . Row i says how token i splits its attention. Across that row, the numbers add to 1.

`values` has shape (T, d_{out}) . Row j is the value vector of token j .

`context` has shape (T, d_{out}) . Row i is the context vector of token i :

$$\mathbf{c}_i = \sum_j a_{ij} \mathbf{v}_j$$

a_{ij} is how much token i looks at token j . $\mathbf{v}_j$ is token j 's value.
 $\mathbf{c}_i$ is token i after it has mixed the sentence.

Use these weights. Token 0 mostly looks at itself. Token 1 looks at both halves.

```
attn_weights =  
[[0.88, 0.12],  
 [0.50, 0.50]]
```

And these values:

```
values =  
[[1.0, 0.0],  
 [0.0, 1.0]]
```

Token 0:

```
c0 = 0.88 * [1.0, 0.0] + 0.12 * [0.0, 1.0]  
    = [0.88, 0.12]
```

Token 1:

```
c1 = 0.50 * [1.0, 0.0] + 0.50 * [0.0, 1.0]  
    = [0.50, 0.50]
```

So the context matrix is

```
[[0.88, 0.12],  
 [0.50, 0.50]]
```

Token 0 is still almost its own value, with a little of token 1 mixed in. Token 1 mixes the two values half and half. That mix is the representation after the token has looked at the sentence. The context vector is not the attention score. It is the values blended by those scores.

Four steps, in this order:

1. Project. $Q = x @ W_q$, $K = x @ W_k$, $V = x @ W_v$.
2. Score. `attn_scores[i, j]` is the dot product of query i and key j .
3. Weights. Divide those scores by $\sqrt{d}$, then softmax each row (`dim=-1`).
4. Mix. `context[i] = sum_j attn_weights[i, j] * v_j`.

The first three steps decide who to look at, and how much. The fourth step is the one that moves the information.

One picture. Each token is a person in the room.

- **query**: what I am looking for
- **key**: the label I wear, so others can find me
- **value**: the content I actually hand over
- **context vector**: after I have asked the whole room, the bag I packed from what people handed me, weighted by how relevant each person was

People in the same sentence pack different bags, so each token gets its own context vector.

A batch only adds a leading dimension, shape `(B, T, d_out)`. Each sample, and each token, still does this same multiply.

```
In [22]: import torch
attn_weights = torch.tensor([[0.88, 0.12],
                             [0.50, 0.50]])
values = torch.tensor([[1.0, 0.0],
                       [0.0, 1.0]])
c0 = 0.88 * values[0] + 0.12 * values[1]
c1 = 0.50 * values[0] + 0.50 * values[1]
context = attn_weights @ values
print("c0", c0)
print("c1", c1)
print(context)
print("shape", tuple(attn_weights.shape), "last axis", attn_weights.shape[-1])
print("sum on dim=-1, one number per row", attn_weights.sum(dim=-1).tolist())

c0 tensor([0.8800, 0.1200])
c1 tensor([0.5000, 0.5000])
tensor([[0.8800, 0.1200],
        [0.5000, 0.5000]])
shape (2, 2) last axis 2
sum on dim=-1, one number per row [1.0, 1.0]
```

The same two tokens, now from Q, K, and V

No batch. Two tokens. The input `x` is width 2. These three matrices are tiny on purpose, so each product is one multiplication.

```
x =
[[1, 0],    # token 0
 [0, 1]]    # token 1
```

```
W_q = [[2],
       [0]]
```

```
W_k = [[1],
       [0]]
```

```
W_v = [[1, 0],
       [0, 1]]
```

Project. A row of $\mathbf{x}$ times a matrix is that token's Q, K, or V.

Token 0's query: $[1, 0] @ [[2], [0]] = 2$

Token 1's query: $[0, 1] @ [[2], [0]] = 0$

Keys, the same way: token 0's key is 1 , token 1's key is 0 .

$\mathbf{W}_V$ is the identity, so V is $\mathbf{x}$ again:

```
V =  
[[1, 0],  
 [0, 1]]
```

Score. $\text{attn_scores}[i, j]$ is query i times key j .

Token 0: $2 * 1 = 2$ against token 0, and $2 * 0 = 0$ against token 1.

