

# 自然语言处理研讨课（实践课）

## 第 6 章序列标注实践：BiLSTM+CRF

潘宇轩

2025 年 12 月 8 日

### 实验成果总结

模型类型	验证集 F1	测试集 F1	训练耗时 (RTX 4090)
BiLSTM+CRF (基线)	88.64%	~83.00%	~30 分钟 (25 epoch)
<b>BiLSTM+CRF (优化后)</b>	<b>90.40%</b>	83.73%	~35 分钟 (25 epoch)
BERT (预训练微调)	96.33%	<b>91.52%</b>	~14 分钟 (5 epoch)

\* 注：由于 *BiLSTM* 串行计算特性及 *CRF* 动态规划开销，在 5 万条样本全量训练下耗时较长，故未进行大规模网格搜索。

## 目 录

1 引言 . . . . .	3
2 技术背景 . . . . .	3
2.1 BIO 标注体系 . . . . .	3
2.2 BiLSTM+CRF 模型架构 . . . . .	3
2.3 变长序列的批次处理 . . . . .	4
2.4 维特比算法 . . . . .	6
3 实验数据与环境配置 . . . . .	7
3.1 数据集概况 . . . . .	7
3.2 硬件环境与计算约束 . . . . .	7
3.3 数据处理的动态优化 . . . . .	8

<b>4 基线模型实现</b>	<b>9</b>
4.1 基线架构设计	9
4.2 初始训练配置	9
4.3 基线训练中的问题暴露	9
<b>5 模型系统的演进与优化</b>	<b>9</b>
5.1 模型容量与正则化的平衡	10
5.2 参数初始化的细节调整	10
5.3 训练稳定性的关键修复	11
<b>6 实验结果与深入分析</b>	<b>11</b>
6.1 训练过程的稳定性分析	11
6.2 最终性能与 BERT 的跨维度对比	12
6.3 交互式推理系统的开发	14
<b>7 预训练模型的引入与突破</b>	<b>14</b>
7.1 困境与破局：一次“冲动”的尝试	14
7.2 BERT 模型配置	15
7.3 降维打击：从“苦劳”到“红利”	15
<b>8 总结</b>	<b>16</b>
8.1 从“调参”到“认知”的转变	16
8.2 准确率与效率的博弈	16
8.3 遗憾	16

## 1 引言

序列标注 (Sequence Labeling) 是自然语言处理中最基础也最考验细节的任务之一。与文本分类“一锤子买卖”将整个文本映射为单一标签不同，序列标注要求对输入序列中的每一个元素（字或词）进行独立的分类判断，同时还需要考虑前后标签之间的强依赖关系。在实际业务中，无论是如火如荼的知识图谱构建，还是智能问答系统，都离不开高质量的命名实体识别 (Named Entity Recognition, NER)。

在模型选择方面，本次实验我选择了经典的 BiLSTM+CRF 作为切入点。尽管在 Transformer 横行的今天，该架构显得有些“传统”，但它完美结合了深度学习 (LSTM 的特征提取能力) 和概率图模型 (CRF 的结构化预测能力) 的优势，是理解序列标注任务特性的最佳样本。在完成基线模型的实现后，我重点关注了训练稳定性的优化，并引入了 BERT 预训练模型进行对比，以探讨“特征工程”与“预训练知识”之间的性能鸿沟。

## 2 技术背景

### 2.1 BIO 标注体系

在命名实体识别任务中，如何准确界定实体的边界是一个核心问题。如果仅仅判断一个字“是不是实体”，我们将无法区分两个相邻的同类型实体（例如“北京上海”会被识别连在一起）。为此，本实验采用了标准的 **BIO** 标注体系。

如图 1 所示，该体系通过引入位置信息来解决边界模糊问题。其中 **B (Begin)** 标签专门用于标记实体的起始字符，**I (Inside)** 标签用于标记实体的内部字符，而 **O (Outside)** 则表示非实体字符。

以图中的样本为例：“赵阳在中科院工作”。

- “赵”字被标注为 **B-PER**，明确了人名实体的开始；
- “阳”字标注为 **I-PER**，表示人名的延续；
- 随后的“在”字为 **O**，切断了实体流；
- 紧接着“中”字标记为 **B-ORG**，开启了一个新的机构名实体。

这种标注方式使得模型不仅能识别“是什么”，还能精确定判“从哪开始，到哪结束”，这是后续模型能够进行结构化预测的数据基础。

### 2.2 BiLSTM+CRF 模型架构

本实验构建的序列标注系统采用了经典的 **BiLSTM+CRF** 架构。这套架构的设计哲学是将深度学习强大的特征提取能力与概率图模型严谨的逻辑约束能力相结合。如

