Pre-Training and Fine-Tuning Large Language Models via Unidirectional Modeling for Text Generation

Ruijun Feng
Program Analysis Group
University of New South Wales

03/07/2024

What is large language model

- Large language model (LLM) is an artificial neural network, usually with **millions or billions of parameters**, trained on **vast amounts of text data** to understand and generate human-like language.

Unidirectional modelling

- The LLM only see text in **one direction**.

Unidirectional modelling

- The LLM only see text in **one direction**.
- **Predict the next word** in a sequence based only on the **preceding context**.

Unidirectional modelling

- The LLM only see text in **one direction**.
- **Predict the next word** in a sequence based only on the **preceding context**.

"The cat sits on the mat."→"The cat [MASK]"

Unidirectional modelling

- The LLM only see text in **one direction**.
- **Predict the next word** in a sequence based only on the **preceding context**.
- Famous models: GPT, Llama, etc.

Focus of this presentation

- Basic components used in LLM, including **Tokenization**, **Embedding layer**, and **Self-Attention layer**.

Focus of this presentation

- Basic components used in LLM, including **Tokenization**, **Embedding layer**, and **Self-Attention layer**.
- Process of Pre-Training and Fine-Tuning LLM via **Teacher Forcing**.

Focus of this presentation

- Basic components used in LLM, including **Tokenization**, **Embedding layer**, and **Self-Attention layer**.
- Process of Pre-Training and Fine-Tuning LLM via **Teacher Forcing**.
- Examples based on GPT2.

Tokenization

- The purpose of tokenization is to convert **strings into integers**, so the LLM can read them.

Tokenization

- Tokenization will cut a **sentence** into **tokens**, then use a **dictionary (vocabulary)** to store the mapping relationships between **tokens** and their corresponding **integer ids**.

Tokenization

- Suppose we are using code data to pre-train an LLM, the code fragment looks like this:

```
def add(a, b):
    return a + b
```

Tokenization

- The string format with "\n" of the code fragment:

```
def add(a, b):\n    return a + b
```

Tokenization

- The string format with "\n" of the code fragment:

def add(a, b):\n   return a + b

- After tokenization via GPT2 ("_" means space):

["def", "_add", "(", "a", ",", "_b", "):", "\n", "_", "_", "_", "_return", "_a", "_+", "_b"]

Tokenization

- Vocabulary for each token:

{"**def**": 4299, "_**add**": 751, "**(**": 7, "**a**": 64, "**,**": 11, "**_b**": 275, "**):**": 2599, "**\n**": 198, "**_**": 220, "_**return**": 1441, "**_a**": 257, "**_+**": 1343}

Tokenization

- The string format with escape characters "\n" of the code fragment:

def add(a, b):\n   return a + b

- After tokenization via GPT2 ("_" means space):

["def", "_add", "(", "a", ",", "_b", "):", "\n", "_", "_", "_", "_return", "_a", "_+", "_b"]

- Integer ids:

[4299, 751, 7, 64, 11, 275, 2599, 198, 220, 220, 220, 1441, 257, 1343, 275]

Tokenization

- To encode more semantic relationships, each integer id will be converted into an **embedding vector via embedding layer**.

Embedding layer

- What is embedding layer in LLM?

Embedding layer

- Embedding layer is a **list of trainable vectors**. Each vectors have the **same dimensions**.

vector 0: [[0.1, -0.2, 0.3, 0.4, ..., 0.6],
vector 1: [0.7, -0.8, 0.9, -1.0, ..., -1.2],
vector 2: [-1.3, 1.4, -1.5, 1.6, ..., 1.8],
... ...,
vector n: [2.5, -2.6, 2.7, -2.8, ..., -3.0]]

Embedding layer

- There are two types of embedding layers used in GPT2:
  - Word embedding layer
  - Positional embedding layer

Word embedding layer

- Integer id is the index, use it to index the list in the word embedding layer.
- The corresponding vector is the vectorized representation of that token.

Token: #

Integer id: 2 ⇒

vector 0: [[0.1, -0.2, 0.3, 0.4, ..., 0.6],
vector 1: [0.7, -0.8, 0.9, -1.0, ..., -1.2],
vector 2: [-1.3, 1.4, -1.5, 1.6, ..., 1.8],
... ...,
vector n: [2.5, -2.6, 2.7, -2.8, ..., -3.0]]

Word embedding layer

- "**def**" is a token, its integer id is "**4299**".
- The 4299-th vector of the word embedding layer is the word embedding vector of "**def**".

```
• # Get the embedding vector of "def"
• word_embedding = word_embedding_layer(torch.tensor([4299]))
```

Word embedding layer

```
# Declare a word embedding layer
word_embedding_layer = torch.nn.Embedding(50257, 768)
# Initialize a token ids tensor
token_ids = torch.tensor([4299, 751, 7, 64, 11, 275,
2599, 198, 220, 220, 220, 1441, 257, 1343, 275])
# Get the embedding vectors of the token ids
word_embeddings = word_embedding_layer(token_ids)
```

Word embedding layer

	Token	Token ID	Word Embedding Vector
0	def	4299	[-0.075, -0.083, 0.146, ..., -0.057]
1	_add	751	[0.129, -0.006, 0.129, ..., 0.111]
2	(	7	[-0.130, -0.212, 0.132, ..., -0.075]
3	a	64	[-0.176, -0.099, 0.206, ..., -0.055]
4	,	11	[0.011, -0.003, 0.032, ..., -0.060]
5	_b	275	[0.047, -0.018, 0.062, ..., 0.132]
6	):	2599	[-0.080, -0.217, -0.003, ..., 0.116]
7	\n	198	[-0.001, 0.018, 0.053, ..., -0.035]
8	_	220	[0.096, -0.091, 0.085, ..., 0.126]
9	_	220	[0.096, -0.091, 0.085, ..., 0.126]
10	_	220	[0.096, -0.091, 0.085, ..., 0.126]
11	_return	1441	[0.029, -0.021, 0.103, ..., -0.124]
12	_a	257	[-0.051, 0.006, 0.047, ..., -0.038]
13	_+	1343	[-0.045, -0.093, 0.013, ..., -0.112]
14	_b	275	[0.047, -0.018, 0.062, ..., 0.132]

Word embedding layer

- Word embedding layer **cannot describe the relative positions** between different tokens.

