CS498: Algorithmic Engineering

Lecture 24: Sequence Modeling, Tokenization & Attention

Elfarouk Harb

University of Illinois Urbana-Champaign

Week 14

Outline

CNNs Are Great for Images. But...

CNNs exploit spatial locality and translation invariance. Perfect for pixel grids.

CNNs Are Great for Images. But...

CNNs exploit spatial locality and translation invariance. Perfect for pixel grids.

But many problems are sequences: text, audio, time series, DNA.

CNNs Are Great for Images. But...

CNNs exploit spatial locality and translation invariance. Perfect for pixel grids.

But many problems are sequences: text, audio, time series, DNA.

- Order matters: “Dog bites man” $\neq$ “Man bites dog”

CNNs Are Great for Images. But...

CNNs exploit spatial locality and translation invariance. Perfect for pixel grids.

But many problems are sequences: text, audio, time series, DNA.

- Order matters: “Dog bites man” $\neq$ “Man bites dog”
- Length varies: a tweet has 10 words, a novel has 100,000

CNNs Are Great for Images. But...

CNNs exploit spatial locality and translation invariance. Perfect for pixel grids.

But many problems are sequences: text, audio, time series, DNA.

- Order matters: "Dog bites man" $\neq$ "Man bites dog"
- Length varies: a tweet has 10 words, a novel has 100,000
- Long-range dependencies: "The cat that I saw yesterday at the park was orange."

CNNs Are Great for Images. But...

CNNs exploit spatial locality and translation invariance. Perfect for pixel grids.

But many problems are sequences: text, audio, time series, DNA.

- Order matters: "Dog bites man" $\neq$ "Man bites dog"
- Length varies: a tweet has 10 words, a novel has 100,000
- Long-range dependencies: "The cat that I saw yesterday at the park was orange."

A CNN with kernel size 3 cannot connect word 1 to word 50 without many layers.

Language Modeling

Task: given previous tokens, predict the next one.

Language Modeling

Task: given previous tokens, predict the next one.

$p(x_t \mid x_1, x_2, \ldots, x_{t-1})$

Language Modeling

Task: given previous tokens, predict the next one.

$p(x_t \mid x_1, x_2, \ldots, x_{t-1})$

Example: "The students opened their ________"

Language Modeling

Task: given previous tokens, predict the next one.

$p(x_t \mid x_1, x_2, \ldots, x_{t-1})$

Example: "The students opened their ________"
$\Rightarrow$ "books" ($p \approx 0.35$), "laptops" ($p \approx 0.08$), "elephants" ($p \approx 0.0001$)

Language Modeling

Task: given previous tokens, predict the next one.

$p(x_t \mid x_1, x_2, \ldots, x_{t-1})$

Example: "The students opened their ________"
$\Rightarrow$ "books" ($p \approx 0.35$), "laptops" ($p \approx 0.08$), "elephants" ($p \approx 0.0001$)

Applications:

Language Modeling

Task: given previous tokens, predict the next one.

$p(x_t \mid x_1, x_2, \ldots, x_{t-1})$

Example: "The students opened their ________"

$\Rightarrow$ "books" ($p \approx 0.35$), "laptops" ($p \approx 0.08$), "elephants" ($p \approx 0.0001$)

Applications:

- Text generation: ChatGPT, Claude, Gemini
- Autocomplete: GitHub Copilot, Gmail Smart Compose
- Translation, summarization, code generation

N-Grams

Bigram: predict the next word from only the previous word.

N-Grams

Bigram: predict the next word from only the previous word.

$$P(w_t \mid w_{t-1}) = \frac{\text{count}(w_{t-1}, w_t)}{\text{count}(w_{t-1})}$$

N-Grams

Bigram: predict the next word from only the previous word.

$$\mathrm{P}(\mathrm{w_t} \mid \mathrm{w_{t-1}}) = \frac{\mathrm{count}(\mathrm{w_{t-1}},\ \mathrm{w_t})}{\mathrm{count}(\mathrm{w_{t-1}})}$$

Example: "the cat" appears 50 times, "the" appears 1,000 times.

$$\mathrm{P}(\mathrm{cat} \mid \mathrm{the}) = 50/1000 = 0.05$$

N-Grams

Bigram: predict the next word from only the previous word.

$$\mathrm{P}(\mathrm{w}_{\mathrm{t}} \mid \mathrm{w}_{\mathrm{t}-1}) = \frac{\mathrm{count}(\mathrm{w}_{\mathrm{t}-1}, \mathrm{w}_{\mathrm{t}})}{\mathrm{count}(\mathrm{w}_{\mathrm{t}-1})}$$

Example: “the cat” appears 50 times, “the” appears 1,000 times.

$\mathrm{P}(\mathrm{cat} \mid \mathrm{the}) = 50/1000 = 0.05$

Problem: “the purple elephant” never appeared in training.

N-Grams

Bigram: predict the next word from only the previous word.

$$P(w_t \mid w_{t-1}) = \frac{\text{count}(w_{t-1}, w_t)}{\text{count}(w_{t-1})}$$

Example: "the cat" appears 50 times, "the" appears 1,000 times.

$P(\text{cat} \mid \text{the}) = 50/1000 = 0.05$

Problem: "the purple elephant" never appeared in training.

$P(\text{elephant} \mid \text{the purple}) = 0$. Not low. Zero.

N-Grams

Bigram: predict the next word from only the previous word.

$$P(w_t \mid w_{t-1}) = \frac{\text{count}(w_{t-1}, w_t)}{\text{count}(w_{t-1})}$$

Example: "the cat" appears 50 times, "the" appears 1,000 times.

$P(\text{cat} \mid \text{the}) = 50/1000 = 0.05$

Problem: "the purple elephant" never appeared in training.

$P(\text{elephant} \mid \text{the purple}) = 0$. Not low. Zero.

N-grams memorize co-occurrences but cannot generalize.

Cross-Entropy and Perplexity

How do we measure language model quality?

Cross-Entropy and Perplexity

How do we measure language model quality?

Cross-entropy loss (average over a sequence of length T):

$$L = -\frac{1}{T}\sum_{t=1}^{T} \log p(x_t \mid x_{<t})$$

Cross-Entropy and Perplexity

How do we measure language model quality?

Cross-entropy loss (average over a sequence of length T):

$$L = -\frac{1}{T}\sum_{t=1}^{T} \log p(x_t \mid x_{<t})$$

Lower L = model assigns higher probability to the actual next token.

Cross-Entropy and Perplexity

How do we measure language model quality?

Cross-entropy loss (average over a sequence of length T):

$$L = -\frac{1}{T} \sum_{t=1}^{T} \log p(x_t \mid x_{<t})$$

Lower L = model assigns higher probability to the actual next token.

Perplexity: PPL = exp(L). "How many tokens is the model choosing between?"

Cross-Entropy and Perplexity

How do we measure language model quality?

Cross-entropy loss (average over a sequence of length T):

$$L = -\frac{1}{T}\sum_{t=1}^{T} \log p(x_t \mid x_{<t})$$

Lower L = model assigns higher probability to the actual next token.

Perplexity: PPL = exp(L). "How many tokens is the model choosing between?"

PPL	Meaning
$\|V\| = 50{,}000$	Random guessing
≈ 20–30	Good language model
1	Perfect prediction

What We Need

A model for sequences must be:

What We Need

A model for sequences must be:

1. Variable length (sentences differ in length)

What We Need

A model for sequences must be:

1. Variable length (sentences differ in length)
2. Long range (subject-verb agreement across many words)

What We Need

A model for sequences must be:

1. Variable length (sentences differ in length)
2. Long range (subject-verb agreement across many words)
3. Differentiable (train with gradient descent)
4. Parallel (fast on GPUs)

What We Need

A model for sequences must be:

1. Variable length (sentences differ in length)
2. Long range (subject-verb agreement across many words)
3. Differentiable (train with gradient descent)
4. Parallel (fast on GPUs)

N-grams fail at (1), (2), (3). CNNs struggle with (1) and (2).

What We Need

A model for sequences must be:

1. Variable length (sentences differ in length)
2. Long range (subject-verb agreement across many words)
3. Differentiable (train with gradient descent)
4. Parallel (fast on GPUs)

N-grams fail at (1), (2), (3). CNNs struggle with (1) and (2).

Two questions remain:

1. How do we represent text as numbers? → Tokenization
2. What architecture processes the result? → Attention

What Does Training Data Look Like?

A language model trains on a massive concatenation of documents:

What Does Training Data Look Like?

A language model trains on a massive concatenation of documents:

```
Once upon a time there was a cat.
<|endoftext|>
The weather today is sunny.
<|endoftext|>
def hello():
    print(”hi”)
<|endoftext|>
```

What Does Training Data Look Like?