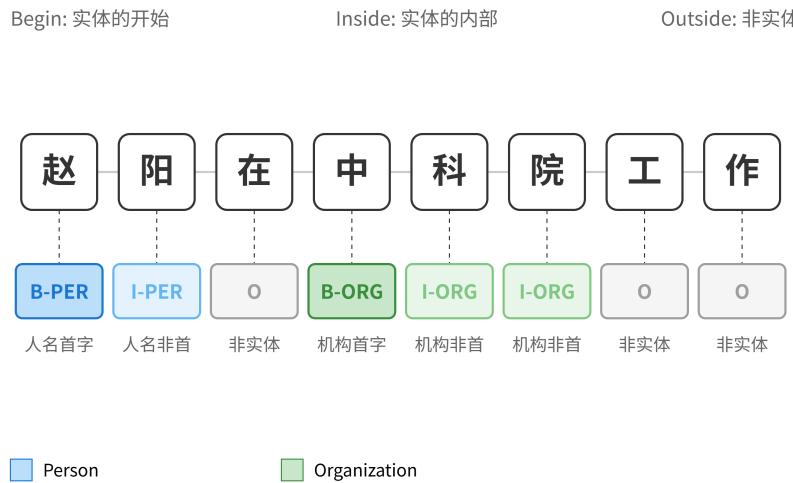


图 1: BIO 标注体系示例

图 2 所示，数据流经三个关键层级：

首先是底层的 **Embedding** 层，它将离散的汉字（如“我”、“爱”、“北”、“京”）映射为稠密的实数值向量。这些向量构成了模型的输入基石。

中间层采用了 **双向长短期记忆网络 (BiLSTM)**。与单向 RNN 不同，BiLSTM 包含前向和后向两个通道。对于序列中的任意一个字（例如图中黄色的  $h_2$ ），前向 LSTM 提供了其上文的信息 ( $h_{1\_f}$ )，后向 LSTM 提供了其下文的信息 ( $h_{3\_b}$ )。这种双向机制确保了每个时间步的特征表示都融合了整个句子的完整上下文语境，解决了“断章取义”的问题。

最顶层是 **条件随机场 (CRF)**。虽然 LSTM 输出了每个字属于各个标签的“发射分数”，但它无法感知标签之间的依赖关系（例如 **I-PER** 绝不可能直接出现在 **O** 之后）。CRF 层通过学习一个全局的转移矩阵，对整个标签序列进行打分。如图所示，CRF 实际上是在所有可能的路径中寻找一条得分最高且逻辑合法的路径，从而输出最终的标签序列。

### 2.3 变长序列的批次处理

在深度学习的工程实践中，为了充分利用 GPU 的矩阵并行计算能力，我们必须将多个样本组成一个批次 (Batch)。然而，自然语言中的句子长度参差不齐，这就引入了对齐问题。

如图 3 所示，我们采用了 **Padding (填充)** 机制。系统会自动识别当前批次中最长的句子长度，并将较短的句子（如第一句“我爱 NLP”）用特殊的 PAD 符号填充至该长度。

然而，简单的填充会带来副作用：这些无意义的 PAD 符号如果不加处理，会参与

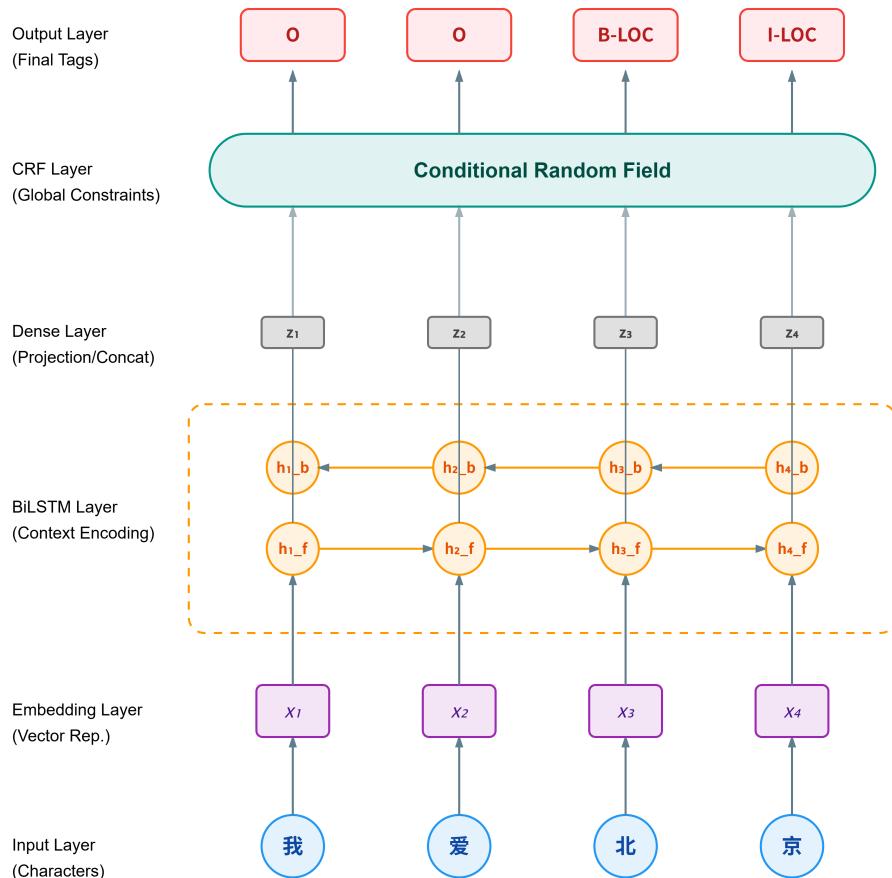


图 2: BiLSTM+CRF 模型整体架构：自底向上依次为字符嵌入、双向上下文编码以及 CRF 全局约束解码。

LSTM 的运算并产生错误的梯度，甚至干扰 CRF 的路径计算。为此，我们同步生成了一个 **Attention Mask 矩阵**（如图右侧绿色部分所示）。该矩阵由 0 和 1 组成，精确标记了哪些位置是真实数据（1），哪些是填充数据（0）。在模型的前向传播和损失计算过程中，Mask 矩阵充当了“过滤器”，强制模型忽略填充区域，确保了计算结果的准确性。

