Generative Pre-Trained Transformer

- Yoonseo Kim (CS 18)

References

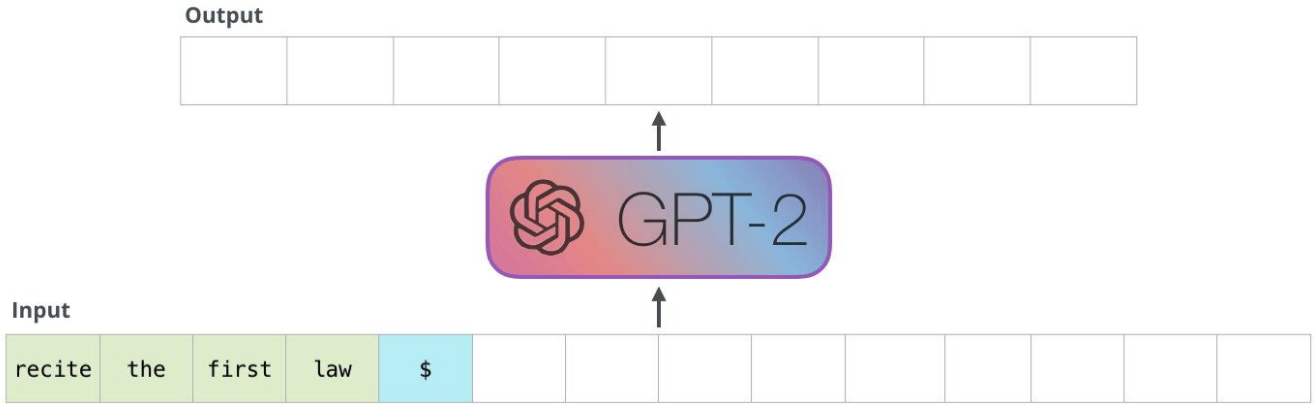
1. http://jalammar.github.io/illustrated-gpt2/
2. https://www.youtube.com/watch?v=MV1I2JPAxVY
3. https://www.groundai.com/project/megatron-lm-training-multi-billion-parameter-language-models-using-gpu-model-parallelism/1#S2.F2
4. https://github.com/affiljoo3581/GPT2
5. https://ainote.tistory.com/17
6. https://github.com/jessevig/bertviz
7. https://arxiv.org/pdf/1710.05941.pdf
8. https://www.youtube.com/watch?v=mMa2PmYJlCo

Prior knowledges

Prior knowledges

1. What is GPT?
2. Few Shot, One Shot, Zero Shot
3. Size
4. Tokenize

1. What is GPT?



Generative ≈ Creating

Language model that generates the appropriate **token** to come next by referring previous **tokens**

1. What is GPT?

Meet me at the park.

meet **me** **at** **the** **park**

Token



GPT

Today weather is hot and ☐ →

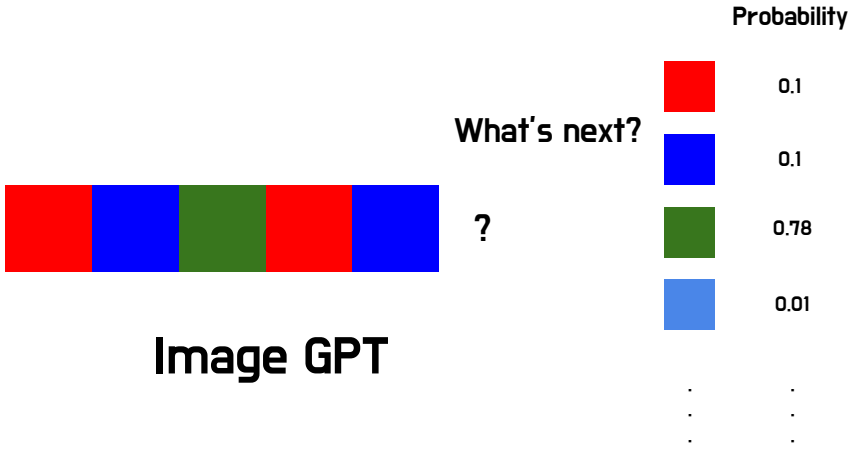
BERT

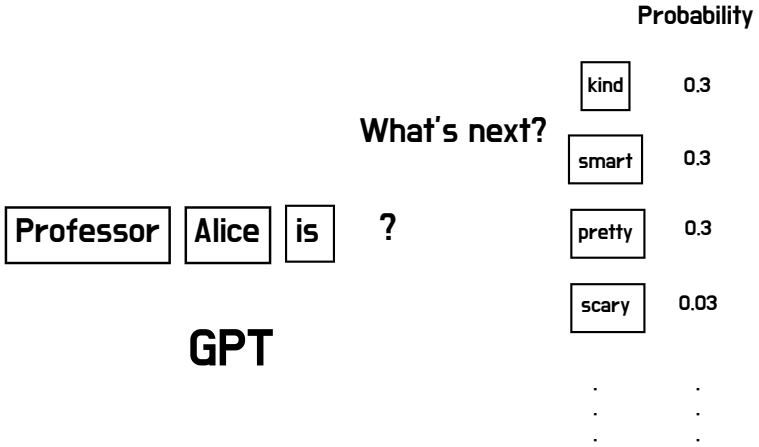
Today ☐ is hot and gloomy.

← →

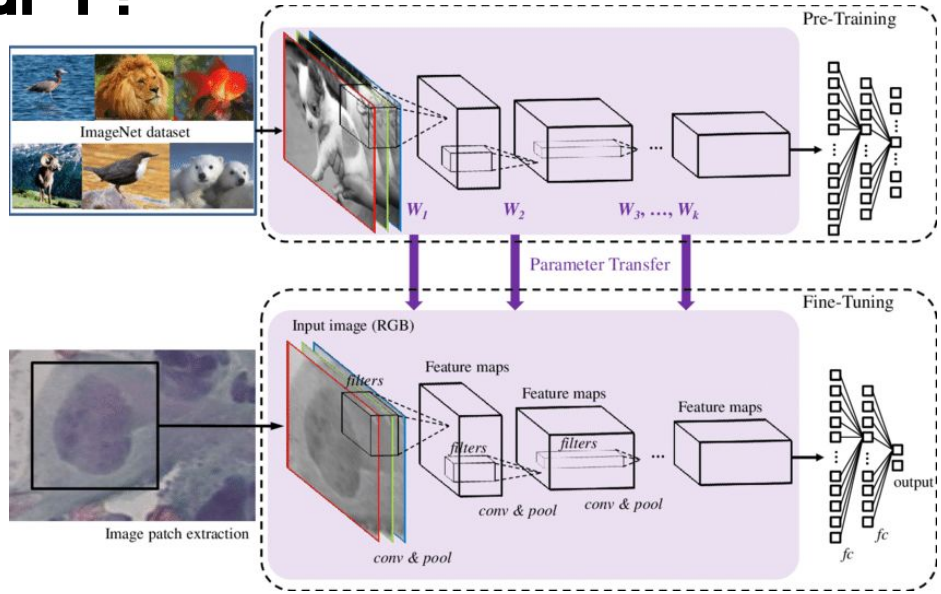
Language Model

1. What is GPT?


Probability
What's next?
?
0.1
0.1
0.78
0.01
Image GPT


Probability
What's next?
Professor
Alice
is
?
kind
0.3
smart
0.3
pretty
0.3
scary
0.03
GPT

1. What is GPT?



Pre-trained = 사전학습된

Trained with the existing corpus without fine-tuning
(Also it shows excellent performances!!)

1. What is GPT?

리뷰	라벨
굳 ㅋ	1
음악이 주가 된, 최고의 음악영화	1
괜찮네요오랜만포켓몬스터잼있어요	1
최고!!!!!!!!!!!!!!!!	1
이게 영화야?	0
돈만있으면 내가 이것보단 더잘만들겠따 ㅋ	0
아이디어가 아주 좋다 재밌다	?

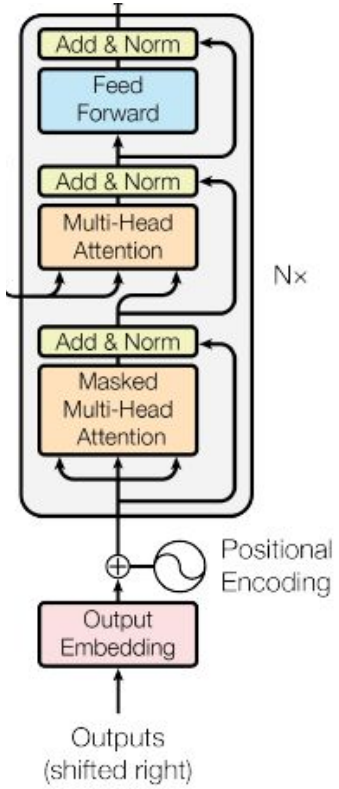
NSMC (Naver Sentiment Movie Corpus)

1. What is GPT?

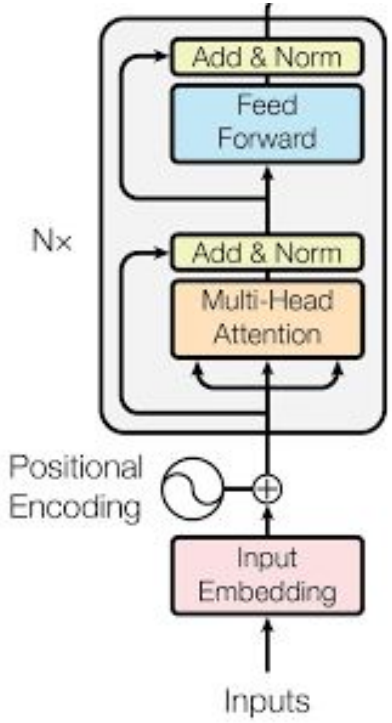
Transformer

GPT's model structure is
similar to Decoder of Transformer

But GPT != Transformer



Transformer Decoder



GPT

2. Few Shot

굳 ㅋ : 1
음악이 주가 된, 최고의 음악영화 : 1
돈만있으면 내가 이것보다 더잘만들겠따 : 0
괜찮네요오랜만포켓몬스터잼있어요 : 1
최고!!!!!!!!!!!!!!!!!!!! : 1
이게 영화야? : 0
아이디어가 아주 좋다 재밌다 :

Few Shot Input

굳 ㅋ : 1
음악이 주가 된, 최고의 음악영화 : 1
돈만있으면 내가 이것보다 더잘만들겠따 : 0
괜찮네요오랜만포켓몬스터잼있어요 : 1
최고!!!!!!!!!!!!!!!!!!!! : 1
이게 영화야? : 0
아이디어가 아주 좋다 재밌다 : 1
아직 너무나도 좋아 : 1
최고영화 나오니 좋아요 : 1
오랜만포켓몬스터 이거는 굉장한 영화임 : 0

Output

2. Zero Shot

Analysis the sentiment of Korean movie review as 0 (negative), 1 (positive)

아이디어가 아주 좋다 재밌다 :

Zero Shot Input

Analysis the sentiment of Korean movie review as 0 (negative), 1 (positive)

아이디어가 아주 좋다 재밌다 : Idea is good very interesting 0.84

아이디어가 아주 똑같다 재밌다 : Idea is very similar interesting 0.83

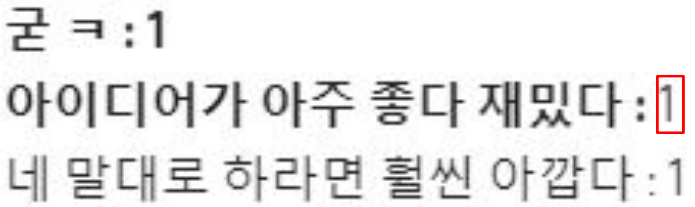
아이디어가 아주 좋지 않다 들었다 : Idea is good not listened to 0.5

Output

2. One Shot

굳 ㅋ : 1
아이디어가 아주 좋다 재밌다 :

One Shot Input



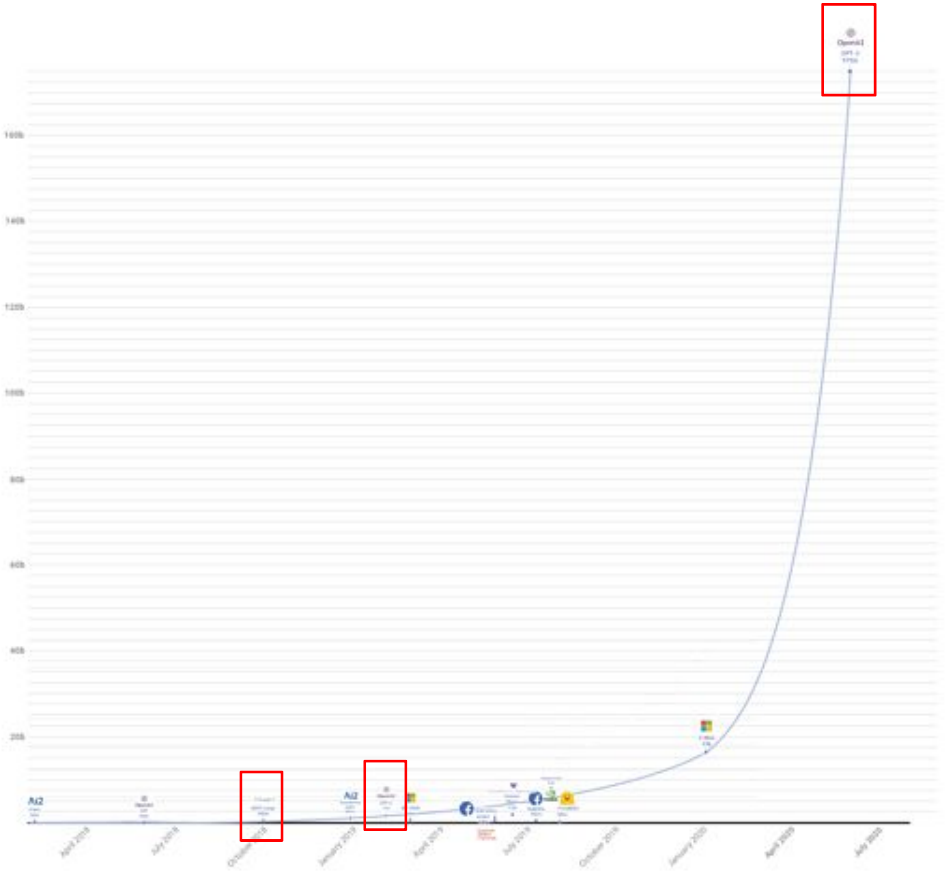
Output

3. Size

Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B or “GPT-3”	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}