A language model trains on a massive concatenation of documents:

```
Once upon a time there was a cat.
<|endoftext|>
The weather today is sunny.
<|endoftext|>
def hello():
    print("hi")
<|endoftext|>
```

- Documents separated by <|endoftext|>

What Does Training Data Look Like?

A language model trains on a massive concatenation of documents:

```
Once upon a time there was a cat.
<|endoftext|>
The weather today is sunny.
<|endoftext|>
def hello():
    print("hi")
<|endoftext|>
```

- Documents separated by <|endoftext|>
- Mixture of English, code, other languages, math...

What Does Training Data Look Like?

A language model trains on a massive concatenation of documents:

```
Once upon a time there was a cat.
<|endoftext|>
The weather today is sunny.
<|endoftext|>
def hello():
    print("hi")
<|endoftext|>
```

- Documents separated by <|endoftext|>
- Mixture of English, code, other languages, math...
- This raw text needs to become a sequence of integers (we're doing sequence modeling!)

Characters vs Words vs Subwords

Characters: "hello" $\rightarrow$ [h, e, l, l, o] $=$ 5 tokens.
Small vocab ($\sim$256). But sequences become very long.

Characters vs Words vs Subwords

Characters: "hello" $\rightarrow$ [h, e, l, l, o] $=$ 5 tokens.
Small vocab ($\sim$256). But sequences become very long.

Words: "hello world" $\rightarrow$ [hello, world] $=$ 2 tokens.
Short sequences.

Characters vs Words vs Subwords

Characters: “hello” $\rightarrow$ [h, e, l, l, o] = 5 tokens.

Small vocab ($\sim$256). But sequences become very long.

Words: “hello world” $\rightarrow$ [hello, world] = 2 tokens.

Short sequences.

But “rizzler” = unknown. Vocab $>$ 100K.

Characters vs Words vs Subwords

Characters: "hello" $\rightarrow$ [h, e, l, l, o] = 5 tokens.
Small vocab ($\sim$256). But sequences become very long.

Words: "hello world" $\rightarrow$ [hello, world] = 2 tokens.
Short sequences.
But "rizzler" = unknown. Vocab > 100K.

Subwords: "unhappiness" $\rightarrow$ [un, happi, ness].
Common words stay whole. Rare words split into known pieces. Tunable vocab size.

Characters vs Words vs Subwords

Characters: "hello" $\to$ [h, e, l, l, o] = 5 tokens.
Small vocab ($\sim$256). But sequences become very long.

Words: "hello world" $\to$ [hello, world] = 2 tokens.
Short sequences.
But "rizzler" = unknown. Vocab > 100K.

Subwords: "unhappiness" $\to$ [un, happi, ness].
Common words stay whole. Rare words split into known pieces. Tunable vocab size.

All modern LLMs use subword tokenization.

Unicode and UTF-8 Bytes

Characters $\rightarrow$ Unicode code points $\rightarrow$ UTF-8 bytes (1 to 4 bytes each).

Unicode and UTF-8 Bytes

Characters → Unicode code points → UTF-8 bytes (1 to 4 bytes each).

```
>>> list("hello".encode("utf-8"))
[104, 101, 108, 108, 111]          # 5 bytes for 5 chars

>>> list("こんにちは".encode("utf-8"))
[227, 129, 147, 227, 130, 147, ...] # 15 bytes for 5 chars
```

Unicode and UTF-8 Bytes

Characters → Unicode code points → UTF-8 bytes (1 to 4 bytes each).

```
>>> list("hello".encode("utf-8"))
[104, 101, 108, 108, 111]          # 5 bytes for 5 chars

>>> list(" こんにちは".encode("utf-8"))
[227, 129, 147, 227, 130, 147, ...] # 15 bytes for 5 chars
```

"こんにちは" = 5 characters but 15 bytes. Each Japanese character uses 3 bytes in UTF-8.

Unicode and UTF-8 Bytes

Characters → Unicode code points → UTF-8 bytes (1 to 4 bytes each).

```
>>> list("hello".encode("utf-8"))
[104, 101, 108, 108, 111]          # 5 bytes for 5 chars

>>> list(" こんにちは".encode("utf-8"))
[227, 129, 147, 227, 130, 147, ...] # 15 bytes for 5 chars
```

"こんにちは" = 5 characters but 15 bytes. Each Japanese character uses 3 bytes in UTF-8.

Key insight: a byte vocabulary of just 256 tokens can represent any text in any language.

Why Not Just Use Bytes?

"hello" = 5 bytes. Fine.

Why Not Just Use Bytes?

"hello" = 5 bytes. Fine.

A 1,000-word essay ≈ 5,000 bytes. A 10-page paper ≈ 50,000 bytes. Too long.

Why Not Just Use Bytes?

"hello" = 5 bytes. Fine.

A 1,000-word essay ≈ 5,000 bytes. A 10-page paper ≈ 50,000 bytes. Too long.

Attention (which we will see soon) costs $O(T^2)$. Doubling T quadruples compute.

Why Not Just Use Bytes?

"hello" = 5 bytes. Fine.

A 1,000-word essay $\approx$ 5,000 bytes. A 10-page paper $\approx$ 50,000 bytes. Too long.

Attention (which we will see soon) costs $O(T^2)$. Doubling T quadruples compute.

We want tokens bigger than bytes but smaller than words:

Why Not Just Use Bytes?

"hello" = 5 bytes. Fine.

A 1,000-word essay $\approx$ 5,000 bytes. A 10-page paper $\approx$ 50,000 bytes. Too long.

Attention (which we will see soon) costs $O(T^2)$. Doubling T quadruples compute.

We want tokens bigger than bytes but smaller than words:

- Common words ("the", "ing") = single token
- Rare words = split into a few pieces

Why Not Just Use Bytes?

"hello" = 5 bytes. Fine.

A 1,000-word essay ≈ 5,000 bytes. A 10-page paper ≈ 50,000 bytes. Too long.

Attention (which we will see soon) costs $O(T^2)$. Doubling T quadruples compute.

We want tokens bigger than bytes but smaller than words:

- Common words ("the", "ing") = single token
- Rare words = split into a few pieces

This is exactly what Byte Pair Encoding (BPE) Tokenizer does.

BPE: The Idea

1. Start with 256 byte tokens

BPE: The Idea

1. Start with 256 byte tokens
2. Find the most frequent adjacent pair in the corpus

BPE: The Idea

1. Start with 256 byte tokens
2. Find the most frequent adjacent pair in the corpus
3. Merge that pair into a new token with new id.
4. Repeat until desired vocab size

BPE: The Idea

1. Start with 256 byte tokens
2. Find the most frequent adjacent pair in the corpus
3. Merge that pair into a new token with new id.
4. Repeat until desired vocab size

Example: if (e, s) is the most frequent pair, merge into a single token "es".

BPE: The Idea

1. Start with 256 byte tokens
2. Find the most frequent adjacent pair in the corpus
3. Merge that pair into a new token with new id.
4. Repeat until desired vocab size

Example: if (e, s) is the most frequent pair, merge into a single token "es".

Every merge adds exactly 1 token. Want 50,257 tokens? Do 50,000 merges.

BPE: The Idea

1. Start with 256 byte tokens
2. Find the most frequent adjacent pair in the corpus
3. Merge that pair into a new token with new id.
4. Repeat until desired vocab size

Example: if (e, s) is the most frequent pair, merge into a single token "es".

Every merge adds exactly 1 token. Want 50,257 tokens? Do 50,000 merges.

Greedy compression: each step maximally compresses the corpus by removing the most frequent pair.

But: We Can't Scan the Whole Corpus Per Step

Training corpus can be hundreds of GB.

But: We Can't Scan the Whole Corpus Per Step

Training corpus can be hundreds of GB.

Counting adjacent pairs over ALL of it for each merge step = impractical.

But: We Can't Scan the Whole Corpus Per Step

Training corpus can be hundreds of GB.

Counting adjacent pairs over ALL of it for each merge step = impractical.

Solution: pre-tokenize first.

But: We Can't Scan the Whole Corpus Per Step

Training corpus can be hundreds of GB.

Counting adjacent pairs over ALL of it for each merge step = impractical.

Solution: pre-tokenize first.

1. Split on <|endoftext|> and other special tokens (prevents cross-document merges)

But: We Can't Scan the Whole Corpus Per Step

Training corpus can be hundreds of GB.

Counting adjacent pairs over ALL of it for each merge step = impractical.

Solution: pre-tokenize first.

1. Split on <|endoftext|> and other special tokens (prevents cross-document merges)
2. Split each piece into "words" using a regex (think of it as doing .split(" ") but smarter)

But: We Can't Scan the Whole Corpus Per Step

Training corpus can be hundreds of GB.

Counting adjacent pairs over ALL of it for each merge step = impractical.

Solution: pre-tokenize first.

