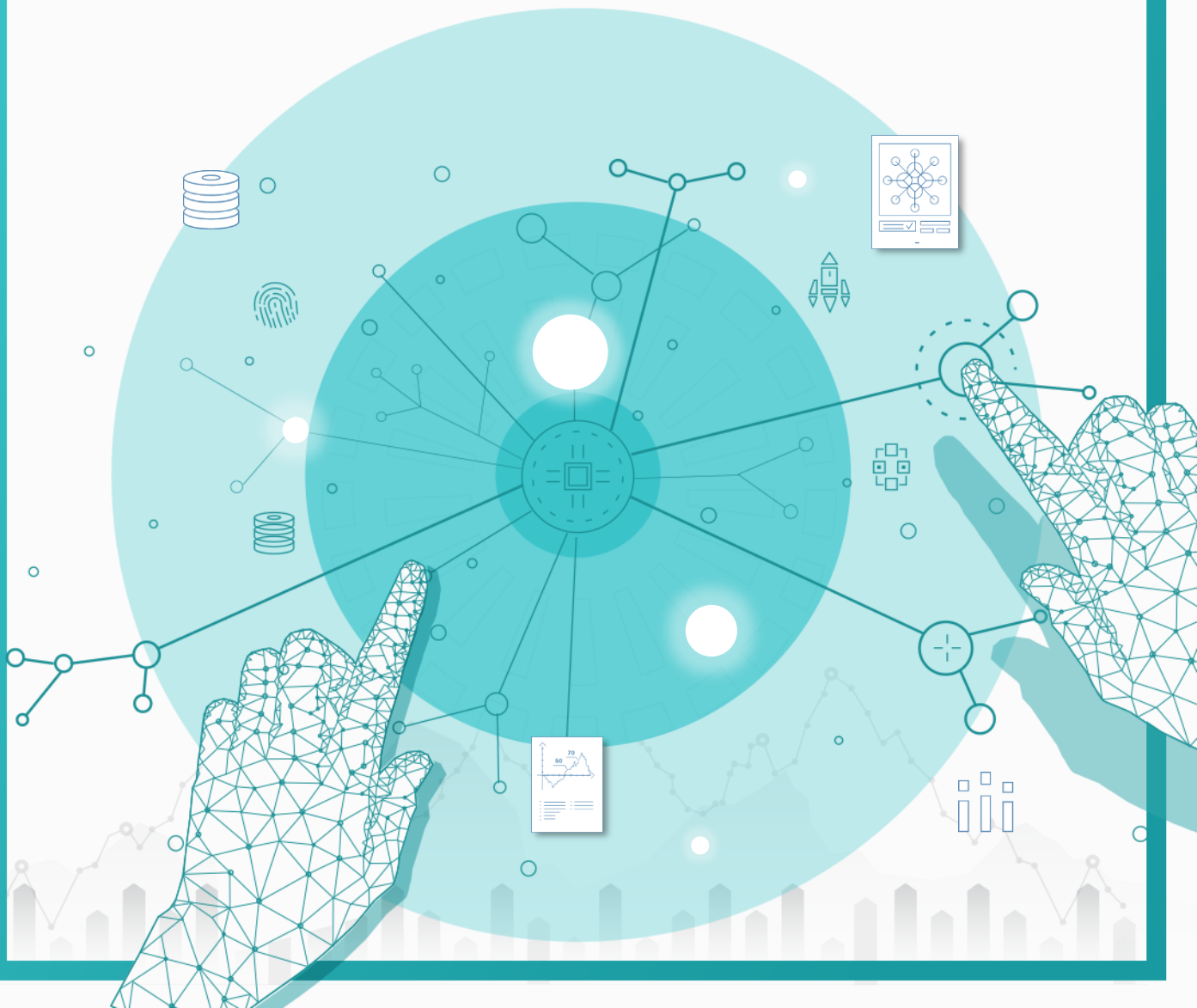
한국기술교육대학교
온라인평생교육원

# 파이썬을 활용한 인공지능 자연어 처리(실습)

## 자연어 처리를 위한 Transformer 적용



# 자연어 처리를 위한 Transformer 적용

## 학습 목표

1. Attention과 Transformer를 구현할 수 있다.
2. Transformer를 감정분석에 적용할 수 있다.

## 학습 내용

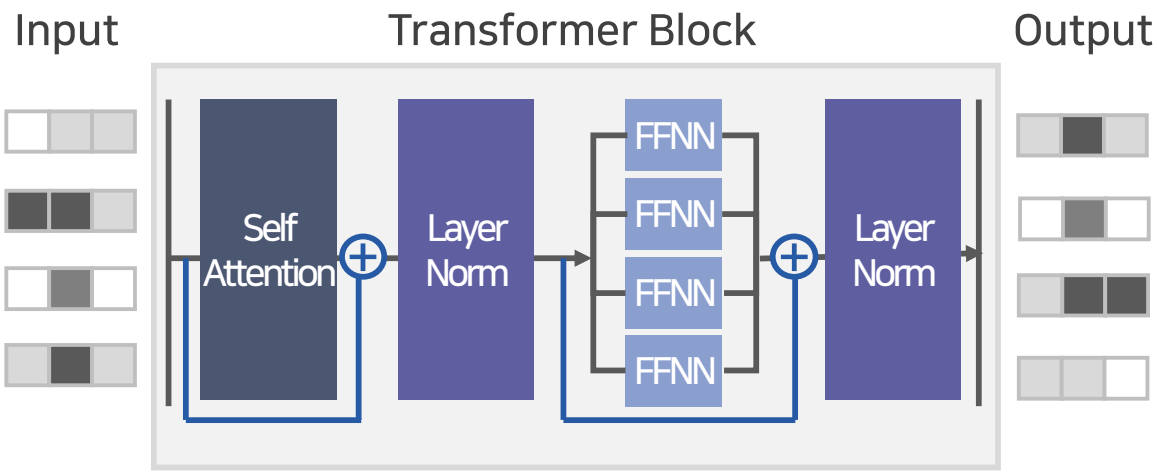
1. Attention과 Transformer 구현
2. Transformer 감성분석 적용

# 1. Attention과 Transformer 구현

## 1) Self-Attention과 Transformer

### (1) Transformer Block 내부 구조

Self Attetion, Normalization Layer#1,
Feed Forward Neural Network, Normalization Layer#2 및
Residual Network(Skip Connection) 등으로 구성



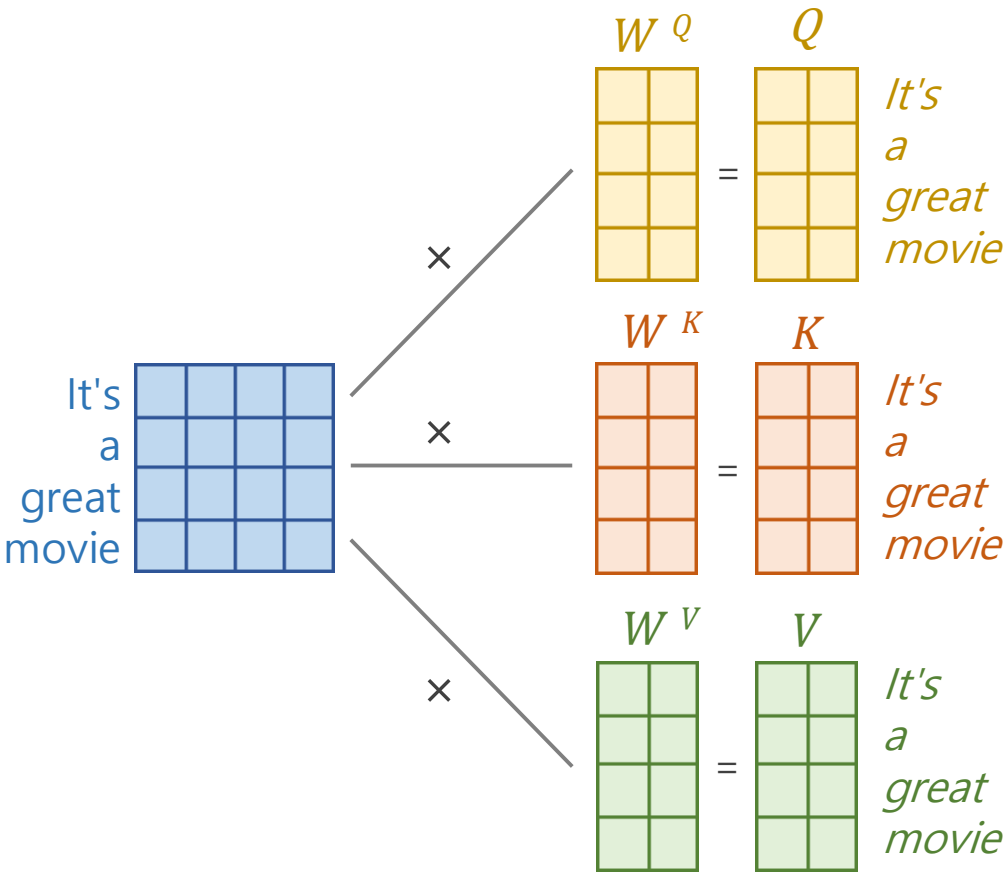

### (2) Query, Key, Value 계산

Self-Attention의 첫 단계는 입력 문장에 대한
Query, Key, Value 계산

Input 문장의 512크기의 벡터와 학습할 Weight(WQ, WK, WV)를 곱하여 64크기의 Query, Key, Value 벡터 생성

# 1. Attention과 Transformer 구현

## 2) Self-Attention 구현



## 2) Self-Attention 구현

### (1) Query, Key, Value 계산

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

    b, l, d = x.size()
    h = self.h

    queries = self.WQ(x).view(b, l, h, d).transpose(1, 2).contiguous().view(b
    keys = self.WK(x).view(b, l, h, d).transpose(1, 2).contiguous().view(b *
    values = self.WV(x).view(b, l, h, d).transpose(1, 2).contiguous().view(b

