CNN、RNN、Transformer 实现

蔚全爱

July 2025

1 CNN

1.1 CNN 的理论

CNN 卷积公式：

离散卷积

$$Y_{c_{\text{out}},i,j} = \sum_{c_{\text{in}}=0}^{C_{\text{in}}-1} \sum_{m=0}^{k_H-1} \sum_{n=0}^{k_W-1} W_{c_{\text{out}},c_{\text{in}},m,n} \cdot X_{c_{\text{in}},i+m,j+n} + b_{c_{\text{out}}} \tag{1}$$

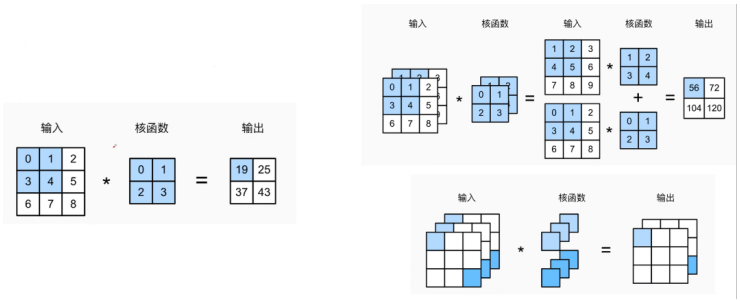


Figure 1: CNN

- 滑动窗口与卷积逐元素相乘相加，核函数是可学的参数，所以不需要翻转
- （RGB）三维通道多输入通道 + 每个都有独自的核函数（一组核函数）= 不同通道的结果相加
- 多维度输出：多输入通道 + 多组核函数
- CNN 可以理解为一种稀疏的 MLP，局部连接，具体实现过程与上述公式不同，采用 for 循环实现，效率过低，那实现就是理解为首先把这个输入拉长之后，局部连接，就可以使用稀疏矩阵加速实现

1.2 CNN 实现

- 数据集：MNIST, 通道数为 1，黑白的，维度为 28*28
- （RGB）三维通道多输入通道 + 每个都有独自的核函数（一组核函数）= 不同通道的结果相加
- 多维度输出：多输入通道 + 多组核函数
- CNN 可以理解为一种稀疏的 MLP，局部连接，具体实现过程与上述公式不同，采用 for 循环实现，效率过低，那实现就是理解为首先把这个输入拉长之后，局部连接，就可以使用稀疏矩阵加速实现

```
class MyCNN(nn.Module):
    def __init__(self):
        super(MyCNN, self).__init__()
        self.conv = MyConv2d(1, 8, 3)#(输入, 输出, 核)
        self.bn = MyBatchNorm2d(8)
        self.pool = nn.MaxPool2d(2)
        #滑动的次数: 输入-核大小/步长向下取整 + 1
        self.fc = nn.Linear(8 * 13 * 13, 10)  # 28 - 3 + 1 =
            26 -> pool -> 13

    def forward(self, x):
        x = self.conv(x)
        x = self.bn(x)
        x = F.relu(x)
        x = self.pool(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x
```

卷积层

```
    class MyConv2d(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size):
        super(MyConv2d, self).__init__()
        # W
        # Parameter可以把一个张量变成一个可学习的参数
        # 乘0.01是为了初始化时的数值不太大
        self.weight = nn.Parameter(torch.randn(out_channels,
            in_channels, kernel_size, kernel_size) * 0.01)
        self.bias = nn.Parameter(torch.zeros(out_channels))
        self.kernel_size = kernel_size

    def forward(self, x):
        batch_size, in_channels, H, W = x.shape
        out_channels = self.weight.shape[0]
        k = self.kernel_size
        # 步长为1, 且不padding
        out_H = H - k + 1
```

```
        out_W = W - k + 1

        output = torch.zeros((batch_size, out_channels, out_H,
            out_W), device=x.device)

        for b in range(batch_size):
            for oc in range(out_channels):
                for ic in range(in_channels):
                    for i in range(out_H):
                        for j in range(out_W):
                            region = x[b, ic, i:i + k, j:j + k]
                            output[b, oc, i, j] +=
                                torch.sum(region *
                                self.weight[oc, ic])
                output[b, oc] += self.bias[oc]
        return output
```

