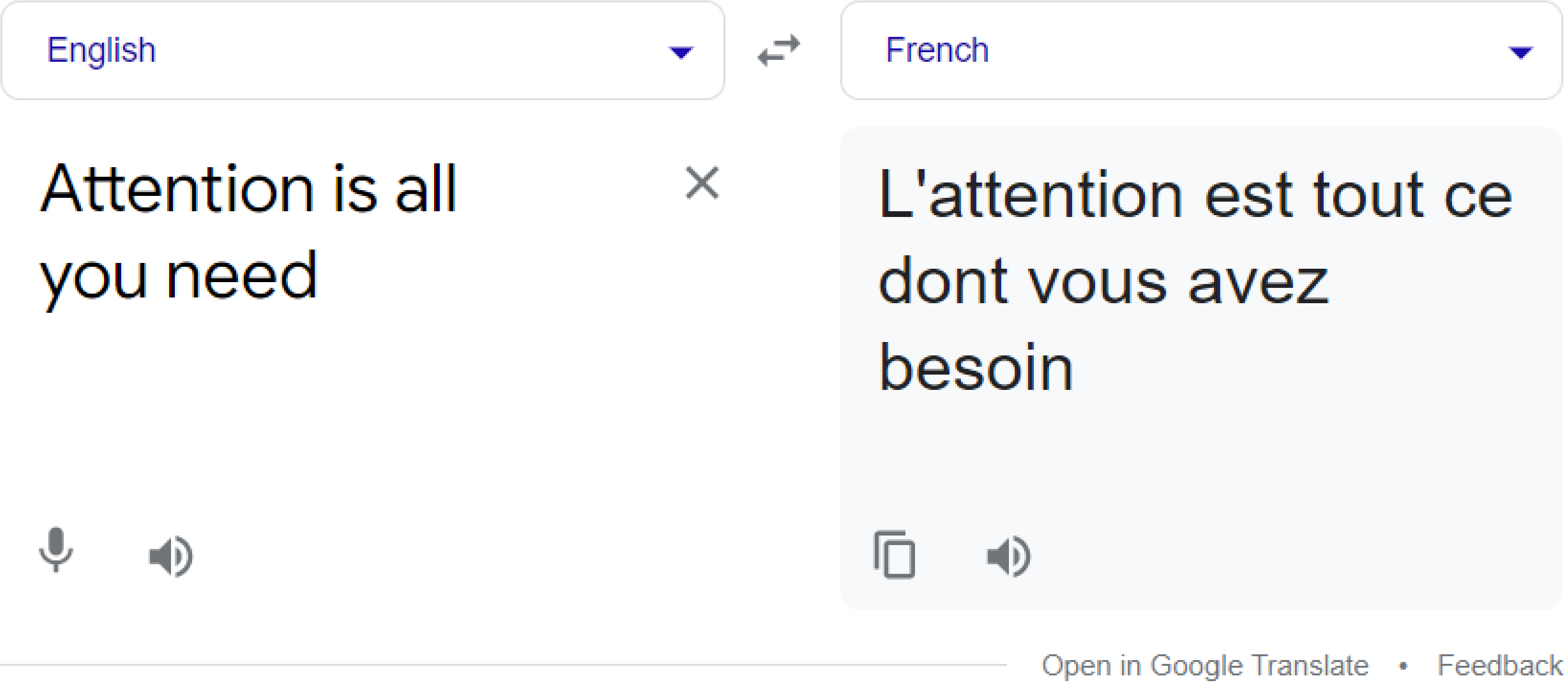
Attention Is All You Need

- 2017 -

TASK



영어를 프랑스어로 번역해보자 [(2014 Workshop on Statistical Machine Translation)]

