在机器学习之中，我们要把获得的文本变成机器码以供学习
那么首先，要对文本进行分词处理，具体来说，就是获得需要编码的最小文本单位
后面embedding就是将这些最小文本每一个作为一个整体来编码
值得注意的是：最原始的文本处理方式就是直接划分成为char / word

- 缺点：char需要处理的序列长度太长/word不能灵活的处理单词变形，词汇表可能会爆炸
- 优点：char将词汇表保持在很小的尺寸/word便于处理和理解，上下文长度不长
  我们的目的是找到一种分词办法，使得它比word更加高效
  也就是word的基础上，获得处理复杂单词的能力，能够应对单词变形

目录

一、预分词算法

对于中文服务器选手，当然要明白对于中文的处理！！

关键挑战：

- 歧义切分：`"结婚的和尚未结婚的"` → `结婚/的/和/尚未/结婚/的` vs `结婚/的/和尚/未/结婚/的`
- 未登录词识别：`"双减政策"`（新词）
- 专有名词保留：`"北京市海淀区"`（地名）

（1）使用查表法（MM）

- 策略：使用百万级词条**贪婪匹配**最长的匹配词条
- **示例**：
  词典 = `["北京大学", "北京", "大学"]`
  输入：`"北京大学"` → 输出：`["北京大学"]`（优先匹配最长词）
- **优点**：
  - 能够匹配大部分单词
  - 能够匹配专有名词

- **缺陷**：
  - 无法处理未登录词（`"量子计算"`→拆为`["量","子","计","算"]`）
  - 歧义场景失效：`"使用户满意"` → 错误匹配`["使用", "户", "满意"]`

（2）基于统计的序列标注（CRF/HMM）

- 策略：将分词问题变成学习中文**单词边界分类**问题，需要借助深度学习

（2.1）思路突破

1. token和label问题：
   - **标签体系**（BIES）：
     - `B`：词语起始字
     - `I`：词语中间字
     - `E`：词语结束字
     - `S`：单字词
   - **特征工程：**
     - 训练字嵌入向量，能够使用skip-gram等模型
2. 模型设置：
   - **编码器（Encoder）**：学习字符的上下文表示
     - 主流选择：**BiLSTM**（捕捉长距离依赖）或 **Transformer**（并行高效）
   - **解码器（Decoder）**：预测每个字符的标签
     - **SoftMax**：独立预测每个位置标签（忽略标签间依赖）
     - **CRF层**：**必选组件**！强制标签转移合法（如I不能接S）

```
Error parsing Mermaid diagram!

Cannot read properties of null (reading 'getBoundingClientRect')
```

（2.2）全流程分析+model

步骤1：数据预处理

- **输入文本**：`"人工智能改变世界"`
- **字符级拆分**：`["人", "工", "智", "能", "改", "变", "世", "界"]`
- **标签标注**（BIES）：

字符：	**人**	**工**	**智**	**能**	**改**	**变**	**了**	**世**	**界**
标签	B	E	B	E	B	E	S	B	E

步骤2：特征工程

- **字符嵌入（Char Embedding）：**
  - 初始化：随机向量 或 预训练字向量（如中文Word2Vec）

步骤3：模型构建

```
class BiLSTM_CRF(nn.Module):
    def __init__(self, vocab_size, tag_to_ix, embedding_dim, hidden_dim):
        """
        初始化模型。

        参数:
        - vocab_size: 词汇表的大小。
        - tag_to_ix: 标签到索引的映射字典。
        - embedding_dim: 词嵌入的维度。
        - hidden_dim: LSTM隐藏层的维度。
        """        super(BiLSTM_CRF, self).__init__()
        self.embedding_dim = embedding_dim
        self.hidden_dim = hidden_dim
        self.vocab_size = vocab_size
        self.tag_to_ix = tag_to_ix
        self.tagset_size = len(tag_to_ix)

        # 1. 词嵌入层
        self.embedding = nn.Embedding(vocab_size, embedding_dim)

        # 2. BiLSTM层
        #    - input_size: embedding_dim
        #    - hidden_size: hidden_dim // 2 (因为是双向的，两个方向拼接)
        #    - num_layers: 1        #    - bidirectional: True
self.lstm = nn.LSTM(embedding_dim, hidden_dim // 2,
                            num_layers=1, bidirectional=True)

        # 3. 线性层
        #    将BiLSTM的输出映射到标签空间，得到发射分数
        self.hidden2tag = nn.Linear(hidden_dim, self.tagset_size)

        # 4. CRF层
        #    使用 torchcrf 库
        self.crf = CRF(self.tagset_size, batch_first=True)

    def _get_lstm_features(self, sentence):
        """
        此函数从输入句子中提取 BiLSTM 特征（发射分数）。
        """        # 词嵌入
```

```
        embeds = self.embedding(sentence).view(len(sentence), 1, -1)
        # LSTM 前向传播
        lstm_out, _ = self.lstm(embeds)
        # 调整形状以匹配线性层
        lstm_out = lstm_out.view(len(sentence), self.hidden_dim)
        # 映射到标签空间
        lstm_feats = self.hidden2tag(lstm_out)
        return lstm_feats

    def forward(self, sentence, tags, mask=None, reduction: str = 'sum'):
        """
        计算CRF层的负对数似然损失。

        参数:
        - sentence: 输入的句子序列 (tensor of word indices)。
        - tags: 真实的标签序列 (tensor of tag indices)。
        - mask: 句子的掩码，用于处理padding (1表示真实词，0表示padding)。
        - reduction: 损失的聚合方式 ('sum', 'mean', 'token_mean')。

        返回:
        - 损失值 (tensor)。
        """        # 从BiLSTM获取发射分数
        # 形状: (seq_length, num_tags)
        emissions = self._get_lstm_features(sentence)

        # 将形状调整为 (batch_size, seq_length, num_tags) 以匹配CRF层要求
        # 在这个例子中，batch_size = 1
        emissions = emissions.unsqueeze(0)
        tags = tags.unsqueeze(0)

        if mask is not None:
            mask = mask.unsqueeze(0).bool()

        # 调用CRF层的 forward 方法计算损失
        # 注意：CRF层返回的是负对数似然损失，所以我们需要取其相反数
        loss = -self.crf(emissions, tags, mask=mask, reduction=reduction)
        return loss

    def decode(self, sentence, mask=None):
        """
        使用维特比算法解码，找到最优的标签序列。

        参数:
        - sentence: 输入的句子序列 (tensor of word indices)。
        - mask: 句子的掩码。
```

```
        返回:
        - 最优路径的分数和路径本身 (list of tag indices)。
        """        # 从BiLSTM获取发射分数
        emissions = self._get_lstm_features(sentence)
        emissions = emissions.unsqueeze(0)  # 增加 batch 维度

        if mask is not None:
            mask = mask.unsqueeze(0).bool()

        # 调用 CRF 层的 decode 方法
        # 它会返回一个列表，其中包含每个序列的最优标签路径
        best_path = self.crf.decode(emissions, mask=mask)
        return best_path[0]  # 因为 batch_size=1，所以取第一个结果
```

