CCT

정하연

2024 Winter Break

목차

A Table of Contents

논문 정리

➢Escaping the Big Data Paradigm with Compact Transformers

- arXiv, 2021

- Transformer 등장 이후 모델의 크기, 데이터 크기가 계속 커지는 경향이 있었음
- Encoder, Decoder를 계속 쌓을 수록 좋은 성능을 보였기 때문
  - 예를 들어, GPT-1에서 GPT-4로 갈 수록 Encoder를 더 많이 쌓음.
- 본 논문에서는 작은 크기의 학습이 가능한 Compact Transformer를 소개함

논문 정리

➢Main Contributions

- Extending transformer-based research to small data regimes, by introducing ViT-Lite, which can be trained from scratch and achieve high accuracy on datasets such as CIFAR-10.
- Introducing Compact Vision Transformer (CVT) with a new sequence pooling strategy, which pools over out- put tokens and improves performance.
- Introducing Compact Convolutional Transformer (CCT) to increase performance and provide flexibility for input image sizes while also demonstrating that these variants do not depend as much on Positional Embedding compared to the rest.

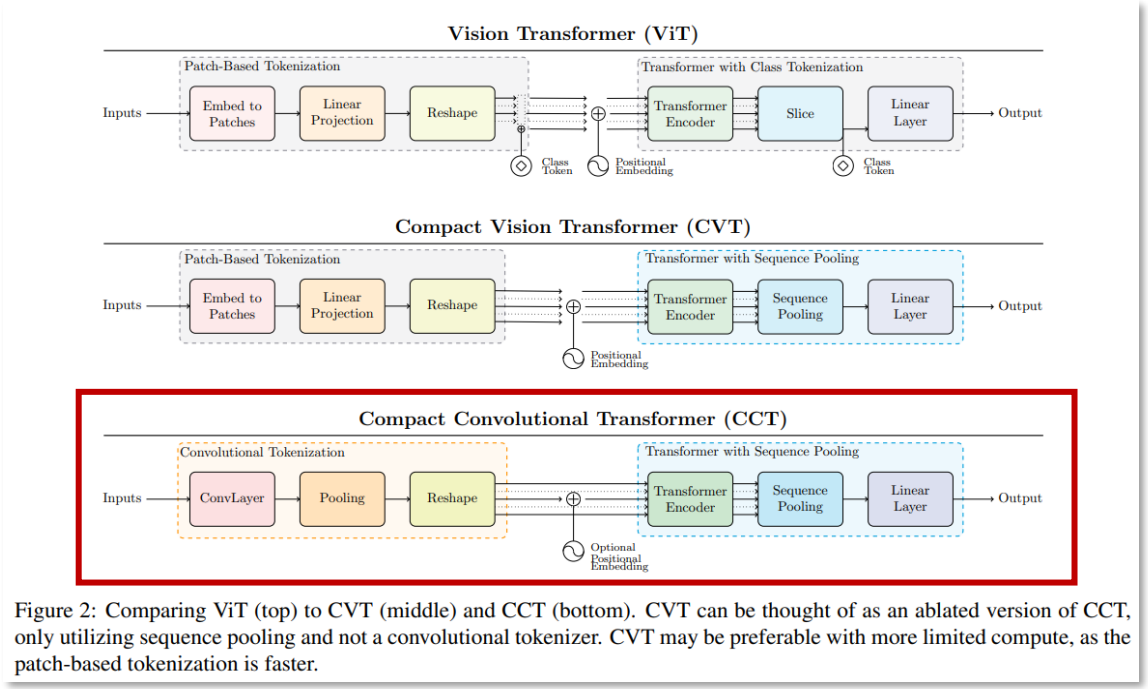


Figure 2: Comparing ViT (top) to CVT (middle) and CCT (bottom). CVT can be thought of as an ablated version of CCT, only utilizing sequence pooling and not a convolutional tokenizer. CVT may be preferable with more limited compute, as the patch-based tokenization is faster.

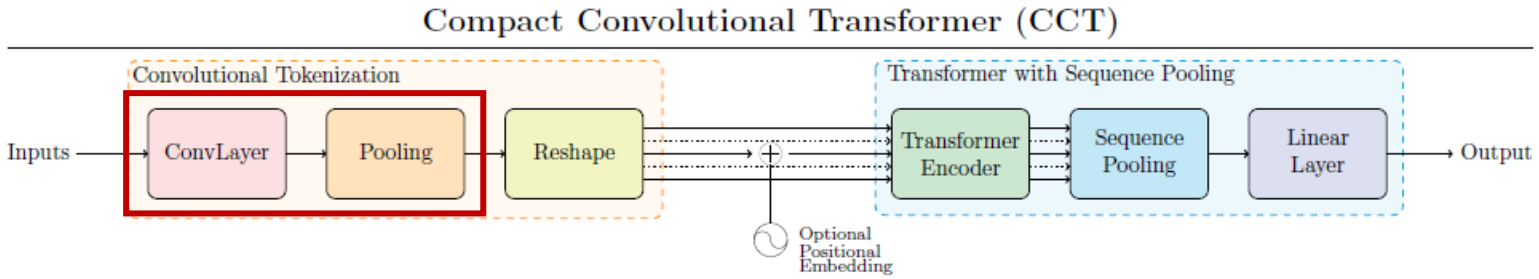
논문 정리

➢Why “Convolutional” Tokenization?

- *“ Transformers lack some of the inductive biases inherent to CNNs, such as translation equivariance and locality, and therefore do not generalize well when trained on insufficient amounts of data.”*
- “ 트랜스포머 모델이 컨볼루션 신경망(CNNs)에 내재된 일부 귀납적 편향(translation equivariance 및 locality와 같은)을 가지고 있지 않기 때문에, 충분하지 않은 양의 데이터로 훈련되었을 때 일반화가 잘 되지 않는다.”
  - Translation equivariance : 이미지 패턴이 어디에 위치해 있는지
  - Locality : 이미지 내 특정 패턴과 근처 픽셀 간의 관계
  - 트랜스포머는 위와 같은 inductive biases을 가지고 있지 않은 대신, self-attention mechanism을 사용하여 입력 시퀀스의 (모든 위치 간의) 관계를 고려함.

1. Convolutional Tokenization

1. ConvLayer가 여러 개면, input-output 맞추기 위해
2. Padding=2로 해야 사이즈가 줄지 않음



In order to introduce an inductive bias into the model, we replace patch and embedding in ViT-Lite and CVT, with a simple convolutional block. This block follows conventional design, which consists of a single convolution, *ReLU* activation, and a max pool. Given an image or feature map $\mathbf{x} \in \mathbb{R}^{H \times W \times C}$:

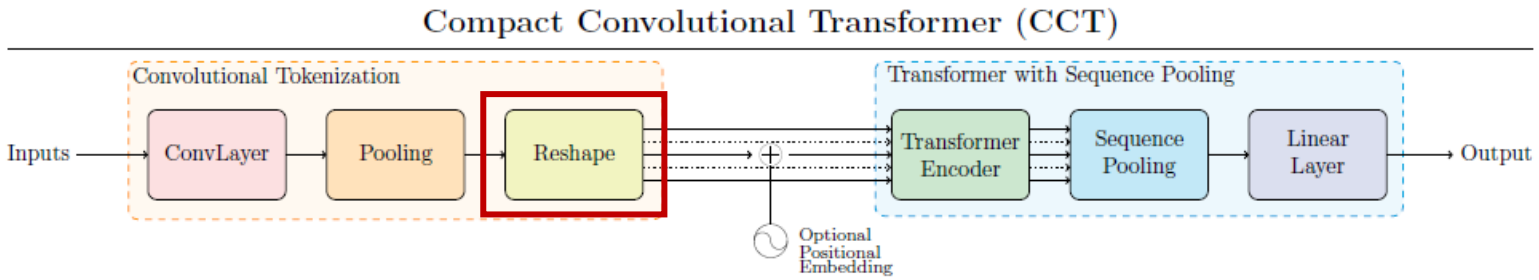
$$\mathbf{x}_0 = \text{MaxPool}(\text{ReLU}(\text{Conv2d}(\mathbf{x}))) \quad (2)$$

Table 6: Tokenizers in each variant.