Word embedding layer

	Token	Token ID	Word Embedding Vector
0	def	4299	[-0.075, -0.083, 0.146, ..., -0.057]
1	_add	751	[0.129, -0.006, 0.129, ..., 0.111]
2	(	7	[-0.130, -0.212, 0.132, ..., -0.075]
3	a	64	[-0.176, -0.099, 0.206, ..., -0.055]
4	,	11	[0.011, -0.003, 0.032, ..., -0.060]
5	_b	275	[0.047, -0.018, 0.062, ..., 0.132]
6	):	2599	[-0.080, -0.217, -0.003, ..., 0.116]
7	\n	198	[-0.001, 0.018, 0.053, ..., -0.035]
8	_	220	[0.096, -0.091, 0.085, ..., 0.126]
9	_	220	[0.096, -0.091, 0.085, ..., 0.126]
10	_	220	[0.096, -0.091, 0.085, ..., 0.126]
11	_return	1441	[0.029, -0.021, 0.103, ..., -0.124]
12	_a	257	[-0.051, 0.006, 0.047, ..., -0.038]
13	_+	1343	[-0.045, -0.093, 0.013, ..., -0.112]
14	_b	275	[0.047, -0.018, 0.062, ..., 0.132]

Positional embedding layer

- Positional embedding layer aims to describe the **relative positions** between different tokens using the **position index**.

Positional embedding layer

```
# Declare a position embedding layer
position_embedding_layer = torch.nn.Embedding(1024,
768)
```

Positional embedding layer

- List of tokens:

["def", "_add", "(", "a", ",", "_b", "):", "\n", "_", "_", "_", "_return", "_a", "_+", "_b"]

Positional embedding layer

- List of tokens:

["**def**", "_**add**", "**(**", "**a**", "**,**", "_**b**", "**):**", "**\n**", "_", "_", "_", "_**return**", "_**a**", "_**+**", "_**b**"]

- The tokenized code fragment has 15 tokens, so the position index is [0, 1, …, 14].

Positional embedding layer

- List of tokens:

["**def**", "_**add**", "**(**", "**a**", "**,**", "_**b**", "**):**", "**\n**", "_", "_", "_", "_**return**", "_**a**", "_**+**", "_**b**"]

- The tokenized code fragment has 15 tokens, so the position index is [0, 1, …, 14].
- "**def**" and 0, "_**add**" and 1, …, "_**b**" and 14

Positional embedding layer

```
# Declare a position embedding layer
position_embedding_layer = torch.nn.Embedding(1024,
768)
# Initilize a postion index from 0 to 14
position_index = torch.arange(15)
```

Positional embedding layer

```
# Declare a position embedding layer
position_embedding_layer = torch.nn.Embedding(1024,
768)
# Initilize a postion index from 0 to 14
position_index = torch.arange(15)
# Get the embedding vectors of the position index
position_embeddings =
position_embedding_layer(position_index)
```

Positional embedding layer

	Position Index	Token	Position Embedding Vector
0	0	def	[-0.019, -0.197, 0.004, ..., 0.054]
1	1	_add	[0.024, -0.054, -0.095, ..., -0.000]
2	2	(	[0.004, -0.085, 0.055, ..., -0.021]
3	3	a	[-0.000, -0.074, 0.106, ..., -0.007]
4	4	,	[0.008, -0.025, 0.127, ..., -0.007]
5	5	_b	[0.010, -0.034, 0.131, ..., -0.007]
6	6	):	[0.003, -0.021, 0.120, ..., -0.003]
7	7	\n	[0.003, -0.003, 0.117, ..., -0.007]
8	8	_	[-0.001, -0.002, 0.111, ..., -0.010]
9	9	_	[0.005, 0.002, 0.118, ..., -0.006]
10	10	_	[0.002, 0.006, 0.100, ..., -0.006]
11	11	_return	[-0.004, 0.017, 0.107, ..., -0.006]
12	12	_a	[0.000, 0.017, 0.097, ..., -0.008]
13	13	_+	[0.004, 0.020, 0.105, ..., 0.000]
14	14	_b	[0.001, 0.023, 0.096, ..., -0.002]

Final input embedding

- By adding the **word embedding vectors** and **positional embedding vectors**, we have the input embedding that encodes all token information.

```
# Combine the word and position embeddings
final_embeddings = word_embeddings + position_embeddings
```

Final input embedding

- In GPT2, the size of vocabulary is set to **50257**.
- The maximum number of position indexes is set to **1024**.
- The dimension of the encoded embedding vector is **768**.