1. Split on <|endoftext|> and other special tokens (prevents cross-document merges)
2. Split each piece into "words" using a regex (think of it as doing .split(" ") but smarter)
3. Build a frequency table of these pre-tokens

But: We Can't Scan the Whole Corpus Per Step

Training corpus can be hundreds of GB.

Counting adjacent pairs over ALL of it for each merge step = impractical.

Solution: pre-tokenize first.

1. Split on <|endoftext|> and other special tokens (prevents cross-document merges)
2. Split each piece into "words" using a regex (think of it as doing .split(" ") but smarter)
3. Build a frequency table of these pre-tokens

Result: {" the": 2,000,000, "Hello": 5,000, " world": 50,000, ...}

But: We Can't Scan the Whole Corpus Per Step

Training corpus can be hundreds of GB.

Counting adjacent pairs over ALL of it for each merge step = impractical.

Solution: pre-tokenize first.

1. Split on <|endoftext|> and other special tokens (prevents cross-document merges)
2. Split each piece into "words" using a regex (think of it as doing .split(" ") but smarter)
3. Build a frequency table of these pre-tokens

Result: {" the": 2,000,000, "Hello": 5,000, " world": 50,000, ...}
This table fits in memory. Now run BPE on this compact table.

Pre-tokenization in Code

```
>>> import re
>>> PAT = r"'s|'t|'re|'ve|'m|'ll|'d| ?\w+| ?[^\s\w]+|\s+(?!\S)|\s+"
>>> re.findall(PAT, "Hello world! It's a test.")
['Hello', ' world', '!', " It", "'s", ' a', ' test', '.']
```

Pre-tokenization in Code

```
>>> import re
>>> PAT = r"'s|'t|'re|'ve|'m|'ll|'d| ?\w+| ?[^\s\w]+|\s+(?!\S)|\s+"
>>> re.findall(PAT, "Hello world! It's a test.")
['Hello', ' world', '!', " It", "'s", ' a', ' test', '.']
```

Contractions split off: “It’s” → “ It” + “’s”

Pre-tokenization in Code

```
>>> import re
>>> PAT = r"'s|'t|'re|'ve|'m|'ll|'d| ?\w+| ?[^\s\w]+|\s+(?!\S)|\s+"
>>> re.findall(PAT, "Hello world! It's a test.")
['Hello', ' world', '!', " It", "'s", ' a', ' test', '.']
```

Contractions split off: "It's" → " It" + "'s"

Spaces attach to the next word: " world", " test"

Pre-tokenization in Code

```
>>> import re
>>> PAT = r"'s|'t|'re|'ve|'m|'ll|'d| ?\w+| ?[^\s\w]+|\s+(?!\S)|\s+"
>>> re.findall(PAT, "Hello world! It's a test.")
['Hello', ' world', '!', " It", "'s", ' a', ' test', '.']
```

Contractions split off: "It's" → " It" + "'s"

Spaces attach to the next word: " world", " test"

Each pre-token is converted to its UTF-8 bytes. The frequency table:

Pre-tokenization in Code

```
>>> import re
>>> PAT = r"'s|'t|'re|'ve|'m|'ll|'d| ?\w+| ?[^\s\w]+|\s+(?!\S)|\s+"
>>> re.findall(PAT, "Hello world! It's a test.")
['Hello', ' world', '!', " It", "'s", ' a', ' test', '.']
```

Contractions split off: "It's" → " It" + "'s"

Spaces attach to the next word: " world", " test"

Each pre-token is converted to its UTF-8 bytes. The frequency table:

{" the": 2M, " of": 1.1M, "Hello": 5K, " world": 50K, ...}

Pre-tokenization in Code

```
>>> import re
>>> PAT = r"'s|'t|'re|'ve|'m|'ll|'d| ?\w+| ?[^\s\w]+|\s+(?!\S)|\s+"
>>> re.findall(PAT, "Hello world! It's a test.")
['Hello', ' world', '!', " It", "'s", ' a', ' test', '.']
```

Contractions split off: "It's" → " It" + "'s"

Spaces attach to the next word: " world", " test"

Each pre-token is converted to its UTF-8 bytes. The frequency table:

{" the": 2M, " of": 1.1M, "Hello": 5K, " world": 50K, ...}

Now we can run BPE on this compact table instead of the raw corpus.

BPE Worked Example: Setup

Frequency table after pre-tokenization:

BPE Worked Example: Setup

Frequency table after pre-tokenization:

Pre-token (as bytes)	Frequency
l o w	5
l o w e r	2
w i d e s t	3
n e w e s t	6

BPE Worked Example: Setup

Frequency table after pre-tokenization:

Pre-token (as bytes)	Frequency
l o w	5
l o w e r	2
w i d e s t	3
n e w e s t	6

Initial vocabulary: {l, o, w, e, r, i, d, s, t, n}

BPE Worked Example: Setup

Frequency table after pre-tokenization:

Pre-token (as bytes)	Frequency
l o w	5
l o w e r	2
w i d e s t	3
n e w e s t	6

Initial vocabulary: {l, o, w, e, r, i, d, s, t, n}

Count all adjacent pairs, weighted by frequency.

BPE Worked Example: Merge 1

Adjacent pair counts (each weighted by the pre-token frequency):

BPE Worked Example: Merge 1

Adjacent pair counts (each weighted by the pre-token frequency):

(l,o): 7	(o,w): 7	(w,e): 8
(e,r): 2	(w,i): 3	(i,d): 3
(d,e): 3	(e,s): 9	(s,t): 9
(n,e): 6	(e,w): 6	

BPE Worked Example: Merge 1

Adjacent pair counts (each weighted by the pre-token frequency):

(l,o): 7	(o,w): 7	(w,e): 8
(e,r): 2	(w,i): 3	(i,d): 3
(d,e): 3	(e,s): 9	(s,t): 9
(n,e): 6	(e,w): 6	

Tie at 9 between (e,s) and (s,t). Lexicographic tiebreak → pick (s,t).

BPE Worked Example: Merge 1

Adjacent pair counts (each weighted by the pre-token frequency):

(l,o): 7	(o,w): 7	(w,e): 8
(e,r): 2	(w,i): 3	(i,d): 3
(d,e): 3	(e,s): 9	(s,t): 9
(n,e): 6	(e,w): 6	

Tie at 9 between (e,s) and (s,t). Lexicographic tiebreak → pick (s,t).

Merge (s,t) → st. Updated table:

BPE Worked Example: Merge 1

Adjacent pair counts (each weighted by the pre-token frequency):

(l,o): 7	(o,w): 7	(w,e): 8
(e,r): 2	(w,i): 3	(i,d): 3
(d,e): 3	(e,s): 9	(s,t): 9
(n,e): 6	(e,w): 6	

Tie at 9 between (e,s) and (s,t). Lexicographic tiebreak → pick (s,t).

Merge (s,t) → st. Updated table:

l o w	5
l o w e r	2
w i d e st	3
n e w e st	6

BPE Worked Example: Merges 2–3

After merge 1: pair (e, st) now appears $3 + 6 = 9$ times.

BPE Worked Example: Merges 2–3

After merge 1: pair (e, st) now appears $3 + 6 = 9$ times.

Merge 2: (e, st) $\rightarrow$ est.

BPE Worked Example: Merges 2–3

After merge 1: pair (e, st) now appears $3 + 6 = 9$ times.

Merge 2: (e, st) $\rightarrow$ est.

l o w	5
l o w e r	2
w i d est	3
n e w est	6

BPE Worked Example: Merges 2–3

After merge 1: pair (e, st) now appears $3 + 6 = 9$ times.

Merge 2: (e, st) $\rightarrow$ est.

l o w	5
l o w e r	2
w i d est	3
n e w est	6

Now (l, o) appears $5 + 2 = 7$ times (highest remaining).

BPE Worked Example: Merges 2–3

After merge 1: pair (e, st) now appears $3 + 6 = 9$ times.

Merge 2: (e, st) $\rightarrow$ est.

l o w	5
l o w e r	2
w i d est	3
n e w est	6

Now (l, o) appears $5 + 2 = 7$ times (highest remaining).
Merge 3: (l, o) $\rightarrow$ lo.

BPE Worked Example: Merges 2–3

After merge 1: pair (e, st) now appears $3 + 6 = 9$ times.

Merge 2: (e, st) $\rightarrow$ est.

l o w	5
l o w e r	2
w i d est	3
n e w est	6

Now (l, o) appears $5 + 2 = 7$ times (highest remaining).
Merge 3: (l, o) $\rightarrow$ lo.

lo w	5
lo w e r	2
w i d est	3
n e w est	6

BPE Worked Example: Merges 2–3

After merge 1: pair (e, st) now appears $3 + 6 = 9$ times.