Model	# Layers	# Convs	Kernel	Stride
ViT-Lite-7/8	7	1	8×8	8×8
ViT-Lite-7/4	7	1	4×4	4×4
CVT-7/8	7	1	8×8	8×8
CVT-7/4	7	1	4×4	4×4
CCT-2/3x2	2	2	3×3	1×1
CCT-7/3x1	7	1	3×3	1×1
CCT-7/7x2	7	2	7×7	2×2

```python
class ConvolutionalTokenizer(nn.Module):
    def __init__(self, cfg):
        super().__init__()

        self.num_conv_layers = cfg.conv_token.num_conv_layers
        self.kernel_size = cfg.conv_token.kernel_size
        self.conv_layers = nn.ModuleList()

        conv_layer = nn.Conv2d(cfg.conv_token.input_channels,
                               cfg.conv_token.output_channels,
                               cfg.conv_token.kernel_size,
                               cfg.conv_token.stride,
                               cfg.conv_token.padding)
        relu = nn.ReLU()
        maxpool = nn.MaxPool2d(kernel_size=self.kernel_size, stride=cfg.conv_token.stride)
        layer = nn.Sequential(conv_layer, relu, maxpool)
        self.conv_layers.append(layer)

        for   in range(self.num_conv_layers - 1):
            conv_layer = nn.Conv2d(cfg.conv_token.output_channels, # not input_channels
                                   cfg.conv_token.output_channels,
                                   cfg.conv_token.kernel_size,
                                   cfg.conv_token.stride,
                                   cfg.conv_token.padding)
            layer = nn.Sequential(conv_layer, relu, maxpool)
            self.conv_layers.append(layer)
```

1. Convolutional Tokenization



In order to introduce an inductive bias into the model, we replace patch and embedding in ViT-Lite and CVT, with a simple convolutional block. This block follows conventional design, which consists of a single convolution, *ReLU* activation, and a max pool. Given an image or feature map $\mathbf{x} \in \mathbb{R}^{H \times W \times C}$:

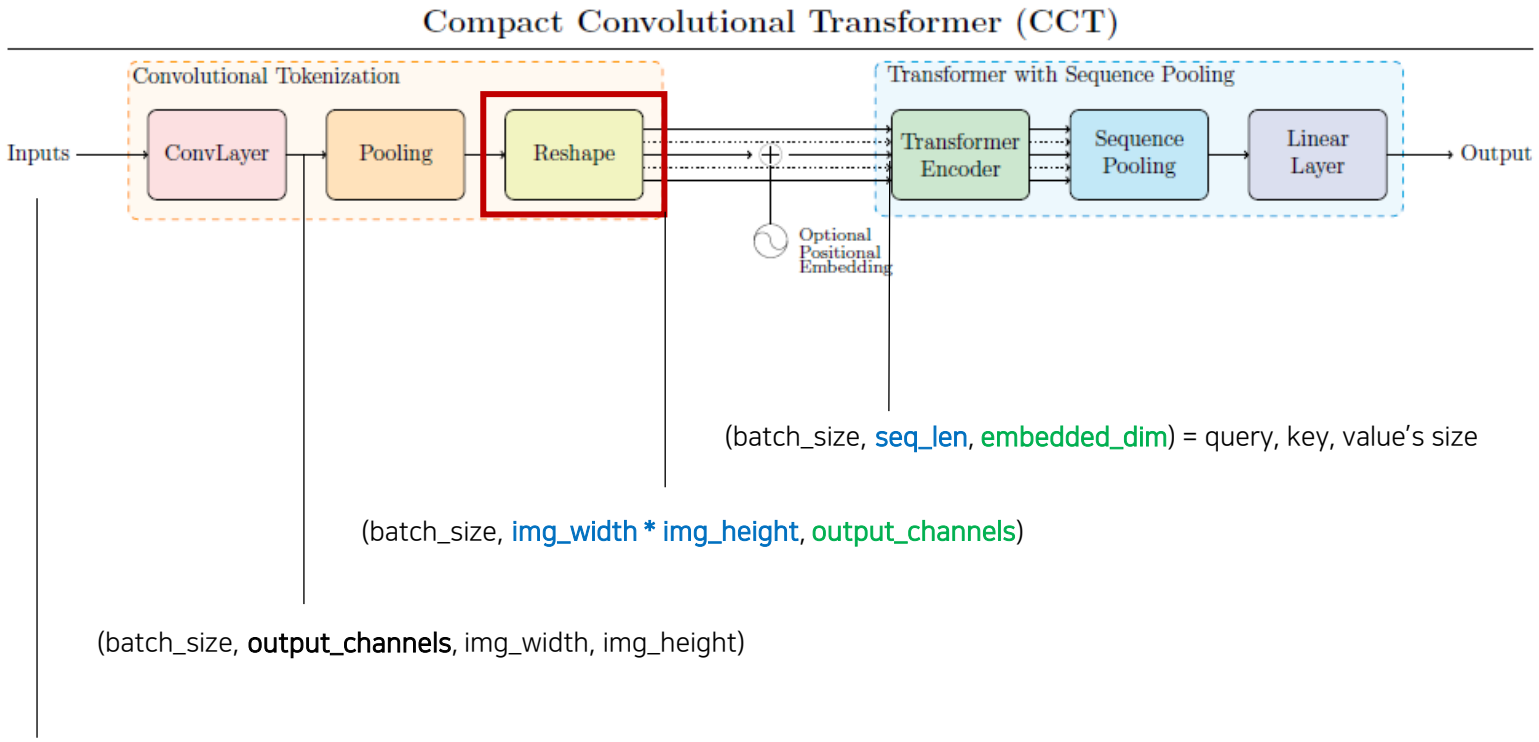
$$\mathbf{x}_0 = \text{MaxPool}(\text{ReLU}(\text{Conv2d}(\mathbf{x}))) \quad (2)$$

Table 6: Tokenizers in each variant.

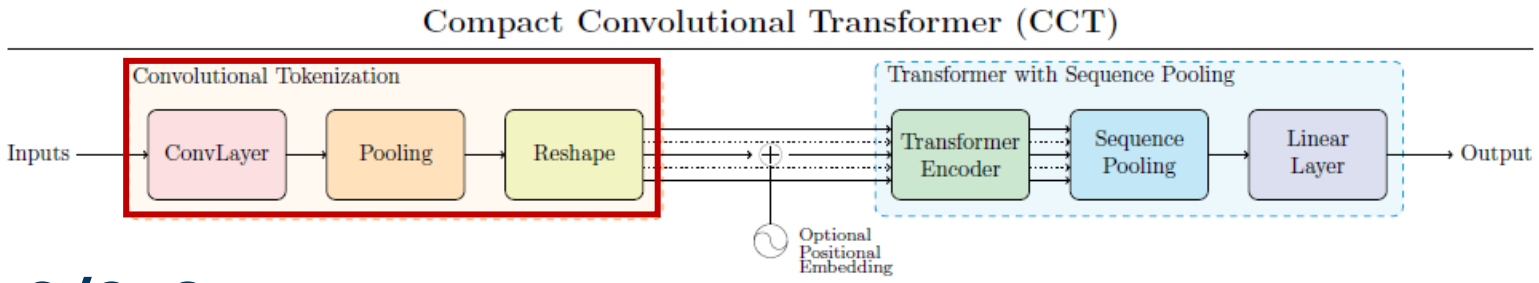
Model	# Layers	# Convs	Kernel	Stride
ViT-Lite-7/8	7	1	8×8	8×8
ViT-Lite-7/4	7	1	4×4	4×4
CVT-7/8	7	1	8×8	8×8
CVT-7/4	7	1	4×4	4×4
CCT-2/3x2	2	2	3×3	1×1
CCT-7/3x1	7	1	3×3	1×1
CCT-7/7x2	7	2	7×7	2×2

```python
def forward(self, x):
    # Input tensor shape: [batch_size, input_channels, height, width] = [1, 3, 32, 32]

    # print(x.shape) # torch.Size([1, 3, 32, 32]), if CCT-2-3x2

    for conv_layer in self.conv_layers:
        x = conv_layer(x)
        # print(x.shape) # torch.Size([1, 128, 32, 32]), if CCT-2-3x2

    # 차원 변경 (Transpose) - (1, 128, 32, 32) -> (1, 32, 32, 128)
    x_transposed = torch.transpose(x, 1, 2)
    x_transposed = torch.transpose(x, 2, -1)

    # Reshape - (1, 32, 32, 128) -> (1, 32 * 32, 128)
    x_reshaped = x_transposed.reshape(x.size(0), -1, x.size(1))
    # print(x_reshaped.shape) # (1, 1024, 128)

    # Output tensor shape: [batch_size, flattend, output_channel = d_model = embed_dim]

    return x_reshaped
```

