CSE 592: Foundations of AI — University of Michigan

Lecture 12: Deep Learning II (text and Transformers)



Gabriel Poesia
Assistant Professor, CSE
University of Michigan, Ann Arbor
poesia@umich.edu
https://gpoesia.com

Plan

1. Text as vectors
2. Language modeling
3. Recurrent neural networks
4. Attention
5. Transformers

Administrivia

Grades released for HW1 - please review

Midterm exam in 2 weeks (Mon, Oct 26th). Same time as lecture: 9-10:20

Two rooms: DOW 1010 (this one) and DOW 1014 (same hallway)

We'll post a room assignment on Ed next week

Closed-book, one sheet of handwritten notes allowed (front and back).

Where we left off

> 🧠 **Recall**
> Deep learning: pick an architecture and a loss. Train with backprop + an optimizer.

Last lecture: images.

Pixels are already numbers.

We can resize every image to the input size of our neural model.

CNNs reuse the same small filters across the image.

> ? **Question**
> Today: how do we build a deep neural network that processes **text**?

Three questions

To train a network on text, we need to answer three questions.

1. **Representation:** text is made of discrete symbols. How do we turn it into vectors?
2. **Loss:** most available text has no labels in it (e.g., classes). What should the network predict?
3. **Architecture:** How will our neural entwork take inputs of arbitrary lengths?

Unlike images, we can't just resize a paragraph!

Let's answer each of these questions in the next sections.

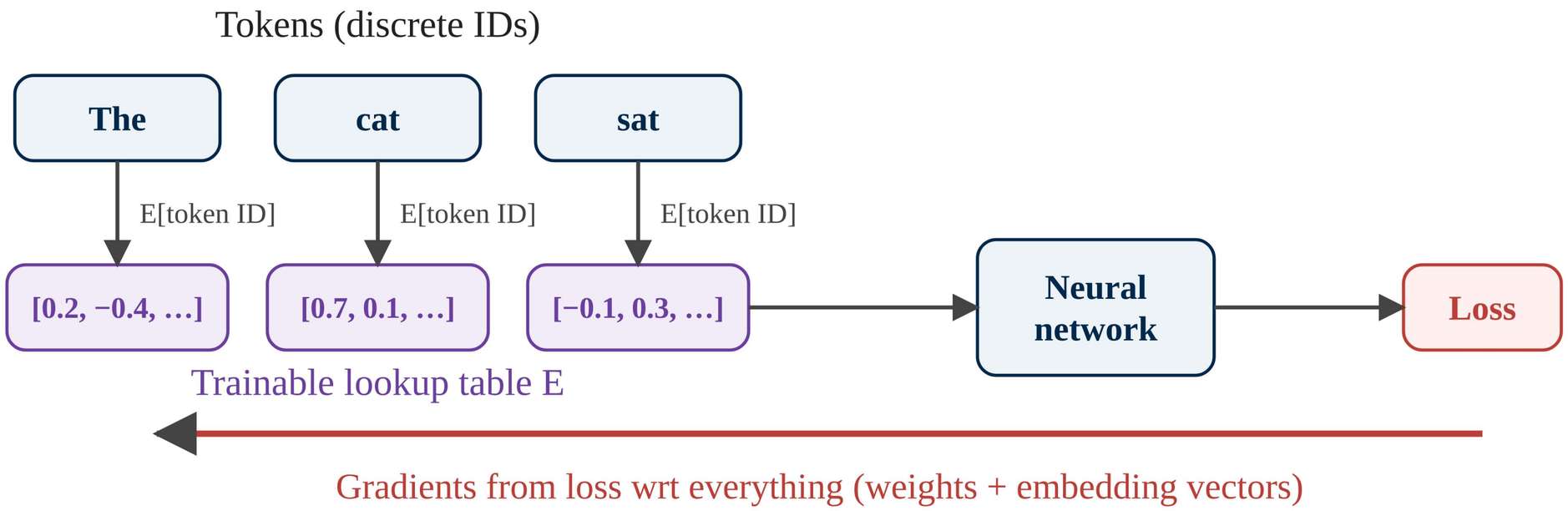
Text as vectors

From tokens to vectors

Question 1: how do we turn text into vectors?

A: Split the text into **tokens**, from a finite **vocabulary**. Each token has an ID.

Then, use an **embedding table** stores one vector per token ID. Look up each token's vector.



The table contains learnable parameters: we'll train these vectors with backprop too!

The embedding table

Setting vocabulary size V, vector width d: we get a table $E \in \mathbb{R}^{V \times d}$.

Token x_i is an ID. Its vector is row x_i of the table:

$$e_i = E[x_i, :]$$

Same as multiplying a one-hot vector by E: just a linear layer!

Q: Suppose “cat” appears twice in the input. What's the gradient for its row?

$$\frac{\partial \mathcal{L}}{\partial E[\text{cat}, :]} = \sum_{i:\, x_i = \text{cat}} \frac{\partial \mathcal{L}}{\partial e_i}$$

A: The sum of the gradients from both occurrences.

Tokens that don't appear: zero gradient for their rows.

What should the tokens be?

We still need to choose the vocabulary. How would we encode a sentence like `I love CSE592`?

Words I | love | UNK

Huge vocabulary, can't have all possible sequences as words... (common choice: have an UNK to denote 'everything else')

Bytes I | ␣ | l | o | v | e | ␣ | C | S | E | 5 | 9 | 2

256 tokens encode any text (UTF-8), in any language. But sequences get long.

Subwords I | ␣love | ␣CS | E | 59 | 2

Middle ground: common substrings get one token, rare/new words need a few pieces.

Most common choice now is the third (subword tokens).

How do we define the subwords?

Byte-pair encoding (BPE)

Idea (Byte-Pair Encoding): learn the token vocabulary from a corpus of text with a compression objective.

Sennrich, Haddow & Birch (ACL 2016): BPE for neural translation, originally over characters.

1. Start with all 256 bytes as tokens. Any text can be encoded.
2. Find the most frequent pair of adjacent tokens in the corpus.
3. Add that pair as a new token. Merge it everywhere.
4. Repeat until the vocabulary has the size we want.

To tokenize new text: start from bytes, apply the merges in the order we learned them.

We choose the vocabulary size. Bytes are always there as a fallback.

This “training” happens once, before training the neural network.