Model Architecture

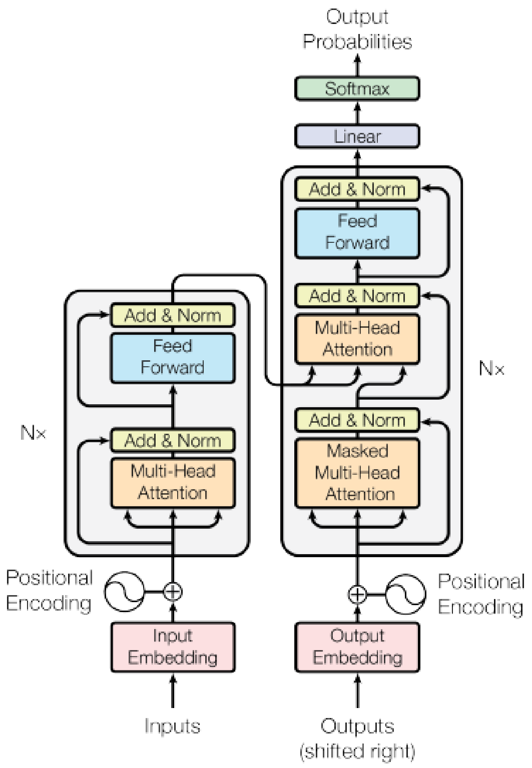
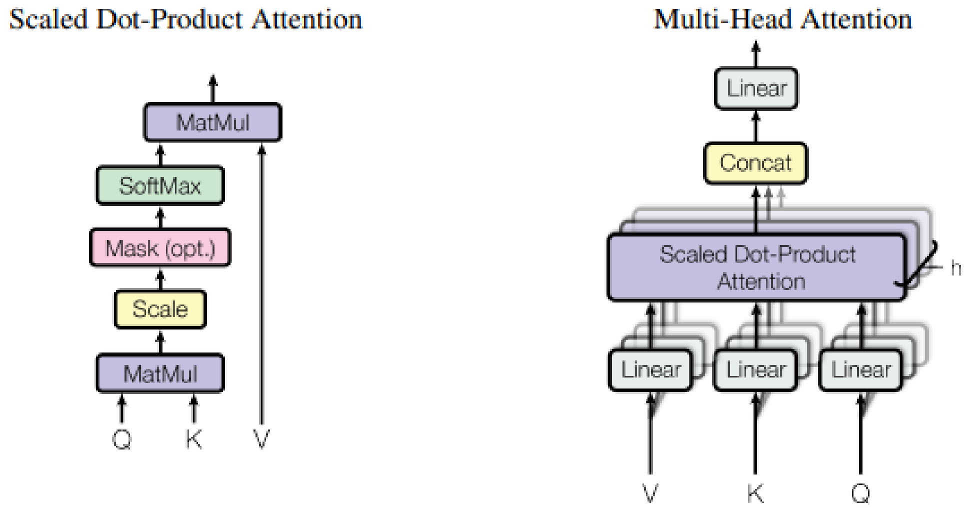


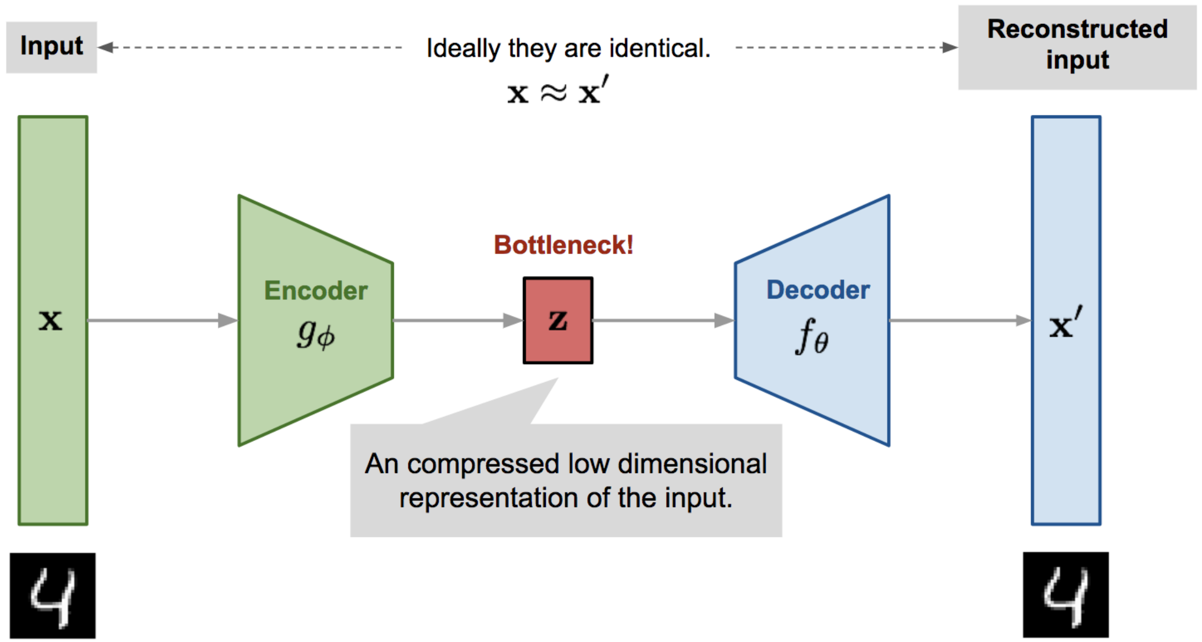
Figure 1: The Transformer - model architecture.

