The Mathematics of Machine Learning 2026

Lecture 13

Tim van Erven

University of Amsterdam

Slides largely based on:

Dive into Deep Learning Book, Jay Alammar's blog

Outline

Softmax

Generative Pre-trained Transformer (GPT) Models

1. The Learning Task: Next-Word Prediction
2. GPT Architecture
3. From Output Vectors to Predicted Token Probabilities

GPT: Main Components

4. Simplified Decoder Block Structure
5. Attention
6. Positionwise Feed-forward Network

GPT: Details

7. Positional Encoding
8. Full Decoder Block Structure
9. Generating Text
10. Further Important Ingredients

Q&A

Softmax

The **softmax** function maps a vector of scores $z \in \mathbb{R}^K$ to a probability vector over K classes:

$$\text{softmax}(z)_k = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}} \qquad \text{for } k = 1, \ldots, K.$$

- You have already seen this in the context of multiclass logistic regression, where $z_k = X^\top \beta_k$.

Softmax

The **softmax** function maps a vector of scores $z \in \mathbb{R}^K$ to a probability vector over K classes:

$$\text{softmax}(z)_k = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}} \qquad \text{for } k = 1, \ldots, K.$$

- You have already seen this in the context of multiclass logistic regression, where $z_k = X^\top \beta_k$.
- By scaling the scores, softmax interpolates between the uniform distribution and the maximum (hence the name):

Lemma

Let $\mathcal{M} = \{k : z_k = \max_j z_j\}$ be the set of maximizers of z.

- *For $\lambda = 0$, $\text{softmax}(\lambda z)$ is the uniform distribution on $\{1, \ldots, K\}$.*
- *As $\lambda \to \infty$, $\text{softmax}(\lambda z) \to$ uniform distribution on $\mathcal{M}$.*

Proof. The case $\lambda = 0$ is trivial. For $\lambda \to \infty$: divide numerator and denominator by $e^{\lambda z^*}$, where $z^* = \max_k z_k$. Then $e^{\lambda(z_k - z^*)} = 1$ for $k \in \mathcal{M}$, and $e^{\lambda(z_k - z^*)} \to 0$ for $k \notin \mathcal{M}$. □

Outline

Softmax

Generative Pre-trained Transformer (GPT) Models

1. The Learning Task: Next-Word Prediction
2. GPT Architecture
3. From Output Vectors to Predicted Token Probabilities

GPT: Main Components

4. Simplified Decoder Block Structure
5. Attention
6. Positionwise Feed-forward Network

GPT: Details

7. Positional Encoding
8. Full Decoder Block Structure
9. Generating Text
10. Further Important Ingredients

Q&A

The Learning Task: Next-Word Prediction

Given a sequence of words $w_1, \ldots, w_t$, predict a **probability distribution** $\hat{P}(w_{t+1} \mid w_1, \ldots, w_t)$ over the next word w_{t+1}.

Examples:

- "Once upon a _____"
- "The capital of France is _____"
- "Time flies like an _____"

Log loss: $-\log \hat{P}(w_{t+1} \mid w_1, \ldots, w_t)$

The Learning Task: Next-Word Prediction

Given a sequence of words $w_1, \ldots, w_t$, predict a **probability distribution** $\hat{P}(w_{t+1} \mid w_1, \ldots, w_t)$ over the next word w_{t+1}.

Examples:

- "Once upon a _____"
- "The capital of France is _____"
- "Time flies like an _____"

Log loss: $-\log \hat{P}(w_{t+1} \mid w_1, \ldots, w_t)$

Remarks:

- w_{t+1} comes from a large set of V candidate words, the **vocabulary**. E.g. $V = 50\,257$ for GPT-2 [3].
- So this is a **multi-class classification** task with V classes and log loss, like multiclass logistic regression
- Challenge: the **number of words t in the input is not fixed!**

The Learning Task: Next-Word Prediction

Given a sequence of words $w_1, \ldots, w_t$, predict a **probability distribution** $\hat{P}(w_{t+1} \mid w_1, \ldots, w_t)$ over the next word w_{t+1}.

Examples:

- "Once upon a _____"
- "The capital of France is _____"
- "Time flies like an _____"

Log loss: $-\log \hat{P}(w_{t+1} \mid w_1, \ldots, w_t)$

Training data:

- A long sequence of text that combines multiple data sets: e.g. web crawl, books, code, news, dialogs, ...
- Model can predict w_{t+1} given $w_1, \ldots, w_t$ for $t = 0, \ldots, n-1$, where n is the **context window length**. E.g. $n = 1024$ for GPT-2 [3], $n = 2048$ for GPT-3 [1], $n > 10^5$ for recent models
- So train on chunks of size n from the training data.
- **Data very far from i.i.d.** within and between chunks!

Tokens Instead of Words

Actually, instead of *words* we use **tokens**.

