机器学习编程课笔记

2025 年 7 月 30 日

1 卷积神经网络（CNN）

总体结构概述

卷积神经网络是一类深度学习模型，专门用于处理具有网格结构的数据，如图像（二维网格）。它的典型结构包括：

- **卷积层（Convolution Layer）**：通过滑动窗口操作提取图像局部特征。
- **批归一化层（Batch Normalization）**：加速训练并提升稳定性。
- **激活函数（如 ReLU）**：引入非线性变换。
- **池化层（Pooling Layer）**：进行下采样操作，降低特征维度。
- **全连接层（Fully Connected Layer）**：用于最终分类或回归输出。

MyConv2d 模块解析

Listing 1: 自定义二维卷积实现

```
class MyConv2d(nn.Module):
    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)
        self.bias = nn.Parameter(torch.zeros(out_channels))
        self.kernel_size = kernel_size
```

在初始化阶段，定义了卷积核的权重和偏置，其中权重是四维张量。这里的权重乘以 0.01 是为了避免初始值过大导致训练不稳定；偏置则初始化为 0。

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

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

在前向传播中，首先根据输入尺寸和卷积核大小手动计算输出图像的高宽。在上面的实现中，并未涉及 padding（填充）和 stride（步幅）这两个在实际卷积操作中非常关键的参数。Padding 是指在输入特征图的边缘进行零填充，其主要作用是控制输出特征图的空间尺寸。当不使用 padding 时，每次卷积会导致输出尺寸变小，边缘信息也可能被忽略。而使用适当的 padding（例如设置为 kernel size 的一半），可以在不改变空间维度的情况下提取边缘特征，从而实现所谓的 “same” 卷积。Stride 则是控制卷积核在输入上滑动的步长，默认值为 1，表示每次移动一个像素；当 stride 设为更大的值时，输出特征图的尺寸会相应缩小，特征提取变得更稀疏，计算速度加快，但可能会损失部分细节。在当前的 `MyConv2d` 实现中，stride 被默认固定为 1，padding 为 0，因此输出尺寸为 $H - k + 1$，即每次卷积操作都会压缩特征图的空间大小。

```
        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
```

使用五重嵌套循环手动实现卷积操作。每次选取输入特征图中的一个局部区域，与对应卷积核进行逐元素相乘并求和，然后再加上偏置。

最终输出是一个维度为 $(\text{batch}, \text{out\_channels}, \text{out\_H}, \text{out\_W})$ 的张量。这种实现的运行速度很慢，效率很低。

MyBatchNorm2d 模块解析

Listing 2: 二维批归一化层

```
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))
        # 把cpu上的移动到gpu上
        self.register_buffer('running_mean', torch.zeros(num_features))
        self.register_buffer('running_var', torch.ones(num_features))
```

该模块用于对每个通道的输出进行标准化处理。使用 γ 和 β 两个可学习参数对归一化结果进行缩放与平移。

```
    def forward(self, x):
        if self.training:
            mean = x.mean([0, 2, 3])
            var = x.var([0, 2, 3], unbiased=False)
            # 动量加权求和，计算当前的均值和过去的均值加权
            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:
            mean = self.running_mean
            var = self.running_var
        # 由于维数不一样，扩充维数便于计算
        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]
```

在训练时，按通道对输入求均值和方差，并更新移动均值和方差；在测试时则使用历史值。为了实现逐通道归一化操作，对均值和方差进行广播，使得它们的维度与输入一致。

MyCNN 网络结构解释

Listing 3: MyCNN 网络

```
class MyCNN(nn.Module):
    def __init__(self):
        super(MyCNN, self).__init__()
        self.conv = MyConv2d(1, 8, 3)
        self.bn = MyBatchNorm2d(8)
        self.pool = nn.MaxPool2d(2)
        self.fc = nn.Linear(8 * 13 * 13, 10) # 28 - 3 + 1 = 26 -> pool -> 13
```

输入图像为 28×28，经过大小为 3 的卷积核后，变为 26×26；再经过大小为 2 的最大池化，缩小为 13×13。最终展平成 $8 \times 13 \times 13$ 的一维向量，输入至全连接层输出

10 维的结果。

```
    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
```

前向传播中，卷积、归一化、激活与池化顺序执行，再展平送入全连接层。最终的输出通常用于分类任务（如 MNIST 手写数字识别）。

2 循环神经网络（RNN）结构解析

结构与原理概述

循环神经网络（RNN）特别适用于处理序列数据，如文本、语音信号等。其核心在于：当前时刻的隐藏状态不仅由当前输入决定，还会受到前一时刻隐藏状态的影响，从而捕捉时间依赖关系。在前向传播的过程中，模型通过循环结构对每个时间步的输入进行迭代更新，核心计算过程可表示为以下公式：

$$h_t = \tanh(W_{ih}x_t + W_{hh}h_{t-1})$$

其中，x_t 表示第 t 个时间步的输入向量，h_{t-1} 是前一时刻的隐藏状态，W_{ih} 和 W_{hh} 分别是输入到隐藏层、隐藏层到隐藏层的权重矩阵，tanh 是双曲正切非线性激活函数。每一个隐藏状态 h_t 都是当前输入和历史状态的融合，包含了从 x_1 到 x_t 的累积信息。由于该模型仅在最后一个时间步将 h_T 输入至全连接层输出分类结果，因此整个序列的信息被浓缩到了最终的状态向量之中。这种结构适合于文本分类等只需全局信息的任务。如果要处理如文本生成等需要每步输出的任务，则可在每个 t 时刻将 h_t 映射至输出空间，并构建输出序列。

myRNN 模型结构说明

Listing 4: RNN 的构造函数

```
class myRNN(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim, output_dim):
        super().__init__()
        # 对输入进行编码有利于模型的计算和学习，把单独的词汇表示转换成embedding
        self.embedding = nn.Embedding(vocab_size, embed_dim)
        self.hidden_dim = hidden_dim
```

```
    # 参数初始化
    self.W_ih = nn.Linear(embed_dim, hidden_dim)
    self.W_hh = nn.Linear(hidden_dim, hidden_dim)
    self.tanh = nn.Tanh()
    self.fc = nn.Linear(hidden_dim, output_dim)
```