GPT-3 Models

3. Size



Size comparison between Language Models

4. Tokenize

내가	0
나를	1
알지	2
못하는	3
것이	4
나의	5
가장	6
큰	7
문제이다	8

Vocab

내가	나를	알지	못하는	것이	나의	가장	큰	문제이다
0	1	2	3	4	5	6	7	8

Tokenize

4. Word Tokenize

```
>>> word_tokenizer = WordTokenizer()
>>> word_tokenizer.tokenize('내가 나를 알지 못하는 것이 나의 가장 큰 문제이다 嫐')
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> word_tokenizer.decode([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
['내가', '나를', '알지', '못하는', '것이', '나의', '가장', '큰', '문제이다', '[UNK]']
```

Word Tokenize Example

4. Word Tokenize

먹다	먹이다	0	2
먹는다	먹고	1	[UNK]
먹는당	먹자	[UNK]	[UNK]

Can't cope with the transformation of words

먹다	먹이다	0	2
먹는다	먹고	1	[UNK]
먹는당	먹자	[UNK]	[UNK]

The association between words disappears

먹다	먹이다	0	2
먹는다	먹고	1	[UNK]
먹는당	먹자	[UNK]	[UNK]

Vulnerable to Out of Vocabulary

4. Subword Tokenize

```
>>> word_tokenizer = WordTokenizer()
>>> word_tokenizer.tokenize('내가 나를 알지 못하는 것이 나의 가장 큰 문제이다 嫐')
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> word_tokenizer.decode([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
['내가', '나를', '알지', '못하는', '것이', '나의', '가장', '큰', '문제이다', '[UNK]']
```

Word Tokenize Example

```
>>> subword_tokenzier = SubWordTokenizer()
>>> subword_tokenzier.tokenize('내가 나를 알지 못하는 것이 나의 가장 큰 문제이다 嫐')
[0, 1, 2, 3, 4, 5, 6, 7, 8, 2, 10, 11, 12, 13, 14, 15]
>>> subword_tokenzier.decode([0, 1, 2, 3, 4, 5, 6, 7, 8, 2, 10, 11, 12, 13, 14, 15])
['_내', '가', '_나', '를', '_알지', '_못', '하는', '_것', '이', '_나', '의', '_가장', '_큰', '_문제', '이다', '[UNK]']
```

Subword Tokenize Example

4. Subword Tokenize

문제이다 → _문제 이다

Subword Tokenize

```
>>> gpt_tokenizer.Load('vocab/spm.model')
True
>>> gpt_tokenizer.tokenize('내가 나를 알지 못하는 것이 나의 가장 큰 문제이다 媓')
[1891, 6200, 12990, 1983, 181, 6197, 168, 206, 724, 62, 6, 1]
>>> gpt_tokenizer.id_to_piece([1891, 6200, 12990, 1983, 181, 6197, 168, 206, 724, 62, 6, 1])
['_내가', '_나를', '_알지', '_못하는', '_것이', '_나의', '_가장', '_큰', '_문제', '이다', '_', '[UNK]']
```

GPT-3 Subword Tokenizer (Sentencepiece)

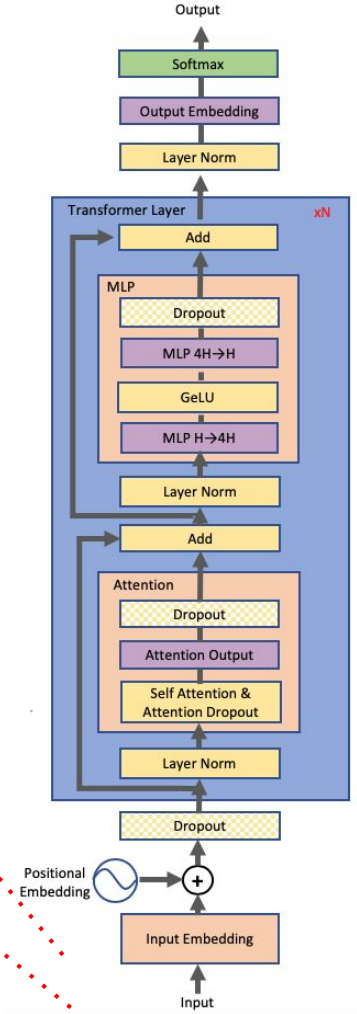
Training

1. Input
2. Learnable Embedding
3. GPT Layers
4. Predicting Next Token

Input



What is an input in GPT and how is it made?



Generation is sequential, but training is not



-> deer



-> truck



-> ship



-> cat

CIFAR 10

I was in my room and suddenly -> someone

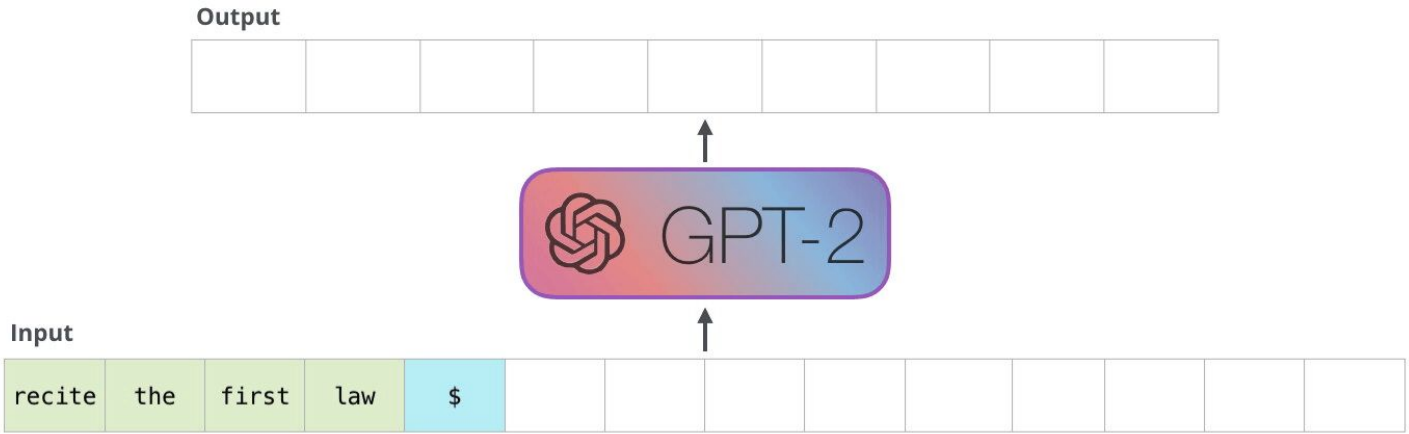
Happy day. sunny day. hot day. cool -> day

I am so sleepy!! I need some -> sleep

Oh. my. god. What -> happened

GPT

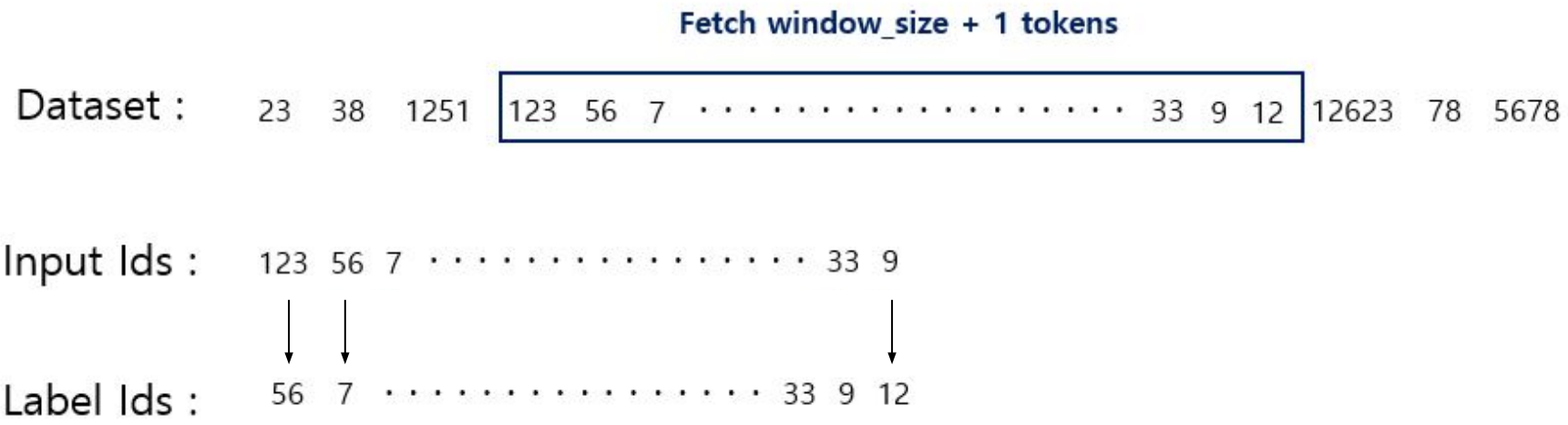
Generation is sequential, but training is not



Autoregressive

BUT training shouldn't work like this.

Input - (1) Input Ids & Label Ids



GPT Input Ids & Label Ids

Input - (1) Input Ids & Label Ids

```python
import h5py
import random
from torch.utils.data import Dataset


class GPTDataset(Dataset):
    def __init__(self, dataset_path: str, seq_len: int, window_size: int, rng: random.Random):
        self.dataset = h5py.File(dataset_path, 'r')['token_ids']
        self.seq_len = seq_len
        self.window_size = window_size
        self.rng = rng
        self.size = len(self.dataset)

    def __len__(self):
        return self.size

    def __getitem__(self, idx):
        start_point = self.rng.randint(0, self.size)
        end_point = min(start_point + (window_size + 1), self.size)
        window = self.dataset[start_point: end_point]  # Windowing

        input_ids = window[:-1]
        label_ids = window[1:]
        return input_ids, label_ids
```

Colored by Color Scripter

GPT Input

(batch_size, window_size)

↓

(batch_size, seq_length)

Input Ids Shape

Input - (1) Input Ids & Label Ids

I was in my room. suddenly someone came into my

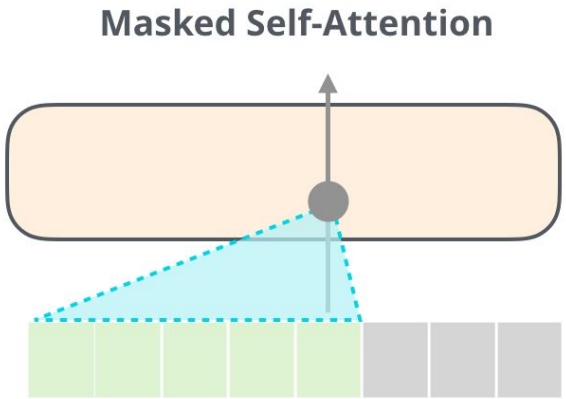
Happy day. sunny day. hot day. cool day is like a

I am so sleepy!! I need some sleep. I should day off

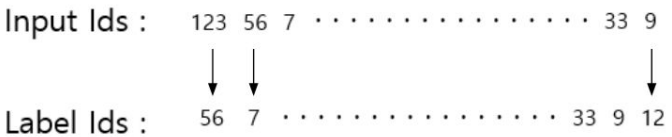
Oh. my. god. What happened to me? Can you believe

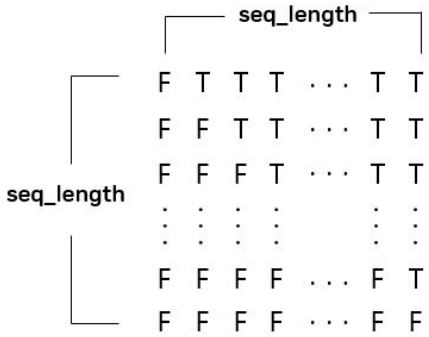
GPT Input

Input - (2) Future Mask



Refer only to previous tokens





Future Mask

Input - (2) Future Mask

```python
def create_future_mask(x: torch.Tensor, offset: int = 0) -> torch.Tensor:

    seq_len = x.size(-1)  # seq_length

    # Create shifted upper triangular matrix.
    future = torch.ones((seq_len, seq_len), dtype=torch.bool, device=x.device)
    future = future.triu(1)

    future_mask = future.view((1,) * (x.ndim - 1) + future.size())
    return future_mask.expand(x.shape + future_mask.shape[-1:])  # (b, s, s)
```

Colored by Color Scripter

GPT Future Mask

6

(seq_len, seq_len)

```
tensor([[True, True, True, True],
        [True, True, True, True],
        [True, True, True, True],
        [True, True, True, True]])
```

7

(seq_len, seq_len)

```
tensor([[False,  True,  True,  True],
        [False, False,  True,  True],
        [False, False, False,  True],
        [False, False, False, False]])
```

9

(1, seq_len, seq_len)

```
tensor([[[False,  True,  True,  True],
         [False, False,  True,  True],
         [False, False, False,  True],
         [False, False, False, False]]])
```

10

(batch_size, seq_len, seq_len)

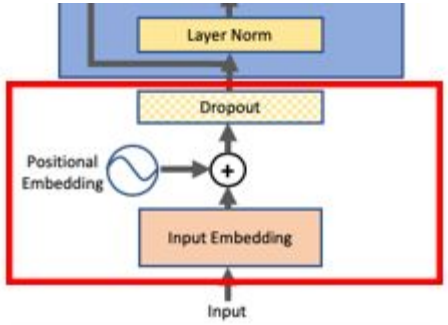
```
tensor([[[False,  True,  True,  True],
         [False, False,  True,  True],
         [False, False, False,  True],
         [False, False, False, False]],

        [[False,  True,  True,  True],
         [False, False,  True,  True],
         [False, False, False,  True],
         [False, False, False, False]]])
```

