به نام خدا

# درس یادگیری عمیق

## تمرین ششم

غزل زمانی نژاد

401722244

1. الف) شبکه کانولوشنی یک فیلتر با وزن های مشخص را بر روی تصویر اعمال می کند و وزن های این فیلتر برای تمامی نواحی تصویر مشترک هستند. به همین دلیل در مقایسه با شبکه مبتنی بر توجه، به تعداد پارامترهای کمتری نیاز دارد. با این وجود، ممکن است نیازی نباشد به تمامی نواحی تصویر بدون توجه به محتوای آن به یک شکل نگاه کنیم. مثلا ممکن است حاشیه تصویر اطلاعات خاصی در بر نداشته باشد. در صورت استفاده از شبکه مبتنی بر توجه، یاد می گیرد که به کدام قسمت ها بیشتر توجه کند. همچنین، فیلتر شبکه کانولوشنی در بعد طول و عرض بر تصویر اعمال می شود در صورتی که در روش هایی از شبکه مبتنی بر توجه، توجه کانالی و مکانی (spatial) اعمال می شود.

   شبکه کانولوشنی به صورت محلی به تصویر نگاه می کند و بدنبال استخراج این نوع از ویژگی هاست در حالیکه در شبکه مبتنی بر توجه، هر پیکسل از تصویر به تمامی پیکسل ها مربوط است و به صورت global به آن نگاه می کند (تقریبا شبیه به لایه fully-connected).

   شبکه کانولوشنی بیشتر قابل تفسیر و دبیاگ است زیرا می توانیم فیلترهای آموخته شده و همچنین خروجی لایه ها را رسم کنیم اما وزن های شبکه مبتنی بر توجه چندان قابل تفسیر نیستند. البته رسم نقشه توجه می تواند به ما نشان دهد شبکه به چه بخش هایی توجه بیشتری داشته.

   ب) شبکه کانولوشنی نسبت به چرخش invariance نیست اما translation invariance است و در صورت استفاده از لایه ادغام، اطلاعات رابطه بین قسمت های مختلف تصویر از بین می رود. با توجه به اینکه در این تصویر چشم، ابرو، و ... وجود دارد (با قرار گرفتن در جای اشتباه) اما برخی اعضا مثل لب و بینی دچار چرخش شده اند، به آن برچسب چهره نمی زند.

   شبکه مبتنی بر توجه به این تصویر برچسب چهره نخواهد زد. مکانیسم توجه وزن های توجه را جوری محاسبه می کند که اهمیت یا ارتباط هر موقعیت را با توجه به سایر موقعیت های تصویر تعیین کند. در هنگام آموزش تصویر را به دنباله ای از ویژگی ها تبدیل می کند و با اضافه کردن positional embedding ارتباط میان قسمت ها را نیز سعی می کند یاد بگیرد. در صورتی که از positional embedding استفاده نشود، به اشتباه به تصویر برچسب چهره می زند.

. .2

$n_{head} = 3, \quad d_k = 10 \quad d_q = 20 \quad d_v = 30 \quad d_h = 100$

$d_o = 50, \quad l_{Sequence} = 64$

head برای : $a(q,k) = w_v^T \tanh(W_q q + W_k k)$

$W_q \to 100 \times 20 \quad W_k \to 100 \times 10 \quad W_v \to 100 \times 30$

head 3 برای : $3(100 \times 20 + 100 \times 10 + 100 \times 30) = 18000$

$O = W_o \begin{bmatrix} h_1 \\ \vdots \\ h_h \end{bmatrix} \quad W_o \to 50 \times 3 \times 100 = 15000$

$\Rightarrow total = 18000 + 15000 = \boxed{33000}$

. .3

$128 \times 128 \times 128 \times 4 \xrightarrow[patch\ 16\times16\times16]{} 8 \times 8 \times 8 \times 4$

تبدیل به $8^3 = 512$ تکه می‌شود. هرکدام $16 \times 16 \times 16 \times 4 = 16384$ درایه دارند

$[512, 2^{14}]$ برود $\xrightarrow[h=768]{} [512, 768]$

$\Rightarrow$ positional dimension $= [512, 768]$

البته با احتساب cls_token ابعاد $[513, 768]$ خواهد شد.