Tokens:

- Typically **sub-word units**, not whole words
- Every token is represented by an **integer ID** in $\{1, \ldots, V\}$
- There exist **special tokens** in the vocabulary. For instance:
  - $\langle \text{BOS} \rangle$: beginning of sequence
  - $\langle \text{EOS} \rangle$: end of sequence

Examples:[1]

"Time flies like an arrow; fruit flies like a banana":

$\langle \text{BOS} \rangle$	time	flies	like	an	arrow	;	fruit	flies	like	a	banana

"Transformers tokenize subwords":

$\langle \text{BOS} \rangle$	transform	ers	tokenize	sub	words	$\langle \text{EOS} \rangle$

[1] Try out tokenizing on gpt-tokenizer.dev

Embeddings

Each token ID i is mapped to a **learned vector** $e_i \in \mathbb{R}^d$.

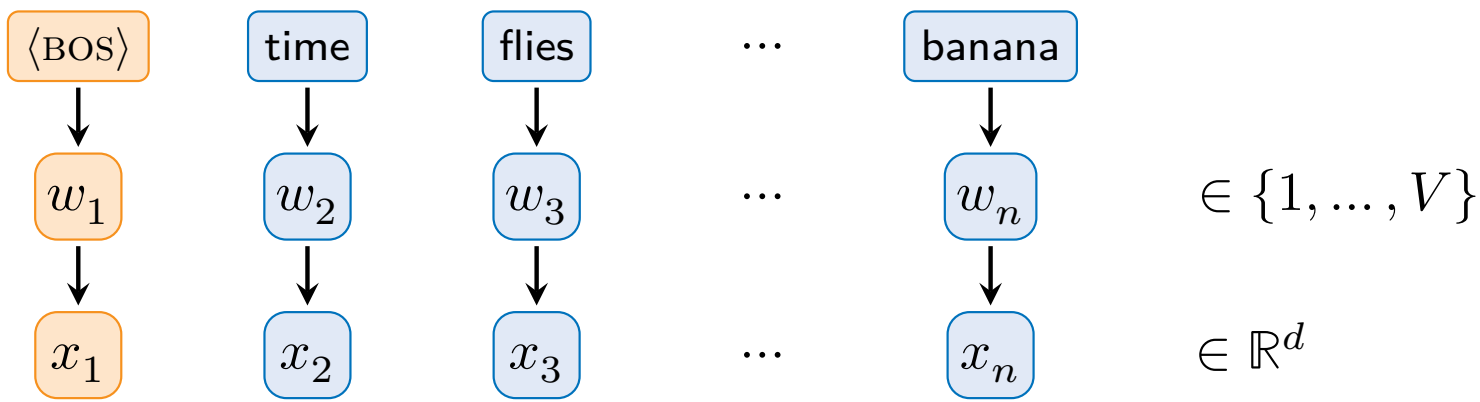
- $d =$ **embedding dimension**
  (e.g. $d = 12\,288$ for the largest version of GPT-3 [1])
- Collected in the **embedding matrix** $E \in \mathbb{R}^{V \times d}$: row i is the embedding $e_i^\top$ of token i
- The entries of E are **learned parameters** of the model, trained together with the rest of the network

Embeddings

Each token ID i is mapped to a **learned vector** $e_i \in \mathbb{R}^d$.

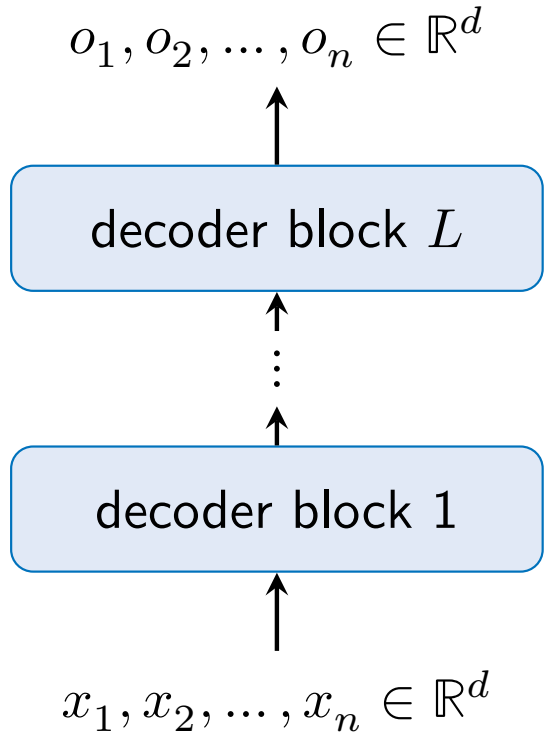
- $d =$ **embedding dimension**
  (e.g. $d = 12\,288$ for the largest version of GPT-3 [1])
- Collected in the **embedding matrix** $E \in \mathbb{R}^{V \times d}$: row i is the embedding $e_i^\top$ of token i
- The entries of E are **learned parameters** of the model, trained together with the rest of the network

A sequence of n tokens with IDs $w_1, \ldots, w_n$ is then represented by the corresponding embedding vectors $x_1, \ldots, x_n \in \mathbb{R}^d$ with $x_t = e_{w_t}$:



GPT Architecture

- Stack of L **decoder blocks**, one on top of the other
- Each block takes n vectors in $\mathbb{R}^d$ and outputs n vectors in $\mathbb{R}^d$: **same shape in, same shape out**
- Each block has its own learned parameters
- o_t only depends on $x_1, \dots, x_t$ and is used to predict token w_{t+1} (which is embedded as x_{t+1})



Scale for various versions of GPT-3 [1]:

- Number of decoder blocks L: $12 - 96$
- Embedding dimension d: $768 - 12\,288$

GPT only uses a 'decoder' part from transformers. General transformers may also have an 'encoder' part, which is useful e.g. for translation between languages.