BPE: example

Corpus: `low` 3 times, `lower` 2 times. For simplicity, start from letters.

Merge	`low` (×3)	`lower` (×2)
Start	`l o w`	`l o w e r`
`l + o → lo`	`lo w`	`lo w e r`
`lo + w → low`	`low`	`low e r`
`e + r → er`	`low`	`low er`

Q: Which pair is most frequent?

A: `(l,o)` and `(o,w)`, 5 times each. Break the tie somehow: say we merge `(l,o)`.

Then `(lo,w)`: 5 times.

Then `(low,e)` and `(e,r)` tie at 2. Merge `(e,r)`.

Q: Suppose we get a new word `lowest` (after training). How will we encode it?

→ `low e s t`.

BPE on real text

GPT-2's tokenizer: byte-level BPE, 50,257 tokens.

`The␣Michigan␣Wolver` `ines␣beat␣Ohio␣State␣in␣Ann␣Arbor.`	53 bytes → 11 tokens
`def␣square(x):␣return␣x␣*␣x`	27 bytes → 9 tokens
`Bonj` `our,␣` `C3` `A7` `a␣va?`	16 bytes → 9 tokens
`ant` `idis` `establishment` `arian` `ism`	28 bytes → 5 tokens

Common words: one token each. The space before a word is part of its token.

Rarer words: a few pieces, like `␣Wolver` + `ines`.

Never unknown: “ç” falls back to its two UTF-8 bytes.

Question 1, answered: text → token IDs → vectors.

Colors alternate between tokens. Generated with OpenAI's `tiktoken` (GPT-2 encoding).

Language modeling

What should the network predict?

Question 2: what do we train the network to do?

We have lots of text: books, web pages, code. Mostly without labels.

Idea: train the network to **continue the text**.

Prefix	Plausible continuation
`2 + 3 =`	`5`
`French for cat:`	`chat`
`def square(x: int) -> int:`	`return x * x`

Continuing text well requires learning many different skills!

The text gives us its prediction problem: **self-supervised**, like diffusion last lecture.

This is the **pretraining** stage of LLMs.

We'll see post-training after the Reinforcement Learning module.

A probability distribution over text

Let's make “continue the text” precise.

> **Definition (Language model)**
> A model $p_\theta(x)$ that assigns a probability to every token sequence $x = (x_1, \ldots, x_T)$.

Training: give high probability to the text in our data.

Generation: sample new text from p_θ.

Two special tokens: BOS marks the beginning, EOS marks the end.



But how do we model a distribution over all possible texts?

Chain rule → next-token prediction

🧠 **Recall**

Chain rule: $p(a, b, c) = p(a)\, p(b \mid a)\, p(c \mid a, b)$.

Apply it to the tokens:

$$p_\theta(x_1, \ldots, x_T) = \prod_{t=1}^{T} p_\theta(x_t \mid x_{<t})$$

$x_{<t}$: all previous tokens, starting with BOS.

Each factor: a distribution over the **next token**, given everything before it.

This is exact (just chain rule). No assumptions so far!

? Question

What kind of prediction problem is each factor?

Classification, with V classes: one per token.

One classification problem per prefix

Suppose some network takes the prefix and outputs a vector $h \in \mathbb{R}^d$.

(We'll see architectures to do this later today)

A linear layer gives one **logit** per token in the vocabulary:

$$z = hW_{\text{out}} + b, \qquad W_{\text{out}} \in \mathbb{R}^{d \times V}$$

Softmax turns logits into probabilities:

$$p_\theta(x_t = v \mid x_{<t}) = \frac{\exp(z_v)}{\sum_{u=1}^{V} \exp(z_u)}$$

Positive, and sums to 1. Like logistic regression, now with V classes.

Maximum likelihood → cross-entropy

Training: choose θ that makes the training text likely.

$$\max_{\theta} \sum_{x \in D} \log p_\theta(x)$$

The log turns the chain-rule product into a sum. Negate it to get a loss:

$$\mathcal{L}(\theta) = -\sum_{x \in D} \sum_{t=1}^{T_x} \log p_\theta(x_t \mid x_{<t})$$

This is the **cross-entropy loss**: minus the log-probability of each correct next token.

Q: The correct next token gets probability 0.5. What's the loss at that position?

A: $-\log 0.5 \approx 0.69$. Probability 1 would give loss 0.

The targets are already in the text

Take one sequence and shift it by one position.

Network input	BOS	The	cat	sat
Target	The	cat	sat	EOS

Every position is a training example: the output at “cat” should predict “sat”.

> ⚠ **Careful**
> The output at “cat” must not see “sat”! Otherwise, the answer is in the input.

Our architecture will need to guarantee this.

Generating text

Once trained, we generate one token at a time:

1. Start from BOS, or from a prompt.
2. Compute the distribution over the next token.
3. Sample a token. Append it to the prefix.
4. Repeat until EOS (or a length limit).

Temperature $\tau > 0$: divide the logits by τ before the softmax.

$$p_\tau(v) = \frac{\exp(z_v/\tau)}{\sum_u \exp(z_u/\tau)}$$

Small τ: almost always the most likely token. Large τ: more random.

How good is a language model?

Average loss per token, on held-out text:

$$\overline{\mathcal{L}} = -\frac{1}{N}\sum_{t=1}^{N} \log p_\theta(x_t \mid x_{<t})$$

Often reported as **perplexity**:

$$\text{perplexity} = \exp(\overline{\mathcal{L}})$$

Q: Suppose a model just predicts uniform probability over all V tokens. What's its perplexity?
A: $\exp(-\log \frac{1}{V}) = V$. Perplexity k: on average, as unsure as picking among k tokens.

> ⚠ **Careful**
> Only compare perplexities between models with the same tokenizer.

OK, so now we have: text as vectors, and a loss function.

Last ingredient: what neural network computes h from a **sequence** of input vectors?

Recurrent neural networks

Variable-length inputs

We need a network that maps a prefix of *any length* to a vector h.

An MLP takes a fixed number of inputs.

Dependencies can be between arbitrary positions, and far away:

> The committee did **not** approve the budget.
> [... three paragraphs about the meeting ...]
> After the meeting, the proposers were completely ____

The network has to remember “not” across all those paragraphs.

