



Transformers and Natural Language Processing

Lecture *Machine Learning* on March 11-13, 2024

Transformers

Attention

Encoder

Applications

Decoder

Mario Stanke
Institut für Mathematik und Informatik
Universität Greifswald

Generative Pre-Trained Transformer

GPT-4 GPT-4 Technical Report, [March 2023](#), [arXiv](#)

"GPT-4 is a Transformer-style model [...] Given both the competitive landscape and the safety implications of large-scale models like GPT-4, this report contains no further details about the architecture", rumored to be a mixture of models, presumably like GPT-3

GPT-3 Language Models are Few-Shot Learners, Brown et al., [2020](#), [arXiv](#)

"We use the same model and architecture as GPT-2 [...] with the exception that we use alternating dense and locally banded sparse attention patterns [...]"

GPT-2 Language Models are Unsupervised Multitask Learners, Radford et al., [2019](#)

"We use a Transformer [...] based architecture for our LMs. The model largely follows the details of the OpenAI GPT model (Radford et al., 2018) with a few modifications [...]" (no principle changes)

"GPT-1" Improving language understanding by generative pre-training, Radford, Narasimhan, Salimans and Sutskever, [2018](#)

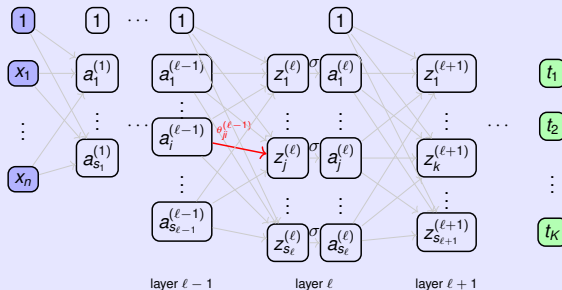
"we use a [multi-layer Transformer decoder](#) [...] for the language model, which is a variant of the transformer" [Vaswani et al., 2017].

"Original" Attention Is All You Need, Vaswani et al., [arXiv](#), [2017](#)



Weights

In a vanilla neural network (multilayer perceptron)



a net activation is a weighted sum of activations of the previous layer:

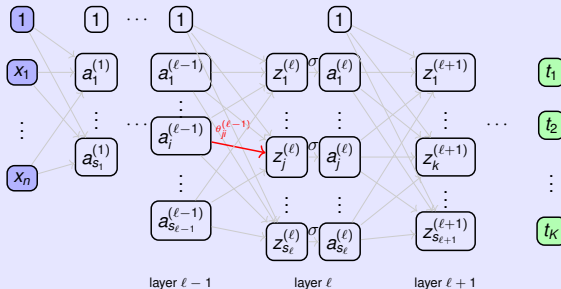
$$z_j^{(\ell)} = \sum_i \theta_{ji}^{(\ell-1)} a_i^{(\ell-1)}$$

The **weights** θ_{ji} are determined by training and are then **fixed**.



Dynamic Weights

In a vanilla neural network (multilayer perceptron)



a net activation is a weighted sum of activations of the previous layer:

$$z_j^{(\ell)} = \sum_i \theta_{ji}^{(\ell-1)} a_i^{(\ell-1)}$$

The **weights** θ_{ji} are determined by training and are then **fixed**.

Idea: Compute the weights themselves with a neural network



Data-dependent weights

The idea to let the weights depend on the input has at least existed since the early 1990s:

Excerpts of Habilitation of Jürgen Schmidhuber (1993)

Kapitel 3

DYNAMISCHE VERBINDUNGEN

1981

Ein Beispiel dafür liefern die wohl erstmals von ~~von~~ der Malsburg vorgeschlagenen 'dynamischen Verbindungen' (e.g. [138]). Bei diesen handelt es sich um **Kanten, deren Gewichte (= Synapsenstärken) sich innerhalb kürzester Zeit von Grund auf ändern können**. Solche '**schnellen Gewichte**' stehen im Kontrast zu den in praktisch allen Netzwerkmodellen verwendeten 'langsamen Gewichten', die sich nur durch wiederholte Trainingseinflüsse signifikant ändern und keine Kurzzeitspeicherfunktion übernehmen können.

