یادگیری عمیق

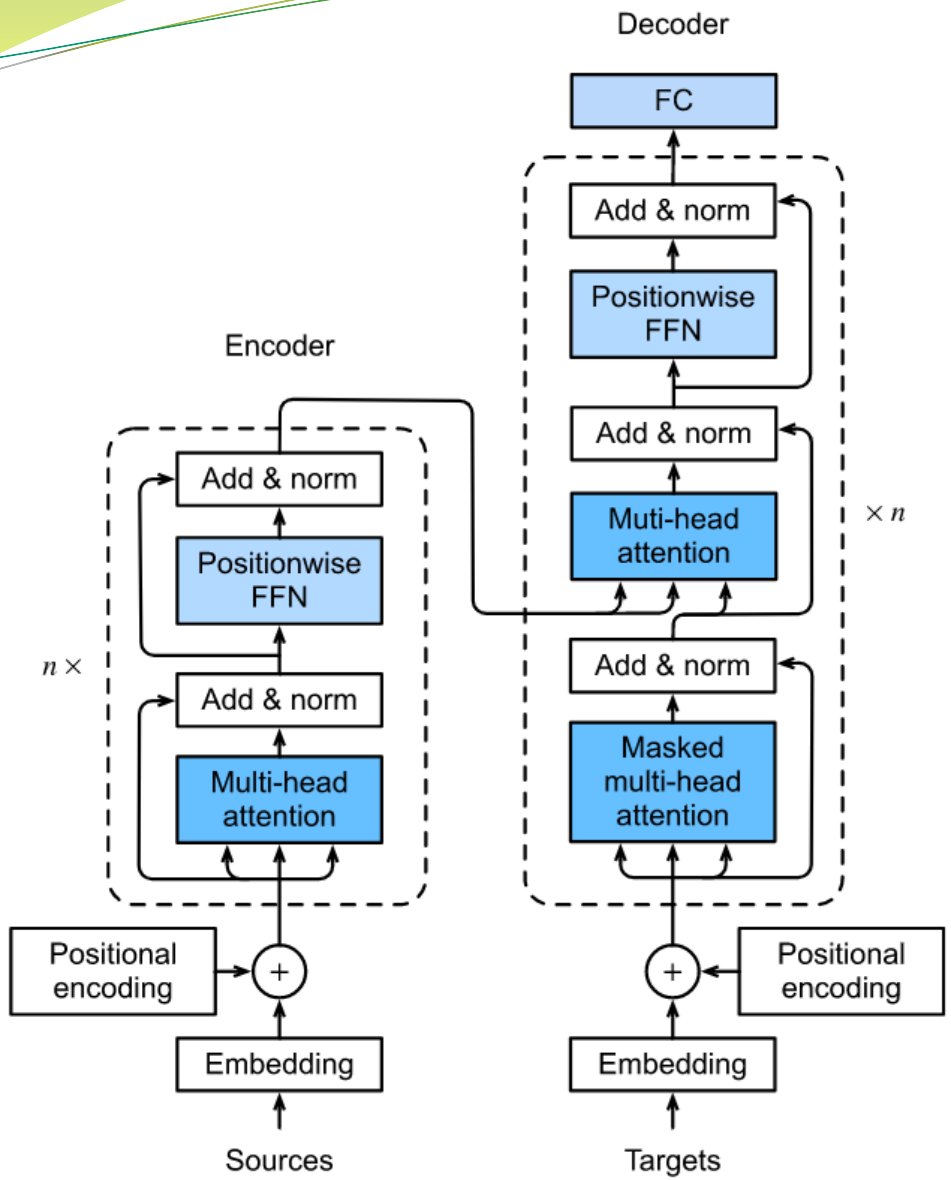
مدرس: محمدرضا محمدی

بهار ۱۴۰۲

مکانیزم‌های توجه

Attention Mechanisms

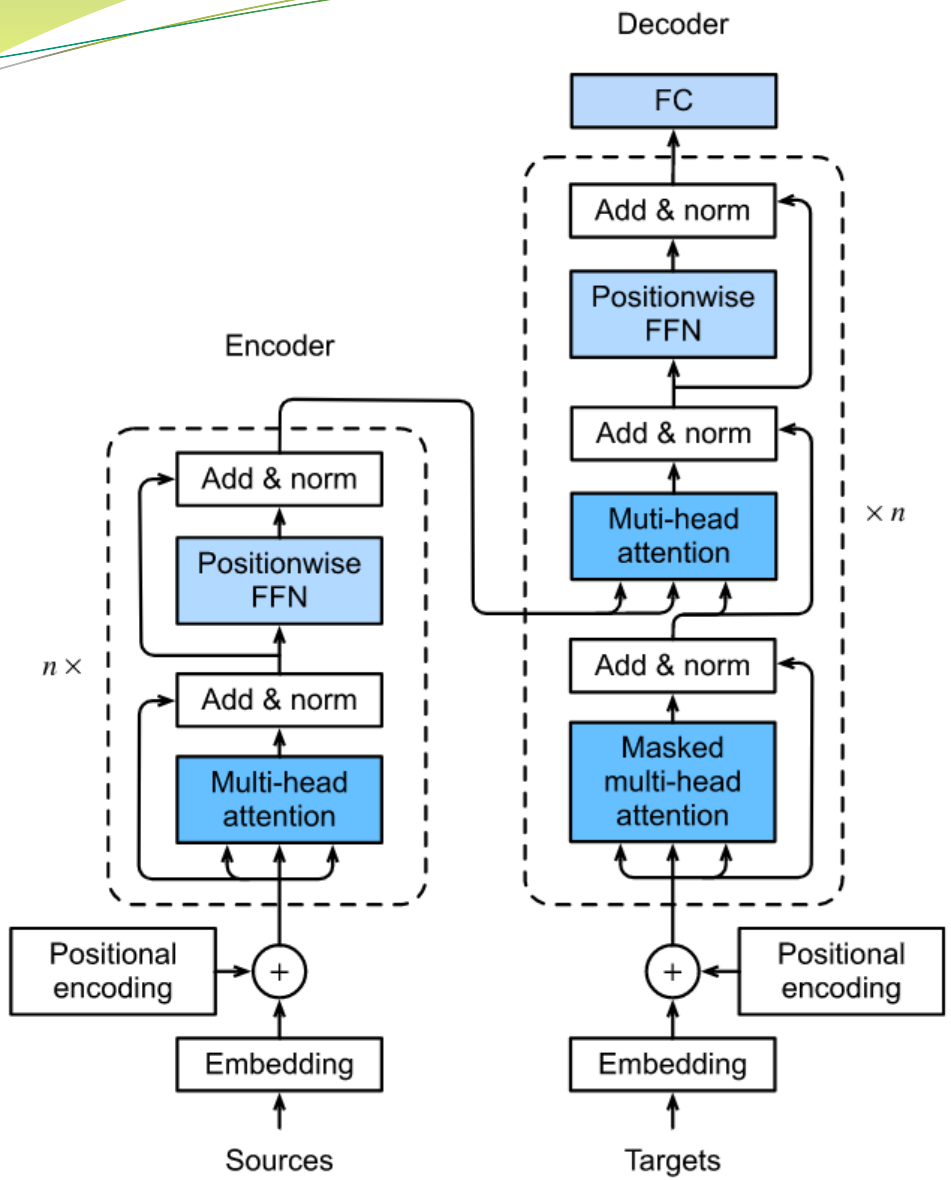
مبدّل (Transformer)

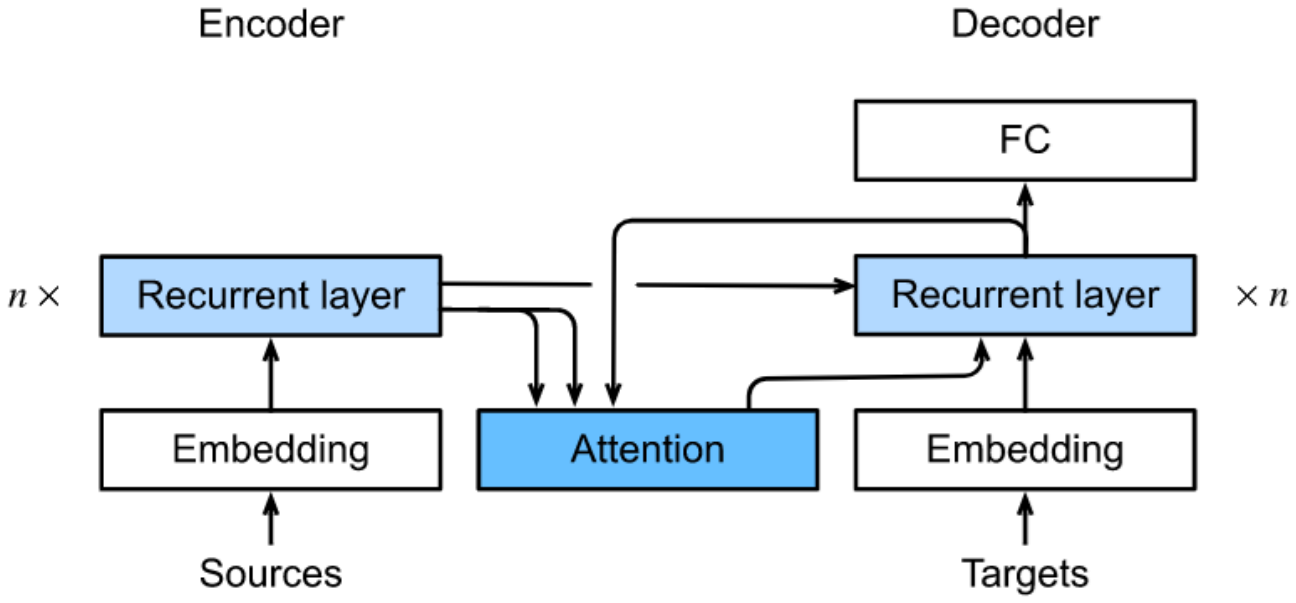


- بر خلاف مدل‌های ابتدایی توجه به خود که برای بازنمایی ورودی به RNNها متکی هستند، مدل مبدّل صرفاً مبتنی بر مکانیزم‌های توجه (بدون هیچ لایه کانولوشنی یا بازگشتی)
- اگرچه در ابتدا برای تبدیل دنباله به دنباله روی داده‌های متنی پیشنهاد شد، مبدّل‌ها در طیف گسترده‌ای از کاربردها مانند پردازش زبان، بینایی، گفتار و یادگیری تقویتی فراگیر شده است
- معماری کلّی مبدّل از ساختار encoder-decoder استفاده می‌کند

