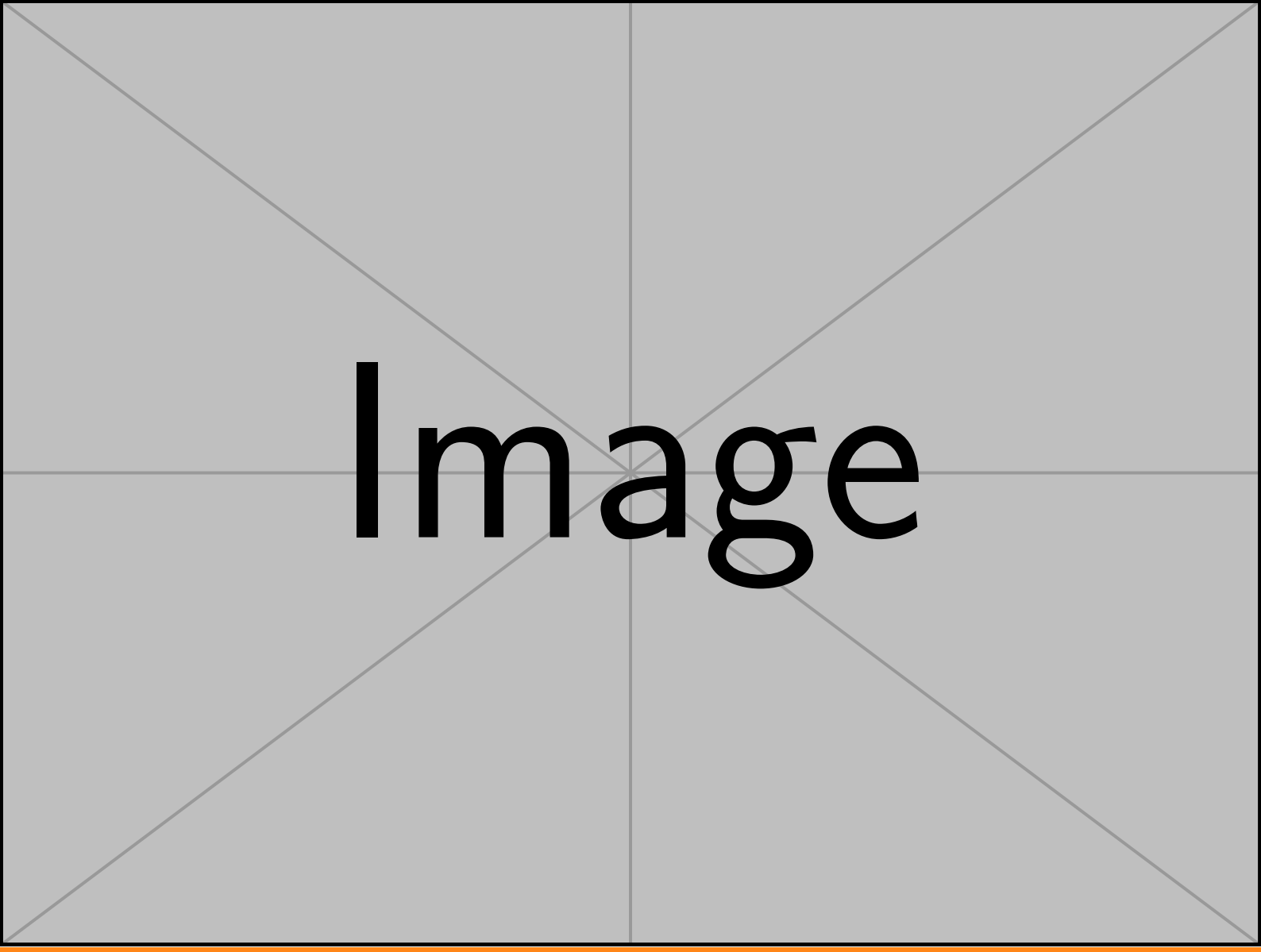


动手学深度学习

作者：黄佳炜

时间：November 7, 2025

目录

第 1 章 循环神经网络

1.1 引言

福州大学 FUZHOU UNIVERSITY

1.1.1 8.1 序列模型 - 完整导航

定义 1.1 (本节内容结构)

- 8.1.1 统计工具
- 8.1.2 训练
- 8.1.3 预测
- 8.1.4 小结

♣

1.1.1.0.1 开场：什么是序列？ 生活中的序列无处不在：

- 你说的每句话
- 一首歌的旋律
- 股票的价格变化
- 每天的气温
- 视频的每一帧
- 聊天记录

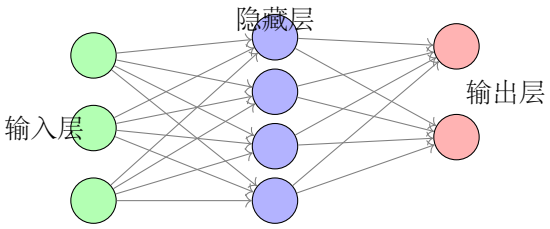
定理 1.1 (序列的特点)

1. 有顺序
   ”我爱你”≠”你爱我”
2. 有依赖
   后面依赖前面的信息
3. 变长
   长度不固定

♡

问题：如何用模型处理序列数据?

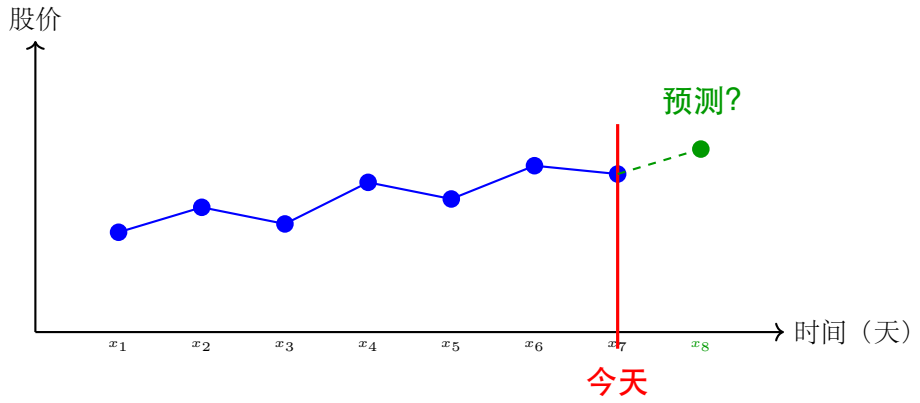
1.1.1.0.2 传统神经网络的局限 标准全连接网络（MLP）：



三大问题：

1. × **固定输入大小**：必须是固定维度的向量
2. × **无记忆能力**：每次输入独立处理，忘记历史
3. × **参数不共享**：处理序列不同位置用不同参数

1.1.1.0.3 问题示例：预测明天的股价



思考：如何预测 x_8（明天的价格)?

定义 1.2 (三种思路)

1. 只看今天：$x_8 = f(x_7)$
2. 看最近几天：$x_8 = f(x_7, x_6, x_5)$
3. 看所有历史：$x_8 = f(x_7, x_6, \ldots, x_1)$

♣

1.2 未命名节

1.2.1 统计工具

1.3 统计工具

1.3.1 8.1.1 统计工具

如何用数学语言描述” 用历史预测未来”?

1.3.1.0.1 自回归模型（Autoregressive Model） 核心思想：当前时刻的值依赖于过去的值

定义 1.3 (数学定义)

$$x_t = f(x_{t-1}, x_{t-2}, \ldots, x_{t-\tau}) + \epsilon_t$$

♣

符号说明：

- x_t：时间步 t 的观测值（我们要预测的）
- $x_{t-1}, x_{t-2}, \ldots, x_{t-\tau}$：过去 τ 个时间步的值
- $f(\cdot)$：某个函数（可以是线性、非线性、神经网络...）
- ϵ_t：随机噪声项
- τ：**回顾窗口**（look-back window）的大小

定理 1.2 (名字的由来)

Auto（自己）+ **Regressive**（回归）= 用自己的历史值来回归

♡

1.3.1.0.2 简单例子：线性自回归（AR 模型） 假设：未来是过去的加权平均

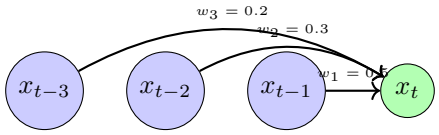
$$x_t = w_1 x_{t-1} + w_2 x_{t-2} + w_3 x_{t-3} + b + \epsilon_t$$

例题 1.1 具体数值例子 假设我们预测股价，学到的参数是：

$$x_t = 0.5x_{t-1} + 0.3x_{t-2} + 0.2x_{t-3} + 1.0$$

如果 $x_{t-1} = 100, x_{t-2} = 98, x_{t-3} = 95$，则：

$$x_t = 0.5 \times 100 + 0.3 \times 98 + 0.2 \times 95 + 1.0 = 99.4$$



1.3.1.0.3 自回归模型的问题

定理 1.3 (问题 1：窗口大小 τ 如何选择？)

- τ 太小：丢失长期信息，如季节性规律
- τ 太大：参数太多，计算复杂，过拟合

♡

定义 1.4 (理想情况)

我们希望：参数数量固定，但能记住所有历史！

♣

1.3.1.0.4 隐变量自回归模型 关键思想：引入一个**隐变量**（hidden state）h_t 来总结历史

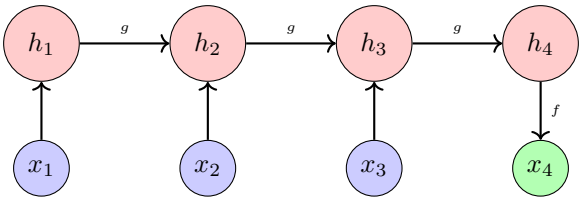
定义 1.5 (数学形式)

$$h_t = g(h_{t-1}, x_{t-1}) \quad \text{（更新记忆）}$$
$$x_t = f(h_t) + \epsilon_t \quad \text{（生成预测）}$$

♣

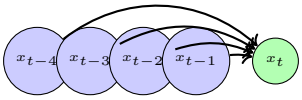
直观理解：

- h_t：像大脑的**记忆**，存储到时刻 t 的所有重要信息
- g：**更新函数**，根据新观测 x_{t-1} 更新记忆
- f：**输出函数**，根据当前记忆预测未来



隐状态像” 记忆”，不断更新

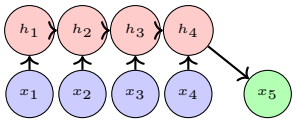
1.3.1.0.5 隐变量模型的优势 标准自回归：



问题：

- 输入维度 = τ
- 历史越长，输入越大
- 参数数量：$\tau \times d$

隐变量模型：



优势：

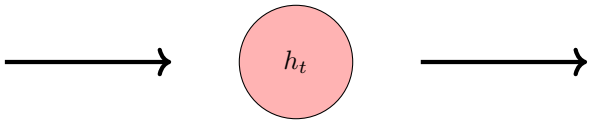
- 输入维度固定
- 无论历史多长
- 参数数量不变

定理 1.4 (关键创新)

h_t 就像一个压缩包，把所有历史信息压缩存储！
这就是循环神经网络（**RNN**）的核心思想！

♡

1.3.1.0.6 隐状态的直观类比



人类记忆	隐状态	循环神经网络
看到新信息	观测到 x_{t-1}	实现隐状态
↓	↓	的具体架构
更新记忆	更新 $h_t = g(h_{t-1}, x_{t-1})$	
↓	↓	
基于记忆决策	预测 $x_t = f(h_t)$	

定义 1.6 (类比)

隐状态 h_t = 你看完前面所有剧集后的记忆
你不记得每一个细节，但记得关键信息！

♣

1.3.2 训练

1.4 8.1.2 训练

1.4.0.0.1 训练序列模型

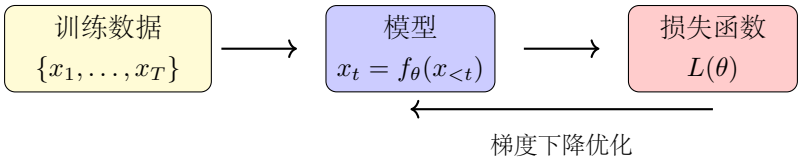
有了模型，如何从数据中学习参数?

1.4.0.0.2 训练的基本思路 **给定**：训练序列 $x_1, x_2, x_3, \ldots, x_T$

目标：找到最优参数，使模型预测尽可能准确

定义 1.7 (训练三步骤)

1. 定义模型：选择 f 和 g 的具体形式
2. 定义损失函数：衡量预测与真实值的差距
3. 优化参数：用梯度下降最小化损失



1.4.0.0.3 例子：训练线性自回归模型 模型：$x_t = w_1 x_{t-1} + w_2 x_{t-2} + b$

损失：均方误差 $L = \frac{1}{T}\sum_{t=1}^{T}(x_t - \hat{x}_t)^2$

1.4.0.0.4 训练循环

```
# training
num_epochs = 10
for epoch in range(num_epochs):
    predictions = model(features)
    loss = loss_fn(predictions, labels)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if (epoch + 1) % 2 == 0:
        print(f'Epoch {epoch+1}, Loss: {loss.item():.4f}')

# Output
# Epoch 2, Loss: 0.0523
# Epoch 4, Loss: 0.0418
# Epoch 6, Loss: 0.0387
# Epoch 8, Loss: 0.0372
# Epoch 10, Loss: 0.0365
```

定义 1.8 (训练过程)

损失逐渐下降 → 模型学到了数据的规律

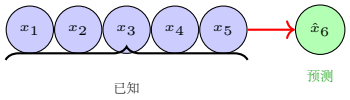
1.5 8.1.3 预测

1.5.1 8.1.3 预测

训练完模型后，如何用它来预测?

1.5.1.0.1 单步预测 vs 多步预测

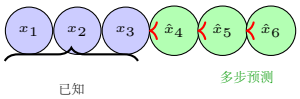
单步预测（One-step Ahead）：



特点：

- 只预测**下一步**
- 使用**真实**历史数据
- 准确度高

多步预测（Multi-step Ahead）：



特点：

- 预测**未来多步**
- 使用**预测值**作输入
- 误差会累积

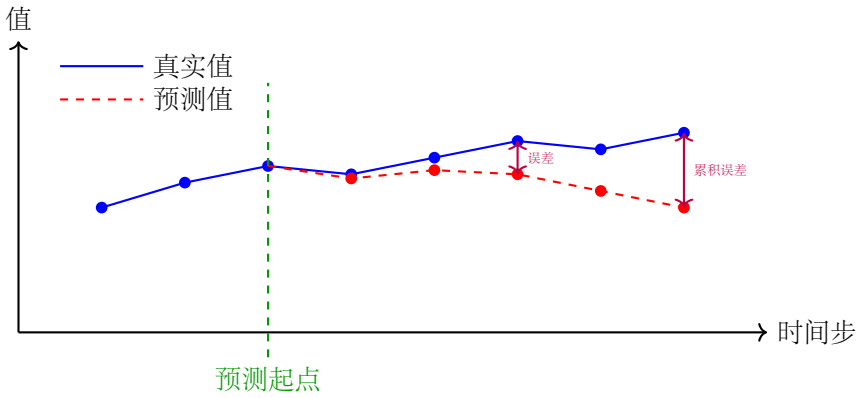
定理 1.5 (核心区别)

单步预测：$\hat{x}_{t+1} = f(x_t, x_{t-1}, \ldots)$ ← 用真实值

多步预测：$\hat{x}_{t+k} = f(\hat{x}_{t+k-1}, \hat{x}_{t+k-2}, \ldots)$ ← 用预测值

♡

1.5.1.0.2 多步预测的误差累积



定理 1.6 (误差累积现象)

- 第 1 步预测有小误差 ϵ_1
- 第 2 步基于有误差的预测，误差变大 $\epsilon_1 + \epsilon_2$
- 第 3 步误差继续累积...
- 预测越远，误差越大！

♡

1.5.1.0.3 单步预测代码示例

```
# Single-step prediction using real historical data
def predict_one_step(model, x_history, tau):
    """ x_history: shape (tau,) Return: next step prediction """
```

```
    with torch.no_grad():
        x_input = x_history[-tau:].reshape(1, -1)
        prediction = model(x_input)
    return prediction.item()

x_true = torch.tensor([1.0, 1.2, 1.1, 1.3, 1.2])
next_value = predict_one_step(model, x_true, tau=2)
print(f"Based on {x_true[-2:].tolist()} predict: {next_value:.3f}")

# Output
# Based on [1.3, 1.2] predict: 1.245
```

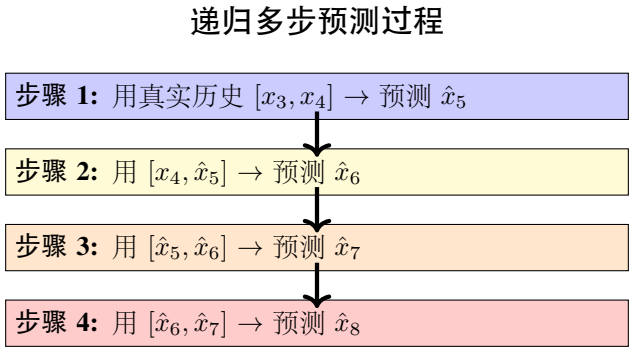
定义 1.9 (关键)

使用真实的历史数据 $x_{t-1}, x_{t-2}, \ldots$ 作为输入 ♣

1.5.1.0.4 多步预测：递归策略

```
# Multi-step prediction using recursive strategy
def predict_multi_step(model, x_history, tau, k_steps):
    """ x_history: initial history data, shape (tau,) k_steps: number of steps to predict Return: k
        prediction values """
    predictions = []
    current_input = x_history[-tau:].clone()

    for _ in range(k_steps):
        # Predict next step
        with torch.no_grad():
            next_pred = model(current_input.reshape(1, -1))
        predictions.append(next_pred.item())

        # Update input with prediction
        current_input = torch.cat([current_input[1:],
                                   next_pred.reshape(1)])

    return predictions

predictions = predict_multi_step(model, x_true, tau=2, k_steps=5)
print(f"Future 5-step predictions: {predictions}")
# Output: Future 5-step predictions: [1.245, 1.233, 1.239, 1.236, 1.237]
```

1.5.1.0.5 多步预测的可视化

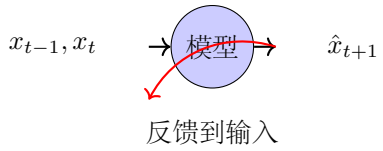


定理 1.7 (注意)

从步骤 2 开始，输入中就包含了预测值，不再是真实值!

1.5.1.0.6 多步预测的两种策略对比 策略 1：递归预测

(Autoregressive/Recursive)



优点:

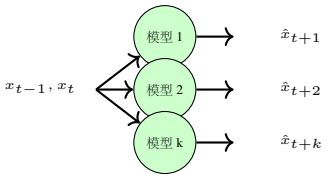
- 只需 1 个模型
- 灵活，可预测任意步数

缺点:

- 误差累积
- 长期预测不准

策略 2：直接预测

(Direct/MIMO)



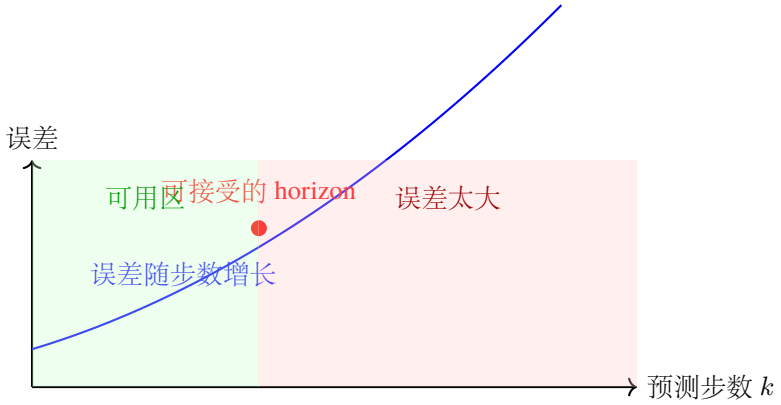
优点:

- 无误差累积
- 每步独立优化

缺点:

- 需要训练 k 个模型
- 计算开销大

1.5.1.0.7 预测 horizon 的选择 问题：应该预测多远的未来?