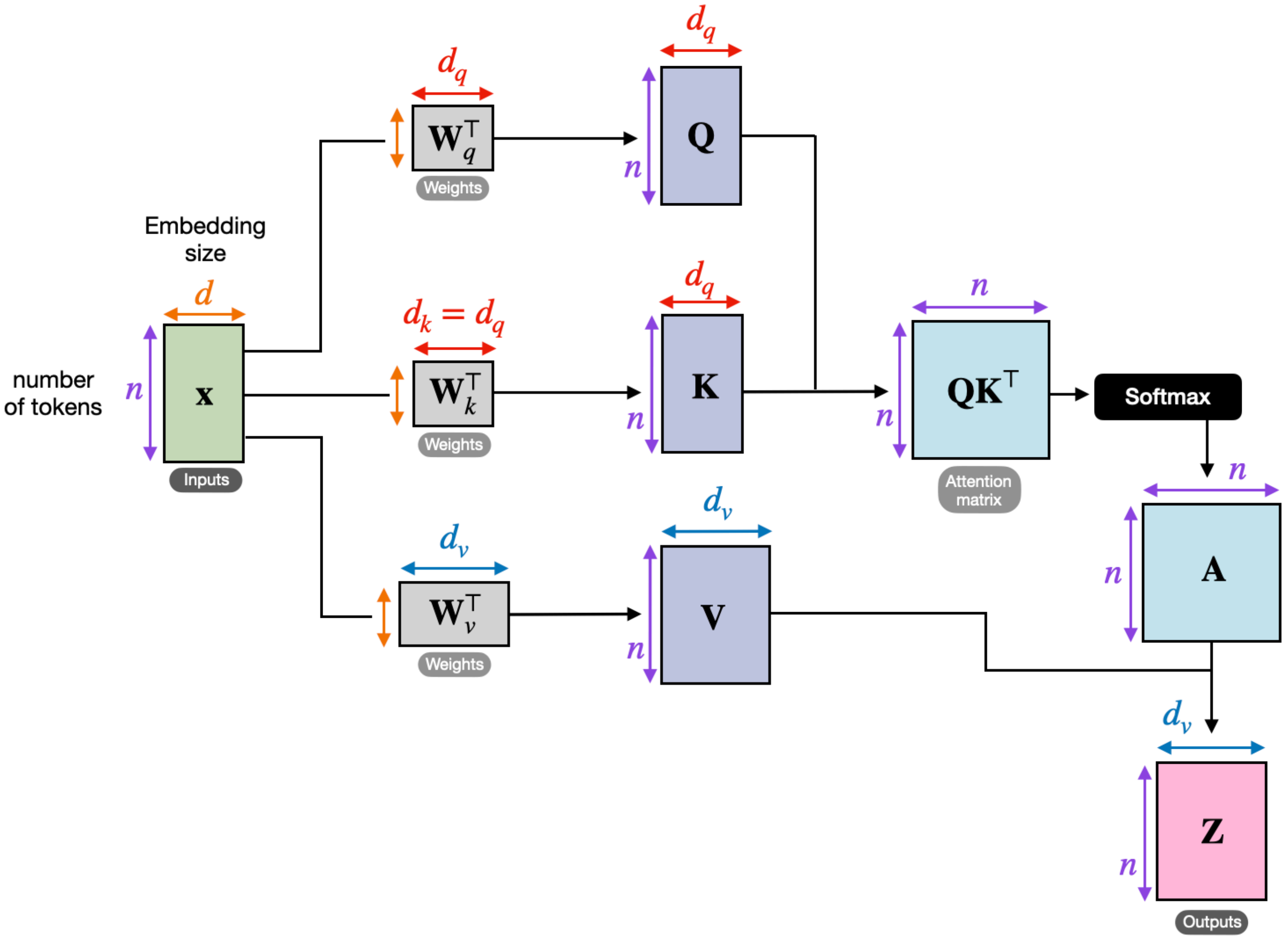
Self-attention layer

- Self-attention layer is used to calculate the relationships between different tokens using the input embedding.

Self-attention layer

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

Self-attention layer

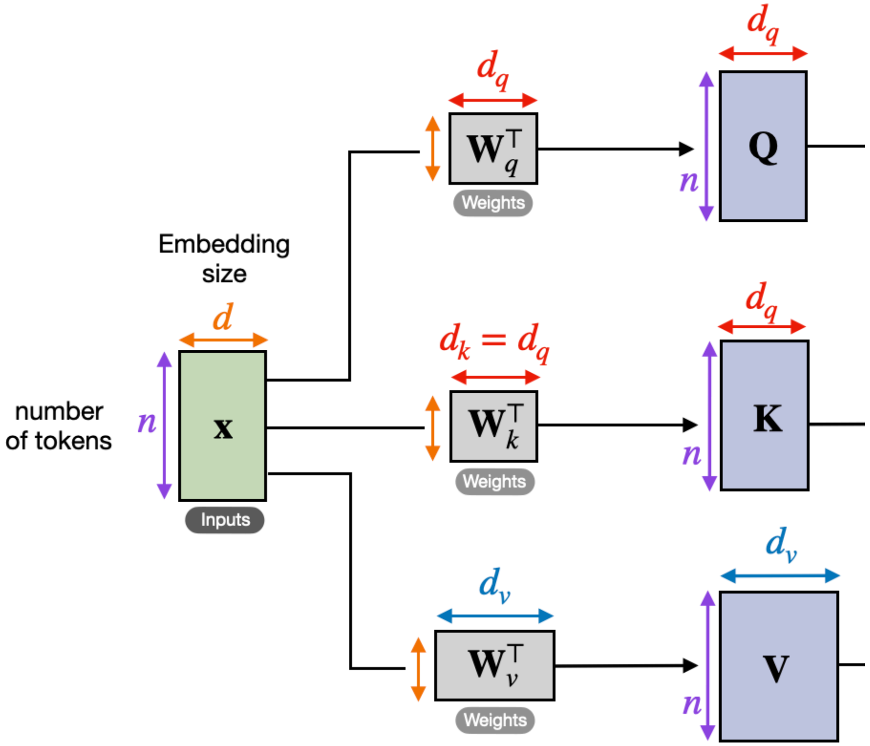


Self-attention layer—Q, K, V

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

Self-attention layer—Q, K, V

- First, three **separate** linear layers are used to encode the input embeddings into **hidden vectors**, denoted as **query (Q)**, **key (K)**, and **value (V)**.



Self-attention layer—Q, K, V

- **Query (**Q**)** is the content that is being looked for.
- **Key (**K**)** is reference of the content.
- **Value (**V**)** is the content that is being searched.

Self-attention layer—Q, K, V

```python
# Define linear layers for Q, K, V
hidden_dim = 768 # Same as the dimension of embedding vectors
linear_q = nn.Linear(hidden_dim, hidden_dim)
linear_k = nn.Linear(hidden_dim, hidden_dim)
linear_v = nn.Linear(hidden_dim, hidden_dim)
# Apply linear transformations to get Q, K, V
Q = linear_q(final_embeddings)
K = linear_k(final_embeddings)
V = linear_v(final_embeddings)
```

Self-attention layer—Q, K, V

- Each Q, K, V contains 15 vectors (corresponding to the number of tokens in the code fragment).