CRF讲解

1. **理解**：模型对于输出的序列是有**特定顺序需求**的，例如B后面不能接上I，I后面不能接上E
   根据现有的知识，我们在序列处理之中学到了CRF，也就是对于标签输出的管理
   尽管模型能够很好的预测，但是，我们能够设置一些限制条件，这些条件当然是可以列举出来的（这是重点）进而来嵌入模型，惩罚模型对于不合理序列的预测
2. **理论讲解：CRF层的作用就是学习这些标签之间的依赖关系和约束**。它会学习到一个“**转移分数**”矩阵，该矩阵定义了从一个标签转移到另一个标签的合理性。

   - **高分转移：** `B → I` (非常合理)
   - **低分（甚至负分）转移：** `B → S` (非常不合理)

   通过引入这些约束，CRF层可以确保模型最终输出的标注序列是全局最优且逻辑上通顺的，而不仅仅是单个词的局部最优选择。
3. **作用原理：**
   CRF层主要做两件事：
   - **计算损失 (Loss Calculation)**：在训练阶段，CRF层不仅仅考虑模型对单个词的预测（这部分通常来自LSTM的输出，我们称之为 **发射分数 Emission Score**），还会结合标签之间的 **转移分数 (Transition Score)**。它会计算出所有可能的标注路径的总分，并使用最大似然估计来最大化“正确”标注路径的分数，同时最小化其他所有“错误”路径的分数。这使得模型在训练时就学会了标签间的转移规则。
   - **解码/预测 (Decoding)**：在预测阶段，当给定LSTM的输出（发射分数）后，CRF层不再是简单地为每个词选择概率最高的标签。相反，它会使用高效的 **维特比算法 (Viterbi Algorithm)**，结合发射分数和已经学好的转移分数，在所有可能的标注序列中，找出一条总分最高的路径作为最终的预测结果。这个过程保证了输出序列的合法性和最优性。
4. **数学公式：**
   发射分数，就是正常的损失，这个时候模型会输出所有标签的概率

转移分数，通过转移矩阵，从**标签** 学习该路径正确的排列顺序，并且同样使用最大似然算计算损失，从而得到最好的矩阵

$$Score(,) = \quad_{i=} mi(_i, _i) \ -_{i=} r(_i, _i)$$

$$\text{最大化预测目标} P() = \frac{ep(Score(,))}{ep(Score(,))} = \frac{\text{正确路径的得分}}{\text{所有其余路径的总得分}}$$

$$\text{损失函数：} Lo = -logP() = -(Score(,) - log()),\ \text{是所有} Score \text{的总和}$$

$$\text{前向算法：} (_i) = -_{=} -()P(_i)P(_i)$$

- `t` 是对应的层数，`α(X)` 是对应层数的以 X 结尾的概率 `Y^t` 是对应层数的label
  假设标签索引：B=0, I=1, E=2, S=3，**转移矩阵为：**

当前标签\下一标签	B(0)	I(1)	E(2)	S(3)
B(0)	-∞	0.8	0.2	-∞
I(1)	-∞	0.6	0.4	-∞
E(2)	0.7	-∞	-∞	0.3
S(3)	0.9	-∞	-∞	0.1

> **关键点**：
>
> - 合法转移有正值（如B→I, B→E）
> - 非法转移设为负无穷（-∞），如B→B, I→B
> - 数值初始化为可学习参数，训练过程中自动优化

总结一下，CRF层可以看作是在神经网络的输出和最终预测之间增加了一个“语法检查器”，这个检查器专门负责检查标签序列的合理性。

二、BPE (Byte Pair Encoding) 分词算法

核心思想： 从基础字符开始，**迭代合并**语料库中出现频率最高的**相邻符号对**，形成新的子词单元，直到达到目标词汇表大小。

（1）训练阶段（构建词汇表）：

输入： 大型文本语料库 + 目标词汇表大小 `vocab_size`
输出： **词汇表**（包含基础字符 + 合并生成的子词） + **合并规则列表**（Merge Rules）