From Output Vectors to Predicted Token Probabilities

At each position t, the network produces an output vector $o_t \in \mathbb{R}^d$. The probability of the next token being $i \in \{1, \ldots, V\}$ is

$$\hat{P}(w_{t+1} = i \mid w_1, \ldots, w_t) = \mathrm{softmax}(z_t)_i,$$

where

$$z_t = Eo_t = \left(\langle o_t, e_1 \rangle, \ldots, \langle o_t, e_V \rangle\right)^\top \in \mathbb{R}^V.$$

Interpretation:

- The inner product $\langle o_t, e_i \rangle$ is a **measure of similarity** between o_t and the embedding e_i of token i. So the embeddings are reused for **unembedding**!
- This is like the predictions for multiclass logistic regression with transformed features o_t and parameter vector e_i for class i

Outline

Softmax

Generative Pre-trained Transformer (GPT) Models

1. The Learning Task: Next-Word Prediction
2. GPT Architecture
3. From Output Vectors to Predicted Token Probabilities

GPT: Main Components

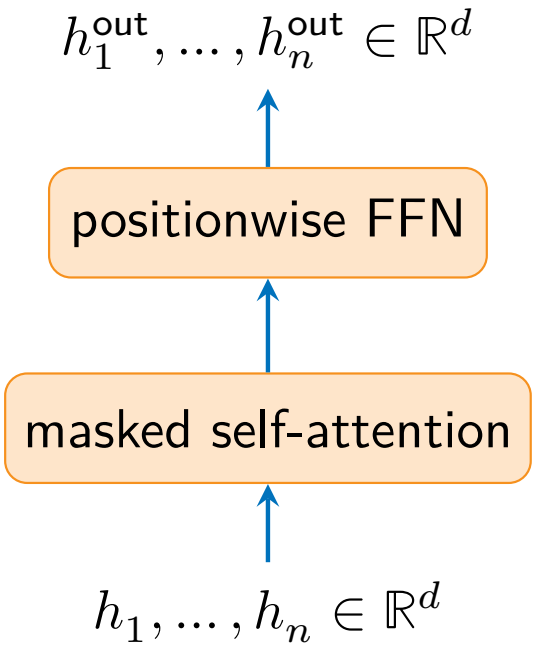
4. Simplified Decoder Block Structure
5. Attention
6. Positionwise Feed-forward Network

GPT: Details

7. Positional Encoding
8. Full Decoder Block Structure
9. Generating Text
10. Further Important Ingredients

Q&A

Simplified Decoder Block Structure



The main parts of each decoder block are:

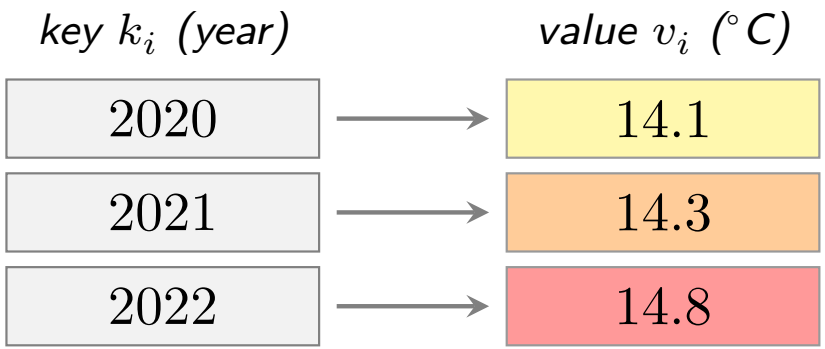
- **Self-attention**: mixes information across positions $t = 1, \ldots, n$
- **Positionwise feed-forward network**: processes each position t independently

The input and output of all parts have the same shape: n vectors in $\mathbb{R}^d$

Attention I: Queries, Keys, and Values

Traditional database: a collection of **key-value pairs**

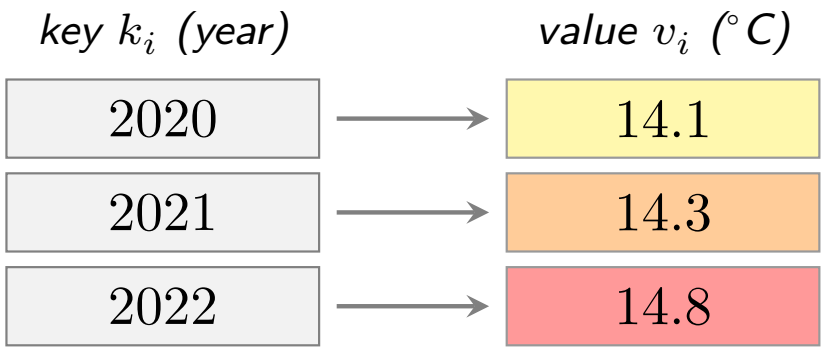
$D = \{(k_1, v_1), \ldots, (k_m, v_m)\}$. A lookup with query q retrieves the value v_i whose key k_i matches q exactly.