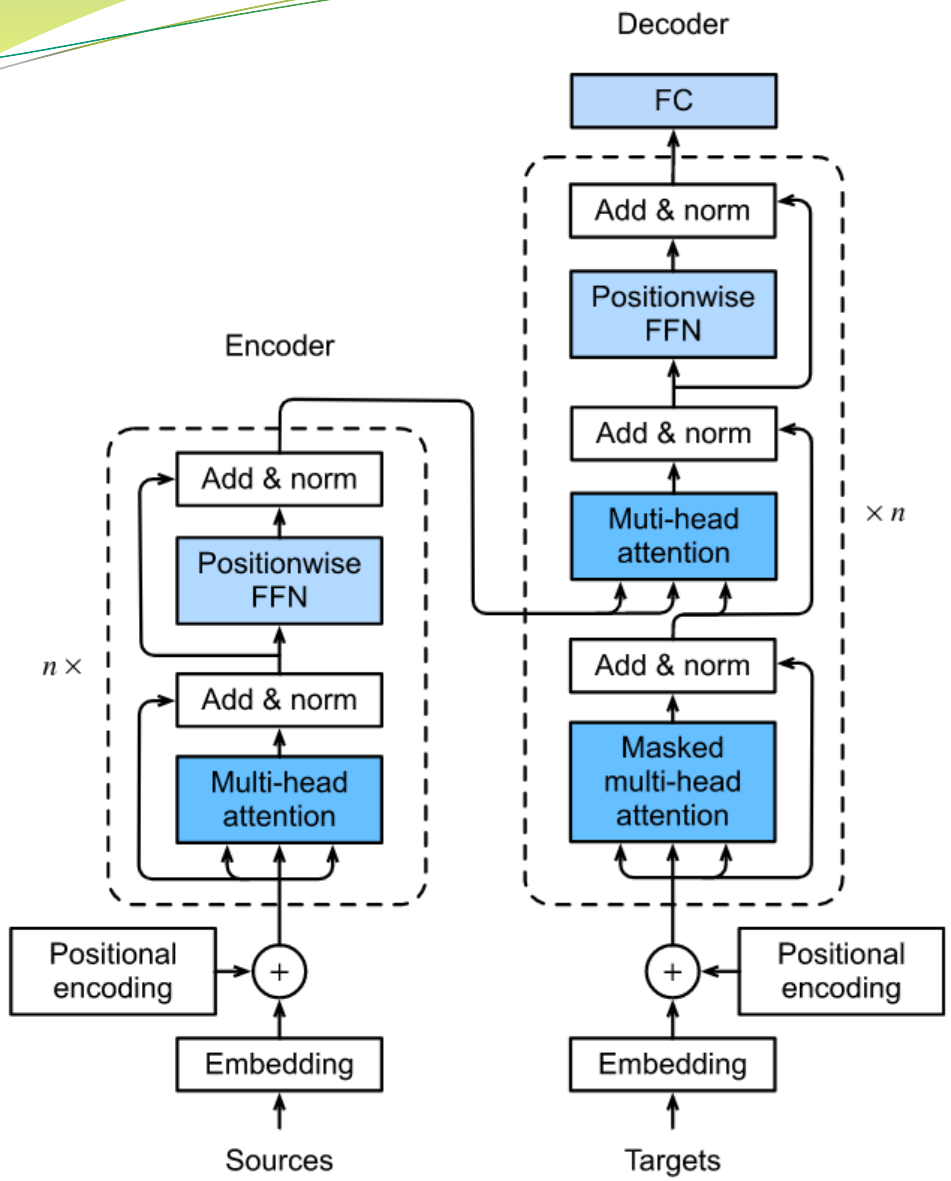
مبدّل (Transformer)

- برخلاف توجه Bahdanau، جانمایی ورودی‌ها و خروجی‌ها با کدگذاری موقعیتی جمع می‌شود و به encoder و decoder وارد می‌شوند

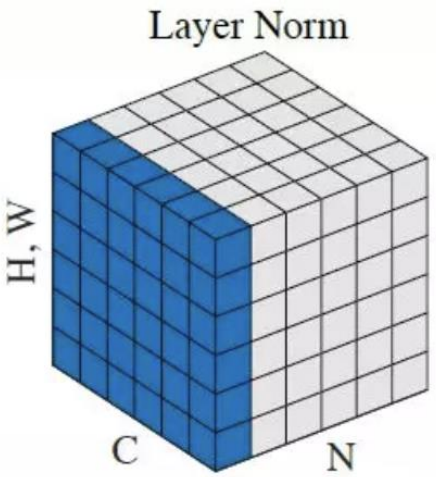




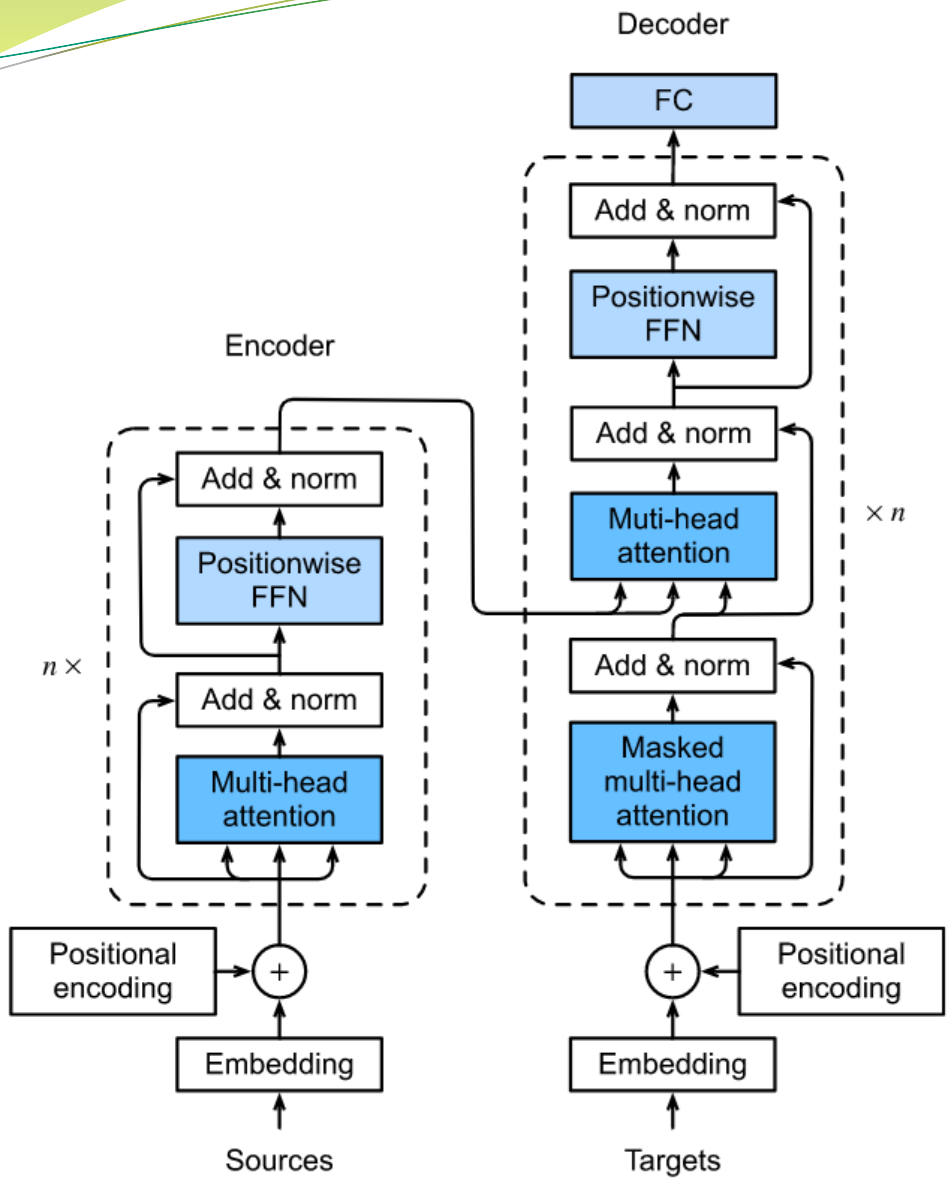
مبدّل (Transformer)



- کدگذار از n لایه کدگذار مشابه تشکیل شده است
- هر لایه کدگذار شامل دو جزء سازنده اصلی است
  - توجه به خود چندسر
    - queries، keys و values همگی خروجی لایه کدگذار قبلی است
    - با الهام از ResNet، از اتصال باقی‌مانده استفاده می‌شود
    - از نرمال‌سازی لایه‌ای هم استفاده می‌شود
  - شبکه پیش‌خور موقعیتی
    - با استفاده از یک MLP یکسان، بازنمایی در هر موقعیت تبدیل می‌شود
    - از Add & Norm هم استفاده می‌شود



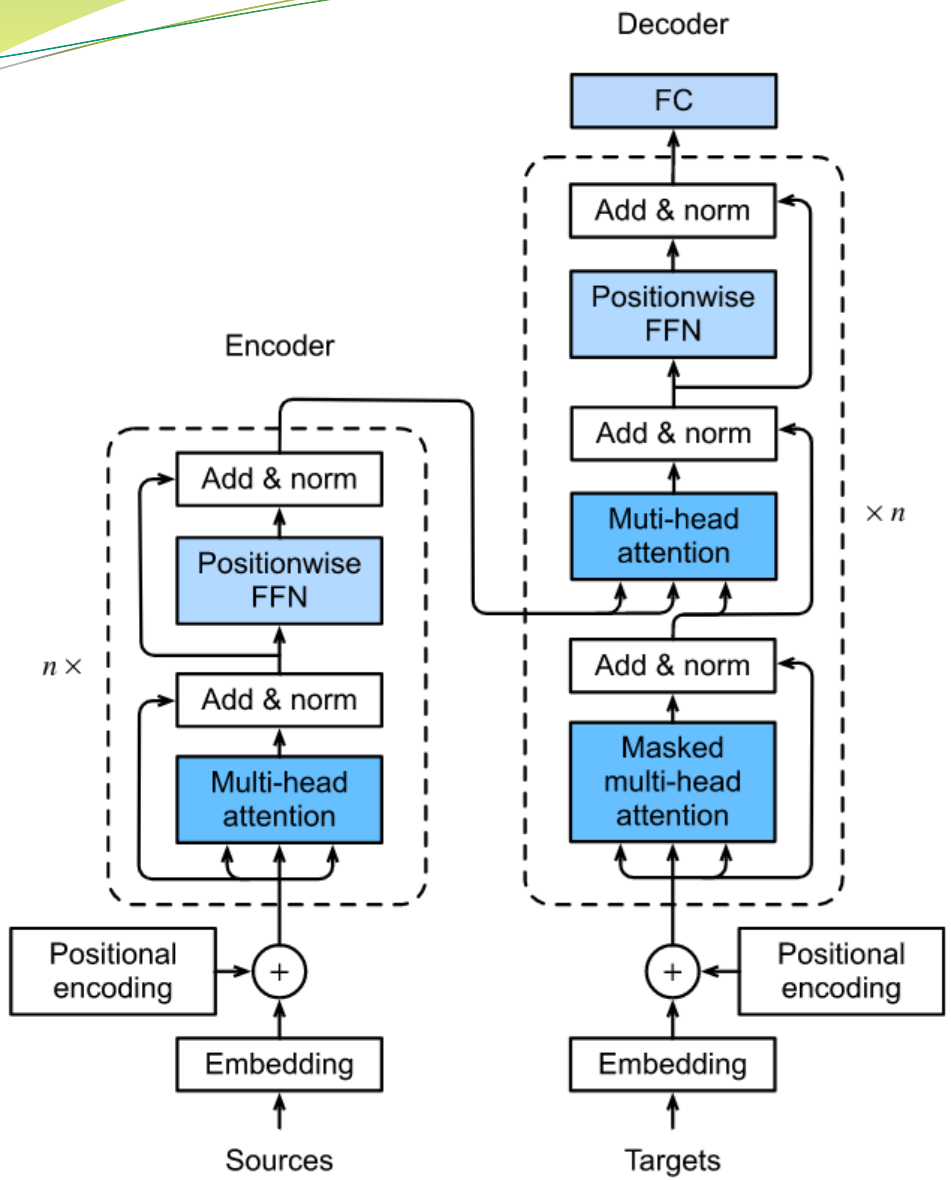
مبدّل (Transformer)




```python
class PositionWiseFFN(nn.Module):
    """Positionwise feed-forward network."""
    def __init__(self, ffn_num_input, ffn_num_hiddens,
                 ffn_num_outputs, **kwargs):
        super(PositionWiseFFN, self).__init__(**kwargs)
        self.dense1 = nn.Linear(ffn_num_input, ffn_num_hiddens)
        self.relu = nn.ReLU()
        self.dense2 = nn.Linear(ffn_num_hiddens, ffn_num_outputs)

    def forward(self, X):
        return self.dense2(self.relu(self.dense1(X)))

ffn = PositionWiseFFN(4, 4, 8)
ffn.eval()
ffn(torch.ones((2, 3, 4)))[0]
```