Encoder-Decoder Structure



Attention Mechanism

Encoder-Decoder Structure



Auto-Encoder Architecture

- ✓ 차원이 줄어들었다가 원래 차원으로 돌아오는 구조
- ✓ Loss: Input x와 output x′의 차이를 최소화
- ✓ 원래의 데이터를 다차원 벡터 z로 압축한 뒤 복원
- ✓ **Concept: z에는 x가 가진 패턴 중 유용한 것들만 쏙쏙**
- ✓ 이상탐지 분야에서 적극 활용
  - ✓ 정상 데이터로 모델을 학습시키면

    이상 데이터가 들어오면 복원을 잘 못할 듯

- ✓ 목표: 영어 문장에 들어있는 패턴을 추출해서

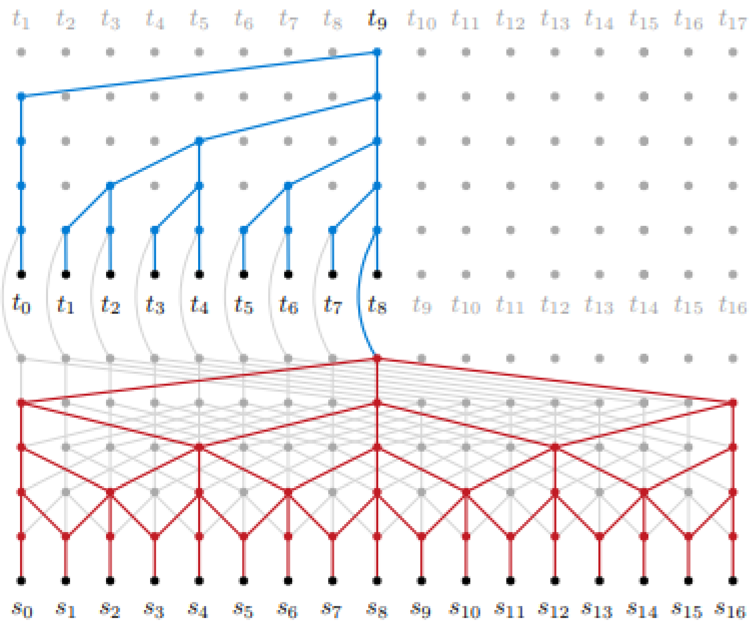
  Decoder 부분에서 프랑스어로 복원한다.

Neural Translation Model

$$p(\mathbf{t}|\mathbf{s}) = \prod_{i=0}^{N} p(t_i|t_{<i}, \mathbf{s})$$

- ✓ 주어진 source (s)를 가지고 target (t)를 예측한다.
- ✓ 각 정답 t에 대한 확률이 1에 가까울수록 Loss가 낮다.
- ✓ **Desiderata (필요조건)**
  - ✓ **1. 연산 시간이 linear (병렬처리 돼야 함)**
  - ✓ **2. source representation이 source 길이와 비례해야 함**
    - ✓ **문장 길이가 길면 representation도 길어져야 함**
  - ✓ **3. input과 target의 길이가 짧아야 함 (layer 너무 깊네 안 돼)**

Previous work



Byte-Net

Neural Machine Translation in Linear Time (2017.03)