Lookup with $q = 2021$ returns 14.3

Attention I: Queries, Keys, and Values

Traditional database: a collection of **key-value pairs** $D = \{(k_1, v_1), \ldots, (k_m, v_m)\}$. A lookup with query q retrieves the value v_i whose key k_i matches q exactly.



Lookup with $q = 2021$ returns 14.3

Attention: weighted average of *all* values, with weights given by similarity of the query to each key:

$$\text{Attention}(q, D) = \sum_{i=1}^{m} \alpha(q, k_i)\, v_i,$$

where $$\alpha(q, k_i) = \text{softmax}\big(\text{score}(q, k_1), \ldots, \text{score}(q, k_m)\big)_i$$

for a scoring function $\text{score}(q, k)$ that measures similarity between q and k.

Attention II: Self-attention

Associate a **query, key and value** with each input vector h_t:

$$q_t = W_q h_t, \qquad k_t = W_k h_t, \qquad v_t = W_v\, h_t \qquad \in \mathbb{R}^d,$$

with **learned parameters** $W_q, W_k, W_v \in \mathbb{R}^{d \times d}$ shared across positions t

Attention II: Self-attention

Associate a **query, key and value** with each input vector h_t:

$$q_t = W_q h_t, \qquad k_t = W_k h_t, \qquad v_t = W_v\, h_t \qquad \in \mathbb{R}^d,$$

with **learned parameters** $W_q, W_k, W_v \in \mathbb{R}^{d\times d}$ shared across positions t

Output:
All positions share the same database $D = \{(k_1, v_1), \ldots, (k_n, v_n)\}$. The output at position t uses query q_t:

$$a_t = \text{Attention}(q_t, D) = \sum_{s=1}^{n} \alpha(q_t, k_s)\, v_s \qquad \in \mathbb{R}^d.$$

Attention II: Self-attention

Associate a **query, key and value** with each input vector h_t:

$$q_t = W_q h_t, \qquad k_t = W_k h_t, \qquad v_t = W_v\, h_t \qquad \in \mathbb{R}^d,$$

with **learned parameters** $W_q, W_k, W_v \in \mathbb{R}^{d\times d}$ shared across positions t

Output:
All positions share the same database $D = \{(k_1, v_1), \dots, (k_n, v_n)\}$. The output at position t uses query q_t:

$$a_t = \text{Attention}(q_t, D) = \sum_{s=1}^{n} \alpha(q_t, k_s)\, v_s \qquad \in \mathbb{R}^d.$$

Scoring function:

$$\text{score}(q, k) = \frac{\langle q, k\rangle}{\sqrt{d_k}}$$

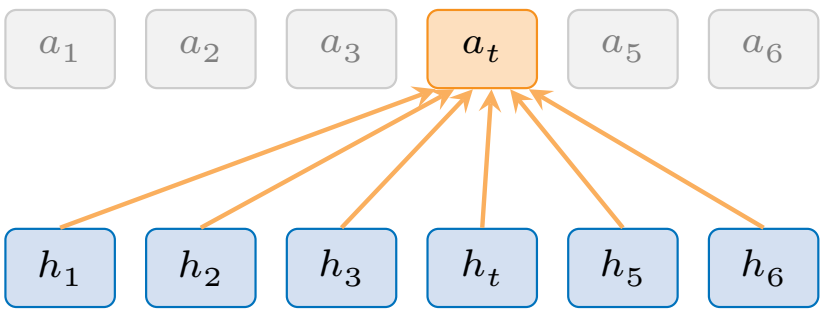
where d_k is the dimension of q_t and k_t (here $d_k = d$). The $\sqrt{d_k}$ keeps the variance at an approximately constant scale.

Attention III: Masked Self-attention

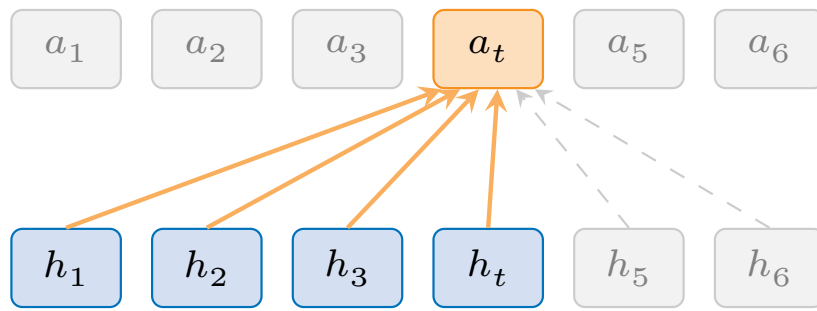
Problem: The prediction for w_{t+1} is based on a_t, which may only depend on past tokens $w_1, \ldots, w_t$. But $a_t = \sum_{s=1}^{n} \alpha(q_t, k_s)\, v_s$ also sums over **future positions** $s > t$.