	Token	Q (15, 768)
0	def	[1.188, -4.549, ..., 2.721]
1	_add	[2.478, -1.816, ..., -1.107]
2	(	[1.280, -1.915, ..., 0.712]
3	a	[1.466, 0.061, ..., -0.108]
4	,	[0.958, -0.852, ..., -0.022]
5	_b	[1.282, -1.552, ..., -0.977]
6	):	[0.879, -2.032, ..., 0.415]
7	\n	[0.315, -0.799, ..., 0.637]
8	_	[0.947, -0.437, ..., -0.532]
9	_	[0.771, -0.351, ..., -0.607]
10	_	[0.768, -0.359, ..., -0.653]
11	_return	[0.007, -0.775, ..., 0.572]
14	_b	[0.718, -1.453, ..., -1.470]

	Token	K (15, 768)
0	def	[-1.083, 1.734, ..., -0.694]
1	_add	[0.081, 1.011, ..., -1.701]
2	(	[-1.008, -1.505, ..., -1.077]
3	a	[-1.647, -1.664, ..., -1.361]
4	,	[0.204, -0.925, ..., -1.277]
5	_b	[0.094, -1.595, ..., -1.411]
6	):	[0.068, -0.975, ..., -1.570]
7	\n	[0.296, -1.048, ..., -0.775]
8	_	[-1.036, -1.265, ..., -0.690]
9	_	[-1.010, -1.216, ..., -0.721]
10	_	[-0.938, -1.202, ..., -0.740]
11	_return	[0.065, -0.034, ..., -1.387]
14	_b	[0.570, -1.329, ..., -1.370]

	Token	V (15, 768)
0	def	[0.092, -0.522, ..., 0.589]
1	_add	[0.505, 0.107, ..., -0.431]
2	(	[-0.038, -0.236, ..., -0.596]
3	a	[0.255, -0.025, ..., -0.514]
4	,	[0.157, 0.019, ..., -0.276]
5	_b	[-0.023, -0.046, ..., -0.314]
6	):	[0.220, 0.038, ..., -0.184]
7	\n	[0.169, 0.043, ..., -0.396]
8	_	[0.222, 0.114, ..., -0.077]
9	_	[0.231, 0.129, ..., -0.094]
10	_	[0.233, 0.137, ..., -0.086]
11	_return	[0.143, 0.072, ..., -0.076]
14	_b	[-0.014, 0.054, ..., -0.275]

Self-attention layer—Q, K, V

- Each vector is 768-dimensional (corresponding to the dimension of embeddings).

	Token	Q (15, 768)
0	def	[1.188, -4.549, ..., 2.721]
1	_add	[2.478, -1.816, ..., -1.107]
2	(	[1.280, -1.915, ..., 0.712]
3	a	[1.466, 0.061, ..., -0.108]
4	,	[0.958, -0.852, ..., -0.022]
5	_b	[1.282, -1.552, ..., -0.977]
6	):	[0.879, -2.032, ..., 0.415]
7	\n	[0.315, -0.799, ..., 0.637]
8	_	[0.947, -0.437, ..., -0.532]
9	_	[0.771, -0.351, ..., -0.607]
10	_	[0.768, -0.359, ..., -0.653]
11	_return	[0.007, -0.775, ..., 0.572]
14	_b	[0.718, -1.453, ..., -1.470]

	Token	K (15, 768)
0	def	[-1.083, 1.734, ..., -0.694]
1	_add	[0.081, 1.011, ..., -1.701]
2	(	[-1.008, -1.505, ..., -1.077]
3	a	[-1.647, -1.664, ..., -1.361]
4	,	[0.204, -0.925, ..., -1.277]
5	_b	[0.094, -1.595, ..., -1.411]
6	):	[0.068, -0.975, ..., -1.570]
7	\n	[0.296, -1.048, ..., -0.775]
8	_	[-1.036, -1.265, ..., -0.690]
9	_	[-1.010, -1.216, ..., -0.721]
10	_	[-0.938, -1.202, ..., -0.740]
11	_return	[0.065, -0.034, ..., -1.387]
14	_b	[0.570, -1.329, ..., -1.370]

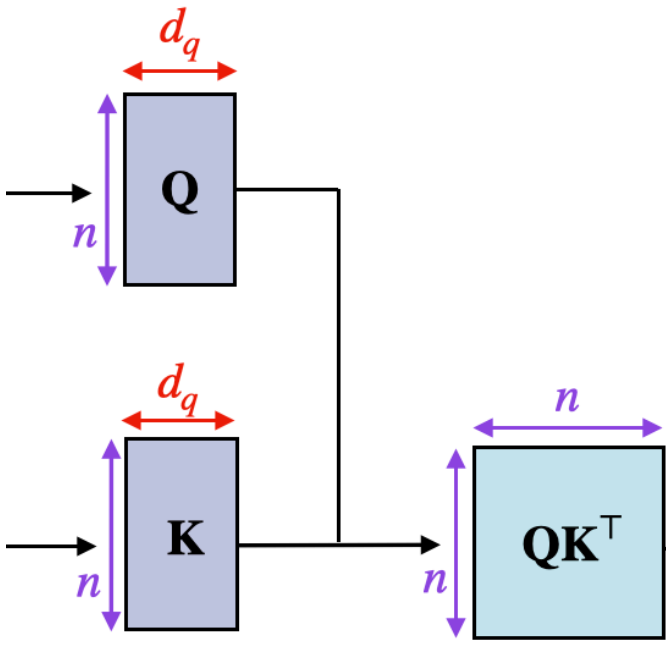
	Token	V (15, 768)
0	def	[0.092, -0.522, ..., 0.589]
1	_add	[0.505, 0.107, ..., -0.431]
2	(	[-0.038, -0.236, ..., -0.596]
3	a	[0.255, -0.025, ..., -0.514]
4	,	[0.157, 0.019, ..., -0.276]
5	_b	[-0.023, -0.046, ..., -0.314]
6	):	[0.220, 0.038, ..., -0.184]
7	\n	[0.169, 0.043, ..., -0.396]
8	_	[0.222, 0.114, ..., -0.077]
9	_	[0.231, 0.129, ..., -0.094]
10	_	[0.233, 0.137, ..., -0.086]
11	_return	[0.143, 0.072, ..., -0.076]
14	_b	[-0.014, 0.054, ..., -0.275]

Self-attention layer —Attention Scores

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

Self-attention layer—Attention Scores

- Use Q and K to get attention scores matrix, which indicates the relevance of each token in the sequence (key) to the current token (query).



