实验一：情感分类实验报告*

张铭徐 [1]
[1] 南开大学计算机学院　学号：2113615

摘要

情感分类任务（Sentiment Classification Task）是自然语言处理（NLP）中的一个重要任务，旨在识别和分类文本中表达的情感倾向。通常，这些情感倾向可以分为正面、负面和中性，或者更细分的类别如高兴、愤怒、悲伤等。情感分类广泛应用于产品评价分析、社交媒体监测、品牌声誉管理等领域。在本次实验中，我们实现了基于卷积神经网络（CNN）以及基于长短周期记忆网络（LSTM）的情感分类模型，探究不同超参数对模型性能的影响，以及不同词向量嵌入模型对模型训练过程中的影响（即是否加载 GloVe 模型作为预训练权重）；最终我们的 CNN 模型能够在测试集上获得 86.5% 的准确率；LSTM 模型能够在测试集上获得 86.9% 的准确率。

1. 实验原理

1.1. 实验概述

本次实验拟进行情感分类任务，即将给定的一段文本 x 进行分类任务，判定其情感类别为正面（输出为 1），或是负面（输出为 0）。在本次实验中，我们拟实现模型 f，即映射 $f : \mathrm{x} \to R \in \{0, 1\}$。

在本次实验中，我们拟采用 IMDB 数据集（定义在 data.py 中），其有 25,000 条电影评论构成，每一条评论都具有标签 0 或 1，代表该条评论是正面或负面，在本次实验中，我们需要在该数据集上对模型进行训练和评估，试图达到最优的模型效果，即具有最强的泛化能力。

*南开大学 2024Fall，自然语言处理课程实验报告

1.2. 数据预处理

首先第一步，我们需要对 IMDB 中的所有数据进行处理，将其中无意义的字符进行剔除，以免影响模型训练效果。我们在 data.py 中定义了数据处理方法，第一步为文本标准化：使用 basic_english_normalize 函数对文本进行基本的标准化处理，包括：

- 将文本转换为小写，确保大小写一致。
- 去除或替换文本中的特定字符，如引号、圆括号、感叹号、问号等。
- 替换
 等 HTML 标签为空格。
- 在标点符号前后添加空格，使得标点符号与单词分离，方便后续的分词。
- 替换多余的空格，确保最终的文本只有单一空格。

随后，我们简单的通过 line_spilt 方法对给定的长句进行分词化，切分为若干 tokens，并存储在词汇表中，最终将原本的评论转换为 tokens 的列表。

接着，我们将原本 tokens 的列表（即若干字符表征的文本）转换为索引序列，每一个位置都转换为词汇表的索引，保证最终得到的是数值组成的序列。

最终，为保证输入数据的规模一致性，我们将文本进行对齐处理，对于长度较短的文本进行填充处理，在文本末端填入特殊字符。

2. 模型结构描述

在该部分中，我们将详细描述在本次实验中采用的模型的结构，分别介绍本次实验实现的 CNN

模型结构，见subsection 2.2；以及 LSTM 的结构，见subsection 2.3。在分别介绍上述模型结构之前，其整体的顶层设计是类似的，我们先介绍针对该问题，模型的顶层设计描述。

2.1. 模型整体结构描述

本次实验我们拟实现的是分类问题，目前机器学习有几种范式，包括监督学习（Supervised Learning），无监督学习（Unsupervised Learning），以及强化学习（Reinforcement Learning），而监督学习中又可大致分为两种主要任务：回归（Regression）以及分类（Classification），本次我们拟要实现的为分类任务。

分类任务的核心思想是通过模型找到一个决策边界 [1]，能够将不同类别的数据样本进行区分。这个决策边界可能是线性的，也可能是复杂的非线性函数。通过优化模型的参数，使得决策边界能够尽可能准确地划分不同类别的数据样本，从而在预测时能够对新输入进行正确分类。那么实际上我们后续所有模型的设计都是围绕上述核心进行的。

在分类任务中，无论模型前面的架构如何设计，我们都要经过一个全连接层，其形状为 $p \times q$，其中 p 为上一层网络的输出维度，q 为最终模型类别数量，其输出值为一个 $n \times q$ 的向量，其中 n 为输入数据个数即 batch_size，对于每一个数据，我们都有 $1 \times q$ 的数据输出向量 $Q[x_1, x_2, \ldots, x_q]$，对于其中任意一维 x_i，我们可以视其为模型对于输入数据属于第 i 类的置信度，我们可以通过 softmax 函数将对应的激活值转换为置信值即概率：

$$y_i = \frac{x_i}{\Sigma_j^q x_j} \tag{1}$$

最终模型的决策为：

$$\text{output label} = \arg\max_i x_i \tag{2}$$

而使用深度学习的方法，就需要优化目标即损失函数，我们在分类任务中，优化目标为最大化训练样本的置信度，即若训练样本 i 属于类 j，我们想要让对应的 x_j 的值尽可能大，我们常用的损失函数为交叉熵损失函数（Cross Entropy Loss），其定义如下：

$$\mathcal{L} = -\sum_{i=1}^{n}\sum_{j=1}^{q} y_{ij}\log(\hat{y}_{ij}) \tag{3}$$

其中：$\mathcal{L}$ 是模型的总损失，n 是批次中的样本数量（batch size），q 是分类的类别数量，y_{ij} 是样本 i 在第 j 类的真实标签，$\hat{y}_{ij}$ 是样本 i 被预测为第 j 类的概率。这个概率是通过 Softmax 函数将模型的输出值 x_i 转换而来的。