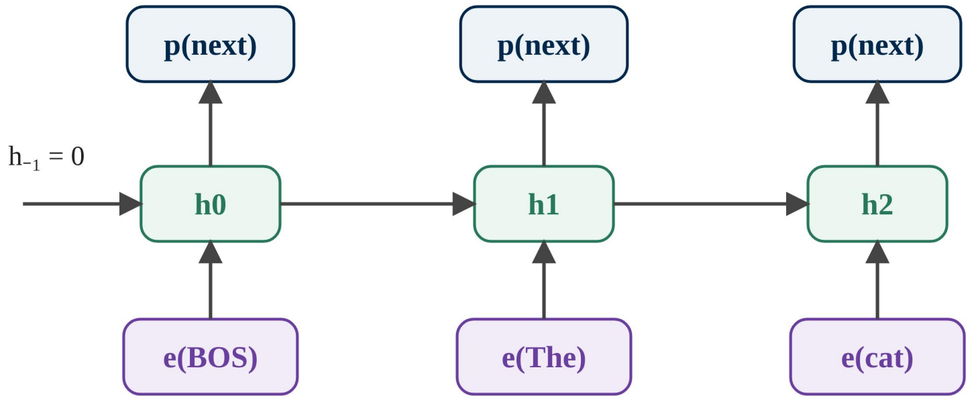
Recurrent neural networks

Idea (Recurrent Neural Networks, or RNNs):

Take one input token at a time. Keep a fixed-dimension **state** vector h_i that summarizes the prefix so far.

On each step: combine the next token's embedding with the previous state.

$$h_i = \tanh(e_i W_x + h_{i-1} W_h + b_h)$$



Predict the next token from h_i: logits + softmax, as before.

Same weights at every position, so any length works.

Unroll the network for one example

Character-level example: predict the next character in `hello`.

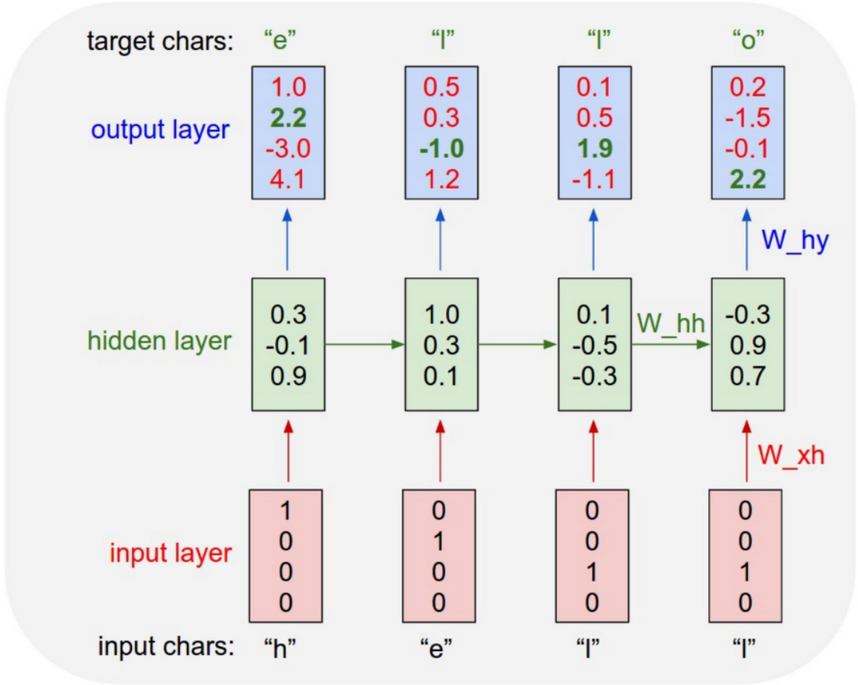
Inputs: `h e l l`.

Targets: `e l l o`.

Each column: logits, softmax, and a loss.

> **? Question**
> The input “l” appears twice. How can the two predictions differ?

Different states: each summarizes a different prefix.



Karpathy, The Unreasonable Effectiveness of RNNs (2015) — Character-level example. Numbers in the output layer are logits, not probabilities.

Backpropagation through time

Unrolled for one sequence, the RNN is an ordinary computation graph.

$$\mathcal{L} = \ell_1 + \ell_2 + \cdots + \ell_T$$

Backprop as usual. One difference: the same weights appear at every position.

Gradient will be the sum of the contributions from every position.

Just like “cat” appearing twice in the embedding table!

This is called **backpropagation through time (BPTT)**.

What does the state remember?

Ada was born in **London**.
[... many sentences about her work ...]
Ada's birthplace was ____

When the RNN reads “London”, it doesn't know what we'll ask later.

We might ask about her work, her family, or anything else in the text.

Challenge: all of it has to fit in one fixed-size state, overwritten at every token.

Long-range dependencies

Ada was born in **London**.
[... many sentences about her work ...]
Ada's birthplace was ____

“London” reaches the question only through every state in between:

$$h_i \to h_{i+1} \to \cdots \to h_T$$

Unrolled, that's a very deep network: one layer per token in between!

Gradients travel the same path backwards, multiplied once per step.

Recall lecture 11: multiply by 0.9 fifty times, and you get 0.005.

So gradients from distant tokens tend to vanish (or explode).

Computation: h_i needs h_{i-1} first.

We can't process positions in parallel, even during training.

Can we at least make remembering easier?

LSTM: learn what to keep

Long short-term memory (LSTM): an RNN with a memory vector c_i (separate from hidden state) and learned gates.

$$c_i = f_i \odot c_{i-1} + g_i \odot \tilde{c}_i$$

- Forget gate $f_i \in (0,1)^d$: how much old memory to keep.
- Input gate $g_i \in (0,1)^d$: how much new content $\tilde{c}_i$ to add.
- An output gate picks what to expose in the state h_i.

Gates are computed from the input and previous state, and trained with the same loss.

Q: Have we seen this before? Set $f_i = 1$:

$$c_i = c_{i-1} + g_i \odot \tilde{c}_i$$

If $f_i \approx 1$, this is like a residual connection, like in ResNet! Across time steps instead of layers.

Becomes easier to learn to keep specific things, and makes gradients flow better long-range

But it's still sequential on sequence length for training (need to generate each c_i, h_i to compute loss).

Hochreiter & Schmidhuber (1997): the original LSTM had no forget gate (always $f_i = 1$); Gers et al. (2000) added it. Explained in Olah, Understanding LSTM Networks (2015). Simplified notation.

What can a character-level LSTM learn?