Self-attention layer—Attention Scores

- Calculate QK^T and get a $n \times n$ scaled attention scores matrix, where the n is the number of tokens in the code fragment (i.e., 15).

```
# Calculate the matrix multiplication between Q and K^T
attention_scores = torch.matmul(Q, K.transpose(-2, -1))
```

Self-attention layer—Attention Scores

- First line describes the relationships of "**def**" token to other tokens.

	def	_add	(	a	,	_b	):	\n	_	_	_	_return	_a	_+	_b
def	1221.9	410.3	394.8	432.0	497.4	531.8	423.4	473.5	589.4	588.0	589.4	445.1	529.3	547.6	534.9
_add	339.1	555.8	471.9	424.8	378.1	420.2	374.6	328.0	294.6	285.0	279.3	350.6	277.5	297.0	341.3
(	262.5	382.1	401.5	373.3	341.1	374.7	339.9	306.7	284.1	275.1	271.1	302.5	254.7	283.6	311.0
a	291.6	355.9	348.6	476.9	341.1	357.1	303.1	344.4	281.3	274.0	268.4	289.6	297.7	279.5	288.8
,	246.4	327.8	344.1	330.8	374.4	354.0	253.3	305.9	259.9	249.2	244.0	223.3	265.0	240.4	280.3
_b	338.1	345.3	351.6	377.2	385.3	442.3	324.2	342.4	323.6	313.2	307.2	287.1	313.7	276.6	369.8
):	450.5	343.7	303.7	329.4	296.9	335.4	448.7	307.3	300.0	292.2	288.3	242.0	267.1	298.9	279.2
\n	308.8	309.5	313.9	316.5	328.2	329.2	264.2	350.8	263.8	257.2	252.1	229.2	251.2	239.4	270.4
_	419.0	275.6	285.2	302.4	309.8	358.1	295.8	270.5	349.8	343.0	339.2	258.8	257.4	289.3	308.4
_	422.1	268.2	275.0	293.6	299.0	348.6	287.2	261.7	343.5	337.3	333.5	252.7	251.5	282.7	301.8
_return	456.9	301.8	277.2	314.2	284.0	317.6	267.9	309.5	316.1	310.0	306.4	342.9	252.7	264.0	271.7
_a	433.5	283.1	251.3	278.8	287.4	300.1	203.5	267.9	253.9	250.0	247.4	207.1	284.2	231.1	267.8
_+	372.1	245.7	242.7	265.9	251.2	282.2	241.0	258.1	270.4	264.5	260.4	194.8	220.5	306.2	236.4
_b	373.8	282.7	273.8	305.1	314.0	379.0	261.3	286.4	282.4	275.6	270.5	239.9	279.4	230.2	326.7

Self-attention layer—Attention Scores

- The variance of the attention scores is too big (`8936.96`).
- **Vanishing gradient problem**: the gradients become too small to effectively update the parameters during training.

```
>>> # Calculate the variance of the attention scores
>>> variance = torch.var(attention_scores)
>>> print("\nVariance of Attention Scores:", variance.item())
>>> Variance of Attention Scores: 8936.9609375
```

Self-attention layer—Scaled Attention Scores

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

Self-attention layer—Scaled Attention Scores

- Calculate QK^T/ sqrt(d) to scale the attention scores matrix, where d is the hidden dim (i.e., 768) for scaling.

```
# Scale the attention scores by the square root of the hidden dimension
scaled_attention_scores = attention_scores / (Q.size(-1) ** 0.5)
```

Self-attention layer—Scaled Attention Scores

	def	_add	(	a	,	_b	):	\n	_	_	_	_return	_a	_+	_b
def	44.1	14.8	14.2	15.6	17.9	19.2	15.3	17.1	21.3	21.2	21.3	16.1	19.1	19.8	19.3
_add	12.2	20.1	17.0	15.3	13.6	15.2	13.5	11.8	10.6	10.3	10.1	12.7	10.0	10.7	12.3
(	9.5	13.8	14.5	13.5	12.3	13.5	12.3	11.1	10.3	9.9	9.8	10.9	9.2	10.2	11.2
a	10.5	12.8	12.6	17.2	12.3	12.9	10.9	12.4	10.2	9.9	9.7	10.4	10.7	10.1	10.4
,	8.9	11.8	12.4	11.9	13.5	12.8	9.1	11.0	9.4	9.0	8.8	8.1	9.6	8.7	10.1
_b	12.2	12.5	12.7	13.6	13.9	16.0	11.7	12.4	11.7	11.3	11.1	10.4	11.3	10.0	13.3
):	16.3	12.4	11.0	11.9	10.7	12.1	16.2	11.1	10.8	10.5	10.4	8.7	9.6	10.8	10.1
\n	11.1	11.2	11.3	11.4	11.8	11.9	9.5	12.7	9.5	9.3	9.1	8.3	9.1	8.6	9.8
_	15.1	9.9	10.3	10.9	11.2	12.9	10.7	9.8	12.6	12.4	12.2	9.3	9.3	10.4	11.1
_	15.2	9.7	9.9	10.6	10.8	12.6	10.4	9.4	12.4	12.2	12.0	9.1	9.1	10.2	10.9
_return	16.5	10.9	10.0	11.3	10.2	11.5	9.7	11.2	11.4	11.2	11.1	12.4	9.1	9.5	9.8
_a	15.6	10.2	9.1	10.1	10.4	10.8	7.3	9.7	9.2	9.0	8.9	7.5	10.3	8.3	9.7
_+	13.4	8.9	8.8	9.6	9.1	10.2	8.7	9.3	9.8	9.5	9.4	7.0	8.0	11.0	8.5
_b	13.5	10.2	9.9	11.0	11.3	13.7	9.4	10.3	10.2	9.9	9.8	8.7	10.1	8.3	11.8

```
>>> # Calculate the variance of the scaled attention scores
>>> variance = torch.var(scaled_attention_scores)
>>> print("\nVariance of Scaled Attention Scores:", variance.item())
>>> Variance of Scaled Attention Scores: 11.636682052612
```

Self-attention layer—Softmax and Masking

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

Self-attention layer—Softmax and Masking

- Before softmax, masking operation is used to cast the **upper triangle** as **-∞**
- This ensure the LLM can **only see tokens from left to right**.
- Because in the text generation task, only the **preceding context** is available.