    w_prime = torch.bmm(queries, keys.transpose(1, 2)) / np.sqrt(d)
    w = F.softmax(w_prime, dim=-1)

    out = torch.bmm(w, values).view(b, h, l, d)

    out = out.transpose(1, 2).contiguous().view(b, l, h * d)

    return self.unifyheads(out)
```

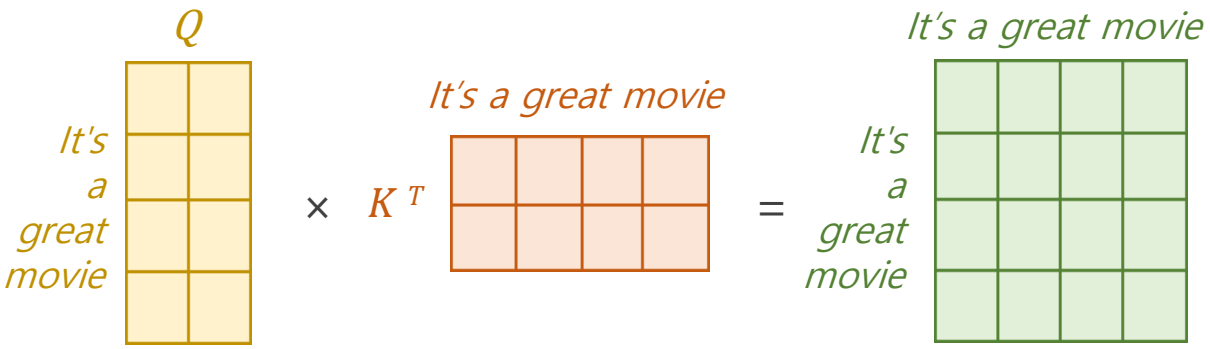
Query에 Key의 Transpose한 행렬을 내적(Dot Product)

Query와 Key가 특정 문장에서 중요한 역할을 하고 있는 경우 Transformer는 이들 사이의 내적(Dot Product) 값을 크게 하는 방향으로 학습

내적 값이 커지면 해당 Query와 Key가 벡터 공간상 가까이에 있을 확률 높음

## 2) Self-Attention 구현

### (2) Self-Attention의 계산




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

    b, l, d = x.size()
    h = self.h

    queries = self.WQ(x).view(b, l, h, d).transpose(1, 2).contiguous().view
    keys = self.WK(x).view(b, l, h, d).transpose(1, 2).contiguous().view(b
    values = self.WV(x).view(b, l, h, d).transpose(1, 2).contiguous().view(

    w_prime = torch.bmm(queries, keys.transpose(1, 2)) / np.sqrt(d)
    w = F.softmax(w_prime, dim=-1)

    out = torch.bmm(w, values).view(b, h, l, d)

    out = out.transpose(1, 2).contiguous().view(b, l, h * d)

    return self.unifyheads(out)
```

