Transformer Network

Attention is all you need.

Why Transformer?

AMECA – ENGINEERED ARTS

- Transformer can accommodate multi-modal data – text, image, video, trajectories, action, control sequences, ….
- Essential for AGI (Artificial General Intelligence)

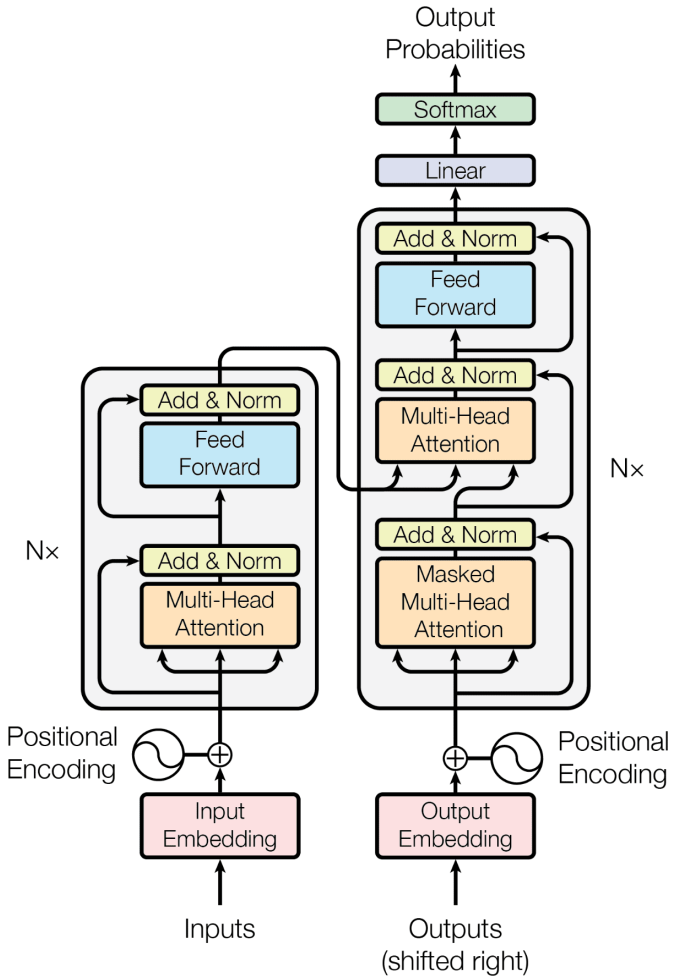
Example: RT-2 (Vision-Language-Action Model)

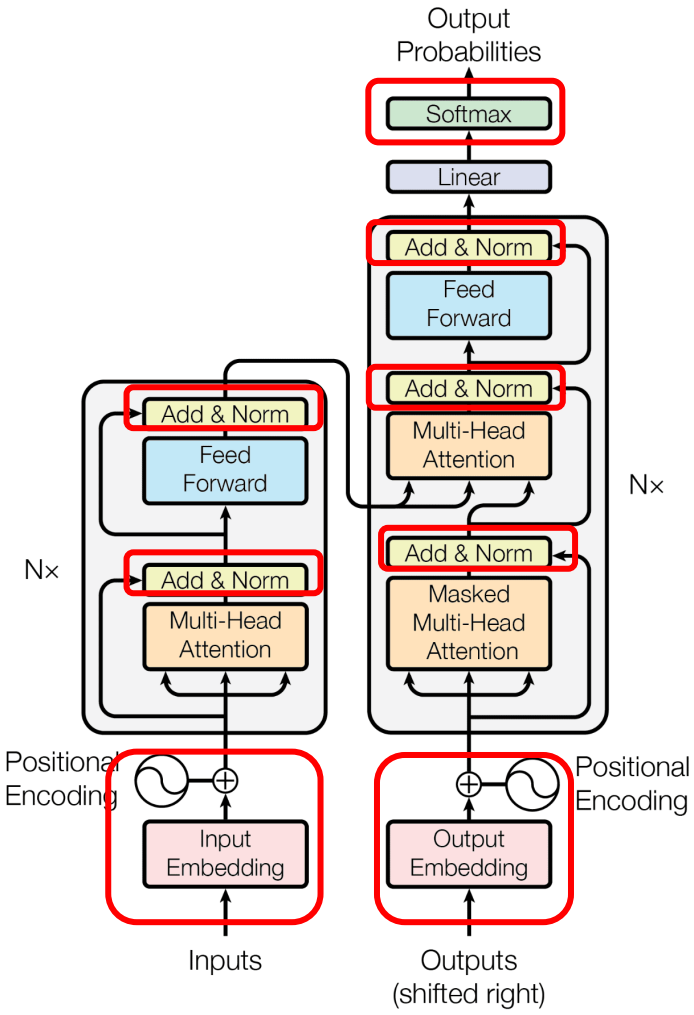
- Vision-Language-Action Models Transfer Web Knowledge to Robotic Control (July 2023, Google DeepMind)

Attention Is All You Need

- Transformer Network - Google 2017
- Developed for language model, replacing RNN(Recurrent Neural Network) completely