1. **预处理与初始化：** (Tokenizer)
   - **文本归一化：** (pre_tokenizer：去除文本之中一无法识别的字符 & 规范化 & 处理空格)

- 小写化（可选，如 `BERT` 使用，`GPT` 不使用）
- Unicode 规范化（如 NFKC，将全角字符转半角，统一写法）
- 清理非法字符、控制字符
- **处理空格：** 将空格替换为特殊符号 `_` (U+2581) 或 `,`。**这是关键！** 它让算法能区分单词边界，尤其对无空格语言（如中文）至关重要。
  - 示例：`"natural language"` -> `"_natural _language"`

- **拆分基础单元：**
  - 将归一化后的文本拆分为**字符级**序列（包括 `_`）。
    - 示例：`"_natural"` -> `['_', 'n', 'a', 't', 'u', 'r', 'a', 'l']`
- **初始化词汇表：**
  - 统计所有**唯一字符**（包括 `_`）作为初始词汇表 `Vocab`。
  - 添加必要的 **特殊Token**：
    - `<unk>`：未知词
    - `<pad>`：填充
    - `<s>` / `</s>`：句子开始/结束 (可选)
    - `<mask>`：掩码 (BERT)
  - 此时 `Vocab = {'_', 'n', 'a', 't', 'u', 'r', 'l', 'g', 'p', 'o', 'c', 'e', 's', 'i', ... , '<unk>', '<pad>', ...}`

2. **迭代合并（核心循环）：**(获得核心的分词规则)
   - **统计相邻符号对频率：**
     - 遍历整个语料库，统计**当前词汇表下**所有**相邻符号对**（bigram）出现的频率。
     - 以初始状态（字符级）处理 `"_natural"`：
       - 符号序列：`['_', 'n', 'a', 't', 'u', 'r', 'a', 'l']`
       - 相邻对：`('_','n')`, `('n','a')`, `('a','t')`, `('t','u')`, `('u','r')`, `('r','a')`, `('a','l')`
     - 假设 `('a','t')` 在整个语料中出现频率最高（如 1500 次）。
   - **合并最高频对：**
     - 将最高频符号对 `('a','t')` **合并**成一个新符号 `"at"`。
     - 将 `"at"` **加入**词汇表 `Vocab`。
     - **更新语料库：** 将所有出现 `'a'` 后紧跟 `'t'` 的地方替换为 `"at"`。
       - `"_natural"` 更新为：`['_', 'n', 'at', 'u', 'r', 'a', 'l']`
       - `"_processing"`（假设存在）可能变为：`['_', 'p', 'r', 'o', 'c', 'e', 'ss', 'i', 'ng']` → `['_', 'p', 'r', 'o', 'c', 'e', 'ss', 'ing']` (如果 `('i','ng')` 也被合并)
   - **重复：**
     - 重新统计当前符号序列的相邻对频率。

- 例如，在新序列 `['_', 'n', 'at', 'u', 'r', 'a', 'l']` 中，新的相邻对有 `('_','n')`, `('n','at')`, `('at','u')`, `('u','r')`, `('r','a')`, `('a','l')`。
- 假设 `('n','at')` 现在频率很高（因为 `"nat"` 是常见组合），合并为 `"nat"` 加入词汇表。
- 更新语料：`"_natural"` -> `['_', 'nat', 'u', 'r', 'a', 'l']`

- **终止条件：** 循环执行，直到：
  - 词汇表大小 `len(Vocab)` 达到预设的 `vocab_size`。
  - 或没有更多可合并的相邻对（频率低于阈值）。

3. **最终输出：** （当前的tokenizer 已经配置了pre_tokenize 和 合并规则 两个算法，足够处理文本）
   - **词汇表 (Vocab)：** 包含所有基础字符、合并生成的子词、特殊Token。
   - **合并规则列表 (Merge Rules)：** 按合并顺序存储所有合并操作。**这是BPE的核心！**
     - 示例规则：`('a','t') → 'at'`, `('n','at') → 'nat'`, `('u','r') → 'ur'`, ...

- detail：对于空格在分词中的重要作用讨论：
  - 英文：
    1. 事实就是，分词必须建立在每一个单词之中，否则模型对于跨单词token的理解毫无意义，所以我们需要将分词建立在单词层面，意味着我们要分离每一个单词
    2. 从算法的角度思考，我们搞清楚思路：我们希望能够在单词层面进行编码，使得编码之后的token具有能够代表输入输出的实际含义，那么使用什么代表空格呢？实际上就是一些后缀，例如 `ly` , `cal` , `tion` , 那么模型如何利用这些后缀来输出空格呢？实际上在单词最后添加显式分割符号，不就能够使得模型学习到输出空格的方式？eg : 对于单词 `apple juice` 分词成为 `["apple</w>" , "app</w>"] -> ['a' ... 'e</w>']` and `['a', "p",'p</w>'] -> ['app' , 'le</w>']` `['app</w>']` 也就是模型能够知道，这个app是一个前缀，之后不应该输出空格休止，而后者虽然字面上也是app，但是它包含了休止符，可以作为一个whole单词或者后缀来看，这样就实现了没有显式后缀带来的难以区分的问题
    3. 在解码的时候，也能够根据 `</w>` 解码得到空格作为文本
  - 中文
    1. 在中文之中，并没有空格这一说，现在就要聚焦如何对于文本预处理，使得文本能够变成一个一个词组，以便我们添加后缀
    2. pre-tokenizer 直接影响最终编码的质量，已知的处理：简体字繁体字归一化，复杂文字的特殊处理，下面将开设章节讲解预分词的算法 ，请查找目录