定义 1.10 (经验法则)

- 短期预测 (1-10 步)：通常可靠
- 中期预测 (10-100 步)：取决于数据规律性
- 长期预测 (>100 步)：非常困难，需要强正则化

1.5.1.0.8 实战：预测效果对比

```
# sequence
x_true = [1.0, 1.2, 1.1,
        1.3, 1.2, 1.4,
        1.3, 1.5, 1.4]

# Single-step predictionusing real
single_step = []
for i in range(3, 9):
    pred = model(x_true[i-2:i])
    single_step.append(pred)

# [1.25, 1.20, 1.35, ...]
```

```
# multi-steppredictionprediction
multi_step = []
input = x_true[1:3] # [1.2, 1.1]
for _ in range(6):
    pred = model(input)
    multi_step.append(pred)
    input = [input[1], pred]

# [1.25, 1.18, 1.27, ...]
```

时间步	真实值	单步预测	多步预测
4	1.3	1.25	1.25
5	1.2	1.20	1.18
6	1.4	1.35	1.27
7	1.3	1.32	1.22

定理 1.8 (观察)

多步预测的误差明显大于单步预测！

1.5.1.0.9 预测的实际应用

定义 1.11 (不同场景的预测需求)

- 天气预报：需要未来 3-7 天预测（多步）
- 股票交易：高频交易只关心下一刻（单步）
- 语言模型：生成文本需要长序列预测（多步）
- 异常检测：只需要预测下一个值并比较（单步）

♣

定理 1.9 (关键权衡)

- 单步预测：准确但只能看”一步之遥”
- 多步预测：能看”更远”但不够准确
- 实际应用需要根据任务选择合适的策略

♡

1.6 8.2 文本预处理

1.6.0.0.1 进入 8.2 文本预处理

从数值序列到文本序列

定义 1.12 (之前学的（8.1 节）)

- 处理数值序列（股价、温度等）
- 输入和输出都是连续值

♣

定义 1.13 (现在要学的（8.2 节）)

- 处理文本序列（句子、文章等）
- 输入和输出是离散符号（词、字符）
- 需要特殊的预处理步骤

♣

定理 1.10 (核心问题)

如何把文本转换成数字，让神经网络能处理?

♡

1.6.1 8.2 文本预处理 - 章节导航

定义 1.14 (本节内容)

- 8.2.1 读取数据集
- 8.2.2 词元化 (Tokenization)
- 8.2.3 词表 (Vocabulary)
- 8.2.4 整合所有功能

♣

定理 1.11 (学习目标)

1. 学会读取和清洗文本
2. 理解不同的词元化策略
3. 掌握构建词表的方法
4. 能将文本转为数字序列

♡

1.7 未命名节

1.7.1 读取数据集

1.7.2 8.2.1 读取数据集

1.7.3 8.2.1 读取数据集

第一步：获取原始文本

1.7.3.0.1 为什么需要文本预处理?

原始文本:
"The Time Machine, by H.G. Wells [1898]"
"It was at ten o'clock to-day that..."

↓ 需要转换

数字序列:
234, 567, 123, 89, 456, ...
计算机才能处理!

定义 1.15 (文本预处理的挑战)

- 文本是离散的符号
- 包含大小写、标点、特殊字符
- 需要统一的处理流程

♣

1.7.3.0.2 时光机器数据集 数据集：H.G. Wells 的科幻小说《时光机器》(The Time Machine, 1895)

例题 1.2 为什么选这个数据集?

- 英文文本，便于处理
- 大小适中（约 30KB，3 万多词）
- 语言规范，句子结构清晰
- 开源免费

原始文本片段:
"The Time Machine, by H. G. Wells [1898]"
"I"
"The Time Traveller (for so it will be convenient to speak of him)"
"was expounding a recondite matter to us..."

定理 1.12 (特点)

包含标题、章节标记、对话、描述等多种文本形式

♡

1.7.3.0.3 读取文本文件

```
1 import os
2 import requests
3
4 # data
```

```
def download_data():
    url = 'http://d2l-data.s3-accelerate.amazonaws.com/timemachine.txt'
    response = requests.get(url)
    with open('timemachine.txt', 'wb') as f:
        f.write(response.content)

def read_time_machine():
    with open('timemachine.txt', 'r', encoding='utf-8') as f:
        lines = f.readlines()
    return lines

lines = read_time_machine()
print(f"Total lines: {len(lines)}")
print(f"First 3 lines:")
for i in range(3):
    print(f"Line {i}: {lines[i]}")
```

输出:

```
Total lines: 3221
Line 0: The Time Machine, by H. G. Wells [1898]
Line 1:
Line 2: I
```

1.7.3.0.4 原始文本的”脏数据” 问题：原始文本包含很多”噪声”

常见噪声:

- 大小写混杂
  `"The", "the", "THE"`
- 标点符号
  `"Hello!", "Hello?", "Hello."`
- 特殊字符
  `"don't", "it's", "---"`
- 空行和空格
- 数字
  `"year 1898"`

定理 1.13 (为什么要清洗？)

- 减少词汇量
- 统一表示形式
- 避免稀疏性问题
- 提高模型泛化能力

♡

例题 1.3 例子 不清洗:

”The”, ”the”, ”THE” → 3 个词

清洗后:

”the” → 1 个词

1.7.3.0.5 文本清洗

```
import re

def read_time_machine():
    with open('timemachine.txt', 'r') as f:
        lines = f.readlines()

    clean_lines = []
    for line in lines:
        # 1.
        line = line.lower()
        # 2.
        line = re.sub('[^a-z]+', ' ', line)
        # 3.
        line = line.strip()
        # 4.
        if line:
            clean_lines.append(line)

    return clean_lines

lines_raw = read_raw()
lines_clean = read_time_machine()
print("Raw:", lines_raw[0])
print("Clean:", lines_clean[0])
```

输出：

```
Raw: The Time Machine, by H. G. Wells [1898]
Clean: the time machine by h g wells
```

1.7.3.0.6 正则表达式快速入门 正则表达式（Regex）：强大的文本匹配工具

模式	含义	示例
[a-z]	匹配小写字母	"a", "b", "z"
[A-Z]	匹配大写字母	"A", "B", "Z"
[0-9]	匹配数字	"0", "5", "9"
[a-zA-Z]	匹配所有字母	"a", "B", "z"
^	取反	[^a-z] = 非小写字母
+	一个或多个	[a-z]+ = 多个字母

例题 1.4 应用示例

- 把所有**非小写字母**替换为空格
- 把所有**数字**替换为特殊标记

1.7.3.0.7 清洗前后对比 清洗前：

```
The Time Machine,
by H. G. Wells [1898]

I
```

```

The Time Traveller
(for so it will be
convenient to speak of him)
was expounding a
recondite matter to us.
```

清洗后：

```
the time machine by h g wells

i

the time traveller for so
it will be convenient to
speak of him was expounding
a recondite matter to us
```

定义 1.16 (观察)

- 全部小写
- 标点消失
- 数字消失
- 保留了空格（词与词之间的分隔）

♣

1.7.4 8.2.2 词元化

1.7.5 8.2.2 词元化（Tokenization）

把文本拆分成” 词元”（token）

1.7.5.0.1 什么是词元？ 词元（Token）：文本的基本处理单位

原始句子：”the time traveller was smiling”

↓

the　time　traveller　was　smiling

5 个词元（tokens）

定理 1.14 (为什么需要词元化？)

- 神经网络只能处理固定大小的输入
- 需要将连续的文本拆分成离散单元
- 词元是特征提取的基础

♡

1.7.5.0.2 词元化的三种粒度

句子：”natural language processing”

1. 词级别（Word-level）
`["natural", "language", "processing"]`
→ 3 个词元

2. 字符级别（Character-level）
`["n","a","t","u","r","a","l"," ","l","a","n","g",...]`
→ 27 个词元

3. 子词级别（Subword-level）
`["natur", "al", "language", "process", "ing"]`
→ 5 个词元

1.7.5.0.3 词级别词元化 策略：按空格或标点分割

优点：

- ✓语义清晰
- ✓符合人类理解
- ✓适合大多数任务

缺点：

- × 词表很大
- × 未登录词（OOV）问题
- × 形态变化（如：run/running）

例题 1.5OOV 问题 训练集：

`"dog", "cat", "run"`

测试时遇到：

`"dogs", "running", "bird"`

→ 模型不认识!

定义 1.17 (词表大小)

英文：常用词汇 5 万-10 万

中文：常用汉字 3 千-5 千，但词汇量更大 ♣

1.7.5.0.4 词级别词元化代码

```python
def tokenize_word_level(text):
    """Simple function"""
    return text.split()

text = "the time traveller was smiling"
tokens = tokenize_word_level(text)
print(f"tokens: {tokens}")
print(f"tokens数: {len(tokens)}")

# Output
# tokens: ['the', 'time', 'traveller', 'was', 'smiling']
# tokens: 5

# data
lines = read_time_machine()
```

```
all_tokens = []
for line in lines:
    all_tokens.extend(tokenize_word_level(line))

print(f"总tokens数: {len(all_tokens)}")
print(f"前20个tokens: {all_tokens[:20]}")
```

输出:

```
总tokens数: 32775
前20个tokens: ['the', 'time', 'machine', 'by', 'h', 'g', ...]
```

1.7.5.0.5 字符级别词元化 策略: 把每个字符作为一个词元

优点:

- ✓词表极小（26 个字母）
- ✓无 OOV 问题
- ✓能处理任意文本
- ✓对拼写错误鲁棒

缺点:

- × 序列变得很长
- × 训练慢
- × 难以捕捉词级别语义

例题 1.6 例子 **输入:**

"cat"

词级别:

1 个词元: ["cat"]

字符级别:

3 个词元: ["c","a","t"]

定理 1.15 (序列长度)

”hello world” (11 个字符)

vs

”hello world” (2 个单词)

序列长度差 5 倍!

♡

1.7.5.0.6 字符级别词元化代码

```
def tokenize_char_level(text):
    """tokens"""
    return list(text)

text = "the time"
tokens = tokenize_char_level(text)
print(f"tokens: {tokens}")
print(f"tokens数: {len(tokens)}")

# Output
# tokens: ['t', 'h', 'e', ' ', 't', 'i', 'm', 'e']
```

```
# tokens: 8

# Calculate vocabulary size
lines = read_time_machine()
text = ' '.join(lines)
chars = set(text)
print(f"字符集大小: {len(chars)}")
print(f"字符集: {sorted(chars)}")
```

输出:

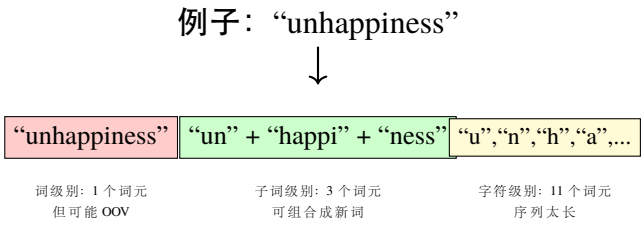
```
字符集大小: 27
字符集: [' ', 'a', 'b', 'c', ..., 'z']
```

定义 1.18 (观察)

词表从 5 万缩小到 27 个! 但序列长度增加 5-10 倍

♣

1.7.5.0.7 子词级别词元化 核心思想: 介于词和字符之间的折中方案



定义 1.19 (子词的好处)

- 可以组合出训练集中没见过的词
- 例如: ”unhappy” = ”un” + ”happy”
- 即使没见过”unhappy”, 也能理解它的意思

♣

1.7.5.0.8 常见子词算法

1. **BPE (Byte Pair Encoding)**
   - 从字符开始, 迭代合并最频繁的相邻 pair
   - GPT-2, RoBERTa 使用
2. **WordPiece**
   - 类似 BPE, 但用 likelihood 而非频率
   - BERT 使用
3. **SentencePiece**
   - 直接从原始文本训练, 不依赖预分词
   - 支持多语言
   - T5, XLNet 使用

定理 1.16 (现代 NLP 趋势)

大模型几乎都使用子词级别词元化!

平衡了词表大小和序列长度

♡

1.7.5.0.9 三种粒度对比总结

特性	词级别	字符级别	子词级别
词表大小	很大 (5 万 +)	很小 (~100)	中等 (3 万)
序列长度	短	很长	中等
OOV 问题	严重	无	轻微
语义保留	好	差	较好
训练速度	快	慢	中等
应用场景	传统 NLP	生成任务	现代大模型

定义 1.20 (本节使用)

为了简单起见，我们使用词级别词元化
在 8.4 节实现 RNN 时，会用字符级别

♣

1.7.5.0.10 完整词元化函数

```
def tokenize(lines, token_type='word'):
    """ tokens lines: token_type: 'word' 'char' Return: tokens """
    if token_type == 'word':
        return [line.split() for line in lines]
    elif token_type == 'char':
        return [list(line) for line in lines]
    else:
        raise ValueError(f"Unknown token_type: {token_type}")

lines = read_time_machine()
tokens_word = tokenize(lines, 'word')
tokens_char = tokenize(lines, 'char')

print(f"词级别 - 第1行: {tokens_word[0]}")
print(f"字符级别 - 第1行: {tokens_char[0]}")
```

输出:

```
词级别 - 第1行: ['the', 'time', 'machine', 'by', 'h', 'g', 'wells']
字符级别 - 第1行: ['t','h','e',' ','t','i','m','e',...]
```

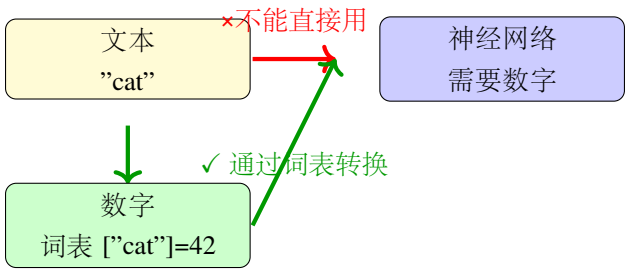
1.7.6 词表

1.7.7 8.2.3 词表

1.7.8 8.2.3 词表（Vocabulary）

建立词元到数字的映射

1.7.8.0.1 为什么需要词表？ **问题**：神经网络不能直接处理文本



定理 1.17 (词表的作用)

建立词元 ↔ 数字的双向映射 ♡

1.7.8.0.2 词表构建示例 输入文本：”the cat sat on the mat”

步骤 1：词元化

```
["the", "cat", "sat", "on", "the", "mat"]
```

步骤 2：统计词频

步骤 3：构建词表（按频率排序）

```
0: <unk>（未知词）
1: the (freq=2)
2: cat (freq=1)
3: mat (freq=1)
4: on (freq=1)
5: sat (freq=1)
```

1.7.8.0.3 特殊词元 除了普通词，词表还包含特殊标记：

符号	含义	用途
<unk>	Unknown	未知词，词表中没有的词
<pad>	Padding	填充，对齐不同长度序列
<bos>	Begin of Sequence	句子开始标记
<eos>	End of Sequence	句子结束标记
<mask>	Mask	掩码，用于 BERT 等模型

例题 1.7 例子 原始：”hello world”

加特殊标记：

```
[<bos>, "hello", "world", <eos>]
```

转数字（假设词表）：

```
[1, 234, 567, 2]
```

1.7.8.0.4 词频统计

```
def count_corpus(tokens):
    """tokens"""
    # tokens1D2D
    if len(tokens) == 0 or isinstance(tokens[0], list):
        # 2D1D
        tokens = [token for line in tokens for token in line]

```

```
    from collections import Counter
    return Counter(tokens)

lines = read_time_machine()
tokens = tokenize(lines, 'word')
counter = count_corpus(tokens)

print(f"总tokens数: {sum(counter.values())}")
print(f"不同tokens数: {len(counter)}")
print(f"最常见的10个词: {counter.most_common(10)}")
```

输出:

总词元数: 32775
不同词元数: 4580
最常见的10个词: [('the', 2261), ('i', 1267), ('and', 1245), ...]

1.7.8.0.5 词表类实现 - 初始化

```
class Vocab:
    def __init__(self, tokens=None, min_freq=0,
                 reserved_tokens=None):
        """ tokens: tokens min_freq: reserved_tokens: tokens (<pad>)  """
        if tokens is None:
            tokens = []
        if reserved_tokens is None:
            reserved_tokens = []

        counter = count_corpus(tokens)
        self._token_freqs = sorted(counter.items(),
                                   key=lambda x: x[1],
                                   reverse=True)

        # tokens0tokens1
        self.idx_to_token = ['<unk>'] + reserved_tokens
        self.token_to_idx = {token: idx
                for idx, token in enumerate(self.idx_to_token)}
```

1.7.8.0.6 词表类实现 - 构建映射

```
        for token, freq in self._token_freqs:
            if freq < min_freq:
                break
            if token not in self.token_to_idx:
                self.idx_to_token.append(token)
                self.token_to_idx[token] = len(self.idx_to_token) - 1

    def __len__(self):
        """vocabulary"""
        return len(self.idx_to_token)

```

```
    def __getitem__(self, tokens):
        """查询tokens对应的索引"""
        if not isinstance(tokens, (list, tuple)):
            return self.token_to_idx.get(tokens, self.unk)
        return [self.__getitem__(token) for token in tokens]

    @property
    def unk(self):
        """未知词的索引，总是0"""
        return 0

    def to_tokens(self, indices):
        """索引转回tokens"""
        if not isinstance(indices, (list, tuple)):
            return self.idx_to_token[indices]
        return [self.idx_to_token[index] for index in indices]
```

1.7.8.0.7 词表使用示例

```
# vocabulary
lines = read_time_machine()
tokens = tokenize(lines, 'word')
vocab = Vocab(tokens)

print(f"vocabulary大小: {len(vocab)}")
print(f"前10个tokens: {vocab.idx_to_token[:10]}")

# tokens
test_tokens = ['the', 'time', 'machine', 'xyz']
indices = vocab[test_tokens]
print(f"tokens: {test_tokens}")
print(f"索引: {indices}")

# tokens
back_tokens = vocab.to_tokens(indices)
print(f"转回: {back_tokens}")
```

输出:

```
词表大小: 4581
前10个词元: ['<unk>', 'the', 'i', 'and', 'of', 'a', ...]
词元: ['the', 'time', 'machine', 'xyz']
索引: [1, 19, 50, 0]
转回: ['the', 'time', 'machine', '<unk>']
```

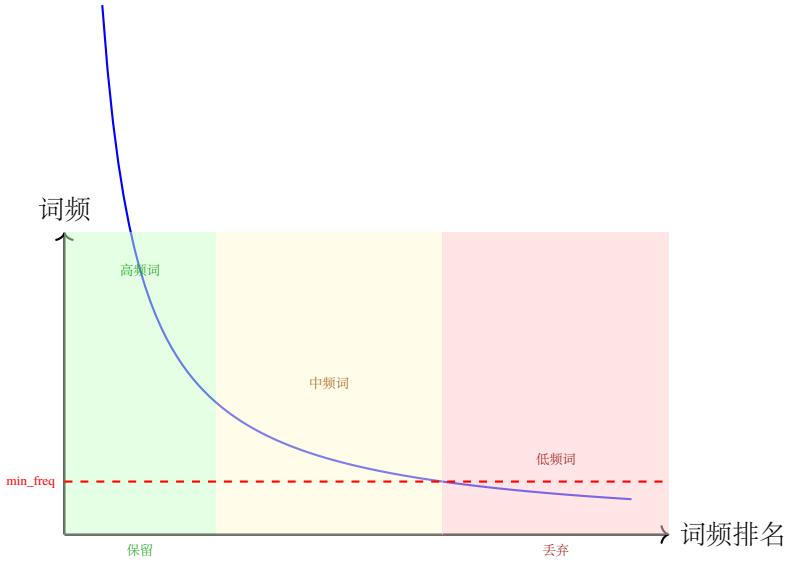
定理 1.18 (注意)

'xyz' 不在词表中，被映射到 <unk> (索引 0)