Input - (2) Future Mask

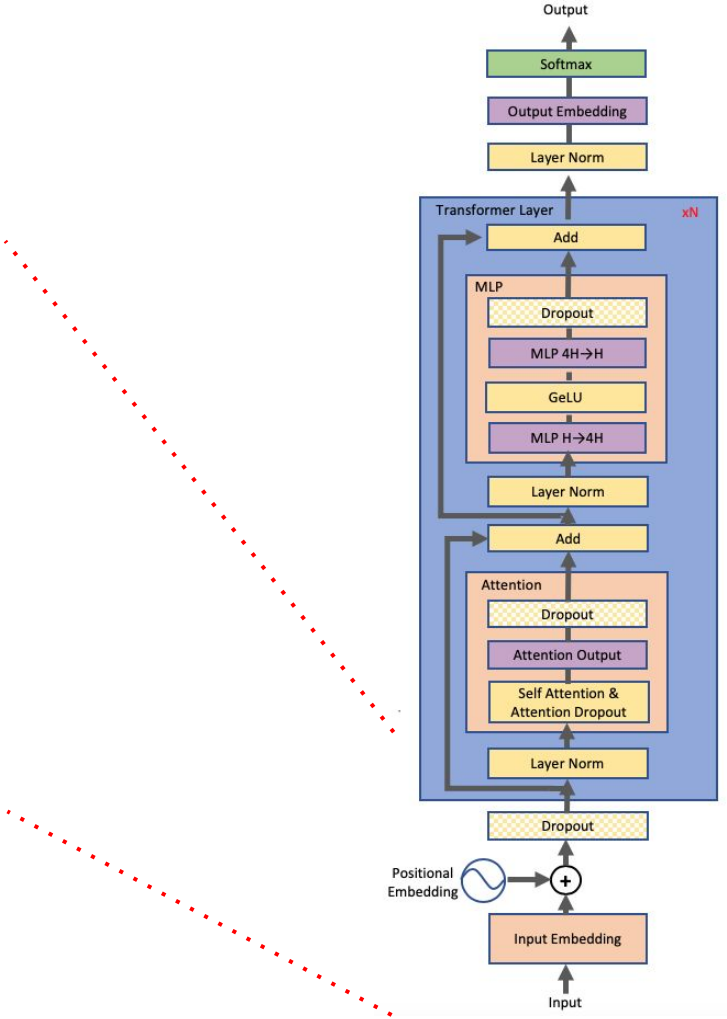
```
>>> future_mask = create_future_mask(input_ids)
>>> future_mask
tensor([[[False,  True,  True,  True],
         [False, False,  True,  True],
         [False, False, False,  True],
         [False, False, False, False]],

        [[False,  True,  True,  True],
         [False, False,  True,  True],
         [False, False, False,  True],
         [False, False, False, False]]])
```

(batch_size, seq_length, seq_length)

Learnable Embedding



How Embedding Works in GPT



Learnable Embedding - (1) Token Embedding

Token	Id	One hot vector
내가	0	[1, 0, 0, 0, 0, 0, 0, 0, 0]
나를	1	[0, 1, 0, 0, 0, 0, 0, 0, 0]
알지	2	[0, 0, 1, 0, 0, 0, 0, 0, 0]
못하는	3	[0, 0, 0, 1, 0, 0, 0, 0, 0]
것이	4	[0, 0, 0, 0, 1, 0, 0, 0, 0]
나의	5	[0, 0, 0, 0, 0, 1, 0, 0, 0]
가장	6	[0, 0, 0, 0, 0, 0, 1, 0, 0]
큰	7	[0, 0, 0, 0, 0, 0, 0, 1, 0]
문제이다	8	[0, 0, 0, 0, 0, 0, 0, 0, 1]

Result of Tokenize & Encoding

Learnable Embedding - (1) Token Embedding

Token	Id	Embedded Vector
내가	0	[0.3, 1.3, 0.2]
나를	1	[0.2, 1.0, 0.3]
알지	2	[-1.1, 4.3, 2.1]
못하는	3	[-0.3, 3.4, 1.4]
것이	4	[1.1, 1.1, 1.3]
나의	5	[0.5, 1.3, 0.4]
가장	6	[0.1, -0.2, 5.3]
큰	7	[-1.7, 2.3, 1.3]
문제이다	8	[3.2, 1.1, 3.1]

Embedding a 9D One-hot vector into 3D

Learnable Embedding - (1) Token Embedding

```python
import torch.nn as nn

token_embeddings = nn.Embedding(vocab_size, hidden_size)
```

Colored by Color Scripter

GPT Token Embedding

0	[0.3, 1.3, 0.2]
1	[0.2, 1.0, 0.3]
2	[-1.1, 4.3, 2.1]
3	[-0.3, 3.4, 1.4]
4	[1.1, 1.1, 1.3]
5	[0.5, 1.3, 0.4]
6	[0.1, -0.2, 5.3]
7	[-1.7, 2.3, 1.3]
8	[3.2, 1.1, 3.1]

torch.nn.Embedding

Learnable Embedding - (1) Token Embedding

0	[0.3, 1.3, 0.2]
1	[0.2, 1.0, 0.3]
2	[-1.1, 4.3, 2.1]
3	[-0.3, 3.4, 1.4]
4	[1.1, 1.1, 1.3]
5	[0.5, 1.3, 0.4]
6	[0.1, -0.2, 5.3]
7	[-1.7, 2.3, 1.3]
8	[3.2, 1.1, 3.1]

torch.nn.Embedding

[1, 0, 0, 0, 0, 0, 0, 0, 0] * [[0.3, 1.3, 0.2], [0.2, 1.0, 0.3], [-1.1, 4.3, 2.1], [-0.3, 3.4, 1.4], [1.1, 1.1, 1.3], [0.5, 1.3, 0.4], [0.1, -0.2, 5.3], [-1.7, 2.3, 1.3], [3.2, 1.1, 3.1]] = [0.3, 1.3, 0.2]

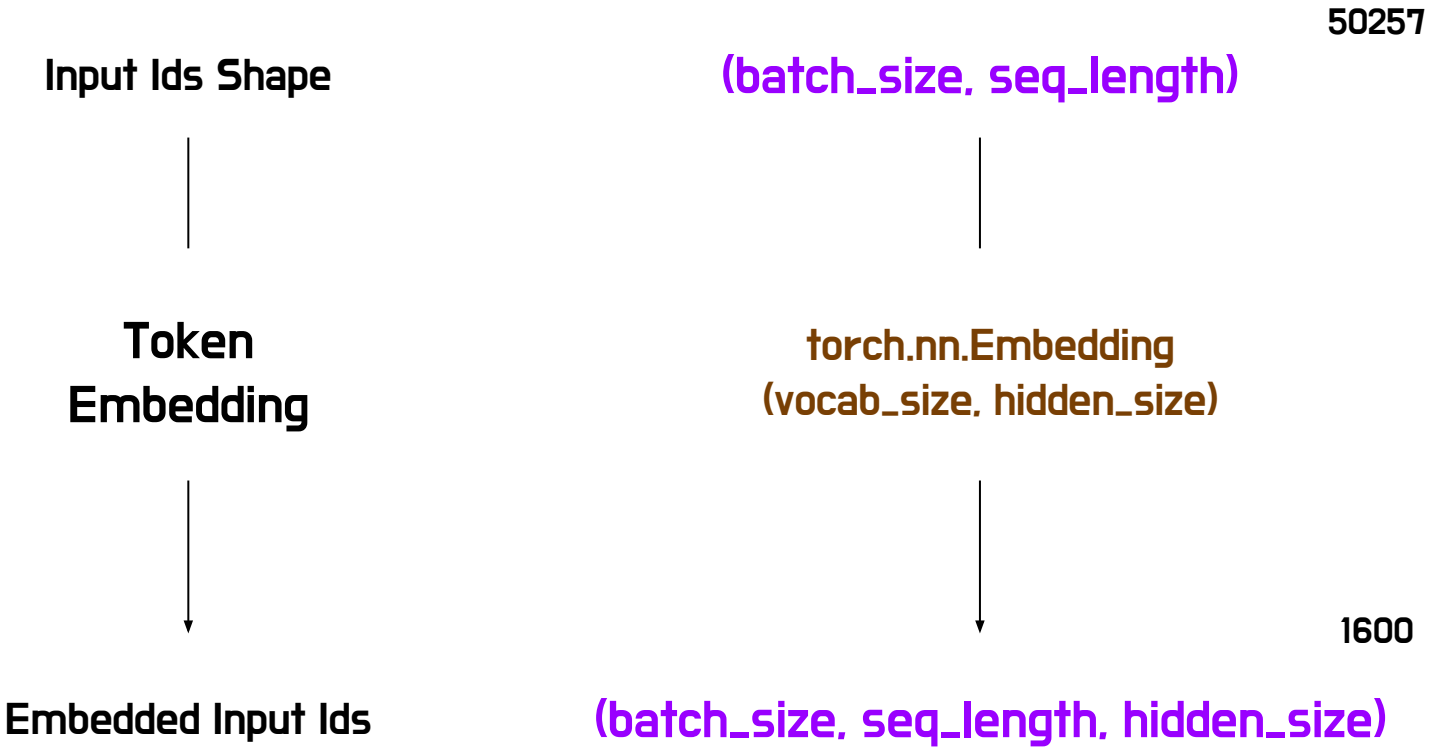
How One-hot vector is embedded

Learnable Embedding - (1) Token Embedding

```
>>> embedding_layer = torch.nn.Embedding(9, 3)
>>> embedding_layer.weight
Parameter containing:
tensor([[ 6.4831e-01,  2.5458e-01, -5.0412e-01],
        [-4.0099e-02,  1.5846e+00, -3.6974e-01],
        [-1.3298e-05,  1.8364e+00, -7.8789e-01],
        [ 1.6661e-01, -1.6003e+00,  3.5276e-01],
        [ 4.8093e-01,  5.0004e-01,  1.6295e-01],
        [ 1.1478e+00, -1.8177e+00, -7.0217e-01],
        [-5.0682e-02, -5.0840e-02, -2.4476e+00],
        [-3.6295e-01,  7.8827e-01, -1.3921e+00],
        [-6.2889e-01,  1.8388e-01, -1.6596e-01]], requires_grad=True)
>>> embedding_layer(torch.tensor(0))
tensor([ 0.6483,  0.2546, -0.5041], grad_fn=<EmbeddingBackward>)
>>> torch.matmul(torch.tensor([1, 0, 0, 0, 0, 0, 0, 0, 0]).float(), embedding_layer.weight)
tensor([ 0.6483,  0.2546, -0.5041], grad_fn=<SqueezeBackward3>)
```

torch.nn.Embedding 예시

Learnable Embedding - (1) Token Embedding



Learnable Embedding - (2) Position Embedding

```python
import torch.nn as nn

position_embeddings = nn.Embedding(seq_len, hidden_size)

def position_embed(input_ids, position_embeddings):

    seq_length = input_ids.size(1)
    position_ids = torch.arange(seq_length, dtype=torch.long, device=input_ids.device)
    position_ids = position_ids.unsqueeze(0).expand_as(input_ids)

    position_embedding_vector = position_embeddings(position_ids)

    return position_embedding_vector
```

Colored

GPT Position Embedding

7

```
tensor([0, 1, 2, 3])
```

(seq_length)

8

```
tensor([[0, 1, 2, 3],
        [0, 1, 2, 3]])
```

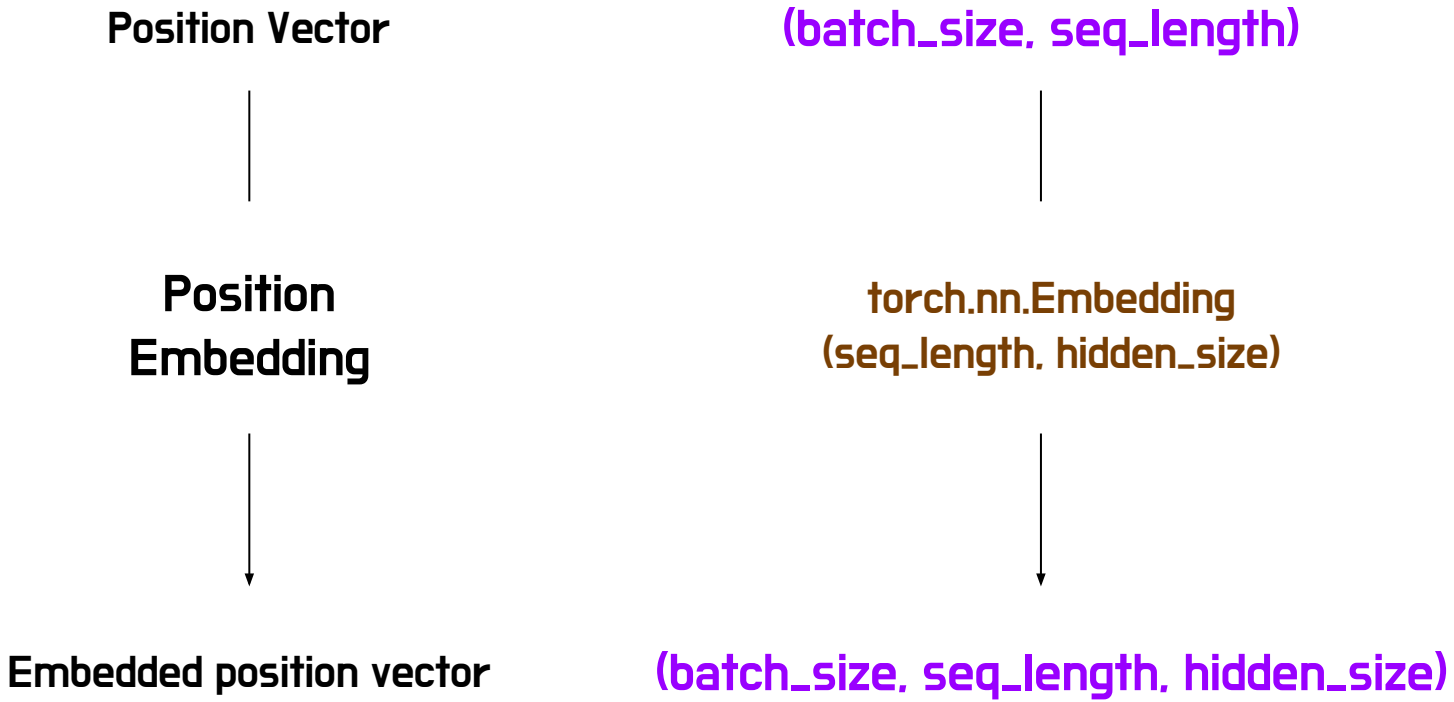
(batch_size, seq_length)

9

```
tensor([[[-0.4798, -0.6253, -2.6828, -0.4392, -0.6134, -0.1879],
         [ 1.2209,  0.2281,  0.0834,  0.1165, -0.5430,  0.8894],
         [ 1.7513,  0.7194, -1.0568, -2.6826, -1.5093, -1.7393],
         [ 1.8245, -0.5266,  0.4133, -0.9550, -0.8500, -0.5146]],

        [[-0.4798, -0.6253, -2.6828, -0.4392, -0.6134, -0.1879],
         [ 1.2209,  0.2281,  0.0834,  0.1165, -0.5430,  0.8894],
         [ 1.7513,  0.7194, -1.0568, -2.6826, -1.5093, -1.7393],
         [ 1.8245, -0.5266,  0.4133, -0.9550, -0.8500, -0.5146]]],
       grad_fn=<EmbeddingBackward>)
```

(batch_size, seq_length, hidden_size)

Learnable Embedding - (2) Position Embedding



Learnable Embedding - (3) Dropout

```python
import torch.nn as nn

drop = nn.Dropout(0.1)
final_embed_vector = drop(token_embed_vector + position_embed_vector)
```