LSTMs alone could do quite impressive things! Were a hugely useful architecture for over a decade

Example illustration: blog post from Andrej Karpathy in 2015

Trained LSTMs on different text corpora. Same task: predict the next **character**.

Shakespeare

```
PANDARUS:
Alas, I think he shall be come approached and the day
When little srain would be attain'd into being never fed,
```

Karpathy (2015) — Shakespeare section; short sample excerpt.

Speaker names, line breaks, old-fashioned phrasing.

Linux source code

```c
static void action_new_function(struct s_stat_info *wb)
{
  unsigned long flags;
  int lel_idx_bit = e->edd, *sys & ~((unsigned long) *FIRST_COMPAT);
  buf[0] = 0xFFFFFFFF & (bit << 4);
  min(inc, slist->bytes);
  printk(KERN_WARNING "Memory allocated %02x/%02x, "
    "original MLL instead\n"),
    min(min(multi_run - s->len, max) * num_data_in),
    frame_pos, sz + first_seg);
  div_u64_w(val, inb_p);
  spin_unlock(&disk->queue_lock);
  mutex_unlock(&s->sock->mutex);
```

Karpathy (2015) — Linux Source Code section; generated C-code excerpt.

Indentation, declarations, function calls, comments.

Even LaTeX

Trained on an algebraic geometry textbook: commands, environments, notation.

```
\begin{proof}
We may assume that $\mathcal{I}$ is an abelian sheaf on $\mathcal{C}$.
\item Given a morphism $\Delta : \mathcal{F} \to \mathcal{I}$
is an injective and let $\mathfrak q$ be an abelian sheaf on $X$.
Let $\mathcal{F}$ be a fibered complex. Let $\mathcal{F}$ be a category.
\begin{enumerate}
\item \hyperref[setain-construction-phantom]{Lemma}
\label{lemma-characterize-quasi-finite}
```

Karpathy, The Unreasonable Effectiveness of RNNs (2015) — Algebraic Geometry (LaTeX) section; generated source excerpt.

LaTeX almost compiles, and does look like a proof if you squint!

Trained on a single GPU (about 1 million characters in the corpora, 10M parameter 3-layer LSTM)

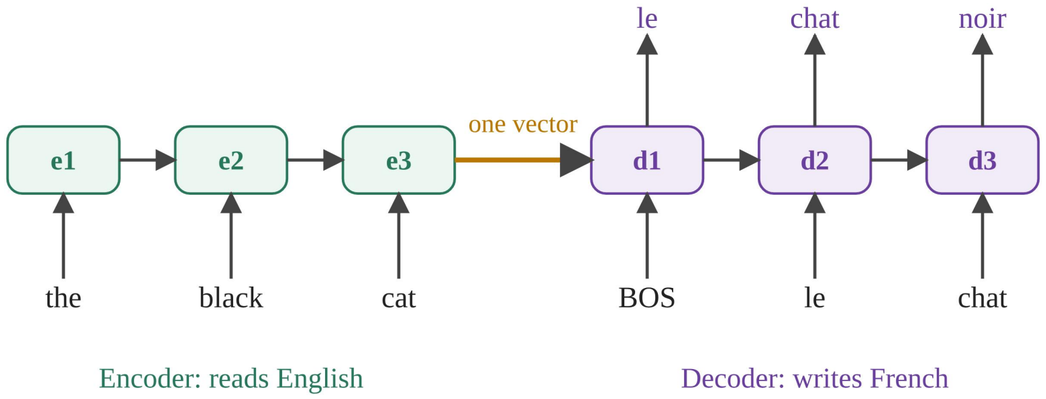
Attention

RNNs for translation

Up until relatively recently (5-10y ago), LSTMs were used for all sorts of language tasks.

Prominent example: **machine translation**.

Input: sentence in source language. Output: the same sentence in target language.



Example: English to French

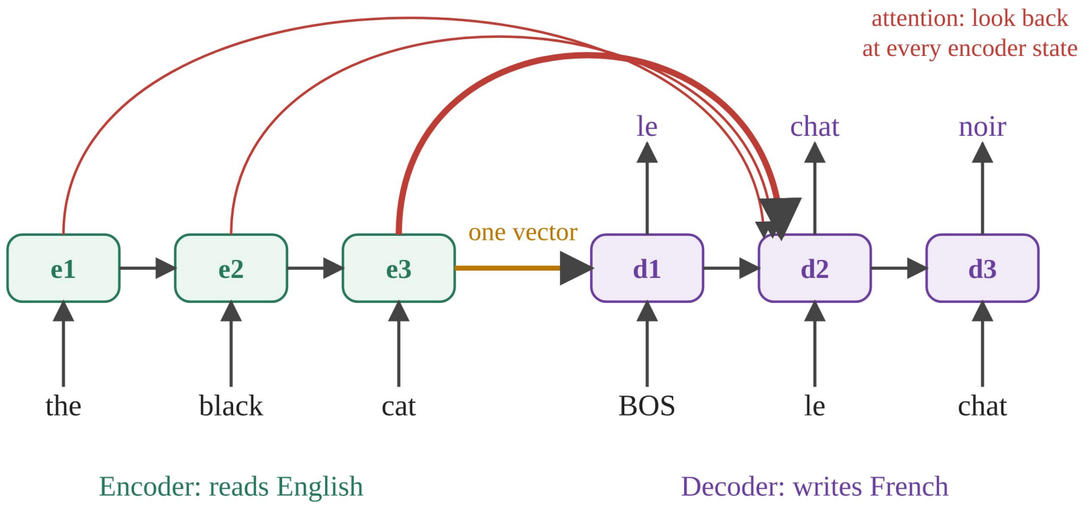
An **encoder** RNN reads the English sentence.

A **decoder** RNN writes the French: a language model that starts from the encoder's last state.

Problem: the whole sentence has to fit in that one vector!

Attention: look back instead of remembering

Bahdanau et al. (2015)'s idea: keep **all** the encoder states around.



At each output step, allow network to "query" (take a weighted average of) all the encoder states.

The network computes the weights. To write “chat”: mostly look at “cat”.

This is the high-level idea of **attention**. Any input word is now one hop away.

Where does the model look?

Each row: one French word being generated.

Each column: one English word. Brighter = more weight.