♡

1.7.8.0.8 最小频率阈值的作用

问题: 低频词很多，但作用不大



定义 1.21 (策略)

- min_freq=0：保留所有词（词表大）
- min_freq=5：至少出现 5 次才保留（减小词表）
- 低频词通常是拼写错误、人名等，可以用 <unk> 代替

♣

1.7.8.0.9 最小频率阈值示例

```
vocab_all = Vocab(tokens, min_freq=0)
vocab_5 = Vocab(tokens, min_freq=5)
vocab_10 = Vocab(tokens, min_freq=10)

print(f"min_freq=0: vocabulary大小={len(vocab_all)}")
print(f"min_freq=5: vocabulary大小={len(vocab_5)}")
print(f"min_freq=10: vocabulary大小={len(vocab_10)}")

counter = count_corpus(tokens)
low_freq = [token for token, freq in counter.items() if freq < 5]
print(f"频率<5的词数: {len(low_freq)}")
print(f"示例: {low_freq[:10]}")
```

输出：

```
min_freq=0: 词表大小=4581
min_freq=5: 词表大小=2034
min_freq=10: 词表大小=1446
频率<5的词数: 3143
示例: ['1898', 'xiv', 'pg', 'redistribution', ...]
```

定理 1.19 (观察)

68% 的词频率 <5，但它们只占总词数的很小比例！

♡

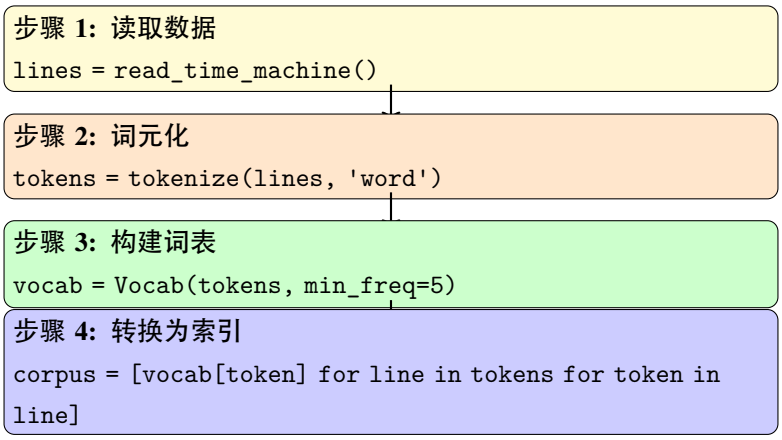
1.7.9 整合所有功能

1.7.10 8.2.4 整合所有功能

1.7.11 8.2.4 整合所有功能

完整的文本预处理流程

1.7.11.0.1 完整流程总结



1.7.11.0.2 一站式函数

```
def load_corpus_time_machine(max_tokens=-1):
    """ Returndatatokensvocabulary max_tokens: tokens, -1 """
    lines = read_time_machine()
    # tokens
    tokens = tokenize(lines, 'char') # comment
    # vocabulary
    vocab = Vocab(tokens)
    corpus = [vocab[token] for line in tokens for token in line]
    if max_tokens > 0:
        corpus = corpus[:max_tokens]

    return corpus, vocab

corpus, vocab = load_corpus_time_machine()
print(f"语料库length: {len(corpus)}")
print(f"vocabulary大小: {len(vocab)}")
print(f"前10个索引: {corpus[:10]}")
print(f"对应tokens: {vocab.to_tokens(corpus[:10])}")
```

1.7.11.0.3 一站式函数输出

```
# Output
语料库length: 170580
vocabulary大小: 28
前10个索引: [1, 8, 5, 0, 1, 9, 13, 5, 0, 13]
对应tokens: ['t', 'h', 'e', ' ', 't', 'i', 'm', 'e', ' ', 'm']
```

定义 1.22 (观察)

- 使用字符级别词元化
- 语料库是一个很长的数字序列（17 万 +）
- 词表只有 28 个字符（26 个字母 + 空格 + 其他）
- 每个索引对应词表中的一个字符

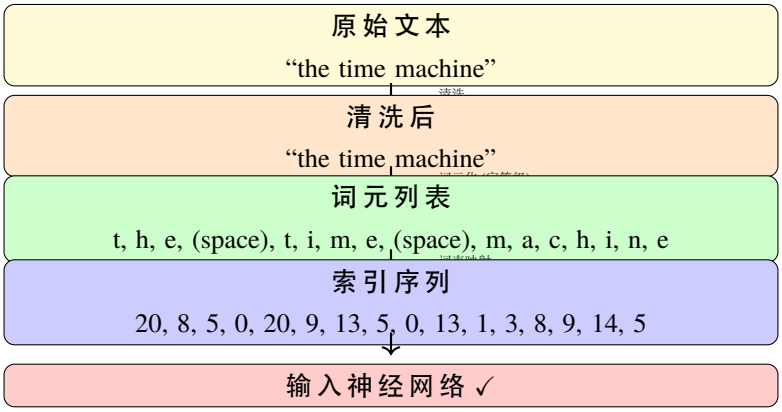
♣

定理 1.20 (为什么用字符级别？)

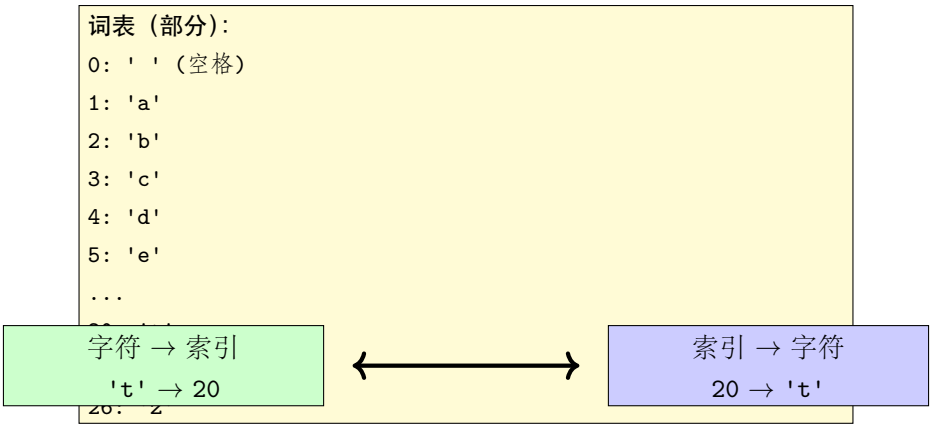
- RNN 训练时序列不宜过长
- 字符级别词表小，便于演示
- 避免 OOV 问题

♡

1.7.11.0.4 文本 → 数字的完整示例



1.7.11.0.5 词表可视化



1.7.11.0.6 完整数据加载器（带批处理）

```
def load_data_time_machine(batch_size, num_steps,
                           use_random_iter=False):
    """ Returndataiterationvocabulary batch_size: num_steps: time step use_random_iter: """
    corpus, vocab = load_corpus_time_machine()

    # dataiteration8.3
    if use_random_iter:
        data_iter = seq_data_iter_random(corpus, batch_size, num_steps)
```

```
    else:
        data_iter = seq_data_iter_sequential(corpus, batch_size, num_steps)

    return data_iter, vocab

batch_size, num_steps = 32, 35
train_iter, vocab = load_data_time_machine(batch_size, num_steps)
```

1.7.12 8.2 节总结

定义 1.23 (核心流程)

1. 读取数据：从文件读取原始文本
2. 清洗：转小写、去标点、去数字
3. 词元化：拆分成词元（词/字符/子词）
4. 构建词表：统计词频，建立索引映射
5. 数值化：将文本转为数字序列

定理 1.21 (关键概念)

- 词元（**Token**）：文本的基本单位
- 词表（**Vocabulary**）：词元 ↔ 索引的映射
- 语料库（**Corpus**）：数值化后的文本序列
- 特殊标记：<unk>, <pad>, <bos>, <eos>

1.7.12.0.1 三种词元化策略对比

词级别： 5 个词元：["I", "love", "natural", "language",
子词级别： 7 个词元：["I", "love", "natur", "al", "language",
字符级别： 33 个词元：["I", " ", "l", "o", "v", "e", " ", "n", ...] 词表小：只需 100 左右 | 序列太长

定义 1.24 (选择建议)

- 教学/演示 → 字符级别（简单）
- 传统任务 → 词级别（快速）
- 现代大模型 → 子词级别（平衡）

1.7.12.0.2 实战 tips 常见问题：

1. **词表太大**
   → 提高 min_freq
   → 使用子词
2. **OOV 太多**
   → 降低 min_freq
   → 使用字符/子词

3. **序列太长**
   → 使用词级别
   → 截断序列

优化建议：

1. 根据任务选择粒度
2. 合理设置频率阈值
3. 考虑特殊标记需求
4. 保存词表以便复用
5. 可视化词频分布

定理 1.22 (下一步)

有了数值化的文本，接下来学习如何构建语言模型（8.3 节）

♡

1.7.13 8.2 节关键代码总结

核心函数：

`read_time_machine()` - 读取并清洗文本
`tokenize(lines, type)` - 词元化
`count_corpus(tokens)` - 统计词频

Vocab 类关键方法：

`vocab[token]` - 词元 → 索引
`vocab.to_tokens(idx)` - 索引 → 词元
`len(vocab)` - 词表大小

数 据 流：

文本 → 清洗 → 词元 → 词表 → 索引 → 神经网络

1.8 8.3 语言模型和数据集

1.8.1 8.3 语言模型和数据集 - 章节导航

定义 1.25 (本节内容)

- 8.3.1 学习语言模型
- 8.3.2 马尔可夫模型与元语法
- 8.3.3 自然语言统计
- 8.3.4 读取长序列数据

♣

定理 1.23 (学习目标)

1. 理解语言模型的数学定义
2. 掌握困惑度（Perplexity）的计算
3. 了解马尔可夫假设
4. 学会构建序列数据迭代器

♡

1.9 未命名节

1.9.1 学习语言模型

1.9.2 8.3.1 学习语言模型

1.9.3 8.3.1 学习语言模型

语言模型的核心：估计文本序列的概率

1.9.3.0.1 什么是语言模型？ 定义：语言模型对文本序列的概率分布进行建模

定义 1.26 (数学表示)

对于文本序列 $x_1, x_2, \ldots, x_T$，语言模型估计其概率：

$$P(x_1, x_2, \ldots, x_T)$$

♣

链式法则分解：

$$P(x_1, x_2, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_1, \ldots, x_{t-1})$$

例题 1.8 直观理解 给定前面的词，预测下一个词的概率

- 已知："the cat sat on the"
- 预测：下一个词是 "mat" 的概率?

1.9.3.0.2 语言模型的应用 1. 文本生成

- 对话系统
- 故事创作
- 代码补全

2. 机器翻译

- 评估译文流畅度
- 选择最佳翻译

3. 语音识别

- 纠正识别错误
- 消除歧义

4. 拼写检查

- 检测错误
- 提供建议

定理 1.24 (核心思想)

好的语言模型能给合理的句子分配高概率，给不合理的句子分配低概率

♡

1.9.3.0.3 语言模型的挑战 问题：直接估计 $P(x_1, x_2, \ldots, x_T)$ 非常困难!

挑战 1：序列长度不固定
不同句子长度不同，如何统一建模?

挑战 2：参数爆炸
如果词表大小为 V，长度为 T 的序列有 V^T 种可能!

挑战 3：数据稀疏
很多序列在训练集中从未出现，如何估计概率?

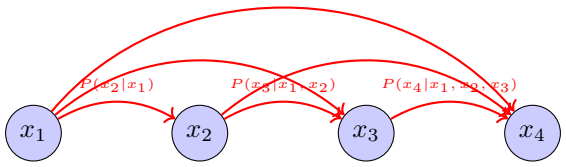
定义 1.27 (解决思路)

- 使用链式法则分解
- 使用马尔可夫假设简化
- 使用神经网络建模条件概率

♣

1.9.3.0.4 条件概率建模 关键：将复杂的联合概率分解为条件概率的乘积

$$P(x_1, x_2, x_3, x_4) = P(x_1) \cdot P(x_2 \mid x_1) \cdot P(x_3 \mid x_1, x_2) \cdot P(x_4 \mid x_1, x_2, x_3)$$



每个词依赖前面所有词

定理 1.25 (关键问题)

如何高效地估计条件概率 $P(x_t \mid x_1, \ldots, x_{t-1})$?

♡

1.9.3.0.5 困惑度（Perplexity） 定义：评估语言模型质量的标准指标

定义 1.28 (数学定义)

$$\text{PPL} = \exp\left(-\frac{1}{n}\sum_{t=1}^{n}\log P(x_t \mid x_1, \ldots, x_{t-1})\right)$$

♣

等价形式：

$$\text{PPL} = \sqrt[n]{\frac{1}{P(x_1, x_2, \ldots, x_n)}}$$

直观理解：

- 平均分支因子
- 模型的“困惑程度”
- 越低越好

取值范围：

- 最好：PPL = 1（完全确定）
- 随机猜测：PPL = $|V|$
- 实际：10-200 之间

1.9.3.0.6 困惑度的直观例子 例子：预测下一个词

模型 A（好）：
“I love” 后面是：
- “you”：0.7
- “it”：0.2
- “apple”：0.1

✓ 集中在合理的词上
困惑度低

模型 B（差）：
“I love” 后面是：
- “you”：0.01
- “it”：0.01
- 其他 998 个词均分

PPL ≈ 100

× 概率分散
困惑度高

1.9.3.0.7 困惑度计算示例

```
import torch
import torch.nn.functional as F

def calculate_perplexity(model, data, vocab):
    """model"""
    model.eval()
    total_loss = 0
    total_tokens = 0

    with torch.no_grad():
        for X, Y in data:
            # X: (batch_size, seq_len)
            # Y: (batch_size, seq_len)
            output = model(X) # (batch_size, seq_len, vocab_size)

            # loss
            loss = F.cross_entropy(
                output.reshape(-1, len(vocab)),
                Y.reshape(-1),
                reduction='sum'
            )

            total_loss += loss.item()
            total_tokens += Y.numel()

    #  = exp()
    perplexity = torch.exp(torch.tensor(total_loss / total_tokens))
    return perplexity.item()
```

1.9.4 马尔可夫模型与元语法

1.9.5 8.3.2 马尔可夫模型与元语法

1.9.6 8.3.2 马尔可夫模型与元语法

简化建模：马尔可夫假设

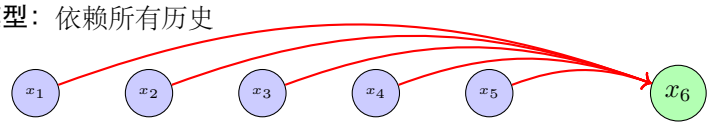
1.9.6.0.1 马尔可夫假设 **问题**：依赖所有历史太复杂

$$P(x_t \mid x_1, x_2, \ldots, x_{t-1})$$

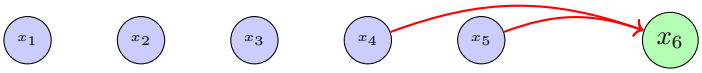
马尔可夫假设：只依赖最近的 τ 个词

$$P(x_t \mid x_1, x_2, \ldots, x_{t-1}) \approx P(x_t \mid x_{t-\tau}, \ldots, x_{t-1})$$

完整模型：依赖所有历史



马尔可夫模型：只依赖最近 $\tau = 2$ 个



1.9.6.0.2 N-gram 模型 **根据 τ 的不同，有不同的 N-gram 模型**：

模型	τ	条件概率
Unigram (1-gram)	0	$P(x_t)$
Bigram (2-gram)	1	$P(x_t \mid x_{t-1})$
Trigram (3-gram)	2	$P(x_t \mid x_{t-2}, x_{t-1})$
4-gram	3	$P(x_t \mid x_{t-3}, x_{t-2}, x_{t-1})$

例题 1.9 例子：Bigram 模型 **句子**："the cat sat on the mat"

概率：

$$P(\text{sentence}) = P(\text{the}) \cdot P(\text{cat} \mid \text{the}) \cdot P(\text{sat} \mid \text{cat}) \cdot P(\text{on} \mid \text{sat}) \cdot P(\text{the} \mid \text{on}) \cdot P(\text{mat} \mid \text{the})$$

1.9.6.0.3 N-gram 模型的估计 **最大似然估计（MLE）**：

$$P(x_t \mid x_{t-\tau}, \ldots, x_{t-1}) = \frac{\text{count}(x_{t-\tau}, \ldots, x_{t-1}, x_t)}{\text{count}(x_{t-\tau}, \ldots, x_{t-1})}$$

例题 1.10Bigram 例子 训练文本："I love you I love coding I love AI"

计算：$P(\text{you} \mid \text{love})$

$$P(\text{you} \mid \text{love}) = \frac{\text{count}(\text{love you})}{\text{count}(\text{love})} = \frac{1}{3}$$

其他：

- $P(\text{coding} \mid \text{love}) = 1/3$
- $P(\text{AI} \mid \text{love}) = 1/3$

1.9.6.0.4 N-gram 模型的问题 优点：

- ✓ 简单高效
- ✓ 易于实现
- ✓ 可解释性强

缺点：

- × 无法捕捉长期依赖
- × 数据稀疏问题严重
- × 参数随 τ 指数增长

定理 1.26 (数据稀疏问题)

例子：训练集中没见过 “love machine”

$\Rightarrow P(\text{machine} \mid \text{love}) = 0$

解决：平滑技术（Laplace 平滑、Good-Turing 平滑等）

♡

定义 1.29 (现代方案)

使用神经网络语言模型（如 RNN）克服这些问题

♣

1.9.6.0.5 N-gram 模型实现

```python
from collections import defaultdict, Counter

class NgramModel:
    def __init__(self, n=2):
        self.n = n
        self.context_counts = defaultdict(Counter)

    def train(self, corpus):
        """trainingN-grammodel"""
        for i in range(self.n - 1, len(corpus)):
            context = tuple(corpus[i - self.n + 1:i])
            word = corpus[i]
            self.context_counts[context][word] += 1

    def prob(self, context, word):
        """计算条件概率 P(word | context)"""
        context = tuple(context)
        if context not in self.context_counts:
            return 0

        total = sum(self.context_counts[context].values())
        return self.context_counts[context][word] / total
```

1.9.7 自然语言统计

1.9.8 8.3.3 自然语言统计

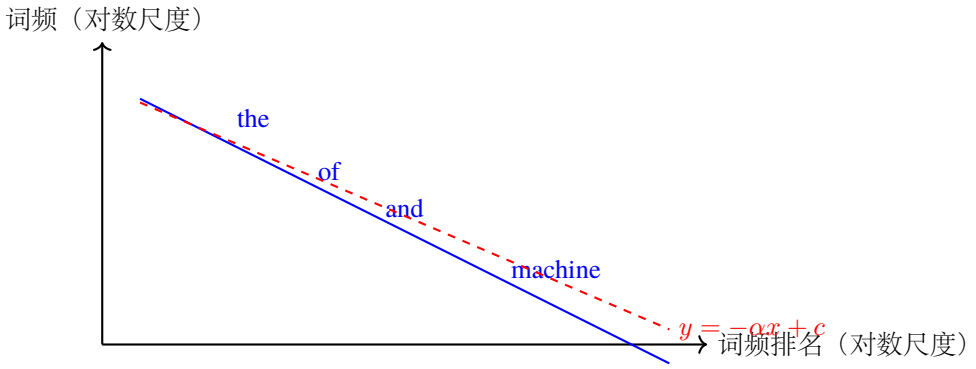
1.9.9 8.3.3 自然语言统计

真实文本的统计规律

1.9.9.0.1 Zipf 定律 定律：词频与其排名成反比

$$\text{frequency}(k) \propto \frac{1}{k^{\alpha}}$$

其中 k 是词频排名，$\alpha \approx 1$



少数高频词 + 大量低频词

1.9.9.0.2 Zipf 定律的含义 长尾分布：

- 少数词非常常见
- 大部分词很少见
- “the”, “of”, “and” 占比很大

影响：

- 词表大小选择
- 低频词处理
- 采样策略

例题 1.11 时光机器数据

- 总词数：32,775
- 不同词数：4,580
- Top 10 占比：30%
- Top 100 占比：50%
- 出现 1 次的词：50%

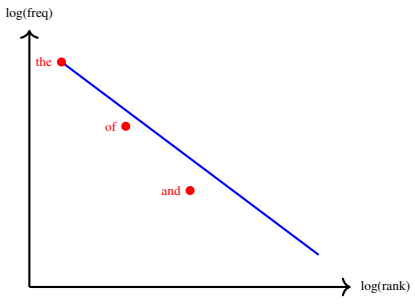
定理 1.27 (启示)

可以用 `min_freq` 过滤低频词，减小词表但保留大部分信息 ♡

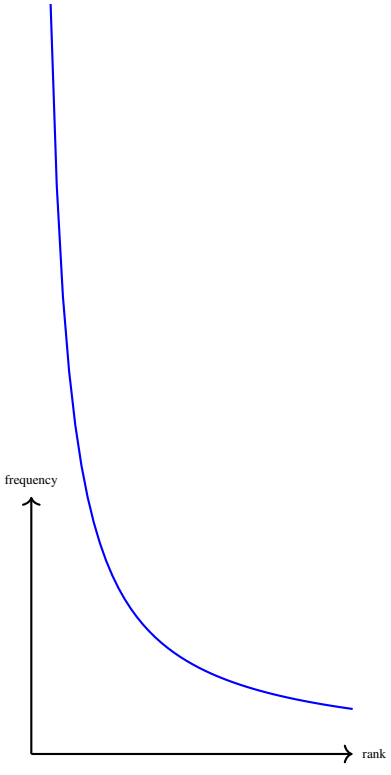
1.9.9.0.3 可视化 Zipf 定律

```
import matplotlib.pyplot as plt
import numpy as np

counter = count_corpus(tokens)
freqs = sorted(counter.values(), reverse=True)

plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)
plt.loglog(range(1, len(freqs) + 1), freqs)
plt.xlabel('Rank')
plt.ylabel('Frequency')
plt.title('Zipf Law (log-log)')

plt.subplot(1, 2, 2)
plt.plot(range(1, 100), freqs[:99])
plt.xlabel('Rank')
plt.ylabel('Frequency')
plt.title('Top 100 words')

plt.tight_layout()
plt.show()
```

1.9.9.0.4 词频统计可视化



对数-对数图（直线）



普通坐标（双曲线）

定义 1.30 (观察)

- 双对数图呈直线 ⇒ 幂律分布
- 少数高频词 + 长长的尾巴
- 符合 Zipf 定律

♣

1.9.10 读取长序列数据

1.9.11 8.3.4 读取长序列数据

1.9.12 8.3.4 读取长序列数据

如何高效地训练序列模型?