2.1.1 **嵌入层** Embedding

在计算机视觉任务中，以彩色图像为例，由于图像本身为 RGB 构成的三维向量，即 $n \times m \times 3$，我们可以直接对输入图像进行处理，通过灰度、滤波等数据增强技术对输入图像进行预处理，以增强有限数据下模型的训练效果。但是到了 NLP 的任务中，由于文本信息在计算机中的表示方法为编码，如 ASCII 码、UniCode 等，该表示方法表示的向量并不具有语义信息，且表示过于稀疏，导致表征空间过大而效果较差，故无法直接输入到神经网络中进行处理，所以无论是哪个模型，我们都需要对文本进行嵌入处理。

实际上嵌入处理是学习到一个从原始向量空间到语义空间到一个映射 $\phi : R^1 \to R^q$，假设原本输入数据尺寸为 $n \times m$，其中 n 为输入数据个数（batch_size），m 为文本长度；针对于原本的每一组文本数据而言，其每一个维度仅仅是无意义的索引，我们通过映射 ϕ，即嵌入操作，会获得一个新矩阵：$n \times m \times q$，其中 q 为嵌入维度。即我们将每一个无意义的索引，映射到了一个高维空间中，该索引在高维空间中即可学习到语义等信息，帮助模型更好的进行分类等任务，而在 Pytorch 中，我们使用的 embedding 层是随机初始化的，我们需要在不断的学习中优化该层的相关参数，达到最好的效果。

另一方面，在 NLP 的历史发展中，对于词的嵌入表示也有较多的研究，如 Word2Vec[2]，GloVe[3]等；其在一个大规模无监督语料库上进行预训练，通过最大化给定中心词，目标词出现的概率（反之亦然）；或通过词向量的内积近似拟合词共现的相关频率。所以进行嵌入的另一种途径即为使用预训练的嵌入

矩阵在下游任务上进行应用。本次实验中，我们在 LSTM 模型中对比了使用自定义 embedding 层的性能和使用预训练嵌入矩阵 GloVe 的性能表现。

2.2. CNN 结构

实现文本分类的第一个模型为卷积神经网络（CNN），我们使用 CNN 进行特征提取以及降维工作，代码如Figure 1所示：

```python
class CNN(nn.Module):
    def __init__(
        self,
        vocab_size,
        embedding_dim,
        n_filters,
        filter_sizes,
        output_dim,
        dropout_rate,
        pad_index,
    ):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=pad_index)
        self.convs = nn.ModuleList(
            [
                nn.Conv1d(in_channels=embedding_dim, out_channels=n_filters, kernel_size=fs)
                for fs in filter_sizes
            ]
        )
        self.fc = nn.Linear(len(filter_sizes) * n_filters, output_dim)
        self.dropout = nn.Dropout(dropout_rate)
```

图 1: CNN 结构

我们下面详细给出每一个组件的设计原理以及数据流的形状：

- **输入层**：模型的输入是一个由单词序列构成的文本数据，表示为一个整数索引序列，其形状为 $[batch\_size, seq\_len]$。$batch\_size$ 是每批次的样本数量，seq_len 是文本的最大长度。
- **嵌入层** (Embedding Layer)：嵌入层将输入的单词索引序列转换为词向量矩阵，映射为 $[batch\_size, seq\_len, embedding\_dim]$ 的张量。该层的作用是将离散的单词表示为连续的稠密向量，便于后续的卷积操作。嵌入矩阵的大小为 $[vocab\_size, embedding\_dim]$，其中 $vocab\_size$ 是词汇表大小，$embedding\_dim$ 是词嵌入向量的维度。
- **卷积层** (Convolutional Layer)：卷积层采用多个一维卷积核（filters）对输入的词向量进行特征提取。每个卷积核的大小为 $[embedding\_dim, kernel\_size]$，其中 $kernel\_size$ 是卷积核的宽度。卷积操作后，得到形状为 $[batch\_size, n\_filters, L]$ 的特征图，其中 $n\_filters$ 是卷积核的数量，L 是经过卷积后序列的长度。卷积操作可以捕捉到文本中不同窗口大小的特征。
- **激活函数与池化层** (Activation and Pooling Layer)：在每个卷积层后，我们使用 ReLU 激活函数，增加模型的非线性表达能力。接着使用最大池化 (MaxPooling) 操作，将每个卷积核输出的特征图取最大值，从而减少特征图的尺寸，并保留最显著的特征。池化后的特征图的形状为 $[batch\_size, n\_filters]$。
- Dropout **层** (Dropout Layer)：在全连接层之前，加入 Dropout 层。其作用是随机将部分神经元的输出设置为零，以减少模型对特定神经元的依赖，可以有效防止模型的过拟合。
- **全连接层** (Fully Connected Layer)：将经过 Dropout 的特征向量输入全连接层，将其映射到目标输出维度。全连接层的输入大小为 $[batch\_size, n\_filters \times \text{len(filter\_sizes)}]$，输出大小为 $[batch\_size, output\_dim]$，其中 $output\_dim$ 是分类任务中的类别数。
- **输出层** (Output Layer)：输出层我们使用 Softmax 函数，将全连接层的输出转换为每个类别的概率分布，从而用于文本分类任务的预测，每一个维度为对应类别的置信度。