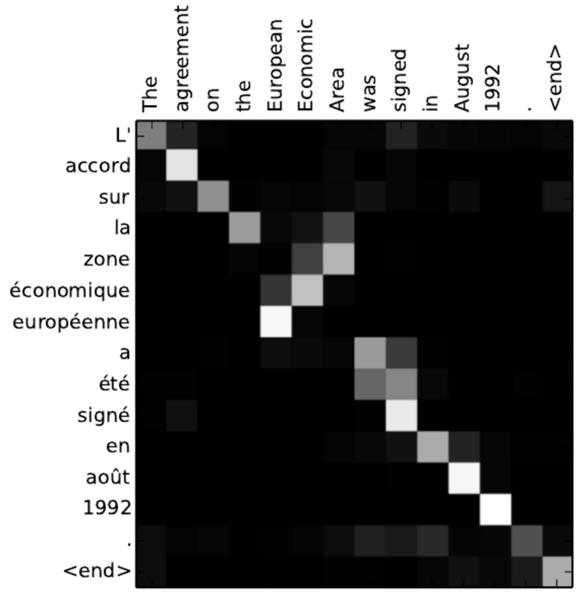
Mostly diagonal, but it handles word order: "european economic area" $\rightarrow$ "zone économique européenne".

No labels to say which words align. Learned from translation pairs alone.

But the RNNs still process one token at a time...

2017: *“Attention Is All You Need”* paper. This introduced the **Transformer**.

Idea: remove the RNN part, build the whole network out of the attention operation.



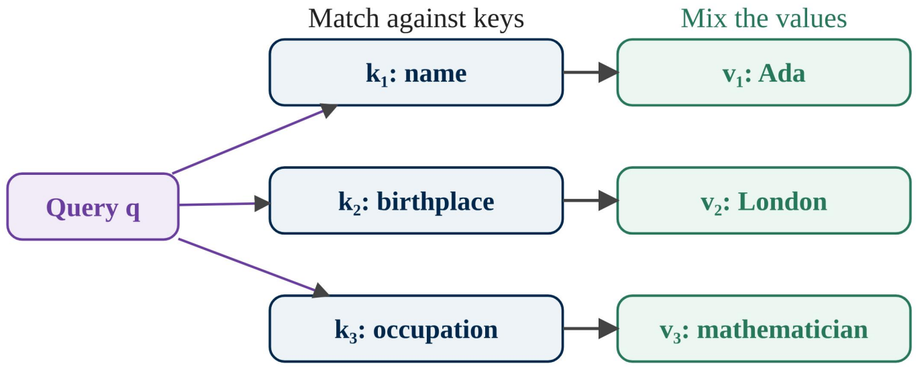
(a)

Bahdanau, Cho & Bengio (ICLR 2015) — Fig. 3(a). Attention weights for one sentence pair.

Attention as a lookup

Let's define attention on its own the way Transformers use it.

Think of dynamically building a lookup table from our prefix: **key-value pairs** (k_j, v_j).



Labels for illustration/intuition, k/v are actually vectors.

At a given step i, network will output a **Query** vector q_i: what it's looking for in the prefix

We'll give it an average of all values v_j weighted by similarity between q_i and k_j

Intuition: a sort of continuous, differentiable "dict"

Scores → attention weights

Score each key with a dot product: $s_j = q \cdot k_j$. A small example:

$$q = (1, 0), \quad k_1 = (\log 2, 0), \quad k_2 = (\log 3, 1), \quad k_3 = (0, -1)$$

Q: What are the three scores?

$$(s_1, s_2, s_3) = (\log 2, \log 3, 0)$$

Turn scores into weights with a softmax:

$$\alpha_j = \frac{e^{s_j}}{\sum_r e^{s_r}}, \qquad (\alpha_1, \alpha_2, \alpha_3) = \left(\frac{2}{6}, \frac{3}{6}, \frac{1}{6}\right)$$

Nonnegative, sum to 1. Larger score ⇒ more weight.

Weights → retrieved vector

Weights: $\alpha = (\frac{1}{3}, \frac{1}{2}, \frac{1}{6})$.

Values: $v_1 = (2, 0)$, $v_2 = (0, 2)$, $v_3 = (0, 0)$.

Q: What vector does this query retrieve?

$$o = \sum_j \alpha_j v_j = \tfrac{1}{3}(2,0) + \tfrac{1}{2}(0,2) + \tfrac{1}{6}(0,0) = (\tfrac{2}{3}, 1)$$

A mix of the values, instead of a hard lookup of one entry.

Every step is differentiable in q, the keys, and the values.

Idea: the network can jointly learn what to ask for, what to match, and what to return.

Scaled dot-product attention

One more detail: scale the scores.

$$s_j = \frac{q \cdot k_j}{\sqrt{d_k}}$$

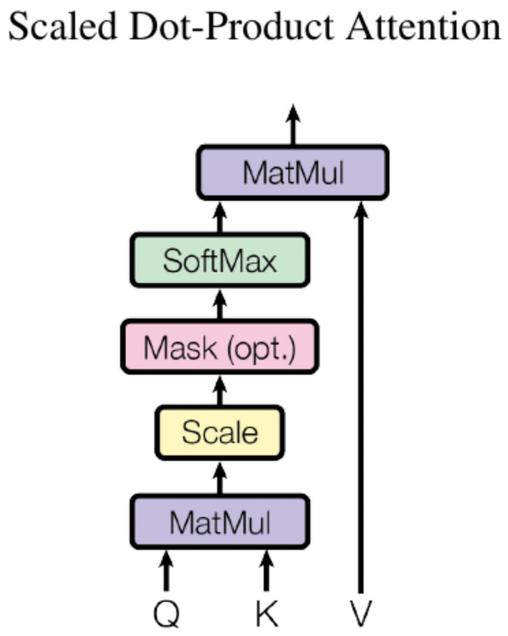
d_k: width of queries and keys.

Say the coordinates are independent, with mean 0 and variance 1. Then:

$$\mathrm{Var}(q \cdot k) = d_k$$

Wide vectors $\Rightarrow$ large scores $\Rightarrow$ softmax puts almost all weight on one key.

Dividing by $\sqrt{d_k}$ brings the variance back to 1.



Vaswani et al., Attention Is All You Need (2017) — Fig. 2, left. We'll get to the mask soon.

Self-attention: the sequence is the memory

Where do queries, keys, and values come from? Computed from the tokens themselves.