1.9.12.0.1 训练 RNN 的挑战 **问题**：完整的文本语料库是一个很长的序列

时光机器语料库：170,580 个字符

太长了！无法一次性处理

↓

解决方案：将长序列分割成小批量

- 每个批量包含多个样本
- 每个样本是固定长度的子序列

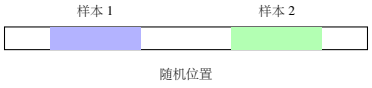
定理 1.28 (关键问题)

1. 如何分割长序列?
2. 如何保持序列的连续性?
3. 如何构建批量数据?

♡

1.9.12.0.2 两种采样策略 1. 随机采样

- 每个批量随机选起始位置
- 批量间相互独立
- 打乱了时序

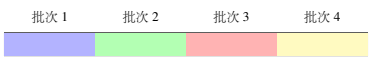


优点:

- 简单
- 并行友好

2. 顺序分区

- 保持批量间的连续性
- 按顺序划分
- 保留了时序关系



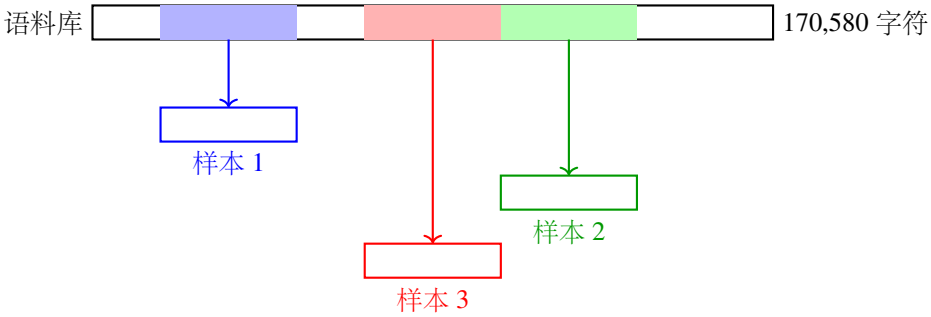
优点:

- 保持连续性
- 更符合实际

定理 1.29 (选择建议)

训练时通常使用顺序分区，以更好地利用上下文信息 ♡

1.9.12.0.3 随机采样详解 策略：从语料库中随机选择起始位置

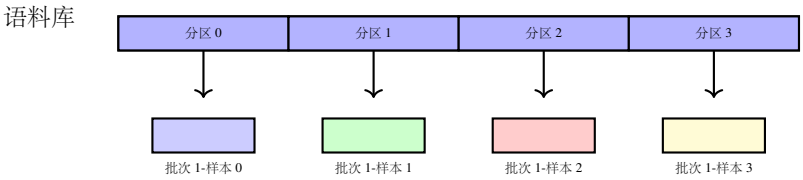


每个样本的起始位置随机选择，批量间无关联

1.9.12.0.4 随机采样实现

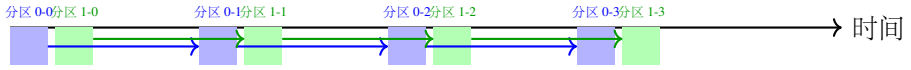
```
def seq_data_iter_random(corpus, batch_size, num_steps):
    """ Random sampling to generate mini-batch subsequences corpus: list of token indices batch_size:
        batch size num_steps: number of time steps per subsequence """
    corpus = corpus[random.randint(0, num_steps - 1):]

    num_subseqs = (len(corpus) - 1) // num_steps

    initial_indices = list(range(0, num_subseqs * num_steps, num_steps))

    random.shuffle(initial_indices)

    def data(pos):
        """Return sequence of length num_steps starting from pos"""
        return corpus[pos: pos + num_steps]

    # Generate batches
    num_batches = num_subseqs // batch_size
    for i in range(0, batch_size * num_batches, batch_size):
        # Get batch_size sequences
        initial_indices_per_batch = initial_indices[i: i + batch_size]

        X = [data(j) for j in initial_indices_per_batch]
        Y = [data(j + 1) for j in initial_indices_per_batch]

        yield torch.tensor(X), torch.tensor(Y)
```

1.9.12.0.5 顺序分区详解 策略：将语料库均匀分成 batch_size 份，顺序读取



批次 2、批次 3... 依次从每个分区顺序读取下一段

1.9.12.0.6 顺序分区的优势



关键：每个分区内部是连续的!
隐状态可以跨批次传递

定理 1.30 (隐状态传递)

- 批次 t 的最终隐状态 $\rightarrow$ 批次 $t+1$ 的初始隐状态
- 保持了长期依赖关系
- 更符合 RNN 的设计理念

♡

1.9.12.0.7 顺序分区实现

```
def seq_data_iter_sequential(corpus, batch_size, num_steps):
    """ sequence """
    offset = random.randint(0, num_steps)
    num_tokens = ((len(corpus) - offset - 1) // batch_size) * batch_size
    Xs = torch.tensor(corpus[offset: offset + num_tokens])
    Ys = torch.tensor(corpus[offset + 1: offset + 1 + num_tokens])

    # (batch_size, -1)
    Xs = Xs.reshape(batch_size, -1)
    Ys = Ys.reshape(batch_size, -1)

    num_batches = Xs.shape[1] // num_steps

    for i in range(0, num_steps * num_batches, num_steps):
        X = Xs[:, i: i + num_steps]
        Y = Ys[:, i: i + num_steps]
        yield X, Y
```

1.9.12.0.8 数据迭代器对比

特性	随机采样	顺序分区
批次间关系	独立	连续
隐状态传递	不需要	需要
实现复杂度	简单	中等
训练效果	较好	更好
适用场景	调试、快速实验	正式训练

例题 1.12 参数设置建议

- batch_size：32-64（取决于 GPU 内存）
- num_steps：35-100（序列长度）
- 总样本数 = batch_size × num_batches

1.9.12.0.9 使用示例

```
# data
corpus, vocab = load_corpus_time_machine()

batch_size, num_steps = 32, 35

random_iter = seq_data_iter_random(corpus, batch_size, num_steps)
for X, Y in random_iter:
    print(f'X shape: {X.shape}, Y shape: {Y.shape}')
    break
# Output: X shape: torch.Size([32, 35]), Y shape: torch.Size([32, 35])

seq_iter = seq_data_iter_sequential(corpus, batch_size, num_steps)
for X, Y in seq_iter:
    print(f'X shape: {X.shape}, Y shape: {Y.shape}')
    # X[i, :] Y[i, :]
    # X[i, j] Y[i, j]
    break
```

定义 1.31 (形状说明)

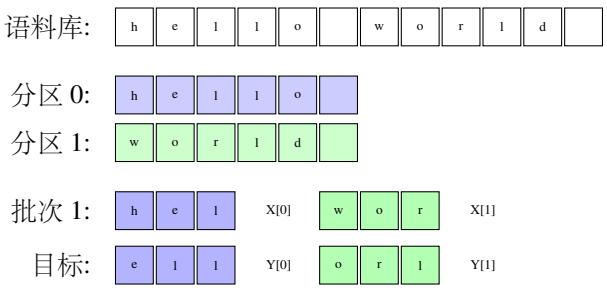
- X: (batch_size, num_steps) - 输入序列
- Y: (batch_size, num_steps) - 目标序列
- X[i, j] 的目标是 Y[i, j] (即 X[i, j+1])

♣

1.9.12.0.10 数据批次的可视化

假设：语料库 = “hello world this is a test”

参数：batch_size=2, num_steps=3



定义 1.32 (关键)

X 的每个字符的下一个字符就是 Y 中对应位置的字符

♣

1.9.12.0.11 完整的数据加载函数

```
class SeqDataLoader:
    """sequencedata"""
    def __init__(self, batch_size, num_steps, use_random_iter, max_tokens):
```

```
        if use_random_iter:
            self.data_iter_fn = seq_data_iter_random
        else:
            self.data_iter_fn = seq_data_iter_sequential

        self.corpus, self.vocab = load_corpus_time_machine(max_tokens)
        self.batch_size, self.num_steps = batch_size, num_steps

    def __iter__(self):
        return self.data_iter_fn(self.corpus, self.batch_size, self.num_steps)

def load_data_time_machine(batch_size, num_steps,
                           use_random_iter=False, max_tokens=10000):
    """Return时光机器data集的iteration器和vocabulary"""
    data_iter = SeqDataLoader(batch_size, num_steps,
                              use_random_iter, max_tokens)
    return data_iter, data_iter.vocab

train_iter, vocab = load_data_time_machine(batch_size=32, num_steps=35)
```

1.9.13 8.3 节总结

定义 1.33 (核心内容)

1. 语言模型：估计文本序列的概率分布

$$P(x_1, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_1, \ldots, x_{t-1})$$

2. 困惑度：评估语言模型的标准指标

$$\text{PPL} = \exp\left(-\frac{1}{n}\sum_{t=1}^{n} \log P(x_t \mid x_{<t})\right)$$

3. 马尔可夫假设：简化长期依赖

$$P(x_t \mid x_{<t}) \approx P(x_t \mid x_{t-\tau}, \ldots, x_{t-1})$$

4. 数据采样：随机采样 vs 顺序分区

♣

1.9.13.0.1 关键概念对比

概念	定义	作用
语言模型	文本序列的概率分布	生成、评估文本
困惑度	exp(负对数似然)	评估模型质量
N-gram	前 $n-1$ 个词预测	简单的语言模型
Zipf 定律	词频 $\propto$ 排名 $^{-1}$	理解词频分布
随机采样	随机起始位置	快速训练
顺序分区	保持连续性	更好效果

1.9.13.0.2 为什么需要 RNN?

N-gram 模型的问题：

- 固定的上下文窗口
- 无法捕捉长期依赖
- 参数随 τ 指数增长

RNN 的解决方案：

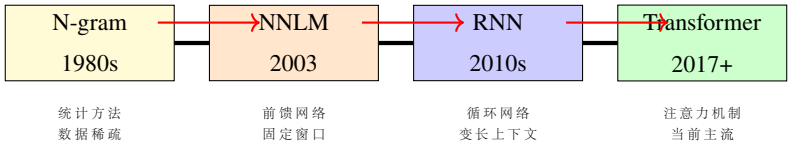
- 变长的上下文（隐状态）
- 能捕捉长期依赖
- 参数数量固定
- 共享参数，泛化能力强

定理 1.31 (下一步)

8.4 节将详细介绍循环神经网络（**RNN**）的原理和实现

♡

1.9.13.0.3 从统计到神经网络



1.10 8.4 循环神经网络

1.10.1 8.4 循环神经网络 - 章节导航

定义 1.34 (本节内容)

- 8.4.1 无隐状态的神经网络
- 8.4.2 有隐状态的循环神经网络
- 8.4.3 基于循环神经网络的字符级语言模型
- 8.4.4 困惑度（Perplexity）

♣

定理 1.32 (学习目标)

1. 理解为什么需要隐状态
2. 掌握 RNN 的前向传播过程
3. 了解 RNN 的参数共享机制
4. 学会用 RNN 构建语言模型

♡

1.11 未命名节

1.11.1 无隐状态的神经网络

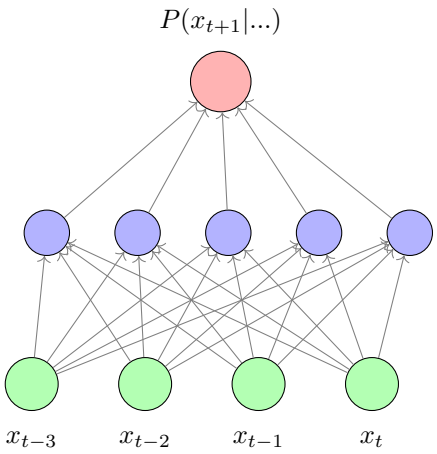
1.11.2 8.4.1 无隐状态的神经网络

1.11.3 8.4.1 无隐状态的神经网络

从简单开始：没有记忆的网络

1.11.3.0.1 尝试 1：直接用 MLP　想法：用多层感知机直接建模语言模型

$$P(x_t \mid x_{t-1}, x_{t-2}, \ldots, x_{t-\tau})$$



问题：

- 输入维度固定为 τ
- 参数数量随 τ 增长
- 无法处理变长序列

1.11.3.0.2 MLP 语言模型的局限

问题 1：窗口大小固定

- $\tau = 4$：只能看 4 个词
- 长期依赖无法捕捉
- “I grew up in France … I speak fluent …”

问题 2：参数数量

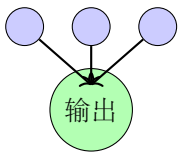
- 输入层：$\tau \times d_{embed}$
- 词表大：$V = 10000$，$\tau = 100 \Rightarrow$ 百万级参数

问题 3：无法参数共享

- 处理位置 1 和位置 2 的 “the” 用不同参数

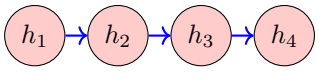
1.11.3.0.3 需要一个”记忆装置”　核心思想：引入**隐状态**作为记忆

MLP：无记忆



每次重新开始

RNN：有记忆



隐状态记住历史

定理 1.33 (关键)

隐状态 h_t 总结了到时刻 t 为止的所有历史信息 ♡

1.11.4 有隐状态的循环神经网络

1.11.5 8.4.2 有隐状态的循环神经网络

1.11.6 8.4.2 有隐状态的循环神经网络

RNN 的核心：循环连接

1.11.6.0.1 RNN 的基本结构 核心思想：隐状态在时间步之间传递

定义 1.35 (数学定义)

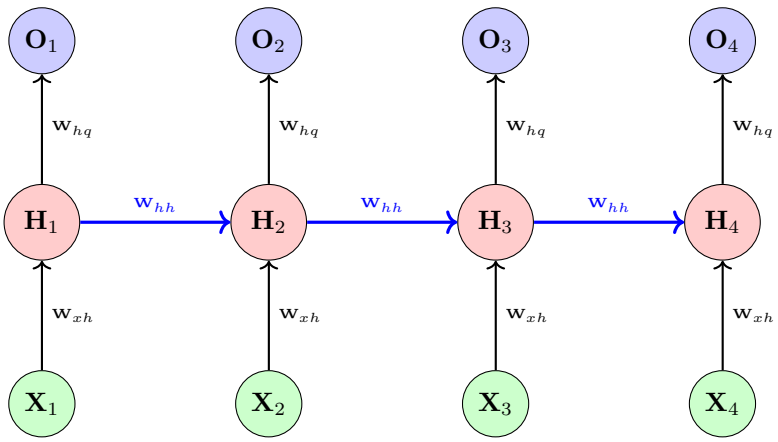
$$\mathbf{H}_t = \phi(\mathbf{X}_t\mathbf{W}_{xh} + \mathbf{H}_{t-1}\mathbf{W}_{hh} + \mathbf{b}_h)$$
$$\mathbf{O}_t = \mathbf{H}_t\mathbf{W}_{hq} + \mathbf{b}_q$$

♣

符号说明：

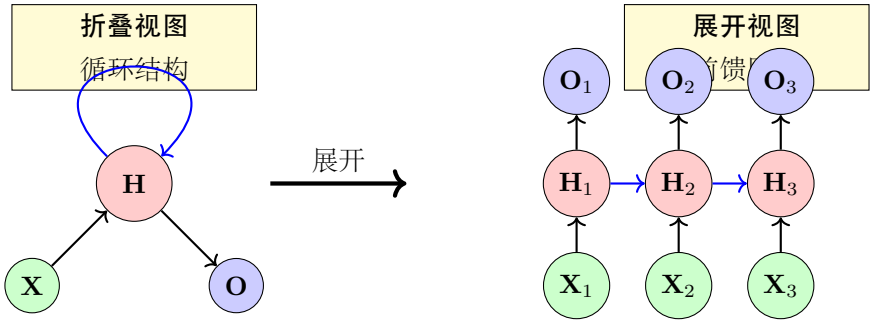
- $\mathbf{X}_t \in \mathbb{R}^{n\times d}$：时间步 t 的输入（n 个样本）
- $\mathbf{H}_t \in \mathbb{R}^{n\times h}$：隐状态（$h$ 是隐藏单元数）
- $\mathbf{O}_t \in \mathbb{R}^{n\times q}$：输出（$q$ 是输出维度）
- $\mathbf{W}_{xh} \in \mathbb{R}^{d\times h}$：输入到隐藏的权重
- $\mathbf{W}_{hh} \in \mathbb{R}^{h\times h}$：隐藏到隐藏的权重
- $\mathbf{W}_{hq} \in \mathbb{R}^{h\times q}$：隐藏到输出的权重
- ϕ：激活函数（通常是 tanh）

1.11.6.0.2 RNN 的计算图



关键：相同的权重在所有时间步共享！

1.11.6.0.3 RNN 的展开视图 将循环展开成前馈网络：

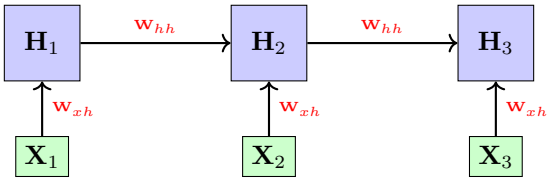


定理 1.34 (关键特点)

- 展开后是一个很深的网络（深度 = 序列长度）
- 所有时间步共享相同的参数 $\mathbf{W}_{xh}, \mathbf{W}_{hh}, \mathbf{W}_{hq}$

♡

1.11.6.0.4 RNN 的参数共享 为什么参数共享?



相同的权重 ⇒ 处理 “the” 用同样的变换

优点:

- ✓ 参数数量与序列长度无关
- ✓ 可以处理任意长度的序列
- ✓ 学到的模式可以迁移到序列的任何位置

1.11.6.0.5 RNN 的前向传播过程 逐步计算:

1. **初始化:** $\mathbf{H}_0 = \mathbf{0}$（或学习得到）
2. **时间步 1:**

$$\mathbf{H}_1 = \tanh(\mathbf{X}_1\mathbf{W}_{xh} + \mathbf{H}_0\mathbf{W}_{hh} + \mathbf{b}_h)$$
$$\mathbf{O}_1 = \mathbf{H}_1\mathbf{W}_{hq} + \mathbf{b}_q$$

3. **时间步 2:**

$$\mathbf{H}_2 = \tanh(\mathbf{X}_2\mathbf{W}_{xh} + \mathbf{H}_1\mathbf{W}_{hh} + \mathbf{b}_h)$$
$$\mathbf{O}_2 = \mathbf{H}_2\mathbf{W}_{hq} + \mathbf{b}_q$$

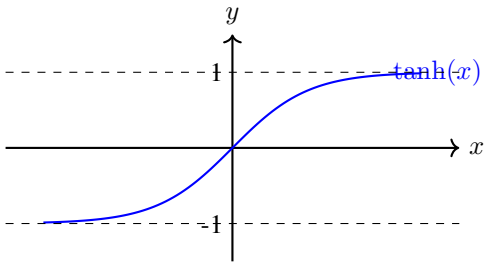
4. **... 重复...**

定理 1.35 (关键)

每一步都依赖前一步的隐状态 $\mathbf{H}_{t-1}$

♡

1.11.6.0.6 激活函数的选择 为什么用 tanh?