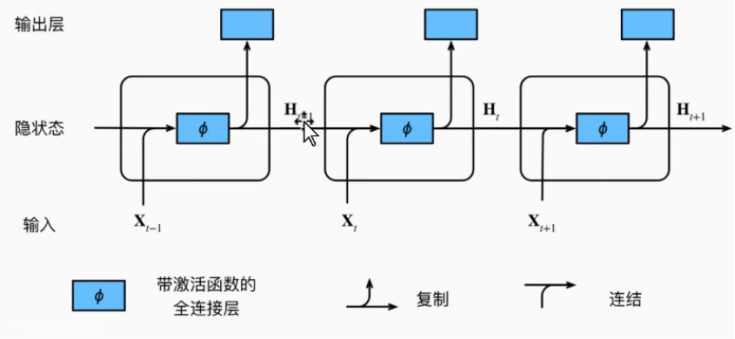
BatchNorm

```
    class MyBatchNorm2d(nn.Module):
    def __init__(self, num_features, eps=1e-5, momentum=0.1):
        super(MyBatchNorm2d, self).__init__()
        self.eps = eps
        self.momentum = momentum
        self.gamma = nn.Parameter(torch.ones(num_features))
        self.beta = nn.Parameter(torch.zeros(num_features))

        self.register_buffer('running_mean',
            torch.zeros(num_features))
        self.register_buffer('running_var',
            torch.ones(num_features))

    def forward(self, x):
        if self.training:
            mean = x.mean([0, 2, 3])
            var = x.var([0, 2, 3], unbiased=False)
            # running_mean更稳定
            self.running_mean = (1 - self.momentum) *
                self.running_mean + self.momentum *
                mean.detach()
            self.running_var = (1 - self.momentum) *
                self.running_var + self.momentum * var.detach()
        else:
        #eval模式和
            mean = self.running_mean
            var = self.running_var
        x_hat = (x - mean[None, :, None, None]) /
            torch.sqrt(var[None, :, None, None] + self.eps)
        return self.gamma[None, :, None, None] * x_hat +
            self.beta[None, :, None, None]
```

2 RNN

2.1 RNN 的理论

- 文本架构：next-token prediction
- 完形填空：挖空预测哪个概率最高
- 文本数据的无监督任务容易实现
- 文本————词汇表（单词映射成数据，词汇表的位置）句子转换数字序列————数字转换成独热编码张量
- 词向量：embedding
- 文本编码存储的是权重矩阵
- 循环，权重共享
- 输出：隐状态再加上一个偏置项，做一个线性变换



$$\mathbf{H}_t = \phi(\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1} \mathbf{W}_{hh} + \mathbf{b}_h).$$

Figure 2: RNN

2.2 RNN 实现

```
class myRNN(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim,
        output_dim):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim)
        self.hidden_dim = hidden_dim

        # 输入到隐藏层线性变换
```

```
        self.W_ih = nn.Linear(embed_dim, hidden_dim)
        # 上一个隐藏状态到当前隐藏状态的线性变换
        self.W_hh = nn.Linear(hidden_dim, hidden_dim)
        self.tanh = nn.Tanh()
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, text):  # 输入形状为 [seq_len,
        batch_size]
        embedded = self.embedding(text)  # [seq_len,
            batch_size, embed_dim]
        seq_len, batch_size, _ = embedded.size()
        h_t = torch.zeros(batch_size,
            self.hidden_dim).to(device)

        # 循环处理序列每个时间步
        for t in range(seq_len):
            x_t = embedded[t]  # 当前时间步的词向量
            h_t = self.tanh(self.W_ih(x_t) + self.W_hh(h_t))
                # 更新隐藏状态

        out = self.fc(h_t)  # 最终输出用最后时刻的隐藏状态计算
        return out
```

- RNN 循环过程，H_t 越靠后，越重要，也就是会遗忘，激活函数，激活函数为 tanh，李普希茨常数小于等于 1，压缩算子，梯度越来越小，导致知识遗忘

3 Transformer

3.1 Transformer 框架

- Transformer 解决了 RNN 的长距离依赖问题
- Transformer 通过自注意力机制，捕捉序列中任意位置之间的关系
- Transformer 由编码器和解码器组成，编码器处理输入序列，解码器生成输出序列
- 自注意力机制允许模型在处理每个词时考虑整个序列的信息
- inputs 可以理解为问题，outputs 可以理解为答案
- prediction next token 为例，以今天星期几为例，outputs 最开始可以理解为占位符，然后预测下一个词，直到预测完毕
- Transformer 缺点：无序列记忆性，轮换不变性，需要加 Positional Encoding：位置编码，将每个词的位置编码为一个向量，加入到词向量中
- Encoder 输出放到 Decoder 中，Decoder 的输入是 Encoder 的输出和上一个时刻的输出

- Encoder 中 Layer Normalization 太多了，只要每个块输出有一个 Layer Normalization 就行了
- Multi-Head Attention：Q、K、V 分别是 Query、Key、Value，Q 和 K 做点积得到注意力权重，然后乘以 V 得到输出，正常只有一个 Attention 头，Multi-Head Attention 是希望注意力更加丰富，图中显示先对 Q、K、V 做线性变换，得到多个头的 Q、K、V，然后分别计算注意力，最后拼接起来，再做一次线性变换
- Transformer 的 Encoder 和 Decoder 都是由多个相同的层堆叠而成，每一层都有 Multi-Head Attention 和 Feed Forward Network(就是 MLP)，首先是注意力矩阵，然后残差连接，MLP，残差连接
- 由于 output 是一个序列，所以需要 Mask，防止模型看到未来的信息，Mask 的作用是将未来的信息置为 0