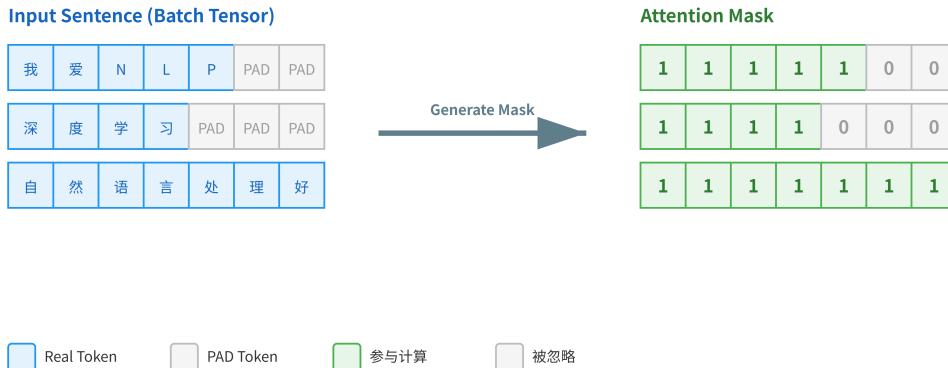


图 3: 批次填充与掩码机制

## 2.4 维特比算法

CRF 层的核心挑战在于解码。对于一个长度为  $n$  的句子和  $k$  个可能的标签，理论上存在的标签序列组合高达  $k^n$  种。如果采用穷举法搜索最优路径，计算量将呈指数级爆炸，导致模型无法实时运行。

为了解决这一问题，我们使用了 **维特比算法（Viterbi Algorithm）**。如图 4 所示，我们将解码过程建模为一个在篱笆网络（Lattice）上的最短路径搜索问题。算法利用了动态规划思想：到达当前时刻某个状态的最优路径，一定源自前一时刻的某个最优路径。

通过这种方式，我们只需要计算并保存每个时刻到达各个节点的最大得分（局部最优解），就能递推地找到全局最优解。图中红色的粗线直观地展示了这一过程：算法最终锁定了一条从 **B-PER** 到 **I-PER** 再到 **O** 的最佳路径，这正是“张三在...”这句话的正确标注序列。维特比算法将解码的时间复杂度降低到了  $O(n \cdot k^2)$ ，使得复杂的 CRF 模型具备了工程实用性。

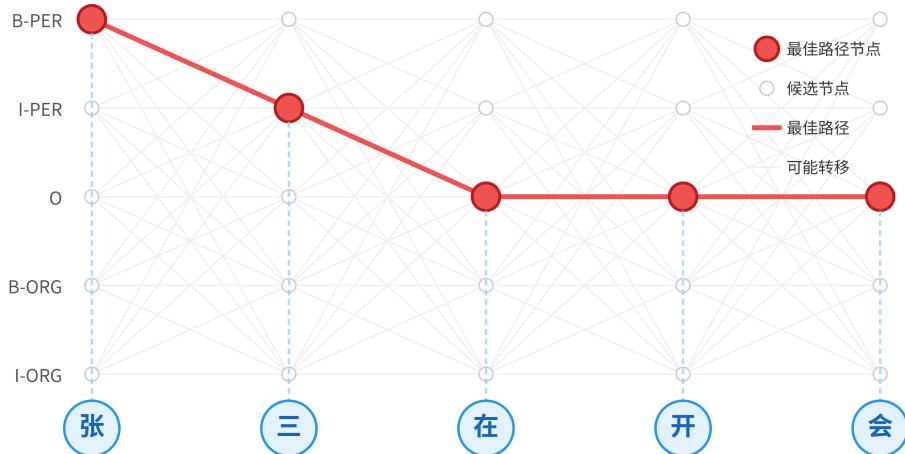


图 4: 维特比算法解码过程示意

### 3 实验数据与环境配置

#### 3.1 数据集概况

本次实验使用的是标准的中文命名实体识别(NER)数据集，主要包含人名(PER)、地名(LOC)和机构名(ORG)三类实体。数据集已被划分为训练集(约5.1万条)、验证集(2000条)和测试集(2000条)。这种划分方式既保证了模型有充足的数据进行参数学习，又留出了独立的验证集用于监控过拟合现象。

从数据分布来看，样本长度普遍在50到100字之间。由于采用 BIO 标注体系，标签类别数共有7类(B/I-PER, B/I-LOC, B/I-ORG, O)，加上CRF所需的START和STOP标签，模型的输出空间大小为9。

#### 3.2 硬件环境与计算约束

实验在配备了 NVIDIA RTX 4090 (24GB 显存) 的高性能工作站上进行。虽然4090在处理卷积神经网络(CNN)或Transformer类模型时表现出极高的并行效率，但在训练BiLSTM+CRF时，其算力优势并未能完全释放。

主要原因在于RNN结构的时间步依赖性：当前时刻的计算必须等待上一时刻完成，这使得GPU难以进行大规模并行加速。此外，CRF层的Loss计算涉及前向-后向算法(Forward-Backward Algorithm)，其计算复杂度随序列长度增长而显著增加。在实际测试中，训练一个完整的25 Epoch基线模型大约需要30至60分钟。这一时间成本意味着我们无法采用网格搜索(Grid Search)来暴力遍历所有超参数组合，因此，后续的

