Transformer

Xiucai Ding

- Transformer is a neural network architecture that has fundamentally changed the approach to Artificial Intelligence.
- Go-to architecture for deep learning models, powering text-generative models like OpenAI's **GPT**, Meta's **Llama**, and Google's **Gemini**.
- Beyond text, Transformer is also applied in audio generation, image recognition, protein structure prediction, and even game playing, demonstrating its versatility across numerous domains.
- Fundamentally, text-generative Transformer models operate on the principle of **next-word prediction**: given a text prompt from the user, what is the *most probable next word* that will follow this input?
- The core innovation and power of Transformers lie in their use of self-attention mechanism, which allows them to process entire sequences and capture long-range dependencies more effectively than previous architectures.

GPT-2 small

- GPT-2 family of models are prominent examples of text-generative Transformers which have **124 million** parameters.
- **Embedding**: Text input is divided into smaller units called **tokens**, which can be words or subwords. These tokens are converted into numerical vectors called **embeddings**, which capture the semantic meaning of words.
- **Transformer Block:** is the fundamental building block of the model that processes and transforms the input data. Each block includes:
  - **Attention Mechanism**, the core component of the Transformer block. It allows tokens to communicate with other tokens, capturing contextual information and relationships between words.
  - **MLP (Multilayer Perceptron) Layer**, a feed-forward network that operates on each token independently. While the goal of the attention layer is to route information between tokens, the goal of the MLP is to refine each token's representation.
- **Output Probabilities**: The final linear and softmax layers transform the processed embeddings into probabilities, enabling the model to make predictions about the next token in a sequence.

```
!pip install transformers torch --quiet
```

[1] ✓ 3m 47.2s

```
from transformers import GPT2Tokenizer, GPT2LMHeadModel
import torch
import pandas as pd
```

[2] ✓ 23.8s

```
# Load GPT-2 Small model and tokenizer
tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
model = GPT2LMHeadModel.from_pretrained("gpt2")
model.eval();
```

[3] ✓ 1m 32.9s

Model	Params	Hidden	Layers	Heads	Per-head Dim
GPT-2 Small	124M	768	12	12	64
GPT-2 Medium	355M	1024	24	16	64
GPT-2 Large	774M	1280	36	20	64
GPT-2 XL	1.5B	1600	48	25	64
GPT-3	175B	12288	96	96	128

Embedding

- It transforms the text into a numerical representation that the model can work with
- 1) tokenize the input, 2) obtain token embeddings, 3) add positional information, and finally 4) add up token and position encodings to get the final embedding

Tokenization

- Tokenization is the process of breaking down the input text into smaller, more manageable pieces called tokens. These tokens can be a word or a subword. The full vocabulary of tokens is decided before training the model: GPT-2's vocabulary has 50,257 unique tokens.

```
tokens = tokenizer.tokenize(prompt)
token_ids = tokenizer.convert_tokens_to_ids(tokens)

# Create a DataFrame to visualize tokens and their corresponding IDs
token_df = pd.DataFrame({
    "Token": tokens,
    "Token ID": token_ids
})
token_df

## Byte Pair Encoding, space
```

[8] ✓ 0.0s

...

	Token	Token ID
0	Data	6601
1	Ġvisualization	32704
2	Ġem	795
3	powers	30132
4	Ġusers	2985
5	Ġto	284

Token Embedding

- GPT-2 (small) represents each token in the vocabulary as a 768-dimensional vector. These embedding vectors are stored in a matrix of shape (50,257, **768**), containing approximately 39 million parameters!

```python
# GPT-2 embeds input tokens using model.transformer.wte (word token embeddings)
embedding_matrix = model.transformer.wte.weight  # shape: [vocab_size, embedding_dim]

# Extract embeddings for the tokens in your prompt
embeddings = embedding_matrix[token_ids[0]]  # shape: [sequence_length, embedding_dim]
print(embeddings)
```

