
# یادگیری عمیق

تمرین ششم

استاد درس: دکتر محمدی

مهسا موفق بهروزی

بهار ۱۴۰۲

## سوال یک)

**الف)** شبکه‌های عصبی کانولوشن (CNN) بیشتر برای پردازش داده‌های grid مانند، مثل تصاویر، با استفاده از لایه‌های کانولوشن و pooling طراحی شده‌اند. این شبکه‌ها از فیلترهای کانولوشنال برای استخراج ویژگی‌های محلی به صورت سلسله مراتبی استفاده می‌کنند و می‌توانند لبه‌ها، shape،texture و باقی الگوهای سطح پایین را استخراج کرده و با ترکیب آن‌ها ویژگی‌های سطح بالاتری ایجاد کنند. همچنین CNNها، translation invariant هستند؛ یعنی می‌توانند ویژگی‌ای را بدون توجه به موقعیت مکانی‌اش در تصویر تشخیص دهند و نسبت به جابه‌جایی invariant اند. CNNها در ثبت الگوهای محلی و کارهایی مانند image classification، تشخیص اشیا و ... بسیار خوب عمل می‌کنند.

شبکه‌های مبتنی بر توجه، بر روی وابستگی‌های global و روابط contextual در ورودی تمرکز می‌کنند. این شبکه‌ها به بخش‌های مختلف ورودی، وزن اختصاص می‌دهند و شبکه یاد می‌گیرد به اطلاعات و بخش‌های مرتبط توجه کند. این ویژگی باعث می‌شود این شبکه‌ها ویژگی‌های معنایی سطح بالا را ثبت کنند و زمینه کلی یک تصویر را درک کنند. این شبکه‌ها، برای کارهایی مانند image captioning، visual question answering و image generation مناسب‌اند. همچنین به نظر من شبکه‌های مبتنی بر توجه، از آن‌جایی که با اختصاص وزن به بخش‌های مختلف، مشخص می‌کنند به چه چیزی توجه کرده‌اند از قابلیت تفسیر بالاتری نسبت به CNNها برخوردارند.

**ب)** شبکه‌های CNN می‌توانند با استفاده از pooling نسبت به جابه‌جایی‌های کوچک حساسیت کمتری داشته باشند؛ همچنین می‌دانیم فیلترهای کانولوشن در فضایی به ابعاد فیلتر به دنبال الگوهای خاصی میگردند. بنابراین ممکن است پترن چشم و ابرو را که صرفا جابجا شده‌اند، تشخیص دهند. اما در این تصویر موقعیت فیچرها نسبت به یک‌دیگر تغییر کرده است که بیش از یک تغییرات جزیی‌ست. همچنین CNNها نسبت به rotation حساسند؛ به این معنی که اگر اجزای صورت را rotate کنیم، نمی‌توانند آن فیچر را شناسایی کنند. مثلا در این تصویر لب و بینی rotate شده‌اند که CNN قادر به تشخیص آن نخواهد بود. البته شاید بتوان با استفاده از data augmentation، با آموزش مدل توسط تصاویری که rotate شده‌اند، مدل را نسبت به rotation، روباست کرد. - در این مثال باید اجزای صورت را rotate کنیم تا نسبت به آن روباست شود-.

بنابراین به نظر من CNN این تصویر را به عنوان چهره انسان تشخیص نخواهد داد.

شبکه‌های مبتنی بر توجه که بیشتر به وابستگی‌های global توجه می‌کنند، با احتمال بیشتری امکان دارد به عنوان تصویر چهره پیش‌بینی کنند.

## سوال دو)

سؤال ۲)

key dimension = 10

query dim = 20

value dim = 30

desired dim for a head = 100

num of heads = 3

Final output = 50

input seq. = 64

$$a(q,k) = w_r^T \tanh(w_q q + w_k k)$$

تعداد پارامترهای برای یک head $= 100 \times 30 + 100 \times 20 + 100 \times 10 = 6000$

برای 3 $\sim$ $3 \times 6000 = 18000$

$$O = w_o \begin{bmatrix} h_1 \\ \vdots \\ h_n \end{bmatrix}$$

تعداد پارامترهای لایه‌ی نهایی $= 50 \times 3 \times 100 = 15000$

تعداد کل پارامترها $= 18000 + 15000 = 33000$

## سوال سه)

سؤال ۳)

input-size $128 \times 128 \times 128$

patch-size $16 \times 16 \times 16$ $\longrightarrow 8 \times 8 \times 8$

$8^3 = 512$ بخش، هر بخش $16 \times 16 \times 16 \times 4$ رنگی

چون ابعاد لایه‌ی مخفی = 768 $\geq$ positional emb. برداری با ابعاد [512, 768] است.