1. Convolutional Tokenization



1. Convolutional Tokenization



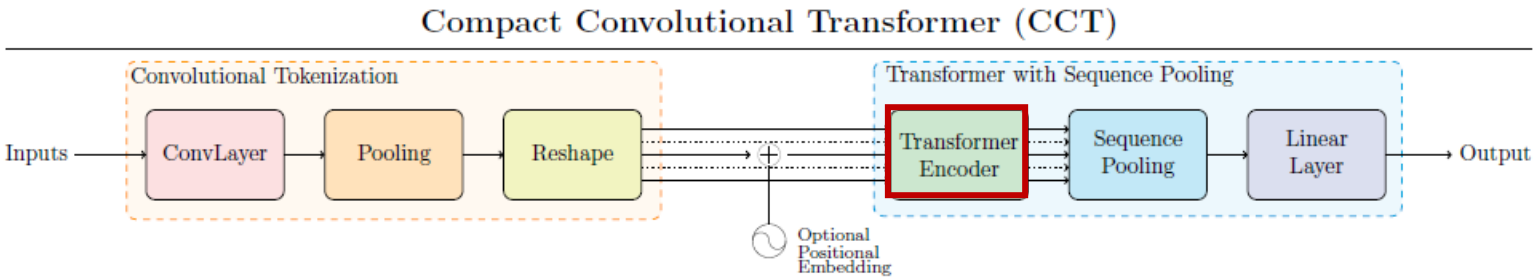
➢예시 : CCT-2/3x2

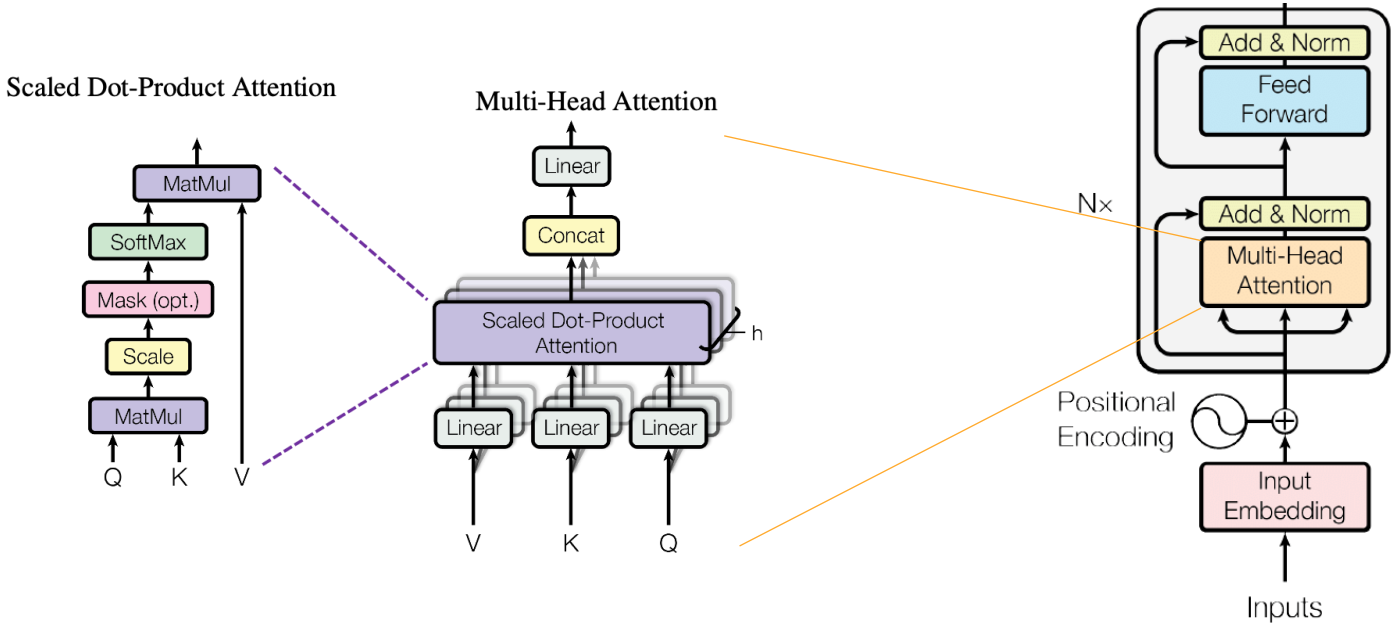
configs > {} CCT-2-3x2.json > ...

```
{
    "conv_token": {
        "num_conv_layers": 2,
        "kernel_size": 3,
        "input_channels": 3,
        "output_channels": 64,
        "stride": 1,
        "num_classes": 10
    },
    "encoder": {
        "d_model": 128,
        "ffn_hidden": 1024,
        "num_heads": 2,
        "drop_prob": 0.1,
        "n_layers": 2
    }
}
```

```
CCT(
  (Convolutional_Tokenization): ConvolutionalTokenizer(
    (conv_layers): ModuleList(
      (0): Sequential(
        (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1))
        (1): ReLU()
        (2): MaxPool2d(kernel_size=3, stride=1, padding=0, dilation=1, ceil_mode=False)
      )
      (1): Sequential(
        (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1))
        (1): ReLU()
        (2): MaxPool2d(kernel_size=3, stride=1, padding=0, dilation=1, ceil_mode=False)
      )
    )
  )
  (Transformer_With_SeqPool): TransformerWithSeqPool(
    (encoder): Encoder(
```

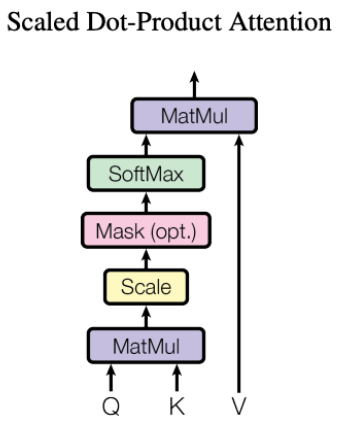
2. Transformer Encoder

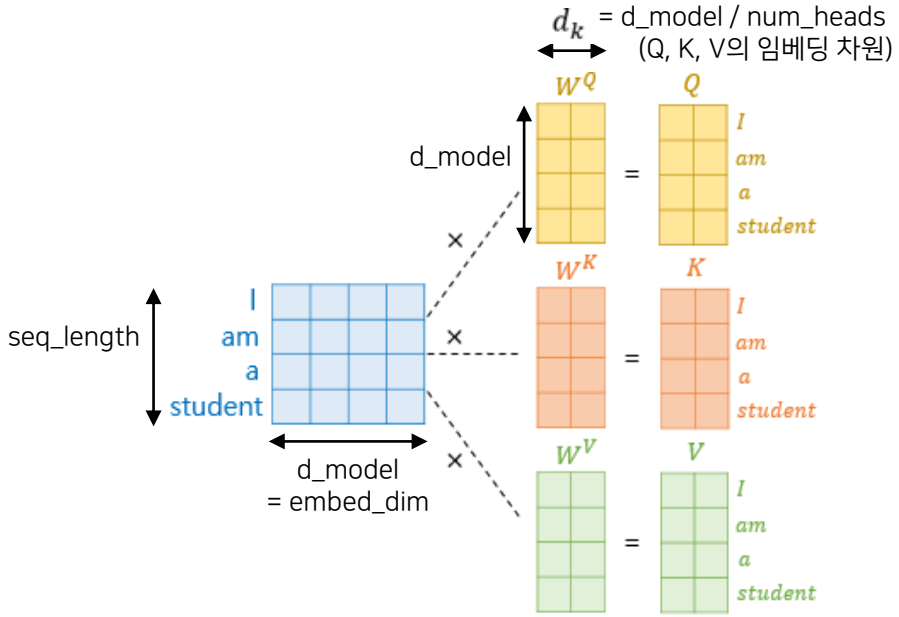




2. Transformer Encoder

➢(Scaled) Self dot-product attention





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

$$= softmax\left(\frac{Q \times K^T}{\sqrt{d_k}}\right) \times V = \text{Attention Value Matrix } a$$

$$= softmax\left(\frac{\begin{matrix} & I & am & a & student \\ I & & & & \\ am & & & & \\ a & & & & \\ student & & & & \end{matrix}}{\sqrt{d_k}}\right) \times V = \text{Attention Value Matrix } a$$

* Why Masked Attention?

: time step 마다 지난 출력과 예측한 현재 토큰만 사용하여 토큰을 생성하기 위해.