4. الف) چالش هایی که در استفاده از ترنسفورمرها (که در ابتدا برای تسک های زبان معرفی شدند) در تسک های بینایی وجود دارد، مربوط به تفاوت حوزه های زبان و بینایی است. مثلا تغییرات بزرگ در entity های بصری و وضوح بالای پیکسل ها در تصاویر در مقایسه با کلمات در متن. برای حل این مشکلات، مکانیزم shifted windows را معرفی کرده اند.

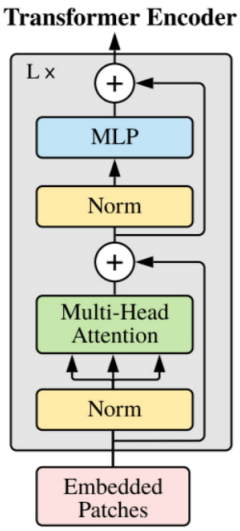
ب) ایده اصلی مقاله swin، استفاده از window based multi-head self attention بجای multi-head self attention است. در ماژول MSA، رابطه میان یک توکن با همه توکن های دیگر محاسبه می شود این نوع محاسبات global، منجر به پیچیدگی درجه دوم می شود که با توجه به تعداد توکن ها ممکن است برای بسیاری از مسائل بینایی -که نیازمند مجموعه زیادی از توکن ها برای پیش بینی یا برای نمایش یک تصویر با رزولوشن بالا هستند- مناسب نباشد. برای اینکه محاسبات کارآمدتر واقع شوند از توجه به خود در یک پنجره محلی استفاده می شود. پنجره ها طوری چیده شده اند که تصویر به صورت غیر همپوشانی به طور مساوی تقسیم شود. اگر هر پنجره شامل M * M پچ باشد و تصویر h * w پچ باشد، پیچیدگی محاسباتی این دو ماژول به صورت زیر است:

$$\Omega(\text{MSA}) = 4hwC^2 + 2(hw)^2C,$$
$$\Omega(\text{W-MSA}) = 4hwC^2 + 2M^2hwC,$$

دلیل استفاده از W-MSA این است که در صورت ثابت بودن M، پیچیدگی آن خطی خواهد بود اما در MSA، درجه دوم است که در صورت بزرگ بودن مقدار h * w ممکن است عملی نباشد.

ج) در مکانیزم W-MSA میان پنجره ها ارتباطی وجود ندارد که این باعث محدود شدن قدرت محاسباتی می شود. استفاده از ماژول SW-MSA باعث می شود بین همسایه هایی که در لایه قبلی همپوشانی نداشتند ارتباط برقرار شود و در تسک های دسته بندی تصاویر، تشخیص شی و تقسیم بندی معنایی موثر واقع می شود.

5. در بخش modeling، ابتدا EncoderBlock را تعریف می کنیم. برای این کار، مطابق تصویر زیر ابتدا از یک لایه نرمال سازی استفاده می کنیم و خروجی آن را به توجه به خود چند سر می دهیم. سپس ورودی اولیه را با خروجی مالتی هد جمع می کنیم (لایه add) تا مقادیر لایه های قبل فراموش نشود و دوباره از لایه نرمال سازی و MLP و add استفاده می کنیم.