## سوال چهار)

**الف)** این معماری که Swin Transformer نام دارد، برای حل چالش انطباق (adapting) معماری transformer برای استخراج ویژگی‌هایی بصری و ارائه یک backbone همه منظوره برای task های مختلف بینایی مانند object detection و image classification و ... می‌باشد. چالش اصلی در انتقال کارایی transformerها از حوزه زبانی به حوزه بینایی، تفاوت در مقیاس و وضوح (resolution) است. Swin transformer این چالش‌ها را با ساختن فیچر مپ‌های سلسله مراتبی و محاسبه‌ی self-attention به صورت محلی در تصویری که توسط windoiwهایی که با یکدیگر همپوشانی ندارند تقسیم شده است، برطرف می‌کند.

**ب)** تفاوت بین بلوک‌های MSA و W-MSA در swin transformer در نحوه‌ی محاسبه‌ی self-attention این دو می‌باشد. MSA توجه به خود را به صورت global، به طوری که رابطه یک توکن با همه‌ی توکن‌های دیگر محاسبه می‌شود، محاسبه می‌کند؛ درحالی‌که W-MSA توجه به خود را در پنجره‌های محلی که تصویر را به نحوی که همپوشانی نداشته باشد تقسیم می‌کنند، محاسبه می‌کند. محاسبات global منجر به پیچیدگی درجه دوم، با توجه به تعداد توکن‌ها، می‌شود و برای بسیاری از مسائل بینایی که به مجموعه‌ی وسیعی از توکن‌ها برای پیش‌بینی نیاز دارند، نامناسب است. Swin transformer بلوک‌های W-MSA را برای مقابله با چالش انطباق معماری transformer برای استخراج ویژگی‌های بصری معرفی می‌کند. این بلوک‌ها swin transformer را قادر می‌سازند تا عناصر بصری در مقیاس‌های مختلف و تصاویر با وضوح بالا را پردازش کنند.

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

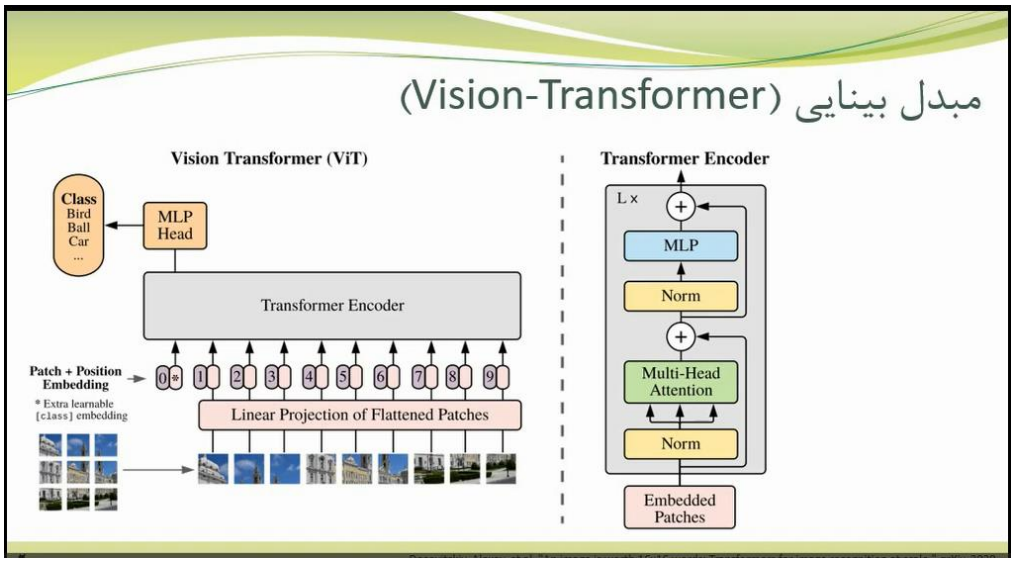
با توجه به پیچیدگی محاسباتی این دو، با فرض اینکه هر پنجره شامل M * M پچ و هر تصویر h * w پچ باشد، پیچیدگی ماژول MSA با توجه به h * w از توان دو است، در صورتی‌که پیچیدگی W-MSA با ثابت گرفتن M (که به صورت پیشفرض ۷ در نظر گرفته شده است) خطی‌ست.