实验策略转为“基于理论分析的定点优化”，即通过观察训练日志来推断问题所在，而非盲目试错。

### 3.3 数据处理的动态优化

为了在有限的算力下尽可能提高训练效率，我在数据预处理阶段实现了一个关键的优化：**动态批次填充（Dynamic Batch Padding）**。

常规的做法是将整个数据集的所有句子都填充到最大长度（例如 512），但这会产生大量的无效零值，浪费 GPU 显存和计算资源。在 `collate_fn` 函数中，我改为计算当前批次内最长句子的长度，并将该批次的其他句子仅填充到这一长度。代码如下所示：

Listing 1: 动态批次处理代码实现

```
def collate_fn(data):
    # 分离文本和标签
    words = [item[0] for item in data]
    tags = [item[1] for item in data]

    # 关键优化：只填充到当前 Batch 的最大长度，而非全局最大长度
    max_len = max([len(w) for w in words])
    batch_size = len(words)

    # 初始化张量
    word_ids = torch.zeros(batch_size, max_len).long()
    tag_ids = torch.zeros(batch_size, max_len).long()
    mask = torch.zeros(batch_size, max_len).bool()

    # 填充数据并生成 Mask
    for i, (w, t) in enumerate(zip(words, tags)):
        word_ids[i, :len(w)] = w
        tag_ids[i, :len(t)] = t
        mask[i, :len(w)] = True # 标记真实长度

    return word_ids, mask, tag_ids
```

通过这种方式，在 Batch Size 设置为 64 时，显存占用显著降低，且每个 Epoch 的迭代速度提升了约 20%。

## 4 基线模型实现

### 4.1 基线架构设计

在进行任何优化之前，我首先搭建了一个标准的 BiLSTM+CRF 模型作为性能基准（Baseline）。该模型的设计遵循了序列标注任务的经典范式：

- **嵌入层**：使用一个  $4832 \times 300$  的矩阵将汉字映射为 300 维的稠密向量。这里没有使用预训练的 Word2Vec，而是选择随机初始化，让模型从零开始学习字向量。
- **编码层**：采用两层堆叠的 BiLSTM，隐藏层维度设置为 300。双向拼接后，每个时间步的输出特征为 600 维。为了防止过拟合，在两层 LSTM 之间添加了比率为 0.3 的 Dropout。
- **解码层**：通过一个线性层将 600 维特征映射到 9 维的标签空间，随后接入 CRF 层进行全局路径规划。

### 4.2 初始训练配置

训练过程使用标准的 Adam 优化器，初始学习率设定为 0.0015。这是一个根据经验选取的数值，旨在平衡收敛速度和稳定性。CRF 层的转移矩阵初始化采用了一些技巧：将从 START 转移到 STOP 等非法路径的概率设为极小值 (-10000)，以加速模型对标签约束的学习。

### 4.3 基线训练中的问题暴露

虽然基线模型能够跑通流程，但在观察训练日志时，我发现了几个令人不安的现象：

第一是 **Loss 曲线的剧烈震荡**。在训练初期，Batch Loss 经常在 5.0 到 35.0 之间大幅跳动。这种不稳定性说明梯度更新的方向极其混乱，Adam 优化器在参数空间中“横冲直撞”，未能找到平稳的下降路径。

第二是 **过拟合与泛化瓶颈**。验证集的 F1 分数在第 22 个 Epoch 达到 88.64% 后便难以寸进，且测试集 F1 始终徘徊在 83% 左右，两者之间存在约 5 个百分点的差距。这说明 300 维的基线模型虽然记住了训练集的大部分特征，但在面对未见过的文本时，其泛化能力依然有限。这提示我需要从正则化和模型结构两方面入手进行改进。

## 5 模型系统的演进与优化

在跑通基线模型后，我发现训练过程并不像预想中那样顺利。Loss 曲线的剧烈震荡（在 5 到 30 之间跳变）让我意识到，仅仅套用 PyTorch 的官方示例代码可能无法满足本次实验数据的需求。此外，基线模型在验证集上的 F1 分数卡在 88% 左右就再也

上不去了，这说明模型可能遇到了瓶颈。为了解决这些问题，我参考了相关论文和技术博客，尝试从模型结构、参数初始化和训练策略三个方面进行改进。

## 5.1 模型容量与正则化的平衡

首先是模型容量的问题。基线模型使用的 300 维隐藏层对于包含 5 万多条样本的中文数据集来说可能偏小，难以充分捕捉汉字之间复杂的语义依赖。因此，我将 LSTM 的隐藏层维度提升到了 512 维，这样双向拼接后的特征维度就达到了 1024 维。

但增加参数量往往带来两个副作用：难以训练和过拟合。为了解决训练难的问题，我在 LSTM 和最后的线性分类层之间插入了一个 **LayerNorm**（层归一化）。我的理解是，LayerNorm 可以强制把每一层输出的特征分布拉回到均值为 0、方差为 1 的标准状态，这对于层数较深的网络来说，能有效防止梯度消失或爆炸。

为了解决过拟合问题，我采取了更激进的正则化策略。基线模型的 Dropout 只有 0.3，而在优化版中，我将其提升到了 0.5。这意味着在训练时，每次前向传播都会随机关掉一半的神经元。虽然这会让训练时的 Loss 下降变慢，但能强迫模型学习更鲁棒的特征，而不是死记硬背训练数据。