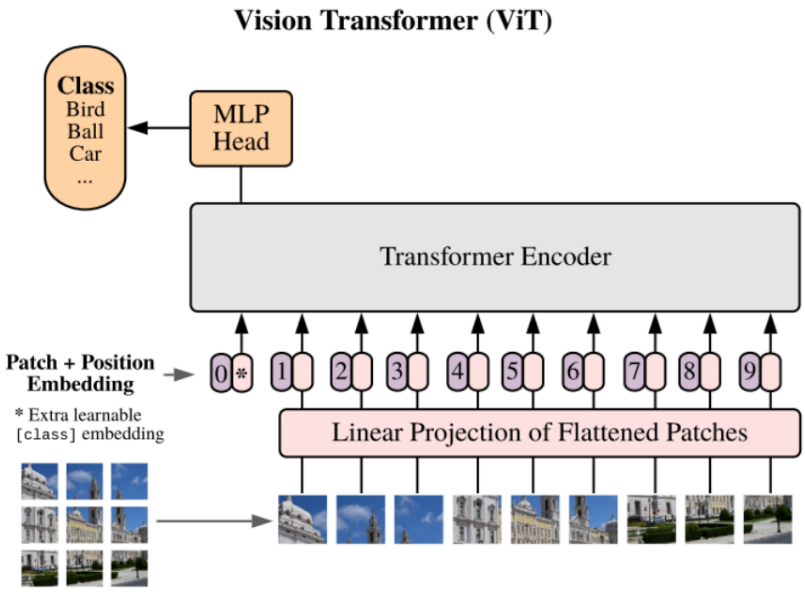



```python
class EncoderBlock(nn.Module):

    def __init__(self, num_heads, d_model, d_feedforward, dropout=0.0):
        super().__init__()
        self.num_heads = num_heads
        # Fill in the missing modules.
        self.ln_1 = nn.LayerNorm(d_model)
        self.self_attention = nn.MultiheadAttention(d_model, num_heads, dropout, batch_first=True)
        self.dropout = nn.Dropout(dropout)
        self.ln_2 = nn.LayerNorm(d_model)
        # self.mlp = ops.MLP(d_model, [d_feedforward, d_model])
        self.mlp = nn.Sequential(
            nn.Linear(d_model, d_feedforward),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(d_feedforward, d_model),
            nn.Sigmoid()
        )

    def forward(self, inputs):
        # Your code goes here.
        # You need to output the encoded values and the attention weights.

        x = self.ln_1(inputs)
        x, W = self.self_attention(x, x, x)
        x = self.dropout(x)
        x = x + inputs
        y = self.ln_2(x)
        y = self.mlp(y)

        return x + y, W
```

در سلول encoder، باید positional embedding را به امبدینگ ورودی اضافه کنیم. سپس از لایه dropout عبور می دهیم. بعد از تک تک لایه های موجود در layers عبور می دهیم که هرکدام خود یک EncoderBlock است.

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

    def __init__(self, seq_length, num_layers, num_heads, d_model, d_feedforward, dropout=0.0):
        super().__init__()
        self.pos_embedding = nn.Parameter(torch.randn(1, seq_length, d_model).normal_(std=0.02))
        # Fill in the missing modules.
        self.dropout = nn.Dropout(dropout)
        self.layers = nn.ModuleList()
        for i in range(num_layers):
            layer = EncoderBlock(num_heads, d_model, d_feedforward, dropout)
            self.layers.append(layer)
        self.ln = nn.LayerNorm(d_model)

    def forward(self, inputs):
        # Your code goes here.
        # You need to output the encoded values and a list of attention weights.

        x = self.pos_embedding + inputs
        x = self.dropout(x)

        W = [None] * len(list(self.layers))
        for i, blk in enumerate(self.layers):
            x, attn_output_weights = blk(x)
            W[i] = attn_output_weights