Merge 2: (e, st) $\rightarrow$ est.

l o w	5
l o w e r	2
w i d est	3
n e w est	6

Now (l, o) appears $5 + 2 = 7$ times (highest remaining).
Merge 3: (l, o) $\rightarrow$ lo.

lo w	5
lo w e r	2
w i d est	3
n e w est	6

Vocabulary after 3 merges: {l, o, w, e, r, i, d, s, t, n} $\cup$ {st, est, lo}

Encoding: Using the Tokenizer

Given text and the ordered merge list, apply merges in order:

Encoding: Using the Tokenizer

Given text and the ordered merge list, apply merges in order:

Encode “lowest”:

Encoding: Using the Tokenizer

Given text and the ordered merge list, apply merges in order:

Encode “lowest”:

Start: [l, o, w, e, s, t]

Encoding: Using the Tokenizer

Given text and the ordered merge list, apply merges in order:

Encode “lowest”:

Start: [l, o, w, e, s, t]
Apply merge 1 (s,t $\rightarrow$ st): [l, o, w, e, st]

Encoding: Using the Tokenizer

Given text and the ordered merge list, apply merges in order:

Encode “lowest”:

Start: [l, o, w, e, s, t]
Apply merge 1 (s,t $\rightarrow$ st): [l, o, w, e, st]
Apply merge 2 (e,st $\rightarrow$ est): [l, o, w, est]

Encoding: Using the Tokenizer

Given text and the ordered merge list, apply merges in order:

Encode "lowest":

Start: [l, o, w, e, s, t]
Apply merge 1 (s,t $\rightarrow$ st): [l, o, w, e, st]
Apply merge 2 (e,st $\rightarrow$ est): [l, o, w, est]
Apply merge 3 (l,o $\rightarrow$ lo): [lo, w, est]

Encoding: Using the Tokenizer

Given text and the ordered merge list, apply merges in order:

Encode “lowest”:

Start: [l, o, w, e, s, t]
Apply merge 1 (s,t $\rightarrow$ st): [l, o, w, e, st]
Apply merge 2 (e,st $\rightarrow$ est): [l, o, w, est]
Apply merge 3 (l,o $\rightarrow$ lo): [lo, w, est]
Apply merge 4 (lo,w $\rightarrow$ low): [low, est]

Encoding: Using the Tokenizer

Given text and the ordered merge list, apply merges in order:

Encode "lowest":

Start: [l, o, w, e, s, t]
Apply merge 1 (s,t $\rightarrow$ st): [l, o, w, e, st]
Apply merge 2 (e,st $\rightarrow$ est): [l, o, w, est]
Apply merge 3 (l,o $\rightarrow$ lo): [lo, w, est]
Apply merge 4 (lo,w $\rightarrow$ low): [low, est]

Final tokens: [low, est]. Look up each in the vocabulary $\rightarrow$ integer IDs.

Encoding: Using the Tokenizer

Given text and the ordered merge list, apply merges in order:

Encode "lowest":

Start: [l, o, w, e, s, t]
Apply merge 1 (s,t $\rightarrow$ st): [l, o, w, e, st]
Apply merge 2 (e,st $\rightarrow$ est): [l, o, w, est]
Apply merge 3 (l,o $\rightarrow$ lo): [lo, w, est]
Apply merge 4 (lo,w $\rightarrow$ low): [low, est]

Final tokens: [low, est]. Look up each in the vocabulary $\rightarrow$ integer IDs.

Decoding: IDs $\rightarrow$ byte sequences $\rightarrow$ concatenate $\rightarrow$ UTF-8 decode. No ambiguity.

Special Tokens

<|endoftext|>: marks document boundaries. Never split by BPE. Fixed ID.

Special Tokens

<|endoftext|>: marks document boundaries. Never split by BPE. Fixed ID.

Other examples: <|start|>, <thinking>, </thinking>

Special Tokens

<|endoftext|>: marks document boundaries. Never split by BPE. Fixed ID.

Other examples: <|start|>, <thinking>, </thinking>

During encoding:

Special Tokens

<|endoftext|>: marks document boundaries. Never split by BPE. Fixed ID.

Other examples: <|start|>, <thinking>, </thinking>

During encoding:

1. Split on special tokens first

Special Tokens

<|endoftext|>: marks document boundaries. Never split by BPE. Fixed ID.

Other examples: <|start|>, <thinking>, </thinking>

During encoding:

1. Split on special tokens first
2. Apply the regex pre-tokenizer to each piece
3. Apply BPE merges

Special Tokens

<|endoftext|>: marks document boundaries. Never split by BPE. Fixed ID.

Other examples: <|start|>, <thinking>, </thinking>

During encoding:

1. Split on special tokens first
2. Apply the regex pre-tokenizer to each piece
3. Apply BPE merges

This prevents merges from crossing document boundaries.

Special Tokens

<|endoftext|>: marks document boundaries. Never split by BPE. Fixed ID.

Other examples: <|start|>, <thinking>, </thinking>

During encoding:

1. Split on special tokens first
2. Apply the regex pre-tokenizer to each piece
3. Apply BPE merges

This prevents merges from crossing document boundaries.

Special tokens are added to the vocabulary before BPE training.

Special Tokens

<|endoftext|>: marks document boundaries. Never split by BPE. Fixed ID.

Other examples: <|start|>, <thinking>, </thinking>

During encoding:

1. Split on special tokens first
2. Apply the regex pre-tokenizer to each piece
3. Apply BPE merges

This prevents merges from crossing document boundaries.

Special tokens are added to the vocabulary before BPE training.
They are never split or merged during encoding.

What About Non-English Text?

Consider the Japanese word “こんにちは” (hello).

What About Non-English Text?

Consider the Japanese word “こんにちは” (hello).

The pre-tokenizer works on text (Unicode), not bytes. The regex sees this as one word.

What About Non-English Text?

Consider the Japanese word “こんにちは” (hello).

The pre-tokenizer works on text (Unicode), not bytes. The regex sees this as one word.

Then it gets converted to bytes: 15 bytes (3 per character).

What About Non-English Text?

Consider the Japanese word “こんにちは” (hello).

The pre-tokenizer works on text (Unicode), not bytes. The regex sees this as one word.

Then it gets converted to bytes: 15 bytes (3 per character).

Initially, each byte is a separate token: [227, 129, 147, 227, 130, 147, ...]

What About Non-English Text?

Consider the Japanese word “こんにちは” (hello).

The pre-tokenizer works on text (Unicode), not bytes. The regex sees this as one word.

Then it gets converted to bytes: 15 bytes (3 per character).

Initially, each byte is a separate token: [227, 129, 147, 227, 130, 147, ...]

If the training corpus has a lot of Japanese, BPE will merge these bytes into larger tokens. Eventually, common Japanese characters or words become single tokens.

What About Non-English Text?

Consider the Japanese word “こんにちは” (hello).

The pre-tokenizer works on text (Unicode), not bytes. The regex sees this as one word.

Then it gets converted to bytes: 15 bytes (3 per character).

Initially, each byte is a separate token: [227, 129, 147, 227, 130, 147, ...]

If the training corpus has a lot of Japanese, BPE will merge these bytes into larger tokens. Eventually, common Japanese characters or words become single tokens.

If the corpus has little Japanese, the bytes stay unmerged. Japanese text uses $\sim$3 tokens per character instead of $\sim$1. This is why multilingual tokenizers need large vocabularies (100K+).

What About Non-English Text?

Consider the Japanese word "こんにちは" (hello).

The pre-tokenizer works on text (Unicode), not bytes. The regex sees this as one word.

Then it gets converted to bytes: 15 bytes (3 per character).

Initially, each byte is a separate token: [227, 129, 147, 227, 130, 147, ...]

If the training corpus has a lot of Japanese, BPE will merge these bytes into larger tokens. Eventually, common Japanese characters or words become single tokens.

If the corpus has little Japanese, the bytes stay unmerged. Japanese text uses $\sim$3 tokens per character instead of $\sim$1. This is why multilingual tokenizers need large vocabularies (100K+).

Takeaway: the byte-level starting point means BPE handles any language. Quality depends on how much of that language is in the training corpus.

Loading a Pre-Trained Tokenizer

You can load GPT-4's tokenizer locally using the tiktoken library:

Loading a Pre-Trained Tokenizer

You can load GPT-4's tokenizer locally using the tiktoken library:

```
import tiktoken
enc = tiktoken.get_encoding("cl100k_base")  # GPT-4 tokenizer
print(enc.encode("Hello world!")) #[9906, 1917, 0]
print(enc.decode([9906, 1917, 0])) #'Hello world!'
```

Loading a Pre-Trained Tokenizer

You can load GPT-4's tokenizer locally using the tiktoken library:

```
import tiktoken
enc = tiktoken.get_encoding("cl100k_base")  # GPT-4 tokenizer
print(enc.encode("Hello world!")) #[9906, 1917, 0]
print(enc.decode([9906, 1917, 0])) #'Hello world!'
```

```
print(enc.n_vocab) #100277
print(enc.encode("<|endoftext|>", allowed_special={"<|endoftext|>"})) #100257
```

Loading a Pre-Trained Tokenizer

You can load GPT-4's tokenizer locally using the tiktoken library:

```
import tiktoken
enc = tiktoken.get_encoding("cl100k_base")  # GPT-4 tokenizer
print(enc.encode("Hello world!")) #[9906, 1917, 0]
print(enc.decode([9906, 1917, 0])) #'Hello world!'
```

```
print(enc.n_vocab) #100277
print(enc.encode("<|endoftext|>", allowed_special={"<|endoftext|>"})) #100257
```

100,256 vocabulary tokens. <|endoftext|> gets its own fixed ID (100257). You can inspect any tokenizer this way: pip install tiktoken.