2.3. LSTM 结构

在实现了 CNN 后，接下来我们实现了 LSTM 结构，LSTM 解决了原本 RNN 的若干问题，其遗忘门、输出门的设计使得其能够有效避免梯度消失，保存长期依赖关系，我们实现的 LSTM 的结构如Figure 2所示：

```python
class LSTM(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim, n_layers, dropout_rate, pad_index, pretrained_embeddings=
        super().__init__()
        # 如果提供了预训练的嵌入矩阵，则使用它，否则随机初始化
        if pretrained_embeddings is not None:
            self.embedding = nn.Embedding.from_pretrained(pretrained_embeddings, padding_idx=pad_index, freeze=freeze_embeddings)
        else:
            self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=pad_index)

        self.lstm = nn.LSTM(embedding_dim, hidden_dim, num_layers=n_layers, dropout=dropout_rate, batch_first=True)
        self.fc = nn.Linear(hidden_dim, output_dim)
        self.dropout = nn.Dropout(dropout_rate)

    def forward(self, ids):
        embedded = self.dropout(self.embedding(ids))
        output, (hidden, cell) = self.lstm(embedded)
        prediction = self.fc(hidden[-1])
        return prediction
```

图 2: LSTM 结构

- **嵌入层** (Embedding Layer): 模型首先通过嵌入层将输入的词汇索引序列映射为稠密的词向量表示。具体来说，嵌入层的大小为 $[vocab\_size, embedding\_dim]$，其中 $vocab\_size$ 是词汇表大小，$embedding\_dim$ 是词嵌入的维度。由于我们在实验中进行了对比，在这里，如果我们提供了预训练的嵌入矩阵（即 GloVe 词向量），则使用该矩阵进行初始化；否则，嵌入层的权重随机初始化。
- LSTM **层** (LSTM Layer): 嵌入后的词向量序列输入到 LSTM 层。LSTM 层的输入形状为 $[batch\_size, seq\_len, embedding\_dim]$，输出形状为 $[batch\_size, seq\_len, hidden\_dim]$，其中 $hidden\_dim$ 是 LSTM 的隐藏层维度。LSTM 层能够捕捉序列中的长短期依赖关系，并具有 n_layers 层 LSTM 堆叠，$dropout\_rate$ 用于在层与层之间进行 Dropout 操作，防止过拟合。
- **全连接层** (Fully Connected Layer): LSTM 层的输出包括隐藏状态 $hidden$ 和单元状态 $cell$，其中我们使用最后一层 LSTM 的最终隐藏状态 $hidden[-1]$ 作为文本的特征表示。该表示经过全连接层（'fc'）映射到输出维度。全连接层的输入大小为 $[batch\_size, hidden\_dim]$，输出大小为 $[batch\_size, output\_dim]$，其中 $output\_dim$ 为分类的类别数量。
- Dropout **层** (Dropout Layer): 在嵌入层输出后，以及 LSTM 输出至全连接层之前，我们加入了 Dropout 层。Dropout 可以防止模型对特定特征的过度依赖，有助于提升模型的泛化能力。

3. 实验结果分析

3.1. 实验超参数设置

我们首先给出在本次实验中使用的超参数以及对应的值，如Table 1所示。

每一个参数的具体含义如下：

- max_length: 输入文本的最大长度，用于截断或填充序列，以保证输入序列具有相同的长度。
- test_size: 验证集所占的比例，用于从训练集中划分出一部分数据作为验证集。
- min_freq: 构建词汇表时的最小词频。低于该频率的词将被视为未知词 (<unk>)。
- batch_size: 每个批次的样本数量，用于定义每次输入到模型中的数据大小。
- embedding_dim: 词嵌入向量的维度，表示每个词被映射到的连续向量空间的大小。
- n_filters: CNN 卷积核的数量，表示卷积操作中不同特征提取器的数量。
- filter_sizes: CNN 卷积核的尺寸，用于定义不同窗口大小的特征提取。
- dropout_rate: Dropout 比例，用于防止过拟合，在训练过程中随机丢弃部分神经元。
- n_epochs: 训练轮数，表示整个训练数据集被模型迭代的次数。
- device: 使用的计算设备，通常为 cuda:X 表示使用第 X 块 GPU 设备进行训练。
- hidden_dim: LSTM 隐藏层的维度，表示 LSTM 中隐藏状态的大小。