- GPT-1 – OpenAI 2018
- GPT-2 – OpenAI 2019
- GPT-3 – OpenAI 2020

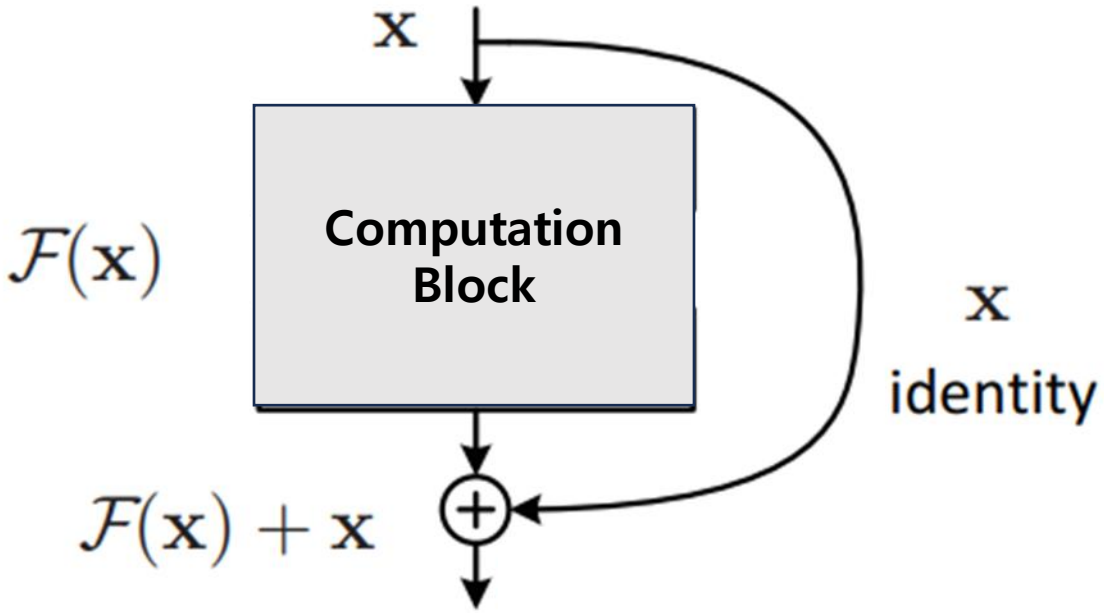


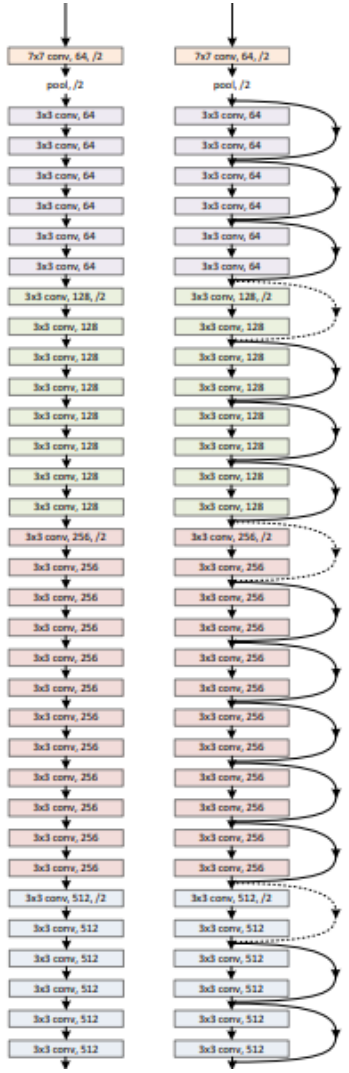

Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
N×
N×
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

Add & Norm

Add – Skip Connection

- ResNet – 2015 Deep Residual Learning for Image Recognition (285432회 인용), 오늘날 Deep Learning처럼 deep – large model을 가능하게 한 간단하지만 강력한 기술





Layer Normalization

- **Layer normalization over features : $\epsilon = 10^{-5}$ for computational stability, the shapes of γ and β are the same as the shape of x**

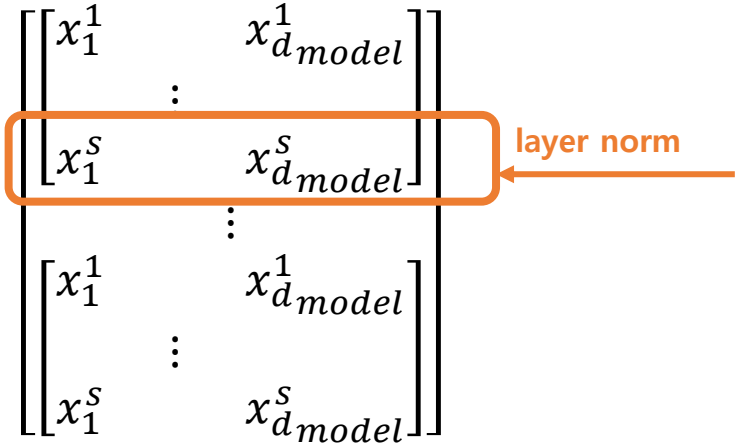
$$y = \frac{x - E[x]}{\sqrt{Var(x) + \epsilon}} \times \gamma + \beta$$

- **The operation targets the last axis of the input tensor (or axes starting from the last axis). If you need to normalize a different axis (e.g., one in the middle), you'll need to rearrange the tensor so that the desired axis is last, apply normalization, and then reverse the permutation.**

① 토큰 임베딩 벡터 x를 정규화하고,

② Gamma 벡터와 element-wise 곱하고,

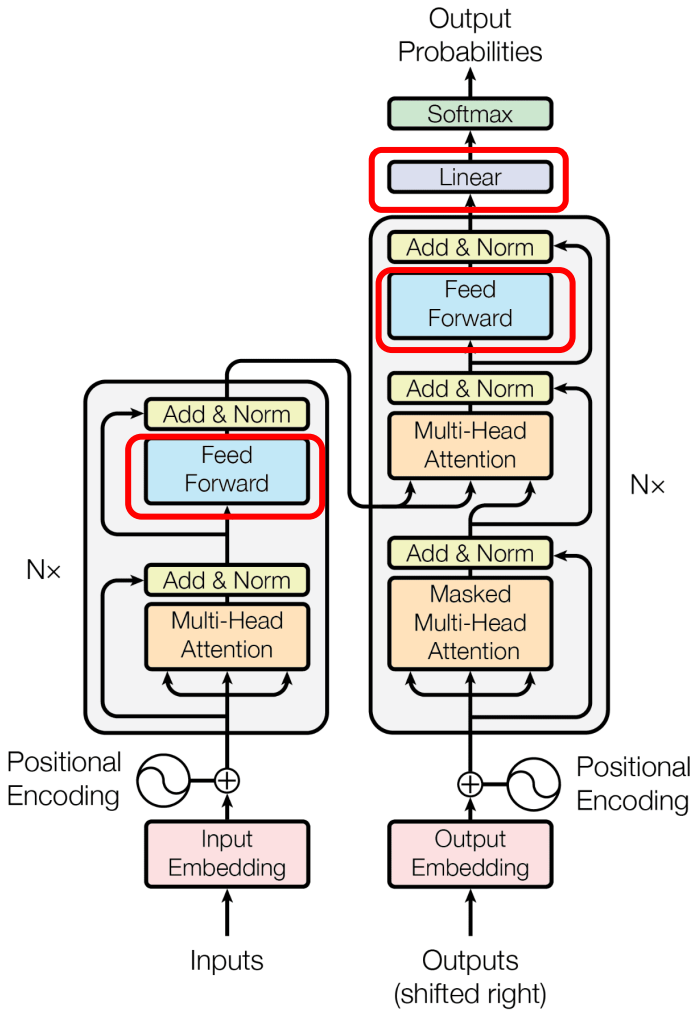
③ Beta 벡터를 더해서 벡터 y를 반환한다.