Nun wird jede Gewichtsvariable $w_{ba} \in W_F$ gemäß

$$w_{ba}(t) = \sigma(w_{ba}(t-1), \square w_{ba}(t)) \quad (3.2)$$

geändert, wobei σ bezüglich aller Parameter differenzierbar sein soll. $\square w_{ba}(t)$ ist eine Funktion von S' Ausgabe und wird durch einen der beiden in den nächsten beiden Unterabschnitten beschriebenen Mechanismen berechnet.

3.1.2 ARCHITEKTUR 2

Stelle für jeden Knoten in F , dem wenigstens eine Verbindung mit einem schnellen Gewicht entspringt, einen Ausgabeknoten in S bereit. Nenne die Menge derartiger Ausgabeknoten VON . Spendiere einen weiteren Ausgabeknoten in S für jeden Knoten in F , zu dem wenigstens eine 'schnellgewichtige' Verbindung führt. Nenne die Menge dieser Ausgabeknoten ZU . Für jede Gewichtsvariable $w_{ba} \in W_F$ haben wir jetzt einen Knoten $s_a \in VON$ und einen Knoten $s_b \in ZU$. Definiere für den Zeitpunkt t

$$\square w_{ba}(t) := g(s_a(t), s_b(t)),$$

wobei g bezüglich all seiner Parameter differenzierbar sein muß. Wir konzentrieren uns auf ein repräsentatives Beispiel, nämlich den Fall, daß g den Multiplikationsoperator darstellt:

$$\square w_{ba}(t) := s_a(t)s_b(t). \quad (3.5)$$

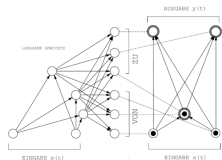


Abbildung 3.2: Grundstruktur eines der 'schnellen' Netzwerke gegeben durch die von langsame, Netz g einen lokalen Ausgangsknoten VON (für den 'Schlüssel') und ZU (für den lokal) vorgegebene Auswertung.

Additional idea (presumably of Schmidhuber): The use of a binary function like **multiplication** to obtain weights for **all pairs** using a linear number of activations.



The Attention Mechanism

Input

| | matrix | shape |
|---------|--------|-----------------|
| queries | Q | (ℓ_q, d_k) |
| keys | K | (ℓ_v, d_k) |
| values | V | (ℓ_v, d_v) |

- The **first dimension** of each matrix is a **sequence position**/index into a vector of so-called *tokens*.
- The sizes of the **second dimensions** are hyperparameters.

The Scaled Dot-Product Attention ...

... is

$$A := \text{Attention}(Q, K, V) := \underbrace{\text{softmax}\left(Q \cdot K^T \frac{1}{\sqrt{d_k}}\right)}_{=: \alpha} \cdot V \quad (1)$$

Each row of A is a convex combination of the rows of V .
The vectors $\alpha[i, :]$ are called **attention scores**.



Self-Attention

queries Q , keys K and values V are computed from a single input $X \in \mathbb{R}^{\ell \times d}$ as:

$$Q = X \cdot W_Q \quad \text{with } W_Q \in \mathbb{R}^{d \times d_k}$$

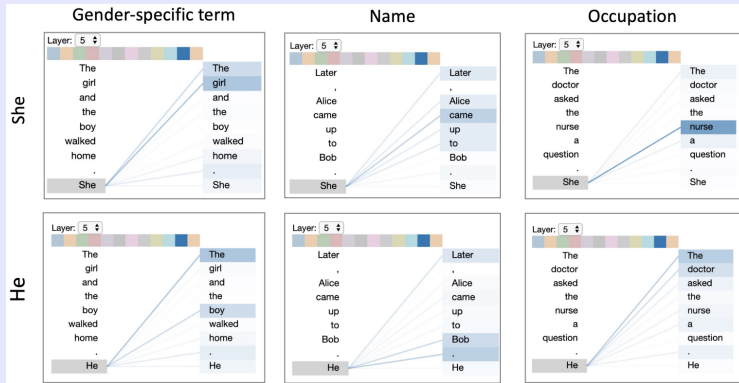
$$K = X \cdot W_K \quad \text{with } W_K \in \mathbb{R}^{d \times d_k}$$