表 1: 实验使用的超参数设置

超参数	**值**
max_length	256
test_size	0.25
min_freq	5
batch_size	256
embedding_dim	300
n_filters	100
filter_sizes	[3, 5, 7]
dropout_rate	0.3
n_epochs	40
device	cuda:1
hidden_dim	256
n_layers	4
pretrained_embeddings	GloVe (6B, 300d)
freeze_embeddings	True / False
learning_rate	0.001
optimizer	Adam

- n_layers: LSTM 堆叠的层数，表示 LSTM 的深度。
- pretrained_embeddings: 使用的预训练 GloVe 嵌入向量，用于初始化词嵌入层。
- freeze_embeddings: 是否冻结嵌入层。如果为 True，则在训练过程中不更新嵌入层的权重。
- learning_rate: 学习率，控制模型权重更新的步长。
- optimizer: 使用的优化器，在这里我们使用 Adam 优化器，用于调整模型参数以最小化损失函数。

3.2. CNN 结果分析

3.2.1 探究不同超参数对性能影响

在这里，我们探究 embedding_dim，n_filters，filter_sizes，对模型性能的影响。在实验中，我们固定学习率 (learning rate) 为 0.001，训练轮数为 10 轮，丢弃率为 0.3。超参数的取值如下：

- embedding_dim: {200, 250, 300}，表示词嵌入向量的维度，用于控制每个词在嵌入空间中的表示大小。
- n_filters: {100, 150, 200}，表示每个卷积核尺寸下的卷积核数量，控制特征提取的丰富度。
- filter_sizes: { {[2, 4, 6]}, {[4, 6, 8]}, {[5, 7, 9]}}，表示每组卷积核的大小。每组包含不同的卷积窗口，用于捕捉不同范围的文本特征。

最终原始实验结果如Table 2所示，此外，我们将固定某一参数的取值，考虑某一参数的取值对最终模型表现的影响，将同一固定参数下，另外两种参数取值视为多次实验，绘制为误差棒的形式，给出性能对比，如Figure 3所示。

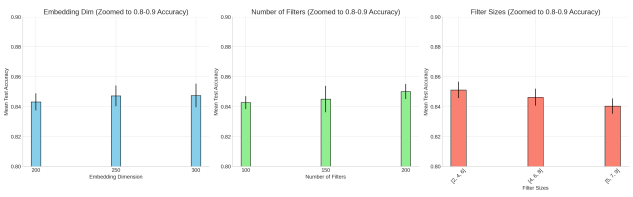


图 3: 不同属性对模型性能影响

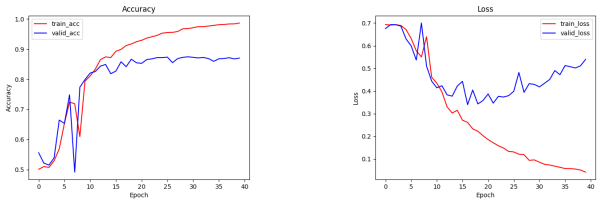
我们发现：

- **对于嵌入维度方面**：在当前实验设置之下，嵌入维度为 300 的模型表现优于嵌入维度为 200、100 的模型，并且误差棒的长度略长，我们推测这是因为较大的嵌入维度能够捕捉到更丰富的词汇语义信息，但也容易导致模型复杂度增加，从而使得训练过程的稳定性有所下降。
- **对于滤波层数方面**：在当前实验设置之下，滤波层数越高，模型性能越好；我们注意到滤波层数为 150 时模型的误差棒较长，推测这是因为中等数量的滤波器在特征提取过程中能够较好地捕捉不同尺度的特征，但可能在某些情况下难以平衡模型的训练时间和准确度，从而导致性能不稳定。
- **对于滤波尺寸方面**：我们发现滤波尺寸较大时，性能相对较差，我们推测这是因为较大的滤波尺寸会在局部特征的提取上变得不够细致，错过一些对情感分类较为重要的细粒度特征，从而影响模型的分类效果。

3.3. LSTM 结果分析

3.3.1 词嵌入对比

在 LSTM 的实验中，我们在轮数为 40 轮、学习率为 0.0005，模型层数为 3 的情况下，使用 GloVe 和不使用 GloVe 的性能表现对比，使用 GloVe 的情况下，训练 loss 和准确率的图如Figure 4所示；不使用 GloVe 的情况下，训练 loss 和准确率的图如Figure 5所示：