Token 1: both products are 0.

```
attn_scores =  
[[2, 0],  
 [0, 0]]
```

Weights. The key width here is 1, so $\sqrt{d} = 1$ and dividing leaves the scores as they are. Softmax each row.

Token 0: $\exp(2) \approx 7.39$, $\exp(0) = 1$, and $7.39 / (7.39 + 1) \approx 0.88$. The other weight is about 0.12 .

Token 1: $\exp(0)$ and $\exp(0)$ are the same number, so the row is $0.50, 0.50$.

Those are the weights from the cell above. The mix $\text{context} = \text{attn_weights} @ V$ is the same sum, so the context is again

```
[[0.88, 0.12],  
 [0.50, 0.50]]
```

$\mathbf{Q}$ in the next cell has shape $(2, 1)$. Two tokens, one number each, standing in a column. $\mathbf{Q.view(-1)}$ does not add or sort those numbers. -1 here means "make one axis, and set its length to however many numbers there are." $2 * 1 = 2$, so the shape becomes $(2,)$ and the print is $[2.0, 0.0]$.

That -1 is not the same job as $\text{dim}=-1$. $\text{view}(-1)$ fills in a length. $\text{dim}=-1$ names the last axis for softmax. On these scores the last axis has length 2, one weight per token, and each row adds to 1.

```
In [23]: import torch
q_col = torch.tensor([[2.],
                      [0.]])
print("column shape", tuple(q_col.shape), "how many numbers", q_col.numel())
print("view(-1) shape", tuple(q_col.view(-1).shape))
print("view(-1)", q_col.view(-1).tolist())
six = torch.arange(6)
print("six numbers", six.tolist())
print("view(2, -1) fills the last size: 6 / 2 = 3", tuple(six.view(2, -1).shape))
print(six.view(2, -1))
print("view(-1, 3) fills the first size: 6 / 3 = 2", tuple(six.view(-1, 3).shape))
print(six.view(-1, 3))
```

```
column shape (2, 1) how many numbers 2
view(-1) shape (2,)
view(-1) [2.0, 0.0]
six numbers [0, 1, 2, 3, 4, 5]
view(2, -1) fills the last size: 6 / 2 = 3 (2, 3)
tensor([[0, 1, 2],
        [3, 4, 5]])
view(-1, 3) fills the first size: 6 / 3 = 2 (2, 3)
tensor([[0, 1, 2],
        [3, 4, 5]])
```

The next cell prints that same `view(-1)` on Q and K from the multiply, and the row sums from `softmax(..., dim=-1)`.

```
In [24]: import torch
x = torch.tensor([[1., 0.],
                  [0., 1.]])
W_q = torch.tensor([[2.],
                    [0.]])
W_k = torch.tensor([[1.],
                    [0.]])
W_v = torch.tensor([[1., 0.],
                    [0., 1.]])
Q = x @ W_q
K = x @ W_k
V = x @ W_v
scores = Q @ K.T
scaled = scores / K.shape[-1]**0.5
weights = torch.softmax(scaled, dim=-1)
context = weights @ V
print("Q shape", tuple(Q.shape), "view(-1)", Q.view(-1).tolist())
print("K shape", tuple(K.shape), "view(-1)", K.view(-1).tolist())
print("scores shape", tuple(scores.shape), "last axis", scores.shape[-1])
print("scaled", scaled.tolist())
print("weights", [[round(v, 2) for v in row] for row in weights.tolist()])
print("sum on dim=-1", [round(float(v), 4) for v in weights.sum(dim=-1).tolist()])
print("context", [[round(v, 2) for v in row] for row in context.tolist()])
```

```

Q shape (2, 1) view(-1) [2.0, 0.0]
K shape (2, 1) view(-1) [1.0, 0.0]
scores shape (2, 2) last axis 2
scaled [[2.0, 0.0], [0.0, 0.0]]
weights [[0.88, 0.12], [0.5, 0.5]]
sum on dim=-1 [1.0, 1.0]
context [[0.88, 0.12], [0.5, 0.5]]