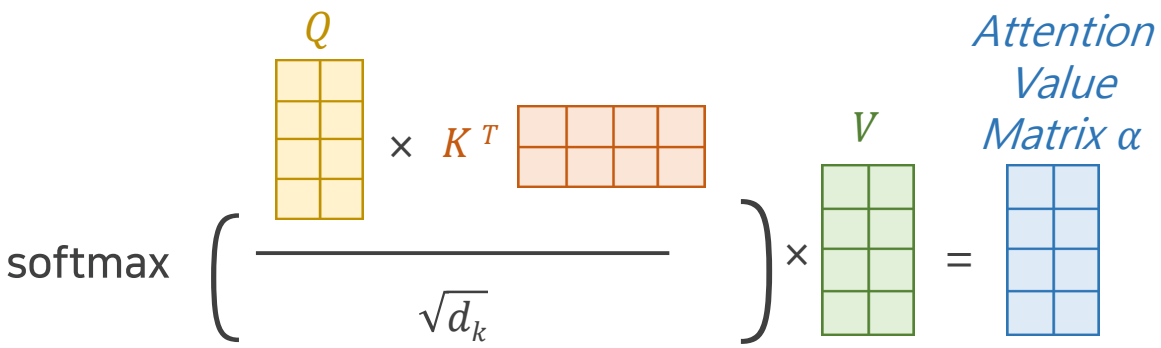
## 2) Self-Attention 구현

### (2) Self-Attention의 계산

Key의 벡터 크기인 64의 root 값인 8로 나눈 후 softmax 함수 적용

Value와 내적(Dot Product)을 곱하여 Attention Value인 Z 계산

- $\sqrt{dk}$ 로 나눈 이유는 Query와 Key의 내적 행렬의 분산을 축소하고 gradient vanishing 발생 방지




$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QKT}{\sqrt{dk}})V$$

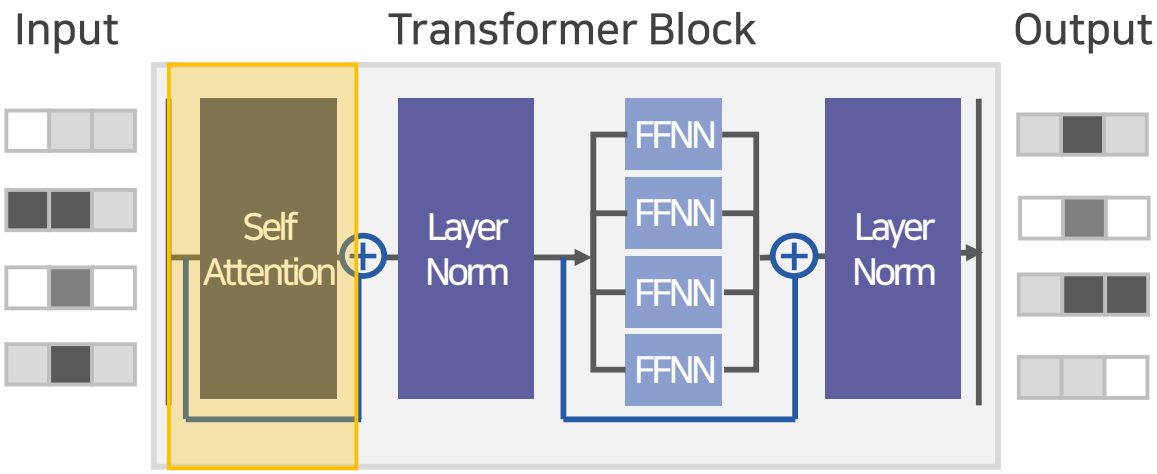
## 2) Self-Attention 구현

### (2) Self-Attention의 계산

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

    b, l, d = x.size()
    h = self.h

    queries = self.WQ(x).view(b, l, h, d).transpose(1, 2).contiguous().view(b
    keys = self.WK(x).view(b, l, h, d).transpose(1, 2).contiguous().view(b *
    values = self.WV(x).view(b, l, h, d).transpose(1, 2).contiguous().view(b

    w_prime = torch.bmm(queries, keys.transpose(1, 2)) / np.sqrt(d)
    w = F.softmax(w_prime, dim=-1)

    out = torch.bmm(w, values).view(b, h, l, d)

    out = out.transpose(1, 2).contiguous().view(b, l, h * d)

    return self.unifyheads(out)
```

## 3) Transformer 구현

### (1) Self-Attention 추가

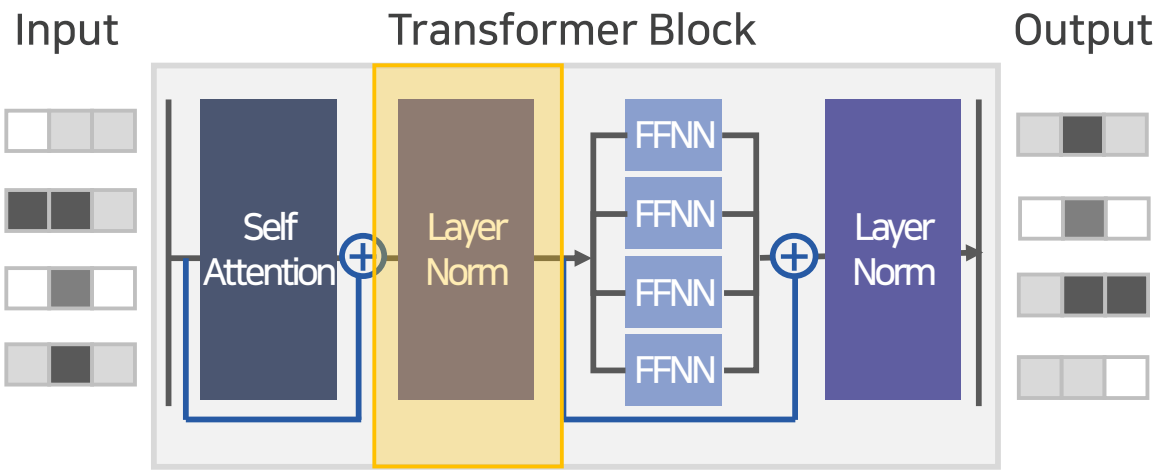