GPT Dropout

```
In[65]: dp = torch.nn.Dropout(0.5)
In[66]: ones = torch.ones(5)
In[67]: ones
Out[67]: tensor([1., 1., 1., 1., 1.])
In[68]: dp(ones)
Out[68]: tensor([2., 2., 2., 2., 0.])
In[69]: dp(ones)
Out[69]: tensor([2., 0., 0., 0., 2.])
```

Dropout Example

Before GPT Layer

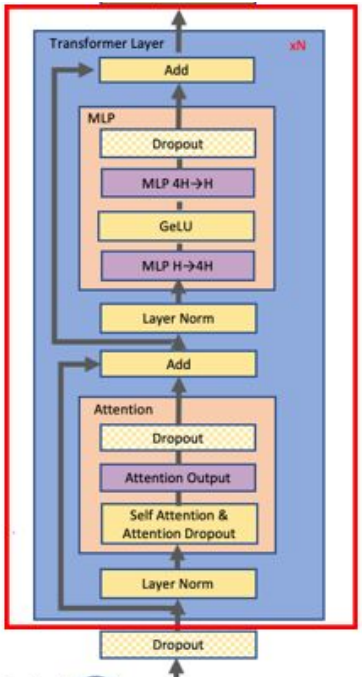
- Final Embedding Vector = Dropout(Token Embedding + Position Embedding)

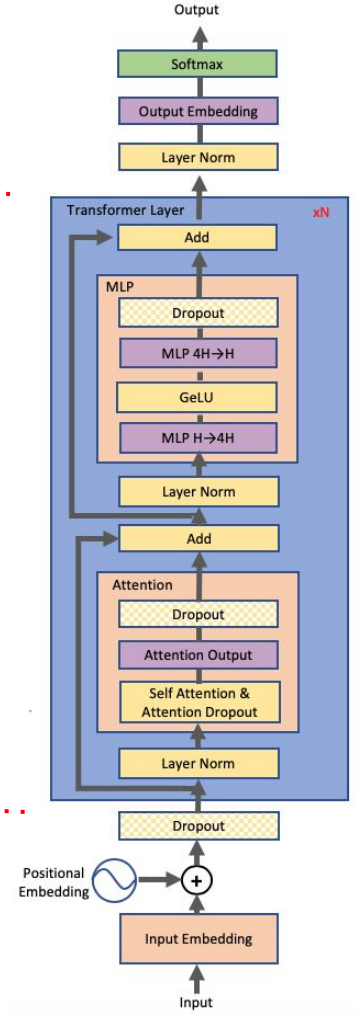
  (batch_size, seq_length, hidden_size)

- Future Mask

  (batch_size, seq_length, seq_length)

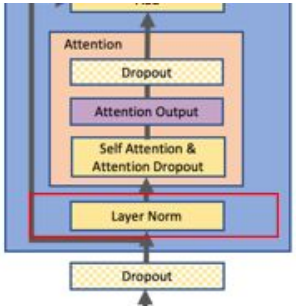
GPT Layer





How a GPT Layer is structured and operated

GPT Layer - (1) Layer Normalization




```python
import torch.nn as nn

layer_norm = nn.LayerNorm(hidden_size, eps=1e-5)
ln_vector = layer_norm(final_embed_vector)

```

GPT Layer Normalization

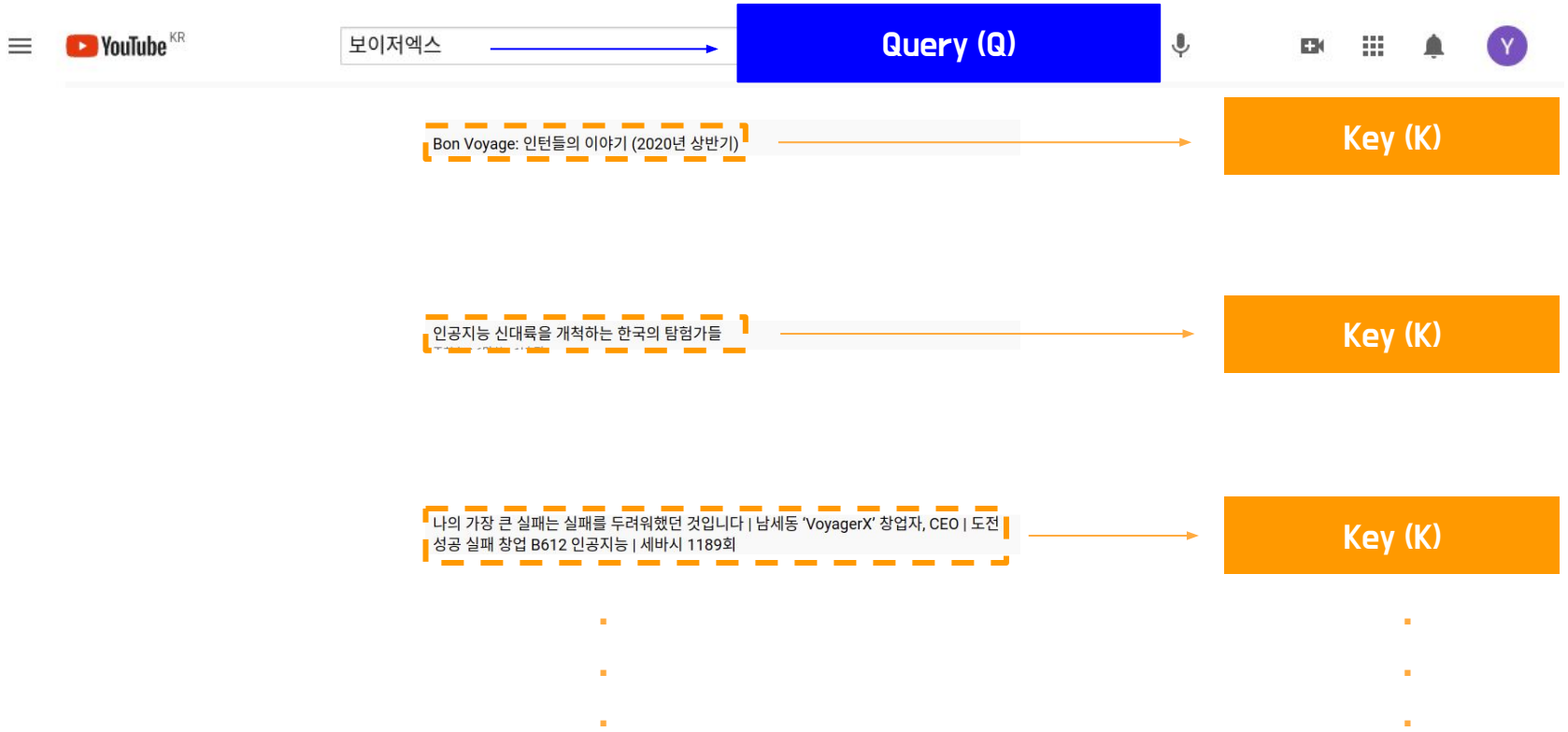
```
>>> before_layer = torch.randint(0, 3, (4, 4)).float()
>>> before_layer
tensor([[1., 2., 1., 0.],
        [1., 1., 0., 0.],
        [0., 1., 0., 1.],
        [1., 0., 2., 1.]])
>>> after_layer = torch.nn.LayerNorm(4)(before_layer)
>>> after_layer
tensor([[ 0.0000,  1.4142,  0.0000, -1.4142],
        [ 1.0000,  1.0000, -1.0000, -1.0000],
        [-1.0000,  1.0000, -1.0000,  1.0000],
        [ 0.0000, -1.4142,  1.4142,  0.0000]],
       grad_fn=<NativeLayerNormBackward>)
```

torch.nn.LayerNorm

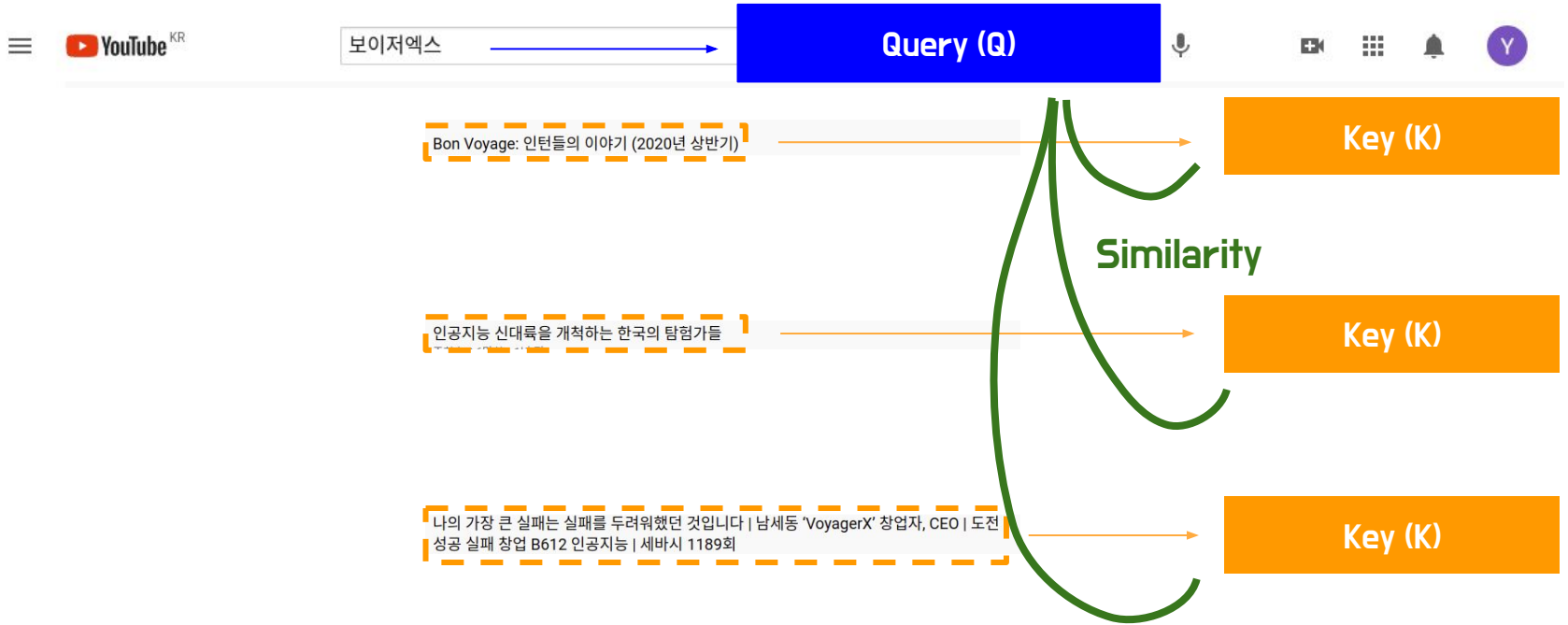
GPT Layer - (2) Masked Multi-Head Self Attention



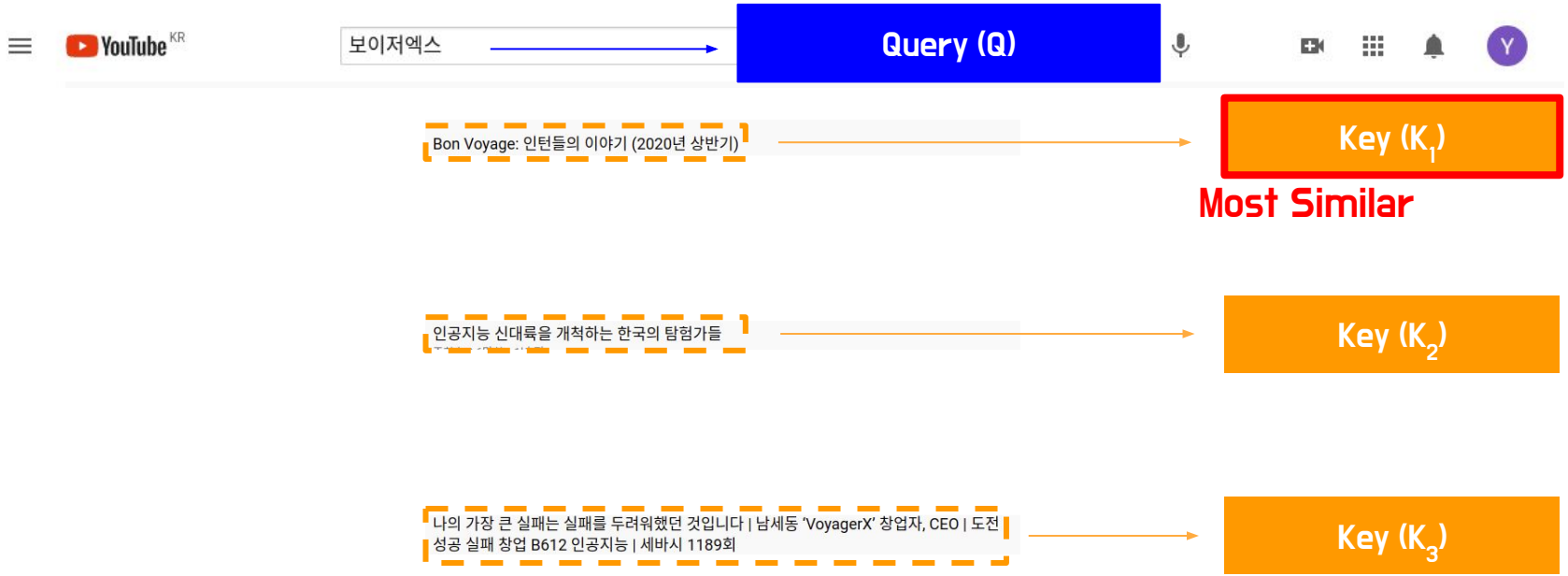
GPT Layer - (2) Masked Multi-Head Self Attention