Fix by masking the future: Force $a_t = \sum_{s=1}^{t} \alpha(q_t, k_s)\, v_s$ by setting scores for future positions to $-\infty$ before the softmax:

$$\widetilde{\text{score}}_t(q_t, k_s) = \begin{cases} \text{score}(q_t, k_s) & \text{if } s \leq t, \\ -\infty & \text{if } s > t. \end{cases}$$



Self-attention



Masked self-attention

NB The future is masked the same way in all decoder blocks.

Attention IV: Multi-head Attention

Multiple Heads:

- Run M **(masked) attention heads in parallel** on the same inputs $h_1, \ldots, h_n$
- Each head has its own learned parameters $W_q^{(m)}, W_k^{(m)}, W_v^{(m)} \in \mathbb{R}^{(d/M)\times d}$ of **reduced height** d/M
- Each head produces outputs $a_1^{(m)}, \ldots, a_n^{(m)} \in \mathbb{R}^{d/M}$ (so the scaling in score now uses $d_k = d/M$)

Attention IV: Multi-head Attention

Multiple Heads:

- Run M **(masked) attention heads in parallel** on the same inputs $h_1, \ldots, h_n$
- Each head has its own learned parameters $W_q^{(m)}, W_k^{(m)}, W_v^{(m)} \in \mathbb{R}^{(d/M)\times d}$ of **reduced height** d/M
- Each head produces outputs $a_1^{(m)}, \ldots, a_n^{(m)} \in \mathbb{R}^{d/M}$ (so the scaling in score now uses $d_k = d/M$)

Combine the Outputs: Concatenate the outputs and project

$$a_t = W_M \begin{pmatrix} a_t^{(1)} \\ \vdots \\ a_t^{(M)} \end{pmatrix} \in \mathbb{R}^d, \qquad \text{with learned parameters } W_M \in \mathbb{R}^{d\times d}.$$

Attention IV: Multi-head Attention

Multiple Heads:

- Run M **(masked) attention heads in parallel** on the same inputs $h_1, \dots, h_n$
- Each head has its own learned parameters $W_q^{(m)}, W_k^{(m)}, W_v^{(m)} \in \mathbb{R}^{(d/M)\times d}$ of **reduced height** d/M
- Each head produces outputs $a_1^{(m)}, \dots, a_n^{(m)} \in \mathbb{R}^{d/M}$ (so the scaling in score now uses $d_k = d/M$)

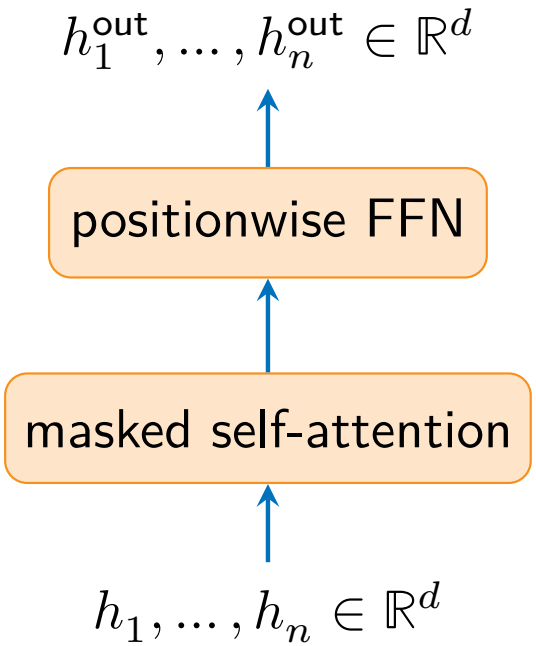
Combine the Outputs: Concatenate the outputs and project

$$a_t = W_M \begin{pmatrix} a_t^{(1)} \\ \vdots \\ a_t^{(M)} \end{pmatrix} \in \mathbb{R}^d, \qquad \text{with learned parameters } W_M \in \mathbb{R}^{d\times d}.$$

Remarks:

- **Interpretation:** different heads can specialize, e.g. in syntactic vs. semantic patterns
- **Computation cost** is similar to single-head attention: $3d^2$ parameters for $W_q^{(m)}, W_k^{(m)}, W_v^{(m)}$ combined (same as single-head), plus d^2 for W_M makes $4d^2$ parameters in total.

Simplified Decoder Block Structure



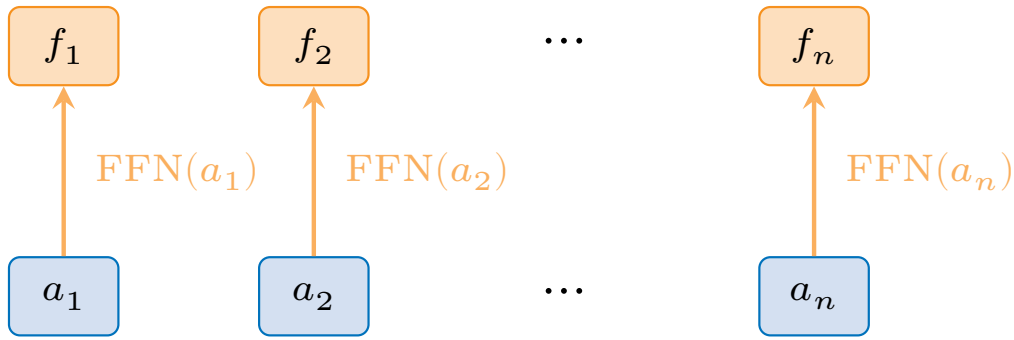
The main parts of each decoder block are:

- **Self-attention**: mixes information across positions $t = 1, \ldots, n$
- **Positionwise feed-forward network**: processes each position t independently

The input and output of all parts have the same shape: n vectors in $\mathbb{R}^d$

Positionwise Feed-forward Network

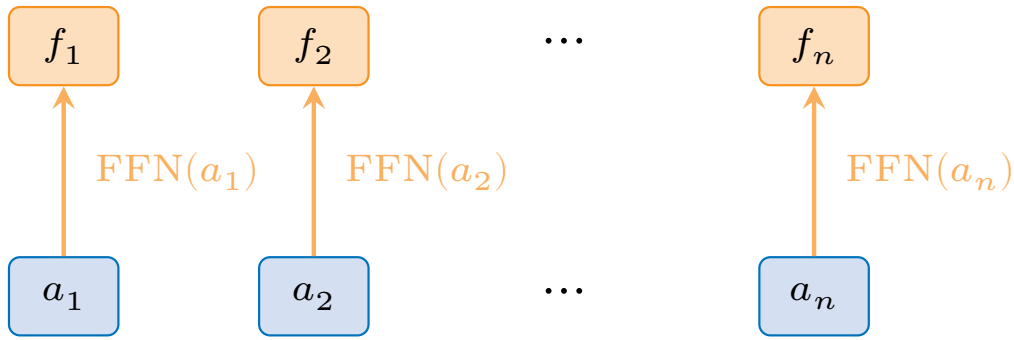
Apply the same two-layer network FFN at every position t:



$$\mathrm{FFN}(a) = W_2\,\sigma(W_1 a + b_1) + b_2 \in \mathbb{R}^d$$

Positionwise Feed-forward Network

Apply the same two-layer network FFN at every position t:



$$\mathrm{FFN}(a) = W_2\,\sigma(W_1 a + b_1) + b_2 \in \mathbb{R}^d$$

- Learned parameters $W_1 \in \mathbb{R}^{d_{\text{ffn}} \times d}$, $W_2 \in \mathbb{R}^{d \times d_{\text{ffn}}}$, $b_1 \in \mathbb{R}^{d_{\text{ffn}}}$, $b_2 \in \mathbb{R}^d$ shared across positions.
- Width of hidden layer: d_{ffn} **wider than input/output dimension** d. E.g. $d_{\text{ffn}} = 4d$.
- Activation σ: ReLU classically; modern models often use a variant called **SwiGLU** (empirically better)

Outline

Softmax

Generative Pre-trained Transformer (GPT) Models

1. The Learning Task: Next-Word Prediction
2. GPT Architecture
3. From Output Vectors to Predicted Token Probabilities

GPT: Main Components

4. Simplified Decoder Block Structure
5. Attention
6. Positionwise Feed-forward Network

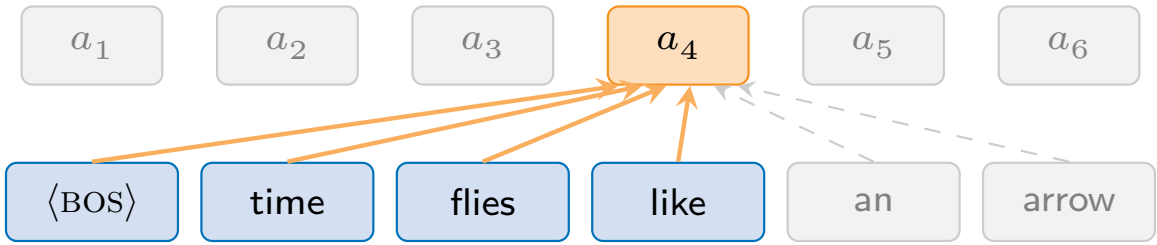
GPT: Details

7. Positional Encoding
8. Full Decoder Block Structure
9. Generating Text
10. Further Important Ingredients

Q&A

Positional Encoding

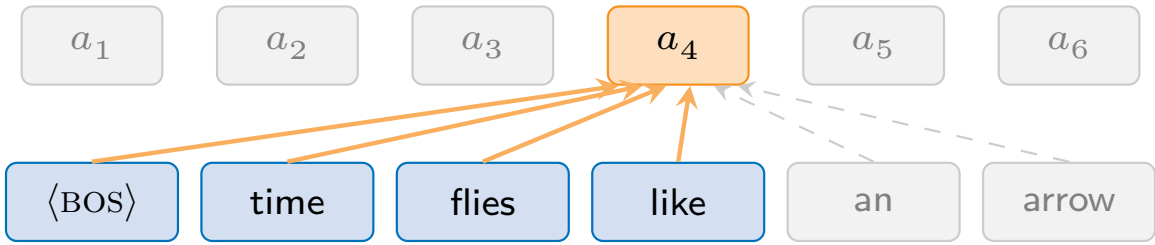
Problem: Self-attention $a_t = \sum_{s=1}^{t} \alpha(q_t, k_s)\, v_s$ has **no notion of word order**.



a_4 would be the same for "flies like time an arrow"!