다음에 나오는 정보를 참고하지 못하도록 하기 위해. (학습 과정에서 뒤 정답을 알게 됨)

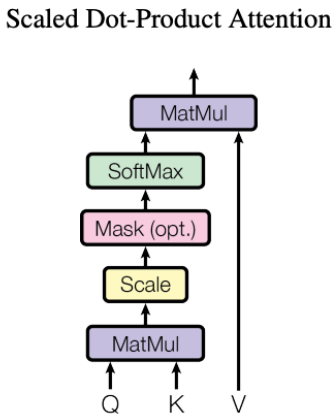
새로운 데이터가 들어왔을 때 직접 생성 못하고, 기존에 알고 있던 결과를 내는 것을 방지하기 위해.

* Q*K^T : 어텐션 스코어 행렬

* Softmax : 어텐션 분포 구하기

2. Transformer Encoder

➢(Scaled) Self dot-product attention

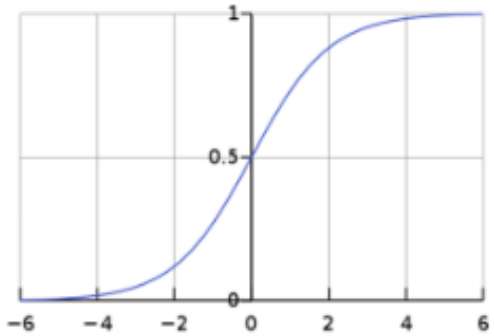



```python
def scaled_dot_product_attention(query, key, value, mask=None):
    # query, key, value: (n_batch, seq_len, d_k)
    # mask: (n_batch, seq_len, seq_len)

    d_k = query.size(-1) # d_k = d_model / num_heads
    attention_score = torch.matmul(query, key.transpose(-2, -1)) # Q x K^T, (n_batch, seq_len, seq_len)
    attention_score = attention_score / math.sqrt(d_k)

    # Masking (optional)
    if mask is not None:
        attention_score = attention_score.masked_fill(mask==0, float('-inf'))
        # mask tensor는 0 or 1로 이루어져 있으며, 0인 위치는 어텐션 스코어에 음의 무한대 값을 채워주어 해당 위치의 가중치를 0으로 만듦
        # softmax 함수 적용 시 exp^(-inf) = 0, 따라서 attention weights가 모두 0이 됨
        # 이렇게 함으로써 마스크가 적용된 위치의 정보를 attention weights에 반영하지 않도록 할 수 있음

    weights = F.softmax(attention_score, dim=-1) # (n_batch, seq_len, seq_len)

    # bmm : 배치 차원을 고려한 행렬-행렬 곱셈
    # Q. matmul vs. mm vs. bmm (https://velog.io/@regista/torch.dot-torch.matmul-torch.mm-torch.bmm)
    result = torch.bmm(weights, value) # (n_batch, seq_len, d_k)

    return result
```

2. Transformer Encoder

➢(Scaled) Self dot-product attention

▼ Why ***Scaled*** Self Dot-Product Attention?



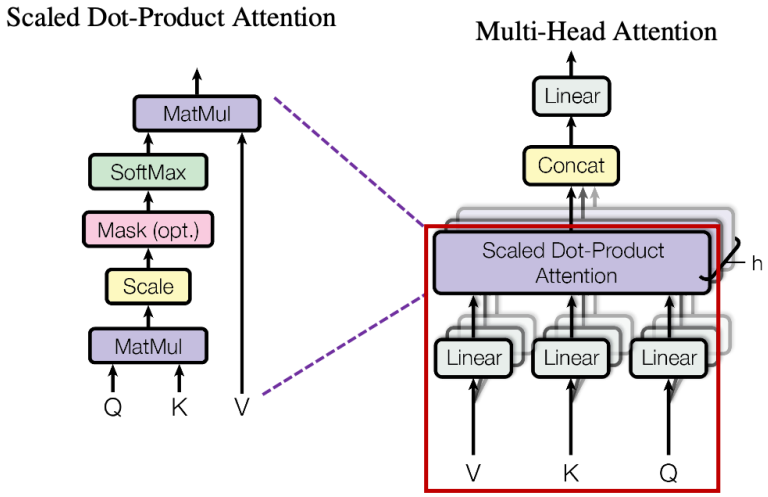
Softmax 함수

- **Gradient Scaling 문제 해결**: 만약 d_k가 큰 경우, QK^T의 값은 더 큰 수가 되어 Softmax 함수를 거치면 매우 작은 기울기를 얻게 됩니다. 이는 모델 학습 시에 Gradient Vanishing(기울기 소멸) 문제와 관련이 있습니다. 따라서 sqrt(d_k)로 나누는 것은 dot-product의 결과를 작은 값으로 스케일링하여 기울기의 크기를 조절해주는 역할을 합니다.
  - Gradient Scaling: explosion or vanishing
- **안정성과 효율성 강화**: Softmax 함수는 지수 함수를 사용하기 때문에 큰 수가 입력으로 들어가면 수치적으로 불안정해질 수 있습니다. 하지만 sqrt(d_k)로 나누면서 값이 조절되어 수치적 안정성이 향상됩니다.

$$y_k = \frac{\exp(a_k)}{\sum_{i=1}^{n} \exp(a_i)}$$

2. Transformer Encoder

➢(Masked) Multi-Head Attention



$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O$$
$$\text{where } \text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

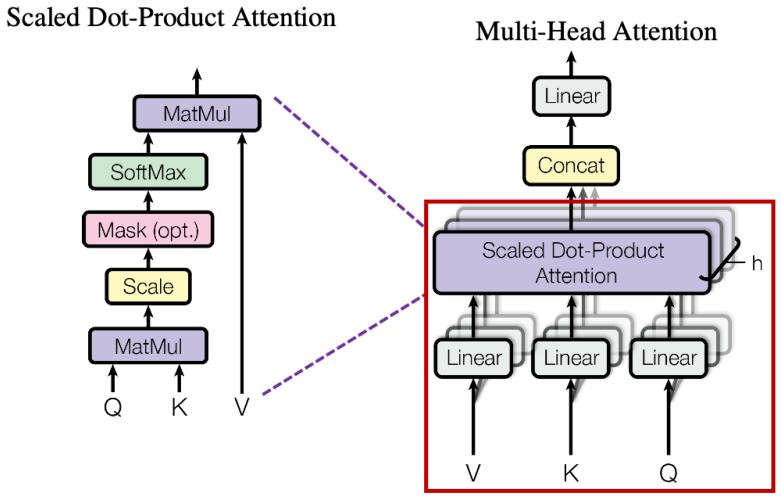
* 여러 개의 서브 공간(subspace)로 나누어 처리하는 방법

* 각 subspace가 head라고 불림

```python
class AttentionHead(nn.Module):
    def __init__(self, embed_dim, head_dim):
        '''
        nn.Linear(input_feature_dimension, output_feature_dimension)
        # Linear Transformation (선형 변환: 차원 변환)
        '''
        super(AttentionHead, self).__init__()
        self.Q = nn.Linear(embed_dim, head_dim)
        self.K = nn.Linear(embed_dim, head_dim)
        self.V = nn.Linear(embed_dim, head_dim)

    # Mask
    def forward(self, query, key, value, mask=None):
        # query = torch.transpose(query, 0, 1)
        # value = torch.transpose(value, 0, 1)
        # key = torch.transpose(key, 0, 1)

        # print("query size : ", query.size())

        attention_output = scaled_dot_product_attention(self.Q(query), self.K(key), self.V(value), mask=mask)
        return attention_output
```

2. Transformer Encoder

➢(Masked) Multi-Head Attention



$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where } \text{head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