ج) ماژول SW-MSA که improvementای از ماژول W-MSA می‌باشد، برای نشان دادن اتصالات بین پنجره‌های غیرهمپوشان همسایه در لایه قبلی معرفی شده است. ماژول W-MSA فاقد اتصالات بین پنجره‌ها می‌باشد و این مساله قدرت مدل‌سازی آن را محدود می‌کند.

## سوال پنج)

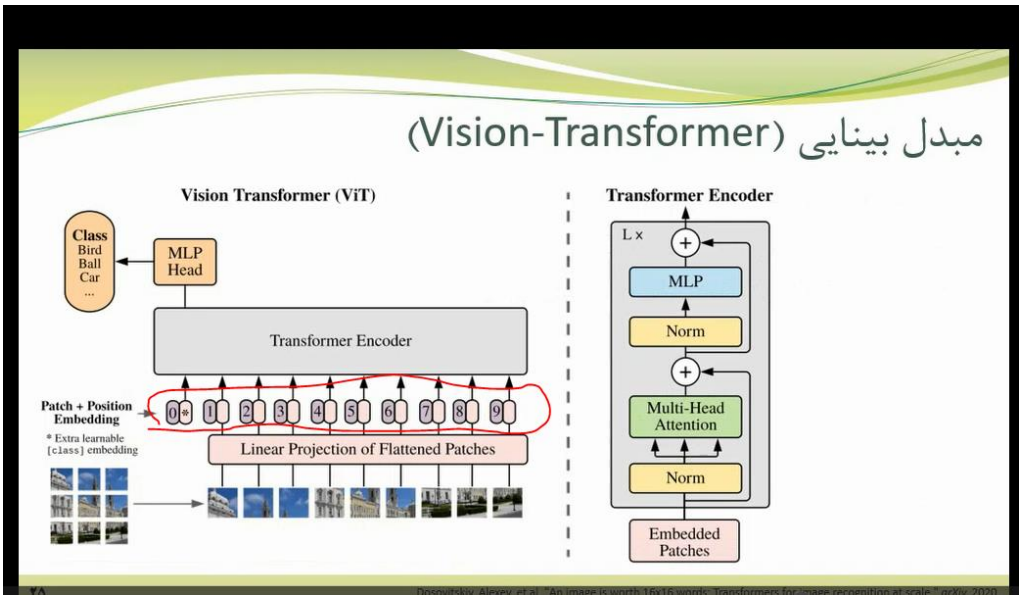
```
[ ] class EncoderBlock(nn.Module):

        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 = None
            # self.self_attention = None
            # self.dropout = None
            # self.ln_2 = None
            # self.mlp = None

            self.ln_1 = nn.LayerNorm(d_model)
            self.self_attention = nn.MultiheadAttention(d_model, num_heads, batch_first=True)
            self.dropout = nn.Dropout(dropout)
            self.ln_2 = nn.LayerNorm(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.
            #torch._assert(input.dim() == 3, f"Expected (batch_size, seq_length, hidden_dim) got {input.shape}")
            x = self.ln_1(inputs)
            x, W = self.self_attention(x, x, x, need_weights=True)
            x = self.dropout(x)
            x = x + inputs
            y = self.ln_2(x)
            y = self.mlp(y)
            return x + y, W
```

در بخش انکدر با توجه به مطالب گفته شده در کلاس (تصویر زیر) ساختار به ترتیب حاصل توالی یک لایه‌ی نرمال‌سازی و یک لایه‌ی اتنشن و سپس جمع ورودی اولیه با حاصل آن و مجدد یک لایه‌ی نرمال‌سازی دیگر و یک mlp می‌باشد و نهایتا مقدار حاصل به mlp با مقدار حاصل از جمع اولیه جمع می‌شود. بخش mlp را با استفاده از یک لایه‌ی Sequential و توالی دو لایه‌ی Linear و یک activation function و یک dropout و نهایتا یک لایه‌ی sigmoid ایجاد کرده‌ایم و از لایه mlp خود پایتورچ استفاده نشده.




```
[13] 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 = None
            # self.layers = nn.ModuleList()
            # for i in range(num_layers):
            #     layer = None
            #     self.layers.append(layer)
            # self.ln = None

            self.dropout = nn.Dropout(dropout)
            self.layers = nn.ModuleList()
            for i in range(num_layers):
              self.layers.append(EncoderBlock(
                  num_heads,
                  d_model,
                  d_feedforward,
                  dropout,
              ))
            self.ln = nn.LayerNorm(d_model)

        def forward(self, inputs):
            inputs = inputs + self.pos_embedding
            out = self.dropout(inputs)
            _out = []
            for idx,module in enumerate(self.layers):
              if type(out) is tuple:
                out = module(out[0])
                _out.append(out[1])
              else:
                out = module(out)
                _out.append(out[1])
            return self.ln(out[0]), _out
            # return x, W
```

