《机器学习编程实践》课程——DAY 3

课程内容

- **CNN 代码实现**
- **RNN 代码实现**
- **transformer 代码实现**

一、 CNN 的实现

1 数学逻辑

- 将输入与核函数逐元素相乘得到输出
- 直接**逐元素相乘**，不再进行顺序调换：核函数可学，顺序调换与否不影响
- **多输入情况**：为每个输入各配置一个核函数，得到一个或多个输出（多个输出：有多组核函数，每组核函数个数与输入个数相同）
- **在 pythorch 中的实现**：将输入和核函数全部拉平，看做一个稀疏的 MLP 网络，可以大幅提高运行效率（否则很慢）

2 代码实现

2.1 CNN 整体框架

```
class MyCNN(nn.Module):
  def __init__(self):
    super(MyCNN, self).__init__()
    self.conv = MyConv2d(1, 8, 3) # 卷积层
    self.bn = MyBatchNorm2d(8) # BatchNorm层
    self.pool = nn.MaxPool2d(2) # Max Pooling层
    self.fc = nn.Linear(8 * 13 * 13, 10) # 28 - 3 + 1 = 26 -> pool -> 13 #
        线性分类器层

  # 逐步得到输出
  def forward(self, x):
    x = self.conv(x) # 引用conv中的forward方法
    x = self.bn(x)
```

```
        x = F.relu(x)
        x = self.pool(x)
        x = x.view(x.size(0), -1) # 先拉平，之后进行线性回归
        x = self.fc(x)
        return x
```

2.2 卷积层

```
class MyConv2d(nn.Module):
    # in_channels: 输入通道数; out_channels: 输出通道数; kernel_size: 卷积核
    def __init__(self, in_channels, out_channels, kernel_size):
        super(MyConv2d, self).__init__()
        self.weight = nn.Parameter(torch.randn(out_channels, in_channels,
            kernel_size, kernel_size) * 0.01) #
            初始化权重参数（使用标准正态分布）
        self.bias = nn.Parameter(torch.zeros(out_channels))
        self.kernel_size = kernel_size

    def forward(self, x):
        # batch_size: 批大小; H: 高度; W: 宽度
        batch_size, in_channels, H, W = x.shape
        out_channels = self.weight.shape[0]
        k = self.kernel_size
        # 得到输出特征图尺寸
        out_H = H - k + 1
        out_W = W - k + 1

    output = torch.zeros((batch_size, out_channels, out_H, out_W),
        device=x.device) # 初始化输出张量

    for b in range(batch_size): # 遍历每个样本(b)
        for oc in range(out_channels): # 遍历每个输出通道(oc)
            for ic in range(in_channels): # 遍历每个输入通道(ic)
                for i in range(out_H): # 遍历输出高度方向(i)
                    for j in range(out_W): # 遍历输出宽度方向(j)
                        region = x[b, ic, i:i + k, j:j + k]
                        output[b, oc, i, j] += torch.sum(region * self.weight[oc,
                            ic]) # 计算区域与权重的点积并累加到输出
                        output[b, oc] += self.bias[oc]
```

```
                return output
```

上文为基于**循环**实现的卷积层，效率较低，实际 PyTorch 实现使用优化后的 C++ 代码。

还可以修改 stride 和定义 padding：

- **stride**：默认为 1；当不为 1 时代表跳步执行（如为 2 时跳一步进行按元素乘积）
- **padding**：默认为 0（无填充），此时边缘只被执行一次，当 padding 为 1 时，为周边再加一圈 0，使得边缘也被执行两次

修改后的代码：

```
# 修改stride=2，同时加入padding（默认值分别为2和1）
class MyConv2d(nn.Module):
  def __init__(self, in_channels, out_channels, kernel_size, stride=2,
      padding=1):
    super(MyConv2d, self).__init__()
    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
    self.stride = stride
    self.padding = padding

  def forward(self, x):
    batch_size, in_channels, H, W = x.shape
    out_channels = self.weight.shape[0]
    k = self.kernel_size
    p = self.padding

    # 计算输出尺寸（考虑padding和stride）
    out_H = (H + 2*p - k) // self.stride + 1
    out_W = (W + 2*p - k) // self.stride + 1

    # 对输入进行padding
    if self.padding > 0:
    x_padded = torch.zeros((batch_size, in_channels, H + 2*p, W + 2*p),
       device=x.device)
    x_padded[:, :, p:p+H, p:p+W] = x # 中心填充
    else:
    x_padded = x
```

```
    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):
                # 计算输入区域的起始位置（考虑stride）
                   h_start = i * self.stride
                   w_start = j * self.stride
                   region = x_padded[b, ic, h_start:h_start + k,
                       w_start:w_start + k]
                   output[b, oc, i, j] += torch.sum(region * self.weight[oc,
                       ic])
                   output[b, oc] += self.bias[oc]
    return output
```

二、 RNN 的实现

1 训练模式

- **predict the next token**

  （1）将”How are you?” 作为输入，预测下一个 token，输出得到一个”I”