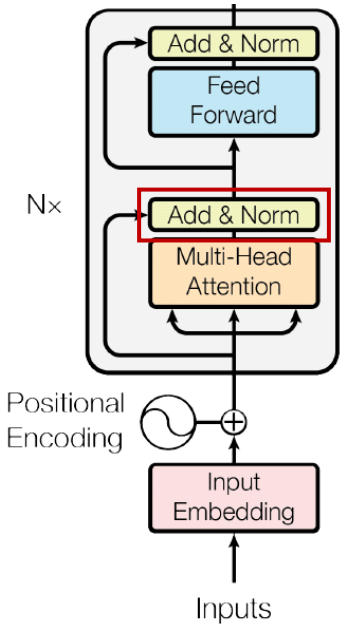
* 여러 개의 서브 공간(subspace)로 나누어 처리하는 방법

* 각 subspace가 head라고 불림

```python
class MultiHeadAttention(nn.Module):
    def __init__(self, cfg):
        super(MultiHeadAttention, self).__init__()

        self.embed_dim = cfg.encoder.d_model # CCT-7/3x1 (256), embed_dim = d_model
        self.num_heads = cfg.encoder.num_heads # CCT-7/3x1 (4)
        self.head_dim = self.embed_dim // self.num_heads # d_k = d_model // num_heads

        self.heads = nn.ModuleList(
            [AttentionHead(self.embed_dim, self.head_dim) for _ in range(self.num_heads)]
        )
        self.fc_layer = nn.Linear(self.embed_dim, self.embed_dim) # modified

    # Mask
    def forward(self, hidden_state, mask=None):

        outs = torch.cat([h(hidden_state, hidden_state, hidden_state, mask=mask)
                                                    for h in self.heads], dim=-1)
        outs = self.fc_layer(outs)
        return outs
```

2. Transformer Encoder

➢Add & Norm



* Add (Residual Connection)

- 입력값과 출력값을 연결.
- 일부 layer를 건너뛰어 데이터가 신경망 구조의 후반부에 도달하는 또 다른 경로를 제공함으로써 Gradient 소실 or 폭주 문제를 완화함.
- Output = Input + Sublayer(Input) *Sublayer = Multi-Head Attention

* Norm (Layer Normalization)

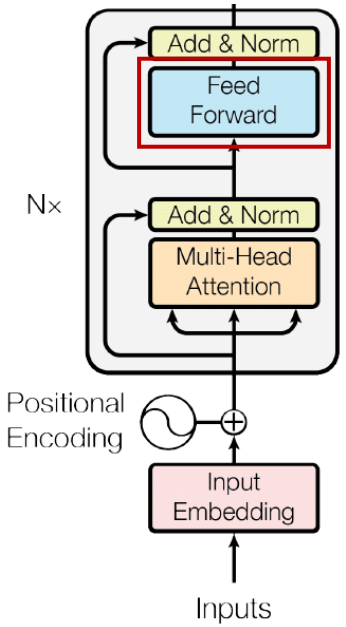
- Residual Connection 결과에 대한 정규화 진행
- layer 간의 안정성 향상, 학습 가속화 (기울기 소실 or 폭주 완화)
- x: 입력, μ: 평균, σ: 표준편차

$$\mathrm{LayerNorm}(x) = \gamma\left(\frac{x-\mu}{\sqrt{\sigma^2+\epsilon}}\right) + \beta$$

```python
class LayerNorm(nn.Module):
    def __init__(self, cfg, eps=1e-12):
        super(LayerNorm, self).__init__()
        self.gamma = nn.Parameter(torch.ones(cfg.encoder.d_model))
        self.beta = nn.Parameter(torch.zeros(cfg.encoder.d_model))
        self.eps = eps

    def forward(self, x):
        mean = x.mean(-1, keepdim=True)
        var = x.var(-1, unbiased=False, keepdim=True)
        # '-1' means last dimension.

        out = (x - mean) / torch.sqrt(var + self.eps)
        out = self.gamma * out + self.beta
        return out
```

2. Transformer Encoder

➢Feed Forward



$$FFN(x) = \underbrace{\max(0, \overbrace{xW_1 + b_1}^{\text{linear transformation}})}_{\text{ReLU}} W_2 + b_2 \quad \text{(linear transformation)}$$

- Multi Head Attention을 수행하여 얻은 각 Head는 관점에 따라 정보가 치우쳐져 있음.
- Position-wise FFN은 각 Head가 만들어낸 Attention을 치우치지 않게 균등하게 섞는 역할을 한다.

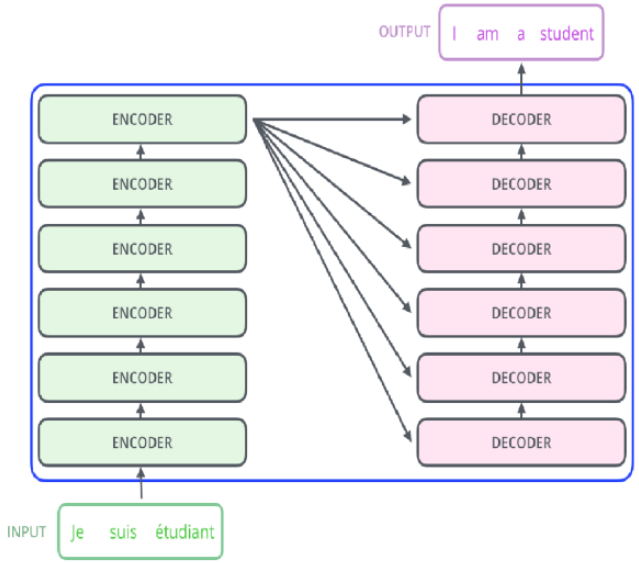
```python
class PositionWiseFeedForward(nn.Module):
    def __init__(self, cfg): # drop_prob=0.1 (default)
        super(PositionWiseFeedForward, self).__init__()
        self.fc_layer1 = nn.Linear(cfg.encoder.d_model, cfg.encoder.ffn_hidden)
        self.fc_layer2 = nn.Linear(cfg.encoder.ffn_hidden, cfg.encoder.d_model)
        self.gelu = nn.GELU()
        self.dropout = nn.Dropout(p=cfg.encoder.drop_prob)

    def forward(self, x):
        out = self.fc_layer1(x)
        out = self.gelu(out)
        out = self.dropout(out)
        out = self.fc_layer2(out)
        return out
```

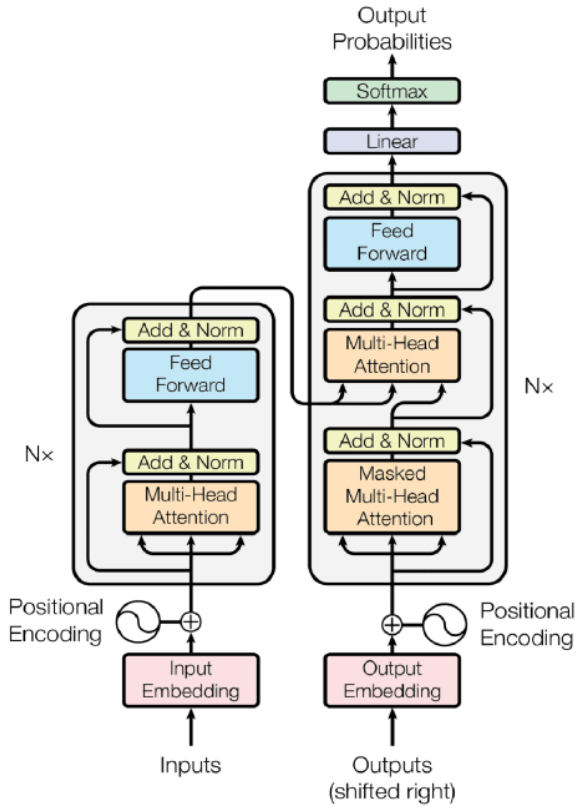
2. Transformer Encoder

➢Encoder Layers

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

    def __init__(self, cfg):
        super().__init__()
        self.attention = MultiHeadAttention(cfg)
        self.norm1 = LayerNorm(cfg)
        self.dropout1 = nn.Dropout(p=cfg.encoder.drop_prob)

        self.ffn = PositionWiseFeedForward(cfg)
        self.norm2 = LayerNorm(cfg)
        self.dropout2 = nn.Dropout(p=cfg.encoder.drop_prob)


    def forward(self, x, src_mask):
        # 1. compute self attention
        x_copy = x
        x = self.attention(x, mask=src_mask)