✓ 0.0s Python

```
tensor([[ 0.1346, -0.0544,  0.1117,  ..., -0.0849,  0.2777, -0.0391],
        [-0.0073, -0.0777,  0.0195,  ..., -0.0439, -0.0856,  0.2545],
        [ 0.0573, -0.0381,  0.0831,  ..., -0.0459,  0.1610,  0.0375],
        [-0.0464, -0.2388,  0.0537,  ..., -0.0751,  0.3154, -0.3336],
        [-0.1429, -0.0300,  0.0659,  ...,  0.0789,  0.0799, -0.0737],
        [-0.0084, -0.1018,  0.0309,  ..., -0.0158,  0.0990,  0.0790]],
       grad_fn=<IndexBackward0>)
```

```python
import pandas as pd

token_id_list = token_ids[0].tolist()
tokens = tokenizer.convert_ids_to_tokens(token_id_list)
df = pd.DataFrame(embeddings.detach().numpy(), index=tokens)
df.columns = [f"dim_{i}" for i in range(df.shape[1])]
df.head()
```

✓ 0.0s Python

	dim_0	dim_1	dim_2	dim_3	dim_4	dim_5	dim_6	dim_7	dim_8	dim_9	...	dim_758	dim_759	dim_760	dim_761	dim_762	dim_763	dim_7
Data	0.134590	-0.054377	0.111676	0.102781	-0.088301	0.114991	-0.220245	-0.002141	-0.069761	-0.148118	...	-0.051076	-0.202269	-0.099972	-0.160342	-0.268336	-0.373275	-0.0253
Ġvisualization	-0.007269	-0.077655	0.019525	0.095097	-0.155397	0.169577	-0.431230	0.054858	0.146526	-0.100686	...	0.144012	0.168718	-0.147809	-0.042908	-0.166734	-0.122340	0.0231
Ġem	0.057298	-0.038135	0.083134	0.080997	0.013071	0.133386	-0.315548	-0.038903	0.003984	0.033547	...	-0.028622	0.092855	-0.055978	-0.175792	-0.116575	0.156816	0.0086
powers	-0.046428	-0.238825	0.053749	-0.224896	-0.089782	-0.027846	-0.303783	-0.184255	-0.215250	0.127391	...	0.221928	-0.010380	-0.097017	0.114352	-0.170354	0.005602	-0.1808
Ġusers	-0.142855	-0.029995	0.065874	0.133380	-0.058403	0.106798	-0.336768	-0.113584	-0.005061	-0.059867	...	-0.021664	-0.200841	-0.049823	-0.219121	-0.216563	-0.072834	0.0040

5 rows × 768 columns

Positional Encoding

- The Embedding layer also encodes information about each token's position in the input prompt. GPT-2 trains its own positional encoding matrix from scratch, integrating it directly into the training process.

```
# Positional encoding matrix: [max_position_embeddings, hidden_dim]
positional_embeddings = model.transformer.wpe.weight
print(positional_embeddings.shape)  # typically (1024, 768)
```

✓ 0.0s

```
torch.Size([1024, 768])
```

```
position_ids = torch.arange(len(token_ids[0]), dtype=torch.long)
position_encoding = positional_embeddings[position_ids]
print(position_encoding.shape)  # (sequence_length, 768)
```

✓ 0.0s

```
torch.Size([6, 768])
```

Final Embedding

- Token and positional encodings to get the final embedding representation.
- This combined representation captures both the semantic meaning of the tokens and their position in the input sequence.