（2）编码阶段（对新文本分词）：

输入： 新句子 + 训练好的词汇表 `Vocab` + 合并规则 `Merge Rules`
输出： Token序列（字符串 或 ID）

1. **预处理：** 应用与训练时相同的归一化（小写、NFKC）和**空格替换**（-> `_`）。
   - 示例输入：`"Natural language processing is fun!"`
   - 归一化+空格处理：`"_natural _language _processing _is _fun !"`
2. **拆分为字符序列：** `['_','n','a','t','u','r','a','l','_','l','a','n','g','u','a','g','e','_','p','r','o','c','e','s','s','i','n','g','_','i','s','_','f','u','n','!']`
3. **应用合并规则（贪婪最长匹配）：**
   - 按**合并规则在训练中出现的顺序**（或按子词长度从长到短），尝试将当前序列中的符号**尽可能合并**成词汇表中存在的子词。
   - **关键逻辑：**
     - 遍历 `Merge Rules` 列表（按训练时的合并顺序）。
     - 对当前序列，查找是否存在该规则对应的符号对。
     - 如果存在，将其合并。
     - **重复**应用所有规则，直到无法再合并。
   - **示例合并过程（简化）：**
     - 规则1：`('s','s') → 'ss'` → 处理 `"processing"` 中的 `['s','s']` -> `['ss']`
     - 规则2：`('i','n') → 'in'` → 处理 `"processing"` 中的 `['i','n']` -> `['in']` (但 `'in'` 可能已在Vocab)
     - 规则3：`('in','g') → 'ing'` → 处理 `['in','g']` -> `['ing']`
     - 规则4：`('p','r') → 'pr'`，`('pr','o') → 'pro'`，`('pro','c') → 'proc'`，`('proc','e') → 'proce'` ... 最终可能合并出 `'processing'` 作为一个Token（如果它在Vocab中）。
     - 对于 `"natural"`：应用规则 `('a','t') → 'at'`，`('n','at') → 'nat'`，`('nat','u') → 'natu'`，`('u','r') → 'ur'` → 最终可能拆分为 `['_','natural']`（如果 `'natural'` 在Vocab）或 `['_','nat','ural']`。
4. **处理未登录词：** 如果最终序列中存在不在 `Vocab` 中的符号（如罕见拼写错误），用 `<unk>` 替换。
5. **输出Token序列：**
   - 字符串Tokens：`['_natural', '_language', '_processing', '_is', '_fun', '!']`
   - Token IDs：通过查词汇表转换为整数序列，如 `[105, 42, 987, 25, 76, 7]`

```
from tokenizers import Tokenizer, models, pre_tokenizers, decoders, trainers

# 1. 初始化一个 BPE Tokenizertokenizer = Tokenizer(models.BPE())

# # 2. 设置预处理器：处理空格、小写、规范化
# tokenizer.pre_tokenizer = pre_tokenizers.Sequence([
#     pre_tokenizers.WhitespaceSplit(),        # 按空格分词（对英文）
#     pre_tokenizers.CharDelimiterSplit('_'),  # 显式添加_ (更推荐用 Whitespace
```

```
隐式处理）
# ])
# 或更通用：ByteLevel（自动处理空格、小写、Unicode）
tokenizer.pre_tokenizer = pre_tokenizers.ByteLevel(add_prefix_space=True)  # 添加_

# 3. 设置解码器：将 _ 转回空格，合并字节
tokenizer.decoder = decoders.ByteLevel()  # 或 decoders.WordPiece(prefix='_')
# 4. 训练
trainer = trainers.BpeTrainer(
    vocab_size=30000,
    special_tokens=["<unk>", "<pad>", "<s>", "</s>", "<mask>"],
    min_frequency=2,                    # 忽略低频词
    show_progress=True,
    initial_alphabet=pre_tokenizers.ByteLevel.alphabet()  # 初始包含256字节
)
files = ["E:\\code\\动手学深度学习\\data\\timemachine.txt"]
tokenizer.train(files, trainer=trainer)

# 5. 保存与加载
tokenizer.save("my_bpe_tokenizer.json")
tokenizer = Tokenizer.from_file("my_bpe_tokenizer.json")

# 6. 使用
text = "Natural language processing is fun!"
encoding = tokenizer.encode(text)
print(encoding.tokens)  # ['_Natural', '_language', '_processing', '_is', '_fun', '!']
print(encoding.ids)     # [105, 42, 987, 25, 76, 7]
```

（3）优点缺点：

- 优点：快速
- 缺点：基于统计学来进行分词，实际上不能够很好理解单词，一是只能得到统计过的，在对于新词或者专有名词的处理比较欠缺。并且低频词会被筛选，可能导致**过度拆分**的语义丢失。
  二是对于跨单词的短语，理解的并不好