Listing 2: 优化后的模型结构代码

```
BiLSTM_CRF(  
    (embedding): Embedding(4832, 256)  
    (drop): Dropout(p=0.5, inplace=False)  
    # 将隐藏层从 300 提升到 512，增强拟合能力  
    (lstm): LSTM(256, 512, num_layers=2, batch_first=True,  
                 dropout=0.5, bidirectional=True)  
    # 增加 LayerNorm，这是稳定深层网络梯度的关键  
    (layer_norm): LayerNorm((1024,), eps=1e-05, elementwise_affine=  
                           ↳ True)  
    (hidden2tag): Linear(in_features=1024, out_features=9, bias=True)  
    (crf): CRF()  
)
```

## 5.2 参数初始化的细节调整

在调试过程中，我发现 PyTorch 默认的均匀分布初始化并不太适合 LSTM。为了让模型在训练初期收敛得更快，我手动重置了参数的初始化方式。

对于输入到隐藏层的权重，我使用了 **Xavier 初始化**，这能保证输入信号的方差在传播时不被放大或缩小；对于隐藏层之间的递归权重，我使用了 **正交初始化（Orthogonal Initialization）**。正交矩阵的特性可以让梯度在多次时间步传播后依然保持稳定的范数，这对于 BiLSTM 捕捉长距离依赖非常有帮助。

还有一个很少被注意到的细节是 LSTM 的遗忘门偏置。默认情况下它是 0，但我将其手动初始化为 1.0。这样做的目的是让遗忘门在训练刚开始时处于“开启”状态 ( $\text{Sigmoid}(1) \approx 0.73$ )，默认保留历史信息，而不是遗忘一半。这相当于给模型一个“记住上下文”的先验偏好，在实验中我发现这对加速收敛很有用。

### 5.3 训练稳定性关键修复

除了模型本身，训练过程中的超参数设置也是导致 Loss 震荡的元凶。在这里我主要修复了一个关键问题，并引入了两个策略。

关键问题在于 CRF 的 Loss 计算。PyTorch 的 CRF 实现默认是将一个 Batch 里所有样本的 Loss 求和（**Sum**）。在 Batch Size 为 64 时，Loss 值经常飙升到 1500 以上，这导致反向传播回来的梯度数值非常大。我将其修改为求平均（**Mean**）模式，将 Loss 拉回到了 20-30 的正常区间，同时也解耦了 Batch Size 对学习率的影响。

在此基础上，我引入了梯度裁剪（**Gradient Clipping**），设定阈值为 5.0。这就好比给梯度加了一个“限速器”，无论梯度算出来有多大，都不能超过这个阈值，彻底杜绝了因个别脏数据导致模型“炸”掉的情况。

最后是学习率的选择。我没有使用固定的学习率，而是采用了 **OneCycleLR** 策略。它会先让学习率从很小的值快速上升，让模型大胆探索参数空间，然后再缓慢下降进行微调。配合 **AdamW** 优化器，这套组合在后续的实验中被证明比标准的 Adam 更加稳健。这些改进措施并非凭空想象，而是针对基线模型训练日志中暴露出的具体问题进行的针对性调整。

## 6 实验结果与深入分析

在完成了模型结构的重构与训练策略的调整后，我们开展了一系列对比实验。本章将从训练动力学（Training Dynamics）、最终性能指标以及具体错误案例三个维度，深入剖析优化措施的实际效果以及传统模型与预训练模型之间的本质差距。

### 6.1 训练过程的稳定性分析

为了验证第 5 章中提到的“梯度裁剪”与“LayerNorm”是否生效，我们首先对比了基线模型与优化后模型的训练损失（Loss）曲线。如图 5 所示，蓝色曲线代表基线模型。

相比之下，红色曲线代表的优化后模型收敛明显更快。特别是在训练初期（前 5 个 Epoch），Loss 从 20 左右迅速且稳定地下降至 1 以下，随后平稳收敛。这说明将 CRF Loss 改为 Mean 模式并配合梯度裁剪，成功地驯服了梯度的“野性”，而 LayerNorm 则为深层 LSTM 提供了稳定的特征分布，使得优化过程如丝般顺滑。