Self-attention layer—Softmax and Masking

```python
# Create a mask to mask the upper triangle of the attention scores matrix
mask = torch.triu(torch.ones_like(attention_scores), diagonal=1).bool()
attention_scores_masked = attention_scores.masked_fill(mask, float('-inf'))
```

Masked Attention Scores Matrix (K x Q):

def	44.1	-inf	-inf	-inf	-inf	-inf	-inf	-inf	-inf	-inf	-inf	-inf	-inf	-inf	-inf
_add	12.2	20.1	-inf	-inf	-inf	-inf	-inf	-inf	-inf	-inf	-inf	-inf	-inf	-inf	-inf
a	10.5	12.8	12.6	17.2	-inf	-inf	-inf	-inf	-inf	-inf	-inf	-inf	-inf	-inf	-inf
,	8.9	11.8	12.4	11.9	[illegible]	-inf	-inf	-inf	-inf	-inf	-inf	-inf	-inf	-inf	-inf
_b	12.2	12.5	12.7	13.6	13.9	[illegible]	-inf	-inf	-inf	-inf	-inf	-inf	-inf	-inf	-inf
):	16.3	12.4	11.0	11.9	10.7	12.1	[illegible]	-inf	-inf	-inf	-inf	-inf	-inf	-inf	-inf
\n	11.1	11.2	11.3	11.4	11.8	11.9	9.5	[illegible]	-inf	-inf	-inf	-inf	-inf	-inf	-inf
_	15.1	9.9	10.3	10.9	11.2	12.9	10.7	9.8	[illegible]	-inf	-inf	-inf	-inf	-inf	-inf
_	15.2	9.7	9.9	10.6	10.8	12.6	10.4	9.4	12.4	[illegible]	-inf	-inf	-inf	-inf	-inf
_	15.4	9.5	9.8	10.4	10.7	12.5	10.3	9.4	12.3	12.1	[illegible]	-inf	-inf	-inf	-inf
_return	16.5	10.9	10.0	11.3	10.2	11.5	9.7	11.2	11.4	11.2	11.1	[illegible]	-inf	-inf	-inf
_a	15.6	10.2	9.1	10.1	10.4	10.8	7.3	9.7	9.2	9.0	8.9	7.5	[illegible]	-inf	-inf
_+	13.4	8.9	8.8	9.6	9.1	10.2	8.7	9.3	9.8	9.5	9.4	7.0	8.0	[illegible]	-inf
_b	13.5	10.2	9.9	11.0	11.3	13.7	9.4	10.3	10.2	9.9	9.8	8.7	10.1	8.3	11.8

Self-attention layer—Softmax and Masking

Softmax equation:

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

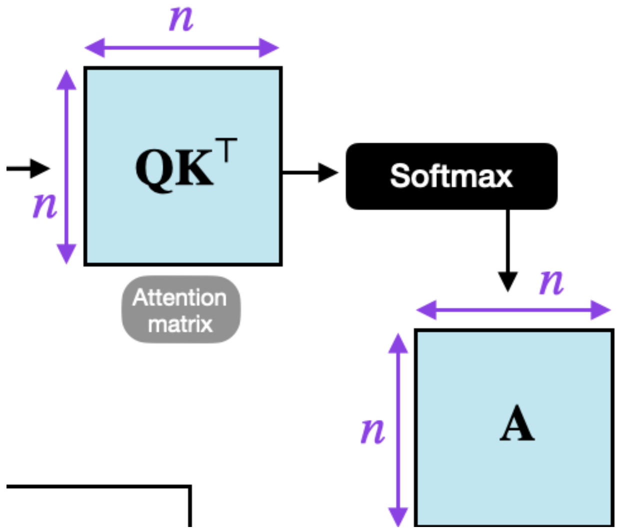
Softmax equation in masking:

$$\sigma(-\infty) = \frac{e^{-\infty}}{\sum_{j=1}^{n} e^{z_j}} = \frac{0}{\sum_{j=1}^{n} e^{z_j}} = 0$$

Self-attention layer—Attention Matrix

- Applying the softmax function to QK^T/sqrt(d) and get the attention matrix A.

```
# Apply the softmax function to get the attention weights matrix A
A = torch.nn.functional.softmax(attention_scores_masked, dim=-1)
```



Self-attention layer—Attention Matrix

Attention Weights Matrix A (after applying softmax):

	def	_add	(	a	,	_b	):	\n	_	_	_	_return	_a	_+	_b
def	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
_add	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
(	0.0	0.3	0.7	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
a	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
,	0.0	0.1	0.2	0.1	0.6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
_b	0.0	0.0	0.0	0.1	0.1	0.8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
):	0.5	0.0	0.0	0.0	0.0	0.0	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
\n	0.1	0.1	0.1	0.1	0.2	0.2	0.0	0.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0
_	0.8	0.0	0.0	0.0	0.0	0.1	0.0	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0
_	0.8	0.0	0.0	0.0	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
_	0.8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
_return	0.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
_a	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
_+	0.8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.0
_b	0.4	0.0	0.0	0.0	0.0	0.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1

Self-attention layer—Attention Matrix

Attention Weights Matrix A (after applying softmax):

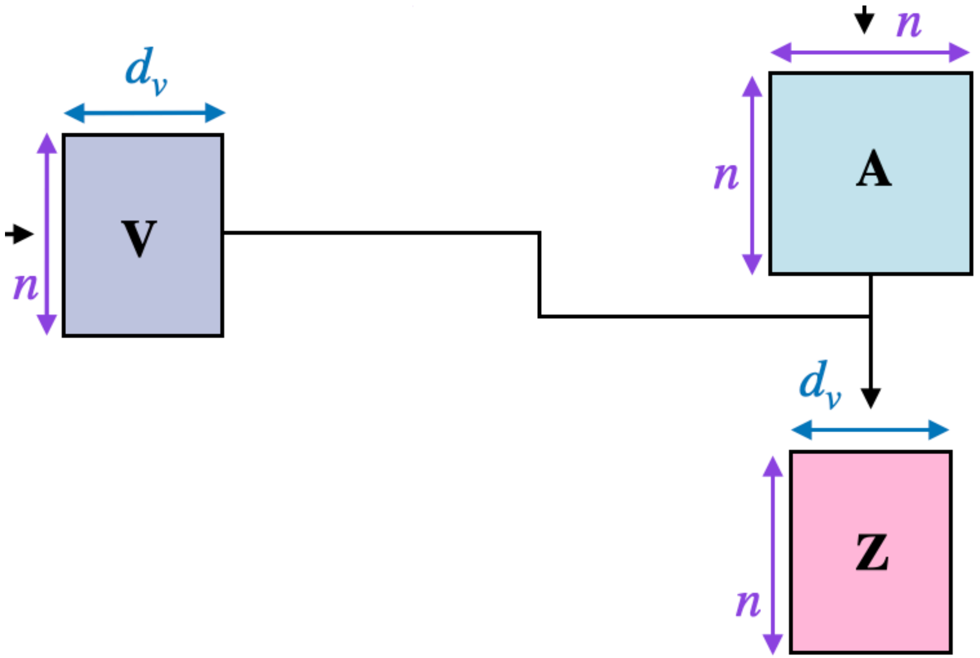
	def	_add	(	a	,	_b	):	\n	_	_	_	_return	_a	_+	_b
def	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
_add	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
(	0.0	0.3	0.7	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
a	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
,	0.0	0.1	0.2	0.1	0.6	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
_b	0.0	0.0	0.0	0.1	0.1	0.8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
):	0.5	0.0	0.0	0.0	0.0	0.0	0.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
\n	0.1	0.1	0.1	0.1	0.2	0.2	0.0	0.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0
_	0.8	0.0	0.0	0.0	0.0	0.1	0.0	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0
_	0.8	0.0	0.0	0.0	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
_	0.8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
_return	0.9	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
_a	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
_+	0.8	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.0
_b	0.4	0.0	0.0	0.0	0.0	0.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1