三、WordPiece分词算法

WordPiece 与 BPE 的本质区别

特性	**BPE**	**WordPiece**
合并目标	最高频的字符对	**最大化语言模型似然**的字符对

特性	BPE	WordPiece
选择标准	频率统计	概率增益（Δ Likelihood）
训练方式	确定性的贪心合并	概率驱动的合并策略

> **关键创新**：WordPiece 用概率评估合并价值，而非单纯依赖频率。

（1）算法

- 构建初始的词汇表一步，是和BPE完全相同，包括拆分成为字符级别 + 添加特殊字符标记
- **合并迭代**：
  - BPE核心逻辑：抓住相邻的最大概率的“对”进行合并。然而忽略了一个问题，有时候这样基于概率的合并不能很好的表现语义，比如A 和 B 在文本之中是关键词，大量出现，但是他们两个没有组合意义，但是由于大量出现就会导致他们相邻的概率变得很高，从而被统计
  - 利用公式

$$\Delta L(u,v) = logP(uv) - [log(P(u) * P(v))]$$

$P(uv)$ 表示在文本中连续出现的概率，$p(u)$和$P(v)$ 表示单个字符出现在文本之中的概率

$$Score = \frac{P(uv)}{P(u)P(v)}$$

$$\Delta L(u,v) = log(Score)$$

这个分数之际上表示了这个短语对于这两个字符的重要性，能够一定程度上表现语义！选择最大的一堆 u，v 进行合并

（2）细节

前缀标记（`##` 符号）

- **作用**：标识非词首子词（如 `##ing`）
- **目的**：避免歧义（如 `"ing"` 作为独立词 vs 后缀）

```
[ "un", "##able", "##t", "re", "##ify", "[UNK]" ]
```

低频词语

- **MIN_COUNT** 经验值：
  - 中文：5-10（BERT中文版用5）
  - 英文：2-5
- **Google原始实现**：`min_count=10`（见BERT源码）

```
from tokenizers import Tokenizer, models, trainers

# 1. 初始化 WordPiece 模型
tokenizer = Tokenizer(models.WordPiece(unk_token="[UNK]"))

# 2. 配置训练器
trainer = trainers.WordPieceTrainer(
    vocab_size=30000,
    special_tokens=["[PAD]", "[UNK]", "[CLS]", "[SEP]", "[MASK]"]
)

# 3. 训练词表
tokenizer.train(files=["corpus.txt"], trainer=trainer)

# 4. 编码文本
output = tokenizer.encode("Hello! 你好吗?")
print(output.tokens)  # 输出: ["[CLS]", "hello", "!", "[UNK]", "[UNK]", "
[UNK]", "?", "[SEP]"]
```

四、Word2Vec词嵌入

承接前文我们既然已经分词成功，那么接下来编码
编码要遵循一下原则：

- 意思相近的单词点积大，反之亦然

（1）下采样

高频词例如the , an , a 在文本，没有很多实际含义，对于他们的文本理解通常要配合实词，也就是说，实词才是最好的训练素材。
如果不丢弃部分高频词，就会出现模型在高频词上投入大量训练成本，但是效果不佳
因此要下采样，丢弃的概率是出现频率的反比

```
def subsample(sentences, vocab):
    """下采样高频词"""
    # 排除未知词元'<unk>'
    sentences = [[token for token in line if vocab[token] != vocab.unk]
                 for line in sentences]
    counter = d2l.count_corpus(sentences)
    num_tokens = sum(counter.values())

    # 如果在下采样期间保留词元，则返回True
    def keep(token):
        return(random.uniform(0, 1) <
```

```
                math.sqrt(1e-4 / counter[token] * num_tokens))

    return ([[token for token in line if keep(token)] for line in sentences],
            counter)

subsampled, counter = subsample(sentences, vocab)
```

（2）模型的设置

skip-gram模型

这是一个根据上下文来获得词向量关联度的模型

- 核心思想：
  - 在词x规定的上下窗口之中，要是出现了词y，那就认为，词y和x具有相关性，那么xy词向量的点积应该变大
- 核心算法：
  1. 选择一个中心词，左右分别选取N个上下文单词
  2. 数学建模为：$P(\ _{i-2}, i-, i, i2\ _{i}) = _{i=i-} P(_{i})$
  3. 理解为出现中心词时，上下文词出现的概率，要是非常准确那就是1
  4. 模型表示为：使用**点积**来作为标准，$P(_{i}) = \frac{ep(_{i})}{_{=i=i-} ep(_{i})}$ 类似交叉熵的思想，要是出现了，P就要变大，点积就要变大，那就意味着关联性增加了
  5. 这个时候，模型想要优化的函数为$() = -_{=\ -} P()$
- 算法改进：
  - Q：由于计算softmax一步要计算对于所有向量的点积，**计算复杂度过高**
  - 数学建模为$P(D = _{c}, _{o}) = \frac{}{ep(-u_{o}v_{c})}$ 表示为是上下文的概率
  - 因此，优化的函数为$() = -_{=\ -} P(D = ,\ )$
  - 还要考虑一个重要思想，也就是负样本，否则模型会向着全部向量都有关联的方向发展
  - 对于每一个正样本，随机采样个负样本，集合称为
  - 数学建模更新为$P(_{i}) = P(D = ,) _{=,} P(D = ,)$

  损失函数为：

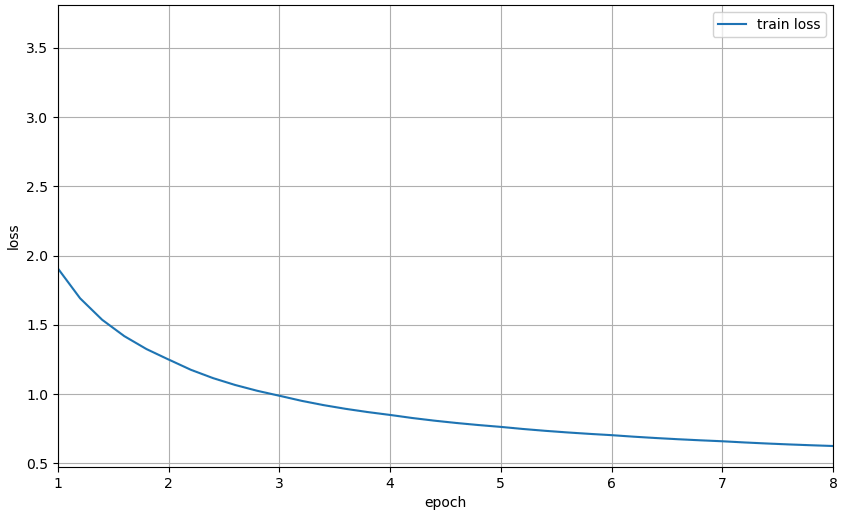
$$_{=} \frac{-(P)}{} = \quad _{=} \frac{-log(Sigmoid(uv)) - _{=,} log(-Sigmoid(v))}{}$$