Tokenization Summary

The full pipeline:

Tokenization Summary

The full pipeline:

Raw Text → Split on Special Tokens → Pre-tokenize → To Bytes → BPE Merge → Token ID

Tokenization Summary

The full pipeline:

Raw Text → Split on Special Tokens → Pre-tokenize → To Bytes → BPE Merge → Token ID

Vocab size is a hyperparameter: GPT-2 = 50,257, GPT-4 = 100,256.

Tokenization Summary

The full pipeline:

Raw Text → Split on Special Tokens → Pre-tokenize → To Bytes → BPE Merge → Token ID

Vocab size is a hyperparameter: GPT-2 = 50,257, GPT-4 = 100,256.

The tokenizer is trained before the language model and is a fixed preprocessing step.

Tokenization Summary

The full pipeline:

Raw Text → Split on Special Tokens → Pre-tokenize → To Bytes → BPE Merge → Token ID

Vocab size is a hyperparameter: GPT-2 = 50,257, GPT-4 = 100,256.

The tokenizer is trained before the language model and is a fixed preprocessing step.

Now we can convert any text to integer sequences. What architecture processes them?

From Token IDs to Vectors

A token ID is just an integer (an index). Neural networks need vectors.

From Token IDs to Vectors

A token ID is just an integer (an index). Neural networks need vectors.

Embedding table: a matrix E of shape [vocab_size, d].

From Token IDs to Vectors

A token ID is just an integer (an index). Neural networks need vectors.

Embedding table: a matrix E of shape [vocab_size, d].

Token ID 464 $\rightarrow$ row 464 of E $\rightarrow$ a vector in $\mathbb{R}^{d}$.

From Token IDs to Vectors

A token ID is just an integer (an index). Neural networks need vectors.

Embedding table: a matrix E of shape [vocab_size, d].

Token ID 464 $\rightarrow$ row 464 of E $\rightarrow$ a vector in $\mathbb{R}^{d}$.

This is a lookup: index in, vector out.

From Token IDs to Vectors

A token ID is just an integer (an index). Neural networks need vectors.

Embedding table: a matrix E of shape [vocab_size, d].

Token ID 464 $\rightarrow$ row 464 of E $\rightarrow$ a vector in $\mathbb{R}^{d}$.

This is a lookup: index in, vector out.

E is learned during training. Initially random, but after training, similar words end up with similar vectors.

From Token IDs to Vectors

A token ID is just an integer (an index). Neural networks need vectors.

Embedding table: a matrix E of shape [vocab_size, d].

Token ID 464 $\rightarrow$ row 464 of E $\rightarrow$ a vector in $\mathbb{R}^d$.

This is a lookup: index in, vector out.

E is learned during training. Initially random, but after training, similar words end up with similar vectors.

Now we have: text $\rightarrow$ token IDs $\rightarrow$ sequence of vectors in $\mathbb{R}^d$.

How Should Positions Communicate?

We have T vectors. How do we process them?

How Should Positions Communicate?

We have T vectors. How do we process them?

MLP: each position independently. Position 3 knows nothing about positions 1 and 2.

How Should Positions Communicate?

We have T vectors. How do we process them?

MLP: each position independently. Position 3 knows nothing about positions 1 and 2.

CNN: local window only (kernel size 3). To see 100 positions away, need $\sim$33 layers.

How Should Positions Communicate?

We have T vectors. How do we process them?

MLP: each position independently. Position 3 knows nothing about positions 1 and 2.

CNN: local window only (kernel size 3). To see 100 positions away, need $\sim$33 layers.

What we want: every position looks at every other position directly, in a single step.

How Should Positions Communicate?

We have T vectors. How do we process them?

MLP: each position independently. Position 3 knows nothing about positions 1 and 2.

CNN: local window only (kernel size 3). To see 100 positions away, need ∼33 layers.

What we want: every position looks at every other position directly, in a single step.

This is attention.

The "it" Example

"The cat sat on the mat because it was soft."

The "it" Example

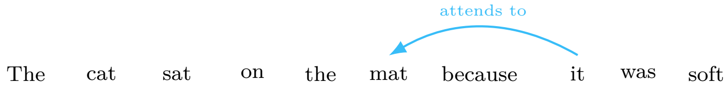
"The cat sat on the mat because it was soft."

What does "it" refer to? "mat" (because "soft" describes mats, not cats).

The “it” Example

“The cat sat on the mat because it was soft.”

What does “it” refer to? “mat” (because “soft” describes mats, not cats).



The "it" Example

"The cat sat on the mat because it was soft."

What does "it" refer to? "mat" (because "soft" describes mats, not cats).

attends to

The cat sat on the mat because it was soft

The model needs "it" at position 8 to attend strongly to "mat" at position 6.

The "it" Example

"The cat sat on the mat because it was soft."

What does "it" refer to? "mat" (because "soft" describes mats, not cats).

attends to

The cat sat on the mat because it was soft

The model needs "it" at position 8 to attend strongly to "mat" at position 6. The relevant word could be anywhere in the sentence. Attention lets each token decide, based on content, what to focus on.

Not Just Language

Attention is a general mechanism:

Not Just Language

Attention is a general mechanism:

- Image captioning: attend to the dog region when generating "dog"

Not Just Language

Attention is a general mechanism:

- Image captioning: attend to the dog region when generating "dog"
- Protein folding: amino acid 50 interacts with amino acid 200 (AlphaFold)

Not Just Language

Attention is a general mechanism:

- Image captioning: attend to the dog region when generating "dog"
- Protein folding: amino acid 50 interacts with amino acid 200 (AlphaFold)
- Recommendations: which past purchases predict the next one?

Not Just Language

Attention is a general mechanism:

- Image captioning: attend to the dog region when generating "dog"
- Protein folding: amino acid 50 interacts with amino acid 200 (AlphaFold)
- Recommendations: which past purchases predict the next one?

Core idea: dynamically weight inputs based on content.

The Database Analogy

Think of attention like a fuzzy database lookup.

The Database Analogy

Think of attention like a fuzzy database lookup.

Regular database: "Find customer with ID 42." $\rightarrow$ exact match, one row.

The Database Analogy

Think of attention like a fuzzy database lookup.

Regular database: "Find customer with ID 42." $\rightarrow$ exact match, one row.

Attention:

The Database Analogy

Think of attention like a fuzzy database lookup.

Regular database: "Find customer with ID 42." $\rightarrow$ exact match, one row.

Attention:

Query (Q): "What am I looking for?"

The Database Analogy

Think of attention like a fuzzy database lookup.

Regular database: "Find customer with ID 42." → exact match, one row.

Attention:

- Query (Q): "What am I looking for?"
- Key (K): "What do I contain?" (each item advertises itself)

The Database Analogy

Think of attention like a fuzzy database lookup.

Regular database: “Find customer with ID 42.” $\rightarrow$ exact match, one row.

Attention:

- Query (Q): “What am I looking for?”
- Key (K): “What do I contain?” (each item advertises itself)
- Value (V): “What information do I carry?” (the actual content)

The Database Analogy

Think of attention like a fuzzy database lookup.

Regular database: "Find customer with ID 42." → exact match, one row.

Attention:

- Query (Q): "What am I looking for?"
- Key (K): "What do I contain?" (each item advertises itself)
- Value (V): "What information do I carry?" (the actual content)

Compare query against all keys → compatibility scores → weighted average of values.

The Database Analogy

Think of attention like a fuzzy database lookup.

Regular database: "Find customer with ID 42." → exact match, one row.

Attention:

- Query (Q): "What am I looking for?"
- Key (K): "What do I contain?" (each item advertises itself)
- Value (V): "What information do I carry?" (the actual content)

Compare query against all keys → compatibility scores → weighted average of values.

Instead of returning one row, return a weighted combination of all rows.

Let's Build It With Numbers

3 tokens: "cat", "sat", "mat". Embedding dimension $d = 4$.

Let's Build It With Numbers

3 tokens: "cat", "sat", "mat". Embedding dimension $d = 4$.