输出范围：$[-1,1]$ | 平滑 | 零中心

优点：

- 输出有界
- 零中心（mean=0）
- 梯度更稳定

也可以用：

- ReLU：计算快
- sigmoid：$[0,1]$ 输出
- 但 tanh 最常用

1.11.6.0.7 RNN 前向传播代码

```
def rnn_forward(inputs, state, params):
    """ inputs: sequence, shape (batch_size, vocab_size) state: (batch_size, num_hiddens) params: [W_xh
        , W_hh, b_h, W_hq, b_q] """
    W_xh, W_hh, b_h, W_hq, b_q = params
    outputs = []
    H = state

    for X in inputs:
        H = torch.tanh(torch.mm(X, W_xh) + torch.mm(H, W_hh) + b_h)
        # Output
        Y = torch.mm(H, W_hq) + b_q
        outputs.append(Y)

    return torch.cat(outputs, dim=0), H
```

定义 1.36 (关键点)

- 循环遍历每个时间步
- 隐状态 H 不断更新
- 返回所有输出和最终隐状态

♣

1.11.6.0.8 隐状态的继续输出　维度分析

定义 1.37 (重要：理解各个张量的形状)

变量	形状	说明
$\mathbf{X}_t$	(n, d)	批量大小 n，输入维度 d
$\mathbf{H}_t$	(n, h)	批量大小 n，隐藏单元数 h
$\mathbf{W}_{xh}$	(d, h)	输入到隐藏
$\mathbf{W}_{hh}$	(h, h)	隐藏到隐藏
$\mathbf{W}_{hq}$	(h, q)	隐藏到输出
$\mathbf{b}_h$	$(h,)$	隐藏层偏置
$\mathbf{b}_q$	$(q,)$	输出层偏置

例题 1.13 示例 批量大小 $n = 32$，输入维度 $d = 256$，隐藏单元 $h = 128$，输出维度 $q = 100$

1.11.7 基于循环神经网络的字符级语言模型

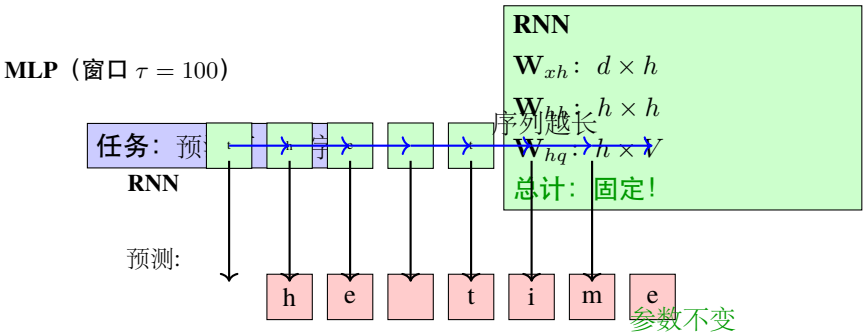
1.11.8 8.4.3 基于循环神经网络的字符级语言模型

1.11.9 8.4.3 基于循环神经网络的字符级语言模型

目标：用 RNN 构建字符级语言模型

实现一个能生成文本的 RNN 模型

1.11.9.0.1 字符级语言模型



定义 1.38 (RNN 的优势)

参数共享使得 RNN 可以高效处理任意长度的序列

1.11.9.0.2 字符级语言模型

给定 “the” 预测 “h”，给定 “th” 预测 “e”...

定理 1.36 (关键思想)

每个时间步都进行预测，使用 **teacher forcing** 训练

1.11.9.0.3 One-Hot 编码 字符 → 向量

字符	One-Hot 向量
'a'	$[1,0,0,0,0]$
'b'	$[0,1,0,0,0]$
'c'	$[0,0,1,0,0]$
'd'	$[0,0,0,1,0]$
'e'	$[0,0,0,0,1]$

定义 1.39 (特点)

- 向量维度 = 词汇表大小
- 每个向量只有一个 1，其余为 0
- 稀疏表示，计算效率低

♣

1.11.10 困惑度

1.11.11 8.4.4 困惑度

1.11.12 8.4.4 困惑度（Perplexity）

定义：评估语言模型性能的指标

$$\textbf{PPL} = \exp\left(-\frac{1}{N}\sum_{t=1}^{N}\log p(x_t|x_{<t})\right)$$

好的模型：

- 高概率预测下一个词
- 低困惑度
- PPL = $\exp(0.25) \approx 1.3$

差的模型：

- 低概率预测下一个词
- 高困惑度
- PPL = $\exp(2.3) \approx 10$

定理 1.37 (直观理解)

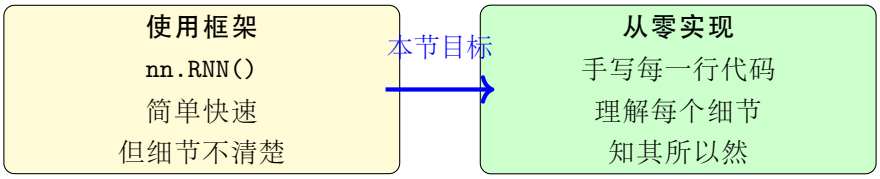
困惑度 = 模型在预测时的"平均选择数量"

♡

1.12 从零开始实现循环神经网络

1.13 8.5 从零开始实现循环神经网络

1.13.0.0.1 从零实现的意义



定义 1.40 (你将学会)

- RNN 的完整工作流程
- 梯度计算和反向传播
- 序列建模的技巧和 tricks
- 为学习更复杂的模型打下基础

♣

1.13.0.0.2 One-Hot 编码的优势

定义 1.41 (为什么用 One-Hot？)

- 简单直观
- 无大小关系（不会让模型认为'a' < 'b'）
- 与序列长度无关（参数量与序列长度无关）

♣

定理 1.38 (与词级别的区别)

- 词级别：通常几千到几万个词
- 字符级别：通常几十个字符
- 字符级别模型参数更少，训练更快

♡

1.13.0.0.3 One-Hot 编码实现

```python
import torch.nn.functional as F

def get_one_hot(indices, vocab_size):
    """␣one-hotvector␣indices:␣shape␣(N,)␣vocab_size:␣"""
    one_hot = torch.zeros(len(indices), vocab_size)

    # 1
    one_hot.scatter_(1, indices.unsqueeze(1), 1)

    return one_hot

indices = torch.tensor([0, 2, 4]) # 'a', 'c', 'e'
one_hot = get_one_hot(indices, vocab_size=5)
print(one_hot)
```

定义 1.42 (输出结果)

```
tensor([[1., 0., 0., 0., 0.],  # 'a'
        [0., 0., 1., 0., 0.],  # 'c'
        [0., 0., 0., 0., 1.]]) # 'e'
```

♣

1.13.0.0.4 独热编码的维度转换 **问题**：字符是离散符号，需要转为向量

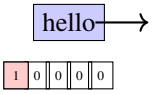
定义 1.43 (独热编码：)

输入：形状为 (batch_size, num_steps) 的索引

输出：形状为 (num_steps, batch_size, vocab_size) 的 one-hot 张量

♣

词表长度为 V 的向量表



长度为 V 的向量，只有一个位置为 1

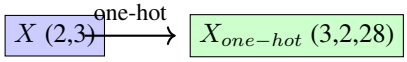
- 每个字符对应一个位置
- 便于神经网络处理
- 但向量维度很高

1.13.0.0.5 批处理独热编码

```python
import torch

# data
batch_size, num_steps, vocab_size = 2, 3, 28
X = torch.tensor([[1, 2, 3], [4, 5, 6]]) # (2, 3)

X_one_hot = get_one_hot(X, vocab_size)
print(f"shape: {X_one_hot.shape}") # (3, 2, 28)
```

定理 1.39 (注意)

转置是为了方便 RNN 按时间步处理：(T, N, V)



批量索引　　批量向量

1.13.0.0.6 RNN 语言模型的完整流程

- **步骤 1: 数据准备** - 文本 → 字符索引 → One-hot 向量
- **步骤 2: RNN 计算** - 循环神经网络处理序列
- **步骤 3: 输出预测** - 计算下一个字符的概率分布
- **步骤 4: 采样** - 根据概率选择下一个字符

1.13.0.0.7 步骤 3: 输出预测

- 向输出层：$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hq} + \mathbf{b}_q$
- 输出维度 = 词表大小
- 稀疏表示（未归一化的 logits）

1.13.0.0.8 步骤 4: 采样　根据概率分布选择下一个字符

1.13.0.0.9 独热编码实现

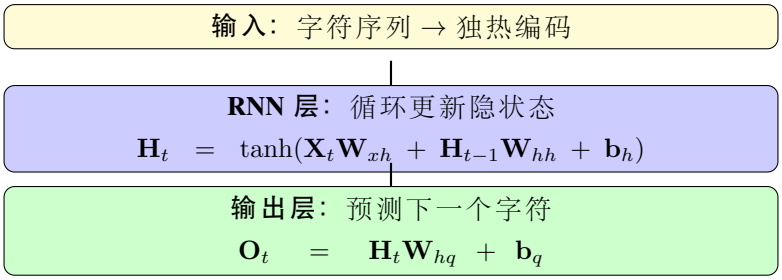
```python
import torch
import torch.nn.functional as F

def one_hot(indices, vocab_size):
    """ indices: (batch_size, num_steps) Return: (batch_size, num_steps, vocab_size) """
    return F.one_hot(indices, vocab_size).float()
```

```

# comment
vocab_size = 28 # 26个字母 + 空格 + 其他
batch_size, num_steps = 2, 5

# comment
X = torch.randint(0, vocab_size, (batch_size, num_steps))
print(f"索引形状:␣{X.shape}") # (2, 5)

# comment
X_one_hot = one_hot(X, vocab_size)
print(f"独热编码形状:␣{X_one_hot.shape}") # (2, 5, 28)
```

1.13.0.0.10 完整的 RNN 模型

```
class RNNModelScratch:
    def __init__(self, vocab_size, num_hiddens):
        self.vocab_size = vocab_size
        self.num_hiddens = num_hiddens

        #  comment
        self.params = self.init_params()

    def init_params(self):
        """Simple function"""
        def normal(shape):
            return torch.randn(size=shape) * 0.01

        # comment
        W_xh = normal((self.vocab_size, self.num_hiddens))
        # comment
        W_hh = normal((self.num_hiddens, self.num_hiddens))
        # comment
        b_h = torch.zeros(self.num_hiddens)
        # comment
        W_hq = normal((self.num_hiddens, self.vocab_size))
        # comment
        b_q = torch.zeros(self.vocab_size)

        # comment
        params = [W_xh, W_hh, b_h, W_hq, b_q]
        for param in params:
            param.requires_grad_(True)

        return params
```

1.13.0.0.11 完整的字符级 RNN 语言模型



定理 1.40 (目标)

最大化正确字符的预测概率 ♡

1.13.0.0.12 字符级模型的优缺点 优点：

- ✓ 词表很小（~100）
- ✓ 无 OOV 问题
- ✓ 可以生成任意词
- ✓ 适合演示和教学

适用场景：

- 拼写纠错
- 文本生成
- DNA 序列分析

缺点：

- × 序列很长
- × 训练慢
- × 需要更多计算
- × 难以捕捉长期依赖

实际应用：

- 现代 NLP 多用词级或子词级

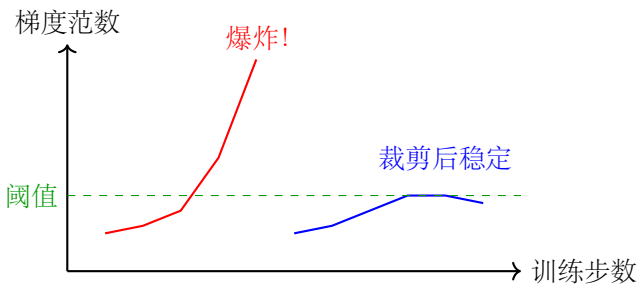
1.13.0.0.13 训练循环

```
def train_epoch(model, train_iter, loss_fn, optimizer):
    """epoch"""
    total_loss = 0
    num_batches = 0

    for X, Y in train_iter:
        # X, Y: (batch_size, num_steps)

        # One-hot编码
        inputs = get_one_hot(X, model.vocab_size)
        # inputs: (num_steps, batch_size, vocab_size)

        # comment
        state = model.init_state(X.shape[0])

        # comment
        outputs, state = model.forward(inputs, state)
        # outputs: (num_steps * batch_size, vocab_size)
```

```
        # comment
        Y = Y.T.reshape(-1) # (num_steps * batch_size,)
        loss = loss_fn(outputs, Y)

        # comment
        optimizer.zero_grad()
        loss.backward()
        # comment
        grad_clipping(model.params, 1)
        optimizer.step()

        total_loss += loss.item()
        num_batches += 1

    return total_loss / num_batches
```

1.13.0.0.14 梯度裁剪的重要性

问题：RNN 容易出现梯度爆炸



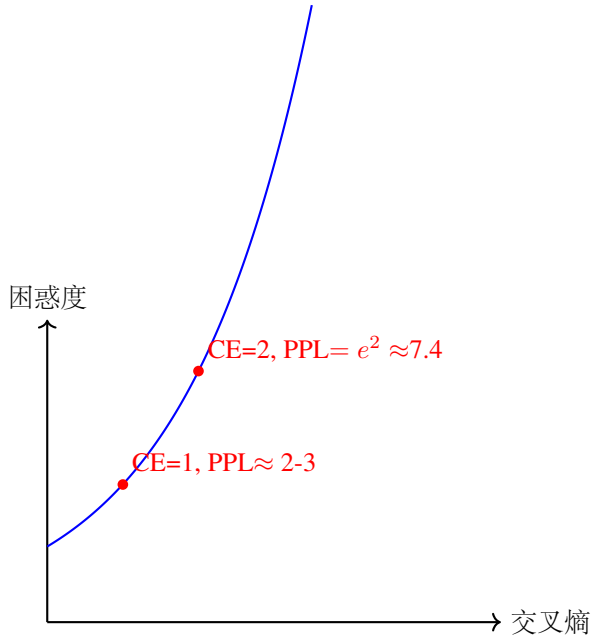
定义 1.44 (梯度裁剪)

$$\mathbf{g} \leftarrow \min\left(1, \frac{\theta}{\|\mathbf{g}\|}\right)\mathbf{g}$$

其中 θ 是阈值（通常取 1 或 5）

♣

1.13.0.0.15 困惑度与交叉熵的关系



$$\text{PPL} = \exp(\text{CE})$$

定理 1.41 (注意)

困惑度对交叉熵的微小差异很敏感（指数关系）

1.13.0.0.16 困惑度的意义

定义 1.45 (直观理解)

困惑度告诉我们：在每一步，模型平均在多少个选项之间“困惑”

例题 1.14 示例

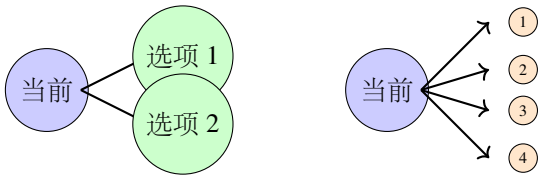
- PPL = 2：模型在 2 个选项间困惑
- PPL = 28：模型在 28 个字符间困惑（相当于随机猜）
- PPL = 1.5：模型比较确定

1.13.0.0.17 计算困惑度

```
def evaluate_perplexity(model, data_iter):
    """modeldata"""
    model.eval()
    total_loss = 0
    total_tokens = 0

    with torch.no_grad():
        for X, Y in data_iter:
            # One-hot编码
            inputs = get_one_hot(X, model.vocab_size)
            state = model.init_state(X.shape[0])

            outputs, state = model.forward(inputs, state)
            # output: (num_steps * batch_size, vocab_size)

```

```
            # loss
            Y = Y.T.reshape(-1)
            loss = F.cross_entropy(outputs, Y, reduction='sum')

            total_loss += loss.item()
            total_tokens += Y.numel()

    # comment = exp(平均交叉熵)
    perplexity = math.exp(total_loss / total_tokens)
    return perplexity

# comment
ppl = evaluate_perplexity(model, test_iter)
print(f"test集困惑度: {ppl:.2f}")
```

1.13.0.0.18 困惑度的实际意义 困惑度 = 平均分支因子



PPL = 2：在 2 个选项间困惑 PPL = 4：在 4 个选项间困惑

1.13.0.0.19 困惑度的实际值 不同模型的困惑度范围：

模型	数据集	PPL
随机模型	任意	~ 词表大小
N-gram (n=3)	Penn Treebank	~140
简单 RNN	Penn Treebank	~120
LSTM	Penn Treebank	~80
LSTM + 正则化	Penn Treebank	~60
Transformer (大)	Penn Treebank	~20-30
GPT-3	网络文本	~20

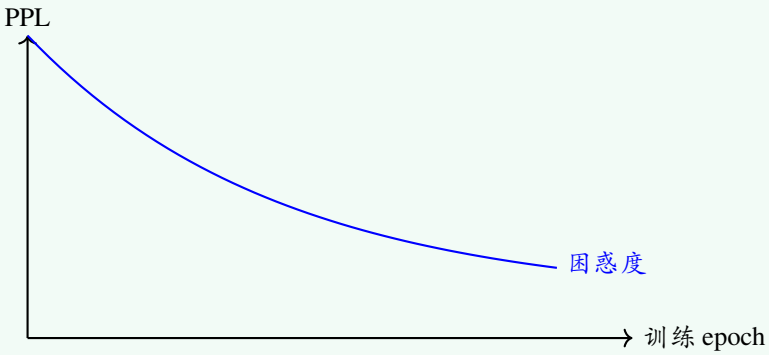
定理 1.42 (注意)

- 不同数据集的 PPL 不具可比性
- 字符级模型的 PPL 通常更低（词表小）
- PPL 是相对指标，越低越好
- PPL=4：平均 4 个选择

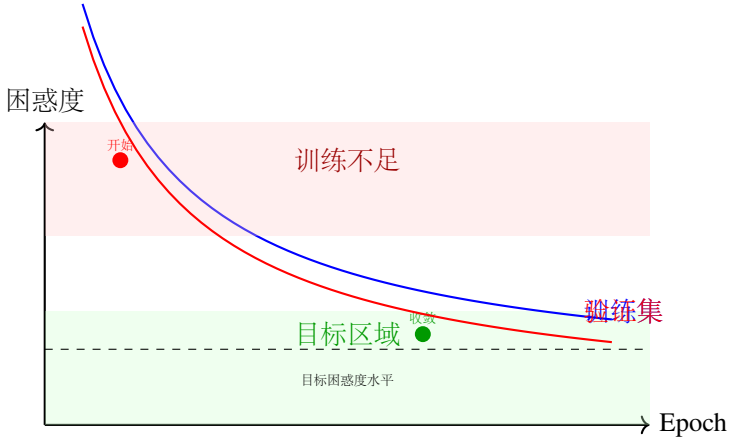
♡

1.13.0.0.20 困惑度与准确率的关系

定义 1.46 (解释)