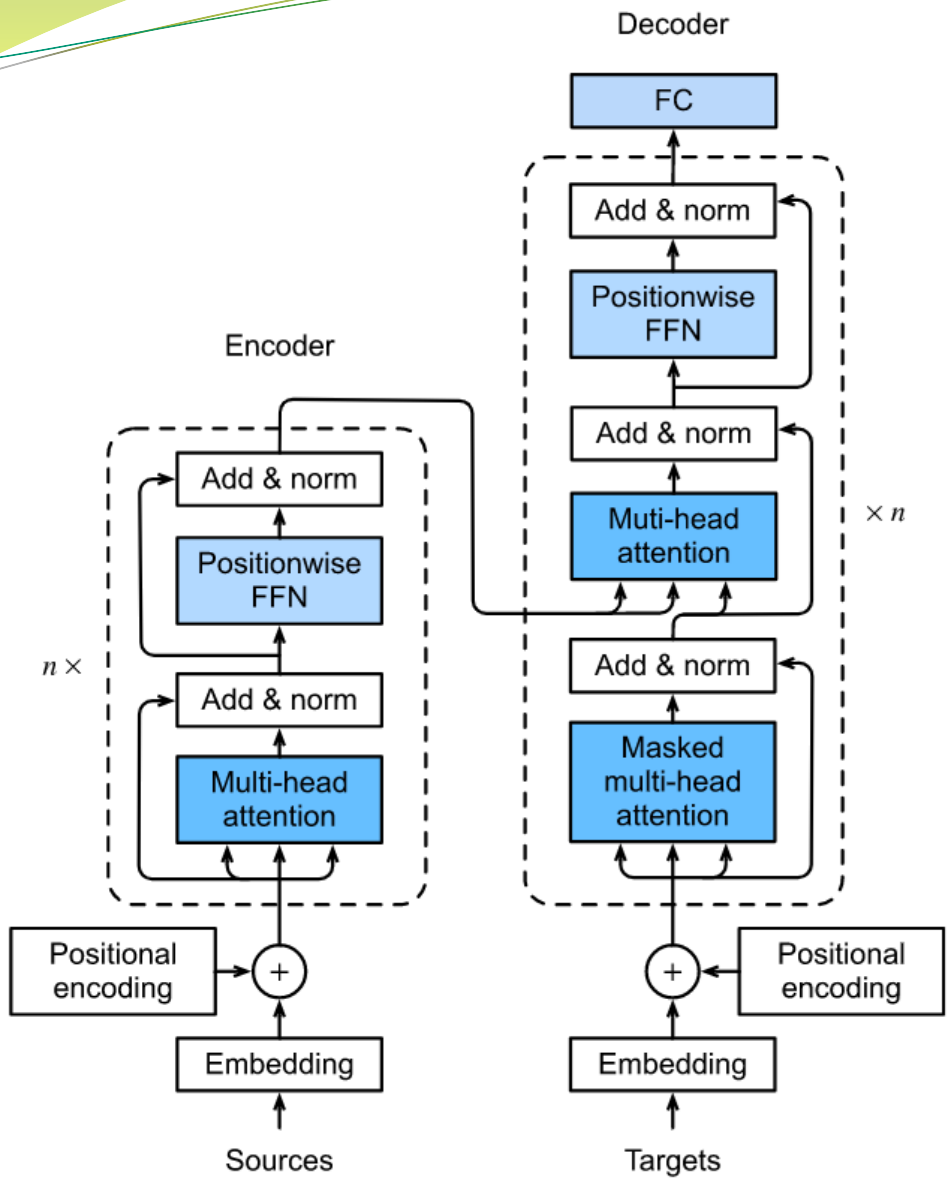
```
tensor([[-0.386, -0.249, -0.341, -0.025, 0.463, -0.432, -0.170, 0.202],
        [-0.386, -0.249, -0.341, -0.025, 0.463, -0.432, -0.170, 0.202],
        [-0.386, -0.249, -0.341, -0.025, 0.463, -0.432, -0.170, 0.202]],
       grad_fn=<SelectBackward0>)
```

مبدّل (Transformer)



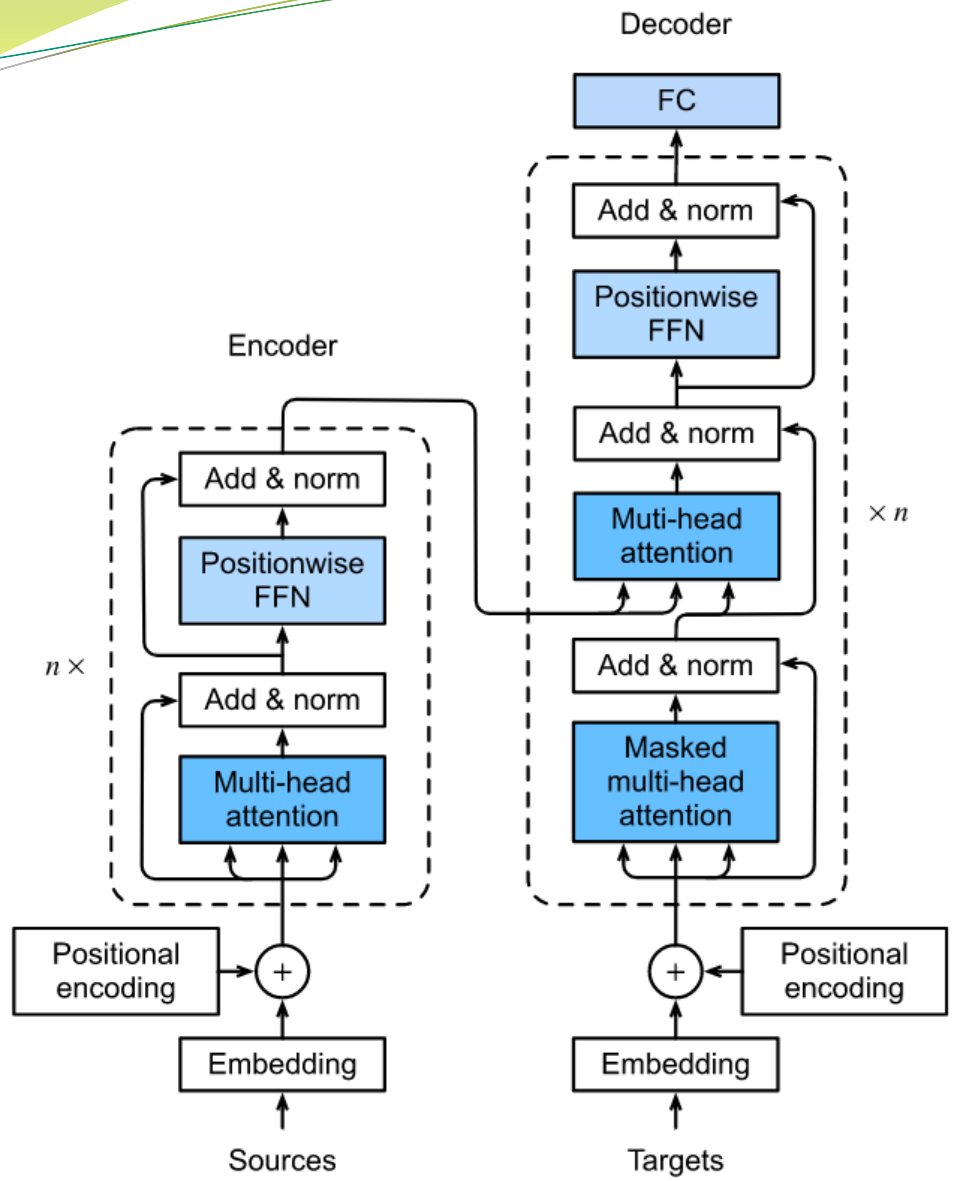

```python
class AddNorm(nn.Module):
    """Residual connection followed by layer normalization."""
    def __init__(self, normalized_shape, dropout, **kwargs):
        super(AddNorm, self).__init__(**kwargs)
        self.dropout = nn.Dropout(dropout)
        self.ln = nn.LayerNorm(normalized_shape)

    def forward(self, X, Y):
        return self.ln(self.dropout(Y) + X)
```

مبدّل (Transformer)



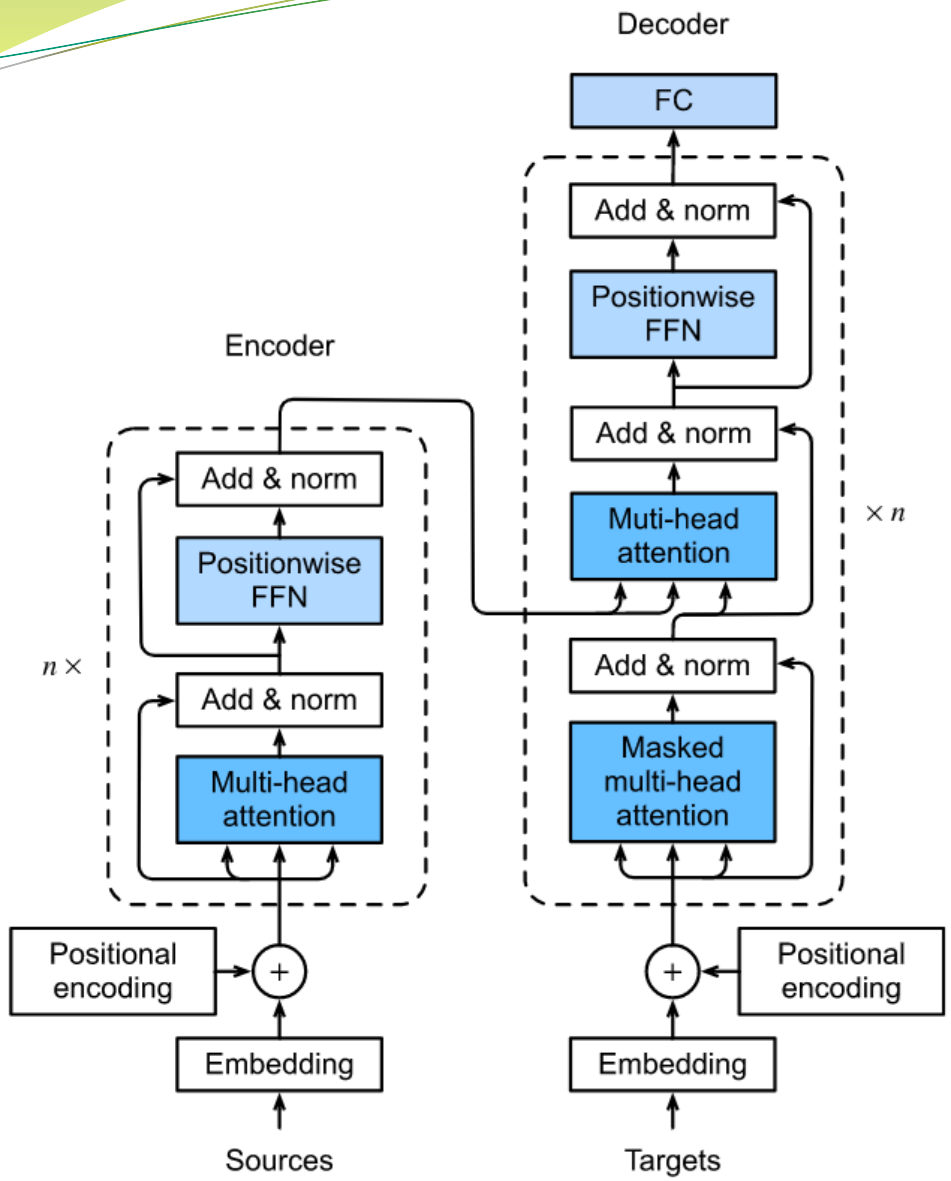

```
class EncoderBlock(nn.Module):
    """Transformer encoder block."""
    def __init__(self, key_size, query_size, value_size,
                 num_hiddens, norm_shape, ffn_num_input,
                 ffn_num_hiddens, num_heads, dropout,
                 use_bias=False, **kwargs):
        super(EncoderBlock, self).__init__(**kwargs)
        self.attention = d2l.MultiHeadAttention(
            key_size, query_size, value_size, num_hiddens,
            num_heads, dropout, use_bias)
        self.addnorm1 = AddNorm(norm_shape, dropout)
        self.ffn = PositionWiseFFN(
            ffn_num_input, ffn_num_hiddens, num_hiddens)
        self.addnorm2 = AddNorm(norm_shape, dropout)

    def forward(self, X, valid_lens):
        Y = self.addnorm1(X, self.attention(X, X, X, valid_lens))
        return self.addnorm2(Y, self.ffn(Y))
```