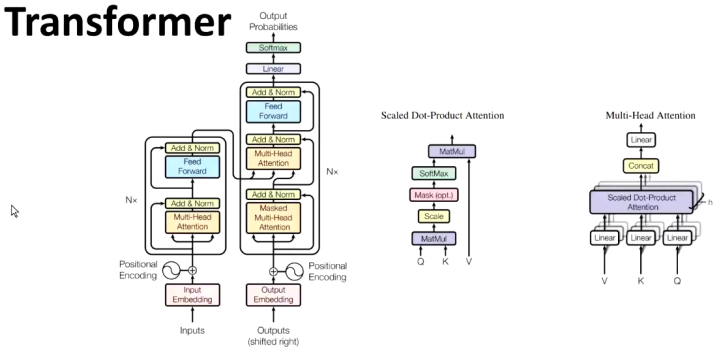


Figure 3: Transformer

3.2 Transformer 实现

Transformer 结构

Listing 1: Transformer 结构

```
class Transformer(nn.Module):
    def __init__(self, n_src_vocab, n_trg_vocab, src_pad_idx,
        trg_pad_idx,
                 d_word_vec=512, d_model=512, d_inner=2048,
                 n_layers=6, n_head=8, d_k=64, d_v=64,
                     dropout=0.1, n_position=200,
                 trg_emb_prj_weight_sharing=True,
                     emb_src_trg_weight_sharing=True,
                 scale_emb_or_prj='prj'):
        super().__init__()
        self.encoder = Encoder(...)
```

```
        self.decoder = Decoder(...)
        self.trg_word_prj = nn.Linear(d_model, n_trg_vocab,
            bias=False)
        if trg_emb_prj_weight_sharing:
            self.trg_word_prj.weight =
                self.decoder.trg_word_emb.weight
        if emb_src_trg_weight_sharing:
            self.encoder.src_word_emb.weight =
                self.decoder.trg_word_emb.weight

    def forward(self, src_seq, trg_seq):
        src_mask = get_pad_mask(src_seq, self.src_pad_idx)
        trg_mask = get_pad_mask(trg_seq, self.trg_pad_idx) &
            get_subsequent_mask(trg_seq)
        enc_output, *_ = self.encoder(src_seq, src_mask)
        dec_output, *_ = self.decoder(trg_seq, trg_mask,
            enc_output, src_mask)
        seq_logit = self.trg_word_prj(dec_output)
        return seq_logit.view(-1, seq_logit.size(2))
```

- Encoder 和 Decoder，线性变换层，还有其中数据处理的掩码 Mask

Encoder 实现

Listing 2: Transformer 的 Encoder 实现

```
class Encoder(nn.Module):
    ''' A encoder model with self attention mechanism. '''
    def __init__(self, n_src_vocab, d_word_vec, n_layers,
        n_head, d_k, d_v,
                 d_model, d_inner, pad_idx, dropout=0.1,
                 n_position=200, scale_emb=False):
        super().__init__()
        self.src_word_emb = nn.Embedding(n_src_vocab,
            d_word_vec, padding_idx=pad_idx)
        self.position_enc = PositionalEncoding(d_word_vec,
            n_position=n_position)
        self.dropout = nn.Dropout(p=dropout)
        self.layer_stack = nn.ModuleList([
            EncoderLayer(d_model, d_inner, n_head, d_k, d_v,
                dropout=dropout)
            for _ in range(n_layers)])
        self.layer_norm = nn.LayerNorm(d_model, eps=1e-6)
        self.scale_emb = scale_emb
        self.d_model = d_model

    def forward(self, src_seq, src_mask, return_attns=False):
        enc_slf_attn_list = []

        enc_output = self.src_word_emb(src_seq)
```

```
        if self.scale_emb:
            enc_output *= self.d_model ** 0.5
        enc_output =
            self.dropout(self.position_enc(enc_output))
        enc_output = self.layer_norm(enc_output)

        for enc_layer in self.layer_stack:
            enc_output, enc_slf_attn = enc_layer(enc_output,
                slf_attn_mask=src_mask)
            if return_attns:
                enc_slf_attn_list += [enc_slf_attn]

        return (enc_output, enc_slf_attn_list) if return_attns
            else (enc_output,)
```

- embedding 层，位置编码，层归一化与 Dropout,for 循环堆叠多个 EncoderLayer, 然后需要显式返回注意力权重矩阵就加一下 self-attention weights