1.13.0.0.21 训练过程中的困惑度变化



定义 1.47 (训练过程)

- 初期：PPL 很高（模型随机猜测）
- 中期：PPL 快速下降（学习规律）
- 后期：PPL 缓慢下降（微调，后期趋于平稳）
- 最终：PPL 稳定（收敛）

定理 1.43 (过拟合检测)

- 训练集 PPL 继续下降，验证集 PPL 开始上升 → 过拟合
- 两者的 PPL 都稳定下降 → 训练良好

1.13.1 8.4 节总结

定义 1.48 (核心概念)

1. 隐状态：记忆历史信息的向量
2. **RNN**：通过隐状态记住历史
3. 参数共享：所有时间步共享权重

1.13.2 8.4 节总结

定义 1.49 (数学公式)

无隐状态网络：MLP 无法处理变长序列

$$\mathbf{H}_t = \phi(\mathbf{X}_t\mathbf{W}_{xh} + \mathbf{b})$$

RNN：通过隐状态记住历史

$$\mathbf{H}_t = \tanh(\mathbf{X}_t\mathbf{W}_{xh} + \mathbf{H}_{t-1}\mathbf{W}_{hh} + \mathbf{b}_h)$$

参数共享：所有时间步共享权重

♣

1.13.3 8.4 节总结

定义 1.50 (模型类型)

- 字符级模型：用独热编码表示字符
- 词级模型：用词嵌入表示词语

♣

定义 1.51 (评估指标)

困惑度：评估模型质量

$$\text{PPL} = \exp\left(-\frac{1}{n}\sum_{t=1}^{n}\log P(x_t|x_{<t})\right)$$

♣

1.13.3.0.1 RNN 的关键特性

1. 参数共享

$\mathbf{W}_{xh}, \mathbf{W}_{hh}, \mathbf{W}_{hy}$ 在所有时间步共享

1.13.3.0.2 RNN 的优缺点 优点：

- ✓ 能处理变长序列
- ✓ 参数共享，所有时间步复用泛化好
- ✓ 参数量与序列长度无关

缺点：

- × 梯度消失/爆炸问题
- × 长距离依赖建模困难
- × 无法并行计算（必须按顺序）

1.13.3.0.3 RNN 的关键特性

2. 顺序处理

必须按顺序处理：$\mathbf{H}_1 \to \mathbf{H}_2 \to \mathbf{H}_3$

3. 理论上的记忆能力

隐状态 $\mathbf{H}_t$ 可以编码所有历史

$\Rightarrow$ 但实际受限于梯度消失/爆炸

定理 1.44 (下一步)

8.5 节将从零实现 RNN，深入理解其工作机制

♡

1.13.3.0.4 RNN 的应用场景 适合 RNN 的任务：

- 语言建模
- 机器翻译
- 语音识别
- 时间序列预测
- 视频分析

RNN 的挑战：

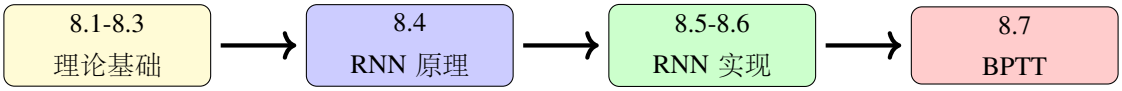
- 梯度消失/爆炸
- 长期依赖问题
- 训练慢（顺序处理）
- 难以并行

定义 1.52 (改进方向)

- LSTM/GRU：解决梯度问题（第 9 章）
- 注意力机制：捕捉长期依赖
- Transformer：完全并行化（后续章节）

latex

1.13.3.0.5 从理论到实践



我们已经理解了 RNN 的**工作原理**
接下来将**从零实现** RNN（8.5 节）

定义 1.53 (已掌握的知识)

- ✓ 为什么需要隐状态
- ✓ RNN 的数学公式
- ✓ 参数共享机制
- ✓ 字符级语言模型
- ✓ 困惑度评估

1.13.4 8.4 节核心公式总结

定义 1.54 (RNN 的核心公式)

$$\begin{aligned}
\text{隐状态更新：} \quad & \mathbf{H}_t = \phi(\mathbf{X}_t\mathbf{W}_{xh} + \mathbf{H}_{t-1}\mathbf{W}_{hh} + \mathbf{b}_h) \\
\text{输出计算：} \quad & \mathbf{O}_t = \mathbf{H}_t\mathbf{W}_{hq} + \mathbf{b}_q \\
\text{概率预测：} \quad & P(x_{t+1}) = \text{softmax}(\mathbf{O}_t) \\
\text{困惑度：} \quad & \text{PPL} = \exp\left(-\frac{1}{n}\sum_{t=1}^{n}\log P(x_t \mid x_{<t})\right)
\end{aligned}$$

定理 1.45 (关键要点)

- 隐状态 $\mathbf{H}_t$ 携带历史信息
- 参数 $\mathbf{W}_{xh}, \mathbf{W}_{hh}, \mathbf{W}_{hq}$ 在所有时间步共享
- 激活函数通常使用 tanh
- 困惑度越低，模型越好

♡

1.13.4.0.1 RNN 与传统模型对比

特性	N-gram	MLP	RNN
处理变长序列	×	×	✓
参数数量	随 τ 指数增长	随 τ 线性增长	固定
长期依赖	受限于 τ	受限于 τ	理论上可以
并行化	✓	✓	×
参数共享	×	×	✓
训练难度	简单	中等	较难

定义 1.55 (RNN 的核心优势)

通过隐状态和参数共享，实现了用固定参数处理任意长度序列

♣

1.13.4.0.2 训练 RNN 的关键技巧 1. 梯度裁剪

- 防止梯度爆炸
- 阈值通常取 1 或 5
- 必不可少!

2. 隐状态初始化

- 零初始化
- 可学习初始化
- 使用前一批次的隐状态

3. 序列采样

- 随机采样：简单
- 顺序分区：效果好
- 保持隐状态连续性

4. 学习率调整

- 从较小值开始
- 使用学习率衰减
- 监控梯度范数

定理 1.46 (经验)

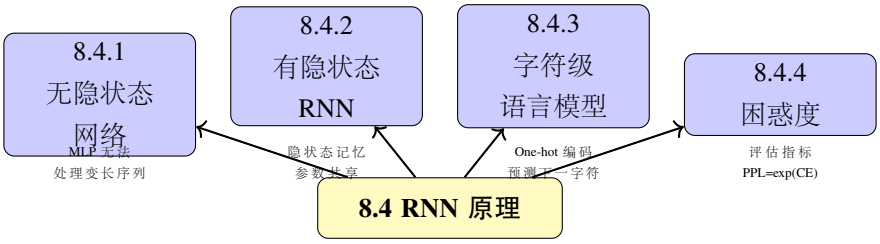
RNN 训练需要仔细调参，梯度裁剪是关键!

♡

1.13.4.0.3 常见问题与解决方案

问题 1：梯度爆炸 梯度突然变得很大	**解决：** 梯度裁剪 + 小学习率
问题 2：梯度消失 长期依赖学不到	**解决：** 使用 LSTM/GRU（第 9 章）
问题 3：训练慢 顺序处理效率低	**解决：** 截断 BPTT + GPU 加速

1.13.5 8.4 节思维导图



核心公式： $\mathbf{H}_t = \tanh(\mathbf{X}_t\mathbf{W}_{xh} + \mathbf{H}_{t-1}\mathbf{W}_{hh} + \mathbf{b}_h)$

1.13.5.0.1 知识检查

定义 1.56 (问题 1)

为什么 RNN 需要隐状态？传统 MLP 有什么问题？

♣

定义 1.57 (问题 2)

RNN 的参数数量与序列长度有关吗？为什么？

♣

定义 1.58 (问题 3)

什么是困惑度？如何解释 PPL=10？

♣

定义 1.59 (问题 4)

为什么字符级 RNN 的困惑度通常比词级低？

♣

定理 1.47 (思考)

如何解决 RNN 的梯度消失问题？（提示：LSTM）

♡

1.13.5.0.2 准备进入 8.5 节

从理论到实践：从零实现 RNN

定义 1.60 (8.4 节回顾)

- 理解了 RNN 的核心思想：用隐状态记忆历史
- 掌握了 RNN 的数学原理：前向传播公式
- 学会了评估模型：困惑度指标

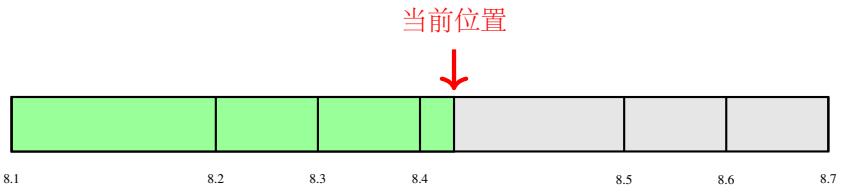
♣

定义 1.61 (8.5 节预告)

- 从零实现 RNN 的每个细节
- 独热编码、初始化参数
- 前向传播、梯度裁剪
- 训练完整的字符级语言模型
- 文本生成

♣

1.13.5.0.3 第 8 章进度总结



已完成：序列模型、文本预处理、语言模型、RNN 原理
下一步：RNN 从零实现（8.5）

1.14 8.5 循环神经网络的从零开始实现

1.14.1 8.5 循环神经网络的从零开始实现

动手实现：从零开始构建 RNN

1.14.2 8.5 节 - 章节导航

定义 1.62 (本节内容)

- 8.5.1 独热编码
- 8.5.2 初始化模型参数
- 8.5.3 循环神经网络模型
- 8.5.4 预测
- 8.5.5 梯度裁剪
- 8.5.6 训练

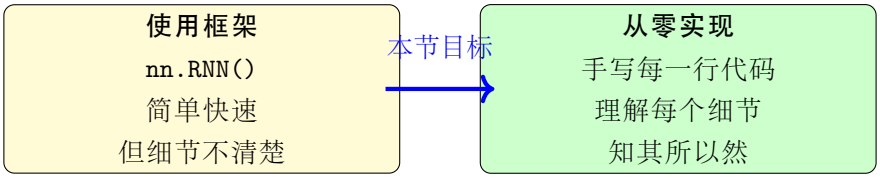
♣

定理 1.48 (学习目标)

1. 手写独热编码函数
2. 理解 RNN 参数初始化
3. 实现 RNN 前向传播
4. 生成文本预测
5. 掌握梯度裁剪技巧
6. 训练完整的字符级语言模型

♡

1.14.2.0.1 从零实现的意义



定义 1.63 (你将学会)

- 如何将文本转为张量
- RNN 的参数有哪些，形状是什么
- 前向传播的每一步计算
- 如何生成新文本
- 训练的完整流程

♣

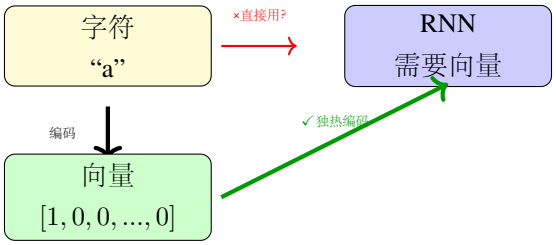
1.14.3 独热编码

1.14.4 8.5.1 独热编码

1.14.5 8.5.1 独热编码

第一步：将字符转为向量

1.14.5.0.1 为什么需要独热编码？ 问题：神经网络只能处理数字

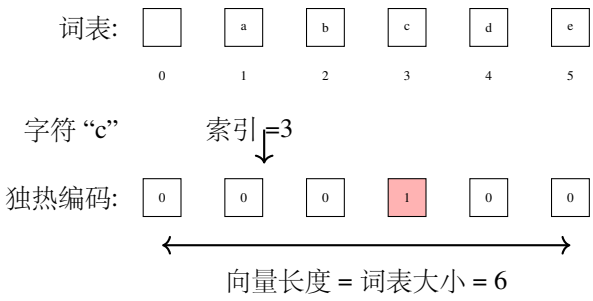


定理 1.49 (核心思想)

用长度为词表大小的向量表示每个字符，只有一个位置为 1，其余为 0

♡

1.14.5.0.2 独热编码详解 假设词表：`vocab = ' ':0, 'a':1, 'b':2, 'c':3, 'd':4, 'e':5`

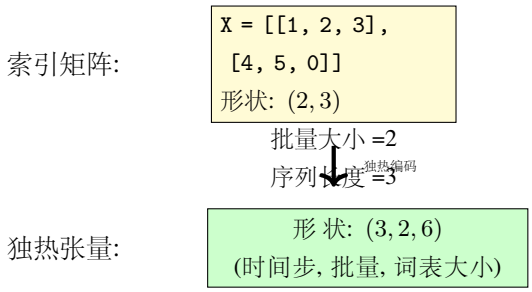


定义 1.64 (特点)

- 向量维度 = 词表大小
- 稀疏表示：只有一个 1，其余全是 0
- 没有顺序关系：“a” 和 “b” 距离相同

♣

1.14.5.0.3 批量独热编码 输入：批量的索引矩阵



定理 1.50 (维度变化)

$(batch\_size, num\_steps) \rightarrow (num\_steps, batch\_size, vocab\_size)$ ♡

1.14.5.0.4 独热编码实现 - 方法 1（手动）

```
import torch

def one_hot_manual(X, vocab_size):
    """ X: (batch_size, num_steps) vocab_size: vocabulary Return: (num_steps, batch_size, vocab_size)
        """
    batch_size, num_steps = X.shape

    output = torch.zeros((num_steps, batch_size, vocab_size))

    # X: (num_steps, batch_size)
    X = X.T

    # 1
    for t in range(num_steps):
       for b in range(batch_size):
           idx = X[t, b]
           output[t, b, idx] = 1

    return output

X = torch.tensor([[1, 2, 3], [4, 5, 0]])
result = one_hot_manual(X, vocab_size=6)
print(f"shape: {result.shape}") # (3, 2, 6)
```

1.14.5.0.5 独热编码实现 - 方法 2（PyTorch）

```
import torch.nn.functional as F

def one_hot(X, vocab_size):
    """ PyTorch X: (batch_size, num_steps) Return: (num_steps, batch_size, vocab_size) """
    # : (num_steps, batch_size)
    X = X.T

    output = F.one_hot(X, vocab_size)

```

```
    # float32neural network
    return output.to(torch.float32)

X = torch.tensor([[1, 2, 3], [4, 5, 0]])
result = one_hot(X, vocab_size=6)
print(f"shape: {result.shape}") # (3, 2, 6)
print(f"data类型: {result.dtype}") # torch.float32

print(f"第1个time step, 第1个样本:")
print(result[0, 0]) # [0, 1, 0, 0, 0, 0]
```

定义 1.65 (推荐)

使用 PyTorch 内置的 F.one_hot, 更高效!

♣

1.14.5.0.6 独热编码的优缺点 优点:

- ✓ 简单直观
- ✓ 字符间无顺序假设
- ✓ 易于理解和实现
- ✓ 适合小词表

适用场景:

- 字符级模型（词表小）
- 分类任务
- 教学演示

缺点:

- × 高维稀疏
- × 无语义信息
- × 内存占用大
- × 无法表示相似性

改进方案:

- 词嵌入（Word Embedding）
- 降维表示
- 预训练向量

定理 1.51 (实际应用)

现代 NLP 通常用词嵌入代替独热编码（后续章节）

♡

1.14.5.0.7 完整的数据准备

```
def prepare_data(batch_size, num_steps):
    """Prepare training data"""
    # data
    train_iter, vocab = load_data_time_machine(
        batch_size, num_steps
    )

    # batch
```

```
    for X, Y in train_iter:
        print(f"Xshape（索引）: {X.shape}") # (batch_size, num_steps)
        print(f"Yshape（标签）: {Y.shape}") # (batch_size, num_steps)

        X_one_hot = one_hot(X, len(vocab))
        print(f"X_one_hotshape: {X_one_hot.shape}")
        # (num_steps, batch_size, vocab_size)

        break

    return train_iter, vocab

train_iter, vocab = prepare_data(batch_size=32, num_steps=35)
print(f"vocabulary大小: {len(vocab)}")
```

1.14.6 初始化模型参数

1.14.7 8.5.2 初始化模型参数

1.14.8 8.5.2 初始化模型参数

第二步：初始化 RNN 的权重和偏置

1.14.8.0.1 RNN 需要哪些参数？ 回顾公式：

$$\mathbf{H}_t = \tanh(\mathbf{X}_t\mathbf{W}_{xh} + \mathbf{H}_{t-1}\mathbf{W}_{hh} + \mathbf{b}_h)$$
$$\mathbf{O}_t = \mathbf{H}_t\mathbf{W}_{hq} + \mathbf{b}_q$$

需要初始化的参数：

1. $\mathbf{W}_{xh}$：输入到隐藏层权重，形状 $(vocab\_size, num\_hiddens)$
2. $\mathbf{W}_{hh}$：隐藏层到隐藏层权重，形状 $(num\_hiddens, num\_hiddens)$
3. $\mathbf{b}_h$：隐藏层偏置，形状 $(num\_hiddens,)$
4. $\mathbf{W}_{hq}$：隐藏层到输出权重，形状 $(num\_hiddens, vocab\_size)$
5. $\mathbf{b}_q$：输出层偏置，形状 $(vocab\_size,)$

定义 1.66 (关键)

这 5 个参数在所有时间步共享！ ♣

1.14.8.0.2 参数形状分析 假设：词表大小 $V = 28$，隐藏单元数 $h = 512$

参数	形状	参数量
$\mathbf{W}_{xh}$	$(28, 512)$	14,336
$\mathbf{W}_{hh}$	$(512, 512)$	262,144
$\mathbf{b}_h$	$(512,)$	512
$\mathbf{W}_{hq}$	$(512, 28)$	14,336
$\mathbf{b}_q$	$(28,)$	28
总计		**291,356**

定理 1.52 (观察)

- 大部分参数在 $\mathbf{W}_{hh}$（90%）
- 与序列长度无关
- 参数量主要由 $num\_hiddens$ 决定

♡

1.14.8.0.3 参数初始化实现

```python
import torch

def get_params(vocab_size, num_hiddens, device):
    """ RNNparameters vocab_size: vocabulary num_hiddens: device: （CPU/GPU） """
    num_inputs = num_outputs = vocab_size

    def normal(shape):
        """正态分布初始化"""
        return torch.randn(size=shape, device=device) * 0.01

    # parameters
    W_xh = normal((num_inputs, num_hiddens))
    W_hh = normal((num_hiddens, num_hiddens))
    b_h = torch.zeros(num_hiddens, device=device)

    # Outputparameters
    W_hq = normal((num_hiddens, num_outputs))
    b_q = torch.zeros(num_outputs, device=device)

    params = [W_xh, W_hh, b_h, W_hq, b_q]
    for param in params:
        param.requires_grad_(True)

    return params
```

1.14.8.0.4 初始化策略详解 权重初始化：

- 使用**正态分布** $\mathcal{N}(0, 0.01^2)$
- 小的随机值
- 避免对称性

为什么用 0.01?