Positional Encoding

Problem: Self-attention $a_t = \sum_{s=1}^{t} \alpha(q_t, k_s)\, v_s$ has **no notion of word order**.



a_4 would be the same for "flies like time an arrow"!

Fix by encoding position in inputs: Replace inputs by $x_t = e_{w_t} + p_t$, where $p_t \in \mathbb{R}^d$ encodes the position t. Let $\omega_i = 10\,000^{-2i/d}$.

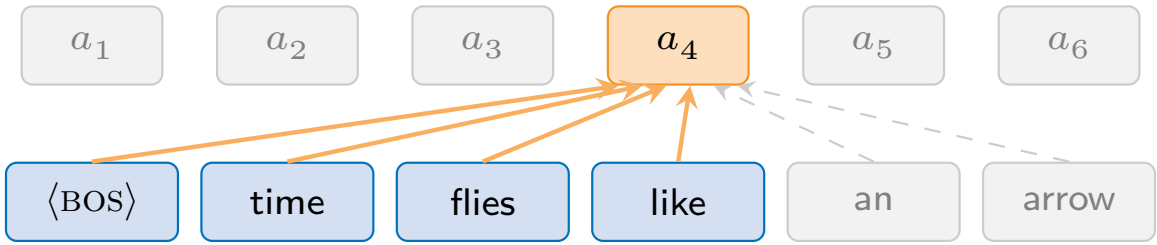
$$p_{t,2i} = \sin(\omega_i\, t), \qquad p_{t,2i+1} = \cos(\omega_i\, t) \qquad \text{for } i = 0, \ldots, d/2 - 1$$

Remarks:

- Similar to the discrete Fourier transform of positions $t = 1, \ldots, n$
- It is not fully clear why it makes sense to add p_t to e_{w_t} instead of e.g. concatenating the two.

Positional Encoding

Problem: Self-attention $a_t = \sum_{s=1}^{t} \alpha(q_t, k_s)\, v_s$ has **no notion of word order**.



a_4 would be the same for "flies like time an arrow"!

Fix by encoding position in inputs: Replace inputs by $x_t = e_{w_t} + p_t$, where $p_t \in \mathbb{R}^d$ encodes the position t. Let $\omega_i = 10\,000^{-2i/d}$.

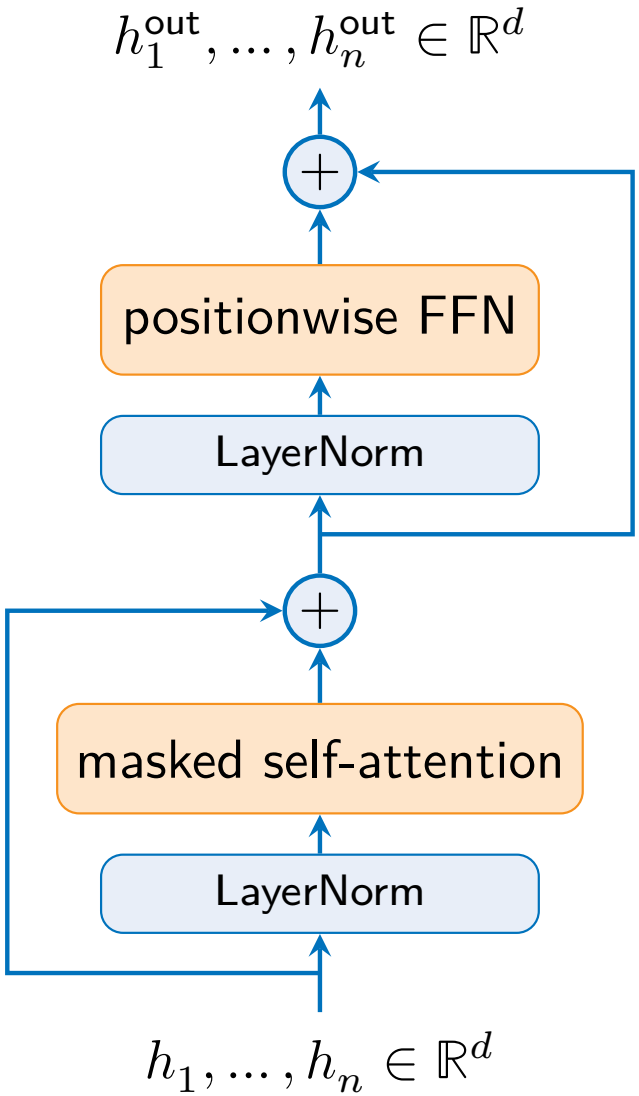
$$p_{t,2i} = \sin(\omega_i\, t), \qquad p_{t,2i+1} = \cos(\omega_i\, t) \qquad \text{for } i = 0, \ldots, d/2 - 1$$

Lemma

There exists a linear operation independent of t that maps $(p_{t,2i}, p_{t,2i+1})$ to $(p_{t+k,2i}, p_{t+k,2i+1})$.