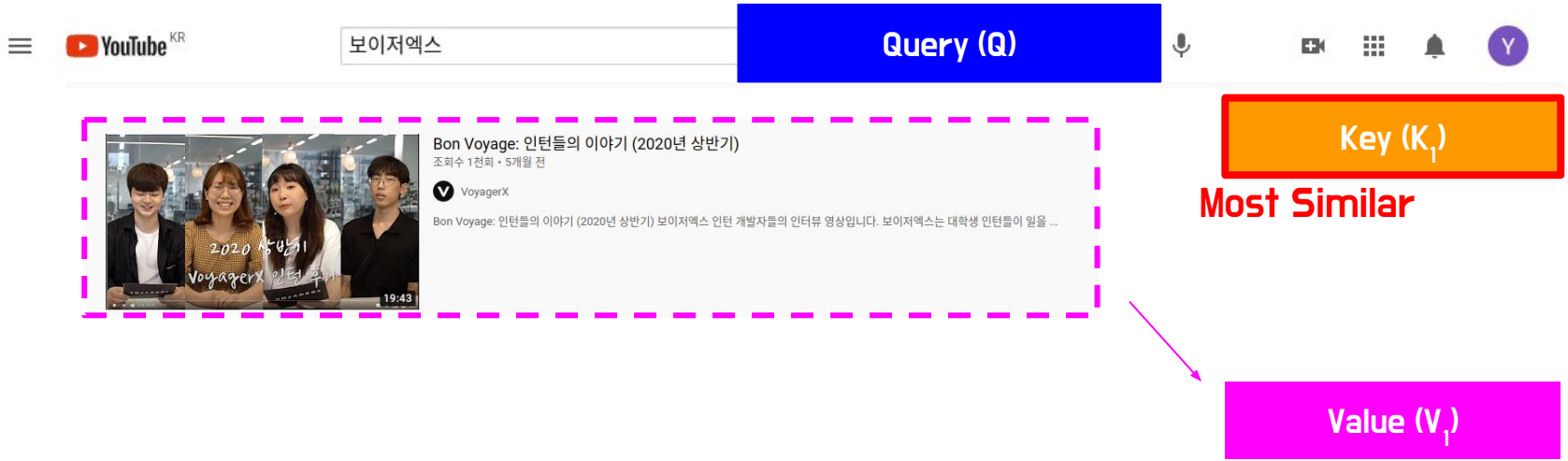
GPT Layer - (2) Masked Multi-Head Self Attention



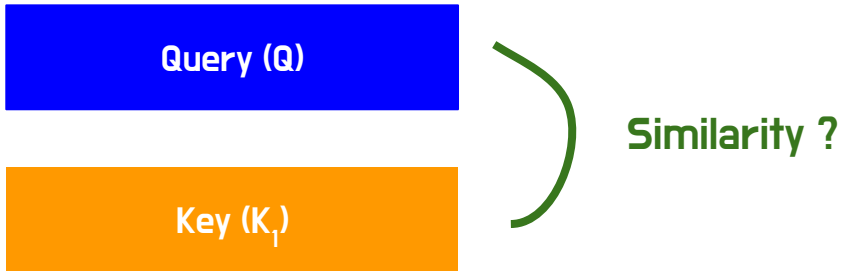
GPT Layer - (2) Masked Multi-Head Self Attention



GPT Layer - (2) Masked Multi-Head Self Attention



GPT Layer - (2) Masked Multi-Head Self Attention



GPT Layer - (2) Masked Multi-Head Self Attention

Cosine SImilarity

$$\text{Cos}\,(A, B) = \frac{A \,.\, B}{|A|\,|B|}$$

GPT Layer - (2) Masked Multi-Head Self Attention

Cosine SImilarity

$$\text{similarity}\,(A, B) = \frac{A \,.\, B}{scaling}$$

GPT Layer - (2) Masked Multi-Head Self Attention

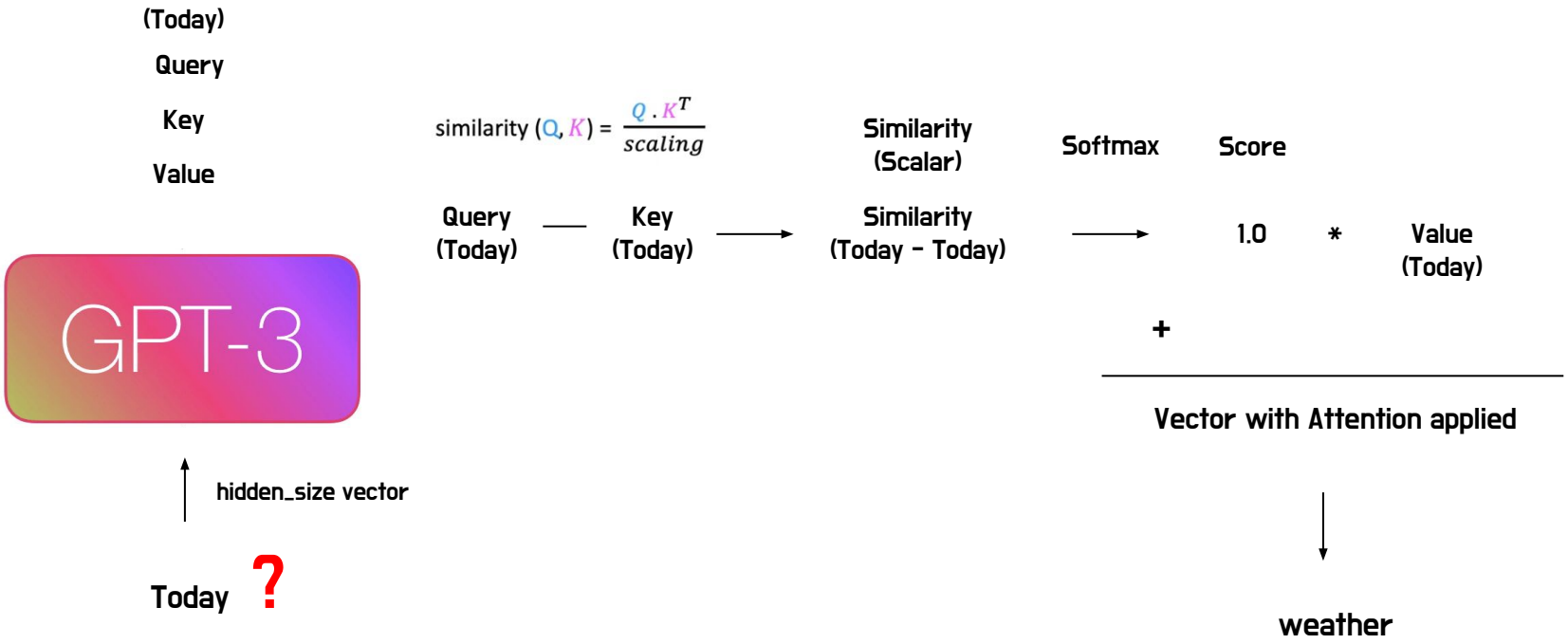
Cosine SImilarity

$$\text{similarity}\,(A, B) = \frac{A \,.\, B^{T}}{scaling}$$

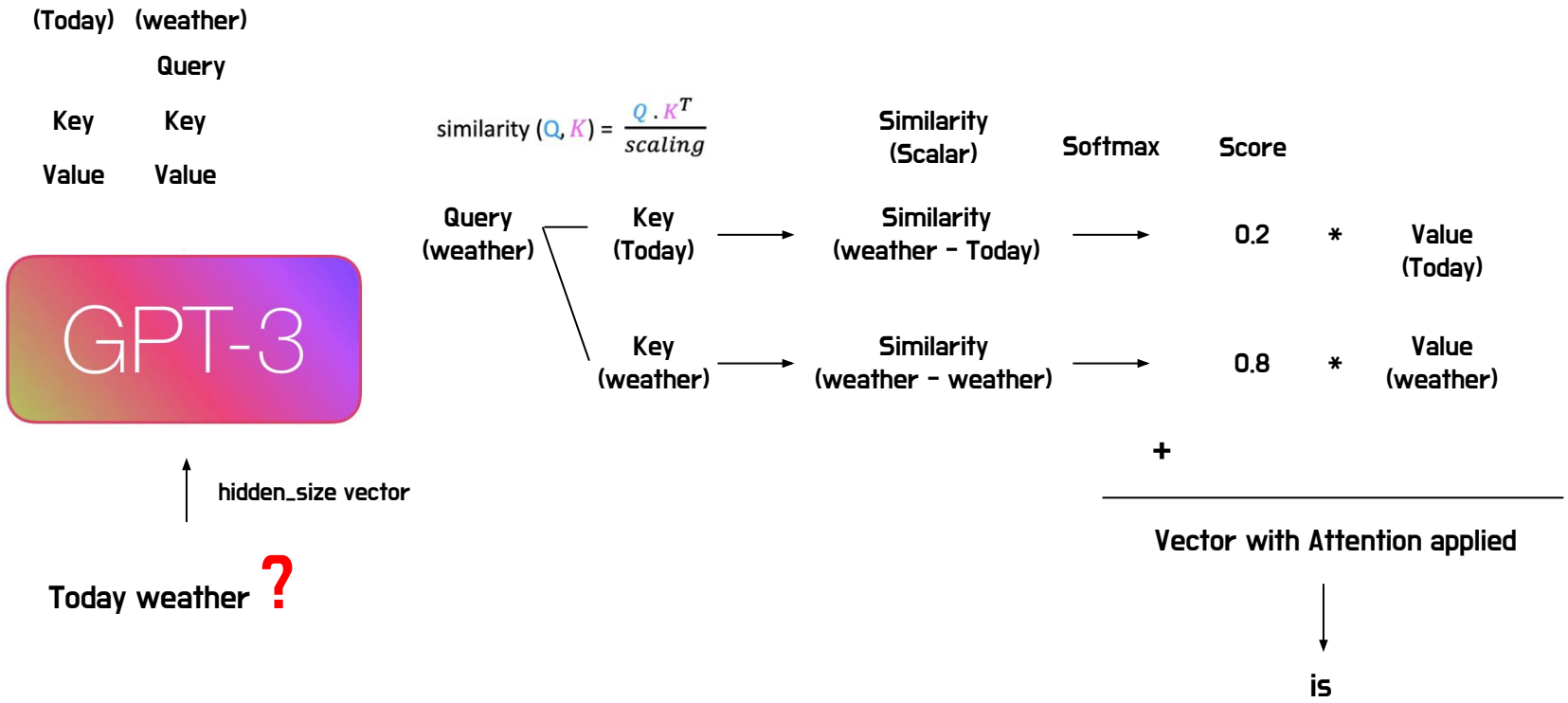


$$\text{similarity}\,(Q, K) = \frac{Q \,.\, K^{T}}{scaling}$$

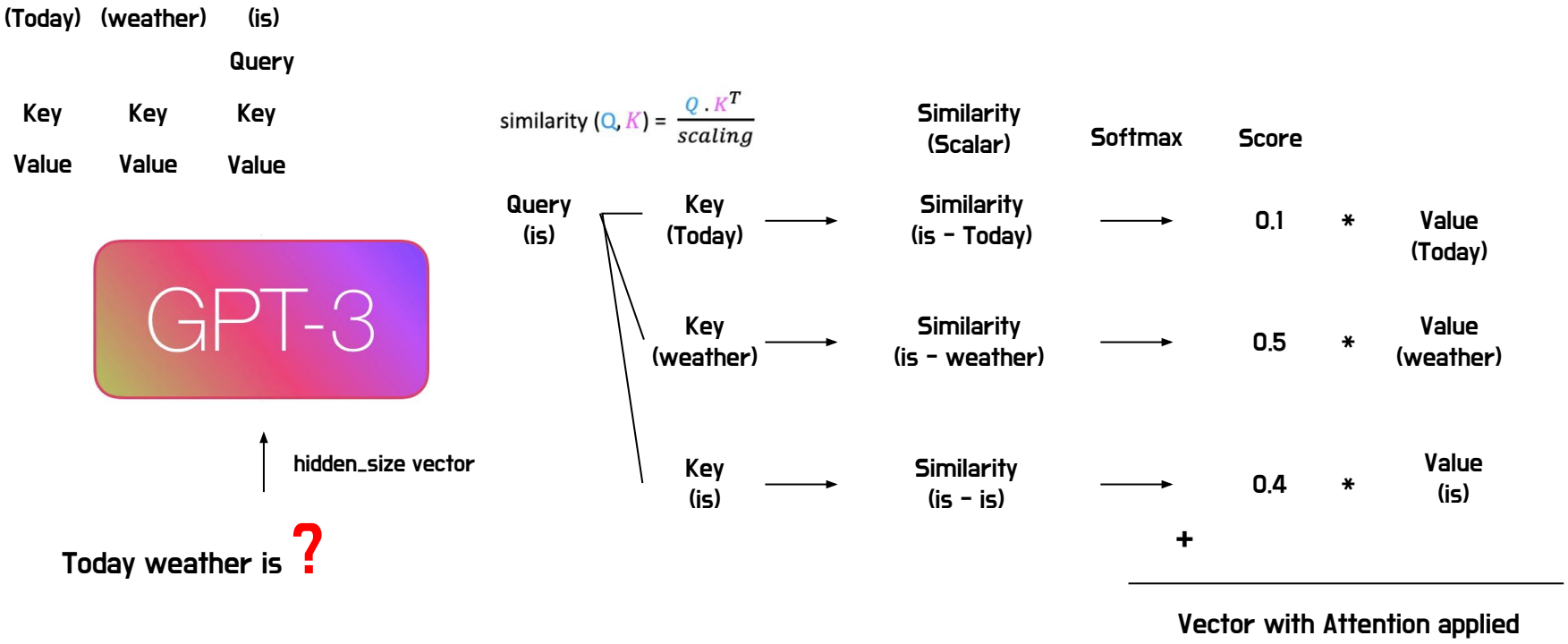
GPT Layer - (2) Masked Multi-Head Self Attention



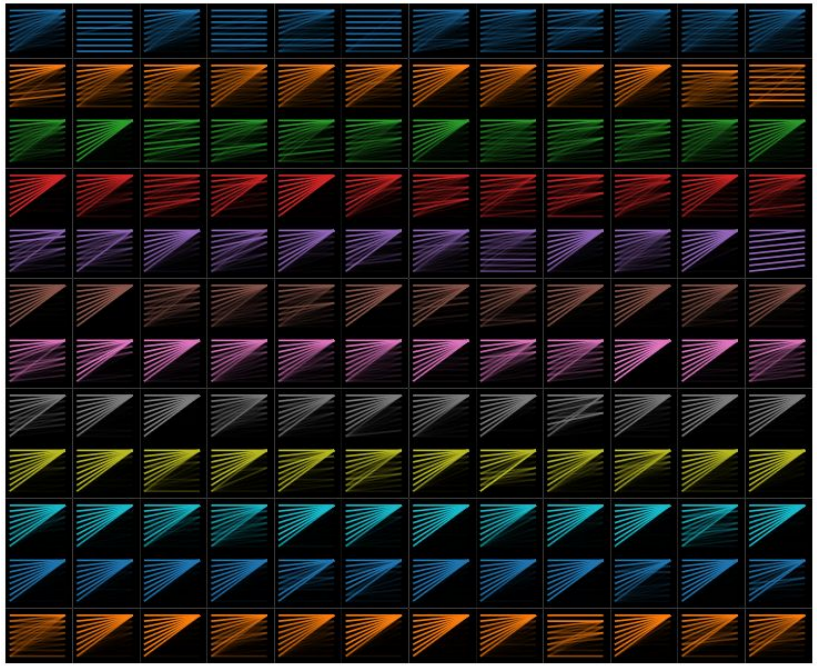
GPT Layer - (2) Masked Multi-Head Self Attention