$$V = X \cdot W_V \quad \text{with } W_V \in \mathbb{R}^{d \times d_v},$$

Unless otherwise stated: $d_k = d_v = d$



Attention Weight Intuition



Jesse Vig: "A Multiscale Visualization of Attention in the Transformer Model", arXiv, 2019

Some attention weights $\alpha[i, :]$ in GPT-2.

- A large attention weight $\alpha[i, j]$ may mean high relative relevance of the vector representation of token j that is **attended to** for the representation of token i that **is attending**.
- Here the parameters were trained to predict the respective next token in 40 GB of filtered texts scraped from the web.

Multi-Head Self-Attention (tf.keras.layers.MultiHeadAttention)

Idea: Apply the attention mechanism multiple times in parallel, concatenate results.

Multi-Head Self-Attention (tf.keras.layers.MultiHeadAttention)

Idea: Apply the attention mechanism multiple times in parallel, concatenate results.

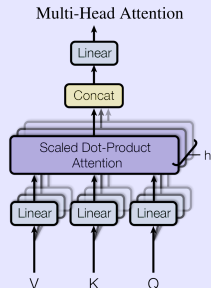
Let $d = h \cdot d_h$. h is called the **number of attention heads**.

$$\text{head}^i := \text{Attention}(\underbrace{X \cdot W_Q^i}_{=:Q^i}, \underbrace{X \cdot W_K^i}_{=:K^i}, \underbrace{X \cdot W_V^i}_{=:V^i}) \quad (i = 1..h), \quad (2)$$

where $W_Q^i, W_K^i, W_V^i \in \mathbb{R}^{d \times d_h}$.

In multi-head self-attention, the heads are concatenated along the last axis:

$$\text{MHA} := \text{Concat}(\text{head}^1, \dots, \text{head}^h) \in \mathbb{R}^{\ell \times d} \quad (3)$$



Vaswani et al., "Attention is all you need", arXiv, 2017



Natural Language Processing

Tokens

```
import tiktoken
encoding = tiktoken.get_encoding("cl100k_base")
# the encoding used by OpenAIs gpt-4,
# gpt-3.5-turbo, text-embedding-ada-002
```

```
encoding.encode("Do no harm.")
```

```
[5519, 912, 11682, 13]
```

(De)tokenization

tokenize : $\{0..255\}^* \rightarrow \{0..\text{max_token_value}\}^*$

$c_1 c_2 \dots c_n \mapsto t_1 t_2 \dots t_\ell$

decode is the inverse of tokenize.

In cl100k_base encoder `max_token_value` = 100276.

Strings are converted to a byte array first.

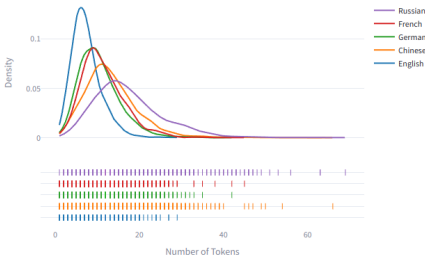
Tokenizer

Language Comparison

Median Token Length for `openai/gpt4`

| English | Chinese | German | French | Russian |
|---------|---------|--------|--------|---------|
| 7 | 12 | 10 | 11 | 16 |

Token Distribution



Example Texts

Randomly sample

| Language | ISO | Text | Num Tokens |
|----------|-------|-----------------------------------|------------|
| German | de-DE | ich möchte gute musik anhören | 8 |
| English | en-US | i want to hear great music | 6 |
| French | fr-FR | je veux entendre de la bonne musi | 9 |
| Russian | ru-RU | я хочу послушать отличную муз | 16 |
| Chinese | zh-CN | 我想听好听的音乐 | 10 |



Yennie Jun

Comparison done on parallel Alexa corpus^a.