```

The input dimension here is 3. The query, key, and value dimension is 2, so the arithmetic is easy to read. GPT usually keeps those dimensions equal.

```

In [25]: x_2 = inputs[1]
         d_in = inputs.shape[1]
         d_out = 2
         torch.manual_seed(123)
         W_query = torch.nn.Parameter(torch.rand(d_in, d_out), requires_grad=False)
         W_key = torch.nn.Parameter(torch.rand(d_in, d_out), requires_grad=False)
         W_value = torch.nn.Parameter(torch.rand(d_in, d_out), requires_grad=False)
         query_2 = x_2 @ W_query
         keys = inputs @ W_key
         values = inputs @ W_value
         print("Q", query_2)
         print("K", keys.shape, "V", values.shape)

```

```

Q tensor([0.4306, 1.4551])
K torch.Size([6, 2]) V torch.Size([6, 2])

```

Scores are $Q K^T$ transpose. Divide by $\sqrt{d}$ before softmax, where d is the key dimension. Large dots would push softmax into one 1 and many zeros. The division keeps the dots in a friendly range. The context vector is those weights times V .

```

In [26]: attn_scores_2 = query_2 @ keys.T
         d_k = keys.shape[1]
         attn_weights_2 = torch.softmax(attn_scores_2 / d_k**0.5, dim=-1)
         context_vec_2 = attn_weights_2 @ values
         print("scores", attn_scores_2)
         print("weights", attn_weights_2)
         print("context", context_vec_2)
         print("weights shape", tuple(attn_weights_2.shape))
         print("sum on dim=-1", attn_weights_2.sum(dim=-1).tolist())

```

```

scores tensor([1.2705, 1.8524, 1.8111, 1.0795, 0.5577, 1.5440])
weights tensor([0.1500, 0.2264, 0.2199, 0.1311, 0.0906, 0.1820])
context tensor([0.3061, 0.8210])
weights shape (6,)
sum on dim=-1 1.0

```

Those learned scores can pick a different neighbor than the raw dots $x @ x.T$, because Q and K are not the raw vectors anymore.

```

In [27]: raw = inputs @ inputs.T
         print("raw winner for journey", int(raw[1].argmax()))
         print("learned winner for journey", int(attn_weights_2.argmax()))

```

raw winner for journey 1
learned winner for journey 1

The same three lines for every token sit in one module. `SelfAttention_v1` stores the three matrices. `SelfAttention_v2` uses `nn.Linear` for the same product.

```
In [28]: import torch.nn as nn

class SelfAttention_v1(nn.Module):
    def __init__(self, d_in, d_out):
        super().__init__()
        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
        attn_weights = torch.softmax(attn_scores / keys.shape[-1]**0.5, dim=-1)
        return attn_weights @ values

torch.manual_seed(123)
print(SelfAttention_v1(d_in, d_out)(inputs).shape)

class SelfAttention_v2(nn.Module):
    def __init__(self, d_in, d_out, qkv_bias=False):
        super().__init__()
        self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_key = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_value = nn.Linear(d_in, d_out, bias=qkv_bias)

    def forward(self, x):
        keys = self.W_key(x)
        queries = self.W_query(x)
        values = self.W_value(x)
        attn_scores = queries @ keys.T
        attn_weights = torch.softmax(attn_scores / keys.shape[-1]**0.5, dim=-1)
        return attn_weights @ values

torch.manual_seed(789)
print(SelfAttention_v2(d_in, d_out)(inputs).shape)

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

Takeaways

1. **Tokenize** cuts text. A **vocab** maps each piece to an id. `<|unk|>` and `<|endoftext|>` are extra rows. **BPE** cuts an unknown spelling into known pieces and still returns ids.

2. Training rows are **windows**: `y` is `x` shifted by one. Row count follows the frame and the stride.
3. **Embeddings** look up a vector. **Position** embeddings are added so the same id in two seats does not look identical.
4. **Self-attention** without matrices: the query is the token's own vector, the score is a dot product, `softmax(..., dim=-1)` makes each row of weights add to 1, and the context vector is a weighted sum of the values.
5. Trainable attention multiplies by `W_query`, `W_key`, and `W_value`, divides the scores by $\sqrt{d}$, then does `weights @ V`.
6. Next class: this attention sits inside a GPT block. Causal masking is not in today's notebook.