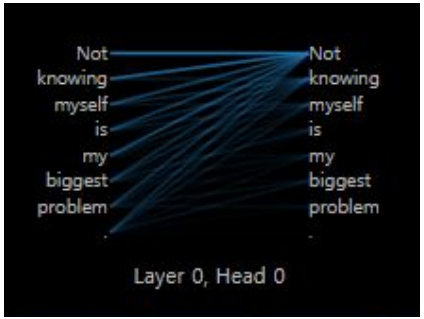


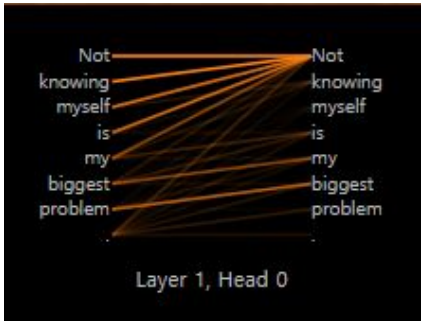
GPT Layer - (2) Masked Multi-Head Self Attention



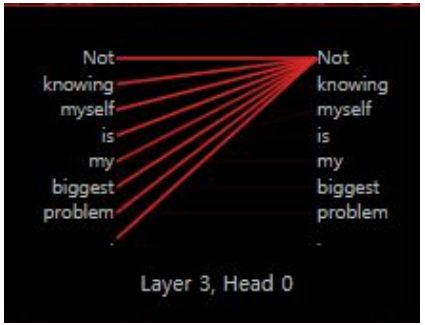
GPT Layer - (2) Masked Multi-Head Self Attention





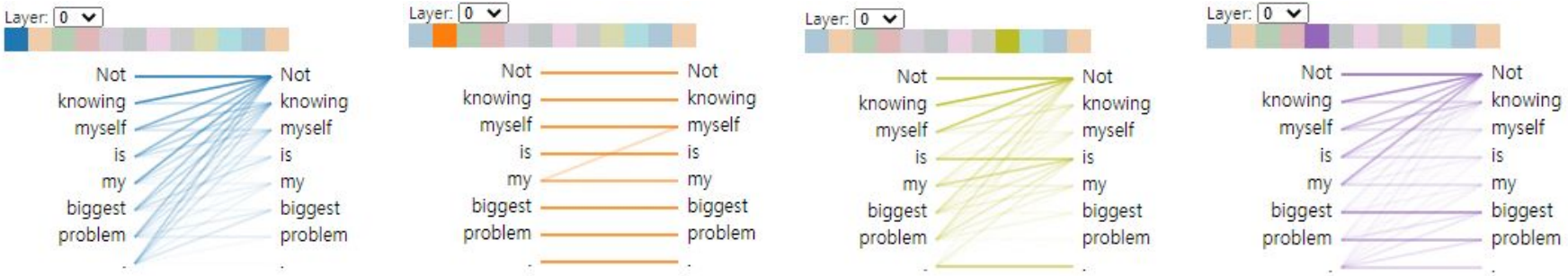






Masked Multi-Head Self Attention (Model View)

GPT Layer - (2) Masked Multi-Head Self Attention

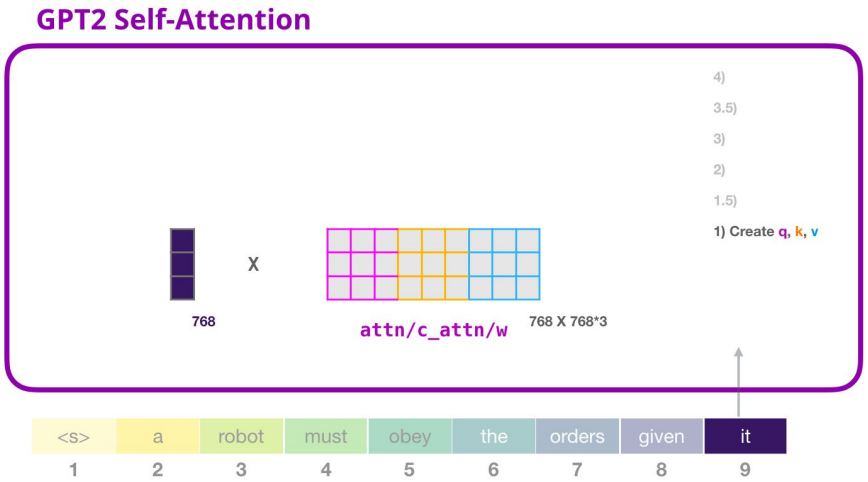


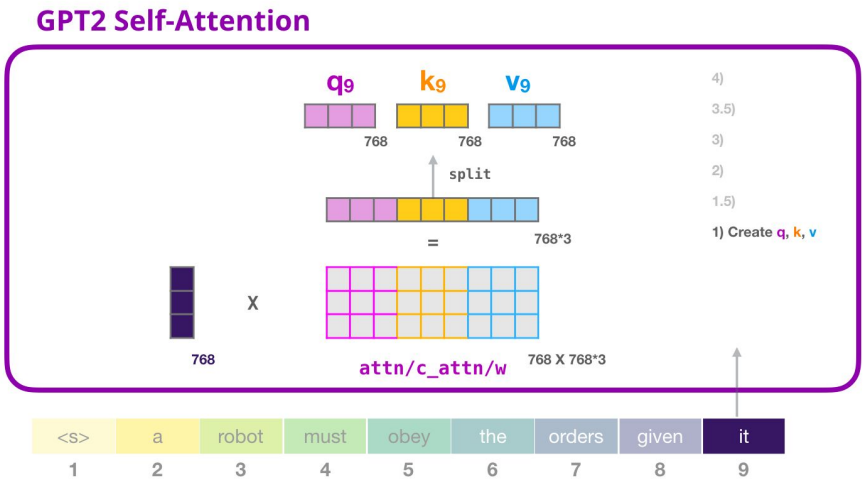
Masked Multi-Head Self Attention (Head View)

GPT Layer - (2) Masked Multi-Head Self Attention

Recap : Final Embedding Vector = Dropout(Token Embedding + Position Embedding)

(batch_size, seq_length, hidden_size) → (batch_size, seq_length, 3 * hidden_size) → Split to Q, K, V (batch_size, seq_length, hidden_size)





GPT Layer - (2) Masked Multi-Head Self Attention

```python
import torch
import torch.nn as nn


class AttentionLayer(nn.Module):

    def __init__(self, heads: int, dims: int, dropout: float = 0.1):
        self.attn = MultiHeadAttention(heads, dropout)
        self.dims = dims
        self.proj = nn.Linear(dims, dims * 3)  # (hideen_size, 3 * hidden_size)
        self.linear = nn.Linear(dims, dims)  # (hidden_size, hidden_size)

    def forward(self, x: torch.Tensor,
                      mask: torch.Tensor,
                ) -> torch.Tensor:

    # Get Query, Key, Value / Shape : (batch_size, seq_length, hidden_size)
    query, key, value  = torch.split(self.proj(x), self.dims, dim=-1)

    score = self.attn(q, k, v, mask)
    score = self.linear(score)

    return score

```

GPT Masked Multi-Head Self Attention

```
In[48]: x
Out[48]:
tensor([[[0.5378, 0.9556, 0.0556],
         [0.5885, 0.9841, 0.8487]]])
In[49]: x.size()
Out[49]: torch.Size([1, 2, 3])
In[50]: qkv = proj(x)
In[51]: qkv
Out[51]:
tensor([[[-0.3486, -0.1192,  0.3584, -0.9130,  0.1688, -0.3246, -0.5575,
          -0.0821, -0.0264],
         [-0.3802, -0.2383,  0.6936, -1.1658, -0.1065, -0.6250, -0.7153,
           0.0435, -0.1149]]], grad_fn=<AddBackward0>)
In[52]: qkv.size()
Out[52]: torch.Size([1, 2, 9])
In[53]: query, key, value = torch.split(qkv, 3, dim=-1)
In[56]: query
Out[56]:
tensor([[[-0.3486, -0.1192,  0.3584],
         [-0.3802, -0.2383,  0.6936]]], grad_fn=<SplitBackward>)
In[57]: key
Out[57]:
tensor([[[-0.9130,  0.1688, -0.3246],
         [-1.1658, -0.1065, -0.6250]]], grad_fn=<SplitBackward>)
In[58]: value
Out[58]:
tensor([[[-0.5575, -0.0821, -0.0264],
         [-0.7153,  0.0435, -0.1149]]], grad_fn=<SplitBackward>)
```

GPT Layer - (2) Masked Multi-Head Self Attention

Embedded Position vector

(batch_size, seq_length, hidden_size)

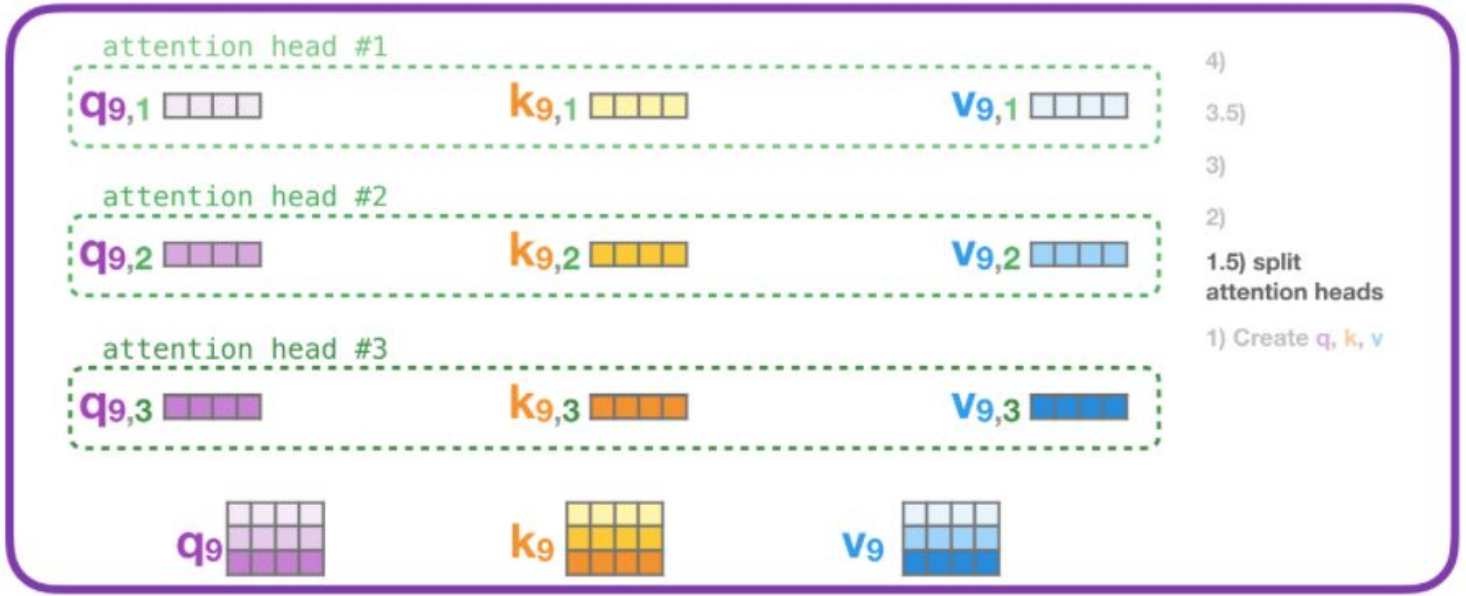
Projection Matrix
Split

torch.nn.Linear (hidden_size, 3 * hidden_size)
torch.split

Query, Key, Value

Q : (batch_size, seq_length, hidden_size)
K : (batch_size, seq_length, hidden_size)
V : (batch_size, seq_length, hidden_size)

GPT Layer - (2) Masked Multi-Head Self Attention



GPT Layer - (2) Masked Multi-Head Self Attention

```python
import math
import torch
import torch.nn as nn
from typing import Optional, Tuple


class MultiHeadAttention(object):

    def __init__(self, heads: int, dropout: float = 0.1):
        super().__init__(dropout)
        self.heads = heads
        self.dropout = nn.Dropout(dropout)

    def forward(self,
                q: torch.Tensor,
                k: torch.Tensor,
                v: torch.Tensor,
                mask: torch.Tensor) -> torch.Tensor:

        # Split the tensors to multi-heads.
        # Shape : (batch_size, seq_length, head_num, one_head_len)
        q = q.view(q.size()[:-1] + (self.heads, q.size(-1) // self.heads))
        k = k.view(k.size()[:-1] + (self.heads, k.size(-1) // self.heads))
        v = v.view(v.size()[:-1] + (self.heads, v.size(-1) // self.heads))
        num_heads_sqrt = math.sqrt(q.size(-2))  # sqrt(head_num)


        q = q.transpose(-3, -2)  # (batch_size, head_num, seq_length, one_head_len)
        k = k.transpose(-2, -1)  # (batch_size, head_num, seq_length, one_head_len)
        v = v.transpose(-3, -2)  # (batch_size, head_num, seq_length, one_head_len)

```

Q : (batch_size, seq_length, hidden_size)
K : (batch_size, seq_length, hidden_size)
V : (batch_size, seq_length, hidden_size)

Q : (batch_size, head_num, seq_length, one_head_size)
K : (batch_size, head_num, one_head_size, seq_length)
V : (batch_size, head_num, seq_length, one_head_size)

GPT Layer - (2) Masked Multi-Head Self Attention

```
# Calculate multi-headed attentions and merge them into one.
k = k.transpose(-2, -1)  # (batch_size, head_num, one_head_len, seq_length)
```

```
# Matrix Multiple Query AND Keys / Divide with sqrt(head_num)
# Shape : (batch_size, head_num, seq_length, seq_length)
score = torch.matmul(q, k) / num_heads_sqrt
```

Q : (batch_size, head_num, seq_length, one_head_size)

K : (batch_size, head_num, one_head_size, seq_length)

↓ Matrix Multiplication

Similarity : (batch_size, head_num, seq_length, seq_length)

batch_size : For each sample
head_num : For each Head
seq_length : For each Token
seq_length : Similarity with all other tokens