Token	x_1	x_2	x_3	x_4
"cat"	1.0	0.0	1.0	0.0
"sat"	0.0	1.0	0.0	1.0
"mat"	1.0	0.0	0.5	0.5

Let's Build It With Numbers

3 tokens: "cat", "sat", "mat". Embedding dimension d = 4.

Token	x_1	x_2	x_3	x_4
"cat"	1.0	0.0	1.0	0.0
"sat"	0.0	1.0	0.0	1.0
"mat"	1.0	0.0	0.5	0.5

"cat" and "mat" are similar (both nouns). "sat" is different (verb).

Let's Build It With Numbers

3 tokens: "cat", "sat", "mat". Embedding dimension $d = 4$.

Token	x_1	x_2	x_3	x_4
"cat"	1.0	0.0	1.0	0.0
"sat"	0.0	1.0	0.0	1.0
"mat"	1.0	0.0	0.5	0.5

"cat" and "mat" are similar (both nouns). "sat" is different (verb).

Goal: each token should gather relevant info from other tokens.

Let's Build It With Numbers

3 tokens: "cat", "sat", "mat". Embedding dimension d = 4.

Token	x_1	x_2	x_3	x_4
"cat"	1.0	0.0	1.0	0.0
"sat"	0.0	1.0	0.0	1.0
"mat"	1.0	0.0	0.5	0.5

"cat" and "mat" are similar (both nouns). "sat" is different (verb).

Goal: each token should gather relevant info from other tokens.
But not equally. A noun should attend more to other nouns than to verbs.

Step 1: Compute Q, K, V

Three learned weight matrices W_Q, W_K, W_V, each 4×2 (projecting $d = 4 \to d_k = 2$).

Step 1: Compute Q, K, V

Three learned weight matrices W_Q, W_K, W_V, each 4×2 (projecting $d = 4 \to d_k = 2$).

$$W_Q = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}, \quad W_K = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad W_V = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Step 1: Compute Q, K, V

Three learned weight matrices W_Q, W_K, W_V, each 4×2 (projecting $d = 4 \to d_k = 2$).

$$W_Q = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}, \quad W_K = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad W_V = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Multiply each embedding by W_Q, W_K, W_V:

Step 1: Compute Q, K, V

Three learned weight matrices W_Q, W_K, W_V, each 4×2 (projecting $d = 4 \to d_k = 2$).

$$W_Q = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}, \quad W_K = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad W_V = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Multiply each embedding by W_Q, W_K, W_V:

Token	q_1	q_2	k_1	k_2	v_1	v_2
"cat"	1.0	0.0	1.0	0.0	2.0	0.0
"sat"	0.0	1.0	0.0	1.0	0.0	2.0
"mat"	1.0	0.0	0.5	0.5	1.5	0.5

Step 1: Compute Q, K, V

Three learned weight matrices W_Q, W_K, W_V, each 4×2 (projecting $d = 4 \to d_k = 2$).

$$W_Q = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}, \quad W_K = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad W_V = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Multiply each embedding by W_Q, W_K, W_V:

Token	q_1	q_2	k_1	k_2	v_1	v_2
"cat"	1.0	0.0	1.0	0.0	2.0	0.0
"sat"	0.0	1.0	0.0	1.0	0.0	2.0
"mat"	1.0	0.0	0.5	0.5	1.5	0.5

W_Q extracts features 1–2 as the query. W_K extracts features 3–4 as the key.

Step 1: Compute Q, K, V

Three learned weight matrices W_Q, W_K, W_V, each 4×2 (projecting $d = 4 \to d_k = 2$).

$$W_Q = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}, \quad W_K = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad W_V = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Multiply each embedding by W_Q, W_K, W_V:

Token	q_1	q_2	k_1	k_2	v_1	v_2
"cat"	1.0	0.0	1.0	0.0	2.0	0.0
"sat"	0.0	1.0	0.0	1.0	0.0	2.0
"mat"	1.0	0.0	0.5	0.5	1.5	0.5

W_Q extracts features 1–2 as the query. W_K extracts features 3–4 as the key.

Step 2: Attention Scores

$\mathrm{score}(\mathrm{i},\mathrm{j}) = \mathbf{q}_\mathrm{i} \cdot \mathbf{k}_\mathrm{j}$. Compute all 9 dot products:

Step 2: Attention Scores

$\text{score}(i, j) = \mathbf{q}_i \cdot \mathbf{k}_j$. Compute all 9 dot products:

$\mathbf{q}_i \cdot \mathbf{k}_j$	cat (k)	sat (k)	mat (k)
cat (q)	$1{\cdot}1 + 0{\cdot}0 = \mathbf{1.0}$	$1{\cdot}0 + 0{\cdot}1 = \mathbf{0.0}$	$1{\cdot}0.5 + 0{\cdot}0.5 = \mathbf{0.5}$

Step 2: Attention Scores

$\text{score}(i, j) = \mathbf{q}_i \cdot \mathbf{k}_j$. Compute all 9 dot products:

$\mathbf{q}_i \cdot \mathbf{k}_j$	cat (k)	sat (k)	mat (k)
cat (q)	$1{\cdot}1 + 0{\cdot}0 = \mathbf{1.0}$	$1{\cdot}0 + 0{\cdot}1 = \mathbf{0.0}$	$1{\cdot}0.5 + 0{\cdot}0.5 = \mathbf{0.5}$
sat (q)	$0{\cdot}1 + 1{\cdot}0 = \mathbf{0.0}$	$0{\cdot}0 + 1{\cdot}1 = \mathbf{1.0}$	$0{\cdot}0.5 + 1{\cdot}0.5 = \mathbf{0.5}$

Step 2: Attention Scores

$\text{score}(i, j) = \mathbf{q}_i \cdot \mathbf{k}_j$. Compute all 9 dot products:

$\mathbf{q}_i \cdot \mathbf{k}_j$	cat (k)	sat (k)	mat (k)
cat (q)	$1{\cdot}1 + 0{\cdot}0 = \mathbf{1.0}$	$1{\cdot}0 + 0{\cdot}1 = \mathbf{0.0}$	$1{\cdot}0.5 + 0{\cdot}0.5 = \mathbf{0.5}$
sat (q)	$0{\cdot}1 + 1{\cdot}0 = \mathbf{0.0}$	$0{\cdot}0 + 1{\cdot}1 = \mathbf{1.0}$	$0{\cdot}0.5 + 1{\cdot}0.5 = \mathbf{0.5}$
mat (q)	$1{\cdot}1 + 0{\cdot}0 = \mathbf{1.0}$	$1{\cdot}0 + 0{\cdot}1 = \mathbf{0.0}$	$1{\cdot}0.5 + 0{\cdot}0.5 = \mathbf{0.5}$

Step 2: Attention Scores

$\text{score}(i, j) = \mathbf{q}_i \cdot \mathbf{k}_j$. Compute all 9 dot products:

$\mathbf{q}_i \cdot \mathbf{k}_j$	cat (k)	sat (k)	mat (k)
cat (q)	$1{\cdot}1 + 0{\cdot}0 = \mathbf{1.0}$	$1{\cdot}0 + 0{\cdot}1 = \mathbf{0.0}$	$1{\cdot}0.5 + 0{\cdot}0.5 = \mathbf{0.5}$
sat (q)	$0{\cdot}1 + 1{\cdot}0 = \mathbf{0.0}$	$0{\cdot}0 + 1{\cdot}1 = \mathbf{1.0}$	$0{\cdot}0.5 + 1{\cdot}0.5 = \mathbf{0.5}$
mat (q)	$1{\cdot}1 + 0{\cdot}0 = \mathbf{1.0}$	$1{\cdot}0 + 0{\cdot}1 = \mathbf{0.0}$	$1{\cdot}0.5 + 0{\cdot}0.5 = \mathbf{0.5}$

"cat" has highest score with "cat" (1.0) and second highest with "mat" (0.5). Nouns match nouns.

Step 3: Softmax → Weights

Apply softmax to each row to get probabilities.

Step 3: Softmax → Weights

Apply softmax to each row to get probabilities.

Row for "cat": scores $[1.0, 0.0, 0.5]$

Step 3: Softmax $\rightarrow$ Weights

Apply softmax to each row to get probabilities.

Row for "cat": scores $[1.0, 0.0, 0.5]$

$$\exp([1.0, 0.0, 0.5]) = [2.72, 1.00, 1.65], \quad \text{sum} = 5.37$$

Step 3: Softmax → Weights

Apply softmax to each row to get probabilities.

Row for "cat": scores $[1.0, 0.0, 0.5]$

$$\exp([1.0, 0.0, 0.5]) = [2.72, 1.00, 1.65], \quad \text{sum} = 5.37$$

$$\text{weights} = [2.72/5.37,\ 1.00/5.37,\ 1.65/5.37] \approx [\mathbf{0.51},\ 0.19,\ 0.31]$$

Step 3: Softmax $\to$ Weights

Apply softmax to each row to get probabilities.