مبدّل (Transformer)



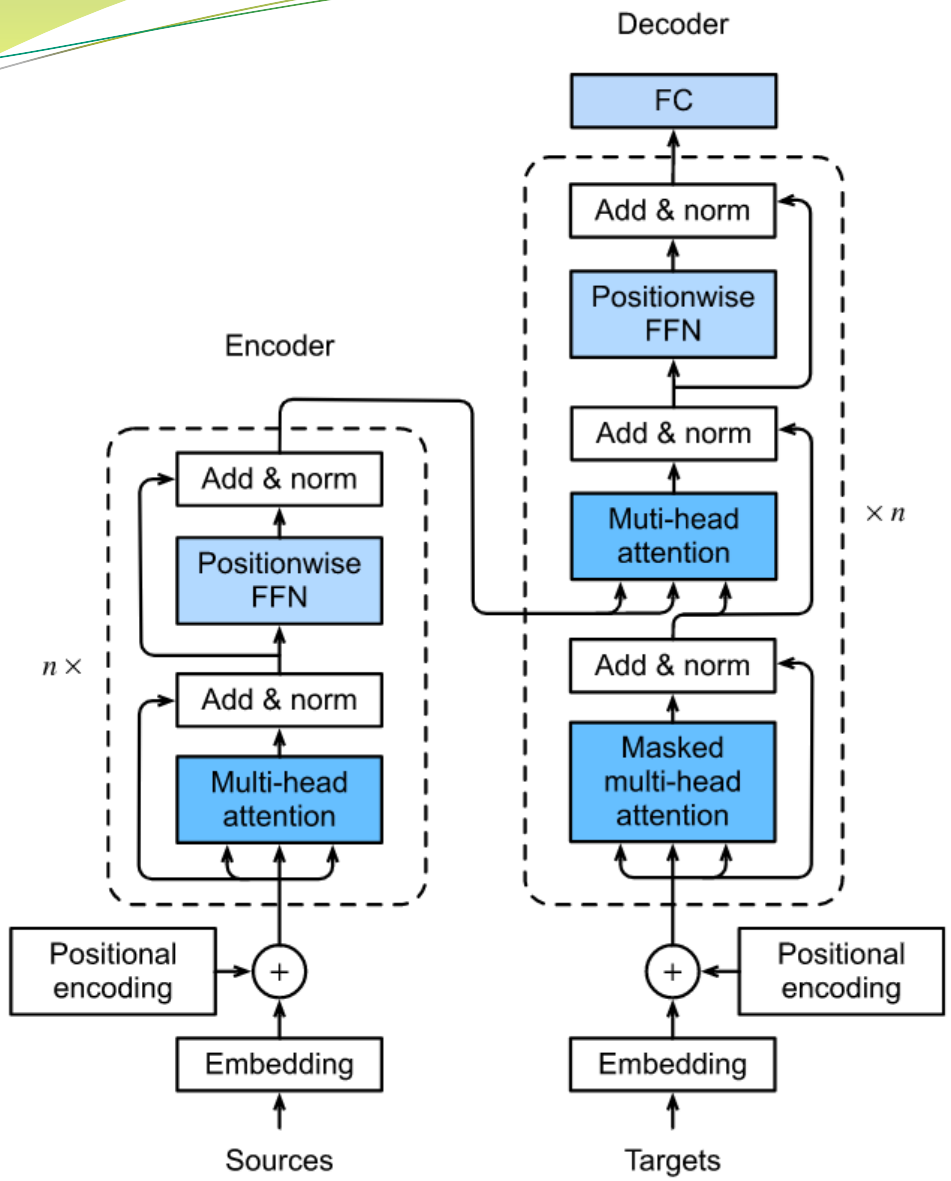

```
class TransformerEncoder(d2l.Encoder):
    """Transformer encoder."""
    def __init__(self, vocab_size, key_size, query_size, value_size,
                 num_hiddens, norm_shape, ffn_num_input, ffn_num_hiddens,
                 num_heads, num_layers, dropout, use_bias=False, **kwargs):
        super(TransformerEncoder, self).__init__(**kwargs)
        self.num_hiddens = num_hiddens
        self.embedding = nn.Embedding(vocab_size, num_hiddens)
        self.pos_encoding = d2l.PositionalEncoding(num_hiddens, dropout)
        self.blks = nn.Sequential()
        for i in range(num_layers):
            self.blks.add_module("block"+str(i),
                EncoderBlock(key_size, query_size, value_size, num_hiddens,
                             norm_shape, ffn_num_input, ffn_num_hiddens,
                             num_heads, dropout, use_bias))

    def forward(self, X, valid_lens, *args):
        # Since positional encoding values are between -1 and 1, the embedding
        # values are multiplied by the square root of the embedding dimension
        # to rescale before they are summed up
        X = self.pos_encoding(self.embedding(X) * math.sqrt(self.num_hiddens))
        self.attention_weights = [None] * len(self.blks)
        for i, blk in enumerate(self.blks):
            X = blk(X, valid_lens)
            self.attention_weights[
                i] = blk.attention.attention.attention_weights
        return X
```

مبدّل (Transformer)



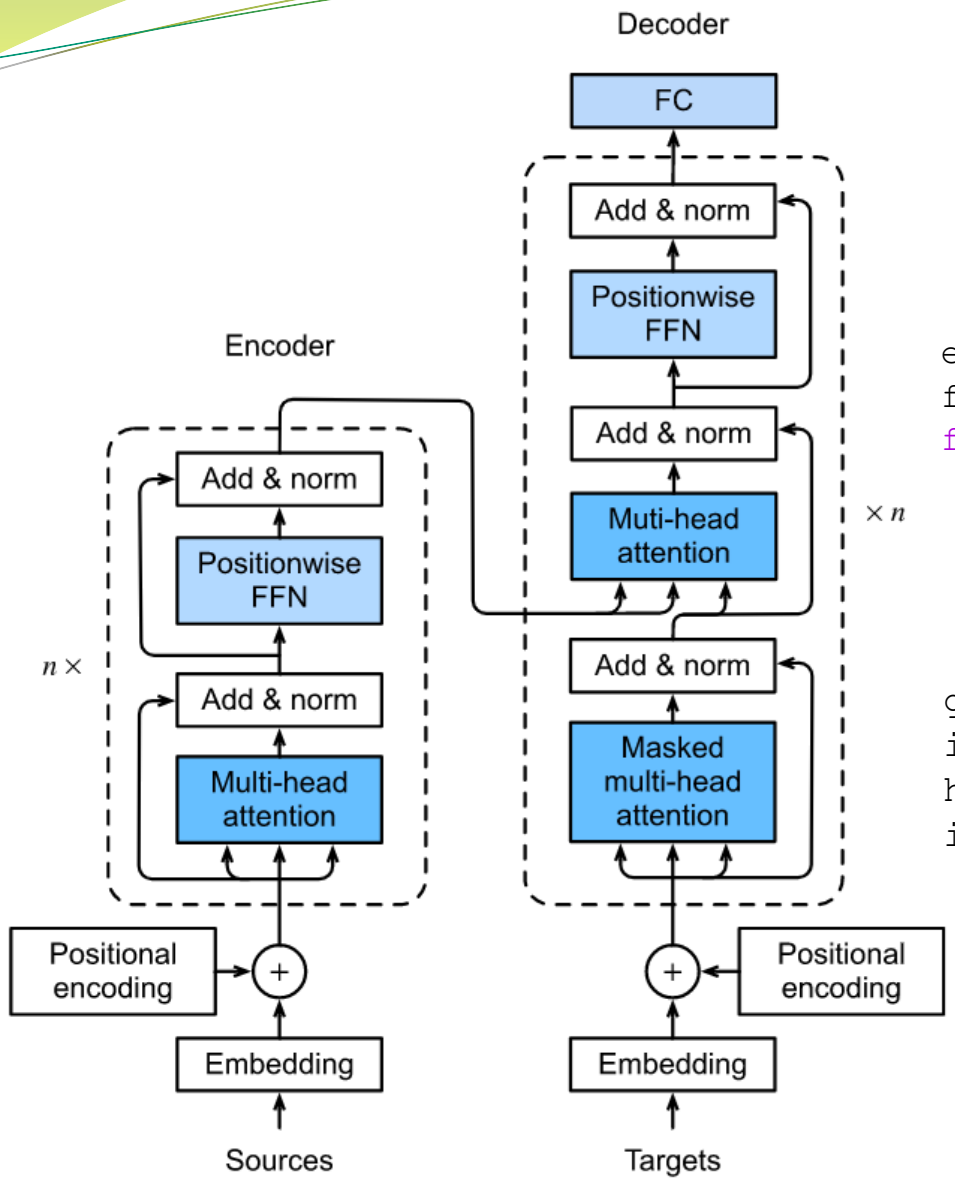
- کدگشا از n لایه کدگشا مشابه تشکیل شده است
- هر لایه کدگشا شامل سه جزء سازنده اصلی است
  - توجه به خود چندسر
    - queries، keys و values همگی خروجی لایه کدگشای قبلی است
    - از آنجائیکه خروجی یکی یکی تولید می‌شود، فقط به خروجی‌های قبلی توجه می‌کند
  - توجه کدگذار-کدگشا
    - queries خروجی لایه کدگشای قبل، keys و values خروجی کدگذار هستند
  - شبکه پیش‌خور موقعیتی

مبدّل (Transformer)




```
class DecoderBlock(nn.Module):
    # The `i`-th block in the decoder
    def __init__( ... ):

        self.attention1 = d2l.MultiHeadAttention( ... )
        self.addnorm1 = AddNorm(norm_shape, dropout)
        self.attention2 = d2l.MultiHeadAttention( ... )
        self.addnorm2 = AddNorm(norm_shape, dropout)
        self.ffn = PositionWiseFFN( ... )
        self.addnorm3 = AddNorm(norm_shape, dropout)

    def forward(self, X, state):

        key_values = torch.cat((state[2][self.i], X), axis=1)
        X2 = self.attention1(X, key_values, key_values,
                             dec_valid_lens)
        Y = self.addnorm1(X, X2)
        Y2 = self.attention2(Y, enc_outputs, enc_outputs,
                             enc_valid_lens)
        Z = self.addnorm2(Y, Y2)
        return self.addnorm3(Z, self.ffn(Z)), state
```

مبدّل (Transformer)



- بعد از آموزش مدل، می‌توان به صورت زیر از آن استفاده کرد:

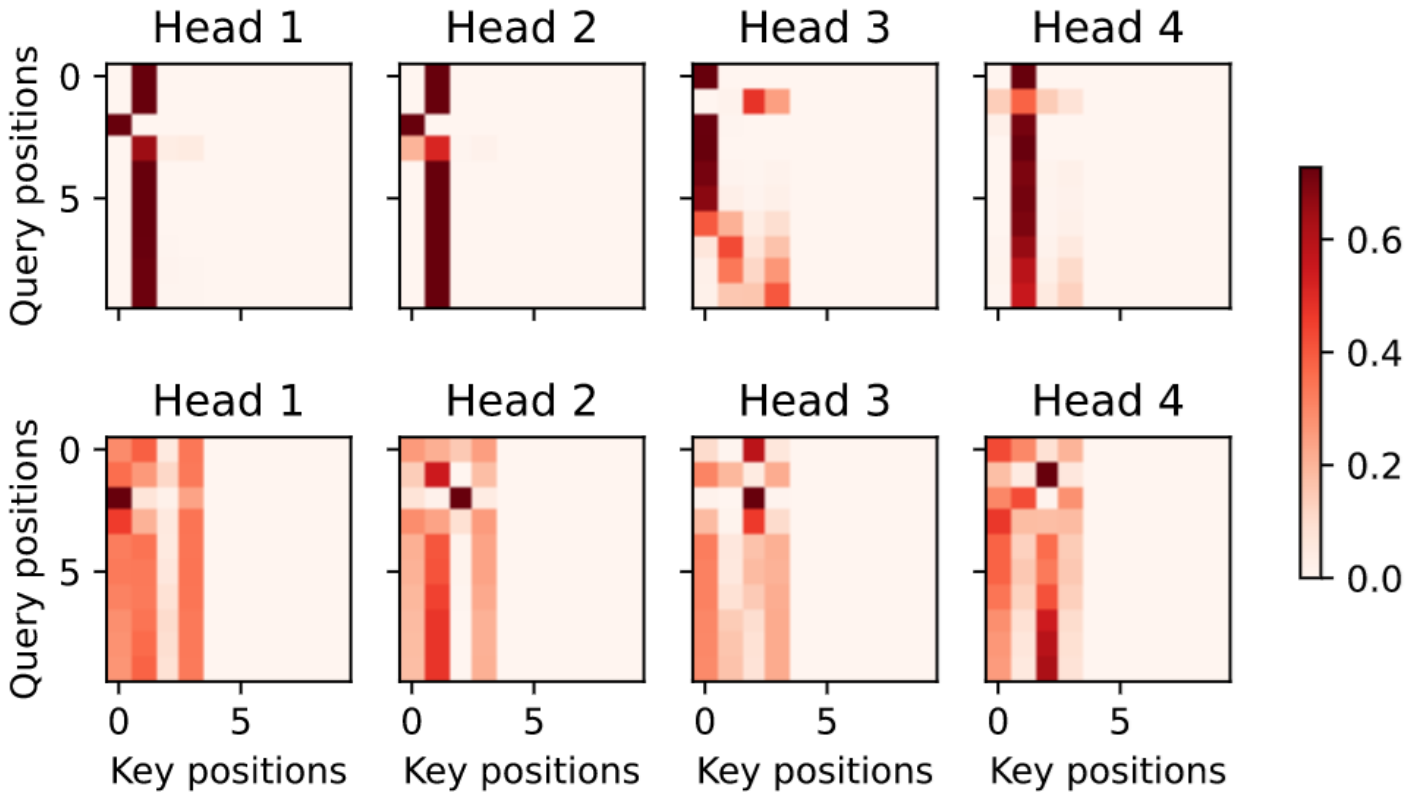
```
engs = ['go .', "i lost .", 'he\'s calm .', 'i\'m home .']
fras = ['va !', 'j\'ai perdu .', 'il est calme .', 'je suis chez moi .']
for eng, fra in zip(engs, fras):
    translation, dec_attention_weight_seq = d2l.predict_seq2seq(
        net, eng, src_vocab, tgt_vocab, num_steps, device, True)
    print(f'{eng} => {translation}, ',
          f'bleu {d2l.bleu(translation, fra, k=2):.3f}')
```

```
go . => va !,  bleu 1.000
i lost . => j'ai perdu .,  bleu 1.000
he's calm . => il est calme .,  bleu 1.000
i'm home . => je suis chez moi .,  bleu 1.000
```

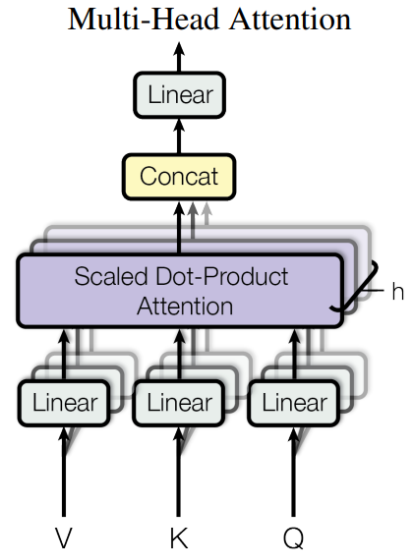
- می‌توان وزن‌های توجه را هم نمایش داد
  - برای جمله آخر این کار را انجام می‌دهیم

مبدّل (Transformer)

```
i'm home . => je suis chez moi .
```



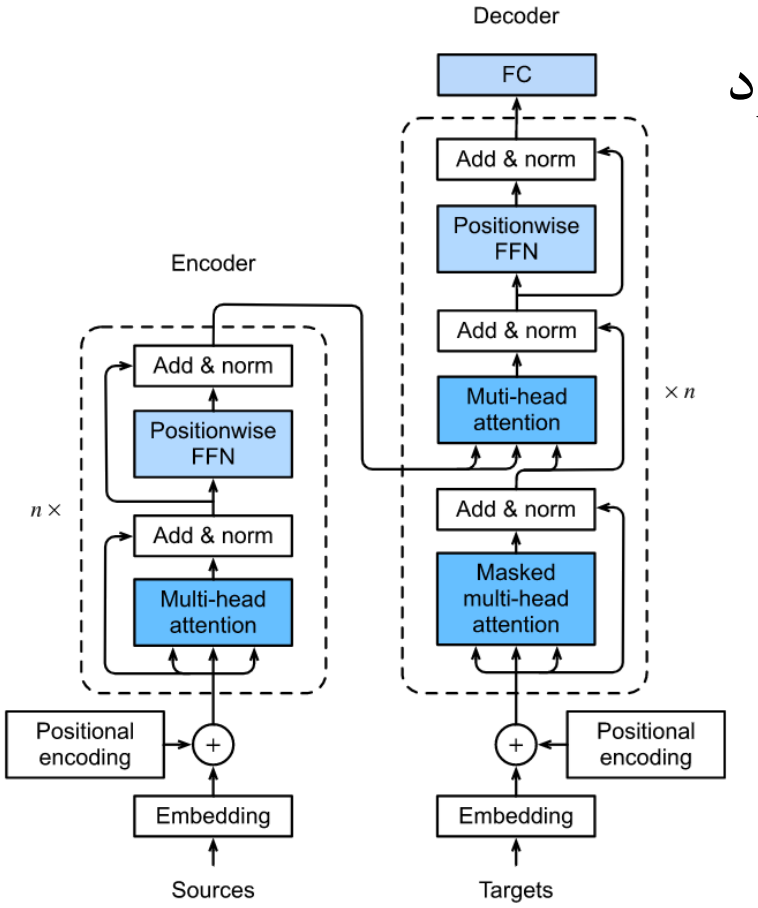
- در توجه‌به‌خود، queries و keys یکسان هستند
- طول دنباله با padding به ۱۰ رسیده است
  - توجه به توکن‌های padding را ماسک می‌کنیم
- نمایش وزن‌های دو لایه توجه کدگذار:

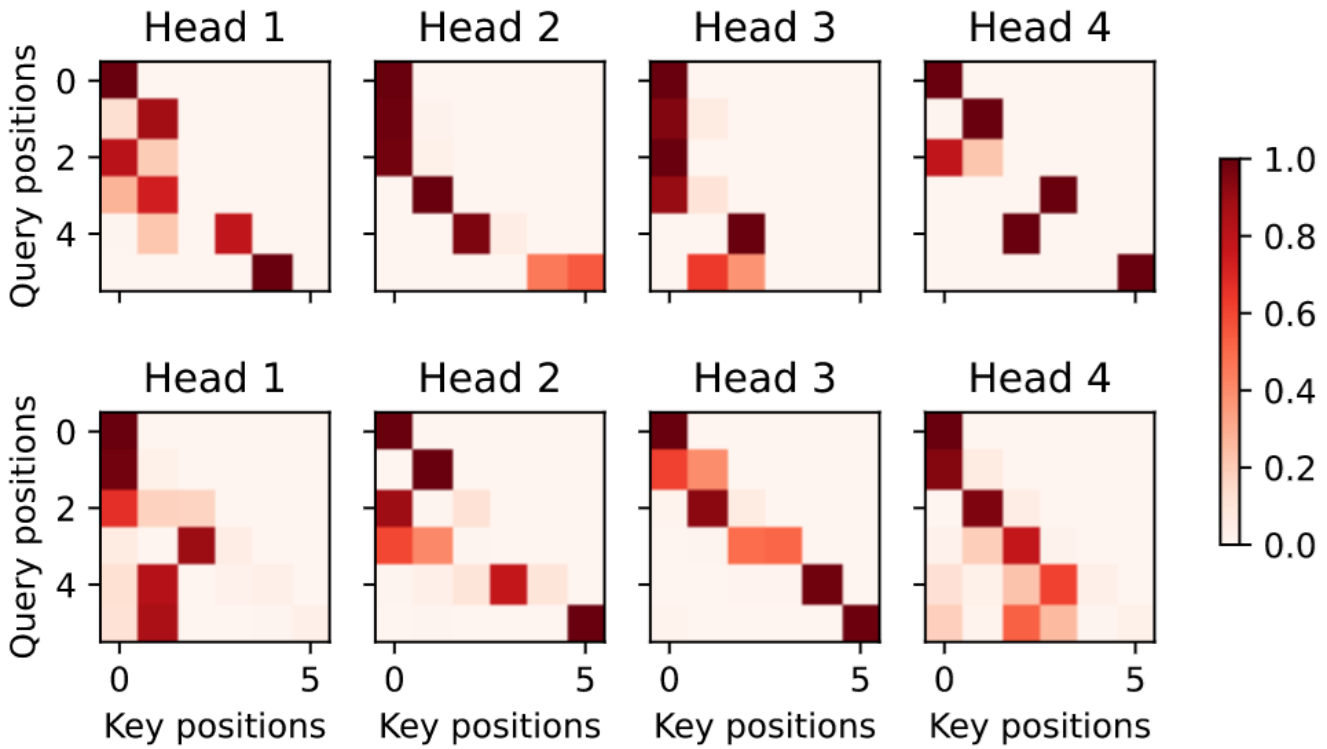


مبدّل (Transformer)

```
i'm home . => je suis chez moi .
```

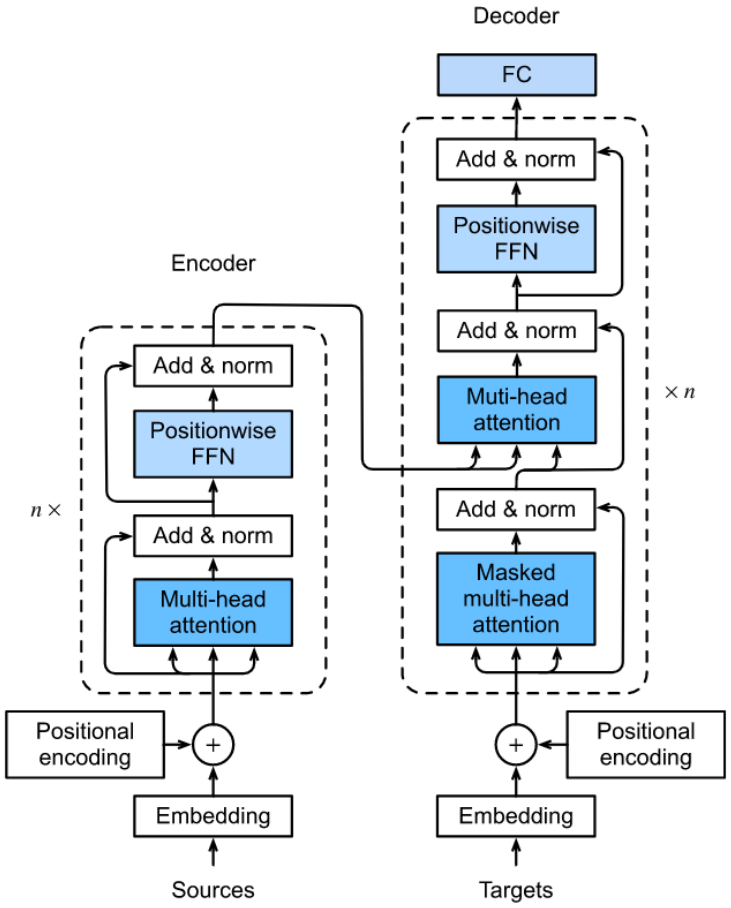
- وزن‌های توجه‌به‌خود کدگشا
- توکن‌ها یکی یکی تولید می‌شوند و توجه صرفا به توکن‌های قبلی خواهد بود