	Today	weather	is
Today	0.3	1.1	6.3
weather	2.1	1.2	2.1
is	5.3	1.2	4.2

GPT Layer - (2) Masked Multi-Head Self Attention

batch_size : 1 / head_num : 2 / seq_length : 3

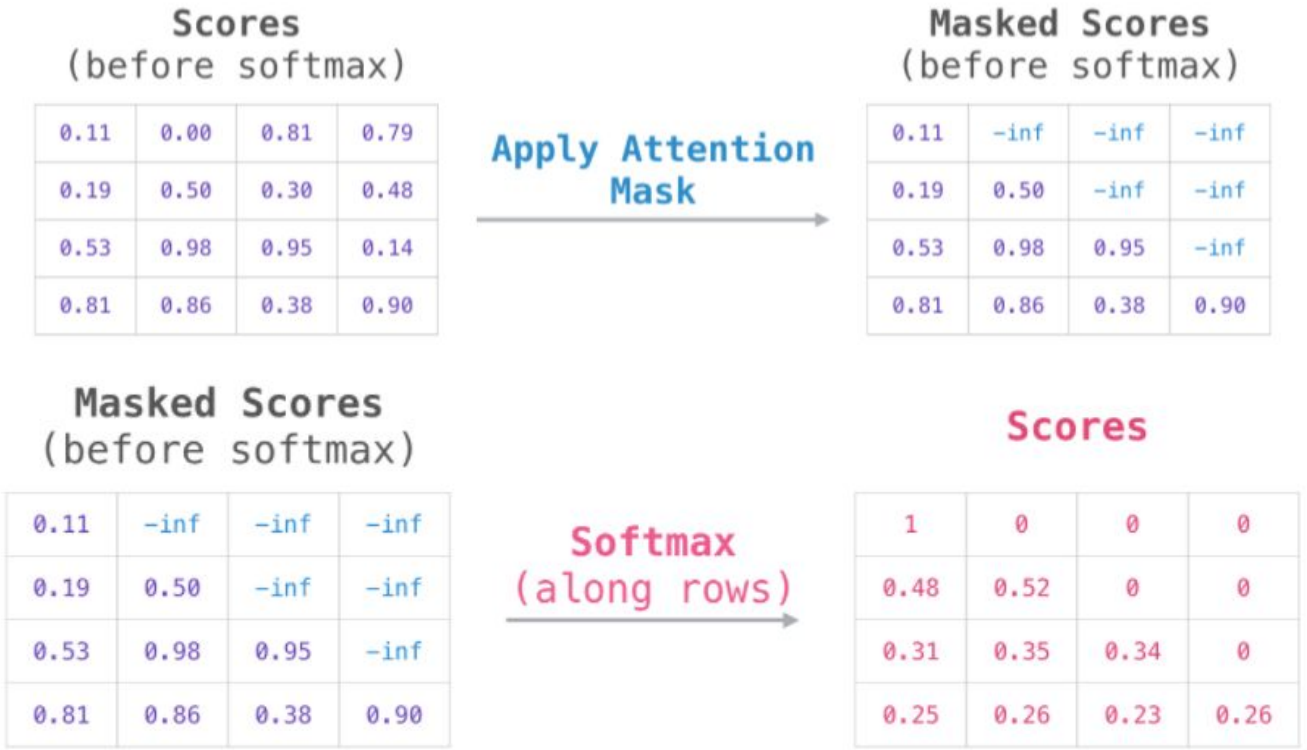
```
>>> score
tensor([[[[0.6110, 0.3211, 0.5879],
          [0.9066, 0.7498, 0.9663],
          [0.0859, 0.9515, 0.0972]],

         [[0.6652, 0.7947, 0.0451],
          [0.2497, 0.3553, 0.5850],
          [0.2024, 0.5119, 0.6343]]]])
>>> score.size()
torch.Size([1, 2, 3, 3])
```

Similarity

```
>>> mask
tensor([[[False,  True,  True],
         [False, False,  True],
         [False, False, False]]])
>>> mask.size()
torch.Size([1, 3, 3])
```

Mask

GPT Layer - (2) Masked Multi-Head Self Attention



GPT Layer - (2) Masked Multi-Head Self Attention

```
# Mask
mask = mask.unsqueeze(-3)  # (batch_size, 1, seq_length, seq_length)
mask = mask.type_as(score)  # (batch_size, head_num, seq_length, seq_length)
```

(1. 3. 3)

```
>>> x = torch.rand((1, 3))
>>> create_future_mask(x)
tensor([[[False,  True,  True],
         [False, False,  True],
         [False, False, False]]])
```

(1. 1. 3. 3)

```
>>> mask = mask.unsqueeze(-3)
>>> mask
tensor([[[[False,  True,  True],
          [False, False,  True],
          [False, False, False]]]])
>>> mask.size()
torch.Size([1, 1, 3, 3])
```

(1. 1. 3. 3)

```
>>> mask = mask.type_as(score)
>>> mask
tensor([[[[0., 1., 1.],
          [0., 0., 1.],
          [0., 0., 0.]]]])
>>> mask.size()
torch.Size([1, 1, 3, 3])
```

(batch_size. seq_length. seq_length)

↓ Add Head Dimension

(batch_size. 1. seq_length. seq_length)

↓ Bool → Float

(batch_size. 1. seq_length. seq_length)

GPT Layer - (2) Masked Multi-Head Self Attention

```python
# Add Large Negative where the mask is true
# Shape : (batch_size, head_num, seq_length, seq_length)
masked_score = score + mask * score.new_tensor(-1e6)

# Softmax
# float that large negative number has added becomes to zero
masked_score  = masked_score.softmax(-1)
masked_score  = self.dropout(masked_score)
```

Shape

(batch_size, head_num, seq_length, seq_length)

Adding large negative number

```
>>> masked_score = score + mask * score.new_tensor(-1e6)
>>> masked_score
tensor([[[[ 6.1096e-01, -1.0000e+06, -1.0000e+06],
          [ 9.0656e-01,  7.4982e-01, -1.0000e+06],
          [ 8.5939e-02,  9.5146e-01,  9.7203e-02]],

         [[ 6.6523e-01, -1.0000e+06, -1.0000e+06],
          [ 2.4973e-01,  3.5529e-01, -1.0000e+06],
          [ 2.0240e-01,  5.1187e-01,  6.3434e-01]]]])
>>> masked_score.size()
torch.Size([1, 2, 3, 3])
```

(1, 2, 3, 3)

Softmax

```
>>> masked_score = masked_score.softmax(-1)
>>> masked_score
tensor([[[[1.0000, 0.0000, 0.0000],
          [0.5391, 0.4609, 0.0000],
          [0.2279, 0.5416, 0.2305]],

         [[1.0000, 0.0000, 0.0000],
          [0.4736, 0.5264, 0.0000],
          [0.2562, 0.3491, 0.3946]]]])
```

(1, 2, 3, 3)

Dropout

```
>>> dp = torch.nn.Dropout(0.1)
>>> dp(masked_score)
tensor([[[[1.1111, 0.0000, 0.0000],
          [0.5990, 0.5121, 0.0000],
          [0.2532, 0.6018, 0.2561]],

         [[1.1111, 0.0000, 0.0000],
          [0.0000, 0.5849, 0.0000],
          [0.2847, 0.3879, 0.4385]]]])
```

(1, 2, 3, 3)

GPT Layer - (2) Masked Multi-Head Self Attention

Word	Value vector	Score	Value X Score
<s>		0.001	
a		0.3	
robot		0.5	
must		0.002	
obey		0.001	
the		0.0003	
orders		0.005	
given		0.002	
it		0.19	
		Sum:	

GPT Layer - (2) Masked Multi-Head Self Attention

score : (batch_size, head_num, seq_length, seq_length)

value : (batch_size, head_num, seq_length, one_head_size)

↓ Matrix Multiplication

attention_score : (batch_size, head_num, seq_length, one_head_size)

```
>>> masked_score
tensor([[[[1.1111, 0.0000, 0.0000],
          [0.3356, 0.7755, 0.0000],
          [0.5575, 0.2729, 0.2808]],

         [[1.1111, 0.0000, 0.0000],
          [0.5957, 0.5154, 0.0000],
          [0.4035, 0.2691, 0.4385]]]])
>>> masked_score.size()
torch.Size([1, 2, 3, 3])
```

(1, 2, 3, 3)

```
>>> value
tensor([[[[0.3834, 0.4149, 0.0386, 0.6322],
          [0.2314, 0.0329, 0.1547, 0.8799],
          [0.9404, 0.0906, 0.3780, 0.1874]],

         [[0.2396, 0.0976, 0.3314, 0.2551],
          [0.8268, 0.1233, 0.8254, 0.9927],
          [0.8714, 0.2468, 0.5333, 0.4027]]]])
>>> value.size()
torch.Size([1, 2, 3, 4])
```

(1, 2, 3, 4)

```
>>> attention_score = torch.matmul(masked_score, value)
>>> attention_score
tensor([[[[0.4260, 0.4610, 0.0429, 0.7025],
          [0.3081, 0.1647, 0.1329, 0.8946],
          [0.5409, 0.2657, 0.1699, 0.6452]],

         [[0.2662, 0.1084, 0.3682, 0.2834],
          [0.5688, 0.1217, 0.6228, 0.6635],
          [0.7012, 0.1808, 0.5897, 0.5466]]]])
>>> attention_score.size()
torch.Size([1, 2, 3, 4])
```

(1, 2, 3, 4)

GPT Layer - (2) Masked Multi-Head Self Attention

attention_score : (batch_size, head_num, seq_length, one_head_size)

↓ transpose

attention_score : (batch_size, seq_length, head_num, one_head_size)

↓ reshape (concat)

attention_score : (batch_size, seq_length, hidden_size)

```
>>> attention_score = attention_score.transpose(-3, -2).contiguous()
>>> attention_score
tensor([[[[0.4260, 0.4610, 0.0429, 0.7025],
          [0.2662, 0.1084, 0.3682, 0.2834]],

         [[0.3081, 0.1647, 0.1329, 0.8946],
          [0.5688, 0.1217, 0.6228, 0.6635]],

         [[0.5409, 0.2657, 0.1699, 0.6452],
          [0.7012, 0.1808, 0.5897, 0.5466]]]])
>>> attention_score.size()
torch.Size([1, 3, 2, 4])
```

(1, 3, 2, 4)

```
>>> original_shape = q.size()[:-3] + (q.size(-2), v.size(-1)  * 2)
>>> attention_score = attention_score.view(original_shape)
>>> attention_score
tensor([[[0.4260, 0.4610, 0.0429, 0.7025, 0.2662, 0.1084, 0.3682, 0.2834],
         [0.3081, 0.1647, 0.1329, 0.8946, 0.5688, 0.1217, 0.6228, 0.6635],
         [0.5409, 0.2657, 0.1699, 0.6452, 0.7012, 0.1808, 0.5897, 0.5466]]])
>>> attention_score.size()
torch.Size([1, 3, 8])
```

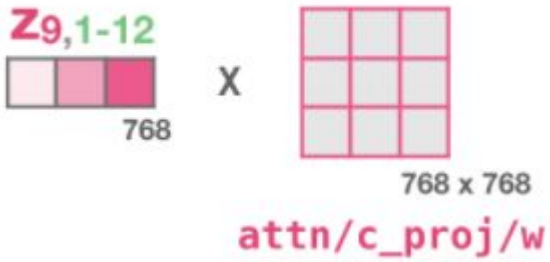
(1, 3, 8)

GPT Layer - (2) Masked Multi-Head Self Attention

attention_score : (batch_size, seq_length, hidden_size)

↓ torch.nn.Linear(hidden_size, hidden_size)

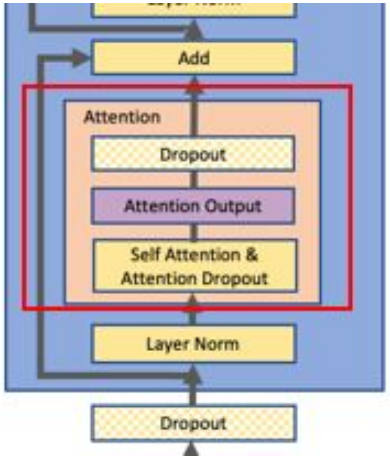
attention_score : (batch_size, seq_length, hidden_size)



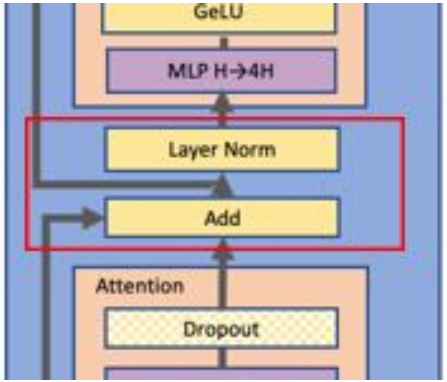

```
>>> torch.nn.Linear(8, 8)(attention_score)
tensor([[[-0.0119, -0.0287,  0.4279, -0.2876,  0.1720, -0.2352,  0.0506,
           0.3110],
         [-0.0359, -0.0296,  0.6278, -0.2626, -0.1402, -0.3118,  0.2231,
           0.4392],
         [-0.1137, -0.0358,  0.5320, -0.2846,  0.0603, -0.3574,  0.3066,
           0.4697]]], grad_fn=<AddBackward0>)
>>> torch.nn.Linear(8, 8)(attention_score).size()
torch.Size([1, 3, 8])
```

(1, 3, 8)

GPT Layer - (3) Skip Connection & Layer Norm

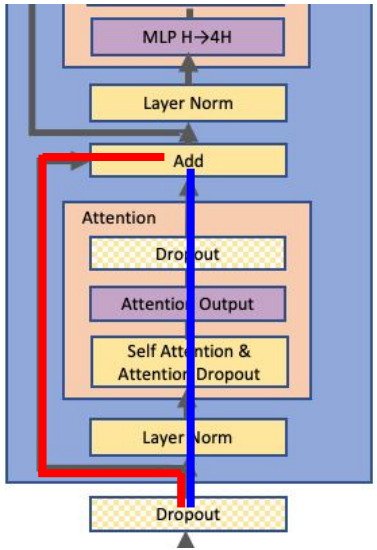


(2) Masked Multi-Head Self Attention



(3) Skip Connection & Layer Norm

GPT Layer - (3) Skip Connection & Layer Norm



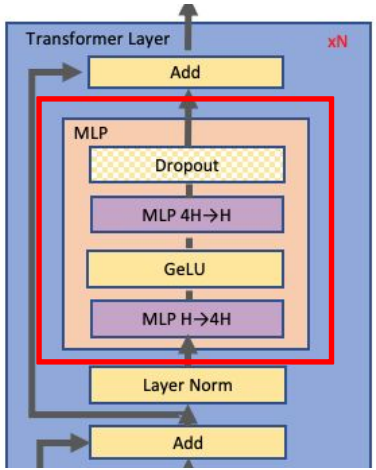
1. Input Vector
   (batch_size, seq_length, hidden_size)