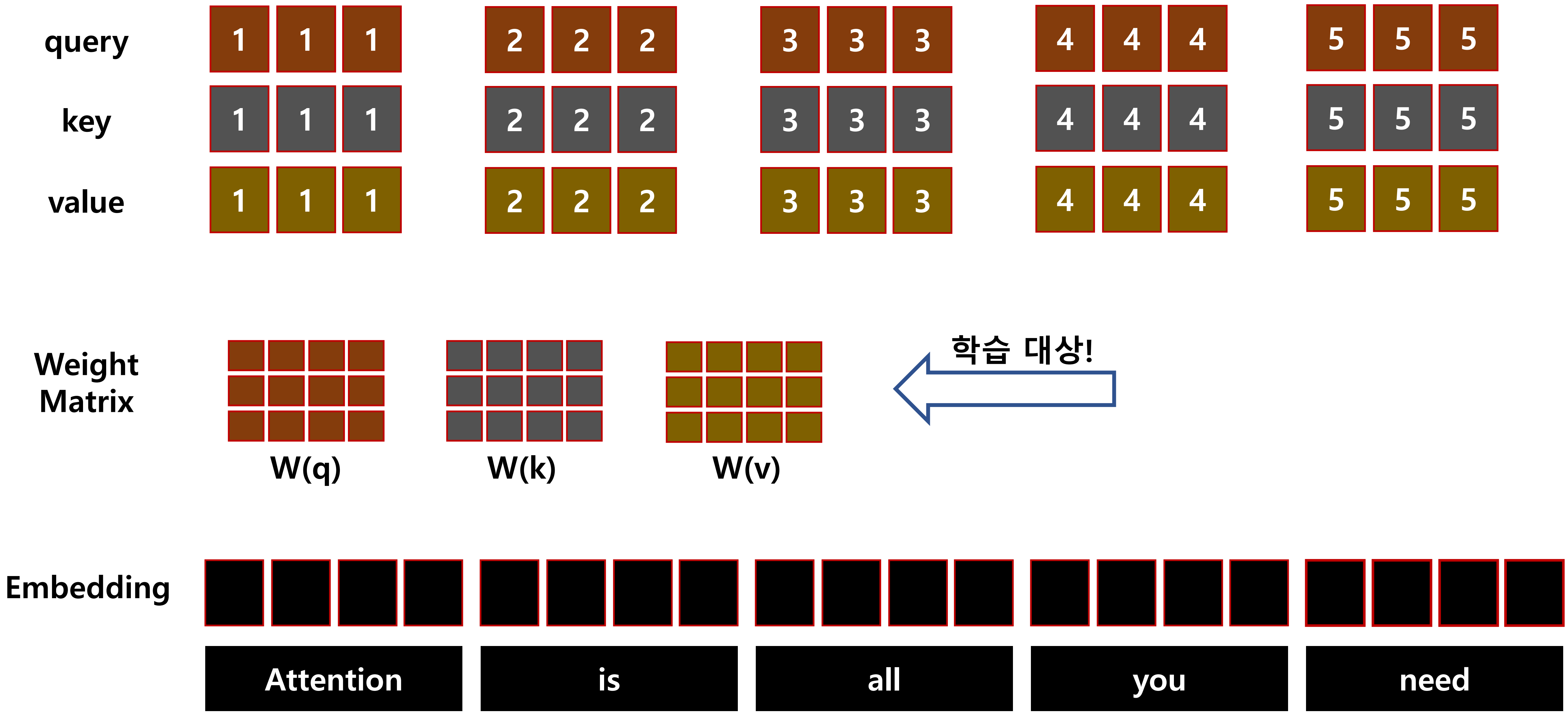
✓ Encoder (t째 노드에서 t+1 이상의 것 사용 함)

- ✓ 1 x 3 크기의 1-D convolution filter 적용 + (Dilation)
- ✓ 1-depth layer의 첫 번째 노드에는 (1,2,3) 단어 정보
- ✓ 2-depth layer의 첫 번째 노드에는 (1,2,3,4,5) 단어 정보

✓ Decoder (t째 노드에서 t+1 이상의 것 사용 안 함)

- ✓ 1-depth layer의 첫 번째 노드에는 앞 layer 1 노드 정보
- ✓ 1-depth layer의 두 번째 노드에는 앞 layer 1,2 노드 정보

Scaled Dot-Product Attention (Setting)



Scaled Dot-Product Attention (attention)

단어 “is”에 대한 embedding = z(0,2) (0번째 layer 2번째 노드)

단어 “is”가 하나의 attention layer를 넘어가고 난 다음 = z(1,2)