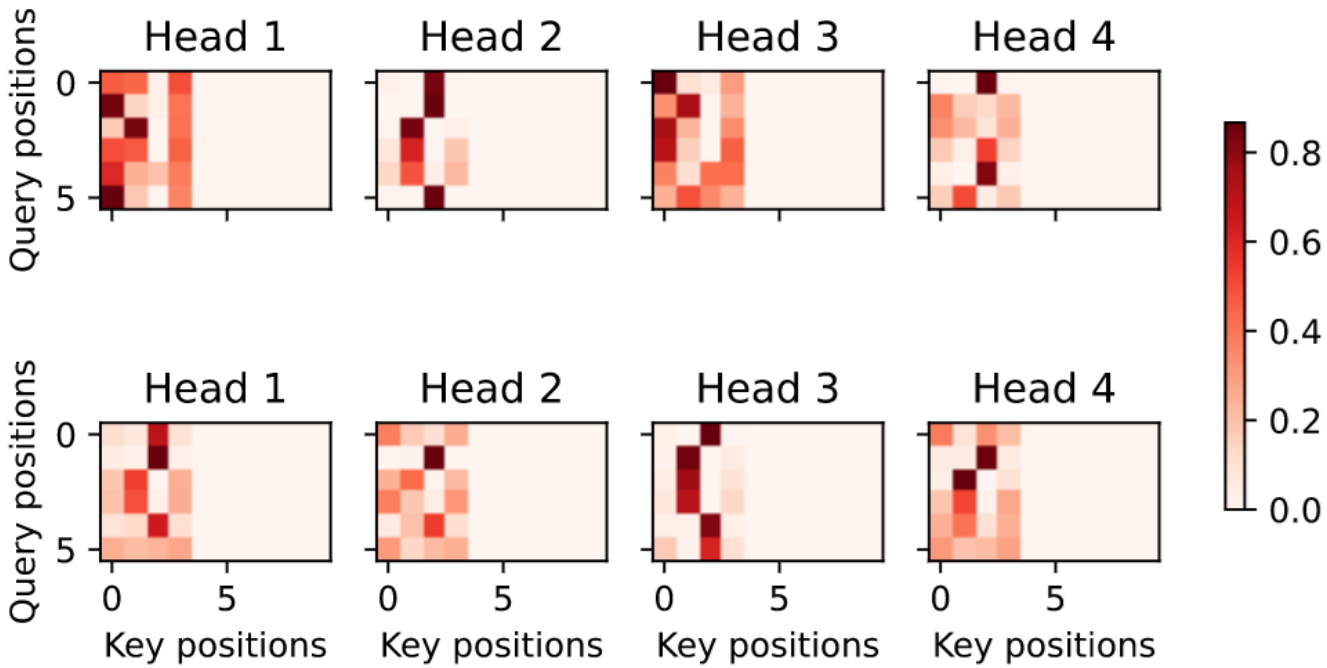


مبدّل (Transformer)

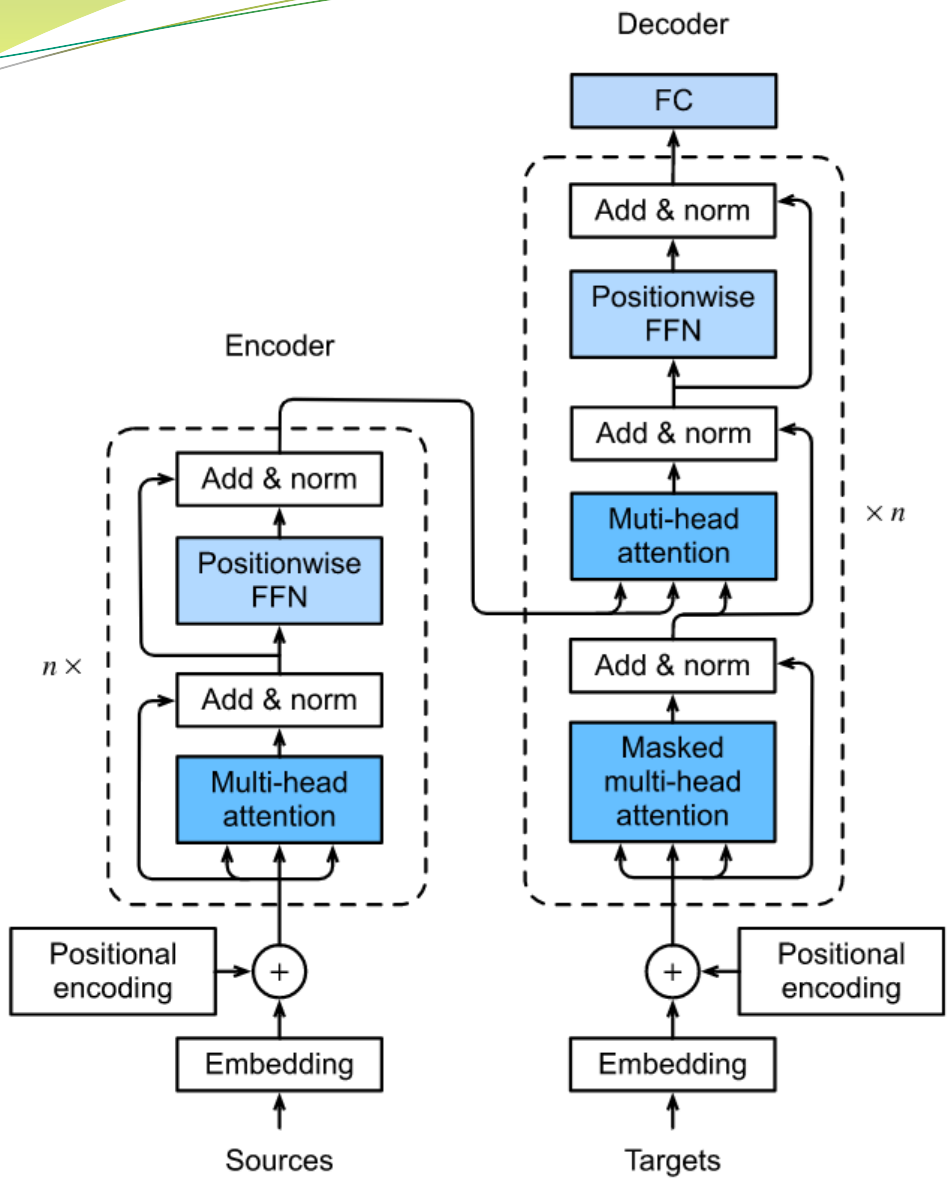
```
i'm home . => je suis chez moi .
```



- وزن‌های توجه کدگذار-کدگشا



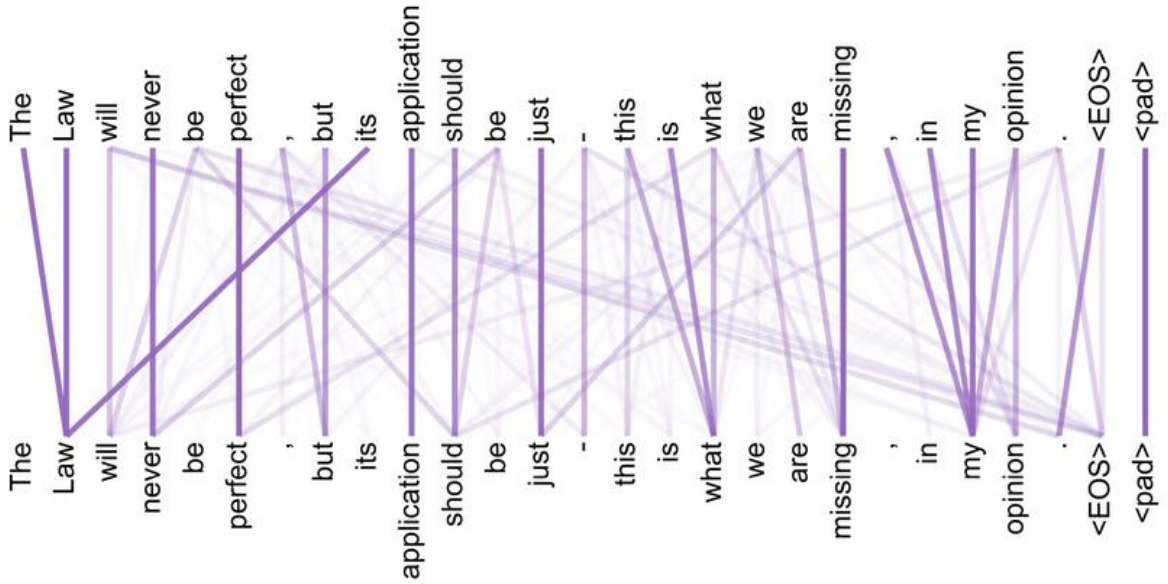
مبدّل (Transformer)



- در مبدل، توجه‌به‌خود چندسر برای بازنمایی دنباله ورودی و دنباله خروجی استفاده می‌شود
  - رمزگشا باید خاصیت Autoregressive را از طریق ماسک حفظ کند
- اتصالات باقی‌مانده و نرمال‌سازی لایه‌ای برای آموزش یک مدل بسیار عمیق مهم هستند
- شبکه پیش‌خور، بازنمایی را در تمام موقعیت‌های دنباله با استفاده از یک MLP یکسان تبدیل می‌کند

مبدل بینایی (Vision-Transformer)

- یک تصویر رنگی با ابعاد 224×224 پیکسل را می‌توان به عنوان یک دنباله با طول $n = 50,176$ در نظر گرفت
  - هزینه محاسباتی و حافظه مورد نیاز خیلی زیاد خواهد بود و به خصوص برای تصاویر بزرگ اصلا قابل استفاده نیست
- می‌توان تصویر را به بخش‌های کوچکی تقسیم کرد و برای هر کدام یک بازنمایی استخراج کرد