  最终，对于单个中心词，我们要使得对于上下文的联合概率最大，反向传播优化词嵌入的权

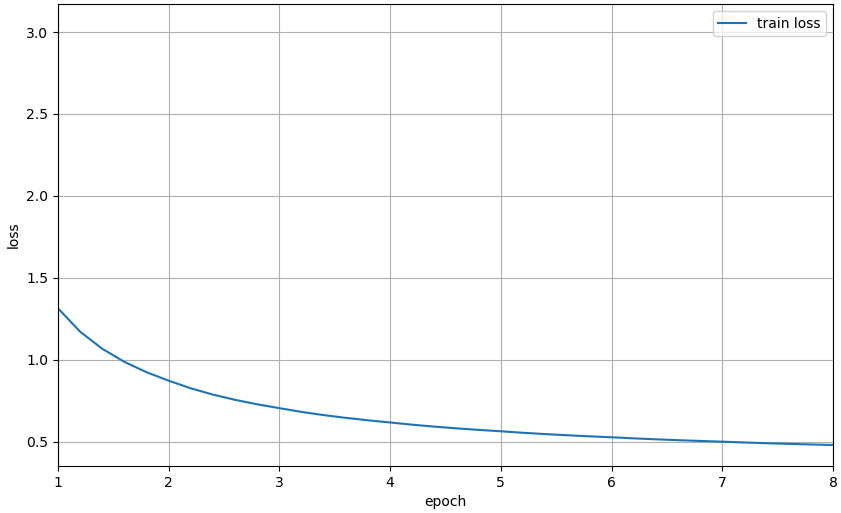
重

在实现的时候，把没有掩码的作为正向样本，掩码的作为负向样本，也就是二分类问题

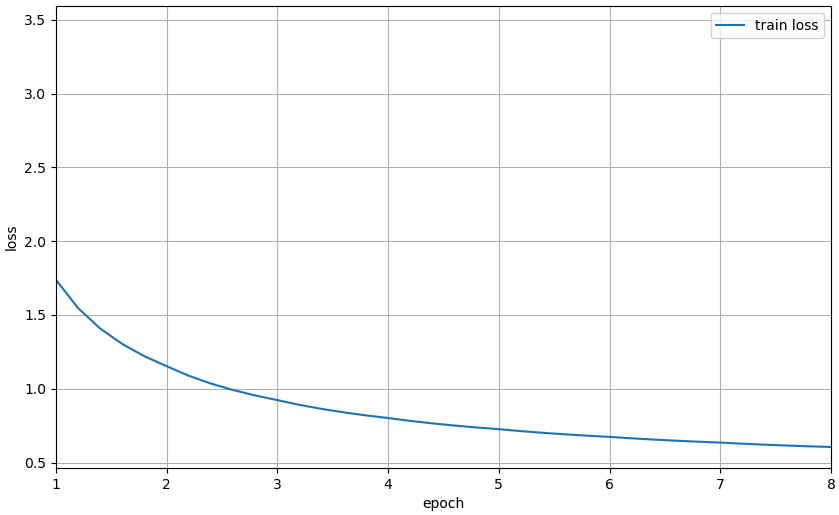
- 在BPE分词的结果上的训练



- 在没有编码上的损失函数



- 在WordPiece上训练的损失函数



我们发现在WordPiece之中分词的效果最显著

FastText 模型词向量表示详解

如何通过词的子字（subword）信息来构建词向量。这与传统的词向量模型（如 Word2Vec 中的 Skip-gram 或 CBOW）有所不同，传统模型通常将每个词视为一个独立的单元。

1. 引入子字特征（Subword Features）

图片首先以单词 "where" 为例，解释了如何获取一个词的子字特征。

- 特殊字符 < 和 >：
  为了区分一个词的开头和结尾，FastText 会在词的左右两边添加特殊字符 < 和 >。
  例如，对于单词 "where"，它会变成 `<where>`。
  - **作用：** 这有助于模型识别词的边界，并区分前缀和后缀。例如，"ing" 作为后缀和 "ing" 作为独立的词，其含义可能不同。
- 子字 n-gram：
  接下来，FastText 从这个添加了特殊字符的词中提取固定长度的子字 n-gram。图片中举例 n=3，即提取长度为3的子字。
  对于 `<where>`，当 n=3 时，提取的子字有：
  - `<wh` + `whe` + `her` + `ere` + `re>` + (特殊字词) `<where>`
- 子字BPE：
  他抛弃了n-gram的计算办法，而是利用我们已知的算法BPE，这样极大拓展了分词的灵活性，能够表示不同长度的子字，也保留了基于统计学得到的词义