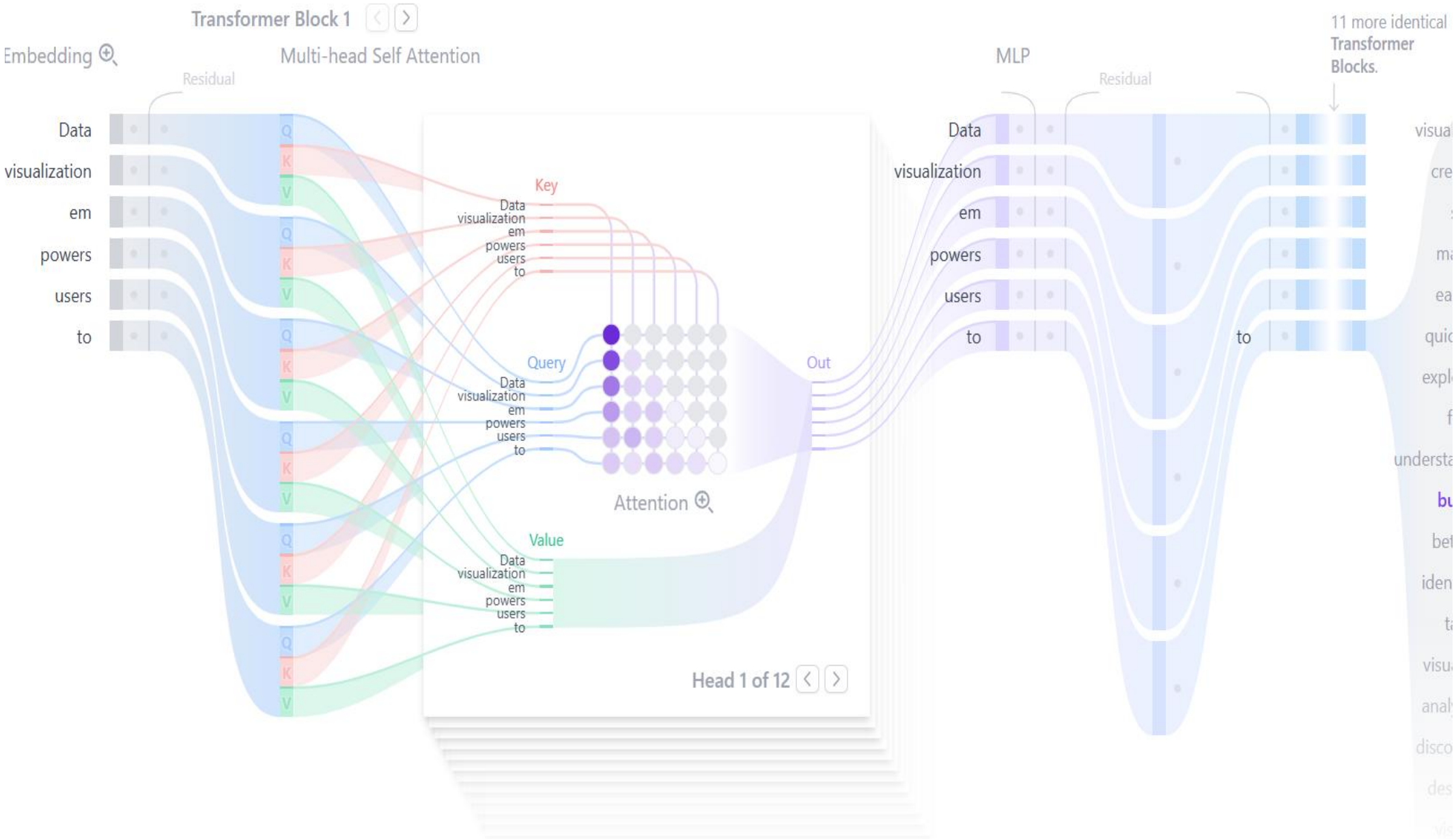
```
# Prompt and token IDs
prompt = "The cat sat"
token_ids = tokenizer.encode(prompt, return_tensors="pt")  # shape: [1, seq_len]

# Token embeddings
token_embeddings = model.transformer.wte(token_ids)  # [1, seq_len, 768]

# Position embeddings
position_ids = torch.arange(token_ids.size(1), dtype=torch.long).unsqueeze(0)  # [1, seq_len]
position_embeddings = model.transformer.wpe(position_ids)  # [1, seq_len, 768]


final_embeddings = token_embeddings + position_embeddings  # [1, seq_len, 768]

print(final_embeddings)
```

[34] ✓ 0.0s

```
tensor([[[-0.0874, -0.2177,  0.0685,  ...,  0.0208,  0.0183,  0.0576],
         [ 0.0339, -0.0172,  0.0691,  ...,  0.0367, -0.0310, -0.0697],
         [ 0.0249, -0.2003,  0.0754,  ...,  0.1283, -0.1179, -0.0668]]],
       grad_fn=<AddBackward0>)
```

Transformer Block

- The core of the Transformer's processing lies in the Transformer block, which comprises **multi-head self-attention** and a **Multi-Layer Perceptron layer**.
- Most models consist of multiple such blocks that are stacked sequentially one after the other. The token representations evolve through layers, from the first block to the last one, allowing the model to build up an intricate understanding of each token.
- This layered approach leads to higher-order representations of the input. The GPT-2 (small) model we are examining consists of 12 such blocks.