구현 예

```
class LayerNorm(nn.Module):
    def __init__(self, dim, eps=1e-5):
        super().__init__()
        self.eps = eps
        self.gamma = nn.Parameter(torch.ones(dim))
        self.beta = nn.Parameter(torch.zeros(dim))
    def forward(self, x):
        mean = x.mean(-1, keepdim=True)
        var = x.var(-1, keepdim=True, unbiased=False)
        x_norm = (x - mean) / torch.sqrt(var + self.eps)
        return self.gamma * x_norm + self.beta


x = torch.randn(2, 3)
layernorm = nn.LayerNorm(x.shape[-1:])
m = layernorm(x)
```

2



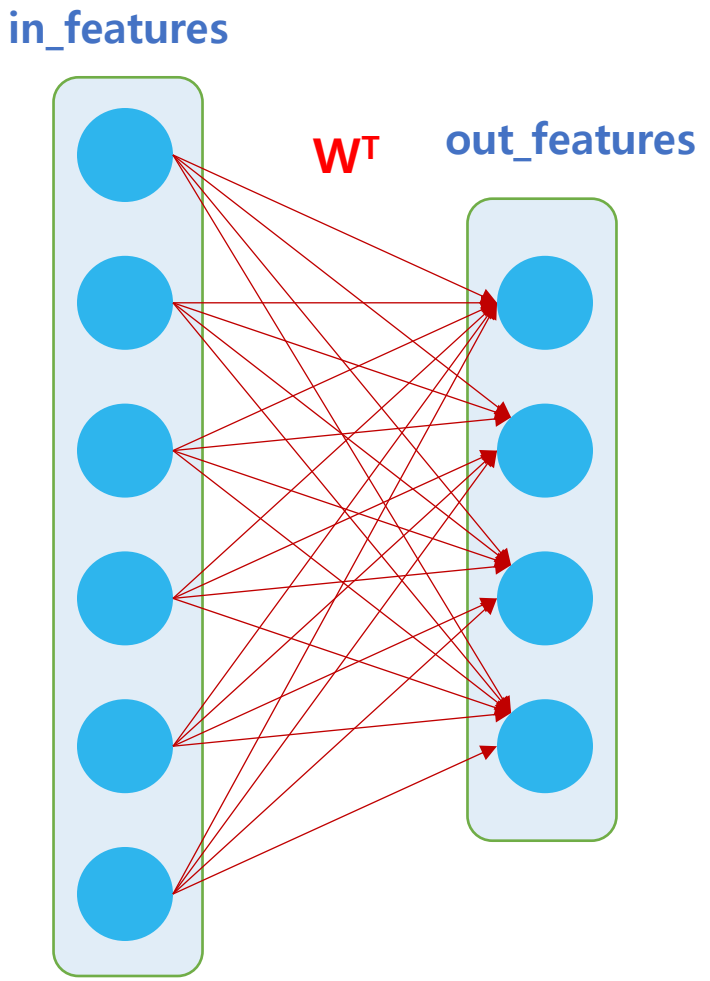
Linear & Feed Forward

Linear Layer

- $Y = XW^T + b$
  - `W = [out_features, in_features], b = [out_features,]`
  - W and b are broadcasted according to the input shape.

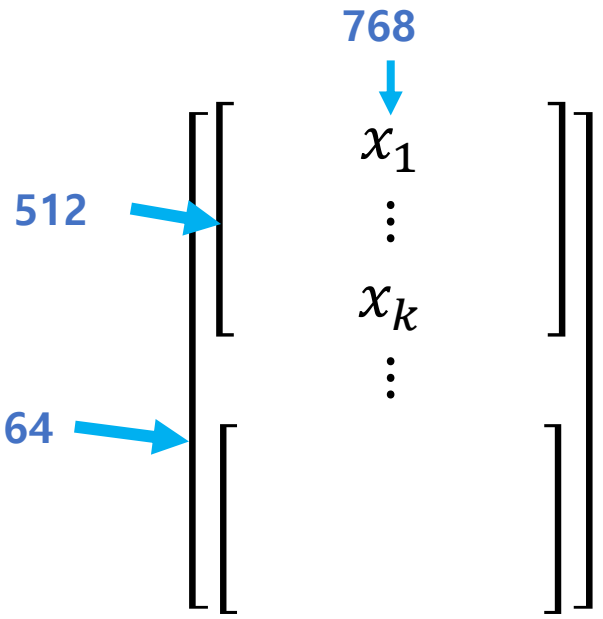
- Weight Matrix is **intentionally flipped** for *efficient computation*
  - Row-major: consecutive elements of a row are stored next to each other in memory. (row-major order in PyTorch, Python, C, Java, Rust).
  - Linear Layer에서, dot-product할 vectors를 row에 놓음(transpose는 실제 data element를 옮기지 않고 indexing에서 해결)

- **Input X could be any shape of tensor if last dimension matches in_features (Python Broadcasting Property).**
  - a vector = [in_features]
  - a matrix = [seq_len, in_features]
  - a 3D tensor = [batch_size, seq_len, in_features]



Example for Linear Layer in GPT-1

$$Y = Linear(X) = XW^T + b$$




```
Y = X @ W.T + b
```

$X = [64 \times 512 \times 768]$

$W = [2304 \times 768]$

$Y = [64 \times 512 \times 2304]$

Matrix multiplication: 1) no communications among input vectors, 2) **position of a vector(token) in input matrix does not affect the multiplication results.** 3) weights are learned over a whole training dataset. **4) GPUs are good at matrix multiplication.**