If sequence length is T, stack the T token vectors as rows of $X \in \mathbb{R}^{T \times d}$.

Three learned projections:

$$Q = XW_Q, \qquad K = XW_K, \qquad V = XW_V$$

$W_Q, W_K \in \mathbb{R}^{d \times d_k}$, $W_V \in \mathbb{R}^{d \times d_v}$. Shared across positions.

Every position asks a query. Every position offers a key and a value.

This is **self-attention**. In translation, decoder queries + encoder keys/values: **cross-attention**.

All queries at once

$$A = \text{softmax}_{\text{rows}}\left(\frac{QK^\top}{\sqrt{d_k}}\right), \qquad O = AV$$

Matrix	Shape	Meaning
Q, K	$T \times d_k$	One query / key per position
$QK^\top$	$T \times T$	Every query–key score
A	$T \times T$	Each row: weights over all positions
V	$T \times d_v$	One value per position
$O = AV$	$T \times d_v$	One retrieved vector per position

A_{ij}: how much position i uses position j's value.

But: for a sentence like "the cat sat on the mat", the output at “cat” should not see “sat”.

Here, full self-attention, every position can query every other!

If we can "see the next token", predicting it is trivial

Causal mask

Row i predicts the token **after** position i.

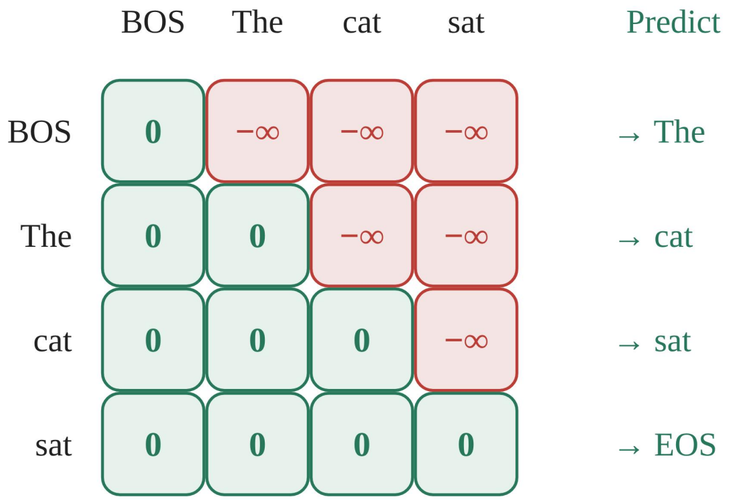
So it should look at positions $j \leq i$, but not $j > i$.

$$M_{ij} = \begin{cases} 0 & j \leq i \\ -\infty & j > i \end{cases}$$

$$A = \text{softmax}_{\text{rows}}\left(\frac{QK^\top}{\sqrt{d_k}} + M\right)$$

$e^{-\infty} = 0$: blocked positions get zero weight.

The diagonal is allowed: “cat” can see itself when predicting “sat”.



Columns: keys / values

Causal mask M. Green = allowed, red = blocked.

A causal attention pattern

Example weights after masking and softmax.

Q: Which rows can use information from “cat”?

A: “cat” and “sat”. Earlier rows can't.

The output at “sat” mixes four values.

The mask not learned, just computed.

The mask is for the decoder. In cases like translation, the encoder does not need a causal mask.



Columns: keys / values

Made-up weights, for illustration.

Multi-head attention

So far we have one look-up (weighted average) per position.

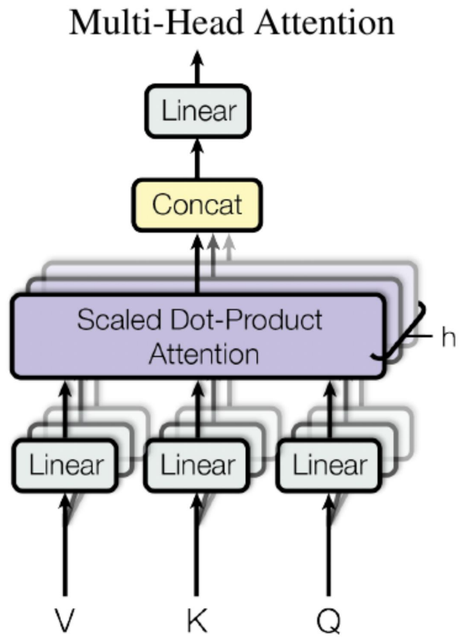
But a token may need several things at once (its subject, the previous word, ...).

Idea: have H **attention heads**, each with its own separate projections:

$$O^{(h)} = \text{Attn}(XW_Q^{(h)}, XW_K^{(h)}, XW_V^{(h)})$$

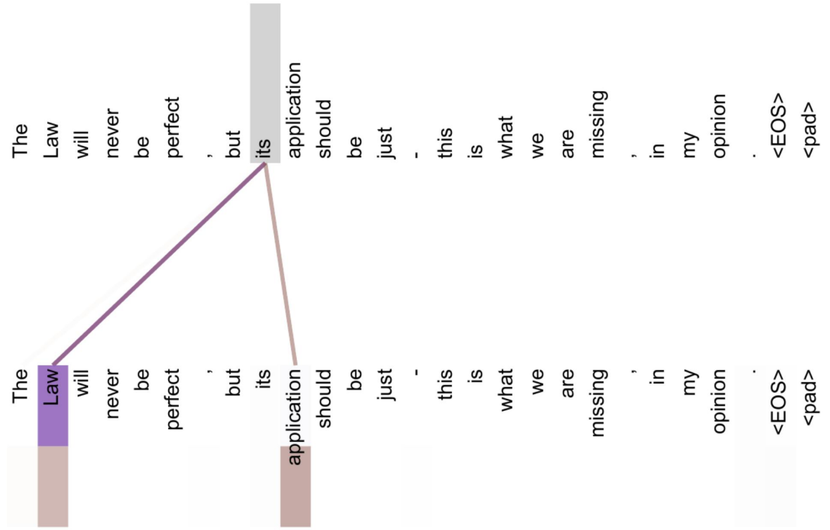
In multi-head attention, we concatenate the heads, then mix them with one more linear layer:

$$\text{MHA}(X) = \text{Concat}(O^{(1)}, \ldots, O^{(H)})W_O$$



Vaswani et al. (2017) — Fig. 2, right. Each head has its own learned linear projections.

What do different heads look at?