Absolute average token lengths unimportant, only relative comparison.

^aFitzGerald et al. , "MASSIVE: A 1M-Example Multilingual Natural Language Understanding Dataset with 51 Typologically-Diverse Languages", *arXiv*, 2022

Natural Language Processing

Some long tokens

```
for t in [87644, 63570, 77621, 77002,  
         57950, 59862, 87378]:  
    print (encoding.decode([t]))
```

```
/*  
.translateAutoresizingMaskIntoConstraints  
ArgumentOutOfRangeException  
ABCDEFGHIJKLMNOPQRSTUVWXYZ  
NullPointerException  
radioactive  
sexkontakte  
*/
```

Find an exhaustive list at <https://gist.github.com/Yardanico/623b3092d0b707119f8c7d90a3596afe>.



Input Embedding

$$\text{Embedding} : \{0..\text{max_token_value}\} \rightarrow \mathbb{R}^d$$

- Is implemented as **one-hot encoding** followed by multiplication with a **weight matrix**.
- The **weight matrix is learned together** with the other parameters of the model.
- Embedding is applied to each token and yields output $X \in \mathbb{R}^{\ell \times d}$.



Positions in the Sequence

Attention and Input Embedding are Permutation Equivariant.

Permuting the tokens and therefore the rows of input embedding $X \in \mathbb{R}^{\ell \times d}$ permutes the rows of the result matrix: Let $P \in \{0, 1\}^{\ell \times \ell}$ be a permutation matrix.

$$\text{SelfAttention}(PX) = P \cdot \text{SelfAttention}(X)$$

Consequence of Permutation Equivariance for NLP

The class of input token sequences such as

`[[dog], [bytes], [man]]`

may depend on the order or tokens. The other layers are also permutation equivariant.



Positions in the Sequence

Attention and Input Embedding are Permutation Equivariant.

Permuting the tokens and therefore the rows of input embedding $X \in \mathbb{R}^{\ell \times d}$ permutes the rows of the result matrix: Let $P \in \{0, 1\}^{\ell \times \ell}$ be a permutation matrix.

$$\text{SelfAttention}(PX) = P \cdot \text{SelfAttention}(X)$$

Consequence of Permutation Equivariance for NLP

The class of input token sequences such as

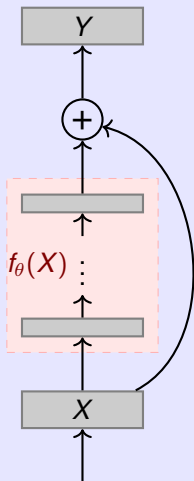
`[[dog], [bytes], [man]]`

may depend on the order or tokens. The other layers are also permutation equivariant.

Positional Embeddings

The original transformer paper¹ proposed to construct a **positional embedding** matrix $P \in \mathbb{R}^{\ell \times d}$ that neither depends on the data nor is trained and to **add** P to input embedding X .

Residual Block



$$Y = f_{\theta}(X) + X \quad (4)$$

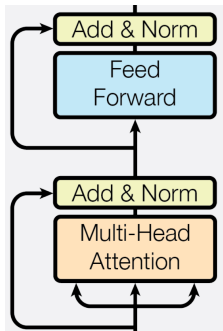
- f_{θ} is some sequence of layers or subnetwork with parameters.
- $f_{\theta}(X)$ must have the same shape as X . If this is not directly the case, this may be achieved with a matrix multiplication at the end of the layer sequence.
- “residual connection”
- Typically, it is easy to learn a near-identity function with (4) by choosing θ such that $f_{\theta} \approx 0$ and the same is not true for f_{θ} itself.
- Idea from image analysis (ResNet 2015) with deep CNNs.
- Safeguard to prevent that stacks of n residual blocks loose accuracy with increasing depth n .
- Can be used as a trick to construct invertible layers (see Reformer later).