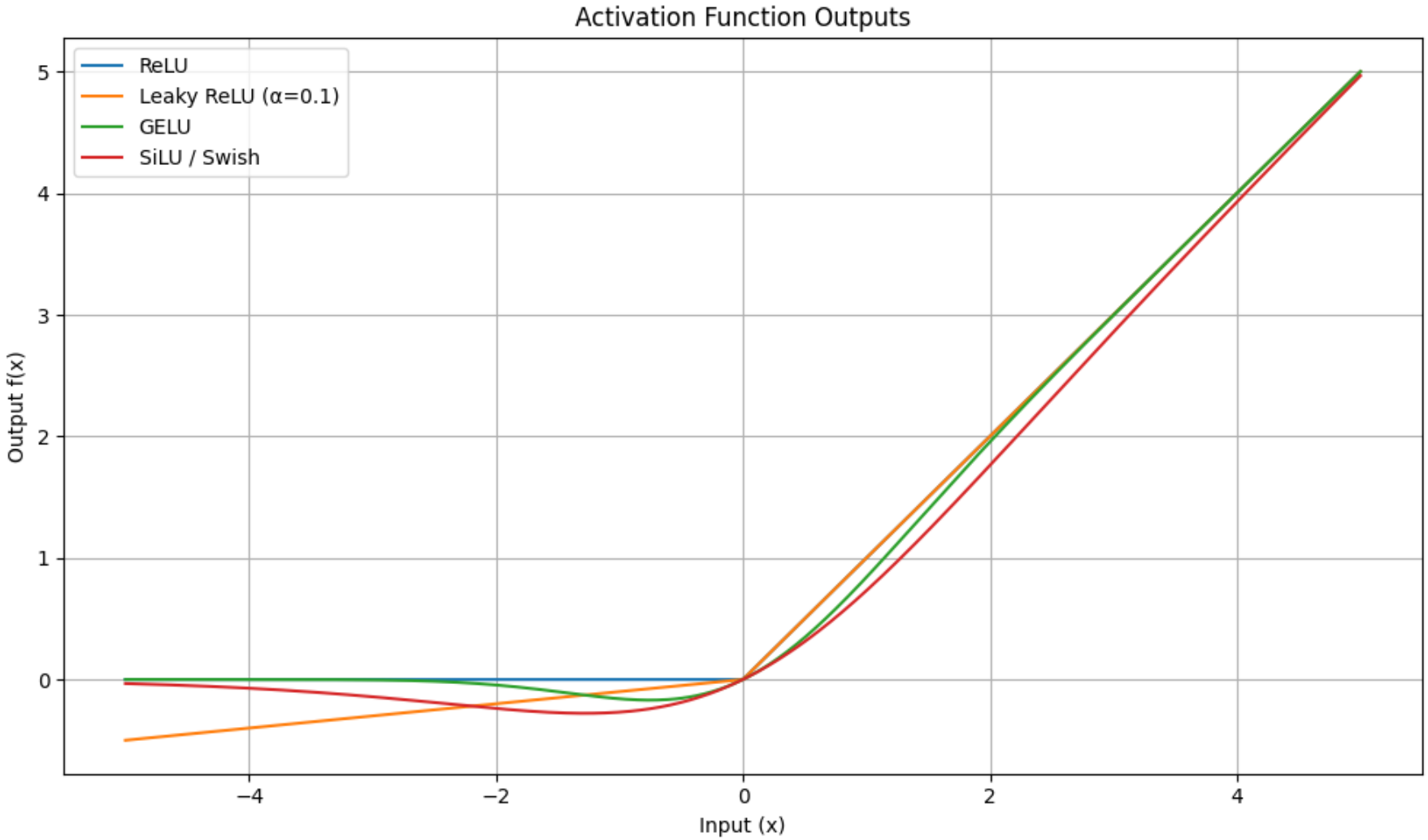
Linear Layer 구현 예

```
class LinearLayer(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(LinearLayer, self).__init__()
        self.weight = nn.Parameter(torch.randn(output_dim, input_dim))
        self.bias = nn.Parameter(torch.randn(output_dim))

    def forward(self, x):
        x = torch.matmul(x, self.weight.t()) + self.bias
        return x

x = torch.randn(2, 10, 5)  # Example input
layer = LinearLayer(5, 3)
output = layer(x)
print(output.shape)
```

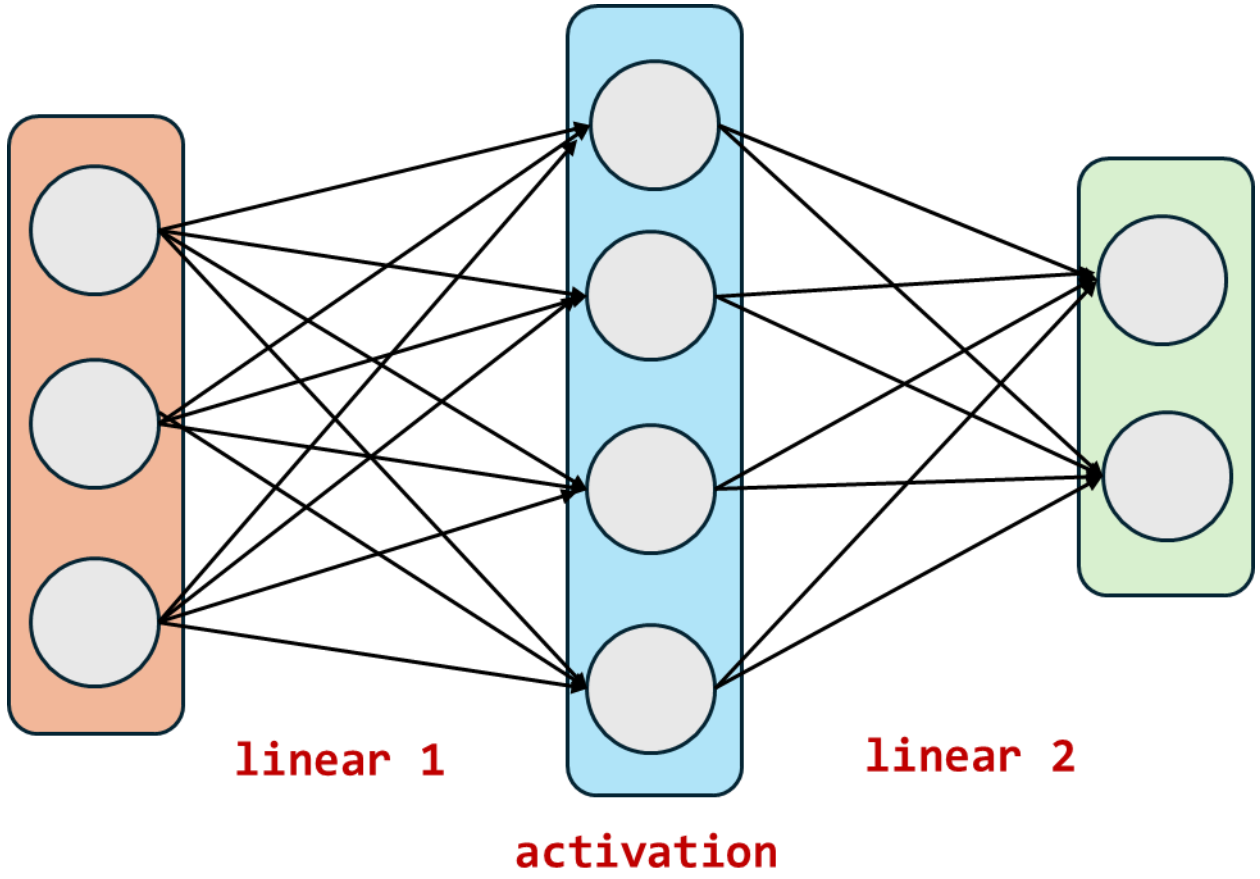
Activation Functions – 미분가능 하여야 함(differentiable)