در این قسمت ورودی‌های حاصل از مدل ترنسفور (بخش بعدی) با یک تگ حاصل از امبدینگ جمع می‌شود.
(مشابه تصویر زیر)

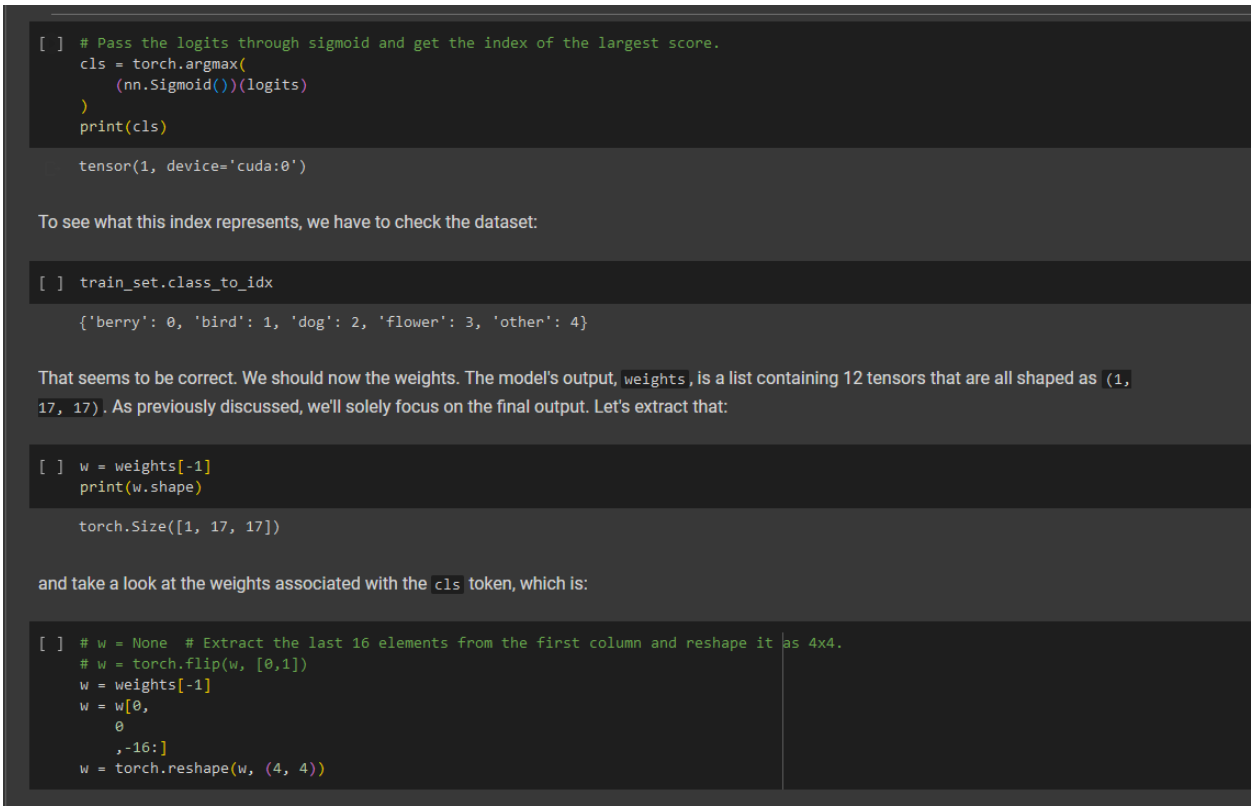



و سپس به قسمت ترنسفور که شامل چند لایه ی انکدور است می‌رود. با توجه به اینکه ورودی در مرحله‌ی اول فاقد پارامتر وزن است و یک تنسور است اما در مراحل بعدی با توجه به خروجی انکودرها (که تاپلی از خروجی و وزن اتنشن‌ها می‌باشد) شرطی قرار داده شده که در مرحله‌ی اول که ورودی تاپل نیست خود پارامتر out و در مرحله بعدی عنصر اول پارامتر out (که نتیجه‌ی شبکه‌ی انکودر هست) ارسال شود. نتایج وزن‌های حاصل از هر مرحله‌ی انکودر هم در پارامتر out_ که لیستی از تنسورها هست ذخیره می‌شود که نهایتا شامل ۱۲ درایه که هر کدام یک تنسور است می‌باشد (۱۲ تعداد لایه های انکودر می‌باشد.)

```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 = None
        # self.encoder = None
        # self.head = None
        # self.class_token = nn.Parameter(torch.zeros(1, 1, d_model))

        self.conv_proj = nn.Conv2d(
                in_channels=3, out_channels=d_model, kernel_size=patch_size, stride=patch_size
        )
        seq_length += 1
        self.encoder = Encoder(
            seq_length,
            num_layers,
            num_heads,
            d_model,
            d_feedforward,
            dropout,
        )
        self.seq_length = seq_length
        self.head =  nn.Linear(d_model, num_classes)
        self.class_token = nn.Parameter(torch.zeros(1, 1, d_model))


    def _process_input(self, x): #change process_input to _process_input
        x = self.conv_proj(x)
        x = x.flatten(2, 3)
        x = x.permute(0, 2, 1)
        return x
```

```python
    def forward(self, x, need_weights=False):
        # Your code goes here.
        # You need to output the encoded values and a list of attention weights.
        x = self._process_input(x)
        n = x.shape[0]
        batch_class_token = self.class_token.expand(n, -1, -1)
        x = torch.cat([batch_class_token, x], dim=1)
        x = self.encoder(x)
        if need_weights:
          W = x[1]
          x = x[0]
          x = x[:, 0]
          x = self.head(x)
          # return x
          return x, W
        else:
          x = x[0]
          x = x[:, 0]