Self-Attention Block (s-block)

$$A = \text{MHA}(X) \quad (\text{multi-head attention})$$

$$N = \text{LayerNormalization}(A + X) \quad (\text{includes residual connection})$$

Parameters: $W_Q^i, W_K^i, W_V^i \in \mathbb{R}^{d \times d_h}$ for $i = 1..h$



modified after Vaswani et al. , "Attention is all you need", arXiv, 2017

Feed-Forward Block (f-block)

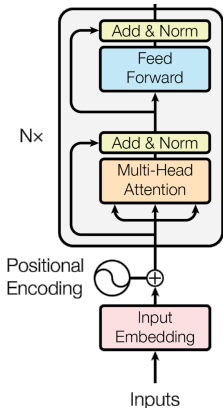
$$F = \text{FeedForward}(N)$$

$$Z = \text{LayerNormalization}(F + N) \quad (\text{includes residual connection})$$

Parameters: 2 weight matrices and 2 bias vectors

The Transformer Encoder Block

maps input X to output Z via one s- and one f-block (left).



Vaswani et al. , "Attention is all you need", arXiv, 2017

The Transformer Encoder

Let $T \in \{0 \dots \text{max_token_value}\}^\ell$ be the input token sequence.

$$Y = \text{Embedding}(T)$$

$$X = Y + P$$

$$Z = f^{(N)} \circ \dots \circ f^{(1)}(X)$$

$f^{(1)}, \dots, f^{(N)}$ are N transformer encoder blocks, each with their own set of (trainable) parameters.

$$Y, X, Z \in \mathbb{R}^{\ell \times d}.$$

GPT-3 hyperparameters

$N = 96$, $d = 12288$, $h = 96$ attention heads, $\ell = 2048$ (token limit)

Text embeddings

- We used a transformer encoder (`text-embedding-ada-002` of OpenAI) to embed texts:

$$e : \{0..255\}^* \rightarrow \mathbb{R}^{1536}$$

- one type of input texts were publication titles from one prof, e.g.

Valuation based on partnership in cross-company value chains

Consistency - and Congruency-Oriented Production Management

Financial valuation of start-up businesses with and without venture capital

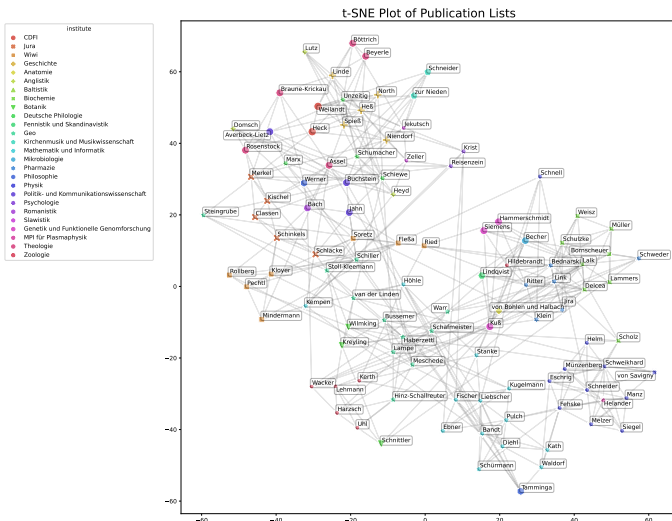
From the high Produce of the fine Art of Plagiarism - a 10 Step Guide

(Roland Rollberg, ABWL)

Cosine similarity of two texts u and v

$$s(u, v) := \frac{e(u)^T e(v)}{\|e(u)\| \cdot \|e(v)\|} \in [-1, 1]$$

Map of Greifswald Scientists (Praktikum Softwaretechnik 2023)



Hannah Thierfeldt, Nora Jungierck, Marcel Gabor, Fiona Bayer, Louisa Henze, Lukasz Reszcynski, MS:

<https://github.com/mslehre/text-embedding>

<http://141.53.34.56/> (VPN?)