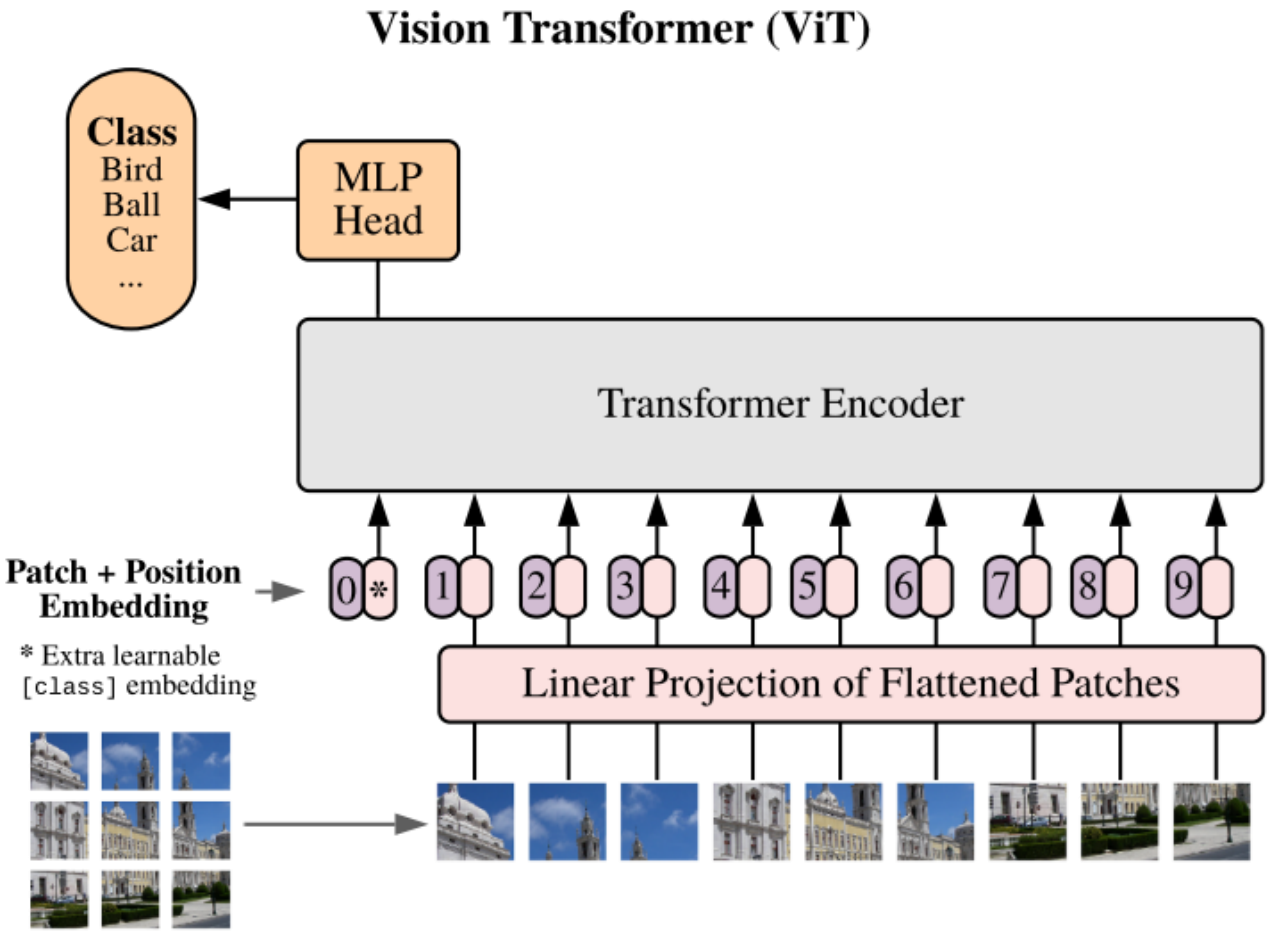


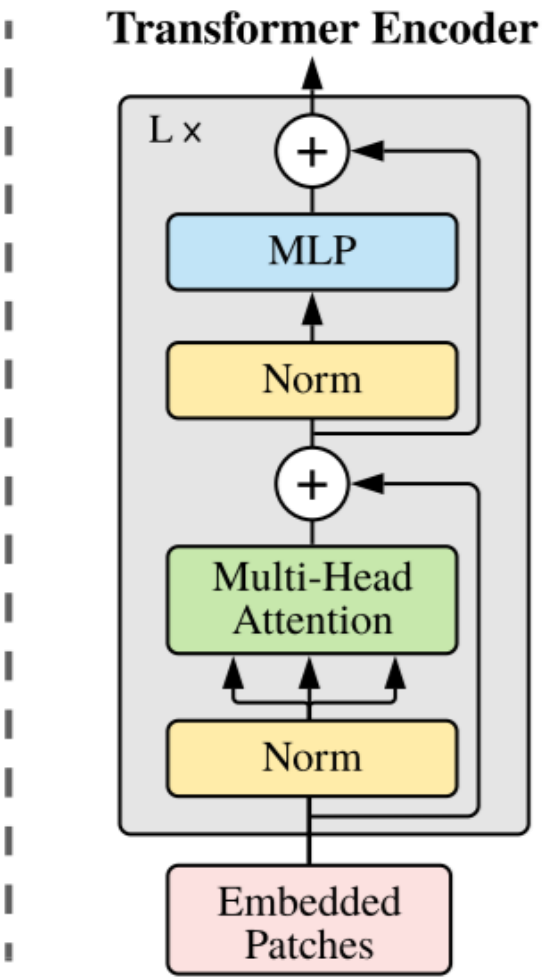
Dosovitskiy, Alexey, et al. "An image is worth 16x16 words: Transformers for image recognition at scale." *arXiv,* 2020..

مبدل بینایی (Vision-Transformer)

Dosovitskiy, Alexey, et al. "An image is worth 16x16 words: Transformers for image recognition at scale." *arXiv,* 2020..

مبدل بینایی (Vision-Transformer)

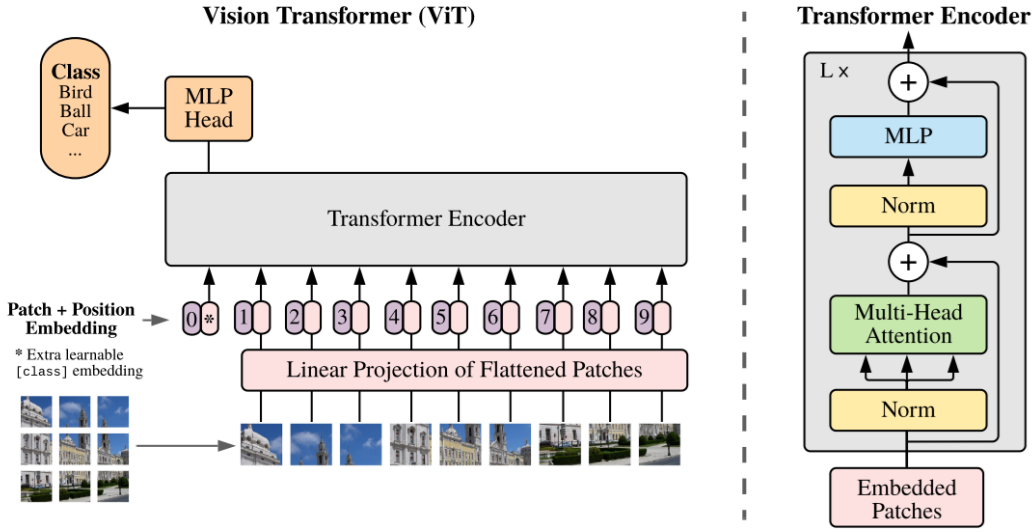




Dosovitskiy, Alexey, et al. "An image is worth 16x16 words: Transformers for image recognition at scale." *arXiv,* 2020..

مبدل بینایی (Vision-Transformer)

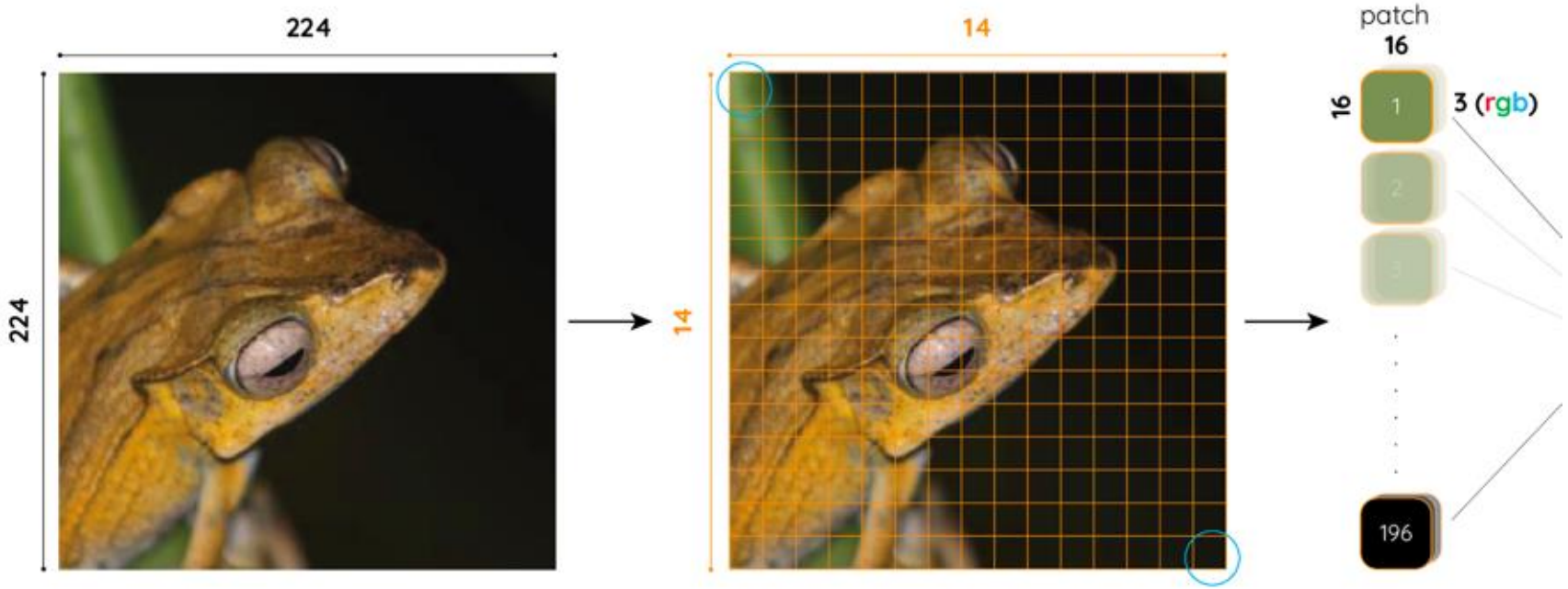
- تصویر به تکه‌های با اندازه ثابت تقسیم می‌شود
- هر تکه به صورت خطی جانمایی می‌شود (patch embeddings)
- جانمایی موقعیت به آن افزوده می‌شود
- دنباله بردارهای حاصل به یک کدگذار مبدّل استاندارد وارد می‌شود
- با GAP ترکیب می‌شوند
- توسط MLP دسته‌بندی انجام می‌شود



Dosovitskiy, Alexey, et al. "An image is worth 16x16 words: Transformers for image recognition at scale." *arXiv,* 2020..

جانمایی تکه‌ها

- یک تصویر با ابعاد $3 \times 224 \times 224$ به $196 = 14 \times 14$ تکه با ابعاد 16×16 تقسیم می‌شود
  - نتیجه یک بردار 196×768 خواهد بود
- با یک لایه خطی به فضای جدید تبدیل می‌شوند