Name	Formula	Output Range	Pros	Cons	Use Cases
ReLU	$\max(0, x)$	$[0, \infty)$	Fast, simple, sparse	Dying ReLU (zero gradient)	CNNs, MLPs (default choice)
Leaky ReLU	x if $x > 0$ else αx	$(-\infty, \infty)$	Avoids dead neurons	α needs tuning	MLPs with ReLU issues
Sigmoid	$\frac{1}{1+e^{-x}}$	$(0, 1)$	Probabilistic output	Vanishing gradient, not zero-centered	Output layer (binary)
Tanh	$\tanh(x)$	$(-1, 1)$	Zero-centered	Still vanishing gradients	RNNs, signal-like data
GELU	$x \cdot \Phi(x)$ (approx.)	$\approx (-0.17, \infty)$	Smooth, great for NLP tasks	Slightly more compute	Transformers (BERT, GPT)
Swish / SiLU	$x \cdot \text{sigmoid}(x)$	$\approx (-0.28, \infty)$	Smooth, self-gating	Slightly slower than ReLU	Vision, NLP, EfficientNet

Activation Functions



Feed Forward – MLP (Multi-Layer Perceptron)



GPT

① linear 1: (dim, 4*dim)
② activation: GELU
③ linear 2: (4*dim, dim)

MLP 구현 예

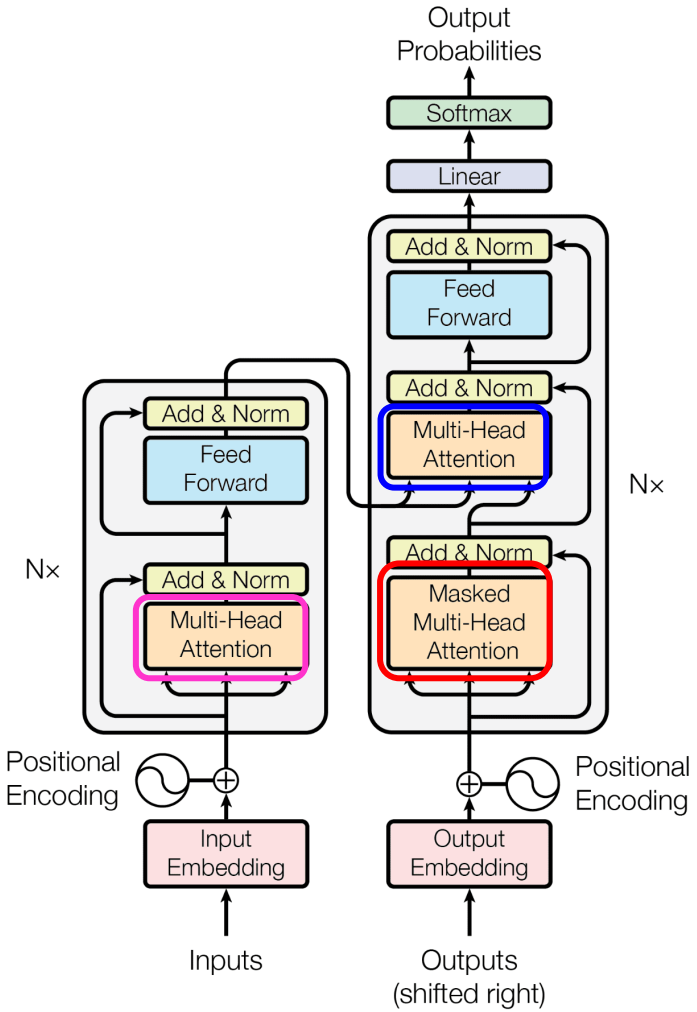
```python
class MLP(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, dropout=0.0, bias=True):
        super().__init__()
        self.c_fc    = nn.Linear(input_dim, hidden_dim, bias=bias)
        self.gelu    = nn.GELU()
        self.c_proj  = nn.Linear(hidden_dim, output_dim, bias=bias)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        x = self.c_fc(x)
        x = self.gelu(x)
        x = self.c_proj(x)
        x = self.dropout(x)
        return x


mlp = MLP(input_dim=768, hidden_dim=3072, output_dim=768, dropout=0.1)
x = torch.randn(2, 10, 768)  # input tensor with batch size 2 and sequence length 10
output = mlp(x)
print(output.shape)          # output: torch.Size([2, 10, 768])
```

(참고) dropout 학습 및 추론 시 동작

- 학습: p 확률로 노드 비활성화 하고 살아 남은 노드의 출력은 1/p 곱하여 보정
- 추론: 정상적인 노드 활성화

```
import torch
import torch.nn as nn

dropout = nn.Dropout(p=0.5)

x = torch.ones(10)

# Training 모드 (뉴런 일부 꺼지고 나머지는 scaling됨)
dropout.train()
print(dropout(x))

# Eval 모드 (dropout 비활성화, scaling 없음)
dropout.eval()
print(dropout(x))
```

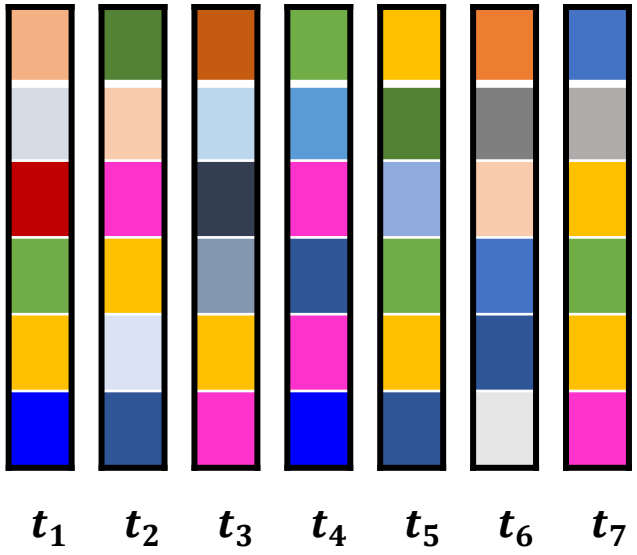
3



Attention (self, causal, cross)