Transformer Block 1
Embedding
Multi-head Self Attention
Residual
MLP
Residual
11 more identical Transformer Blocks.
Data
visualization
em
powers
users
to
Q
K
V
Key
Query
Value
Out
Attention
Head 1 of 12

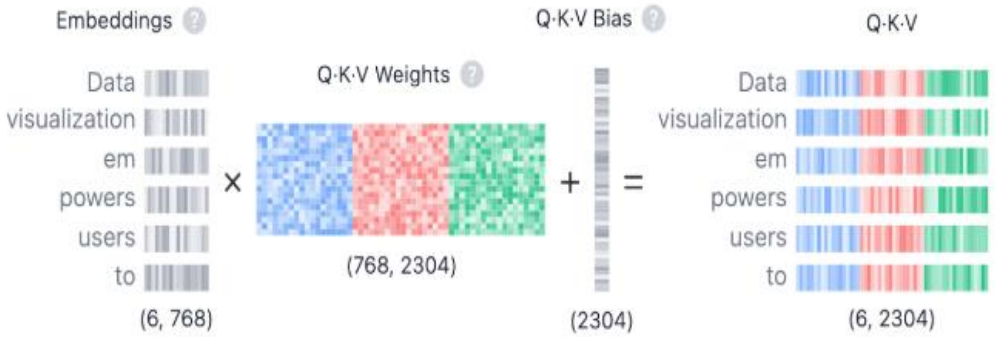
Multi-Head Self-Attention (MHSA)

- The self-attention mechanism enables the model to focus on relevant parts of the input sequence, allowing it to capture complex relationships and dependencies within the data.
- Step 1: Query, Key, and Value Matrices
- Step 2: Multi-Head Splitting
- Step 3: Masked Self-Attention
- Step 4: Output and Concatenation

Step 1: Query, Key, and Value Matrices

- **Query (Q)** is the search text you type in the seEach token's embedding vector is transformed into three vectors: Query (Q), Key (K), and Value (V). These vectors are derived by multiplying the input embedding matrix with learned weight matrices for Q, K, and V.
- **Key (K)** is the title of each web page in the search result window. It represents the possible tokens the query can attend to.
- **Value (V)** is the actual content of web pages shown. Once we matched the appropriate search term (Query) with the relevant results (Key), we want to get the content (Value) of the most relevant pages.

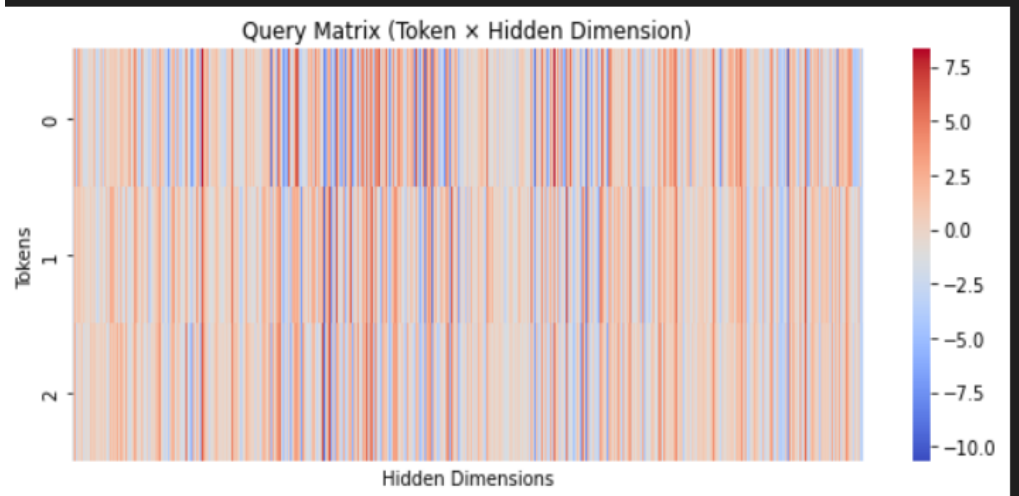
By using these QKV values, the model can calculate attention scores, which determine how much focus each token should receive when generating predictions.