Vaswani et al., Attention Is All You Need (2017) — Fig. 4, bottom. Two heads from the original translation model's encoder, attending from ‘its’.

Example: “The Law will never be perfect, but **its** application should be just - this is what we are missing, in my opinion. <EOS>"

Two attention heads attending from “its” → law, application.

Both find what “its” refers to.

Different heads seem to specialize to different functions!

Note attention here is looking "forward" in the sentence (its → application): this is the encoder

Word order

🧠 **Recall**

One attention head, without the mask:

$$\text{Attn}(X) = \text{softmax}_{\text{rows}}\left(\frac{(XW_Q)(XW_K)^\top}{\sqrt{d_k}}\right)XW_V$$

Nothing in it depends on positions: only on the contents of the rows of X.

If we shuffle the rows of X, the outputs get shuffled the same way.

$$\text{Attn}(PX) = P\,\text{Attn}(X) \qquad (P\text{: a permutation matrix})$$

"Dog bites man" and "man bites dog" would give the same outputs, just reordered.

The causal mask adds some order, but we usually add positions explicitly:

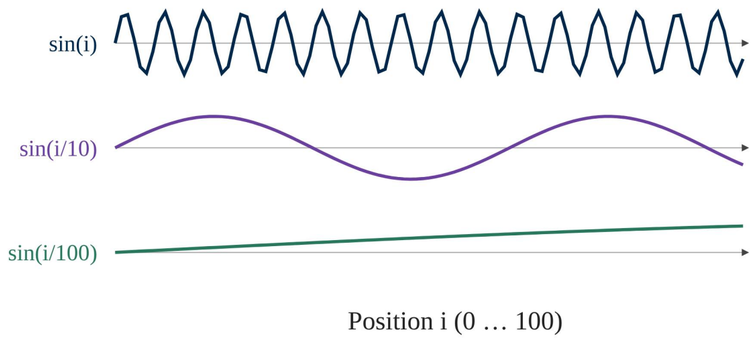
$$X_i = E[x_i, :] + P_i$$

P_i: a vector that encodes position i.

Positional encodings

Original Transformer: sine and cosine waves, at different frequencies.

$$P_{i,2r} = \sin\left(i/10000^{2r/d}\right), \quad P_{i,2r+1} = \cos\left(i/10000^{2r/d}\right)$$



Three of the sine channels.

Fast waves tell nearby positions apart. Slow waves: far-apart positions.

GPT-1/2/3 instead learn a table of position vectors, like token embeddings.

Vaswani et al. (2017), §3.5; Radford et al. (2019), §2.3.

Transformers

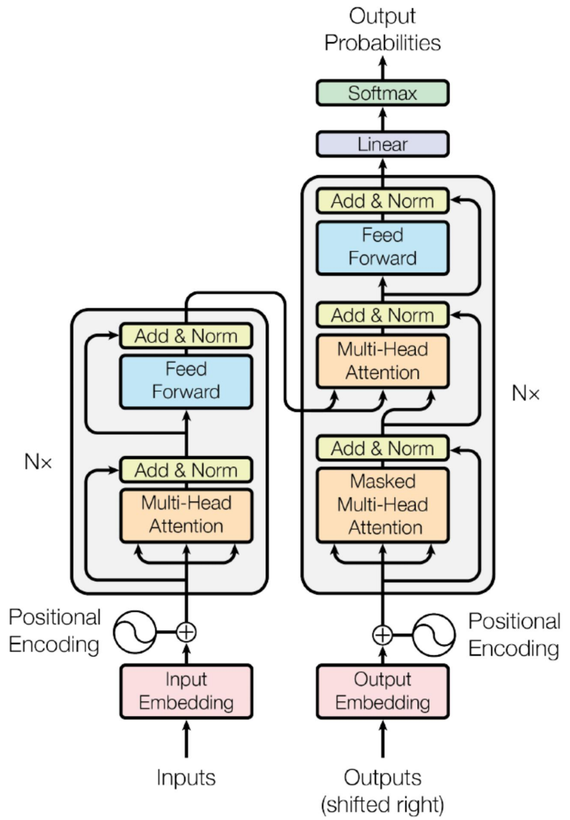
The original Transformer

Back to 2017. Attention Is All You Need: a translation model.

- **Encoder:** self-attention over the English sentence. No mask.
- **Decoder:** causal self-attention over the French written so far.
- **Cross-attention:** decoder queries, encoder keys and values.

Plus MLPs, residual connections, LayerNorm, and positions.

For a language model, we don't need an encoder: there's no separate input sentence.



Vaswani et al. (2017) — Fig. 1. Encoder at left, decoder at right.

A decoder-only language model

Keep only the decoder, without cross-attention.

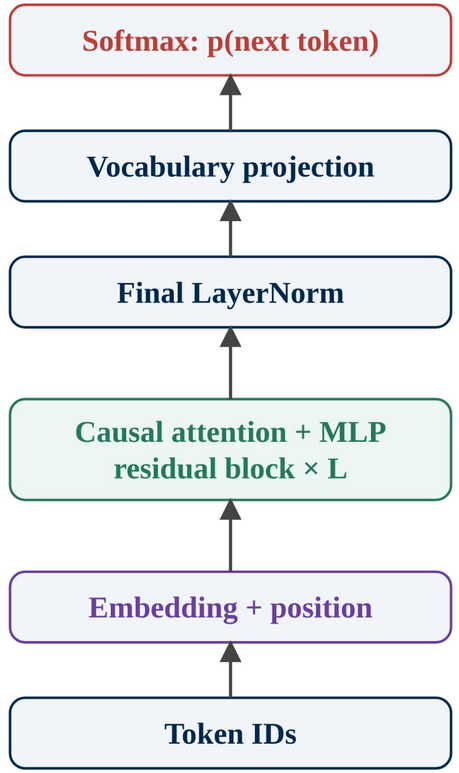
Embed the tokens and add positions.

Then a stack of L blocks. Each: causal self-attention + an MLP.

At the top: logits over the vocabulary, at every position.

Train with cross-entropy loss on the next token.