        # 2. add_and_norm
        x = self.dropout1(x)
        x = self.norm1(x + x_copy)

        # 3. PositionWiseFeedForward
        x_copy = x
        x = self.ffn(x)

        # 4. add and norm
        x = self.dropout2(x)
        x = self.norm2(x + x_copy)
        return x
```

```python
class Encoder(nn.Module):
    def __init__(self, cfg):
        super().__init__()

        self.layers = nn.ModuleList([EncoderLayer(cfg)
                                     for _ in range(cfg.encoder.n_layers)])

    def forward(self, x, src_mask):
        for layer in self.layers:
            x = layer(x, src_mask)

        return x
```





https://www.linkedin.com/pulse/transformer-model-neural-network-which-uses-attention-tejas-bankar/

2. Transformer Encoder

➢예시 : CCT-2/3x2

Table 5: Transformer backbones in each variant.

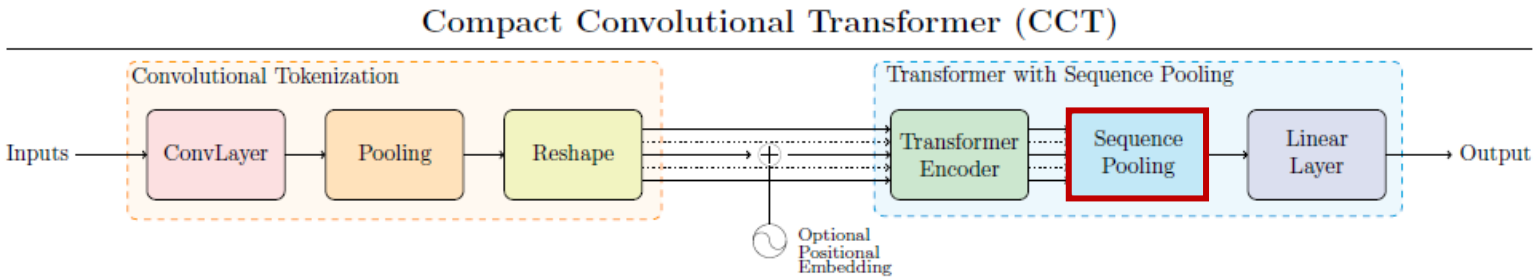
Model	# Layers	# Heads	Ratio	Dim
ViT-Lite-6	6	4	2	256
ViT-Lite-7	7	4	2	256
CVT-6	6	4	2	256
CVT-7	7	4	2	256
CCT-2	2	2	1	128
CCT-4	4	2	1	128
CCT-6	6	4	2	256
CCT-7	7	4	2	256
CCT-14	14	6	3	384

```
(Transformer_With_SeqPool): TransformerWithSeqPool(
  (encoder): Encoder(
    (layers): ModuleList(
      (0): EncoderLayer(
        (attention): MultiHeadAttention(
          (heads): ModuleList(
            (0): AttentionHead(
              (Q): Linear(in_features=128, out_features=64, bias=True)
              (K): Linear(in_features=128, out_features=64, bias=True)
              (V): Linear(in_features=128, out_features=64, bias=True)
            )
            (1): AttentionHead(
              (Q): Linear(in_features=128, out_features=64, bias=True)
              (K): Linear(in_features=128, out_features=64, bias=True)
              (V): Linear(in_features=128, out_features=64, bias=True)
            )
          )
          (fc_layer): Linear(in_features=128, out_features=128, bias=True)
        )
        (norm1): LayerNorm()
        (dropout1): Dropout(p=0.1, inplace=False)
        (ffn): PositionWiseFeedForward(
          (fc_layer1): Linear(in_features=128, out_features=1024, bias=True)
          (fc_layer2): Linear(in_features=1024, out_features=128, bias=True)
          (gelu): GELU(approximate='none')
          (dropout): Dropout(p=0.1, inplace=False)
        )
        (norm2): LayerNorm()
        (dropout2): Dropout(p=0.1, inplace=False)
      )
```

```
      (1): EncoderLayer(
        (attention): MultiHeadAttention(
          (heads): ModuleList(
            (0): AttentionHead(
              (Q): Linear(in_features=128, out_features=64, bias=True)
              (K): Linear(in_features=128, out_features=64, bias=True)
              (V): Linear(in_features=128, out_features=64, bias=True)
            )
            (1): AttentionHead(
              (Q): Linear(in_features=128, out_features=64, bias=True)
              (K): Linear(in_features=128, out_features=64, bias=True)
              (V): Linear(in_features=128, out_features=64, bias=True)
            )
          )
          (fc_layer): Linear(in_features=128, out_features=128, bias=True)
        )
        (norm1): LayerNorm()
        (dropout1): Dropout(p=0.1, inplace=False)
        (ffn): PositionWiseFeedForward(
          (fc_layer1): Linear(in_features=128, out_features=1024, bias=True)
          (fc_layer2): Linear(in_features=1024, out_features=128, bias=True)
          (gelu): GELU(approximate='none')
          (dropout): Dropout(p=0.1, inplace=False)
        )
        (norm2): LayerNorm()
        (dropout2): Dropout(p=0.1, inplace=False)
      )
    )
  )
```

3. SeqPool

➢ Sequence Pooling



forwarded. This operation consists of mapping the output sequence using the transformation $T : \mathbb{R}^{b\times n\times d} \mapsto \mathbb{R}^{b\times d}$. Given:

$$\mathbf{x}_L = \mathrm{f}(\mathbf{x}_0) \in \mathbb{R}^{b\times n\times d}$$

where $\mathbf{x}_L$ is the output of an L layer transformer encoder f, b is batch size, n is sequence length, and d is the total embedding dimension. $\mathbf{x}_L$ is fed to a linear layer $\mathrm{g}(\mathbf{x}_L) \in \mathbb{R}^{d\times 1}$, and softmax activation is applied to the output:

$$\mathbf{x}'_L = \mathrm{softmax}\left(\mathrm{g}(\mathbf{x}_L)^T\right) \in \mathbb{R}^{b\times 1\times n}$$

This generates an importance weighting for each input token, which is applied as follows:

$$\mathbf{z} = \mathbf{x}'_L\mathbf{x}_L = \mathrm{softmax}\left(\mathrm{g}(\mathbf{x}_L)^T\right) \times \mathbf{x}_L \in \mathbb{R}^{b\times 1\times d} \quad (1)$$

By flattening, the output $z \in \mathbb{R}^{b\times d}$ is produced. This output can then be sent through a classifier.

SeqPool allows our network to weigh the sequential embeddings of the latent space produced by the transformer encoder and correlate data across the input data. This can

```python
class SeqPool(nn.Module):
    def __init__(self, cfg):
        super().__init__()
        self.transformer_encoder = Encoder(cfg)
        self.linear = nn.Linear(cfg.encoder.d_model // cfg.encoder.num_heads, 1)

    def forward(self, x):
        # x 차원 가정 : (batch_size, sequence_length, embedding_dim)
        x_L = self.transformer_encoder(x)  # x_L : (batch_size, sequence_length, embedding_dim)

        # linear layer & softmax activation
        x_L_linear = self.linear(x_L)  # x_L_linear : (batch_size, sequence_length, 1)
        x_L_softmax = F.softmax(x_L_linear.transpose(1, 2), dim=-1)  # sequence_length 차원에 softmax

        # Generate importance weighting
        z = torch.matmul(x_L_softmax, x_L)  # z : (batch_size, 1, embedding_dim)
        # squeeze : 텐서의 차원 중 사이즈가 1인 차원을 제거
        z = z.squeeze(1)  # Flatten the output to (batch_size, embedding_dim)

        return z
```

3. SeqPool

➢ Sequence Pooling

forwarded. This operation consists of mapping the output sequence using the transformation $T : \mathbb{R}^{b\times n\times d} \mapsto \mathbb{R}^{b\times d}$. Given:

$$\mathbf{x}_L = \mathrm{f}(\mathbf{x}_0) \in \mathbb{R}^{b\times n\times d}$$

where $\mathbf{x}_L$ is the output of an L layer transformer encoder f, b is batch size, n is sequence length, and d is the total embedding dimension. $\mathbf{x}_L$ is fed to a linear layer $\mathrm{g}(\mathbf{x}_L) \in \mathbb{R}^{d\times 1}$, and softmax activation is applied to the output:

$$\mathbf{x}'_L = \mathrm{softmax}\left(\mathrm{g}(\mathbf{x}_L)^T\right) \in \mathbb{R}^{b\times 1\times n}$$

This generates an importance weighting for each input token, which is applied as follows:

$$\mathbf{z} = \mathbf{x}'_L\mathbf{x}_L = \mathrm{softmax}\left(\mathrm{g}(\mathbf{x}_L)^T\right) \times \mathbf{x}_L \in \mathbb{R}^{b\times 1\times d} \quad (1)$$

By flattening, the output $z \in \mathbb{R}^{b\times d}$ is produced. This output can then be sent through a classifier.

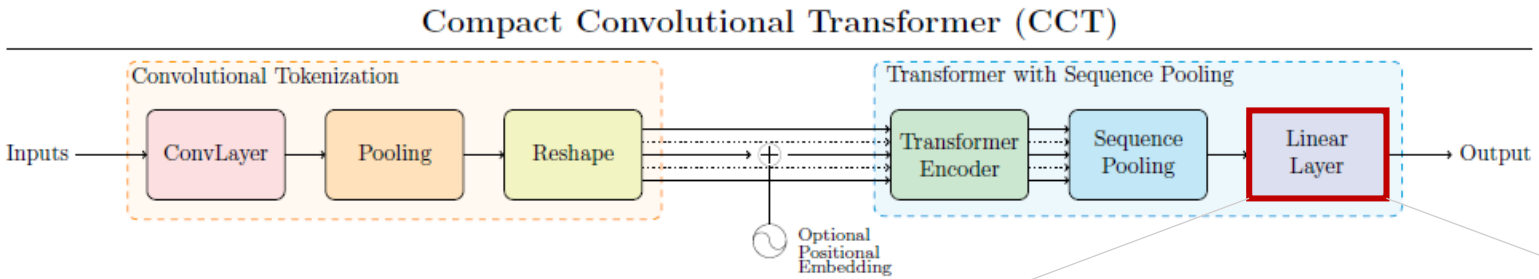
SeqPool allows our network to weigh the sequential embeddings of the latent space produced by the transformer encoder and correlate data across the input data. This can