```
class TransformerBlock(nn.Module):
    def __init__(self, d: int, heads: int=8, n_mlp: int=4):
        super().__init__()

        self.attention = SelfAttention(d, heads=heads)
        self.norm1 = nn.LayerNorm(d)
        self.norm2 = nn.LayerNorm(d)

        self.ff = nn.Sequential(
            nn.Linear(d, n_mlp*d),
            nn.ReLU(),
            nn.Linear(n_mlp*d, d)
        )
```

## 3) Transformer 구현

### (1) Self-Attention 추가

```
def forward(self, x: torch.Tensor) -> torch.Tensor:
    x_prime = self.attention(x)
    x = self.norm1(x_prime + x)

    x_prime = self.ff(x)
    return self.norm2(x_prime + x)
```



# 1. Attention과 Transformer 구현

## 3) Transformer 구현

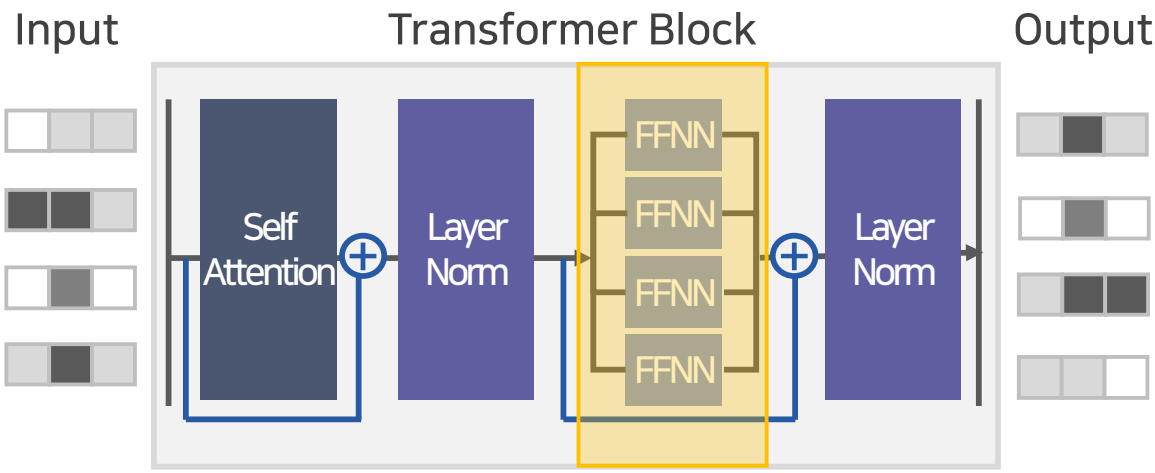
### (1) Self-Attention 추가

```
class TransformerBlock(nn.Module):
    def __init__(self, d: int, heads: int=8, n_mlp: int=4):
        super().__init__()

        self.attention = SelfAttention(d, heads=heads)
        self.norm1 = nn.LayerNorm(d)
        self.norm2 = nn.LayerNorm(d)

        self.ff = nn.Sequential(
            nn.Linear(d, n_mlp*d),
            nn.ReLU(),
            nn.Linear(n_mlp*d, d)
        )
```

```
def forward(self, x: torch.Tensor) -> torch.Tensor:
    x_prime = self.attention(x)
    x = self.norm1(x_prime + x)

    x_prime = self.ff(x)
    return self.norm2(x_prime + x)
```

## 3) Transformer 구현

### (1) Self-Attention 추가



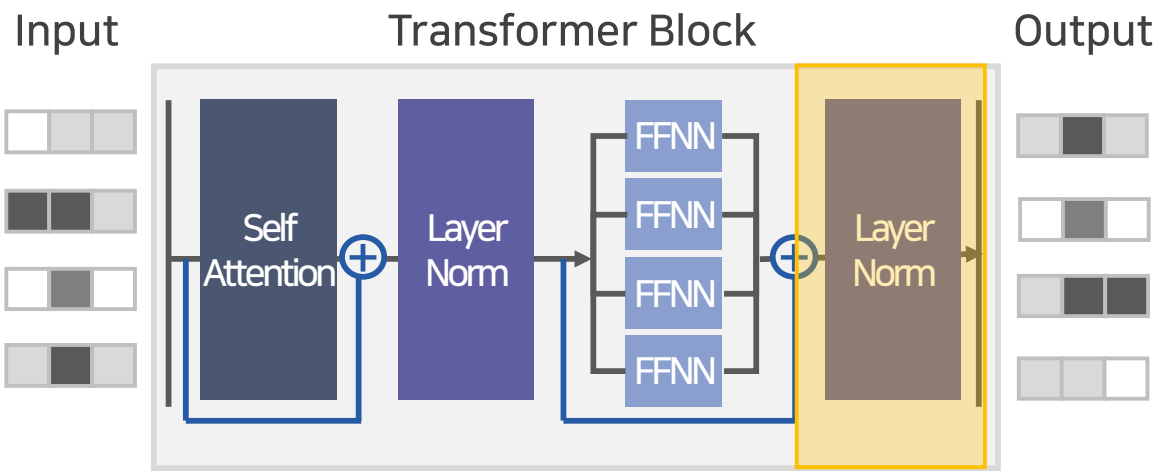