Row for "cat": scores $[1.0, 0.0, 0.5]$

$$\exp([1.0, 0.0, 0.5]) = [2.72, 1.00, 1.65], \quad \text{sum} = 5.37$$

$$\text{weights} = [2.72/5.37,\ 1.00/5.37,\ 1.65/5.37] \approx [\mathbf{0.51},\ 0.19,\ 0.31]$$

All three rows:

Step 3: Softmax → Weights

Apply softmax to each row to get probabilities.

Row for "cat": scores $[1.0, 0.0, 0.5]$

$$\exp([1.0, 0.0, 0.5]) = [2.72, 1.00, 1.65], \quad \text{sum} = 5.37$$

$$\text{weights} = [2.72/5.37,\ 1.00/5.37,\ 1.65/5.37] \approx [\mathbf{0.51},\ 0.19,\ 0.31]$$

All three rows:

Weights α_{ij}	cat	sat	mat
cat →	0.51	0.19	0.31
sat →	0.19	0.51	0.31
mat →	0.51	0.19	0.31

Step 3: Softmax → Weights

Apply softmax to each row to get probabilities.

Row for "cat": scores $[1.0, 0.0, 0.5]$

$$\exp([1.0, 0.0, 0.5]) = [2.72, 1.00, 1.65], \quad \text{sum} = 5.37$$

$$\text{weights} = [2.72/5.37,\ 1.00/5.37,\ 1.65/5.37] \approx [\mathbf{0.51},\ 0.19,\ 0.31]$$

All three rows:

Weights α_{ij}	cat	sat	mat
cat →	0.51	0.19	0.31
sat →	0.19	0.51	0.31
mat →	0.51	0.19	0.31

"cat" attends 51% to itself, 31% to "mat", 19% to "sat". Nouns attend to nouns.

Step 4: Compute Output

Output = weighted sum of value vectors.

Step 4: Compute Output

Output = weighted sum of value vectors.

Output for "cat" $= 0.51 \cdot \mathbf{v}_{\text{cat}} + 0.19 \cdot \mathbf{v}_{\text{sat}} + 0.31 \cdot \mathbf{v}_{\text{mat}}$

Step 4: Compute Output

Output = weighted sum of value vectors.

$$\text{Output for "cat"} = 0.51 \cdot \mathbf{v}_{\text{cat}} + 0.19 \cdot \mathbf{v}_{\text{sat}} + 0.31 \cdot \mathbf{v}_{\text{mat}}$$

$$= 0.51 \cdot (2.0, 0.0) + 0.19 \cdot (0.0, 2.0) + 0.31 \cdot (1.5, 0.5)$$

Step 4: Compute Output

Output = weighted sum of value vectors.

Output for "cat" $= 0.51 \cdot \mathbf{v}_{\text{cat}} + 0.19 \cdot \mathbf{v}_{\text{sat}} + 0.31 \cdot \mathbf{v}_{\text{mat}}$

$= 0.51 \cdot (2.0, 0.0) + 0.19 \cdot (0.0, 2.0) + 0.31 \cdot (1.5, 0.5)$

$= (1.02, 0.0) + (0.0, 0.38) + (0.47, 0.16) = (\mathbf{1.48}, \mathbf{0.53})$

Step 4: Compute Output

Output = weighted sum of value vectors.

Output for "cat" $= 0.51 \cdot \mathbf{v}_{\text{cat}} + 0.19 \cdot \mathbf{v}_{\text{sat}} + 0.31 \cdot \mathbf{v}_{\text{mat}}$

$$= 0.51 \cdot (2.0, 0.0) + 0.19 \cdot (0.0, 2.0) + 0.31 \cdot (1.5, 0.5)$$
$$= (1.02, 0.0) + (0.0, 0.38) + (0.47, 0.16) = (\mathbf{1.48}, \mathbf{0.53})$$

Output for "sat" $= 0.19 \cdot (2.0, 0.0) + 0.51 \cdot (0.0, 2.0) + 0.31 \cdot (1.5, 0.5)$

Step 4: Compute Output

Output = weighted sum of value vectors.

Output for "cat" $= 0.51 \cdot \mathbf{v}_{\text{cat}} + 0.19 \cdot \mathbf{v}_{\text{sat}} + 0.31 \cdot \mathbf{v}_{\text{mat}}$

$$= 0.51 \cdot (2.0, 0.0) + 0.19 \cdot (0.0, 2.0) + 0.31 \cdot (1.5, 0.5)$$
$$= (1.02, 0.0) + (0.0, 0.38) + (0.47, 0.16) = (\mathbf{1.48}, \mathbf{0.53})$$

Output for "sat" $= 0.19 \cdot (2.0, 0.0) + 0.51 \cdot (0.0, 2.0) + 0.31 \cdot (1.5, 0.5)$

$$= (0.38, 0.0) + (0.0, 1.02) + (0.47, 0.16) = (\mathbf{0.84}, \mathbf{1.17})$$

Step 4: Compute Output

Output = weighted sum of value vectors.

Output for "cat" $= 0.51 \cdot \mathbf{v}_{\text{cat}} + 0.19 \cdot \mathbf{v}_{\text{sat}} + 0.31 \cdot \mathbf{v}_{\text{mat}}$
$= 0.51 \cdot (2.0, 0.0) + 0.19 \cdot (0.0, 2.0) + 0.31 \cdot (1.5, 0.5)$
$= (1.02, 0.0) + (0.0, 0.38) + (0.47, 0.16) = (\mathbf{1.48}, \mathbf{0.53})$

Output for "sat" $= 0.19 \cdot (2.0, 0.0) + 0.51 \cdot (0.0, 2.0) + 0.31 \cdot (1.5, 0.5)$
$= (0.38, 0.0) + (0.0, 1.02) + (0.47, 0.16) = (\mathbf{0.84}, \mathbf{1.17})$

"cat"'s output is close to its own value (2.0, 0.0) but pulled toward "mat".

Step 4: Compute Output

Output = weighted sum of value vectors.

Output for "cat" $= 0.51 \cdot \mathbf{v}_{\text{cat}} + 0.19 \cdot \mathbf{v}_{\text{sat}} + 0.31 \cdot \mathbf{v}_{\text{mat}}$
$= 0.51 \cdot (2.0, 0.0) + 0.19 \cdot (0.0, 2.0) + 0.31 \cdot (1.5, 0.5)$
$= (1.02, 0.0) + (0.0, 0.38) + (0.47, 0.16) = (\mathbf{1.48}, \mathbf{0.53})$

Output for "sat" $= 0.19 \cdot (2.0, 0.0) + 0.51 \cdot (0.0, 2.0) + 0.31 \cdot (1.5, 0.5)$
$= (0.38, 0.0) + (0.0, 1.02) + (0.47, 0.16) = (\mathbf{0.84}, \mathbf{1.17})$

"cat"'s output is close to its own value (2.0, 0.0) but pulled toward "mat".
Each token's output is enriched with info from tokens it attends to.

That's Attention.

What we just computed, as one formula:

That's Attention.

What we just computed, as one formula:

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

That's Attention.

What we just computed, as one formula:

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

- $\mathrm{QK}^\top$ = the score matrix (dot products of all query-key pairs)

That's Attention.

What we just computed, as one formula:

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

- $\mathrm{QK}^\top$ = the score matrix (dot products of all query-key pairs)
- Softmax (row-wise) = converts scores to weights that sum to 1

That's Attention.

What we just computed, as one formula:

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

- $\mathrm{QK}^\top$ = the score matrix (dot products of all query-key pairs)
- Softmax (row-wise) = converts scores to weights that sum to 1
- $\times\, \mathrm{V}$ = weighted average of value vectors

That's Attention.

What we just computed, as one formula:

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

- $QK^\top$ = the score matrix (dot products of all query-key pairs)
- Softmax (row-wise) = converts scores to weights that sum to 1
- $\times V$ = weighted average of value vectors

We divide by $\sqrt{d_k}$ for numerical stability (details in L25).

That's Attention.

What we just computed, as one formula:

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

- $QK^\top$ = the score matrix (dot products of all query-key pairs)
- Softmax (row-wise) = converts scores to weights that sum to 1
- $\times V$ = weighted average of value vectors

We divide by $\sqrt{d_k}$ for numerical stability (details in L25).

No magic. Matrix multiplication and a softmax is what gave us the AI boom :D.

Why This Is Powerful

1. Global: every token sees every other in one step. A CNN needs $O(T/k)$ layers.

Why This Is Powerful

1. Global: every token sees every other in one step. A CNN needs $O(T/k)$ layers.
2. Parallel: all scores and outputs computed simultaneously on GPU.