((a)) Accuracy ((b)) Loss

图 4: LSTM 模型的准确率和损失曲线

我们可以看到，不使用 GloVe 的模型最终的准确率为 86.9%，使用 GloVe 的模型最终准确率为

表 2: Test Results for Different Filter Sizes

((a)) Filter Sizes [2, 4, 6]

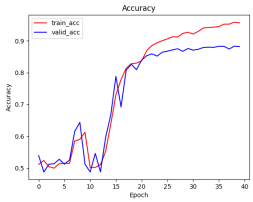
Embedding Dim	Num Filters	Test Loss	Test Accuracy
200	100	0.3501	0.8425
200	150	0.3385	0.8500
200	200	0.3356	0.8531
250	100	0.3409	0.8501
250	150	0.3408	0.8496
250	200	0.3261	0.8583
300	100	0.3478	0.8444
300	150	0.3235	0.8574
300	200	0.3348	0.8555

((b)) Filter Sizes [4, 6, 8]

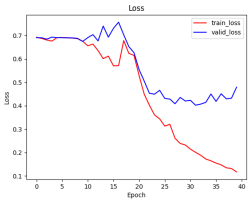
Embedding Dim	Num Filters	Test Loss	Test Accuracy
200	100	0.3574	0.8400
200	150	0.3634	0.8368
200	200	0.3514	0.8465
250	100	0.3557	0.8410
250	150	0.3412	0.8514
250	200	0.3475	0.8501
300	100	0.3483	0.8475
300	150	0.3444	0.8522
300	200	0.3440	0.8514

((c)) Filter Sizes [5, 7, 9]

Embedding Dim	Num Filters	Test Loss	Test Accuracy
200	100	0.3627	0.8381
200	150	0.3622	0.8379
200	200	0.3571	0.8442
250	100	0.3723	0.8369
250	150	0.3680	0.8389
250	200	0.3485	0.8487
300	100	0.3540	0.8440
300	150	0.3900	0.8317
300	200	0.3705	0.8433

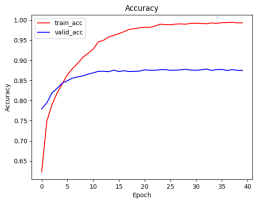


((a)) Accuracy

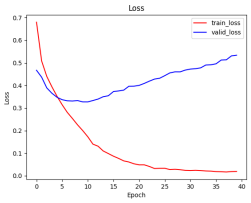


((b)) Loss

图 5: LSTM 模型的准确率和损失曲线（不使用 GloVe）



((a)) Accuracy



((b)) Loss

图 6: CNN 模型的准确率和损失曲线

84.5%，两者差距不大，但是我们可以发现，使用 GloVe 的模型可以更快地收敛，loss 下降速度更快，在训练初期的准确率就很高。这是因为使用了预训练的词嵌入模型，GloVe 向量已经在大规模文本数据上进行了训练，捕捉到了词语之间的语义关系。因此，模型在初始化时就具备了一定的语义知识，能够在训练数据上更快地找到优化方向，从而在训练初期就取得较好的性能。而不使用预训练嵌入的模型，需要从随机初始化的状态开始学习，因此需要更多的训练轮次来逐渐学习到词汇间的关系，最终才能达到较高的准确率。虽然两者在最终准确率上相差不大，但 GloVe 提供的预训练知识能够显著加速模型的训练过程，使得模型在较少的训练轮次下达到更高的效果。

3.3.2 与 CNN 模型对比

我们在其余指标完全相同的情况下，对 CNN 模型进行了测试，定义参数为：LSTM 层数为 3 层，不使用 GloVe，嵌入维度为 300，学习率为 0.0005，丢失率为 0.3。在这种情况之下，CNN 的训练 loss 和准确率如Figure 6所示，LSTM 的性能图如Figure 5所示。

我们可以看到，CNN 的最终准确率为 86.5%，稍低于 LSTM 模型，但由于差距过小，我们有理由怀疑是偶然现象。从训练过程来看，CNN 的更新迭代过程更为平滑，而 LSTM 模型则有着震荡的现象出现，我们推测原因如下：

- **时间步依赖性方面**：LSTM 模型的每一个时间步都会受到前面时间步的影响，这使得 LSTM 在处理输入序列时对输入数据变化十分敏感。因此，当训练数据中存在变化或噪声时，LSTM 容易在训练过程中出现不稳定的情况。
- **超参数敏感度方面**：LSTM 对优化器的选择和学习率设置更为敏感。学习率过高可能导致 LSTM 在训练中发生较大的参数更新，从而出现训练损失不稳定的情况。而 CNN 在这方面的表现更为稳健，即使在相对较高的学习率下，也能保持平滑的训练过程。
- **特征提取方面**：CNN 通过卷积核提取局部特征，并不依赖于序列数据的时间顺序，因此更容易捕捉到稳定的特征。而 LSTM 需要捕捉输入序列中的长短期依赖关系，这使得它在训练过程中对输入数据的微小变化也可能做出较大的反应。