2.词向量构建

- ：词 w 的所有子字集合
  “在fastText中，对于任意词 w，用 表示其长度在3和6之间的所有子字与其特殊子字的并集。”
  - **解释：** 这意味着对于一个词 w，我们会提取所有长度在3到6之间的 n-gram 子字，并且还会包含完整的词本身（特殊子字）。所有这些子字构成了集合 Gw。这个长度范围（3到6）是一个常见的设置，但可以根据任务和语料库进行调整。
- $_g$：子字 g 在词典中的向量
  图片中提到：“假设 $_g$ 是词典中的子字 g 的向量”。
  - **解释：** FastText 在训练时，会为词典中的每一个子字（包括各种 n-gram 和完整的词）学习一个对应的向量 $_g$。这些子字向量是模型的基本构建块。
- ：对于词 w
  构建词向量 的公式：

$$= {}_g{}^g$$

  - **解释：** 这个公式表明，一个词 w 的最终词向量 是其所有子字 g 对应的子字向量 $_g$ 的**简单求和**。
- **核心思想： 权重共享**，这就是 FastText 与传统词向量模型最大的不同之处。传统模型直接为每个词学习一个独立的向量。而 FastText 认为，一个词的含义可以通过其组成部分的子字来表示。通过将子字向量相加，FastText 能够：
  1. **处理未登录词（Out-of-Vocabulary, OOV）：** 如果一个词在训练集中没有出现过，但它的子字在训练集中出现过，FastText 仍然可以通过其子字向量的和来构建这个词的向量，从而对未登录词有更好的泛化能力。
  2. **捕捉词缀信息：** 比如 "running" 和 "walked" 都有动词词根，通过子字特征可以捕捉到这种形态学上的相似性。
  3. **减少模型参数：** 相比于为每个词学习一个独立向量，为子字学习向量可以大大减少需要学习的参数数量，尤其是在词汇量很大的情况下。

五、GloVe词嵌入

全局向量的词嵌入（Global Vectors for Word Representation，简称 GloVe）是一种用于学习词向量的**无监督学习算法**。GloVe 的目标是生成能够捕捉词语之间语义和句法关系的词向量，**使得语义相似的词在向量空间中彼此靠近**。

（1）解决的问题

在 GloVe 出现之前，主流的词向量学习方法主要有两类：

1. **基于局部上下文窗口的方法（如 Word2Vec 的 Skip-gram 和 CBOW）：**
   - **优点：** 能够捕捉词语的局部上下文信息，在语义相似性任务上表现良好。
   - **缺点：** 训练过程是基于局部窗口的迭代，没有直接利用**全局的词**共现统计信息。对于大型语料库，**训练效率可能受限**，且可能无法充分利用全局统计信息。

2. **基于全局矩阵分解的方法（如潜在语义分析 LSA）：**
    - **优点：** 直接利用了整个语料库的全局共现统计信息（通常是词-文档矩阵或词-词共现矩阵），能够捕捉词语的全局语义关系。
    - **缺点：** 无法很好地捕捉词语的细粒度语义和类比关系，因为它们通常基于降维技术，可能丢失一些局部上下文的语义细节。

GloVe 的设计目标是**结合这两类方法的优点**：既能像 Word2Vec 那样捕捉词语的局部上下文语义，又能像 LSA 那样利用全局的共现统计信息，从而在语义和句法类比任务上取得更好的表现。

（2）核心原理（数学推导）

GloVe 的核心思想是，**词向量之间的关系应该与它们的共现概率的对数比率相关联。**

3.1 词共现矩阵 X

首先，我们定义一个词共现矩阵 X。矩阵的元素$_i$ 表示词 在词 i 的上下文中**共现**的次数。

- **上下文窗口：** 在构建共现矩阵时，我们需要定义**一个上下文窗口**。例如，如果窗口大小为 C，那么当词 j 出现在词 i 的左右 C 个词之内时，我们就认为它们共现。
- **对称性或方向性：** 共现可以是无方向的（即 $_i$ 等于 $_i$），也可以是有方向的（例如，只考虑词 j 出现在词 i 之后的共现）。GloVe 论文中通常使用**对称的上下文窗口**。
- **距离衰减：** 通常，GloVe 会对距离较远的共现赋予较小的权重。如果词 j 距离词 i 的距离为 d，那么共现次数可以**加权为 1/d**。 （距离约近，权重越高，越有效）

3.2 共现概率 $P(\ \ i)$

基于共现矩阵 X，我们可以计算**词 j 出现在词 i 上下文中的概率**$P(\ \ i)$：

$$P(\ \ i) = \frac{i}{i}$$

其中 $_i$ 是词 i 的所有上下文词的共现总次数。

3.3 共现概率比率的意义

GloVe 认为，词向量的有效表示应该能够通过简单的数学操作（如点积）来反映词共现概率的对数。更重要的是，它关注**共现概率的对数比率**。

考虑三个词：i（目标词），j（上下文词），k（探测词）。
我们观察 $P(\ \ i)P(\ \ )$ 这个比率：

- 如果词 k 与词 i 相关，但与词 j 不相关，那么 $P(\ \ i)$ 会很大，而 $P(\ \ )$ 会很小，导致比率很大。

- 如果词 k 与词 j 相关，但与词 i 不相关，那么 $P(\ \ i)$ 会很小，而 $P(\ \ )$ 会很大，导致比率很小（接近0）。
- 如果词 k 与词 i 和词 j 都相关，或者都无关，那么比率会接近1。

举例：
假设我们有词 "ice" (冰), "steam" (蒸汽), "solid" (固体), "gas" (气体), "water" (水), "fashion" (时尚)。
令i = ice , j = steam