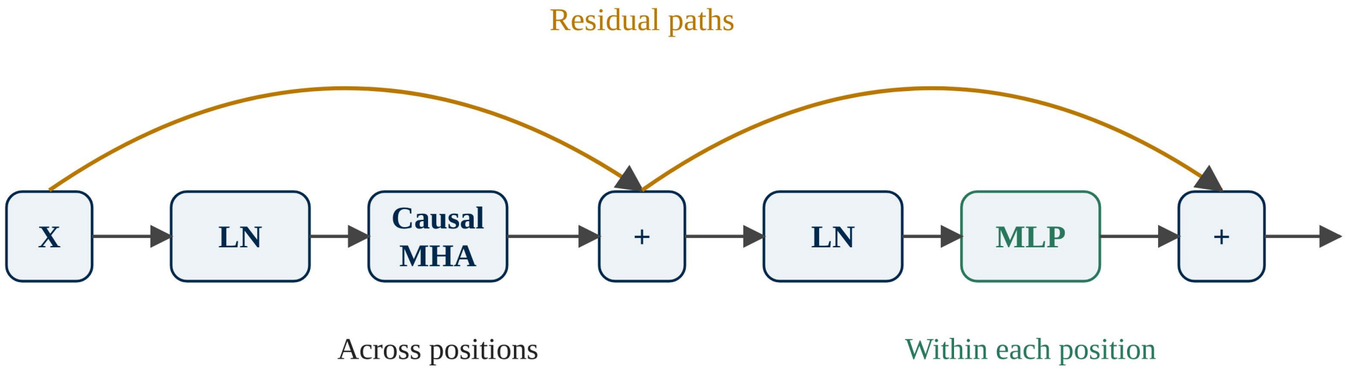
This is the architecture behind GPT.



One output distribution per input position

One Transformer block

Input $X \in \mathbb{R}^{T \times d}$. Output: same shape.



$$U = X + \mathrm{MHA}_{\mathrm{causal}}(\mathrm{LN}_1(X))$$

$$Y = U + \mathrm{MLP}(\mathrm{LN}_2(U))$$

Both pieces from last lecture: residual connections and LayerNorm.

LayerNorm here: each token's row on its own. It never mixes positions.

Stack L blocks, each with its own parameters.

"Pre-LN" form, used by GPT-2 and most later models (the 2017 paper normalizes after the addition). Xiong et al. (2020).

Two kinds of computation

Attention: moves information *between* positions.

MLP: transforms each position on its own.

$$\mathrm{MLP}(u) = \phi(uW_1 + b_1)W_2 + b_2$$

$W_1 \in \mathbb{R}^{d \times d_{\text{ff}}}$, $W_2 \in \mathbb{R}^{d_{\text{ff}} \times d}$. Usually $d_{\text{ff}} = 4d$.

Same MLP at every position. A different MLP in each block.

ϕ: ReLU in the original paper, GELU in GPT.

```python
class PositionwiseFeedForward(nn.Module):
    "Implements FFN equation."

    def __init__(self, d_model, d_ff, dropout=0.1):
        super(PositionwiseFeedForward, self).__init__()
        self.w_1 = nn.Linear(d_model, d_ff)
        self.w_2 = nn.Linear(d_ff, d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        return self.w_2(self.dropout(self.w_1(x).relu()))
```

Rush et al., The Annotated Transformer (2022 edition) — Position-wise feed-forward network: linear → ReLU → linear.

The whole forward pass

Token IDs in. Logits out. (Batch dimension omitted.)

```
X = token_embedding[ids] + position_embedding[:T]
for block in blocks:
    X = X + block.causal_attention(block.norm1(X))
    X = X + block.mlp(block.norm2(X))
X = final_norm(X)
logits = X @ W_out + b_out        # shape: T × V
```

Every line is differentiable: all of it is trained together, by backprop.

The training loop

A batch of token sequences. Shift them to get inputs and targets:

```
inputs  = tokens[:, :-1]
targets = tokens[:, 1:]
logits  = model(inputs)           # B × T × V
loss    = cross_entropy(logits, targets)
optimizer.zero_grad()
loss.backward()
optimizer.step()
```

Same loss as for the RNN. Only the architecture changed.

Optimizer: standard choice would be AdamW with warmup and cosine decay as we've seen

Parallel training, sequential generation

Training: we know all the input tokens.

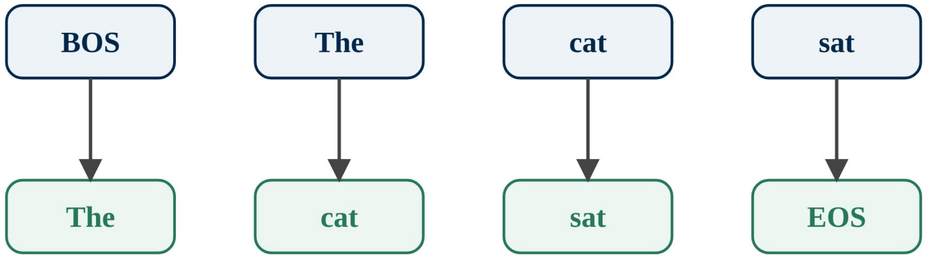
Compute all positions **in parallel (!)**: a few big matrix multiplications.

The mask keeps each prediction from seeing later tokens.

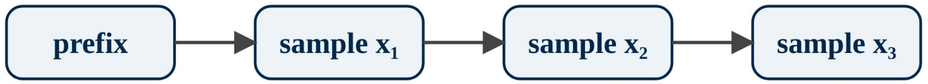
Recall: an RNN has to go one token at a time, even in training. So Transformers train much faster / use GPU parallelism better.

Generation: the next token isn't known until we sample it.

Still one token at a time: sample, append, repeat.



Training: all targets are already available



Generation: each sampled token extends the prefix

The cost of a long context

Every query is compared with every allowed key.

T tokens: about T^2 attention scores, for each head, for each block.

Double the context $\Rightarrow$ four times as many scores.

So models are trained with a maximum context length (GPT-2: 1,024 tokens).

Generation: each new token only needs a query for itself.

Keys and values of earlier tokens don't change: cache and reuse them (**KV cache**).

Vaswani et al. (2017), Table 1.

Summary

- **Representation:** BPE tokens + a learned embedding table.
- **Loss:** chain rule → next-token classification → cross-entropy.
- **RNNs:** any length, but everything goes through one state, one token at a time.
- **Attention:** queries match keys, return a weighted average of values.
- **Transformer:** causal attention + MLPs + residuals + LayerNorm + positions.

Next lecture: generalization in deep learning, and what happens when we scale Transformers up.