Self-attention layer—Z=AV

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

Self-attention layer—Z=AV

- Perform matrix multiplication between attention matrix A and the value matrix V and get the output $Z{=}AV$.
- Z has a shape of $n{\times}d$ (number of tokens × hidden dim), which is 15×768.



Self-attention layer—Z=AV

```python
# Compute the matrix multiplication between A and V
Z = torch.matmul(A, V)
```

	Token	Z (15, 768)
0	def	[0.092, -0.522, ..., 0.589]
1	_add	[0.505, 0.107, ..., -0.431]
2	(	[0.142, -0.124, ..., -0.536]
3	a	[0.255, -0.026, ..., -0.512]
4	,	[0.168, -0.029, ..., -0.378]
5	_b	[0.028, -0.048, ..., -0.319]
6	):	[0.156, -0.243, ..., 0.196]
7	\n	[0.148, -0.044, ..., -0.320]
8	_	[0.096, -0.412, ..., 0.416]
9	_	[0.101, -0.421, ..., 0.446]
10	_	[0.103, -0.424, ..., 0.459]
12	_a	[0.095, -0.504, ..., 0.558]
13	_+	[0.103, -0.406, ..., 0.387]
14	_b	[0.056, -0.195, ..., 0.004]

Each line of Z is a new presentation of the token.

Prediction—Output Logits

The size of vector is not the same as vocabulary.

Cannot predict the next token from vocabulary.

```
         Token                              Z (15, 768)
0          def   [0.092, -0.522, ..., 0.589]
1         _add   [0.505, 0.107, ..., -0.431]
2            (  [0.142, -0.124, ..., -0.536]
3            a  [0.255, -0.026, ..., -0.512]
4            ,  [0.168, -0.029, ..., -0.378]
5           _b  [0.028, -0.048, ..., -0.319]
6           ):   [0.156, -0.243, ..., 0.196]
7           \n  [0.148, -0.044, ..., -0.320]
8            _   [0.096, -0.412, ..., 0.416]
9            _   [0.101, -0.421, ..., 0.446]
10           _   [0.103, -0.424, ..., 0.459]
12          _a   [0.095, -0.504, ..., 0.558]
13          _+   [0.103, -0.406, ..., 0.387]
14          _b   [0.056, -0.195, ..., 0.004]
```

Prediction—Output Logits

- Perform another linear transformation on Z to map each vectors from hidden dim (i.e., 768) into the size of vocabulary (i.e., 50257), which is the **output logits**.
- Output logit indicates which integer id in the vocabulary the predicted token belongs to.

Self-attention layer—Output Logits

```
# Declare the linear layer for the output projection to the vocabulary size
linear_output = nn.Linear(hidden_dim, 50257)
output_logits = linear_output(Z)
```

Self-attention layer—Output Logits

- Each element of this 50257-dimensional vector indicates the probability that the predicted next token should be.

```
                Output Logit (15, 50257)
0  [-3.332, 0.474, -5.038, ..., -3.483]
1     [0.346, 0.275, 0.025, ..., 0.331]
2   [0.625, 0.793, -0.324, ..., -0.682]
3     [0.919, 2.356, 1.432, ..., 2.054]
4     [1.450, 1.957, 0.323, ..., 0.154]
5    [1.076, 1.328, 0.089, ..., -0.479]
6  [-1.479, 0.659, -2.643, ..., -1.749]
7    [0.579, 1.465, 0.031, ..., -0.233]
8  [-2.470, 0.648, -3.941, ..., -2.760]
9  [-2.607, 0.620, -4.076, ..., -2.822]
10 [-2.665, 0.608, -4.131, ..., -2.845]
11 [-3.118, 0.513, -4.745, ..., -3.286]
12 [-3.191, 0.507, -4.854, ..., -3.365]
13 [-2.445, 0.601, -3.867, ..., -2.656]
14 [-0.522, 0.980, -1.689, ..., -1.553]
```

Teacher forcing

- Teacher forcing trains the LLM in a self-supervised manner by using the output logits to **predict the next token**.

Teacher forcing

- For example, the first input token is "**def**".

	Input Token	Expected Next Token	Expected Next Token ID	Output Logit (15, 50257)
0	def	_add	751	[-3.332, 0.474, -5.038, ..., -3.483]
1	_add	(	7	[0.346, 0.275, 0.025, ..., 0.331]
2	(	a	64	[0.625, 0.793, -0.324, ..., -0.682]
3	a	,	11	[0.919, 2.356, 1.432, ..., 2.054]
4	,	_b	275	[1.450, 1.957, 0.323, ..., 0.154]
5	_b	):	2599	[1.076, 1.328, 0.089, ..., -0.479]
6	):	\n	198	[-1.479, 0.659, -2.643, ..., -1.749]
7	\n	_	220	[0.579, 1.465, 0.031, ..., -0.233]
8	_	_	220	[-2.470, 0.648, -3.941, ..., -2.760]
9	_	_	220	[-2.607, 0.620, -4.076, ..., -2.822]
10	_	_return	1441	[-2.665, 0.608, -4.131, ..., -2.845]

Teacher forcing

- For example, the first input token is "**def**". LLM outputs the next token probability distribution of "**def**" as output logit.