```
(seq_pool): SeqPool(
  (transformer_encoder): Encoder(
    (layers): ModuleList(
      (0): EncoderLayer(
        (attention): MultiHeadAttention(
          (heads): ModuleList(
            (0): AttentionHead(
              (Q): Linear(in_features=128, out_features=64, bias=True)
              (K): Linear(in_features=128, out_features=64, bias=True)
              (V): Linear(in_features=128, out_features=64, bias=True)
            )
            (1): AttentionHead(
              (Q): Linear(in_features=128, out_features=64, bias=True)
              (K): Linear(in_features=128, out_features=64, bias=True)
              (V): Linear(in_features=128, out_features=64, bias=True)
            )
          )
          (fc_layer): Linear(in_features=128, out_features=128, bias=True)
        )
        (norm1): LayerNorm()
        (dropout1): Dropout(p=0.1, inplace=False)
        (ffn): PositionWiseFeedForward(
          (fc_layer1): Linear(in_features=128, out_features=1024, bias=True)
          (fc_layer2): Linear(in_features=1024, out_features=128, bias=True)
          (gelu): GELU(approximate='none')
          (dropout): Dropout(p=0.1, inplace=False)
        )
        (norm2): LayerNorm()
        (dropout2): Dropout(p=0.1, inplace=False)
      )
```

```
      (1): EncoderLayer(
        (attention): MultiHeadAttention(
          (heads): ModuleList(
            (0): AttentionHead(
              (Q): Linear(in_features=128, out_features=64, bias=True)
              (K): Linear(in_features=128, out_features=64, bias=True)
              (V): Linear(in_features=128, out_features=64, bias=True)
            )
            (1): AttentionHead(
              (Q): Linear(in_features=128, out_features=64, bias=True)
              (K): Linear(in_features=128, out_features=64, bias=True)
              (V): Linear(in_features=128, out_features=64, bias=True)
            )
          )
          (fc_layer): Linear(in_features=128, out_features=128, bias=True)
        )
        (norm1): LayerNorm()
        (dropout1): Dropout(p=0.1, inplace=False)
        (ffn): PositionWiseFeedForward(
          (fc_layer1): Linear(in_features=128, out_features=1024, bias=True)
          (fc_layer2): Linear(in_features=1024, out_features=128, bias=True)
          (gelu): GELU(approximate='none')
          (dropout): Dropout(p=0.1, inplace=False)
        )
        (norm2): LayerNorm()
        (dropout2): Dropout(p=0.1, inplace=False)
      )
    )
    (linear): Linear(in_features=64, out_features=1, bias=True)
```

4. Linear Layer

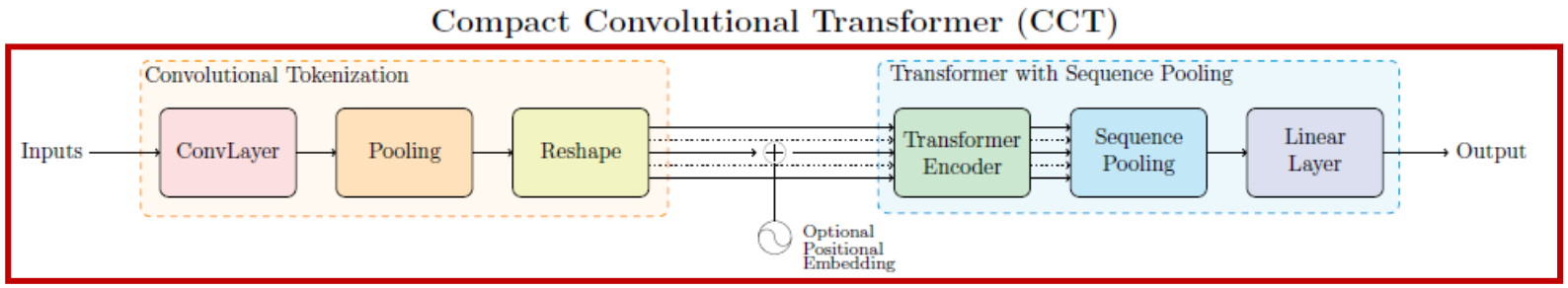
➢ Sequence Pooling




```
    )
    (final_linear): Linear(in_features=64, out_features=10, bias=True)
```

4. CCT

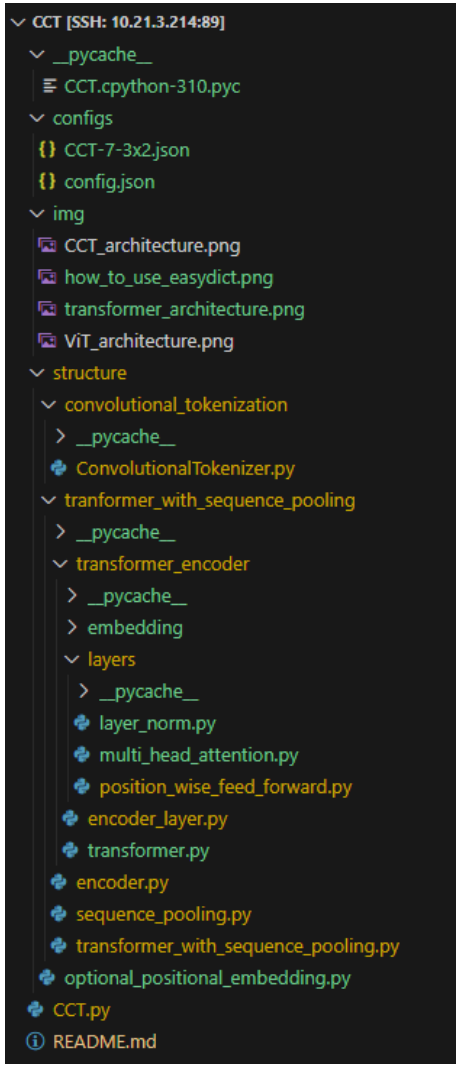
➢ CCT = Convolutional_Tokenization + Transformer_with_SeqPool

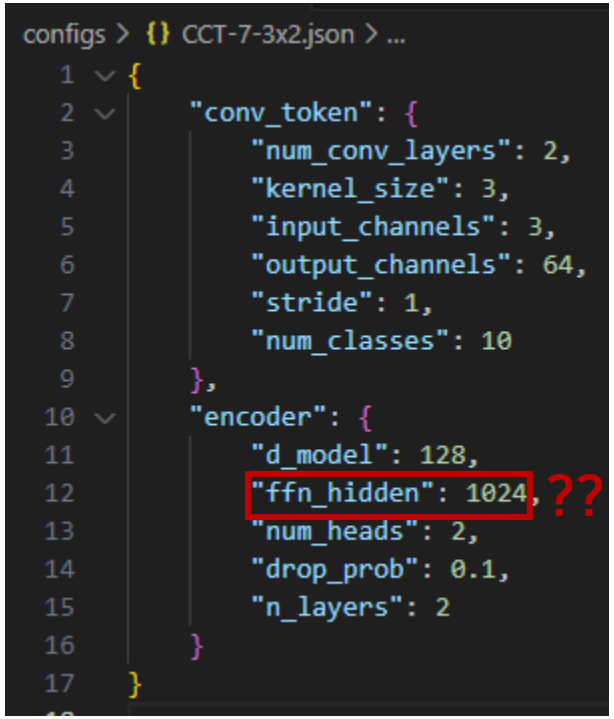