          x = self.head(x)
          # return x
          return x
          pass
```

در این بخش ابتدا seq_length را محاسبه می‌کنیم. این عدد تعداد patch های حاصل در هر image می‌باشد که برابر با سایز عکس تقسیم بر سایز patch به توان ۲ است. باید توجه داشت که patch ها با هم هم‌پوشانی ندارند. در قسمت بعد از یک لایه‌ی کانولوشن استفاده می‌کنیم. این لایه شامل ۳ کانال (RGB) است که کرنل سایز آن معادل سایز هر patch که اینجا مقدار آن ۳۲ است می‌باشد و مقدار stride برابر همان ۳۲ است تا patch ها با هم هم‌پوشانی نداشته باشند. نهایتا نتیجه‌ی حاصل از کانولوشن‌ها برای لایه‌ی انکودر فرستاده می‌شود. قبل از ارسال نتیجه‌ی کانولوشن به لایه‌ی انکدور یک پارامتر قابل یادگیری و غیرضروری class_token که در عمل شرحی از تصویر در سطح بالا است به انتهای دیتا اضافه می‌شود.



```
[ ]  # Pass the logits through sigmoid and get the index of the largest score.
     cls = torch.argmax(
         (nn.Sigmoid())(logits)
     )
     print(cls)

     tensor(1, device='cuda:0')
```

To see what this index represents, we have to check the dataset:

```
[ ]  train_set.class_to_idx

     {'berry': 0, 'bird': 1, 'dog': 2, 'flower': 3, 'other': 4}
```

That seems to be correct. We should now the weights. The model's output, `weights`, is a list containing 12 tensors that are all shaped as `(1, 17, 17)`. As previously discussed, we'll solely focus on the final output. Let's extract that:

```
[ ]  w = weights[-1]
     print(w.shape)

     torch.Size([1, 17, 17])
```

and take a look at the weights associated with the `cls` token, which is:

```
[ ]  # w = None  # Extract the last 16 elements from the first column and reshape it as 4x4.
     # w = torch.flip(w, [0,1])
     w = weights[-1]
     w = w[0,
         0
         ,-16:]
     w = torch.reshape(w, (4, 4))
```

در این بخش ابتدا نتایج حاصل از شبکه به یک لایه‌ی softmax فرستاده می‌شود، سپس مقدار با بالاترین احتمال از طریق تابع argmax پیدا شده و مقدار ایندکس آن برابر با کلاس عکس مورد نظر می‌باشد.در بخش آخر نیست از لیست وزن های محاسبه شده وزن های مربوط به لایه ی آخر (w = weigths[-1]) انتخاب می‌شود سپس در ستون اول آن ۱۶ عنصر آخر استخراج شده و به یک ماتریس ۴*۴ ریشیپ می‌شود و به شکل هیت‌مپ روی تصویر اصلی نمایش داده می‌شود.