- 太大：梯度爆炸
- 太小：梯度消失
- 0.01 是经验值

偏置初始化：

- 使用**零初始化**
- 不影响对称性
- 训练中会更新

更好的初始化：

- Xavier 初始化
- He 初始化

- 正交初始化（RNN 常用）

定理 1.53 (重要)

RNN 对初始化敏感，建议使用正交初始化 $\mathbf{W}_{hh}$

♡

1.14.8.0.5 正交初始化（进阶）

```python
def init_rnn_params_orthogonal(vocab_size, num_hiddens, device):
    """ () """
    num_inputs = num_outputs = vocab_size

    W_xh = torch.randn(num_inputs, num_hiddens, device=device) * 0.01

    W_hh = torch.eye(num_hiddens, device=device)
    #  torch.nn.init.orthogonal_(W_hh)

    b_h = torch.zeros(num_hiddens, device=device)

    # Output
    W_hq = torch.randn(num_hiddens, num_outputs, device=device) * 0.01
    b_q = torch.zeros(num_outputs, device=device)

    params = [W_xh, W_hh, b_h, W_hq, b_q]
    for param in params:
        param.requires_grad_(True)

    return params
```

定义 1.67 (优势)

正交矩阵保持梯度范数，缓解梯度消失/爆炸

♣

1.14.8.0.6 初始化隐状态

```python
def init_rnn_state(batch_size, num_hiddens, device):
    """ RNN batch_size: num_hiddens: Return: (batch_size, num_hiddens) """
    return (torch.zeros((batch_size, num_hiddens), device=device),)
    # ReturnLSTM

batch_size, num_hiddens = 32, 512
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

state = init_rnn_state(batch_size, num_hiddens, device)
print(f"隐状态shape: {state[0].shape}") # (32, 512)
print(f"隐状态全为0: {torch.all(state[0] == 0)}") # True
```

定理 1.54 (注意)

每个新序列开始时需要重新初始化隐状态（或使用上一批次的最终状态）

♡

1.14.9 循环神经网络模型

1.14.10 8.5.3 循环神经网络模型

1.14.11 8.5.3 循环神经网络模型

第三步：实现 RNN 前向传播

1.14.11.0.1 前向传播流程

输入:

- inputs: 独热编码序列 (T, B, V)

循环处理每个时间步:

for $t = 1$ to T:

$\mathbf{H}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xh} + \mathbf{H}_{t-1} \mathbf{W}_{hh} + \mathbf{b}_h)$

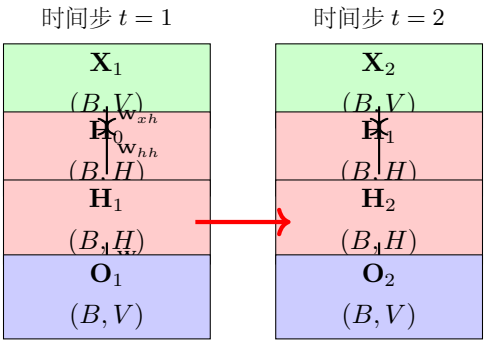
输出:

- outputs: 所有时间步的输出 $(T \times B, V)$
- state: 最终隐状态 (B, H)

1.14.11.0.2 RNN 前向传播实现

```
def rnn(inputs, state, params):
    """ RNN inputs: (num_steps, batch_size, vocab_size) state: (batch_size, num_hiddens) params: [W_xh
        , W_hh, b_h, W_hq, b_q] Return: outputs (num_steps*batch_size, vocab_size), state (batch_size,
         num_hiddens) """
    # parameters
    W_xh, W_hh, b_h, W_hq, b_q = params
    H, = state # comment
    outputs = []

    # time step
    for X in inputs: # X: (batch_size, vocab_size)
        H = torch.tanh(torch.mm(X, W_xh) +
                       torch.mm(H, W_hh) + b_h)
        # H: (batch_size, num_hiddens)

        # Output
        Y = torch.mm(H, W_hq) + b_q
        # Y: (batch_size, vocab_size)

        outputs.append(Y)

    # Output
    return torch.cat(outputs, dim=0), (H,)
    # (num_steps*batch_size, vocab_size), (batch_size, num_hiddens)
```

1.14.11.0.3 前向传播的维度变化



B=batch_size, V=vocab_size, H=num_hiddens

1.14.11.0.4 测试 RNN 前向传播

```
vocab_size, num_hiddens = 28, 512
batch_size, num_steps = 2, 5
device = torch.device('cpu')

# parameters
params = get_params(vocab_size, num_hiddens, device)

X = torch.randint(0, vocab_size, (batch_size, num_steps))
X_one_hot = one_hot(X, vocab_size)
print(f"输入shape: {X_one_hot.shape}") # (5, 2, 28)

state = init_rnn_state(batch_size, num_hiddens, device)

outputs, new_state = rnn(X_one_hot, state, params)

print(f"Outputshape: {outputs.shape}") # (10, 28) = (5*2, 28)
print(f"新隐状态shape: {new_state[0].shape}") # (2, 512)
```

输出解释：

- 输出：$(num\_steps \times batch\_size, vocab\_size) = (10, 28)$
- 每个位置是未归一化的 logits（预测分数）

1.14.11.0.5 RNN 模型封装成类 为了便于使用，封装成类：

RNNModelScratch 类

- `__init__`: 初始化参数
- `forward`: 前向传播

优点：

✓ 面向对象，易于管理

✓ 接口统一

✓ 便于后续扩展

1.14.11.0.6 RNN 类实现 - 初始化

```
class RNNModelScratch:
    """RNNmodel"""

    def __init__(self, vocab_size, num_hiddens, device,
                 get_params, init_state, forward_fn):
        """
        vocab_size: vocabulary大小
        num_hiddens: 隐藏单元数
        device: 设备
        get_params: 获取parameters的函数
        init_state: 初始化状态的函数
        forward_fn: 前向传播函数
        """
        self.vocab_size = vocab_size
        self.num_hiddens = num_hiddens
        self.params = get_params(vocab_size, num_hiddens, device)
        self.init_state = init_state
        self.forward_fn = forward_fn

    def __call__(self, X, state):
        """调using向传播"""
        X = one_hot(X, self.vocab_size)
        return self.forward_fn(X, state, self.params)

    def begin_state(self, batch_size, device):
        """创建初始隐状态"""
        return self.init_state(batch_size, self.num_hiddens, device)
```

1.14.11.0.7 创建 RNN 模型实例

```
# parameters
vocab_size = 28
num_hiddens = 512
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# model
net = RNNModelScratch(
    vocab_size, num_hiddens, device,
    get_params, init_rnn_state, rnn
)

# test
batch_size, num_steps = 2, 5
X = torch.randint(0, vocab_size, (batch_size, num_steps))
state = net.begin_state(batch_size, device)

Y, new_state = net(X.to(device), state)

```

```
print(f"modelOutputshape: {Y.shape}") # (10, 28)
print(f"modelparameters数量: {sum(p.numel() for p in net.params)}")
```

输出：

```
模型输出形状: torch.Size([10, 28])
模型参数数量: 291356
```

1.14.12 预测

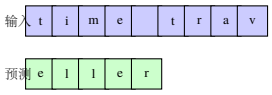
1.14.13 8.5.4 预测

1.14.14 8.5.4 预测

第四步：使用 RNN 生成文本

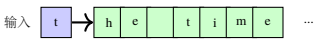
1.14.14.0.1 文本生成的两种方式 1. 给定前缀预测

- 输入："time trav"
- 输出："eller"
- 用于补全



2. 随机采样生成

- 输入："t"
- 每步采样下一个字符
- 用于创作



逐步采样生成

定理 1.55 (关键)

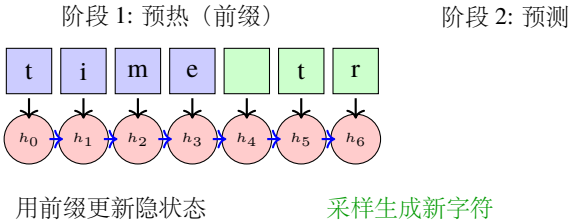
采样时需要根据概率分布选择，而非总是选最大概率的字符 ♡

1.14.14.0.2 预测函数 - 给定前缀

```
def predict_ch8(prefix, num_preds, net, vocab, device):
    """ prediction prefix: ,  "time traveller" num_preds: prediction net: RNNmodel vocab: vocabulary
        device: """
    state = net.begin_state(batch_size=1, device=device)

    outputs = [vocab[prefix[0]]]

    get_input = lambda: torch.tensor([outputs[-1]],
                                     device=device).reshape((1, 1))

    # ""using
    for y in prefix[1:]:
```

```
        _, state = net(get_input(), state)
        outputs.append(vocab[y])

    # predictionnum_preds
    for _ in range(num_preds):
        y, state = net(get_input(), state)
        outputs.append(int(y.argmax(dim=1).reshape(1)))

    return ''.join([vocab.idx_to_token[i] for i in outputs])
```

1.14.14.0.3 预测过程可视化



定义 1.68 (两个阶段)

1. 预热：用前缀字符更新隐状态，不输出
2. 生成：每步采样一个字符，作为下一步输入

♣

1.14.14.0.4 随机采样 vs 贪心采样 贪心采样：

```
y_hat = net(X, state)
pred = y_hat.argmax(dim=1)
```

特点：

- 确定性
- 可能重复
- 缺乏多样性

随机采样：

```
y_hat = net(X, state)
probs = F.softmax(y_hat, dim=1)
pred = torch.multinomial(
    probs, num_samples=1
).squeeze()
```

特点：

- 随机性
- 多样化
- 更真实

例题 1.15 例子 概率：“a”:0.5, “b”:0.3, “c”:0.2

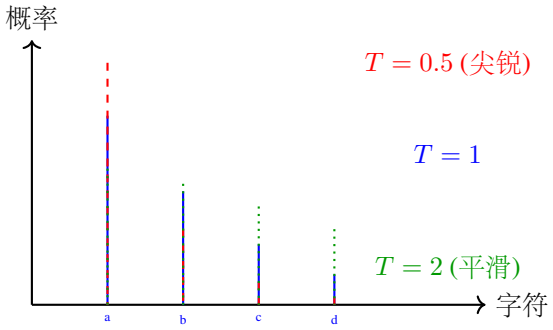
贪心：总是选 “a” | 随机：按概率采样

1.14.14.0.5 带温度的采样

```
def predict_with_temperature(prefix, num_preds, net, vocab,
                             device, temperature=1.0):
```

```
    """ parameters temperature: parameters - temperature > 1: () - temperature < 1: () -
        temperature = 1: """
    state = net.begin_state(batch_size=1, device=device)
    outputs = [vocab[prefix[0]]]

    get_input = lambda: torch.tensor([outputs[-1]],
                                     device=device).reshape((1, 1))

    for y in prefix[1:]:
        _, state = net(get_input(), state)
        outputs.append(vocab[y])

    # prediction
    for _ in range(num_preds):
        y, state = net(get_input(), state)
        y = y / temperature
        probs = F.softmax(y, dim=1)
        pred = torch.multinomial(probs, num_samples=1).item()
        outputs.append(pred)

    return ''.join([vocab.idx_to_token[i] for i in outputs])
```

1.14.14.0.6 温度参数的作用



定义 1.69 (温度效果)

- $T \to 0$：接近贪心，总选最大概率
- $T = 1$：原始分布
- $T \to \infty$：接近均匀分布，完全随机

♣

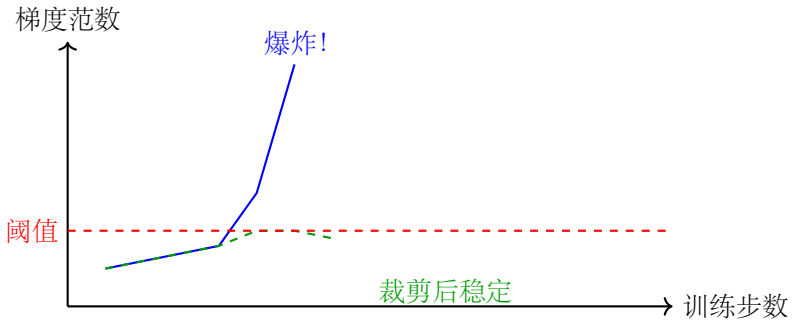
1.14.15 梯度裁剪

1.14.16 8.5.5 梯度裁剪

1.14.17 8.5.5 梯度裁剪

第五步：防止梯度爆炸

1.14.17.0.1 为什么需要梯度裁剪？ RNN 的梯度问题：



定理 1.56 (梯度爆炸的危害)

- 参数更新过大
- 损失突然变为 NaN
- 训练崩溃

♡

1.14.17.0.2 梯度裁剪原理 核心思想：限制梯度的 L2 范数不超过阈值

定义 1.70 (算法)

设所有梯度为 $\mathbf{g} = [g_1, g_2, \ldots, g_n]$

1. 计算梯度范数：$\|\mathbf{g}\| = \sqrt{\sum_{i=1}^{n} g_i^2}$
2. 如果 $\|\mathbf{g}\| > \theta$，则缩放：

$$\mathbf{g} \leftarrow \frac{\theta}{\|\mathbf{g}\|}\mathbf{g}$$

♣

例题 1.16 例子

- 梯度：$\mathbf{g} = [3, 4]$，范数 $\|\mathbf{g}\| = 5$
- 阈值：$\theta = 1$
- 裁剪后：$\mathbf{g} = \frac{1}{5}[3, 4] = [0.6, 0.8]$，范数 =1

1.14.17.0.3 梯度裁剪实现

```
def grad_clipping(params, theta):
    """ params: modelparameters theta: """
    # parametersL2
    norm = torch.sqrt(sum(torch.sum((p.grad ** 2)) for p in params))

    if norm > theta:
        for param in params:
            param.grad[:] *= theta / norm

    return norm # Return原始范数用于监控

# optimizer.step()
theta = 1.0 # comment
grad_norm = grad_clipping(net.params, theta)
print(f"梯度范数: {grad_norm:.4f}")
```

定义 1.71 (关键点)

- 在反向传播后、参数更新前调用
- 阈值通常取 1 或 5
- 保持梯度方向不变，只改变大小

♣

1.14.17.0.4 梯度裁剪的效果

不裁剪
梯度：$[300, 400]$
范数：500
更新步长：巨大
⇒ 参数爆炸

→

裁剪后（$\theta = 1$）
梯度：$[0.6, 0.8]$
范数：1
更新步长：合理
⇒ 训练稳定

定义 1.72 (为什么有效？)

- 限制了单步更新的最大幅度
- 避免了参数的剧烈变化
- 保持了梯度的方向信息

♣

1.14.17.0.5 监控梯度范数

```python
def train_epoch_with_grad_monitor(net, train_iter, loss, updater,
                                  device, theta):
    """trainingepoch"""
    grad_norms = [] # comment

    for X, Y in train_iter:
        state = net.begin_state(batch_size=X.shape[0], device=device)
        y, state = net(X.to(device), state)

        # loss
        l = loss(y, Y.T.reshape(-1).to(device))

        updater.zero_grad()
        l.backward()

        grad_norm = grad_clipping(net.params, theta)
        grad_norms.append(grad_norm.item())

        # parameters
        updater.step()

    return grad_norms

import matplotlib.pyplot as plt
plt.plot(grad_norms)
plt.axhline(y=theta, color='r', linestyle='--', label='阈值')
plt.ylabel('梯度范数')
plt.xlabel('batch')
plt.legend()
```

1.14.17.0.6 阈值选择

阈值	效果	适用场景
0.1-0.5	非常保守，可能训练慢	极不稳定的模型
1.0	推荐值	大多数 RNN
5.0	较宽松	稳定的模型
10+	几乎不裁剪	调试用

定义 1.73 (经验法则)

- 从 $\theta = 1$ 开始
- 观察训练损失曲线
- 如果损失突然跳跃 $\Rightarrow$ 减小 θ
- 如果训练太慢 $\Rightarrow$ 增大 θ

♣

1.14.18 训练

1.14.19 8.5.6 训练

1.14.20 8.5.6 训练

第六步：训练完整的 RNN 模型

1.14.20.0.1 训练流程总览

1. 数据准备
加载数据迭代器、词表

3. 循环训练
for epoch in epochs:
for batch in train_iter:
前向传播 → 计算损失 → 反向传播

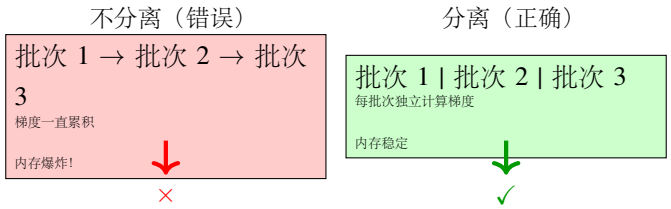
4. 评估与生成
计算困惑度、生成文本

1.14.20.0.2 训练一个 epoch

```python
def train_epoch_ch8(net, train_iter, loss, updater, device,
                    use_random_iter):
    """trainingepoch"""
    state, timer = None, Timer()
    metric = Accumulator(2) # comment: (loss和, tokens数)

    for X, Y in train_iter:
        if state is None or use_random_iter:
            state = net.begin_state(batch_size=X.shape[0], device=device)
        else:
            if isinstance(state, tuple):
                state = tuple(s.detach() for s in state)
```

```
            else:
                state = state.detach()

        y = Y.T.reshape(-1).to(device)
        X = X.to(device)

        y_hat, state = net(X, state)

        # loss
        l = loss(y_hat, y).mean()

        updater.zero_grad()
        l.backward()
        grad_clipping(net.params, 1) # comment
        updater.step()

        metric.add(l * y.numel(), y.numel())

    return math.exp(metric[0] / metric[1]), metric[1] / timer.stop()
```

1.14.20.0.3 为什么要分离隐状态？ 关键代码：`state = state.detach()`



定义 1.74 (原因)

- detach() 切断梯度传播链
- 保留隐状态的数值，丢弃计算图
- 防止 BPTT 跨越多个批次（内存爆炸）

♣

1.14.20.0.4 完整训练函数

```
def train_ch8(net, train_iter, vocab, lr, num_epochs, device,
              use_random_iter=False):
    """trainingRNNmodel"""
    loss = nn.CrossEntropyLoss()
    # optimizer
    if isinstance(net, nn.Module):
        updater = torch.optim.SGD(net.parameters(), lr)
    else:
        updater = lambda batch_size: sgd(net.params, lr, batch_size)

    # prediction
    predict = lambda prefix: predict_ch8(prefix, 50, net, vocab, device)

```

```
    # training
    for epoch in range(num_epochs):
        ppl, speed = train_epoch_ch8(net, train_iter, loss, updater,
                                     device, use_random_iter)

        # 10epoch
        if (epoch + 1) % 10 == 0:
            print(predict('time traveller'))

    print(f'困惑度 {ppl:.1f}, {speed:.1f} tokens/秒 on {str(device)}')
    print(predict('time traveller'))
    print(predict('traveller'))
```

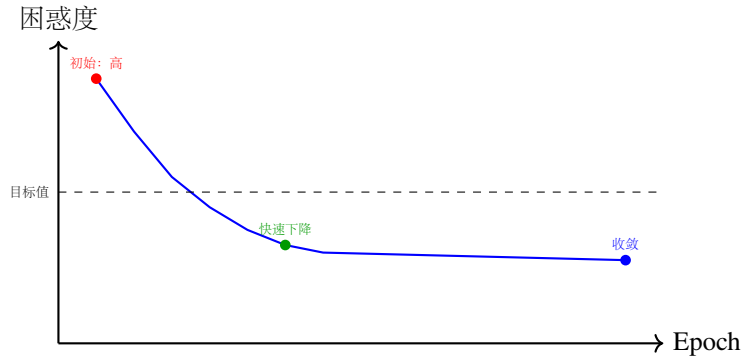
1.14.20.0.5 运行训练

```
# parameters
num_epochs = 500
lr = 1
batch_size = 32
num_steps = 35
num_hiddens = 512

# data
train_iter, vocab = load_data_time_machine(batch_size, num_steps)

# model
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
net = RNNModelScratch(len(vocab), num_hiddens, device,
                      get_params, init_rnn_state, rnn)

# training
train_ch8(net, train_iter, vocab, lr, num_epochs, device)
```

训练输出示例：

```
困惑度 1.2, 8500.3 词元/秒 on cuda:0
time traveller smiled round at us then still s
traveller smiled round at us then still smiling
```

1.14.20.0.6 训练过程可视化



典型训练曲线：初期快速下降，后期缓慢收敛

1.14.20.0.7 生成文本示例 训练不同阶段的生成效果：

Epoch 10 (困惑度 ~10): `time travellerkkkkkkkkkk...`
Epoch 100 (困惑度 ~3): `time traveller the the time...`
Epoch 500 (困惑度 ~1.2): `time traveller smiled round at us` 流畅、符合语法