기존 Layer 특징

- Linear normalization, skip connection & add, linear layer, MLP 는 token 별로 연산 수행
  - 토큰 하나 하나에 대하여 연산을 하기 때문에 토큰 처리하는 순서는 중요하지 않다 – GPU에서 병렬처리 가능
  - 연산 처리 순서는 중요하지 않지만, 출력 토큰 시퀀스 순서는 입력 토큰 시퀀스 순서대로 보존함
- 기존 layer들은 Language Model의 핵심인 조건부 확률을 예측할 수 있는 정보 추출 어려움
  - 위 layer들은 개별 토큰을 독립적으로 처리하기 때문에 토큰 상호 관계를 전혀 고려하지 못함
  - 기존 layer만으로는 주어진 토큰 시퀀스에서 다음 토큰이 나올 확률을 예측하는 조건부 확률 측정 어려움



$$y = \frac{x - E[x]}{\sqrt{Var(x) + \epsilon}} \times \gamma + \beta$$

$$\boldsymbol{Y = Linear(X) = XW^T + b}$$

$$\boldsymbol{y = x + F(x)}$$

어떻게 토큰 시퀀스 상호 관계를 신경망으로 표현할까?

- RNN (Recurrent Neural Network)

① 현재까지 입력된 토큰 시퀀스를 축약적으로 나타내는 상태 벡터를 메모리에 저장하고,

② 토큰을 순서대로 하나씩 입력

③ 현재 입력 토큰과 상태 벡터를 이용하여 다음 토큰을 예측하고, 상태 벡터를 업데이트 한다.

- $\boldsymbol{h_t = f(W_{xh}x_t + W_{hh}h_{t-1} + b_h)}$
- $\boldsymbol{y_t = g(W_{hy}h_t + b_y)}$
- $\boldsymbol{f(\cdot)}$: tanh 또는 ReLU 등 non-linear activation function
- $\boldsymbol{g(\cdot)}$: 보통 softmax

- RNN 특징 – 토큰 시퀀스 상호 관계를 상태 벡터를 이용하여 측정하기 때문에 Language Model의 조건부 확률 측정할 수 있으나, (a) 입력을 순차적으로 입력해야 하고 (b) 상태 벡터가 긴 문맥을 잘 표현하지 못하는 단점이 있다. LSTM이 b-관련 문제점 일부를 개선하였으나 Transformer 모델의 등장으로 영향력 쇠락함.

Attention - 직접적 토큰 시퀀스 상호 관계 연산

- Attention은 내적(dot-product)을 이용하여 시퀀스의 토큰 사이의 유사성을 측정하고 이를 토대로 토큰을 값을 스케일 하여 출력한다.

① 시퀀스에 있는 각 토큰 t_i을 Linear Layer를 이용하여 3개의 토큰 q_i, k_i, v_i 로 확장한다. (예: $t_1 \rightarrow q_1, k_1, v_1$)

② 여기서, q_i를 쿼리(query), k_i를 키(key), v_i를 값(value)이라 부른다.

③ 토큰 t_i 와 t_j 사이의 관계를 q_i 과 k_j 사이의 내적으로 구한다. (t_i 와 t_j 사이의 관계: $q_i \cdot k_j$)

④ 쿼리 q_i 와 시퀀스 내의 모든 키 k_j 와의 내적을 구하고 그에 대하여 softmax를 구한다.

⑤ Softmax 값 s_{ij}를 두 토큰 t_i 와 t_j 사이의 유사성 스코어(score) 라 한다.

⑥ 시퀀스 내의 모든 값 v_k 를 s_{ik} 로 스케일하고 합산한 새로운 토큰 $\sum_k s_{ik} v_k$ 을 시퀀스의 i 번째 토큰으로 출력한다.

- 전 과정을 다음과 같이 하나의 식으로 표현하면,

$$softmax\left(\frac{QK^T}{\sqrt{d_{model}}}\right)V$$

Q (Query), Key (Key), and V (Value)

- **전체 시퀀스 X = [B, T, d_{model}]**
  - B: 샘플 개수, T: 시퀀스 길이, d_{model}: 모델 크기

- **시퀀스 임베딩 텐서 X로부터 Q, K, V 행렬 구하기**
  - 하나의 토큰 임베딩 t_i를 Linear 맵핑으로 3개의 토큰 벡터 q_i, k_i, v_i 로 확장
  - Linear Layer의 웨이트는? $[d_{model}, d_{model}]$
  - $q_i = t_i W_Q^T$, $k_i = t_i W_K^T$, $v_i = t_i W_V^T$
  - 같은 Linear 맵핑으로 전체 시퀀스 X를 같은 크기의 Q, K, V 행렬로 확장
  - $Q = XW_Q^T$, $K = XW_K^T$, $V = XW_V^T$

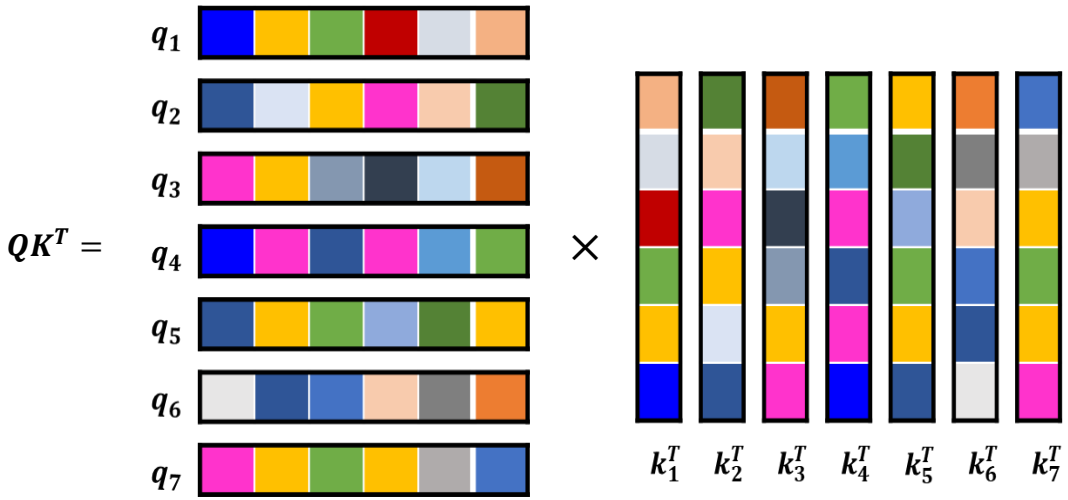
$$Q = \begin{bmatrix} q_1 \\ \vdots \\ q_T \end{bmatrix}, \quad K = \begin{bmatrix} k_1 \\ \vdots \\ k_T \end{bmatrix}, \quad V = \begin{bmatrix} v_1 \\ \vdots \\ v_T \end{bmatrix}$$