```python
class CCT(nn.Module):
    def __init__(self, cfg):
        super().__init__()

        self.Convolutional_Tokenization = ConvolutionalTokenizer(cfg)
        self.Transformer_With_SeqPool = TransformerWithSeqPool(cfg)

    def forward(self, x, src_mask):
        conv_tokenizer_output = self.Convolutional_Tokenization(x)
        cct_ouput = self.Transformer_With_SeqPool(conv_tokenizer_output, src_mask)

        return cct_ouput

config_path = '/home/hayeon/CCT/configs/CCT-2-3x2.json'
with open(config_path, 'r') as f:
    cfg_dict = json.load(f)

cfg = edict(cfg_dict)
model = CCT(cfg)
model = model.cuda()
print(model)
```

* File Structure

- 왼 : Class 별로 자잘하게 구분함
- 오 : Config 파일 구조
  - ffn_hidden : d_model의 4~8배?





Train

```python
def train_one_epoch(epoch, model, trainloader, criterion, optimizer, device, print_interval=10):
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0

    for i, data in enumerate(tqdm(trainloader, desc=f"Epoch {epoch+1}", unit="batch")):
        inputs, labels = data[0].to(device), data[1].to(device)

        optimizer.zero_grad()

        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        _, predicted = outputs.max(1)
        total += labels.size(0)
        correct += predicted.eq(labels).sum().item()

    epoch_loss = running_loss / len(trainloader)
    epoch_acc = 100. * correct / total

    print(f"Epoch {epoch+1} - Loss: {epoch_loss:.3f}, Accuracy: {epoch_acc:.2f}%")
    print("============================")
```

```python
def main():
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    args = parse_arguments()
    config = load_config(args.cfg)

    model = CCT(config)
    if torch.cuda.device_count() > 1:
        print("GPU numbers : ", torch.cuda.device_count())
        model = nn.DataParallel(model)
    model.to(device)

    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)

    trainloader, testloader = get_data_loaders(batch_size=args.batch_size)

    for epoch in range(args.epochs):
        train_one_epoch(epoch, model, trainloader, criterion, optimizer, device)

        if (epoch+1) % 5 == 0:
            model_name = f"{args.cfg.split('.')[0].split('/')[-1]}_{args.batch_size}_{epoch}.pth"
            print(model_name)
            model_dir = "models/CCT-7-7x1_128_ver.2"
            os.makedirs(model_dir, exist_ok=True)
            torch.save(model.state_dict(), os.path.join(model_dir, model_name))

        if (epoch+1) % 5 == 0:
            validate(model, testloader, device)
```

Train & Val

➢CCT-7/7x1

Table 5: Transformer backbones in each variant.

Model	# Layers	# Heads	Ratio	Dim
ViT-Lite-6	6	4	2	256
ViT-Lite-7	7	4	2	256
CVT-6	6	4	2	256
CVT-7	7	4	2	256
CCT-2	2	2	1	128
CCT-4	4	2	1	128
CCT-6	6	4	2	256
CCT-7	7	4	2	256
CCT-14	14	6	3	384

Table 6: Tokenizers in each variant.

Model	# Layers	# Convs	Kernel	Stride
ViT-Lite-7/8	7	1	8×8	8×8
ViT-Lite-7/4	7	1	4×4	4×4
CVT-7/8	7	1	8×8	8×8
CVT-7/4	7	1	4×4	4×4
CCT-2/3x2	2	2	3×3	1×1
CCT-7/3x1	7	1	3×3	1×1
CCT-7/7x2	7	2	7×7	2×2

```
{
    "conv_token": {
        "num_conv_layers": 1,
        "kernel_size": 7,
        "input_channels": 3,
        "output_channels": 128,
        "stride": 1,
        "num_classes": 10,
        "padding": 2
    },
    "encoder": {
        "d_model": 128,
        "ffn_hidden": 1024,
        "num_heads": 4,
        "drop_prob": 0.1,
        "n_layers": 7
    }
}
```

Train & Val

- CCT-7/7x1

```
==========================================================================
Total params: 2,325,771
Trainable params: 2,325,771
Non-trainable params: 0
--------------------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 149.14
Params size (MB): 8.87
Estimated Total Size (MB): 158.02
--------------------------------------------------------------------------
```

Train & Val

➢CCT-7/7x1

```
GPU numbers :  4
Epoch 1: 100%|| 196/196 [01:02<00:00,  3.14batch/s]
Epoch 1 - Loss: 2.233, Accuracy: 15.28%
============================
Epoch 2: 100%|| 196/196 [00:46<00:00,  4.21batch/s]
Epoch 2 - Loss: 2.095, Accuracy: 21.85%
============================
Epoch 3: 100%|| 196/196 [00:46<00:00,  4.20batch/s]
Epoch 3 - Loss: 2.006, Accuracy: 25.69%
============================
Epoch 4: 100%|| 196/196 [00:46<00:00,  4.25batch/s]
Epoch 4 - Loss: 1.914, Accuracy: 29.45%
============================
Epoch 5: 100%|| 196/196 [00:46<00:00,  4.26batch/s]
Epoch 5 - Loss: 1.827, Accuracy: 32.44%
============================
CCT-7-7x1_256_4.pth
Validation Accuracy: 33.86%
Epoch 6: 100%|| 196/196 [00:47<00:00,  4.14batch/s]
Epoch 6 - Loss: 1.769, Accuracy: 34.43%
============================
Epoch 7: 100%|| 196/196 [00:47<00:00,  4.16batch/s]
Epoch 7 - Loss: 1.726, Accuracy: 35.95%
============================
Epoch 8: 100%|| 196/196 [00:47<00:00,  4.11batch/s]
Epoch 8 - Loss: 1.683, Accuracy: 37.72%
============================
Epoch 9: 100%|| 196/196 [00:46<00:00,  4.20batch/s]
Epoch 9 - Loss: 1.659, Accuracy: 38.53%
============================
Epoch 10: 100%|| 196/196 [00:46<00:00,  4.20batch/s]
Epoch 10 - Loss: 1.634, Accuracy: 39.73%
============================
CCT-7-7x1_256_9.pth
Validation Accuracy: 41.66%
```

Train & Val

- CCT-7/7x1

Train & Val

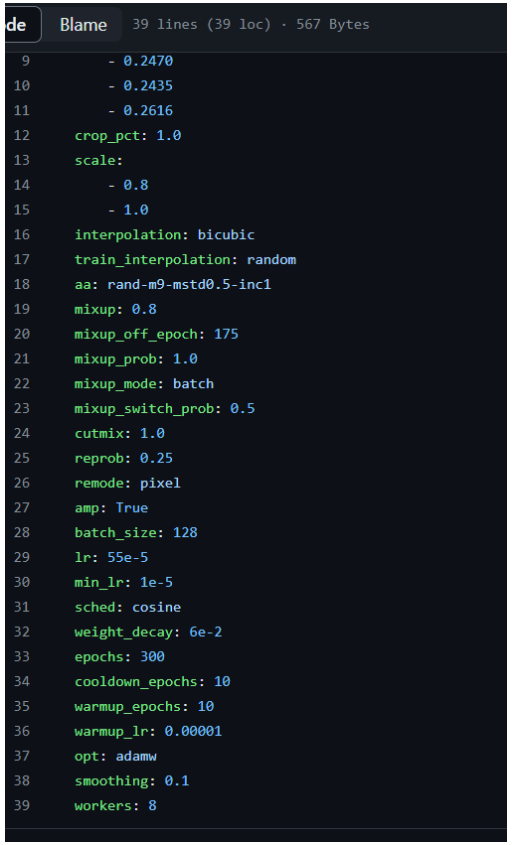
➢CCT-2-3x2

```
============================
Epoch 25: 100%|████████████████████████████████████████████████████████████| 196/196 [00:38<00:00,  5.05batch/s]
Epoch 25 - Loss: 1.280, Accuracy: 53.83%
============================
CCT-2-3x2_256_24.pth
Validation Accuracy: 53.97%
```

끄읏

- 모델 구현 및 학습 과정 전반을 경험
- 모델 구조에 대한 정확한 이해가 있으면 다른 모델들도 짜볼 수 있을 듯

- 조금 더 정돈된 코드
- 어떻게 하면 성능을 높일 수 있는가



감사합니다