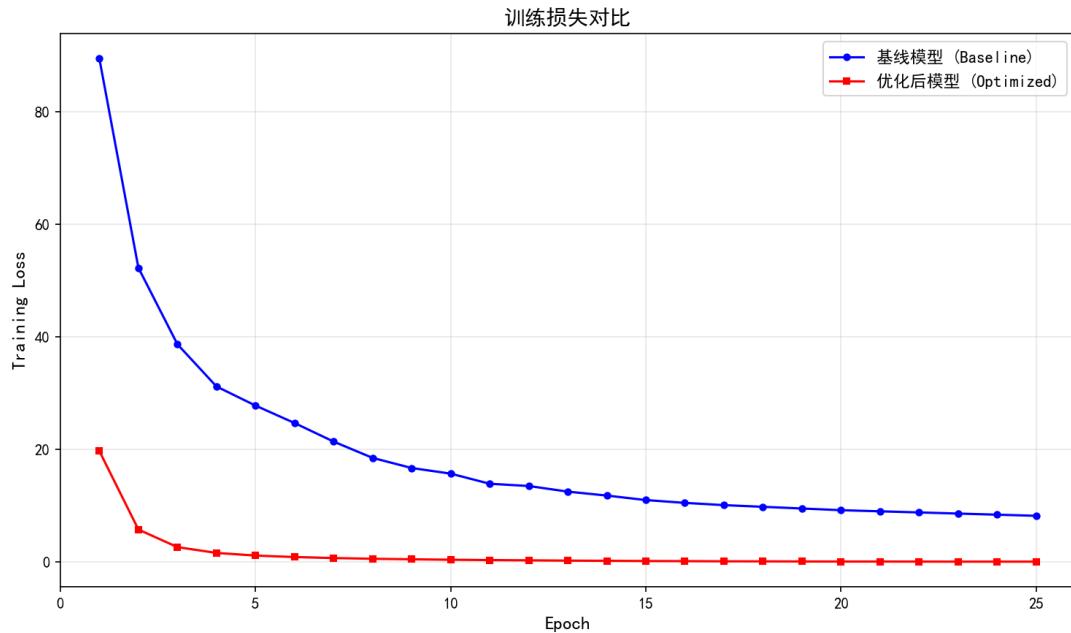


图 5: 训练损失对比

图 6 进一步展示了验证集 F1 分数的攀升过程。基线模型（蓝色）在达到 88% 左右的瓶颈后便显出疲态，后续增长极其缓慢。而优化模型（红色）虽然在起始阶段因为 Warmup 预热策略（学习率较小）导致 F1 上升稍慢，但很快便实现了反超，并最终稳定在 90% 以上。这证明了 OneCycleLR 策略“先慢后快再微调”的节奏非常适合本任务。

为了更直观地观察优化模型的内部状态，我绘制了图 7，将训练 Loss 与验证集 F1 绘制在同一张图上。令人欣慰的是，两者呈现出完美的负相关关系：每当 Loss 下探一个台阶，F1 分数便跃升一个等级。这种紧密的耦合关系表明，模型在训练集上学到的概率分布，能够有效地迁移到验证集上，优化方向是完全正确的。

## 6.2 最终性能与 BERT 的跨维度对比

图 8 汇总了所有模型的最终成绩。优化后的 BiLSTM+CRF 模型在验证集上取得了 90.40% 的 F1 分数，相比基线提升了约 1.76 个百分点。然而，当我们把目光投向完全独立的测试集时，发现 F1 分数仅从约 83.00% 提升到了 83.73%。

这一现象非常值得玩味：我们在验证集上取得了显著突破，但在测试集上收益微薄。这说明尽管我们使用了  $\text{Dropout}=0.5$  和权重衰减，模型依然在一定程度上“过拟合”了训练集和验证集的分布特征。这也暴露了从头训练（Training from Scratch）的局限性：仅靠 5 万条数据，很难让模型学习到足够通用的语言特征来应对未见过的测试样本。