```
class TransformerBlock(nn.Module):
    def __init__(self, d: int, heads: int=8, n_mlp: int=4):
        super().__init__()

        self.attention = SelfAttention(d, heads=heads)
        self.norm1 = nn.LayerNorm(d)
        self.norm2 = nn.LayerNorm(d)

        self.ff = nn.Sequential(
            nn.Linear(d, n_mlp*d),
            nn.ReLU(),
            nn.Linear(n_mlp*d, d)
        )
```

# 1. Attention과 Transformer 구현

## 3) Transformer 구현

### (1) Self-Attention 추가

```
def forward(self, x: torch.Tensor) -> torch.Tensor:
    x_prime = self.attention(x)
    x = self.norm1(x_prime + x)

    x_prime = self.ff(x)
    return self.norm2(x_prime + x)
```



# 1. Attention과 Transformer 구현

## 3) Transformer 구현

### (1) Self-Attention 추가

```
class TransformerBlock(nn.Module):
    def __init__(self, d: int, heads: int=8, n_mlp: int=4):
        super().__init__()

        self.attention = SelfAttention(d, heads=heads)
        self.norm1 = nn.LayerNorm(d)
        self.norm2 = nn.LayerNorm(d)

        self.ff = nn.Sequential(
            nn.Linear(d, n_mlp*d),
            nn.ReLU(),
            nn.Linear(n_mlp*d, d)
        )
```

```
def forward(self, x: torch.Tensor) -> torch.Tensor:
    x_prime = self.attention(x)
    x = self.norm1(x_prime + x)

    x_prime = self.ff(x)
    return self.norm2(x_prime + x)
```

# 2. Transformer 감성분석 적용

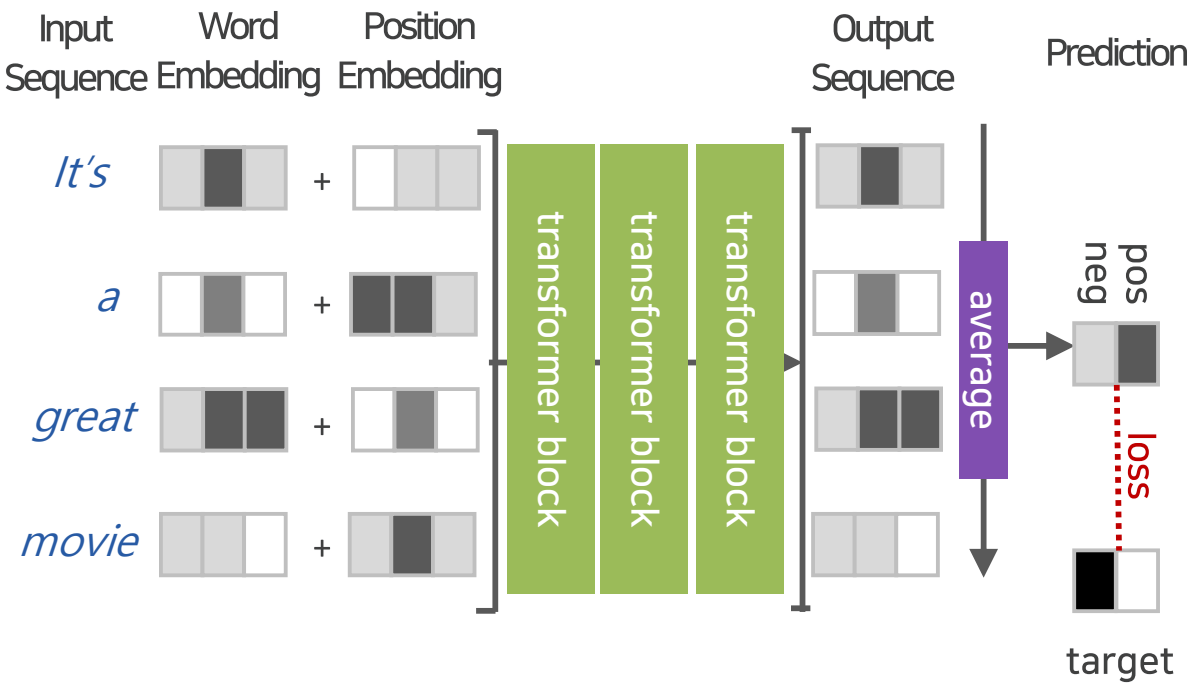
## 1) Transformer 적용 방법

### (1) Transformer 전체 구조

Word embedding과 Positional Embedding 필요

Transformer를 멀티 Layer로 적용

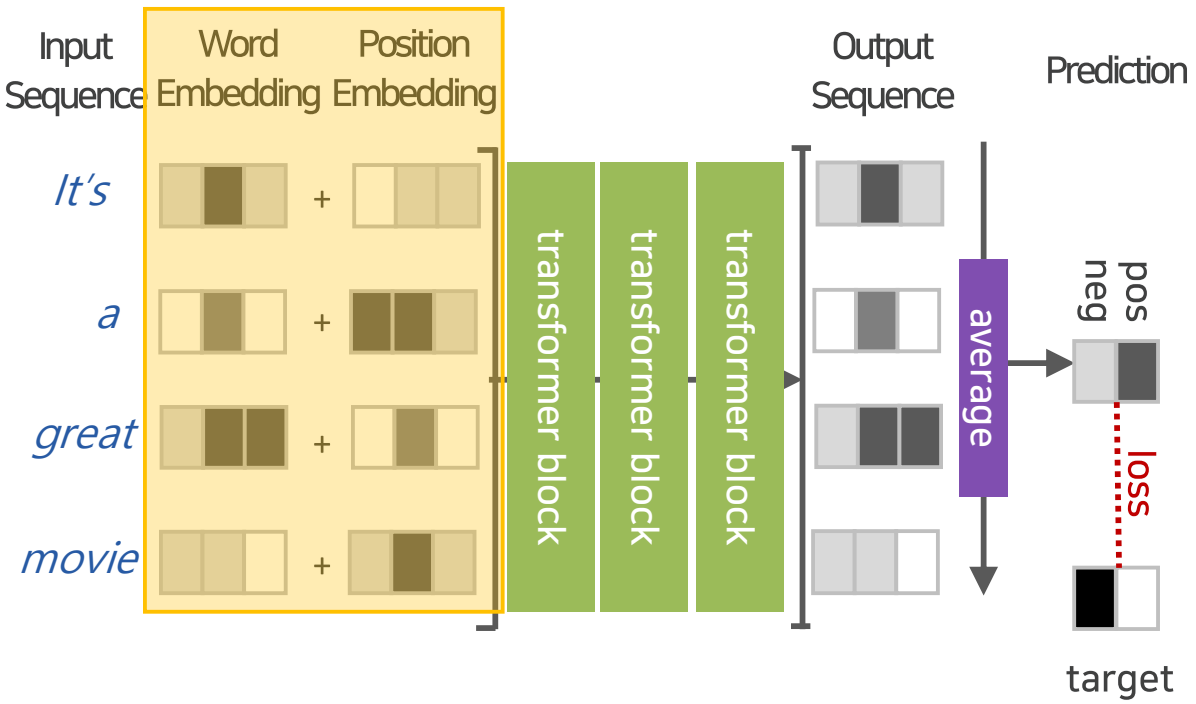
긍정/부정 학습 및 예측을 위한 분류기 추가



# 2. Transformer 감성분석 적용

## 2) Word Embedding과 Positional Embedding Layer

### (1) Word Embedding과 Positional Embedding Layer 추가




### (2) __init__