$$QKV_{ij} = (\sum_{d=1}^{768} \text{Embedding}_{i,d} \cdot \text{Weights}_{d,j}) + \text{Bias}_j$$

```python
# Step 3: apply attention projection layer (which outputs Q || K || V)
qkv_combined = attn_block.c_attn(x)  # shape: [1, seq_len, 2304]

# Step 4: split into Q, K, V
q, k, v = qkv_combined[0].split(768, dim=-1)

import seaborn as sns
import numpy as np

plt.figure(figsize=(10, 4))
sns.heatmap(q.detach().numpy(), cmap="coolwarm", xticklabels=False)
plt.title("Query Matrix (Token x Hidden Dimension)")
plt.xlabel("Hidden Dimensions")
plt.ylabel("Tokens")
plt.show()
```

✓ 0.5s



Step 2: Multi-Head Splitting

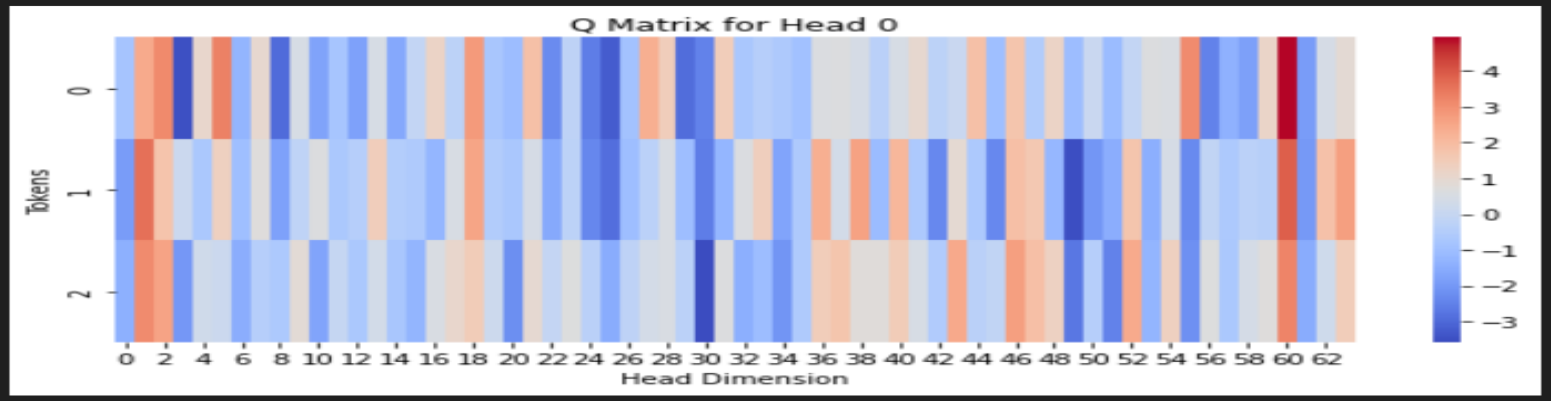
- Query, key, and Value vectors are split into multiple heads—in GPT-2 (small)'s case, into 12 heads. Each head processes a segment of the embeddings independently, capturing different syntactic and semantic relationships.

```
num_heads = attn_block.num_heads   # 12
head_dim = 768 // num_heads        # 64

# Reshape to [num_heads, seq_len, head_dim]
q_heads = q.view(-1, num_heads, head_dim).transpose(0, 1)  # [12, seq_len,
```

✓ 0.0s

```
import seaborn as sns

plt.figure(figsize=(10, 4))
sns.heatmap(q_heads[0].detach().numpy(), cmap="coolwarm")
plt.title("Q Matrix for Head 0")
plt.xlabel("Head Dimension")
plt.ylabel("Tokens")
plt.show()
```