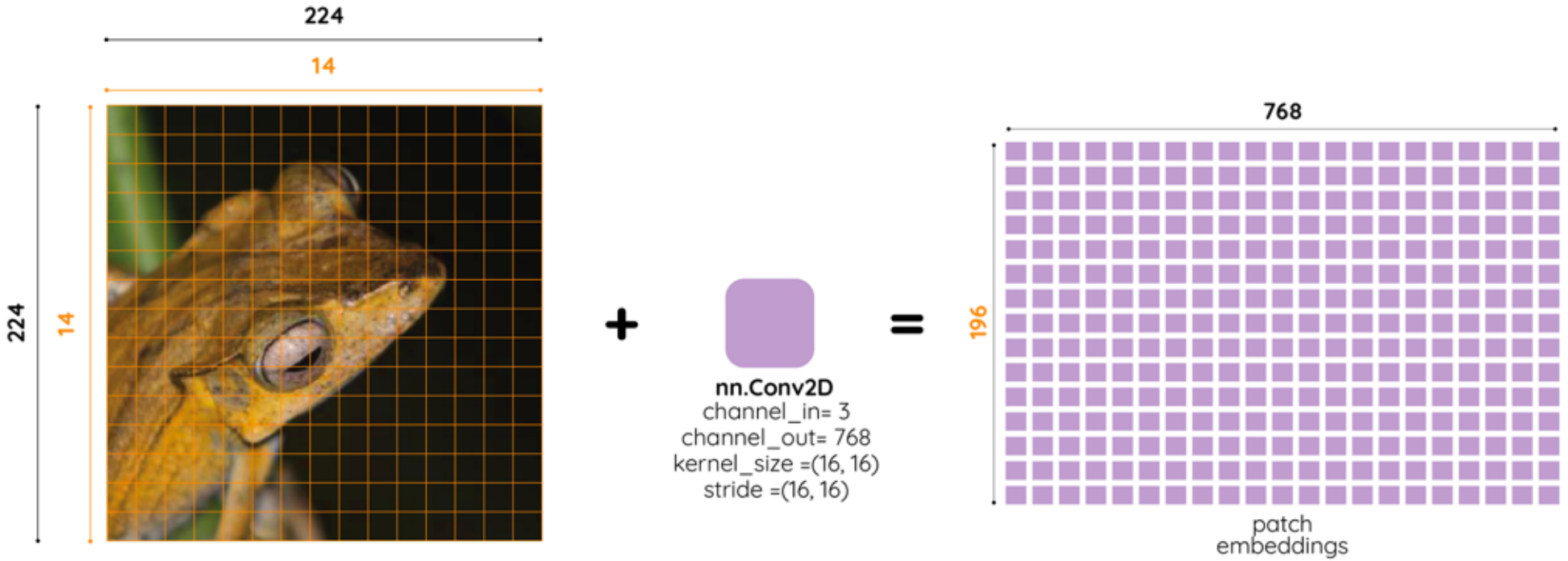


https://tugot17.github.io/Vision-Transformer-Presentation/#/10

جانمایی تکه‌ها

- برای پیاده‌سازی می‌توان از یک لایه کانولوشنی استفاده کرد

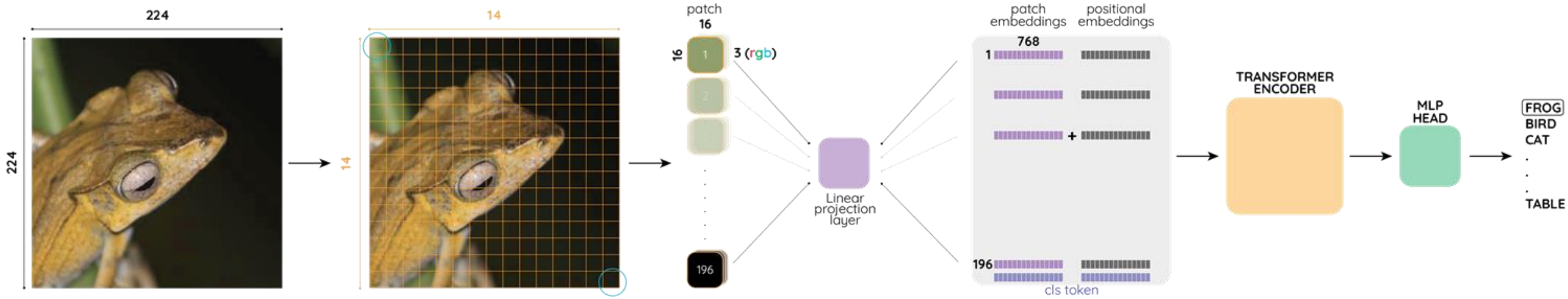
```
x = torch.randn(1, 3, 224, 224)
# 2D conv
conv = nn.Conv2d(3, 768, 16, 16)
conv(x).reshape(-1, 196).transpose(0,1).shape
>> torch.Size([196, 768])
```



https://tugot17.github.io/Vision-Transformer-Presentation/#/12

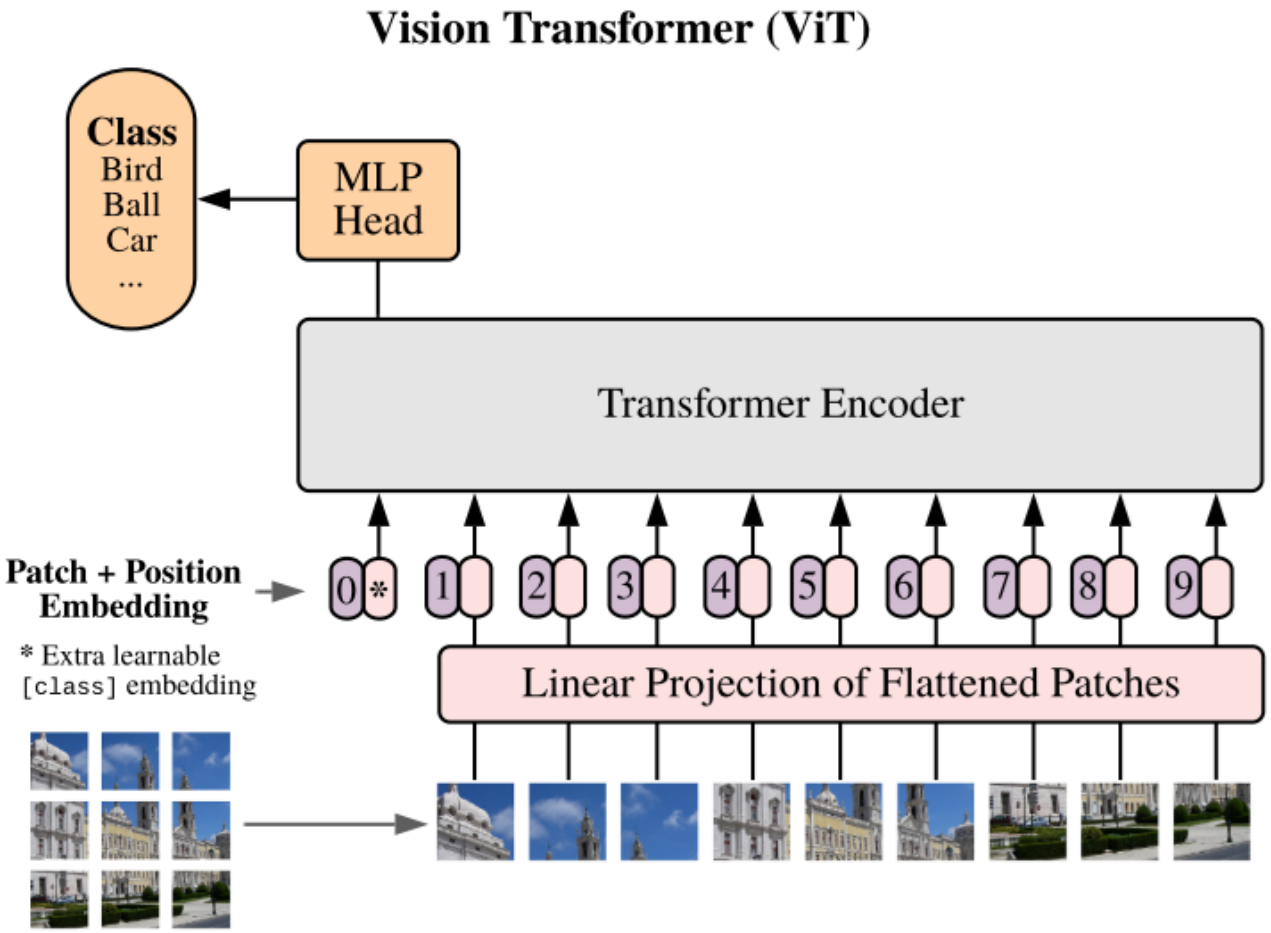
مبدل بینایی (Vision-Transformer)

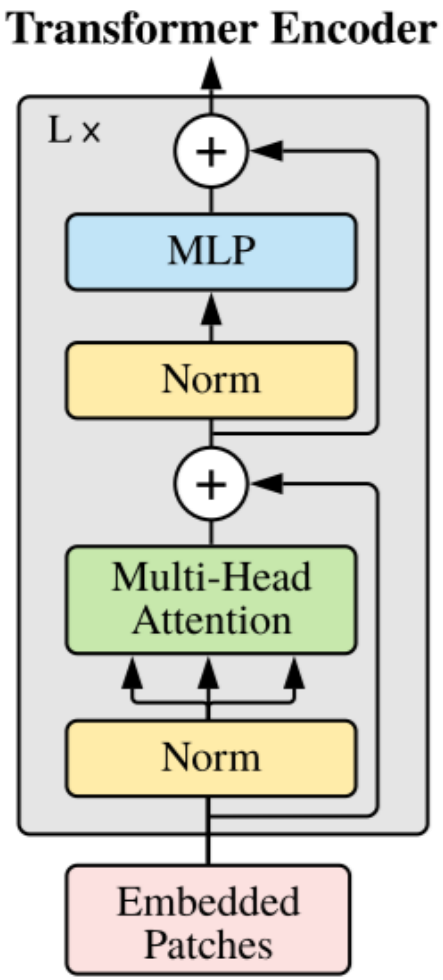
- برای جانمایی موقعیت از روش‌های قابل یادگیری استفاده شده است
- مجموع این دو جانمایی وارد کدگذار مبدّل می‌شود
- خروجی کدگذار وارد یک MLP برای دسته‌بندی می‌شود



Dosovitskiy, Alexey, et al. "An image is worth 16x16 words: Transformers for image recognition at scale." *arXiv,* 2020..

مبدل بینایی (Vision-Transformer)





Dosovitskiy, Alexey, et al. "An image is worth 16x16 words: Transformers for image recognition at scale." *arXiv,* 2020..

```
class ViT(pl.LightningModule):
    def __init__(self, num_transformer_layers, num_classes=1000):
        super().__init__()
        self.criterion = nn.CrossEntropyLoss()

        self.conv_embedding = nn.Conv2d(3, 768, 16, 16)

        self.cls_token = nn.Parameter(torch.zeros(1, 1, embed_dim))

        encoder_layer = nn.TransformerEncoderLayer(d_model=768, nhead=8)
        self.transformer_encoder = nn.TransformerEncoder(encoder_layer, num_layers=num_transformer_layers)

        self.mlp_head = nn.Linear(768, num_classes)

        self.position_embedding_layer = nn.Embedding(197, 768)


    def training_step(self, batch, batch_idx):
        x, y = batch
        logits = self.forward(x)
        loss = self.criterion(logits, y)
        return loss
```


https://tugot17.github.io/Vision-Transformer-Presentation/#/19\

```python
class ViT(pl.LightningModule):
    def forward(self, x):
        batch_size = x.shape[0]
        cls_tokens = self.cls_token.expand(batch_size, -1, -1)

        #(batch_size, 196, 768)
        patches_embedding = self.conv_embedding(x).reshape(-1, 196).transpose(0,1)

        #(batch_size, 197, 768)
        patches_embedding = torch.cat((cls_tokens, patches_embedding), dim=1)

        #(batch_size, 197); 0, 0, ... 196, 196
        positions  = self._assign_positions_to_patches(

        #(batch_size, 197, 768)
        position_embedding = position_embedding_layer(positions)

        #(batch_size, 197, 768)
        final_embedding = patches_embedding + position_embedding

        #(batch_size, 197, 768)
        embedding_output = self.transformer_encoder(final_embedding)

        #(batch_size, 768)
        cls_vector = embedding_output[:, 0, :]

        #(batch_size, num_classes)
        return mlp_head(cls_vector)
```


https://tugot17.github.io/Vision-Transformer-Presentation/#/19\

مبدل بینایی (Vision-Transformer)

- اگر با وزن‌های تصادفی روی ImageNet آموزش ببیند عملکرد ضعیف‌تری از ResNet دارد
- هنگامیکه بر روی مجموعه داده عمومی ImageNet-21k پیش‌آموزش ببیند هم عملکرد ضعیف‌تری دارد
- اما هنگامیکه بر روی مجموعه داده بزرگ JFT-300M پیش‌آموزش ببیند، عملکرد بهتری دارد
- پیش‌آموزش ViT بسیار پر هزینه است
  - در حد ۳۰ هزار دلار!
  - برخی وزن‌های پیش‌آموخته از این لینک قابل دانلود است

Model	Layers	Hidden size D	MLP size	Heads	Params
ViT-Base	12	768	3072	12	86M
ViT-Large	24	1024	4096	16	307M
ViT-Huge	32	1280	5120	16	632M

	Ours-JFT (ViT-H/14)	Ours-JFT (ViT-L/16)	Ours-I21k (ViT-L/16)	BiT-L (ResNet152x4)
ImageNet	**88.55** ± 0.04	87.76 ± 0.03	85.30 ± 0.02	87.54 ± 0.02
ImageNet ReaL	**90.72** ± 0.05	90.54 ± 0.03	88.62 ± 0.05	90.54
CIFAR-10	**99.50** ± 0.06	99.42 ± 0.03	99.15 ± 0.03	99.37 ± 0.06
CIFAR-100	**94.55** ± 0.04	93.90 ± 0.05	93.25 ± 0.05	93.51 ± 0.08
Oxford-IIIT Pets	**97.56** ± 0.03	97.32 ± 0.11	94.67 ± 0.15	96.62 ± 0.23
Oxford Flowers-102	99.68 ± 0.02	**99.74** ± 0.00	99.61 ± 0.02	99.63 ± 0.03
VTAB (19 tasks)	**77.63** ± 0.23	76.28 ± 0.46	72.72 ± 0.21	76.29 ± 1.70
TPUv3-core-days	2.5k	0.68k	0.23k	9.9k

Dosovitskiy, Alexey, et al. "An image is worth 16x16 words: Transformers for image recognition at scale." *arXiv,* 2020..