  （2）将它们组合在一起：”How are you? I” 作为一个新的输入，预测下一个 token，输出得到”am”

  （3）以此类推

- **做完形填空（mask 掩码）**

  把一句话挖几个空（进行 mask 掩码），预测哪几个空概率最高

2 代码实现

```
# 自定义RNN类
class myRNN(nn.Module):
   # vocab_size: 词汇表大小; embed_dim: 词嵌入维度; hidden_dim:
       隐藏层维度; output_dim: 输出维度;
```

```
    def __init__(self, vocab_size, embed_dim, hidden_dim, output_dim):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim) #
            定义词嵌入层，将词汇索引映射到密集向量表示
        self.hidden_dim = hidden_dim

        # 参数初始化
        self.W_ih = nn.Linear(embed_dim, hidden_dim) #
            定义输入到隐藏层的线性变换(对应传统RNN公式中的W_x)
        self.W_hh = nn.Linear(hidden_dim, hidden_dim)
        self.tanh = nn.Tanh() # tanh激活函数
        self.fc = nn.Linear(hidden_dim, output_dim) # 全连接输出层

    def forward(self, text): # [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) #
            初始化隐藏状态h_t为全零张量
        for t in range(seq_len): # 遍历序列
            x_t = embedded[t]
            # 经过输入层到隐藏层、隐藏层到隐藏层的线性变换
            # 相加经过tahn激活函数得到新的隐藏状态
            h_t = self.tanh(self.W_ih(x_t) + self.W_hh(h_t))

        out = self.fc(h_t) # 将最后一个时间步的隐藏状态通过全连接层得到输出
        return out
```

三、 transformer 介绍

1 组成架构

- **编码器**

  由 N 个相同层堆叠而成，每层包含：

  - **多头自注意力机制**：计算输入序列中每个词与其他词的关联权重
  - **前馈神经网络**：对每个位置的表示进行非线性变换
  - **残差链接和层归一化**

- **解码器**

比编码器多一个掩码多头注意力

2 关键组件介绍

2.1 自注意力机制（Self-Attention）

- **输入表示**：每个词转换为 **Query（Q）**、**Key（K）**、**Value（V）**三个向量。
- **注意力分数**：通过 $\text{Softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)$ 计算权重，再加权求和 V。
- **公式**：

$$\text{Attention}(Q,K,V) = \text{Softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

2.2 位置编码（Positional Encoding）

- Transformer 无递归结构，需通过**位置编码**注入序列顺序信息。
- 使用正弦/余弦函数生成固定位置编码，与词向量相加：

$$\begin{cases} PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right) \\ PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right) \end{cases}$$

其中 pos 为位置，i 为维度索引。

2.3 多头注意力（Multi-Head Attention）

- 将 Q、K、V 投影到 h 个子空间（头），独立计算注意力后拼接：

$$\text{MultiHead}(Q,K,V) = \text{Concat}(\text{head}_1,\ldots,\text{head}_h)W^O$$

- 每个头的计算：

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

其中 W_i^Q, W_i^K, W_i^V 为可学习参数矩阵，W^O 为输出投影矩阵。

3 代码实现

3.1 编码器 encoder

```
class Encoder(nn.Module):
  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 = []

        # -- Forward
        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)
            enc_slf_attn_list += [enc_slf_attn] if return_attns else []

        if return_attns:
            return enc_output, enc_slf_attn_list
        return enc_output,
```

3.2 解码器 decoder

```
class Decoder(nn.Module):
    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,
```

3.3 transformer 架构

```
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.src_pad_idx, self.trg_pad_idx = src_pad_idx, trg_pad_idx

      assert scale_emb_or_prj in ['emb', 'prj', 'none']
      scale_emb = (scale_emb_or_prj == 'emb') if trg_emb_prj_weight_sharing
          else False
      self.scale_prj = (scale_emb_or_prj == 'prj') if
          trg_emb_prj_weight_sharing else False
      self.d_model = d_model

      self.encoder = Encoder(
      n_src_vocab=n_src_vocab, n_position=n_position,
      d_word_vec=d_word_vec, d_model=d_model, d_inner=d_inner,
      n_layers=n_layers, n_head=n_head, d_k=d_k, d_v=d_v,
      pad_idx=src_pad_idx, dropout=dropout, scale_emb=scale_emb)

      self.decoder = Decoder(
      n_trg_vocab=n_trg_vocab, n_position=n_position,
      d_word_vec=d_word_vec, d_model=d_model, d_inner=d_inner,
      n_layers=n_layers, n_head=n_head, d_k=d_k, d_v=d_v,
      pad_idx=trg_pad_idx, dropout=dropout, scale_emb=scale_emb)

      self.trg_word_prj = nn.Linear(d_model, n_trg_vocab, bias=False)

      for p in self.parameters():
         if p.dim() > 1:
            nn.init.xavier_uniform_(p)

      assert d_model == d_word_vec, \
```

```
    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)
    if self.scale_prj:
    seq_logit *= self.d_model ** -0.5

    return seq_logit.view(-1, seq_logit.size(2))
```