✓ 0.1s

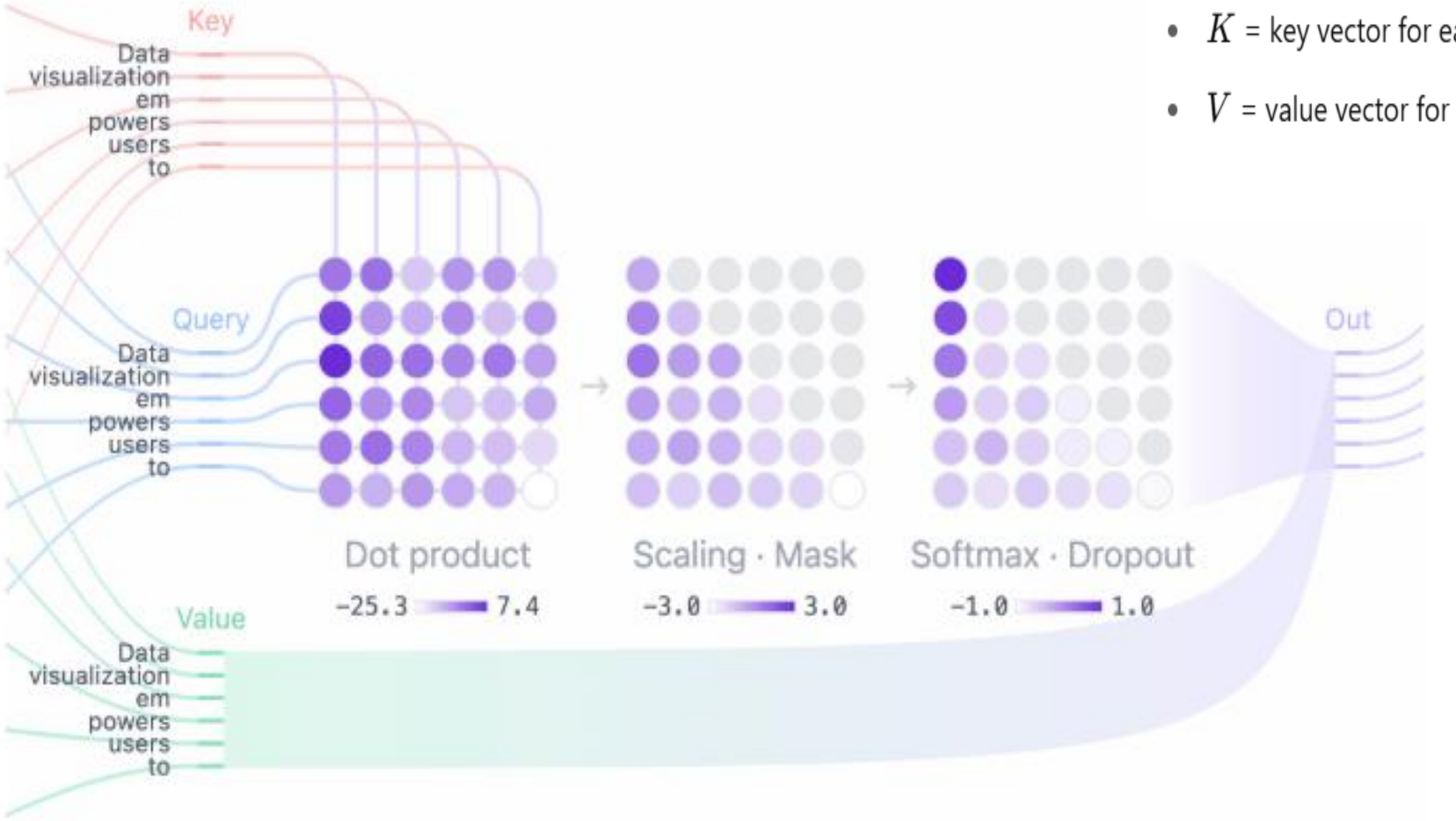


Step 3: Masked Self-Attention

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

Where:

- Q = query vector for each token
- K = key vector for each token
- V = value vector for each token



- **Attention Score**: The dot product of Query and Key matrices determines the alignment of each query with each key, producing a square matrix that reflects the relationship between all input tokens.
- **Masking**: A mask is applied to the upper triangle of the attention matrix to **prevent the model from accessing future tokens**, setting these values to negative infinity. The model needs to learn how to predict the next token without "peeking" into the future.
- **Softmax**: After masking, the attention score is converted into probability by the softmax operation which takes the exponent of each attention score. Each row of the matrix sums up to one and indicates the relevance of every other token to the left of it.