```
self.token_emb = nn.Embedding(num_tokens, d)
self.pos_emb = nn.Embedding(max_seq_len, d)
```

### (3) __forward__

```
tokens = self.token_emb(x)
positions = self.pos_emb(torch.arange(l).to(self.device)).expand(b, l,
embeddings = tokens + positions
```

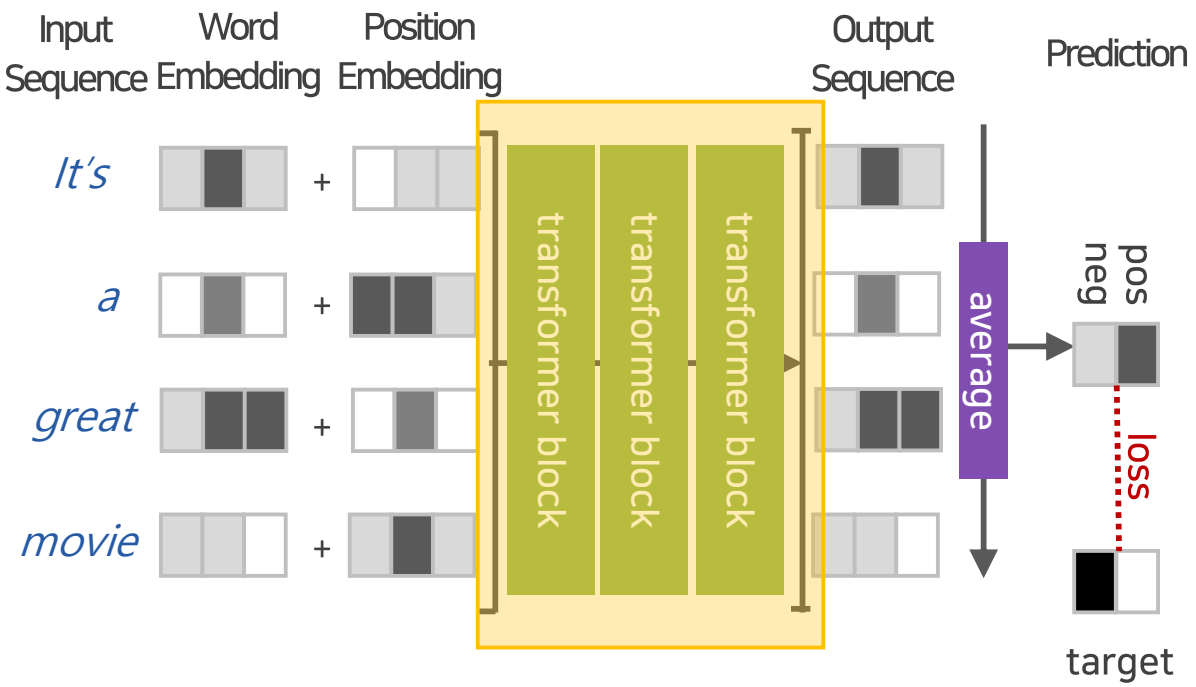
## 2. Transformer 감성분석 적용

### 2) Word Embedding과 Positional Embedding Layer

(4) Positional Embedding을 추가하는 이유?

- RNN + Attention
  - RNN 제거 후 Attention만으로 Transformer 구현
- RNN이 제거되어 토큰의 별도 순서 지정 필요

(5) Transformer Block 추가



# 2. Transformer 감성분석 적용

## 2) Word Embedding과 Positional Embedding Layer

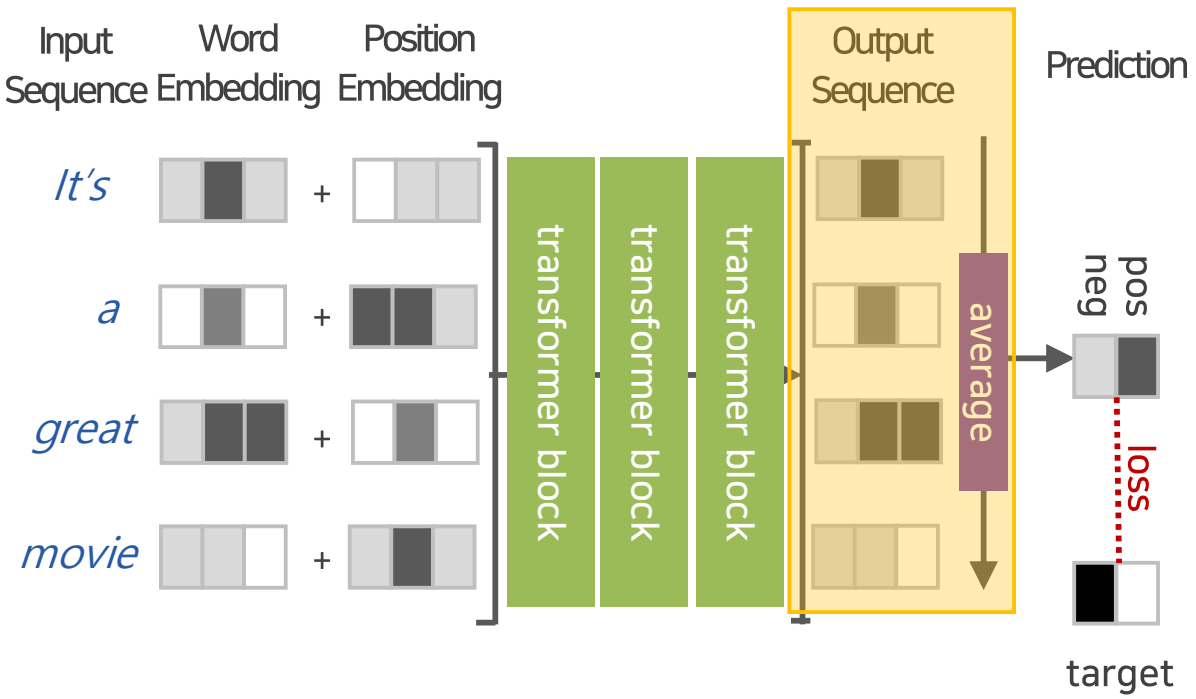
### (6) __init__

```
self.transformer_blocks = nn.Sequential(
    *[TransformerBlock(d=d, heads=heads) for _ in range(depth)]
)
```

### (7) __forward__

```
out = self.transformer_blocks(embeddings)
```

### (8) Classification Layer 추가



# 2. Transformer 감성분석 적용

## 2) Word Embedding과 Positional Embedding Layer

(9) _ _ init_ _

```
self.classification = nn.Linear(d, num_classes)
```

(10) _ _ forward_ _

```
out = out.mean(dim=1)
out = self.classification(out)
```