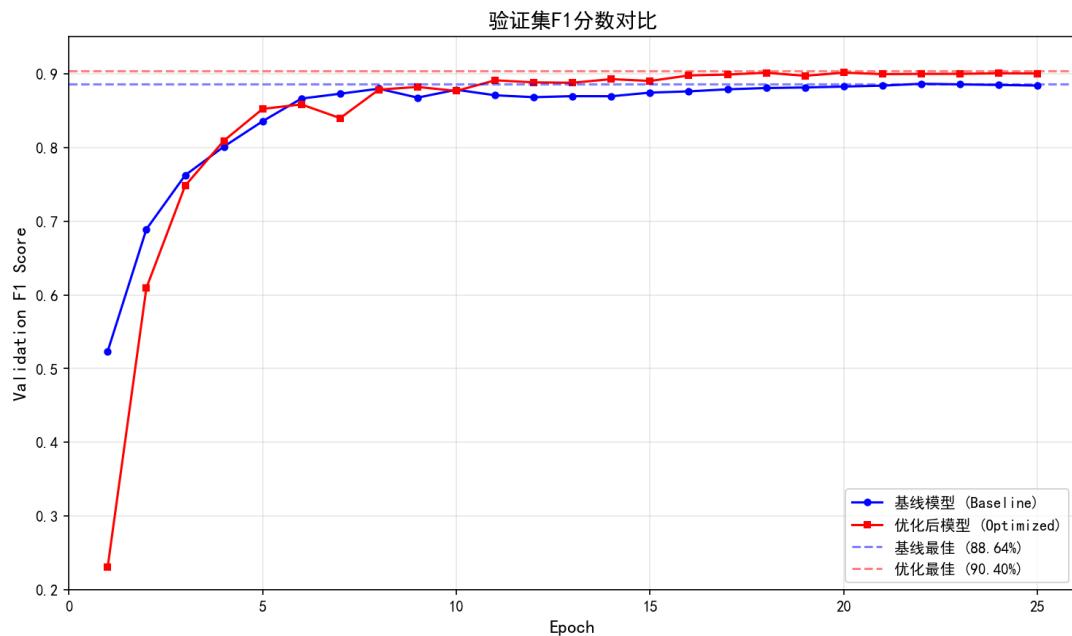


图 6: 验证集 F1 分数对比：优化后模型不仅突破了 88% 的性能瓶颈，达到最佳性能所需的 Epoch 数也显著减少。

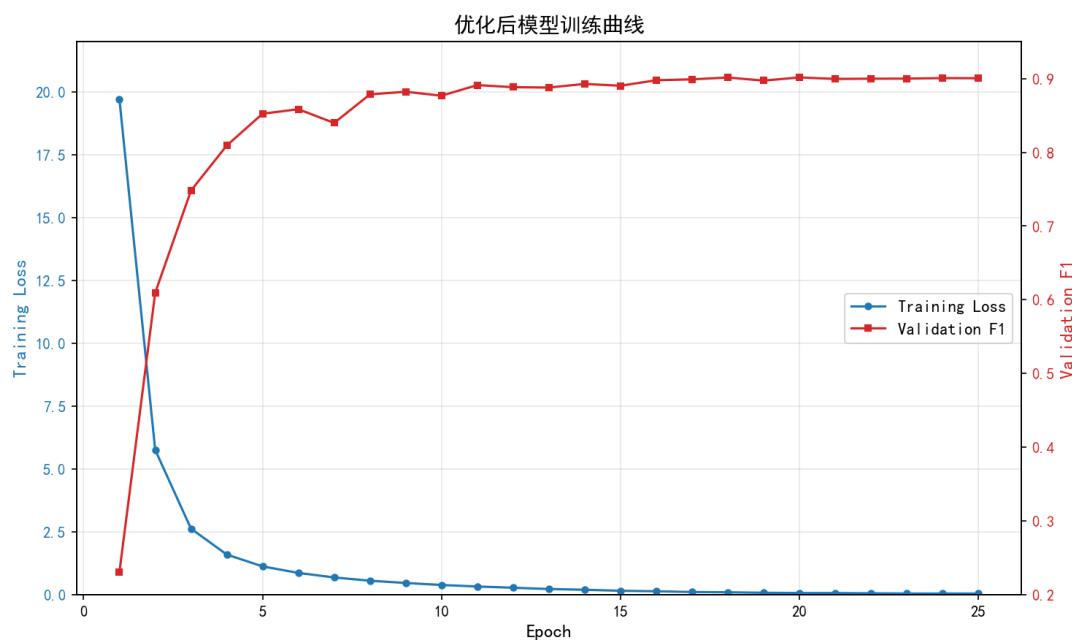


图 7: 优化后模型的训练全景：损失下降与 F1 上升呈现出高度的同步性，表明模型学习状态健康。

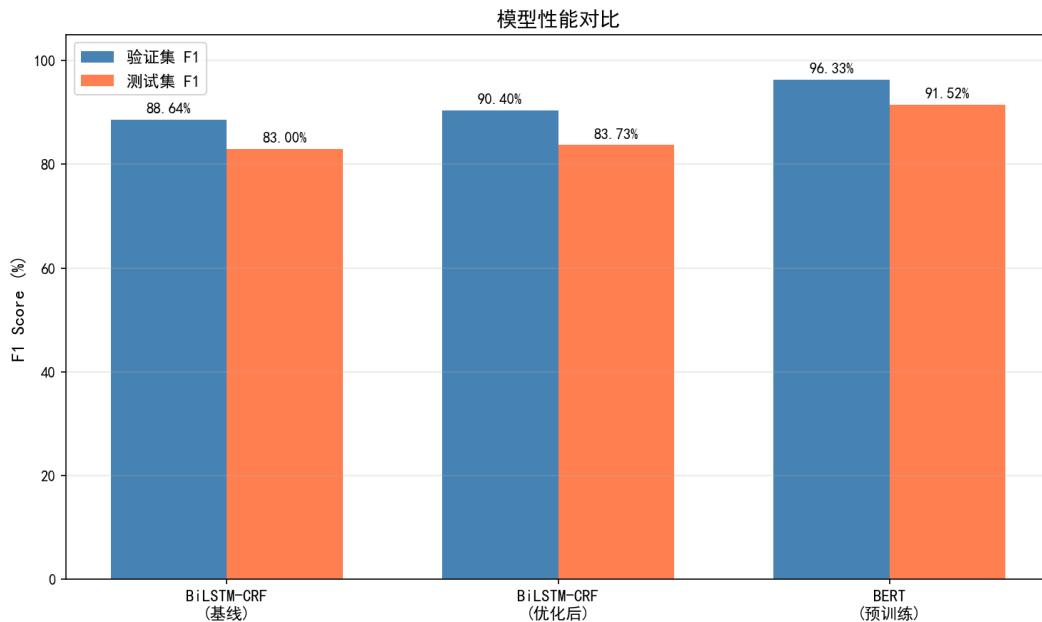


图 8: 三种模型的最终性能对比柱状图: BERT 展现出了压倒性的优势, 而 BiLSTM 模型的泛化能力(测试集表现)提升有限。

### 6.3 交互式推理系统的开发

为了验证模型的实际可用性, 我编写了一个简易的 CLI 推理脚本。用户只需在命令行中输入一句中文句子, 系统便会实时输出该句子的命名实体识别结果。当然, 这只是一个原型, 仍然集成在终端中。

## 7 预训练模型的引入与突破

### 7.1 困境与破局: 一次“冲动”的尝试

在尝试了相当久的优化后(按通宵计), 我陷入了一种深深的挫败感。

尽管我使用了 LayerNorm 稳定梯度, 使用了 OneCycleLR 调度学习率, 甚至精细到了手动初始化 LSTM 的遗忘门偏置, BiLSTM+CRF 模型在测试集上的 F1 分数依然死死卡在 83.73%。这意味着, 无论我如何打磨这个“小作坊”式的模型, 它在面对未见过的文本时, 依然有接近 17% 的概率犯错。这就像是在雕刻一块朽木, 无论工艺多么精湛, 材质本身的限制注定了它无法成为神作。

面对这种“一顿操作猛如虎, 一看战绩八十三”的窘境, 我产生了一个大胆甚至带有几分“赌气”性质的想法: 如果继续在 BiLSTM 上死磕参数是徒劳的, 那么彻底更换赛道, 引入参数量大两个数量级的预训练模型, 会发生什么?