$$\text{scores}_{ij} = \begin{cases} Q_i \cdot K_j^T / \sqrt{d_k}, & \text{if } j \leq i \\ -\infty, & \text{if } j > i \end{cases}$$

Step 4: Output and Concatenation

- The model uses the masked self-attention scores and multiplies them with the value matrix to get the final output of the self-attention mechanism. GPT-2 has 12 self-attention heads, each capturing different relationships between tokens. The outputs of these heads are concatenated and passed through a linear projection.

```python
attn_block = model.transformer.h[0].attn  # Layer 0
# reshape Q/K/V to multi-head format
def reshape_to_heads(x, num_heads):
    batch_size, seq_len, hidden_size = x.shape
    head_dim = hidden_size // num_heads
    return x.view(batch_size, seq_len, num_heads, head_dim).permute(0, 2, 1, 3)

qkv = attn_block.c_attn(final_embeddings)  # [1, seq_len, 2304]
q, k, v = qkv.split(768, dim=2)

q = reshape_to_heads(q, 12)
k = reshape_to_heads(k, 12)
v = reshape_to_heads(v, 12)

import torch.nn.functional as F

d_k = q.shape[-1]
attn_weights = torch.matmul(q, k.transpose(-2, -1)) / d_k**0.5  # [1, 12, seq_len, seq_len]
attn_probs = F.softmax(attn_weights, dim=-1)  # Apply softmax
head_outputs = torch.matmul(attn_probs, v)    # [1, 12, seq_len, 64]
# Recombine all heads: [1, seq_len, 768]
concat_output = head_outputs.permute(0, 2, 1, 3).contiguous().view(1, -1, 768)
final_output = attn_block.c_proj(concat_output)  # [1, seq_len, 768]

print(final_output)
```

✓ 0.0s

```
tensor([[[ 0.5147,  0.9719,  4.1278,  ..., -0.0135,  0.2310,  0.0859],
         [ 0.4348,  2.4321,  3.6738,  ..., -0.1459,  0.3845,  0.2591],
         [-1.8778,  2.9142,  4.3289,  ..., -0.2248,  0.3531,  0.3275]]],
       grad_fn=<ViewBackward0>)
```

MLP: Multi-Layer Perceptron

- After the multiple heads of self-attention capture the diverse relationships between the input tokens, the concatenated outputs are passed through the Multilayer Perceptron (MLP) layer to enhance the model's representational capacity.
- The MLP block consists of two linear transformations with a GELU activation function in between. The first linear transformation increases the dimensionality of the input four-fold from 768 to 3072.
- The second linear transformation reduces the dimensionality back to the original size of 768, ensuring that the subsequent layers receive inputs of consistent dimensions.
- Unlike the self-attention mechanism, the MLP processes tokens independently and simply map them from one representation to another.

The MLP is a **position-wise feedforward network**:

$$\text{MLP}(x) = \text{GEGLU}(xW_1 + b_1)W_2 + b_2$$

- Input: vector of size **768**
- Hidden layer: **3072 dimensions**
- Output: back to **768**

This is **applied independently to each token**.