Why This Is Powerful

1. Global: every token sees every other in one step. A CNN needs O(T/k) layers.
2. Parallel: all scores and outputs computed simultaneously on GPU.
3. Dynamic: weights depend on the input, not fixed like CNN kernels.

Why This Is Powerful

1. Global: every token sees every other in one step. A CNN needs $O(T/k)$ layers.
2. Parallel: all scores and outputs computed simultaneously on GPU.
3. Dynamic: weights depend on the input, not fixed like CNN kernels.
4. Cost: $O(T^2 d)$ time, $O(T^2)$ memory. But depth is $O(1)$.

Why This Is Powerful

1. Global: every token sees every other in one step. A CNN needs $O(T/k)$ layers.
2. Parallel: all scores and outputs computed simultaneously on GPU.
3. Dynamic: weights depend on the input, not fixed like CNN kernels.
4. Cost: $O(T^2d)$ time, $O(T^2)$ memory. But depth is $O(1)$.

Comparing the approaches:

Why This Is Powerful

1. Global: every token sees every other in one step. A CNN needs $O(T/k)$ layers.
2. Parallel: all scores and outputs computed simultaneously on GPU.
3. Dynamic: weights depend on the input, not fixed like CNN kernels.
4. Cost: $O(T^2 d)$ time, $O(T^2)$ memory. But depth is $O(1)$.

Comparing the approaches:

	MLP	CNN	Attention
Weights	Fixed	Fixed	Input-dependent
Range	None	Local	Global
Steps to connect	N/A	$O(T/k)$	$O(1)$
Cost per layer	$O(Td^2)$	$O(Tkd)$	$O(T^2 d)$

Why Not Just Concatenate?

If we want every token to see every other, why not concatenate all embeddings and use an MLP?

Why Not Just Concatenate?

If we want every token to see every other, why not concatenate all embeddings and use an MLP?

$[\mathbf{x}_1; \mathbf{x}_2; \ldots; \mathbf{x}_\mathrm{T}] \in \mathbb{R}^{\mathrm{Td}}$, then multiply by $\mathrm{W} \in \mathbb{R}^{\mathrm{Td}\times\mathrm{Td}}$.

Why Not Just Concatenate?

If we want every token to see every other, why not concatenate all embeddings and use an MLP?

$[\mathbf{x}_1; \mathbf{x}_2; \ldots; \mathbf{x}_T] \in \mathbb{R}^{Td}$, then multiply by $W \in \mathbb{R}^{Td \times Td}$.

Problem 1: W has fixed size. If T changes, W does not fit.

Why Not Just Concatenate?

If we want every token to see every other, why not concatenate all embeddings and use an MLP?

$[\mathbf{x}_1; \mathbf{x}_2; \ldots; \mathbf{x}_T] \in \mathbb{R}^{Td}$, then multiply by $W \in \mathbb{R}^{Td \times Td}$.

Problem 1: W has fixed size. If T changes, W does not fit.

Problem 2: for $T = 1000$, $d = 768$: W has $(768{,}000)^2 \approx 5.9 \times 10^{11}$ entries.

Why Not Just Concatenate?

If we want every token to see every other, why not concatenate all embeddings and use an MLP?

$[\mathbf{x}_1; \mathbf{x}_2; \ldots; \mathbf{x}_T] \in \mathbb{R}^{Td}$, then multiply by $W \in \mathbb{R}^{Td \times Td}$.

Problem 1: W has fixed size. If T changes, W does not fit.
Problem 2: for $T = 1000$, $d = 768$: W has $(768{,}000)^2 \approx 5.9 \times 10^{11}$ entries.
Problem 3: no weight sharing across positions.

Why Not Just Concatenate?

If we want every token to see every other, why not concatenate all embeddings and use an MLP?

$[\mathbf{x}_1; \mathbf{x}_2; \ldots; \mathbf{x}_T] \in \mathbb{R}^{Td}$, then multiply by $W \in \mathbb{R}^{Td \times Td}$.

Problem 1: W has fixed size. If T changes, W does not fit.
Problem 2: for $T = 1000$, $d = 768$: W has $(768{,}000)^2 \approx 5.9 \times 10^{11}$ entries.
Problem 3: no weight sharing across positions.

Attention: same W_Q, W_K, W_V at every position, works for any T, parameters scale as $O(d^2)$ not $O(T^2d^2)$.

What Attention Learns

In a trained model, different layers learn different patterns (different W_Q, W_K, W_V):

What Attention Learns

In a trained model, different layers learn different patterns (different W_Q, W_K, W_V):
Early layers: attend to nearby tokens, punctuation (local patterns).

What Attention Learns

In a trained model, different layers learn different patterns (different W_Q, W_K, W_V):

Early layers: attend to nearby tokens, punctuation (local patterns).

Middle layers: syntactic relationships (subject-verb, adjective-noun).

What Attention Learns

In a trained model, different layers learn different patterns (different W_Q, W_K, W_V):

Early layers: attend to nearby tokens, punctuation (local patterns).

Middle layers: syntactic relationships (subject-verb, adjective-noun).

Later layers: semantic relationships and long-range dependencies.

What Attention Learns

In a trained model, different layers learn different patterns (different W_Q, W_K, W_V):

Early layers: attend to nearby tokens, punctuation (local patterns).

Middle layers: syntactic relationships (subject-verb, adjective-noun).

Later layers: semantic relationships and long-range dependencies.

Analogous to CNNs: early layers detect edges, later layers detect objects.

What Attention Learns

In a trained model, different layers learn different patterns (different W_Q, W_K, W_V):

Early layers: attend to nearby tokens, punctuation (local patterns).

Middle layers: syntactic relationships (subject-verb, adjective-noun).

Later layers: semantic relationships and long-range dependencies.

Analogous to CNNs: early layers detect edges, later layers detect objects. Attention builds a hierarchy of relational patterns rather than spatial ones.

Causal Masking

For generation: token t must not see tokens t+1, t+2, . . .

Causal Masking

For generation: token t must not see tokens t+1, t+2, . . .

(Otherwise the model would "cheat" by looking at the answer.)

Causal Masking

For generation: token t must not see tokens t+1, t+2, . . .

(Otherwise the model would "cheat" by looking at the answer.)

Solution: before softmax, set future scores to $-\infty$.

Causal Masking

For generation: token t must not see tokens t+1, t+2, . . .

(Otherwise the model would "cheat" by looking at the answer.)

Solution: before softmax, set future scores to $-\infty$.

$\exp(-\infty) = 0$, so future tokens get zero weight.

Causal Masking

For generation: token t must not see tokens t+1, t+2, . . .

(Otherwise the model would "cheat" by looking at the answer.)

Solution: before softmax, set future scores to $-\infty$.

$\exp(-\infty) = 0$, so future tokens get zero weight.

Example (3 tokens): the score matrix becomes

$$\begin{pmatrix} s_{11} & -\infty & -\infty \\ s_{21} & s_{22} & -\infty \\ s_{31} & s_{32} & s_{33} \end{pmatrix}$$

Causal Masking

For generation: token t must not see tokens t+1, t+2, . . .

(Otherwise the model would "cheat" by looking at the answer.)

Solution: before softmax, set future scores to $-\infty$.

$\exp(-\infty) = 0$, so future tokens get zero weight.

Example (3 tokens): the score matrix becomes

$$\begin{pmatrix} s_{11} & -\infty & -\infty \\ s_{21} & s_{22} & -\infty \\ s_{31} & s_{32} & s_{33} \end{pmatrix}$$

Token 1 only sees itself. Token 2 sees tokens 1–2. Token 3 sees all.

Key Takeaways

1. Sequences: order matters, length varies, long-range dependencies.

Key Takeaways

1. Sequences: order matters, length varies, long-range dependencies.
2. BPE: bytes $\rightarrow$ pre-tokenize $\rightarrow$ greedy merge $\rightarrow$ subword vocabulary.

Key Takeaways

1. Sequences: order matters, length varies, long-range dependencies.
2. BPE: bytes → pre-tokenize → greedy merge → subword vocabulary.
3. Attention: dynamic, content-based weighting via Q, K, V.

Key Takeaways

1. Sequences: order matters, length varies, long-range dependencies.
2. BPE: bytes $\rightarrow$ pre-tokenize $\rightarrow$ greedy merge $\rightarrow$ subword vocabulary.
3. Attention: dynamic, content-based weighting via Q, K, V.
4. Not just language: attention is a general mechanism across ML.

Key Takeaways

1. Sequences: order matters, length varies, long-range dependencies.
2. BPE: bytes $\rightarrow$ pre-tokenize $\rightarrow$ greedy merge $\rightarrow$ subword vocabulary.
3. Attention: dynamic, content-based weighting via Q, K, V.
4. Not just language: attention is a general mechanism across ML.

Next lecture: $\sqrt{\mathrm{d_k}}$ scaling, multi-head attention, positional encoding, full transformer.