Decoder 实现

Listing 3: Transformer 的 Decoder 实现

```
class Decoder(nn.Module):
    ''' A decoder model with self attention mechanism. '''

    def __init__(
            self, n_trg_vocab, d_word_vec, n_layers, n_head,
                d_k, d_v,
            d_model, d_inner, pad_idx, n_position=200,
                dropout=0.1, scale_emb=False):

        super().__init__()

        self.trg_word_emb = nn.Embedding(n_trg_vocab,
            d_word_vec, padding_idx=pad_idx)
        self.position_enc = PositionalEncoding(d_word_vec,
            n_position=n_position)
        self.dropout = nn.Dropout(p=dropout)
        self.layer_stack = nn.ModuleList([
            DecoderLayer(d_model, d_inner, n_head, d_k, d_v,
                dropout=dropout)
            for _ in range(n_layers)])
        self.layer_norm = nn.LayerNorm(d_model, eps=1e-6)
        self.scale_emb = scale_emb
        self.d_model = d_model

    def forward(self, trg_seq, trg_mask, enc_output, src_mask,
        return_attns=False):
```

```
        dec_slf_attn_list, dec_enc_attn_list = [], []

        # -- Forward
        dec_output = self.trg_word_emb(trg_seq)
        if self.scale_emb:
            dec_output *= self.d_model ** 0.5
        dec_output =
            self.dropout(self.position_enc(dec_output))
        dec_output = self.layer_norm(dec_output)

        for dec_layer in self.layer_stack:
            dec_output, dec_slf_attn, dec_enc_attn = dec_layer(
                dec_output, enc_output,
                    slf_attn_mask=trg_mask,
                    dec_enc_attn_mask=src_mask)
            dec_slf_attn_list += [dec_slf_attn] if
                return_attns else []
            dec_enc_attn_list += [dec_enc_attn] if
                return_attns else []

        if return_attns:
            return dec_output, dec_slf_attn_list,
                dec_enc_attn_list
        return dec_output,
```

- 词嵌入、位置编码、解码器层堆叠、归一化层

Encoder Layer 实现

Listing 4: Transformer 的 Encoder Layer 实现

```
class EncoderLayer(nn.Module):
    ''' Compose with two layers '''

    def __init__(self, d_model, d_inner, n_head, d_k, d_v,
        dropout=0.1):
        super(EncoderLayer, self).__init__()
        self.slf_attn = MultiHeadAttention(n_head, d_model,
            d_k, d_v, dropout=dropout)
        self.pos_ffn = PositionwiseFeedForward(d_model,
            d_inner, dropout=dropout)

    def forward(self, enc_input, slf_attn_mask=None):
        enc_output, enc_slf_attn = self.slf_attn(
            enc_input, enc_input, enc_input,
                mask=slf_attn_mask)
        enc_output = self.pos_ffn(enc_output)
        return enc_output, enc_slf_attn
```

- 由自注意力层和前馈网络组成, 已经加上了残差连接和层归一化

Positional Encoding

- 位置编码：Positional Encoding 其实就是三层的 MLP

Multi-Head-Attention：

Listing 5: Transformer 的 MultiHeadAttention 实现

```
class MultiHeadAttention(nn.Module):
    def __init__(self, n_heads, d_model, d_k, d_v):
        super().__init__()
        self.n_heads = n_heads
        self.d_k = d_k
        self.d_v = d_v
        self.qkv_proj = nn.Linear(d_model, n_heads * (d_k +
            d_k + d_v))
        self.out_proj = nn.Linear(n_heads * d_v, d_model)

    def forward(self, x, mask=None):
        B, L, D = x.size()
        qkv = self.qkv_proj(x).view(B, L, self.n_heads, -1)
        q, k, v = torch.split(qkv, [self.d_k, self.d_k,
            self.d_v], dim=-1)
        q, k, v = [t.transpose(1, 2) for t in (q, k, v)]
        attn_weights = torch.matmul(q, k.transpose(-2, -1)) /
            (self.d_k ** 0.5)
        if mask is not None:
            attn_weights = attn_weights.masked_fill(mask == 0,
                float("-inf"))
        attn = torch.softmax(attn_weights, dim=-1)
        out = torch.matmul(attn, v)
        out = out.transpose(1, 2).contiguous().view(B, L, -1)
        return self.out_proj(out), attn
```

- 使用一个大线性变换矩阵（qkv_proj）一次性将输入 $\boldsymbol{X} \in \mathbb{R}^{B\times L\times d_{\text{model}}}$ 映射为多个注意力头上的 Q、K、V 向量，避免三次重复线性映射。
- 每个注意力头计算 $\text{Attention}(Q,K,V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$
- 将所有注意力头的输出拼接，并通过 out_proj 投影回原始维度。