探测词 k	$P(\ \ ice)$	$P(\ \ em)$	$P(\ \ ice)P(\ \ em)$	**含义**
solid	高	低	很高	与 "ice" 相关，与 "steam" 不相关
gas	低	高	很低	与 "steam" 相关，与 "ice" 相关
water	高	高	1	与两者都相关
fashion	低	低	1	与两者都无关

这种共现概率比率的模式能够清晰地揭示词语之间的语义区别（例如“冰”和“蒸汽”在物理状态上的差异）。GloVe 的目标就是让**词向量能够捕捉这种模式**。

3.4 GloVe 的目标函数推导

GloVe 模型的出发点是：词向量的点积应该与它们共现的对数概率相关。
为了满足这种结构，GloVe 提出了以下形式：

$$_i = log_i$$

这个简化形式是说，两个词向量的点积应该等于它们共现次数的对数。

对于目标函数，我们希望存在一个函数 F 使得：

$$(_i, ,) = \frac{P(\ \ )}{P(\ \ i)}$$

其中 $_i, ,$ 分别是词 $i, ,$ 的词向量。 表示上下文词的向量，GloVe 中对每个词学习两个向量：一个作为中心词的向量 ，一个作为上下文词的向量 。
为了简化，我们希望 F 能够表示为 $(_i -)$ 的形式，因为**向量的差值**能够捕捉词之间的关系。
由于共现概率比率是乘法形式，而向量操作通常是加法形式，我们自然想到使用对数形式：

$$(_i, ,) = logP(\ \ i) - logP(\ \ )$$

为了处理 $P(\ \ i)P(\ \ )$ 的比率，我们希望：

$$_i - \ = logP(\ \ i) - logP(\ \ )$$

为了实现这个目标，GloVe 提出了以下损失函数（或目标函数）：

$$= \quad _{i=} \quad _{=}(_i)(_i \ _i \ - log_i)^2$$

让我们分解这个目标函数中的各项：

- V：词汇表的大小。
- $_i \ ^d$：中心词 i 的词向量（维度为 d）。
- ^{d}：上下文词 j 的词向量（维度为 d）。
  - GloVe 为每个词学习两个向量：一个作为中心词的 $_i$，一个作为上下文词的 。在训练结束后，通常会将 $_i$ 和 $_i$ **相加或取平均作为最终的词向量**。
- $_i$ ：中心词 i 的偏置项。
- ：上下文词 j 的偏置项。
  - 偏置项的作用是弥补 $_i$ 无法完全捕捉所有信息的情况，类似于线性回归中的截距。它们使得模型在 $_i$为零时也能有合理的表现。
- $_i$：词 i 和词 j 的共现次数。
- log_i：共现次数的对数。如果 $_i$=0，则 log_i 是负无穷，这在实际中需要特殊处理（通常通过 $(_i)$ 函数来避免）。
- $(_i)$：**权重函数（Weighting Function）**。这是一个非递减函数，用于给不同的共现次数赋予不同的权重。
  - **作用：**
    1. **处理 Xij=0 的情况：** 当 Xij=0 时，logXij 无意义。f(0) 被定义为0，这样共现次数为0的词对就不会对损失函数产生贡献。
    2. **降低高频共现的权重：** 像 "the", "a" 这样的停用词会频繁共现，但它们的共现信息可能不如低频词对那么有区分度。f(Xij) 可以限制这些高频共现的贡献，防止它们主导训练过程
    3. **提升低频共现的权重：** 确保即使是低频但有意义的共现也能被模型学习到。

  GloVe 论文中提出的权重函数形式为：

  f(x)=(x/x_{max})^α\ \ \ \ if\ \ x<x_{max}\ \ else\ \ 1

  - x 代表 $_i$。
  - $_m$ 是一个超参数，表示截断点（例如 $_m$=100）。当共现次数超过 $_m$ 时，权重恒定为1。
  - α 是一个超参数（通常取 α=0.75）。它使得权重函数在 $_m$ 时是非线性的，可以更好地平衡高频和低频共现。

目标函数的直观理解：

这个目标函数可以理解为：我们希望通过学习词向量 $_i$, 和偏置项 $_i$, ，使得它们的点积加上偏置项，尽可能地接近词 i 和词 j 共现次数的对数。平方误差项 $(\ \ )^2$确保了这种接近性。权重函数 $(_i)$ 则对不同频率的共现进行加权，以优化学习效果。

奇妙之处

- 能够使用()函数来缩放上下文影响力，这一点胜过skip-gram
- 能够看到全局的信息，建模具有考虑全局的特点，代价就是训练之前要对全文进行统计处理
- 即使建模可能导致词向量点积大于1，但是在实际使用过程之中会归一化处理，这不是一个难点

3.5 训练过程

GloVe 模型通过**随机梯度下降（Stochastic Gradient Descent, SGD）或其变种（如 Adam）来优化上述目标函数。在训练过程中，模型会迭代地更新词向量 $_i$, 和偏置项 $_i$, ，以最小化损失函数。

（3）GloVe 的优势

- **结合全局和局部信息：** GloVe 成功地将全局共现统计信息和局部上下文信息结合起来，使得生成的词向量在多种任务上表现优异。
- **高效性：** 相比于 Word2Vec，GloVe 可以直接利用预先计算好的共现矩阵，训练过程相对高效。
- **可解释性：** 目标函数基于共现概率的对数比率，这使得模型具有一定的可解释性，能够更好地理解词语之间的语义关系。
- **类比推理能力：** GloVe 向量在词语类比任务（如 "man:woman::king:?"）上表现出色，这表明它能够捕捉到词语之间更复杂的线性关系。