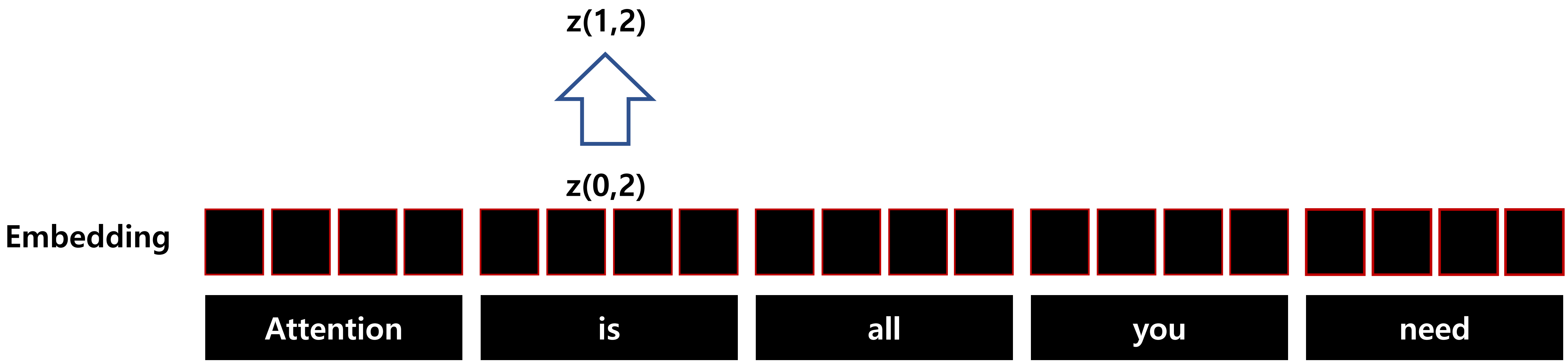
- z(1,2) = attention(2,1)*value(1,1) + a(2,2)*v(1,2) + a(2,3)*v(1,3) + a(2,4)*v(1,4) + a(2,5)*v(1,5)
- attention(2,1) = 2번 노드에 대한 1번 노드의 Attention

Scaled Dot-Product Attention (feed-forward)

단어 “is”에 대한 embedding = z(0,2) (0번째 layer 2번째 노드)

단어 “is”가 하나의 attention layer를 넘어가고 난 다음 = z(1,2)

- z(1,2) = attention(2,1)*value(1,1) + a(2,2)*v(1,2) + a(2,3)*v(1,3) + a(2,4)*v(1,4) + a(2,5)*v(1,5)
- attention(2,1) = 2번 노드에 대한 1번 노드의 Attention



Scaled Dot-Product Attention (attention)

attention(2,1) = 2번 노드에 대한 1번 노드의 attention

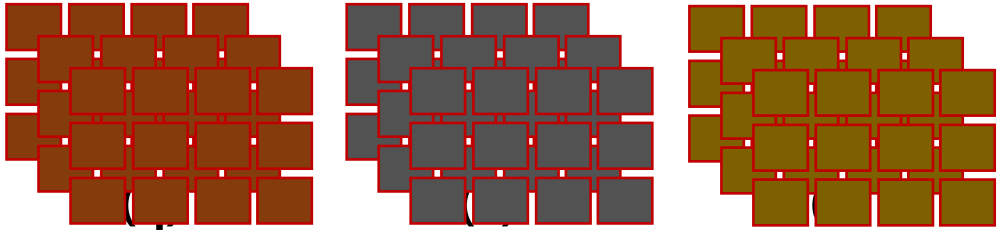
$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

attention(2,1)*value(1,1)

$$\textbf{Soft-max(} \begin{bmatrix} 2 & 2 & 2 \end{bmatrix} * \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \; / \; \sqrt{3} \;) * \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

Multi-head Attention

Weight Matrix



W(q) **W(k)** **W(v)**

- ✓ 하나의 attention layer를 거치고 나온 z의 벡터는 W(v)의 차원에 의존적
- ✓ 논문에서 원래 embedding dimension: 512
- ✓ 하나의 W(v)는 d(k) * 64 dimension
- ✓ 8종류의 weight matrix를 사용하여 64 dimension을 8개 얻고,
  최종적으로 연결하여 다시 512 dimension으로 만들어 냄