	Input Token	Expected Next Token	Expected Next Token ID	Output Logit (15, 50257)
0	def	_add	751	[-3.332, 0.474, -5.038, ..., -3.483]
1	_add	(	7	[0.346, 0.275, 0.025, ..., 0.331]
2	(	a	64	[0.625, 0.793, -0.324, ..., -0.682]
3	a	,	11	[0.919, 2.356, 1.432, ..., 2.054]
4	,	_b	275	[1.450, 1.957, 0.323, ..., 0.154]
5	_b	):	2599	[1.076, 1.328, 0.089, ..., -0.479]

Teacher forcing

- For example, the first input token is "**def**". LLM outputs the next token probability distribution of "**def**" as output logit. The highest value in the output logit is the predicted next token. It should be the token after "**def**", which is "_**add**" and its id is 751.

	Input Token	Expected Next Token	Expected Next Token ID	Output Logit (15, 50257)
0	def	_add	751	[-3.332, 0.474, -5.038, ..., -3.483]
1	_add	(	7	[0.346, 0.275, 0.025, ..., 0.331]

Teacher forcing

- Calculate the cross-entropy loss between the output logits and the expected next token ids to update the embedding layers and the self-attention layers.

```
# Calculate the cross-entropy loss
loss =
torch.nn.functional.cross_entropy(output_logits[0:14, :].v
iew(-1, 50257), token_ids[1:15].view(-1))
loss.backward()
```

Teacher forcing

- For the last row, it predicts the token after "**_b**", which does not have any valid next token. So, it does not participate in loss calculation.

No valid next token

def add(a, b):\n return a + b

	Input Token	Expected Next Token	Expected Next Token ID	Output Logit (15, 50257)
14	_b	None	None	[-0.522, 0.980, -1.689, ..., -1.553]

Pre-training

- After pre-training, the LLM have learned lots of knowledge and capable of generating text based on the given context.

Supervised fine-tuning

- However, the answer provided by LLM may not satisfy the need of the user.

Supervised fine-tuning

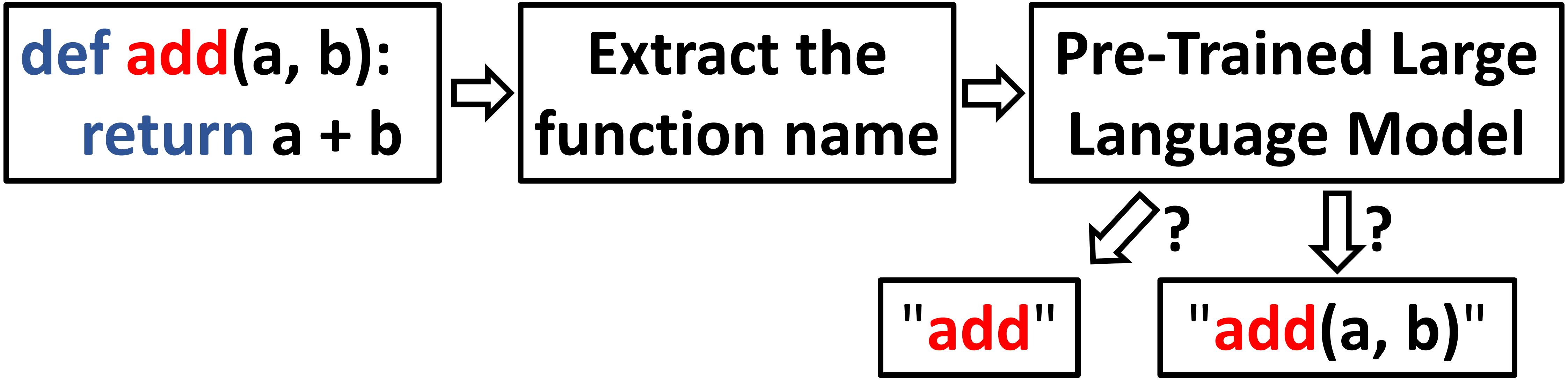
- For example, if we want the LLM to extract the function name from the code fragment.

```
def add(a, b):
    return a + b
```

⇨ **Extract the function name** ⇨ **Pre-Trained Large Language Model**

Supervised fine-tuning

- For example, if we want the LLM to extract the function name from the code fragment.



Supervised fine-tuning

- Supervised fine-tuning address this by fine-tuning the pre-trained LLM on a target dataset with **input and output pair**, following the same training process as teacher forcing.

Supervised fine-tuning

- Usually, some **chat templates** and **special tokens** will be added to instruct the LLM.

Supervised fine-tuning

- For example, Llama use <INST> and <SYST> as special tokens and chat templates.

Supervised fine-tuning

- <<SYS>><</SYS>> provides context or explanation for the expected role of the LLM.
- [INST][/INST] tags are used to encompass entire instructions, such as user questions and inputs.

Supervised fine-tuning

- For example, the input output pair after applying chat template would looks like in Llama:

[INST] <<SYS>> You are a helpful assistant in programming. <</SYS>>

Extract the function name from the code.

```
def add(a, b):
    return a + b [/INST]
add.
```

Supervised fine-tuning

- For example, the input output pair after applying chat template would looks like in Llama:

[INST] <<SYS>> You are a helpful assistant in programming. <</SYS>>

Extract the function name from the code.

Role

def add(a, b):

return a + b [/INST]

add.

Supervised fine-tuning

- For example, the input output pair after applying chat template would looks like in Llama:

[INST] <<SYS>> You are a helpful assistant in programming. <</SYS>>

Extract the function name from the code.

```
def add(a, b):
    return a + b [/INST]
```

Input question

add.

Supervised fine-tuning

- For example, the input output pair after applying chat template would looks like in Llama:

[INST] <<SYS>> You are a helpful assistant in programming. <</SYS>>

Extract the function name from the code.

```
def add(a, b):
    return a + b [/INST]
```

add. ⇨ Expected output

Thank you