2. Vector through Masked Multi-Head Self Attention
   (batch_size, seq_length, hidden_size)

   → Element-wise Addition = Skip Connection

   → Layer Normalization

GPT Layer - (4) Feed Forward Neural Network



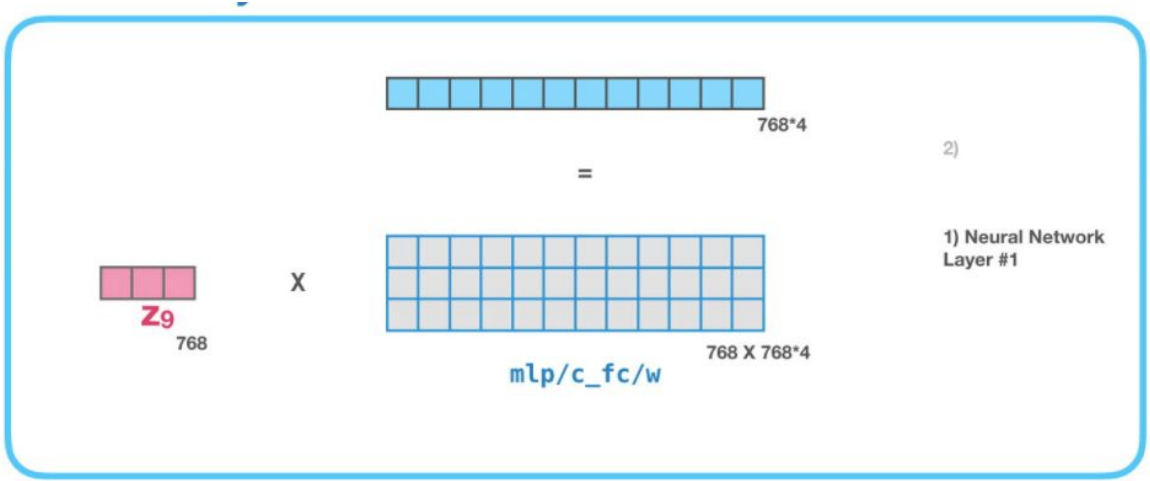

```python
import torch.nn as nn

class PositionWiseFeedForward(nn.Sequential):

    def __init__(self, dims: int, rate: int = 4, dropout: float = 0.1):
        super().__init__(
            nn.Linear(dims, dims * rate),  # (b, s, 4 * h)
            gelu(),  # Activation Function
            nn.Dropout(dropout),
            nn.Linear(dims * rate, dims))  # (b, s, h)

```

GPT Layer - (4) Feed Forward Neural Network

attention_output : (batch_size, seq_length, hidden_size)

↓ torch.nn.Linear(hidden_size, 4 * hidden_size)

(batch_size, seq_length, 4 * hidden_size)

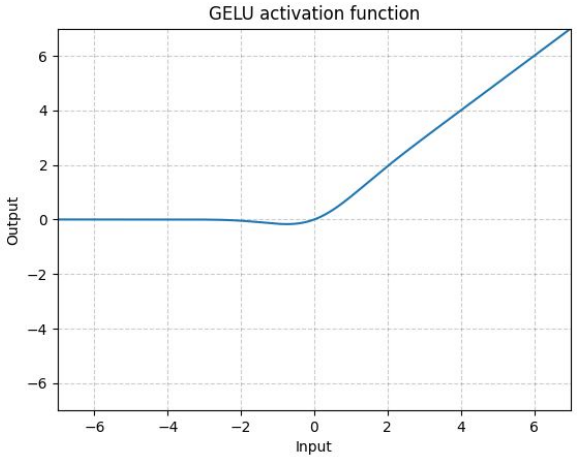


GPT Layer - (4) Feed Forward Neural Network

(batch_size, seq_length, 4 * hidden_size)

↓ GeLU

(batch_size, seq_length, 4 * hidden_size)



Gaussian Error Linear Unit

GPT Layer - (4) Feed Forward Neural Network

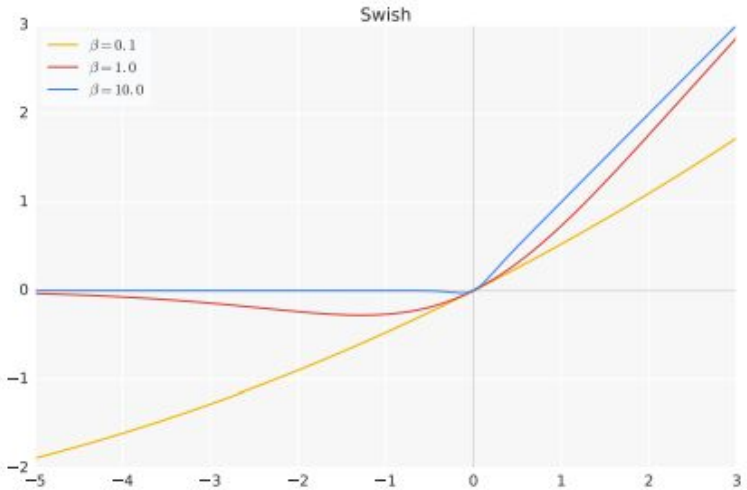


Figure 4: The Swish activation function.

```python
import torch.nn as nn


class Swish(nn.Module):

    def __init__(self):
        super().__init__()
        self.sigmoid = nn.Sigmoid()

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = x * self.sigmoid(x)

        return x

```

PyTorch Swish

GPT Layer - (4) Feed Forward Neural Network

(batch_size, seq_length, 4 * hidden_size)

torch.nn.Dropout(0.1)

(batch_size, seq_length, 4 * hidden_size)

```python
import torch.nn as nn

class PositionWiseFeedForward(nn.Sequential):

    def __init__(self, dims: int, rate: int = 4, dropout: float = 0.1):
        super().__init__(
            nn.Linear(dims, dims * rate),  # (b, s, 4 * h)
            gelu(),  # Activation Function
            nn.Dropout(dropout),
            nn.Linear(dims * rate, dims))  # (b, s, h)

```

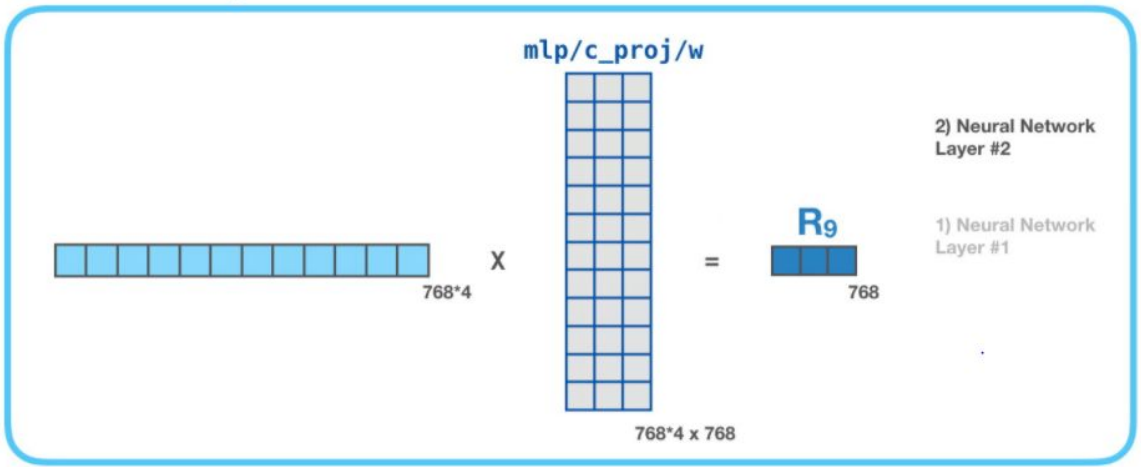
GPT Dropout

GPT Layer - (4) Feed Forward Neural Network

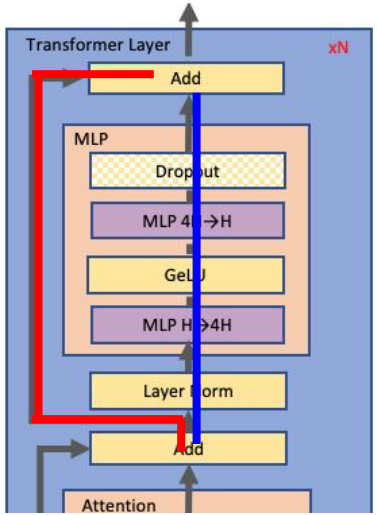
(batch_size, seq_length, 4 * hidden_size)

↓ torch.nn.Linear(4 * hidden_size, hidden_size)

(batch_size, seq_length, hidden_size)



GPT Layer - (5) Skip Connection

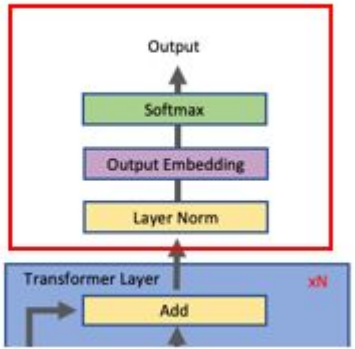


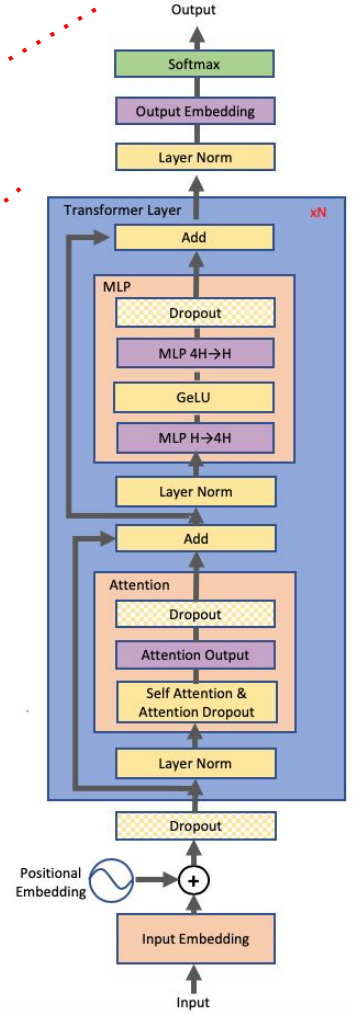
1. Vector skipped connection after attention (batch_size, seq_length, hidden_size)

2. Vector after Feed Forward Neural Network

→ Element-wise Addition = Skip Connection

Predicting Next Token





How to predict the next token using the vector from the continuous GPT blocks

Predicting Next Token - (1) Layer Normalization

(batch_size, seq_length, hidden_size)

torch.nn.LayerNorm

(batch_size, seq_length, hidden_size)

The output vector of the last GPT block is just Skip Connected.
It has not yet gone through Layer Normalization.

Therefore it finally goes through Layer Normalization.

Predicting Next Token - (2) Output Embedding

(batch_size, seq_length, hidden_size)

torch.nn.Linear(hidden_size, vocab_size)

(batch_size, seq_length, vocab_size)

hidden_size vector → vocab_size vector (logit)

weight of Token Embedding layer is used

Output Embedding is a Linear Layer that uses the weight of Token Embedding

Predicting Next Token - (2) Output Embedding

```
[ [0.3, 1.3, 0.2],
  [0.2, 1.0, 0.3],
  [-1.1, 4.3, 2.1],
  [-0.3, 3.4, 1.4],
  [1.1, 1.1, 1.3],
  [0.5, 1.3, 0.4],
  [0.1, -0.2, 5.3],
  [-1.7, 2.3, 1.3],
  [3.2, 1.1, 3.1]  ]
```

Token Embedding

(batch_size, seq_length, hidden_size)

↓ torch.nn.Linear(hidden_size, vocab_size)

(batch_size, seq_length, vocab_size)

```
[0.3, 1.3, 0.2]  *  [ [0.3, 0.2, -1.1, -0.3, 1.1, 0.5, 0.1, -1.7, 3.2],
                      [1.3, 1.0, 4.3, 3.4, 1.1, 1.3, -0.2, 2.3, 1.1],
                      [0.2, 0.3, 2.1, 1.4, 1.3, 0.4, 5.3, 1.3, 3.1]  ]
                  =  [1.8, 1.4, 5.7, 4.6, 2.0, 1.9, 0.8, 2.7, 3.0]
```

Vector | Output Embedding | Logit

The process of calculating the "appropriateness" of each token by the final vector from GPT layers

Predicting Next Token - (2) Output Embedding

```
>>> token_embedding = nn.Embedding(3, 6)
>>> output_embedding = nn.Linear(6, 3)
>>> output_embedding.weight = token_embedding.weight
>>> output_embedding.weight
Parameter containing:
tensor([[-0.4066, -0.0070, -0.3933, -1.6442, -1.5988,  0.5242],
        [ 0.9026,  2.1269,  1.7459,  0.2944, -0.2470,  0.1850],
        [ 0.6907, -1.0515, -1.7631, -1.0439,  0.8408,  1.0850]],
       requires_grad=True)
>>> output_embedding.weight.size()
torch.Size([3, 6])
```

Output Embedding

hidden_size → vocab_size Linear Layer

has same weight shape with Token Embedding

Both weights are 2D matrices of (vocab_size, hidden_size).

Predicting Next Token - (2) Output Embedding

```
>>> input_ids = torch.randint(0, 3, (2, 4))
>>> input_ids
tensor([[0, 0, 1, 2],
        [2, 2, 1, 0]])
>>> embed_vector = token_embedding(input_ids)
>>> embed_vector
tensor([[[-0.4066, -0.0070, -0.3933, -1.6442, -1.5988,  0.5242],
         [-0.4066, -0.0070, -0.3933, -1.6442, -1.5988,  0.5242],
         [ 0.9026,  2.1269,  1.7459,  0.2944, -0.2470,  0.1850],
         [ 0.6907, -1.0515, -1.7631, -1.0439,  0.8408,  1.0850]],

        [[ 0.6907, -1.0515, -1.7631, -1.0439,  0.8408,  1.0850],
         [ 0.6907, -1.0515, -1.7631, -1.0439,  0.8408,  1.0850],
         [ 0.9026,  2.1269,  1.7459,  0.2944, -0.2470,  0.1850],
         [-0.4066, -0.0070, -0.3933, -1.6442, -1.5988,  0.5242]]],
       grad_fn=<EmbeddingBackward>)
```

```
>>> output_embedding(embed_vector)
tensor([[[ 5.5999, -0.9882,  1.5457],
         [ 5.5999, -0.9882,  1.5457],
         [-1.3152,  8.6410, -4.8203],
         [ 1.1064, -4.9327,  7.8500]],

        [[ 1.1064, -4.9327,  7.8500],
         [ 1.1064, -4.9327,  7.8500],
         [-1.3152,  8.6410, -4.8203],
         [ 5.5999, -0.9882,  1.5457]]], grad_fn=<AddBackward0>)
```

Predicting Next Token - (3) Softmax

(batch_size, seq_length, vocab_size) Logit

Cross Entropy Loss ((batch_size, seq_length, vocab_size), (batch_size, seq_length))

Label

Summary

1. Language model / Tokens
2. Concept of Self Attention
3. How to train GPT EFFICIENTLY

Thanks