1 Transformers

Attention

Encoder

Applications

Decoder

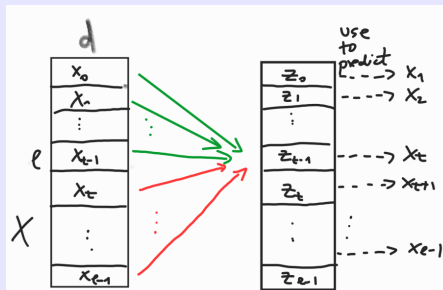
Predicting the Next Token

Use the sequence of tokens $x_0 \dots, x_{t-1}$ up to present time $t - 1$ to **predict the next token**:

$$P(X_t = \cdot | x_{t-1}, \dots, x_0).$$

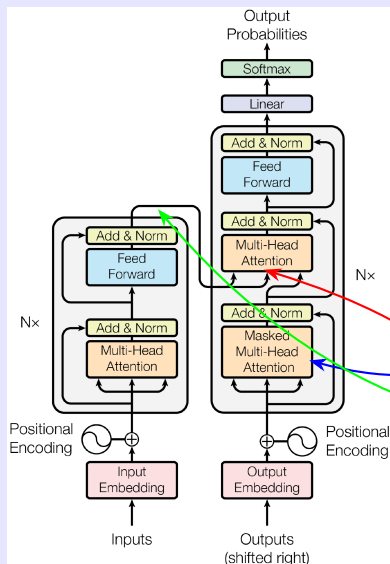
Reuse encoder, but ignore future: unidirectional (causal) attention

- z_{t-1} is a function of the tokens x_0, \dots, x_{t-1} seen so far (green arrows)
- as estimate of x_t is computed as a function of z_{t-1} , e.g. with a Dense layer
- if $j > i$, token i should not attend to token j (red arrows, ignored)



The Original Transformer ...

... uses encoder and decoder



- the **decoder** (right-hand side) **cross-attends** to the output of the **encoder** (left-hand side)
- cross-attention**: the queries correspond to the tokens of the output sequence (produced so far), the keys and values are from the embedding from the input sequence (e.g. prompt, **“context”**)

$x = \text{causal_self_attention}(x)$
 $x = \text{cross_attention}(x, \text{context})$



Transformers

- Attention
- Encoder
- Applications
- Decoder

Transformer Encoder and Decoder

Flow of information



Jakob Uszkoreit, <https://ai.googleblog.com/2017/08/transformer-novel-neural-network.html>

Generative Pre-Trained Transformer (GPT)

“GPT-1”^a

is a **decoder-only** transformer.

- **generative pre-training**, unsupervised on a large corpus of texts
$$-\sum_t \log P(x_t | x_{t-1}, \dots, x_0) \rightarrow \min$$
- starting from these parameters, GPT-1 was then trained for each of a set of supervised tasks (“fine-tuned”) with minimal extension of the architecture.

GPT-2^b

- With GPT-2 already, a **single model** was used to perform all 8 benchmarking tasks (**multitask learning**)
- task-specific prompt-design
- training: “we scraped all outbound links from **Reddit**, a social media platform, which received at least 3 karma”
- $\ell = 1024$

^aRadford et al., “Improving Language Understanding by Generative Pre-Training”, 2018

^bRadford et al., “Language Models are Unsupervised Multitask Learners”, 2019

Generative Pre-Trained Transformer (GPT)

GPT-3^a

- task-agnostic
- more parameters (175 billion)
- $\ell = 2048$

GPT-4^b

- fine-tuned using Reinforcement Learning from Human Feedback (RLHF)
- > 1 trillion parameters (10^{12})? (details unpublished)
- $\ell = 32768$

^aBrown et al., "Language Models are Few-Shot Learners", 2020

^b"GPT-4 Technical Report", 2023



Prompt to GPT4:

Compute **causal attention** for the following query matrix Q , key matrix K and values matrix V .

$$Q = \begin{bmatrix} [1, 0], \\ [0, 2], \\ [1, -2] \end{bmatrix}$$

$$K = \begin{bmatrix} [3, 4], \\ [-2, -3], \\ [1, 0] \end{bmatrix}$$

$$V = \begin{bmatrix} [3, 3], \\ [4, 4], \\ [2, 2] \end{bmatrix}$$

Please write numpy code to compute the causal attention.