Positional Encoding

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{model}})$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{model}})$$

```python
class PositionalEncoding(nn.Module):

    def __init__(self, d_model: int, dropout: float = 0.1, max_len: int = 5000):
        super().__init__()
        self.dropout = nn.Dropout(p=dropout)

        position = torch.arange(max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) * (-math.log(10000.0) / d_model))
        pe = torch.zeros(max_len, 1, d_model)
        pe[:, 0, 0::2] = torch.sin(position * div_term)
        pe[:, 0, 1::2] = torch.cos(position * div_term)
        self.register_buffer('pe', pe)

    def forward(self, x: Tensor) -> Tensor:
        """
        Args:
            x: Tensor, shape [seq_len, batch_size, embedding_dim]
        """
        x = x + self.pe[:x.size(0)]
        return self.dropout(x)
```

```python
max_len = 5000
dropout = 0.1

position = torch.arange(max_len).unsqueeze(1)
print(position.shape)
print(position[:3])
```

```
torch.Size([5000, 1])
tensor([[0],
        [1],
        [2]])
```

```python
d_model  = 128
div_term = torch.exp(torch.arange(0, d_model, 2) * (-math.log(10000.0) / d_model))

print(div_term.shape)
print(div_term[:3])
```

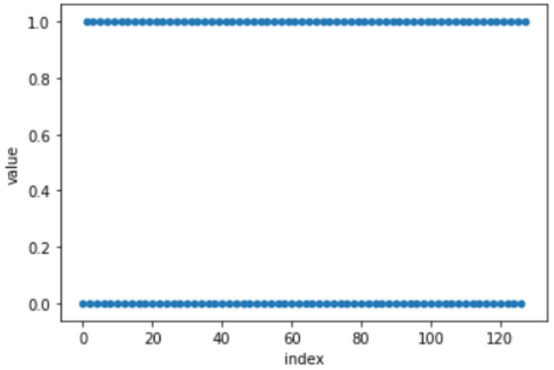
```
torch.Size([64])
tensor([1.0000, 0.8660, 0.7499])
```

```python
print((position * div_term).shape)
print((position * div_term)[1])
```

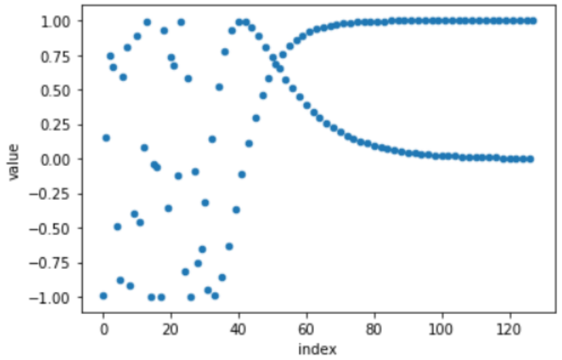
```
torch.Size([5000, 64])
tensor([1.0000e+00, 8.6596e-01, 7.4989e-01, 6.4938e-01, 5.6234e-01, 4.8697e-01,
        4.2170e-01, 3.6517e-01, 3.1623e-01, 2.7384e-01, 2.3714e-01, 2.0535e-01,
        1.7783e-01, 1.5399e-01, 1.3335e-01, 1.1548e-01, 1.0000e-01, 8.6596e-02,
        7.4989e-02, 6.4938e-02, 5.6234e-02, 4.8697e-02, 4.2170e-02, 3.6517e-02,
        3.1623e-02, 2.7384e-02, 2.3714e-02, 2.0535e-02, 1.7783e-02, 1.5399e-02,
        1.3335e-02, 1.1548e-02, 1.0000e-02, 8.6596e-03, 7.4989e-03, 6.4938e-03,
        5.6234e-03, 4.8697e-03, 4.2170e-03, 3.6517e-03, 3.1623e-03, 2.7384e-03,
        2.3714e-03, 2.0535e-03, 1.7783e-03, 1.5399e-03, 1.3335e-03, 1.1548e-03,
        1.0000e-03, 8.6596e-04, 7.4989e-04, 6.4938e-04, 5.6234e-04, 4.8697e-04,
        4.2170e-04, 3.6517e-04, 3.1623e-04, 2.7384e-04, 2.3714e-04, 2.0535e-04,
        1.7783e-04, 1.5399e-04, 1.3335e-04, 1.1548e-04])
```

Positional Encoding (Plot)