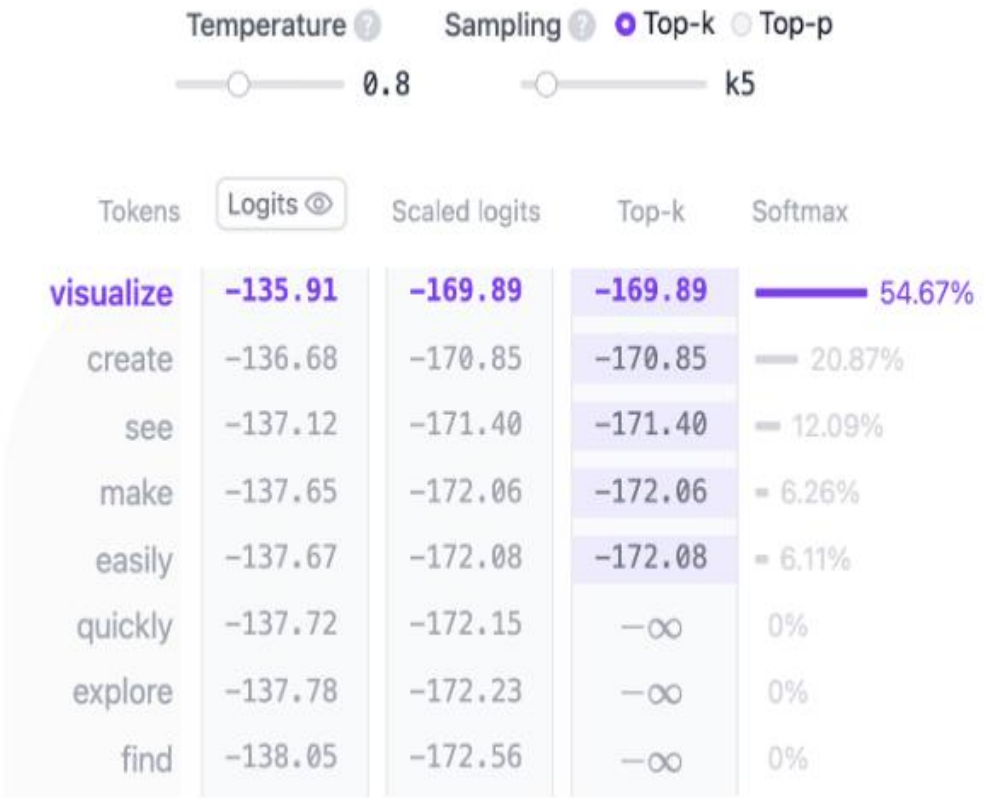
```
mlp_block = model.transformer.h[0].mlp
mlp_output = mlp_block(final_output)  # shape: [1, seq_len, 768]
print(mlp_block)
```

✓ 0.0s

```
GPT2MLP(
  (c_fc): Conv1D(nf=3072, nx=768)
  (c_proj): Conv1D(nf=768, nx=3072)
  (act): NewGELUActivation()
  (dropout): Dropout(p=0.1, inplace=False)
)
```

Output Probabilities

- After the input has been processed through all Transformer blocks, the output is passed through the final linear layer to prepare it for token prediction.
- This layer projects the final representations into a 50,257 dimensional space, where every token in the vocabulary has a corresponding value called **logit**.
- Any token can be the next word, so this process allows us to simply rank these tokens by their likelihood of being that next word.
- We then apply the softmax function to convert the logits into a probability distribution that sums to one. This will allow us to sample the next token based on its likelihood.




```python
prompt="Data visualization empowers users to"
inputs = tokenizer(prompt, return_tensors="pt")

with torch.no_grad():
    outputs = model(**inputs)
    logits = outputs.logits

# Extract logits for the last token in the prompt
last_token_logits = logits[0, -1, :]
#probabilities = torch.softmax(last_token_logits, dim=-1)
temperature = 0.7 # Try 0.7, 1.0, or 1.5
scaled_logits = last_token_logits / temperature
probabilities = torch.softmax(scaled_logits, dim=-1)

# Get top 5 predictions
top_k = torch.topk(probabilities, k=5)
top_k_words = [tokenizer.decode([idx]) for idx in top_k.indices]

# Display results
results_df = pd.DataFrame({
    "Token": top_k_words,
    "Probability": top_k.values.numpy()
})
results_df["Probability"] = results_df["Probability"].round(4)
results_df
```

✓ 0.0s

	Token	Probability
0	visualize	0.3958
1	create	0.1318
2	see	0.0707
3	make	0.0333
4	easily	0.0324

- **The temperature** hyperparameter plays a critical role in this process. Mathematically speaking, it is a very simple operation: model output logits are simply divided by the temperature:

➢ temperature = 1: Dividing logits by one has no effect on the softmax outputs.

➢ temperature < 1: Lower temperature makes the model more confident and deterministic by sharpening the probability distribution, leading to more predictable outputs.

➢ temperature > 1: Higher temperature creates a softer probability distribution, allowing for more randomness in the generated text – what some refer to as model “creativity”.

- **top-k sampling**: Limits the candidate tokens to the top k tokens with the highest probabilities, filtering out less likely options.
- **top-p sampling**: Considers the smallest set of tokens whose cumulative probability exceeds a threshold p, ensuring that only the most likely tokens contribute while still allowing for diversity.