3.4. 问题发现

我们从Figure 4Figure 5,Figure 6可以发现，就算我们加入了防止过拟合的 Dropout，但是仍然出现了这种情况，即训练集误差仍在下降而测试集误差反而上升。在这种情况下，有以下几种方法可以缓解该问题：

- **增加数据集的规模**：我们可以使用更多的训练样本可以帮助模型学习到更丰富的特征，从而减少模型的过拟合现象。如果可以获取到更多标注样本，可以通过扩大训练数据集来提升模型的泛化能力。
- **数据增强操作**：我们可以对训练数据进行数据增强操作，例如随机删除、同义词替换、词序打

乱等操作，可以增加模型对噪声的鲁棒性，并模拟更多的样本变体，从而减轻过拟合的影响。

- **降低模型的复杂度**：此外，我们可以通过减少模型的参数量（如减少 LSTM 层数、隐藏单元数量或卷积核数量），使模型变得更简单，从而减轻模型对训练数据的过度拟合（即模型自由度变低）[1]。
- **正则化方法**：除了 Dropout 之外，还可以使用 L2 正则化（权重衰减），在损失函数中加入模型参数的 L2 范数，抑制过大的权重值，从而防止模型对训练数据的过度拟合。
- **使用早停策略**：通过在训练过程中监控验证集的损失，如果验证集的损失在若干个 epoch 内不再下降或出现上升，则提前停止训练，选择验证集损失最小的模型作为最终模型。
- **调低学习率**：降低学习率可以使模型在训练过程中以更小的步伐更新参数，从而更精细地调整模型的权重，避免在训练集上过度拟合。

4. 总结

在本次实验中，我们针对情感分类任务，设计并实现了两种神经网络模型：卷积神经网络（CNN）和长短期记忆网络（LSTM），并探索了不同超参数和预训练词向量对模型性能的影响。通过实验，我们得出了以下主要结论：

- **模型性能对比**：在相同的实验条件下，LSTM 模型在测试集上达到了 86.9% 的准确率，而 CNN 模型达到了 86.5% 的准确率。虽然 LSTM 的最终准确率略高于 CNN，但两者的差距较小，可能存在一定的偶然性。这表明在情感分类任务中，虽然 LSTM 能更好地捕捉文本序列的时序信息，但 CNN 对文本的局部特征提取能力也非常强大，能够在较短时间内获得较好的结果。
- **收敛速度的差异**：实验表明，使用预训练的 GloVe 词向量模型，可以显著加速 LSTM 的收敛过程。相比于从头学习词嵌入的模型，使用 GloVe 的 LSTM 在训练初期便表现出了更高的准确率和更快的 loss 下降速度。这是因为 GloVe 在大规模语料上训练过，具备了丰富的语义信息，能够为下游任务提供更好的初始表示，使得模型能够更快地找到优化方向。
- **稳定性分析**：CNN 模型在训练过程中表现得更为平稳，loss 曲线和准确率曲线变化较为顺滑。而 LSTM 模型在训练中出现了一定程度的震荡。这种现象主要是由于 LSTM 模型对时间序列数据较为敏感，以及其在反向传播中容易受到梯度消失或梯度爆炸的影响。尽管 LSTM 使用了门控机制来缓解这些问题，但在长序列数据或学习率不合适时，仍然可能导致训练过程的不稳定。
- **超参数影响**：我们在实验中探索了词嵌入维度（embedding_dim）、卷积核数量（n_filters）、卷积核尺寸（filter_sizes）和 Dropout 比例（dropout_rate）对模型性能的影响。实验结果显示，较高的词嵌入维度和适量的卷积核数量有助于提升模型的准确率；不同的卷积核尺寸组合能够捕捉到不同粒度的特征信息，从而提升 CNN 模型的泛化能力。此外，合理的 Dropout 设置能够有效防止模型过拟合，提升模型在验证集上的表现。
- **模型选择的权衡**：在情感分类任务中，选择 CNN 还是 LSTM 需要根据具体需求进行权衡。CNN 模型在短文本特征提取和快速训练方面具有优势，适用于对训练时间要求较高的场景。LSTM 模型则更适合处理具有复杂时序关系的长文本数据，但训练时间较长且对超参数设置更为敏感。

在本次实验的基础上，未来的改进方向包括：

- **尝试其他预训练词向量**：如使用 BERT 或 FastText 等更先进的预训练词向量模型，比较其在情感分类任务中的性能表现。
- **优化模型结构**：可以尝试结合 CNN 和 LSTM 的混合模型，如使用 CNN 提取局部特征后再送入 LSTM 捕捉全局依赖关系，从而提高模型的分类能力。
- **调整超参数**：进一步细化超参数搜索，如优化学习率、批次大小等，寻找更适合的参数配置，以