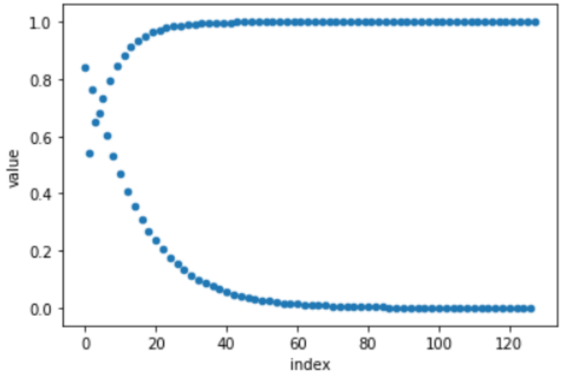
Position = 0



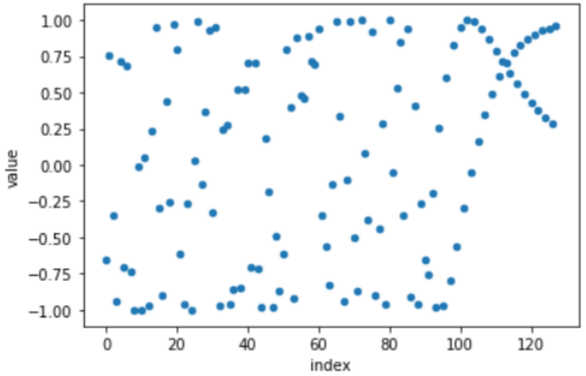
Position = 1000



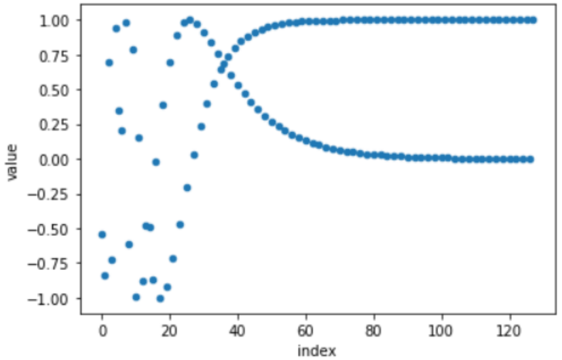
Position = 1



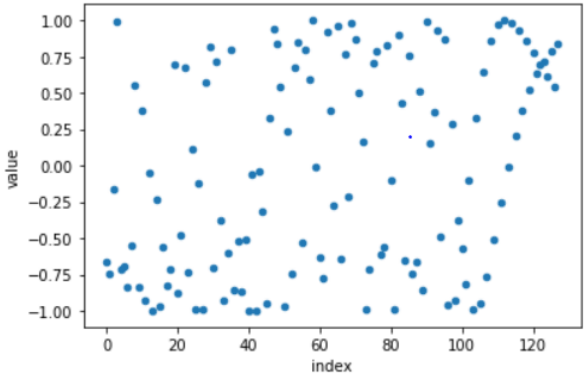
Position = 2500



Position = 10



Position = 4999



Masking

```python
def generate_square_subsequent_mask(sz: int) -> Tensor:
    """Generates an upper-triangular matrix of -inf, with zeros on diag."""
    return torch.triu(torch.ones(sz, sz) * float('-inf'), diagonal=1)

sz = 3

print(torch.ones(sz, sz)) ; print('')
print(torch.ones(sz, sz) * float('-inf')) ; print('')
print(torch.triu(torch.ones(sz, sz) * float('-inf'), diagonal=1)) ; print('')
```

```
tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]])

tensor([[-inf, -inf, -inf],
        [-inf, -inf, -inf],
        [-inf, -inf, -inf]])

tensor([[0., -inf, -inf],
        [0., 0., -inf],
        [0., 0., 0.]])
```

sz = 문장 길이

대각 상단이 –inf로 된 square matrix

Masking

```python
def _scaled_dot_product_attention(
    q,
    k,
    v,
    attn_mask = None,
    dropout_p = 0.0
):

    B, Nt, E = q.shape
    q = q / math.sqrt(E)

    # (B, Nt, E) x (B, E, Ns) -> (B, Nt, Ns)

    if attn_mask is not None:
        attn = torch.baddbmm(attn_mask, q, k.transpose(-2, -1))
    else:
        attn = torch.bmm(q, k.transpose(-2, -1))

    attn = softmax(attn, dim=-1)
    if dropout_p > 0.0:
        attn = dropout(attn, p=dropout_p)
    # (B, Nt, Ns) x (B, Ns, E) -> (B, Nt, E)
    output = torch.bmm(attn, v)
    return output, attn
```

B = 배치 사이즈
Nt = 문장 길이
E = Embedding dimension

TORCH.BADDBMM

`torch.baddbmm(input, batch1, batch2, *, beta=1, alpha=1, out=None)` → Tensor

Performs a batch matrix-matrix product of matrices in `batch1` and `batch2`. `input` is added to the final result.