- **시퀀스 내 모든 토큰 간의 내적 구하기**
  - 토큰 t_i 와 t_j 사이의 유사성을 q_i 과 k_j 사이의 내적으로 측정 - $q_i \cdot k_j = q_i k_j^T$
  - 시퀀스 전체 토큰 간의 내적은 행렬 곱하기로 구함 - QK^T
  - QK^T 의 크기는? [B, T, d_{model}] x [B, d_{model}, T] → [B, T, T]

$q_i \cdot k_j =$  q_i ×  v_j^T

유사도 스코어 s_{ij}

- 시퀀스 내 모든 토큰 간의 유사성 스코어 구하기
  - $softmax\left(\frac{QK^T}{\sqrt{d_{model}}}\right)$: [B, T, T] 행렬
  - $\sqrt{d_{model}}$로 나누어 모델 크기가 커져서 내적이 커지는 것을 보정
  - 각 샘플의 소프트 맥스 행렬의 원소 s_{ij}는 토큰 t_i와 토큰 t_i의 유사도
  - 유사도 스코어 행렬의 크기는? [B, T, T]

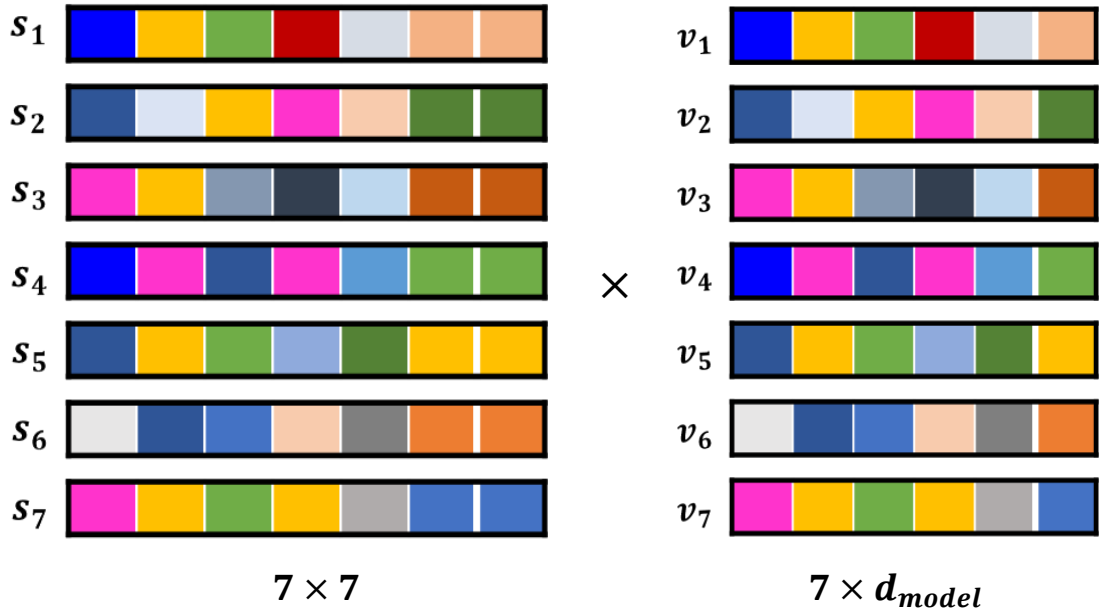


Attention 출력

▪ 유사도 스코어를 반영한 토큰 출력하기

- 토큰 t_i는 시퀀스 내 토큰과의 유사도를 반영하여 값(value) 행렬로부터 새로운 토큰으로 변환·출력된다.
- $t_i^{new} = s_{i1}v_1 + s_{i2}v_2 + \cdots + s_{iT}v_T$, 이를 행렬 형태로 바꾸면

- $t_i^{new} = [s_{i1} \quad \cdots \quad s_{iT}] \begin{bmatrix} v_1 \\ \vdots \\ v_T \end{bmatrix}$
- 시퀀스 내 모든 토큰에 대한 새로운 토큰을 구하면,
- $\begin{bmatrix} t_1^{new} \\ \vdots \\ t_T^{new} \end{bmatrix} = \begin{bmatrix} s_{11}v_1 + \cdots + s_{13}v_3 \\ \vdots \\ s_{T1}v_1 + \cdots + s_{TT}v_3 \end{bmatrix} = \begin{bmatrix} [s_{11} & \cdots & s_{1T}] \\ & \vdots & \\ [s_{T1} & \cdots & s_{TT}] \end{bmatrix} \begin{bmatrix} v_1 \\ \vdots \\ v_T \end{bmatrix}$
- $softmax\left(\frac{Q \cdot K^T}{\sqrt{d_{model}}}\right)V$
- 그림에서, s_i는 토큰 t_i에 상응하는
- 유사도 스코어 $s_i = [s_{i1}, \ldots, s_{iT}]$를 나타냄



Scaled-Dot-Product-Attention (SDPA)

```
import torch.nn.functional as F # PyTorch가 효율적으로 구현 - 직접 torch로 구현 사용(X)
B, T, d = 2, 7, 32   # 배치, 시퀀스 길이, 채널
q = torch.randn(B, T, d); k = torch.randn(B, T, d); v = torch.randn(B, S, d)


out = F.scaled_dot_product_attention(
    q, k, v,
    attn_mask=None,          # bool 또는 float 마스크 가능
    dropout_p=0.0,           # 학습 시만 >0 사용
    is_causal=False,         # 트랜스포머 디코더이면 True
)
```

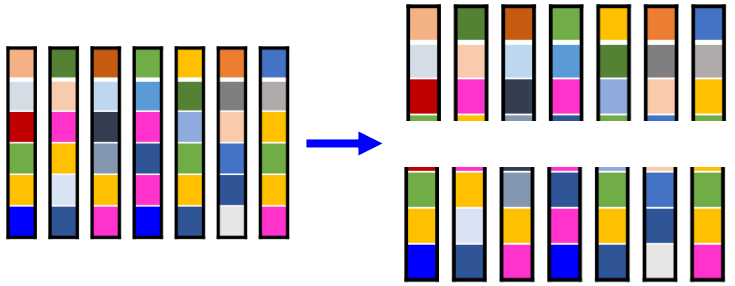
- q, k, v를 인자로 하여 함수 호출
- is_causal=True는 Causal-Attention을 위한 LLM 기본 옵션