进一步提升模型性能。

- **使用更大的数据集**：在更大规模的数据集上进行训练和验证，以验证模型在真实应用场景下的泛化能力。

通过本次实验，我们深入理解了情感分类任务的原理和实现过程，以及不同模型在该任务中的优劣势。未来工作将继续探索更好的模型结构和优化方法，以提升文本分类任务中的模型性能。本次实验用到的代码可以在section 5中找到。

本次实验在 Nvidia-A6000 上进行实验，CUDA 版本为 12.2，使用 2.2.2 版本的 Torch，0.17.2 版本的 Torchtext 进行实验，Ubuntu 版本为 20.04。

参考文献

[1] C. M. Bishop. Pattern recognition and machine learning. Springer google schola, 2:1122–1128, 2006. 2, 8

[2] T. Mikolov. Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781, 2013. 2

[3] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. In Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP), pages 1532–1543, 2014. 2

5. Appendix

5.1. CNN 模型定义

```python
class CNN(nn.Module):
    def __init__(
        self,
        vocab_size,
        embedding_dim,
        n_filters,
        filter_sizes,
        output_dim,
        dropout_rate,
        pad_index,
    ):
        super().__init__()
        self.embedding = nn.Embedding(
            vocab_size, embedding_dim,
            padding_idx=pad_index)
        self.convs = nn.ModuleList(
            [
                nn.Conv1d(in_channels=
                    embedding_dim,
                    out_channels=n_filters,
                    kernel_size=fs)
                for fs in filter_sizes
            ]
        )
        self.fc = nn.Linear(len(
            filter_sizes) * n_filters,
            output_dim)
        self.dropout = nn.Dropout(
            dropout_rate)

    def forward(self, ids):
        embedded = self.dropout(self.
            embedding(ids))
        embedded = embedded.permute(0, 2,
            1)
        conved = [F.relu(conv(embedded))
            for conv in self.convs]
        pooled = [F.max_pool1d(conv, conv.
            shape[2]).squeeze(2) for conv in
             conved]
        cat = self.dropout(torch.cat(pooled
            , dim=-1))
        prediction = self.fc(cat)
        return prediction
```

Listing 1: CNN 模型

此段代码实现了一个基于卷积神经网络（CNN）的文本分类模型。嵌入层首先将输入的文本转换为词向量，然后通过不同卷积核对输入进行特征提取。池化层用于减少特征图的维度并保留最显著的特征。最后，通过全连接层输出预测结果。整个网络结构较为简单，但在处理文本分类任务中能够取得较好的效果，尤其在短文本或固定大小的输入上具有优势。

5.2. LSTM 模型定义

```python
class LSTM(nn.Module):
    def __init__(self, vocab_size,
        embedding_dim, hidden_dim,
        output_dim, n_layers, dropout_rate,
        pad_index, pretrained_embeddings=
        None, freeze_embeddings=False):
        super().__init__()
        if pretrained_embeddings is not
            None:
            self.embedding = nn.Embedding.
                from_pretrained(
                pretrained_embeddings,
                padding_idx=pad_index,
                freeze=freeze_embeddings)
        else:
            self.embedding = nn.Embedding(
                vocab_size, embedding_dim,
                padding_idx=pad_index)

        self.lstm = nn.LSTM(embedding_dim,
            hidden_dim, num_layers=n_layers,
             dropout=dropout_rate,
            batch_first=True)
        self.fc = nn.Linear(hidden_dim,
            output_dim)
        self.dropout = nn.Dropout(
            dropout_rate)

    def forward(self, ids):
```

```
        embedded = self.dropout(self.
            embedding(ids))
        output, (hidden, cell) = self.lstm(
            embedded)
        prediction = self.fc(hidden[-1])
        return prediction
```

Listing 2: LSTM 模型

在这一部分代码中，我们实现了基于 LSTM 的模型。嵌入层可使用随机初始化的词嵌入或预训练的 GloVe 词向量来加速收敛。LSTM 通过捕捉输入序列中的长短期依赖关系来学习特征，并通过全连接层输出预测。LSTM 相比于 CNN 更适合处理序列任务，尤其在长文本和具有复杂时序依赖的任务中具有优势。

5.3. GloVe 权重矩阵构建

```
embedding_dim = 300
glove = torchtext.vocab.GloVe(name='6B'
    , dim=embedding_dim)
vocab_size = len(vocab)
pad_index = vocab["<pad>"]
embedding_matrix = torch.zeros(
    vocab_size, embedding_dim)
for i, token in enumerate(vocab.
    get_itos()):
    if token in glove.stoi:
        embedding_matrix[i] = glove[
            token]
    else:
        embedding_matrix[i] = torch.
            randn(embedding_dim)
embedding_matrix = embedding_matrix.
    float()
```

Listing 3: GloVe 矩阵构建

这段代码用于加载预训练的 GloVe 词嵌入，并将其应用于模型的嵌入层中。对于在 GloVe 词汇表中找不到的词，我们使用随机初始化向量来代替。使用预训练的词嵌入能够显著加速模型的收敛过程，并提升模型在情感分类任务中的表现，尤其是在数据量较小的情况下，预训练词向量能够帮助模型更快地学习到语义信息。