Answer of GPT-4

python

 Copy code

```
import numpy as np
from scipy.special import softmax

# Define the query, key, and value matrices
Q = np.array([[1, 0], [0, 2], [1, -2]])
K = np.array([[3, 4], [-2, -3], [1, 0]])
V = np.array([[3, 3], [4, 4], [2, 2]])

# Compute the dot product of Q and K transpose
dot_product = np.dot(Q, K.T)

# Define the mask for the causal attention: each position
# can only attend to preceding positions and itself.
mask = np.tril(np.ones_like(dot_product)) * float('inf')
mask[mask == 0] = float('-inf')

# Apply the mask to the dot product
masked_dot_product = dot_product + mask

# Apply softmax to the masked dot product to get attention weights
attention_weights = softmax(masked_dot_product)

# Compute the final output as a weighted sum of the values
output = np.matmul(attention_weights, V)

print(output)
```

This code does the following:

1. It calculates the dot product of Q and the transpose of K. The resulting matrix contains the unnormalized attention scores.
2. It generates a mask to enforce the causality condition (a query can only attend to keys at the same position or earlier). A mask with the shape of the dot product is created. All elements in the lower triangle (including the diagonal) are filled with positive infinity, and all other elements are filled with negative infinity.
3. It applies the mask to the dot product matrix. This is done by adding the mask to the dot product.
4. It applies the softmax function to the masked dot product to get the attention weights.
5. It computes the final output as the matrix multiplication (dot product) of the attention weights and the V matrix. The output matrix is the result of the causal attention.

Answer of GPT-4

python

Copy code

```
import numpy as np
from scipy.special import softmax
```

```
# Define the query, key, and value matrices
Q = np.array([[1, 0], [0, 2], [1, -2]])
K = np.array([[3, 4], [-2, -3], [1, 0]])
V = np.array([[3, 3], [4, 4], [2, 2]])
```

```
# Compute the dot product of Q and K transpose
dot_product = np.dot(Q, K.T)
```

```
# Define the mask for the causal attention: each position
# can only attend to preceding positions and itself.
```

```
mask = np.tril(np.ones_like(dot_product)) * float('inf')
mask[mask == 0] = float('inf')
```

produces
NaN and error

incorrect &
not needed
anywhere

```
# Apply the mask to the dot product
masked_dot_product = dot_product + mask
```

+ M

```
# Apply softmax to the masked dot product to get attention weights
attention_weights = softmax(masked_dot_product)
```

```
# Compute the final output as a weighted sum of the values
output = np.matmul(attention_weights, V)
```

```
print(output)
```

this way the
whole matrix
axis=-1 sums to 1,
but each row
should sum to 1

5/8
points

instead:

$$M = (1 - \text{np.tril}(\dots)) * (-1e9)$$

to obtain

$$\begin{pmatrix} 0 & -1e9 & -1e9 \\ 0 & 0 & -1e9 \\ 0 & 0 & 0 \end{pmatrix}$$

or: use np.where

This code does the following:

1. It calculates the dot product of Q and the transpose of K. The resulting matrix contains the unnormalized attention scores. ✓
2. It generates a mask to enforce the causality condition (a query can only attend to keys at the same position or earlier). A mask with the shape of the dot product is created. All elements in the lower triangle (including the diagonal) are filled with positive infinity, and all other elements are filled with negative infinity. *good*
should be 0 should be -∞
3. It applies the mask to the dot product matrix. This is done by adding the mask to the dot product. ✓
4. It applies the softmax function to the masked dot product to get the attention weights. ✓
5. It computes the final output as the matrix multiplication (dot product) of the attention weights and the V matrix. The output matrix is the result of the causal attention. ✓