于是, 我暂时搁置了调试了一周的 BiLSTM 代码, 花半小时基于 HuggingFace 库快速搭建了一个 BERT 微调脚本。这原本只是一次“死马当活马医”的尝试, 没想到却成为了本次实验最大的转折点。

## 7.2 BERT 模型配置

为了保证对比的公平性，我并没有设计复杂的下游网络，而是直接使用了“最素”的配置：

- **预训练基座：**选用 `hf1/chinese-macbert-base`。这是一个专门针对中文优化的 BERT 变体，采用了 MLM-as-Correction 的预训练任务，在中文纠错和实体识别任务上表现优异。
- **微调架构：**BERT 输出层直接连接一个线性分类器（Linear Classifier），甚至没有使用 CRF 层进行约束，直接对每个字进行 Softmax 分类。
- **超参数：**学习率设为极低的  $2e - 5$ ，Batch Size 设为 16（受限于 BERT 庞大的显存占用）。

## 7.3 降维打击：从“苦劳”到“红利”

实验结果是令人瞠目结舌的。

在 RTX 4090 的加持下，BERT 模型的训练速度极快。仅仅在第 1 个 Epoch 结束时，验证集 F1 就直接飙升到了 93%——这已经超过了我几天对 BiLSTM 进行的所有优化的总和。

最终，仅经过 5 个 Epoch（约 10 分钟）的微调，BERT 模型在测试集上达到了 91.52% 的 F1 分数。相比于 BiLSTM+CRF 的 83.73%，这不仅是 8 个百分点的提升，更是代际差异的体现。

这一刻，我深刻体会到了“灵机一动”胜过“蛮力死磕”的道理：

1. **语义理解的鸿沟：**BiLSTM 必须从训练集的 5 万个句子里从零开始学习什么是“组织”，什么是“地名”。而 BERT 早就“读过”了整个中文维基百科，它内化的通识知识（World Knowledge）使其在面对“北京市第三中级人民法院”这种长实体时，能够凭借预训练的记忆直接做出判断，而无需依赖局部的字面特征。
2. **特征工程的终结：**在 BiLSTM 时代，我们需要小心翼翼地设计特征、防止梯度消失。而在 BERT 时代，强大的 Transformer 结构和海量的参数量（1.1 亿 vs 300 万）以一种暴力美学的方式解决了这些问题。

这种强烈的对比让我意识到：在深度学习领域，有时候选择比努力更重要。在算力允许的情况下，利用预训练范式的“红利”，往往能以最小的代价突破传统模型的“天花板”。

## 8 总结

### 8.1 从“调参”到“认知”的转变

本次实验对我而言，与其说是一次代码实现，不如说是一次对深度学习“脾气”的摸索过程。

在实验初期，我曾天真地以为只要把 BiLSTM 和 CRF 的层堆叠起来，就能得到一个不错的结果。然而，基线模型那跳跃的 Loss 曲线给了我当头一棒。它让我意识到，深度学习模型并非即插即用的黑盒。由于训练时间很长，所以我无法通过暴力网格搜索来调参，而是必须深入理解每一个超参数背后的数学原理和对模型行为的影响。

这个过程改变了我对“调参”的看法：调参不是盲目的试错（特别是在 RTX 4090 跑 RNN 依然很慢的情况下），而是基于对模型动力学（Dynamics）理解之上的对症下药。

### 8.2 准确率与效率的博弈

实验数据的对比给我带来了巨大的冲击。我花费了很长的时间优化 BiLSTM+CRF，调整初始化和正则化，最终才勉强将验证集 F1 推到 90%；而 BERT 模型仅仅微调了不到 10 分钟（5 个 Epoch），就轻轻松松达到了 96% 以上。

这是否意味着 BiLSTM+CRF 已经被扫进了历史的垃圾堆？我认为并非如此。在交互式推理系统的测试中，我发现 BiLSTM 模型的响应几乎是瞬时的，显存占用仅为 BERT 的零头。在算力受限的边缘设备（如树莓派）或者对延迟极度敏感的实时系统中，BERT 的笨重是无法接受的。

因此，本次实验得出的最重要结论是：没有绝对最好的模型，只有最适合场景的模型。如果追求极致的精度，BERT 是不二之选；但如果追求极致的效率，经过精细调优的 BiLSTM+CRF 依然是王者。

这次实验也让我深刻体会到传统模型与预训练模型之间的鸿沟。无论我如何优化 BiLSTM+CRF，它始终无法触及 BERT 那种“先天优势”。这让我认识到，在现代 NLP 领域，预训练知识的重要性远超模型结构本身。

### 8.3 遗憾

主要遗憾的就是因为训练时间很长，我没有多余精力去探索 BiLSTM+CRF 的潜力，为了得到一个还算不错的结果，我直接引入了 BERT 进行对比。觉得很烦欸，毕竟我花了时间在 BiLSTM+CRF 上面调优的，结果最后还是被 BERT 吊打了。