Proof: By the angle-sum identities, $A = \begin{pmatrix} \cos(\omega_i k) & \sin(\omega_i k) \\ -\sin(\omega_i k) & \cos(\omega_i k) \end{pmatrix}$ sends $(\sin(\omega_i\, t), \cos(\omega_i\, t))^\top$ to $(\sin(\omega_i(t+k)), \cos(\omega_i(t+k)))^\top$. □

Full Decoder Block Structure



Two extra components wrapped around each sublayer:

- **Skip connections ($\oplus$)**: add the input of each sublayer back to its output, $h \mapsto h + \text{Sublayer}(h)$. Helps because Sublayer only needs to learn deviations from identity function.
- **LayerNorm**: normalize vectors $h_1, \ldots, h_n$ to zero mean and unit variance; then apply a learned affine transformation. Stabilizes training.

Applying LayerNorm *before* each sublayer (pre-LN, as shown) is standard from GPT-2 onward; the original transformer applied it *after* (post-LN).

Generating Text

Autoregressive generation: start from $\langle \text{BOS} \rangle$ (optionally followed by a user prompt), then repeatedly

1. Run the network on the current sequence $w_1, \ldots, w_t$
2. Generate a token w_{t+1} from $\hat{P}(w_{t+1} \mid w_1, \ldots, w_t)$ (see below)
3. Stop when the model emits the end-of-sequence token: $w_{t+1} = \langle \text{EOS} \rangle$ Otherwise append w_{t+1} to the sequence and repeat

Generating Text

Autoregressive generation: start from $\langle \text{BOS} \rangle$ (optionally followed by a user prompt), then repeatedly

1. Run the network on the current sequence $w_1, \ldots, w_t$
2. Generate a token w_{t+1} from $\hat{P}(w_{t+1} \mid w_1, \ldots, w_t)$ (see below)
3. Stop when the model emits the end-of-sequence token: $w_{t+1} = \langle \text{EOS} \rangle$ Otherwise append w_{t+1} to the sequence and repeat

Generating w_{t+1} from $\hat{P}(\cdot \mid w_1, \ldots, w_t)$ [2]:

- **Regular sampling:** draw w_{t+1} from the full distribution. Diverse, but occasionally picks low-probability junk and becomes incoherent.
- **Greedy:** Deterministically choose the most likely token. Lack of variation; can get stuck repeating the same phrase again and again.
- **Top-k sampling:** keep only the k most probable tokens, renormalize, and sample. Common middle ground.
- **Top-p / nucleus sampling:** Top-k for smallest k such that total probability mass of all top-k items is at least p.

Further Important Ingredients

- **Reinforcement learning from human feedback (RLHF):**
  - Humans rank model outputs, and a separate reward model is trained to predict which responses humans prefer
  - The GPT model parameters are then adjusted by reinforcement learning to maximize human satisfaction (as predicted by the reward model)
  - RLHF aligns model behavior with human goals: it improves helpfulness, safety, instruction-following, and conversational quality beyond what next-token prediction alone provides.

Further Important Ingredients

- **Reinforcement learning from human feedback (RLHF):**
  - Humans rank model outputs, and a separate reward model is trained to predict which responses humans prefer
  - The GPT model parameters are then adjusted by reinforcement learning to maximize human satisfaction (as predicted by the reward model)
  - RLHF aligns model behavior with human goals: it improves helpfulness, safety, instruction-following, and conversational quality beyond what next-token prediction alone provides.
- **Run time:** $O(n^2 d + nd^2)$ operations per transformer block
  - $n^2 d$ from self-attention $\rightarrow$ limits practical size of context window n
  - nd^2 from the linear projections and feed-forward layer

Further Important Ingredients

- **Reinforcement learning from human feedback (RLHF):**
  - Humans rank model outputs, and a separate reward model is trained to predict which responses humans prefer
  - The GPT model parameters are then adjusted by reinforcement learning to maximize human satisfaction (as predicted by the reward model)
  - RLHF aligns model behavior with human goals: it improves helpfulness, safety, instruction-following, and conversational quality beyond what next-token prediction alone provides.
- **Run time:** $O(n^2 d + nd^2)$ operations per transformer block
  - $n^2 d$ from self-attention $\rightarrow$ limits practical size of context window n
  - nd^2 from the linear projections and feed-forward layer
- **Parameter count:** Extremely large. E.g. GPT-3 has 175 billion parameters [1].

Outline

Softmax

Generative Pre-trained Transformer (GPT) Models

1. The Learning Task: Next-Word Prediction
2. GPT Architecture
3. From Output Vectors to Predicted Token Probabilities

GPT: Main Components

4. Simplified Decoder Block Structure
5. Attention
6. Positionwise Feed-forward Network

GPT: Details

7. Positional Encoding
8. Full Decoder Block Structure
9. Generating Text
10. Further Important Ingredients

Q&A

Questions

Questions about any part of the course?

References

[1] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901, 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

[2] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi. The curious case of neural text degeneration. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=rygGQyrFvH.

[3] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language models are unsupervised multitask learners. Unpublished, 2019. URL https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.