5.4. 绘图及日志记录

```
plt.clf()
plt.plot(metrics_LSTM["train_losses"],
    label="train_loss", color="red")
plt.plot(metrics_LSTM["valid_losses"],
    label="valid_loss", color="blue")
plt.title("Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
plt.savefig("loss_LSTM.png")

plt.clf()
plt.plot(metrics_LSTM["train_accs"],
    label="train_acc", color="red")
plt.plot(metrics_LSTM["valid_accs"],
    label="valid_acc", color="blue")
plt.title("Accuracy")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.legend()
plt.savefig("acc_LSTM.png")

with open(f"{time.strftime('%Y-%m-%d-%H
    -%M%S')}_LSTM.log", "w") as f:
    f.write(f"test_loss: {test_loss:.3f
        }, test_acc: {test_acc:.3f}\n")
    f.write(f"hyper-parameters: {args}\
        n")
    f.write(f"metrics: {metrics_LSTM}\n
        ")
    f.write(f"vocab: {vocab}\n")
    f.write(f"model: {model}\n")
    f.write(f"optimizer: {optimizer}\n"
        )
    f.write(f"criterion: {criterion}\n"
        )
    f.write(f"device: {args.device}\n")
    f.write(f"special_tokens: {
        special_tokens}\n")
    f.write(f"unk_index: {unk_index}\n"
        )
    f.write(f"pad_index: {pad_index}\n"
        )
    f.write(f"vocab_size: {vocab_size}\
```

```
        n")
    f.write(f"output_dim: {output_dim}\
        n")
```

Listing 4: 绘图及日志记录

该代码用于记录模型训练的关键信息，包括训练与验证过程中的损失值与准确率，并通过可视化生成相应的图像。此外，它将超参数设置、训练结果等写入日志文件。记录实验过程中的这些信息有助于后续的分析和比较，特别是在超参数调优过程中能够直观地展示模型的收敛情况和性能变化。

5.5. Tuning Parameters

```
embedding_dim = [200, 250, 300]
n_filters = [100, 150, 200]
filter_sizes = [[2, 4, 6], [4, 6, 8], [5,
    7, 9]]
dropout_rate = [0.3]
results = []
for i in range(3):
for j in range(3):
for k in range(3):
for l in range(1):
model = CNN(
    vocab_size=vocab_size,
    embedding_dim=embedding_dim[i],
    n_filters=n_filters[j],
    filter_sizes=filter_sizes[k],
    output_dim=output_dim,
    dropout_rate=dropout_rate[l],
    pad_index=pad_index,
)
optimizer = optim.Adam(model.parameters(),
    lr=0.0005)
criterion = nn.CrossEntropyLoss()
metrics = collections.defaultdict(list)
model = model.to(args.device)
criterion = criterion.to(args.device)
metrics_CNN = collections.defaultdict(list)
print("Now CNN performance:\n")
for epoch in range(args.n_epochs):
    train_loss, train_acc = train(
        train_data_loader, model, criterion,
         optimizer, args.device)
    valid_loss, valid_acc = evaluate(
        valid_data_loader, model, criterion,
         args.device)
    metrics_CNN["train_losses"].append(
        train_loss)
    metrics_CNN["train_accs"].append(
        train_acc)
    metrics_CNN["valid_losses"].append(
        valid_loss)
    metrics_CNN["valid_accs"].append(
        valid_acc)
    if not metrics_CNN["valid_accs"] or
        valid_acc > max(metrics_CNN["
        valid_accs"]):
        torch.save(model.state_dict(), "
            best_model_CNN.pth")
        best_valid_acc = valid_acc
    print(f"epoch: {epoch}")
    print(f"train_loss: {train_loss:.3f},
        train_acc: {train_acc:.3f}")
    print(f"valid_loss: {valid_loss:.3f},
        valid_acc: {valid_acc:.3f}")

test_loss, test_acc = evaluate(
    test_data_loader, model, criterion, args
    .device)
print(f"test_loss: {test_loss:.3f},
    test_acc: {test_acc:.3f}")

results.append({
    "embedding_dim": embedding_dim[i],
    "n_filters": n_filters[j],
    "filter_sizes": filter_sizes[k],
    "dropout_rate": dropout_rate[l],
    "test_loss": test_loss,
    "test_acc": test_acc
})

del model

print(results)
with open(f"{time.strftime('%Y-%m-%d-%H-%M
    -%S')}_CNN_tuning.log", "w") as f:
f.write(f"results: {results}\n")
```

Listing 5: Tuning Parameters

该代码用于进行超参数调优实验。通过遍历不同的嵌入维度、卷积核数量、卷积核尺寸和丢失率，测试模型在这些设置下的表现。每次实验后，将相应的测试集损失值和准确率记录下来，并保存实验结果。通过这种方式，我们能够在实验中找出最优的超参数组合，从而提升模型性能。