文本首先通过嵌入层编码成向量形式，这样能更好地捕捉语义信息。之后，使用两组线性变换矩阵 W_ih 和 W_hh，分别对应输入和前一隐藏状态。

```
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)
    # xt+1包含了xt-1、ht-1、ht的所有的信息
    for t in range(seq_len):
        x_t = embedded[t]
        h_t = self.tanh(self.W_ih(x_t) + self.W_hh(h_t))
    out = self.fc(h_t)
    return out
```

输入是一个长度为 T 的文本序列（每一列是一个单词的编号）。模型逐步迭代每一时刻，更新隐藏状态。最终隐藏状态 h_T 被输入全连接层，得到分类结果。注释中指出隐藏状态逐步累积历史信息，因此即使没有使用显式的输出列表 'output'，最后状态也已包含完整的语义信息。

3 Transformer

总体结构概述

Transformer 是一种基于自注意力机制(Self-Attention)的序列建模架构，由 Vaswani 等人于 2017 年提出。与 RNN 不同，Transformer 摒弃了递归结构，完全依赖并行的注意力机制进行信息交互，因此在处理长序列时效率更高、表现更稳定。

Transformer 的典型结构包括：

- **输入嵌入与位置编码**：将输入词映射为向量并注入位置信息。
- **多层编码器（Encoder）和解码器（Decoder）堆叠结构**：每层包含多头自注意力、前馈网络及残差连接。
- **注意力模块**：用于捕捉序列中不同位置间的依赖关系。
- **多头机制**：并行化多个注意力头以增强模型对不同关系的建模能力。

Transformer 的核心思想是将每个词（token）的表示向量与序列中所有其他词进行交互，通过加权聚合形成更具语义的表达，进而支持翻译、摘要、对话等各类自然语言处理任务。

编码器模块：Encoder

Listing 5: Encoder 构造函数

```
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):
```

Encoder 的输入包括词表大小、词向量维度、层数、注意力头数量、模型维度、前馈层宽度等超参数。其中 `scale_emb` 控制是否对词向量进行缩放，以增强数值稳定性。

```
        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)
```

每个词首先通过嵌入层映射为向量，并加入位置编码，使模型能感知词语顺序（注意 Transformer 本身不具备序列建模能力，需显式编码位置信息）。

```
        self.layer_stack = nn.ModuleList([
            EncoderLayer(d_model, d_inner, n_head, d_k, d_v, dropout=dropout)
            for _ in range(n_layers)])
```

Encoder 内部由多层 `EncoderLayer` 构成，每层包含自注意力和前馈子结构，堆叠实现逐层抽象表达。

Listing 6: Encoder 前向传播

```
    def forward(self, src_seq, src_mask, return_attns=False):
        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)
```

输入序列经词嵌入和位置编码处理后，逐层传入编码器层，每一层进行注意力计算与前馈更新，最后输出一个包含上下文语义的特征表示。

解码器模块：Decoder

Listing 7: Decoder 构造函数

```
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):
```

解码器与编码器结构类似，但每层包含三种注意力：自注意力、编码器-解码器注意力、前馈网络。模型将目标序列的已生成部分作为输入，通过自回归生成目标输出。

Listing 8: Decoder 前向传播

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

解码器将嵌入和位置编码后的目标序列输入每一层，并依次执行：

1. 自注意力，允许模型访问已生成的目标词；
2. 编码器-解码器注意力，从编码器的输出中读取源语言信息；
3. 前馈网络增强非线性建模能力。

缩放点积注意力机制

Listing 9: Scaled Dot-Product Attention

```
class ScaledDotProductAttention(nn.Module):
    def forward(self, q, k, v, mask=None):
        attn = torch.matmul(q / self.temperature, k.transpose(2, 3))
        if mask is not None:
            attn = attn.masked_fill(mask == 0, -1e9)
        attn = self.dropout(F.softmax(attn, dim=-1))
        output = torch.matmul(attn, v)
        return output, attn
```

这是 Transformer 的核心机制，计算公式如下：

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

其中 Q、K、V 分别是查询、键、值向量，d_k 是缩放因子，mask 用于避免模型访问未来信息（用于解码器）。

多头注意力机制

Listing 10: Multi-Head Attention

```
class MultiHeadAttention(nn.Module):
    def forward(self, q, k, v, mask=None):
        q = self.w_qs(q).view(sz_b, len_q, n_head, d_k)
        k = self.w_ks(k).view(sz_b, len_k, n_head, d_k)
        v = self.w_vs(v).view(sz_b, len_v, n_head, d_v)

        q, k, v = q.transpose(1, 2), k.transpose(1, 2), v.transpose(1, 2)

        q, attn = self.attention(q, k, v, mask=mask)

        q = q.transpose(1, 2).contiguous().view(sz_b, len_q, -1)
        q = self.dropout(self.fc(q)) + residual
        q = self.layer_norm(q)
        return q, attn
```

多头注意力将输入分别映射成多个子空间，通过多个缩放点积注意力并行计算后拼接，最后再线性映射回模型维度。每个注意力头都可专注于不同的特征子空间，增强模型表达能力。残差连接和 LayerNorm 保证梯度稳定性和训练收敛。