        x = self.ln(x)
        return x, W
```

در سلول ViT، ابتدا باید تصویر را پردازش کنیم. یعنی با عبور از لایه کانولوشنی آن را به شکل مورد نظر (patchهای مختلف) تبدیل کنیم. سپس cls_token هم به امبدینگ patchها اضافه می کنیم. آنها را از کدگذار (که قبلا تعریف کردیم) عبور می دهیم. و در نهایت برای طبقه بندی، از لایه head که یک لایه کاملا متصل است استفاده می کنیم.




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

    def __init__(self, image_size, patch_size, num_classes, num_layers, num_heads, d_model, d_feedforward, dropout=0.0):
        super().__init__()
        self.patch_size = patch_size
        seq_length = (image_size // patch_size) ** 2
        # Fill in the missing modules.
        self.conv_proj = nn.Conv2d(3, d_model, kernel_size=patch_size, stride=patch_size)
        self.encoder = Encoder(seq_length+1, num_layers, num_heads, d_model, d_feedforward, dropout)
        self.head = nn.Linear(d_model, num_classes)
        self.class_token = nn.Parameter(torch.zeros(1, 1, d_model))

    def process_input(self, x):
        x = self.conv_proj(x)
        x = x.flatten(2, 3)
        x = x.permute(0, 2, 1)
        return x

    def forward(self, x, need_weights=False):
        # Your code goes here.
        # You need to output the encoded values and a list of attention weights.
        batched_cls_token = self.class_token.expand(x.size(0), -1, -1)
        patches_embedding = self.process_input(x)
        patches_embedding = torch.cat((batched_cls_token, patches_embedding), dim=1)

        x, W = self.encoder(patches_embedding)
        x = x[:, 0]
        x = self.head(x)

        return (x, W) if need_weights else x
```

مدل را با وزن های pre-trained مقداردهی اولیه می کنیم و سپس در ۵ ایپاک آن را آموزش می دهیم.

دقت مدل در پایان آموزش به ۷۴ درصد می رسد. مدل underfit است. برای دستیابی به نتیجه بهتر،

می توانیم مدل را برای تعداد ایپاک بیشتری آموزش دهیم.

```
1 trainer.run(train_loader, max_epochs=5)
```

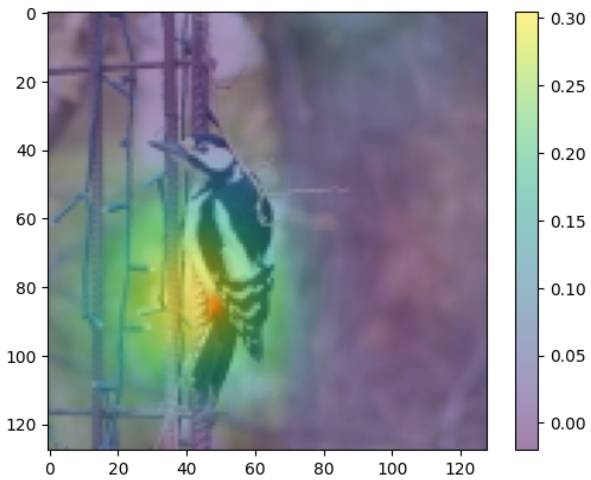
```
accuracy: 0.51
accuracy: 0.57
accuracy: 0.65
accuracy: 0.69
accuracy: 0.74
State:
        iteration: 1875
        epoch: 5
        epoch_length: 375
        max_epochs: 5
        output: 1.6819865703582764
        batch: <class 'list'>
        metrics: <class 'dict'>
        dataloader: <class 'torch.utils.data.dataloader.DataLoader'>
        seed: <class 'NoneType'>
        times: <class 'dict'>
```

برای نمونه تصویر پرنده، عکس را به شبکه می دهیم و مقدار logit, weight را می گیریم و از تابع فعال سازی sigmoid استفاده می کنیم. سپس کلاس ورودی را حساب می کنیم.

```
1 # Pass the logits through sigmoid and get the index of the largest score.
2 sig = nn.Sigmoid()
3 output = sig(logits)
4 prediction = torch.argmax(output)
5 print(prediction)
```

```
tensor(4, device='cuda:0')
```

در نهایت وزن هایی که مدل آموخته را بر روی تصویر می اندازیم تا ببینیم مدل برای پیش بینی به چه نقاطی از تصویر توجه کرده است.



با وجود اینکه پیش بینی برچسب مدل نادرست بوده است، اما مدل به بخش های درستی توجه کرده است. در صورتی که آموزش مدل را ادامه دهیم، احتمالا مدل برچسب صحیح را پیش بینی خواهد کرد.

**منابع**


https://ai.stackexchange.com/questions/25099/what-is-the-difference-between-attention-gate-and-cnn-filters

https://towardsdatascience.com/what-is-wrong-with-convolutional-neural-networks-75c2ba8fbd6f