`batch1` and `batch2` must be 3-D tensors each containing the same number of matrices.

If `batch1` is a $(b \times n \times m)$ tensor, `batch2` is a $(b \times m \times p)$ tensor, then `input` must be broadcastable with a $(b \times n \times p)$ tensor and `out` will be a $(b \times n \times p)$ tensor. Both `alpha` and `beta` mean the same as the scaling factors used in `torch.addbmm()`.

$$\text{out}_i = \beta\ \text{input}_i + \alpha\ (\text{batch1}_i \mathbin{@} \text{batch2}_i)$$

```python
mask   = torch.triu(torch.ones(sz, sz) * float('-inf'), diagonal=1)
query  = torch.tensor([[1,2,3,4,5],[4,5,6,7,8],[7,8,9,10,11]]).to(torch.float)
key    = torch.tensor([[1,2,3,4,5],[4,5,6,7,8],[7,8,9,10,11]]).to(torch.float).transpose(-2,-1)

print(mask) ; print('')
print(query) ; print('')
print(torch.mm(query, key))
```

```
tensor([[0., -inf, -inf],
        [0., 0., -inf],
        [0., 0., 0.]])

tensor([[ 1.,  2.,  3.,  4.,  5.],
        [ 4.,  5.,  6.,  7.,  8.],
        [ 7.,  8.,  9., 10., 11.]])

tensor([[ 55., 100., 145.],
        [100., 190., 280.],
        [145., 280., 415.]])
```

Code Exercise (scale dot attention)

```
def scale_dot_product_attention(q,k,v, mask=False):

    # (3,128) head:4 -> 4, 3, 32
    # 4, 3, 32 -> 4, 32, 3

    head_size, sentence_size, embedding_size = q.shape
    q = q / np.sqrt(embedding_size)

    attn = np.matmul(q,k.transpose(0,2,1))

    # q.shape                                  = (head_size, sentence_size, emedding_size)
    # k.transpose(0,2,1).shape                 = (head_size, emedding_size, sentence_size)
    # np.matmul(q, k.transpose(0,2,1)).shape = (head_size, sentence_size, sentence_size)

    if mask:
        def square_mask(head_size, sentence_size):
            single_mask = np.triu(np.ones([sentence_size,sentence_size]) * float('-inf'), k=1)
            multi_mask  = np.dstack([single_mask]*head_size)
            return np.array([multi_mask[:,:,0],multi_mask[:,:,1],multi_mask[:,:,2]])
        attn += square_mask(sentence_size)

    def softmax(attn):
        return np.exp(attn) / np.sum(np.exp(attn), axis=2)[:,:,None]
    attn = softmax(attn)
    attn = np.matmul(attn, v).transpose(1,0,2).reshape(sentence_size,-1)
    return attn
```

Code Exercise (multi head attention)

```python
# multi_head_attention_forward (pytorch github 참조)
def multi_head_attention(
    X,         # X.shape (sentence_size, embedding_size)
    head_size, # multi head 몇 개?
    q,         # q.shape (sentence_size, embedding_size)   / W_q.shape (embedding_size, embedding_size)
    k,         # k.shape (sentence_size, embedding_size)   / W_k.shape (embedding_size, embedding_size)
    v          # v.shape (sentence_size, embedding_size)   / W_v.shape (embedding_size, embedding_size)
    ):

     sentence_size, embedding_size = X.shape

     q = np.expand_dims(q, axis=1) # (sentence_size, embedding_size) -> (sentence_size, 1, embedding_size)
     k = np.expand_dims(k, axis=1)
     v = np.expand_dims(v, axis=1)

     # (3, 128) -> (3, 1, 128) -> head=4 -> (3,4,32).transpose(1,0,2) -> (4,3,32) -> (head_size, sentence_size, embdding_size / head_size)
     #  128 -> [32][32][32][32] -> [128]

     # (sentence_size, 1, embedding_size) -> (head_size, sentence_size, embedding_size / head_size)
     q = q.reshape(sentence_size, head_size, embedding_size / head_size).transpose(1,0,2)
     k = k.reshape(sentence_size, head_size, embedding_size / head_size).transpose(1,0,2)
     v = v.reshape(sentence_size, head_size, embedding_size / head_size).transpose(1,0,2)

     new_X = scale_dot_product_attention(sentence_size, q,k,v)
     return new_X
```