Multi-Head Attention

1. 입력 텐서 X를 Q (Query), K (Key), V (Value)로 변환

- 크기가 [B, T, d_{model}]인 Tensor X를 3개의 Tensor K, V, Q로 변환
- 실제 구현은 Linear Layer를 사용

```
ln = torch.nn.Linear(d_model, 3 * d_model, bias=False) # bias는 옵션
Q, K, V = ln(X).chunk(3, dim=-1)
```



$d_{model} = 6$, n_head=2

2. Reshape K, V, Q from [B, T, d_{model}] to [B, n_head, T, d_{model} //n_head]

- B, T, d_{model} = X.shape
- K = K.view(B, T, n_head, d_{model}//n_head).transpose(1, 2)
- V와 Q도 같은 방식으로 reshape하여, multi-head tensor로 만들어서 attention에 입력
- (Remark) 4D Tensor에 대한 matrix multiplication은 근본적으로 2D와 다를 것이 없음

3. F.scaled_dot_product_attention 호출

- y = F.scaled_dot_product_attention(Q, K, V, attn_mask=None, dropout_p=0.0, is_causal=False)

4. 원래 dimension으로 복원

- y = y.transpose(1, 2).contiguous().vies(B, T, d_{model})

Attention 종류

- **Self Attention**
- 유사성 스코어를 계산할 때, 시퀀스 내 모든 토큰 사용하여 softmax 계산

- **Causal Attention**
- 유사성 스코어를 계산할 때, 시퀀스에서 현재까지의 토큰만 사용하여 softmax 계산
- 예를 들어, 토큰 t_3와는 t_1, t_2, t_3 토큰과의 유사성 스코어를 계산하여 다음 토큰 t_4를 예측
- Causal Masking을 이용하여 causal attention 수행

- **Cross Attention**
- 입력 텐서 X에서 Q만 생성
- 외부 텐서 Y에서 K, V를 생성
- 생성된 Q, K, V로 self attention을 수행

Masking for Causality

- **Training 할 때,**
  - ① 트랜스포머 모델은 시퀀스 내 모든 토큰을 한 번에 입력하기 때문에 next token을 예측할 때 'cheating' 가능하다.
  - ② 이를 방지하기 위하여, next token을 예측할 때 현재 토큰 이후의 토큰 정보를 masking한다.
  - ③ 예를 들어, 345를 예측할 때, 입력의 8491 이후 토큰 정보를 마스킹 하거나,
  - ④ 1692를 예측할 때, 8492, 345, 257 이후의 토큰 정보를 마스킹 한다.
- **Inference 할 때,**
  - ① 입력 토큰이 8491이면 345를 예측하고 예측된 토큰과 8491을 합해서 8491, 345가 다시 입력으로 들어 간다.
  - ② 입력 8491, 345에서 257을 예측하고, 다시 8491, 345, 257이 입력으로 들어 간다.
  - ③ 근본적으로 cheating이 불가능하며, 이 과정은 중단 조건이 만족될 때까지 반복된다.
  - ④ LLM은 이러한 Autoregression 방식으로 문장을 생성한다.

345	257	1692	393	257	9379	30	220

8491	345	257	1692	393	257	9379	30

Causality

- 토큰 t_i 에 대하여 유사성을 고려해야 할 시퀀스 내 토큰은,

① $t_1, t_2, \ldots, t_i$ 이며 (Note: 자기 자신 토큰도 포함)

② 그에 해당하는 key-value pairs는 $(k_1, v_1), (k_2, v_2), \ldots, (k_i, v_i)$ 이다. (여기서, $i \leq T$)

③ 쿼리 q_i 에 대하여, 부분 유사성 스코어 $[s_{i1} \quad \ldots \quad s_{ii}]$ 를 계산 한다.

- 토큰 t_i 와 유사성 스코어를 고려한 t_i^{new}는 다음과 같다.

$$s_{i1}v_1 + s_{i2}v_2 + \cdots + s_{ii}\,v_i$$

- 시퀀스 내 모든 토큰에 대한 식은,

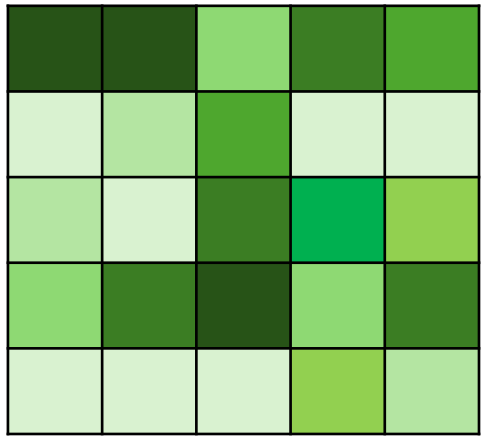
$$\begin{bmatrix} s_{11}v_1 \\ s_{21}v_1 + s_{22}v_2 \\ \vdots \\ s_{T1}v_1 + s_{22}v_2 + \cdots + s_{TT}v_T \end{bmatrix} = \begin{bmatrix} [s_{11} & & &] \\ [s_{21} & s_{22} & &] \\ & & \vdots & \\ [s_{T1} & s_{T2} & \ldots & s_{TT}] \end{bmatrix} \cdot \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_T \end{bmatrix}$$

How to implement the Mask?

- After performing QK^T, add the -∞ for those we want to exclude.
- 실제 구현은 더하지 않고 replace한다.

$$softmax(QK^T + Mask)$$

QK^T



+

$Mask$

0	-inf	-inf	-inf	-inf
0	0	-inf	-inf	-inf
0	0	0	-inf	-inf
0	0	0	0	-inf
0	0	0	0	0

Implementation

```
max_seq = 2048
mask = torch.tril(torch.ones(max_seq, max_seq)).view(1, 1, max_seq, max_seq)

att = (q @ k.transpose(-2, -1)) / (1.0 / math.sqrt(k.size(-1)))
att = att.masked_fill(mask[:,:,:seq,:seq] == 0, float('-inf'))
att = F.softmax(att, dim=-1)
y = att @ v

y = y.transpose(1, 2).contiguous().view(batch, seq, n_dim)
```