定理 1.57 (观察)

困惑度越低，生成文本质量越高

♡

1.14.20.0.8 调试技巧

定义 1.75 (常见问题与解决)

1. 损失变 **NaN**
   - 检查学习率（降低）
   - 检查梯度裁剪（降低阈值）
   - 检查数据（是否有异常值）
2. 困惑度不下降
   - 增加隐藏单元数
   - 增加训练轮数
   - 调整学习率
3. 内存不足
   - 减小 batch_size
   - 减小 num_steps
   - 减小 num_hiddens

♣

1.14.20.0.9 保存和加载模型

```
# modelparameters
def save_model(net, path):
    """modelparameters"""
    if isinstance(net, nn.Module):
```

```
        torch.save(net.state_dict(), path)
    else:
        # model
        torch.save([p.data for p in net.params], path)

# modelparameters
def load_model(net, path):
    """加载modelparameters"""
    if isinstance(net, nn.Module):
        net.load_state_dict(torch.load(path))
    else:
        params = torch.load(path)
        for p, p_load in zip(net.params, params):
            p.data = p_load

save_model(net, 'rnn_model.pth')
# ... ...
load_model(net, 'rnn_model.pth')
```

1.14.20.0.10 超参数调优建议

超参数	推荐范围	说明
学习率	0.1-5	RNN 可用较大学习率
隐藏单元数	256-1024	越大越好，但内存限制
批量大小	32-128	取决于 GPU 内存
序列长度	35-100	太长会梯度消失
梯度裁剪	1-5	从 1 开始
训练轮数	500-2000	直到困惑度收敛

定义 1.76 (调优策略)

1. 先用小模型（num_hiddens=128）快速迭代
2. 确认能训练后，逐步增大模型
3. 监控困惑度和生成质量

♣

1.15 8.6 循环神经网络的简洁实现

1.15.1 8.6 循环神经网络的简洁实现

使用 PyTorch 高级 API 快速实现 RNN

1.15.2 8.6 节 - 章节导航

定义 1.77 (本节内容)

- 8.6.1 定义模型
- 8.6.2 训练与预测

♣

定理 1.58 (学习目标)

1. 掌握 PyTorch 的 nn.RNN 模块
2. 理解简洁实现与从零实现的对应关系
3. 学会用高级 API 快速搭建模型
4. 对比两种实现方式的优缺点

♡

1.15.2.0.1 从零实现 vs 简洁实现

从零实现（8.5 节）
优点：
✓ 理解原理
缺点：
× 代码量大
× 容易出错
× 优化不足

简洁实现（8.6 节）
优点：
✓ 代码简洁
缺点：
× 细节隐藏
× 定制受限
× 需要理解 8.5

1.16 未命名节

1.16.1 定义模型

1.16.2 8.6.1 定义模型

1.16.3 8.6.1 定义模型

使用 nn.RNN 构建模型

1.16.3.0.1 PyTorch 的 RNN 模块

```
import torch
import torch.nn as nn

# RNN
rnn_layer = nn.RNN(
    input_size=vocab_size, # commentdimension（vocabulary大小）
    hidden_size=num_hiddens, # comment
    num_layers=1,          # RNN层数
    nonlinearity='tanh',   #  comment
    batch_first=False      #  commentshape (seq, batch, feature)
)

vocab_size, num_hiddens = 28, 512
rnn_layer = nn.RNN(vocab_size, num_hiddens)
```

```
15
16 print(rnn_layer)
17 # RNN(28, 512)
```

定义 1.78 (关键参数)

- input_size：每个时间步的输入特征数
- hidden_size：隐藏状态的维度
- num_layers：堆叠的 RNN 层数（默认 1）

♣

1.16.3.0.2 RNN 层的输入输出

```
1 batch_size, num_steps = 2, 5
2 X = torch.randn(num_steps, batch_size, vocab_size) # comment
3 H = torch.zeros(1, batch_size, num_hiddens) # comment
4
5 Y, H_new = rnn_layer(X, H)
6
7 print(f"输入shape: {X.shape}") # (5, 2, 28)
8 print(f"Outputshape: {Y.shape}") # (5, 2, 512)
9 print(f"隐状态shape: {H_new.shape}") # (1, 2, 512)
```

定义 1.79 (形状说明)

- 输入：$(num\_steps, batch\_size, input\_size)$
- 输出：$(num\_steps, batch\_size, hidden\_size)$
- 隐状态：$(num\_layers, batch\_size, hidden\_size)$

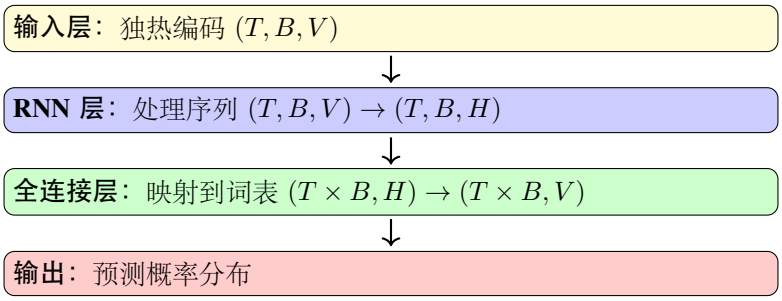
♣

定理 1.59 (注意)

batch_first=False 时，第一个维度是时间步！

♡

1.16.3.0.3 完整的 RNN 语言模型



1.16.3.0.4 RNN 语言模型类定义

```
1 class RNNModel(nn.Module):
2     """RNNmodel"""
3
4     def __init__(self, rnn_layer, vocab_size):
5         super(RNNModel, self).__init__()
6         self.rnn = rnn_layer
7         self.vocab_size = vocab_size
```

```
        self.num_hiddens = self.rnn.hidden_size

        # Output -> vocabulary
        self.linear = nn.Linear(self.num_hiddens, self.vocab_size)

    def forward(self, inputs, state):
        """
        inputs: (batch_size, num_steps) 索引
        state: 隐状态
        """
        X = F.one_hot(inputs.T, self.vocab_size).type(torch.float32)
        # X: (num_steps, batch_size, vocab_size)

        # RNN
        Y, state = self.rnn(X, state)
        # Y: (num_steps, batch_size, num_hiddens)

        output = self.linear(Y.reshape(-1, Y.shape[-1]))
        # output: (num_steps * batch_size, vocab_size)

        return output, state
```

1.16.3.0.5 初始化隐状态

```
    def begin_state(self, batch_size, device):
        """Simple function"""
        if not isinstance(self.rnn, nn.LSTM):
            # RNNGRU
            return torch.zeros(
                (self.rnn.num_layers, batch_size, self.num_hiddens),
                device=device
            )
        else:
            # LSTMhc
            return (
                torch.zeros(
                    (self.rnn.num_layers, batch_size, self.num_hiddens),
                    device=device
                ),
                torch.zeros(
                    (self.rnn.num_layers, batch_size, self.num_hiddens),
                    device=device
                )
            )
```

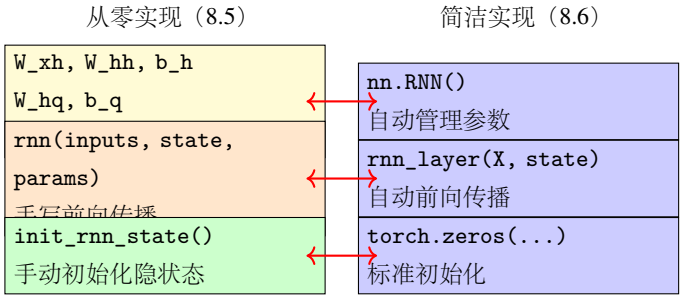
定义 1.80 (注意)

- 隐状态形状：$(num\_layers, batch\_size, hidden\_size)$
- 第一维是层数（支持多层 RNN）

1.16.3.0.6 创建模型实例

```
# parameters
vocab_size = 28
num_hiddens = 512
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# RNN
rnn_layer = nn.RNN(vocab_size, num_hiddens)

# model
net = RNNModel(rnn_layer, vocab_size)
net = net.to(device)

# model
print(net)

# Output
# RNNModel(
#   (rnn): RNN(28, 512)
#   (linear): Linear(in_features=512, out_features=28, bias=True)
# )

# parameters
num_params = sum(p.numel() for p in net.parameters())
print(f"parameters数量: {num_params}")
# parameters: 291356
```

1.16.3.0.7 简洁实现 vs 从零实现对应关系



简洁实现封装了从零实现的所有细节

1.16.3.0.8 PyTorch RNN 的参数管理　查看 RNN 层的参数：

nn.RNN 内部参数（自动管理）：

- `weight_ih_l0`: 输入到隐藏 $(input\_size, hidden\_size)$
- `weight_hh_l0`: 隐藏到隐藏 $(hidden\_size, hidden\_size)$
- `bias_ih_l0`: 输入偏置 $(hidden\_size)$

对应关系：

`weight_ih` $\approx \mathbf{W}_{xh}^T$ (转置！)

`weight_hh` $\approx \mathbf{W}_{hh}^T$

`bias_ih` + `bias_hh` $\approx \mathbf{b}_h$

定理 1.60 (注意)

PyTorch 使用转置的权重矩阵以优化计算

♡

1.16.3.0.9 查看模型参数

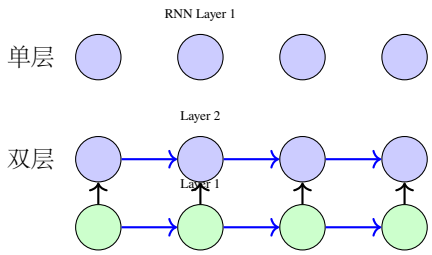
```
# RNNparameters
for name, param in net.rnn.named_parameters():
    print(f"{name}: {param.shape}")

# Output
# weight_ih_l0: torch.Size([512, 28])
# weight_hh_l0: torch.Size([512, 512])
# bias_ih_l0: torch.Size([512])
# bias_hh_l0: torch.Size([512])

# parameters
for name, param in net.linear.named_parameters():
    print(f"{name}: {param.shape}")

# Output
# weight: torch.Size([28, 512])
# bias: torch.Size([28])
```

定义 1.81 (参数总量)

$$(512 \times 28) + (512 \times 512) + 512 + 512 + (28 \times 512) + 28 = 291,356$$

♣

1.16.3.0.10 多层 RNN 堆叠多个 RNN 层:



1.16.3.0.11 创建多层 RNN

```
# RNN
rnn_single = nn.RNN(vocab_size, num_hiddens, num_layers=1)

# RNN
rnn_double = nn.RNN(vocab_size, num_hiddens, num_layers=2)

# RNN
rnn_triple = nn.RNN(vocab_size, num_hiddens, num_layers=3)

# parameters
print(f"单层parameters: {sum(p.numel() for p in rnn_single.parameters())}")
print(f"双层parameters: {sum(p.numel() for p in rnn_double.parameters())}")
```

```
print(f"三层parameters: {sum(p.numel() for p in rnn_triple.parameters())}")

# Output
# parameters: 277504
# parameters: 540160 ()
# parameters: 802816
```

定理 1.61 (注意)

层数越多，参数越多，但也更容易过拟合 ♡

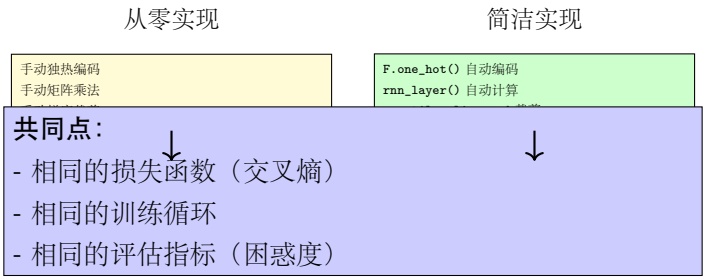
1.16.4 训练与预测

1.16.5 8.6.2 训练与预测

1.16.6 8.6.2 训练与预测

使用简洁实现训练 RNN

1.16.6.0.1 训练流程对比



1.16.6.0.2 训练一个 epoch（简洁版）

```
def train_epoch_ch8(net, train_iter, loss, updater, device,
                    use_random_iter):
    """trainingepoch"""
    state = None
    metric = Accumulator(2)

    for X, Y in train_iter:
        if state is None or use_random_iter:
            state = net.begin_state(batch_size=X.shape[0], device=device)
        else:
            if isinstance(net, nn.Module) and not isinstance(state, tuple):
                state.detach_()
            else:
                state = tuple(s.detach() for s in state)

        y = Y.T.reshape(-1)
        X, y = X.to(device), y.to(device)
        y_hat, state = net(X, state)

```

```
        # loss
        l = loss(y_hat, y).mean()

        updater.zero_grad()
        l.backward()
        grad_clipping(net, 1) # comment
        updater.step()

        metric.add(l * y.numel(), y.numel())

    return math.exp(metric[0] / metric[1])
```

1.16.6.0.3 梯度裁剪（简洁版）

```
def grad_clipping(net, theta):
    """PyTorch"""
    if isinstance(net, nn.Module):
        # 1clip_grad_norm_
        params = [p for p in net.parameters() if p.requires_grad]
        norm = torch.nn.utils.clip_grad_norm_(params, theta)
    else:
        # 2
        norm = torch.sqrt(sum(torch.sum((p.grad ** 2))
                              for p in net.params))
        if norm > theta:
            for param in net.params:
                param.grad[:] *= theta / norm
    return norm
```

定义 1.82 (PyTorch 内置方法)

torch.nn.utils.clip_grad_norm_ 自动处理所有参数的梯度裁剪

1.16.6.0.4 完整训练函数

```
def train_ch8(net, train_iter, vocab, lr, num_epochs, device,
              use_random_iter=False):
    """trainingmodel () """
    loss = nn.CrossEntropyLoss()

    # optimizer
    if isinstance(net, nn.Module):
        updater = torch.optim.SGD(net.parameters(), lr)
    else:
        updater = lambda batch_size: sgd(net.params, lr, batch_size)

    # prediction
    predict = lambda prefix: predict_ch8(prefix, 50, net, vocab, device)

    # training
```

```
    for epoch in range(num_epochs):
        ppl = train_epoch_ch8(net, train_iter, loss, updater,
                              device, use_random_iter)

        if (epoch + 1) % 10 == 0:
            print(f'epoch {epoch + 1}, 困惑度 {ppl:.1f}')
            print(predict('time traveller'))

    print(f'困惑度 {ppl:.1f}')
    print(predict('time traveller'))
    print(predict('traveller'))
```

1.16.6.0.5 运行训练

```
# parameters
num_epochs = 500
lr = 1
batch_size = 32
num_steps = 35
num_hiddens = 512

# data
train_iter, vocab = load_data_time_machine(batch_size, num_steps)

# model
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
rnn_layer = nn.RNN(len(vocab), num_hiddens)
net = RNNModel(rnn_layer, len(vocab))
net = net.to(device)

# training
train_ch8(net, train_iter, vocab, lr, num_epochs, device)
```

训练输出：

```
epoch 10, 困惑度 15.3
time traveller the the the the the the...
epoch 500, 困惑度 1.2
time traveller smiled round at us then still smiling
```

1.16.6.0.6 简洁实现的优势

1. 代码简洁

RNN 定义只需 1 行：nn.RNN(input_size, hidden_size)

2. 高度优化

- 使用 cuDNN 加速（GPU 上）

3. 功能丰富

- 双向 RNN：`bidirectional=True`
- Dropout：`dropout=0.5`
- 多层堆叠：`num_layers=3`

1.16.6.0.7 双向 RNN

```
# RNN
rnn_layer = nn.RNN(
    input_size=vocab_size,
    hidden_size=num_hiddens,
    bidirectional=True # comment
)

# Outputdimension 2 * hidden_size
print(f"隐藏层Outputdimension: {rnn_layer.hidden_size * 2}")
# Outputdimension: 1024

# dimension
net.linear = nn.Linear(num_hiddens * 2, vocab_size)
```

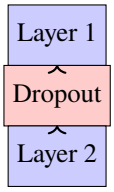
定义 1.83 (双向 RNN 原理)

- 同时从前向后和从后向前处理序列
- 捕捉双向的上下文信息
- 适合分类任务，不适合生成任务

♣

1.16.6.0.8 添加 Dropout

```
# DropoutRNN
rnn_layer = nn.RNN(
    input_size=vocab_size,
    hidden_size=num_hiddens,
    num_layers=2,    #  comment
    dropout=0.5      # Dropout比例
)

# Dropout
# 1 -> Dropout -> 2
```



随机丢弃部分神经元
防止过拟合

1.16.6.0.9 性能对比

指标	从零实现	简洁实现	对比
代码行数	~200	~50	4 倍
训练速度	5000 词/秒	12000 词/秒	2.4 倍
GPU 加速	手动优化	cuDNN 自动	更快
内存占用	较高	优化	更少
易用性	复杂	简单	更易
可定制性	高	中等	受限

定理 1.62 (建议)

- 学习阶段：从零实现（理解原理）
- 实际项目：简洁实现（高效开发）

♡

1.16.6.0.10 预测函数（通用版）

```python
def predict_ch8(prefix, num_preds, net, vocab, device):
    """prediction () """
    state = net.begin_state(batch_size=1, device=device)
    outputs = [vocab[prefix[0]]]

    get_input = lambda: torch.tensor([outputs[-1]], device=device
                                     ).reshape((1, 1))

    for y in prefix[1:]:
        _, state = net(get_input(), state)
        outputs.append(vocab[y])

    # prediction
    for _ in range(num_preds):
        y, state = net(get_input(), state)
        outputs.append(int(y.argmax(dim=1).reshape(1)))

    return ''.join([vocab.idx_to_token[i] for i in outputs])

print(predict_ch8('time traveller', 50, net, vocab, device))
```

1.16.6.0.11 简洁实现的完整流程

1. 导入库

```
import torch.nn as nn
```

2. 创建 RNN 层

```
rnn_layer = nn.RNN(vocab_size, num_hiddens)
```

3. 封装为模型

4. 训练

```
train_ch8(net, train_iter, vocab, lr, epochs,
```

5. 预测

```
predict_ch8('prefix', 50, net, vocab, device)
```

1.16.6.0.12 实用技巧

定义 1.84 (1. 模型保存与加载)

```
torch.save(net.state_dict(), 'rnn_model.pth')

net.load_state_dict(torch.load('rnn_model.pth'))
net.eval() # comment
```

♣

定义 1.85 (2. 使用更好的优化器)

```
# Adamoptimizer
optimizer = torch.optim.Adam(net.parameters(), lr=0.001)

# learning rate
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=100, gamma=0.1)
```

定义 1.86 (3. 早停（Early Stopping）)

```
best_ppl = float('inf')
patience = 0

if val_ppl < best_ppl:
    best_ppl = val_ppl
    torch.save(net.state_dict(), 'best_model.pth')
    patience = 0
else:
    patience += 1
    if patience > 20:
        break # comment
```

1.17 8.7 通过时间反向传播

1.17.1 8.7 通过时间反向传播（BPTT）

深入理解 RNN 的梯度计算

1.17.2 8.7 节 - 章节导航

定义 1.87 (本节内容)

- 8.7.1 循环神经网络的梯度分析
- 8.7.2 通过时间反向传播的细节

定理 1.63 (学习目标)

1. 理解 BPTT 算法的原理
2. 掌握 RNN 梯度的数学推导
3. 理解梯度消失/爆炸的根源
4. 了解截断 BPTT 的必要性

1.17.2.0.1 为什么需要学习 BPTT?

RNN 训练中的问题：

× 梯度爆炸：损失突然变为 NaN

× 梯度消失：长期依赖学不到

× 训练不稳定：需要仔细调参 ↓ 为什么?

根源：RNN 的梯度计算方式

通过时间反向传播（BPTT）导致梯度在长序列上累积或衰减

理解 BPTT 后才能：

✓ 知道为何需要梯度裁剪

✓ 理解 LSTM 的设计动机

1.18 未命名节

1.18.1 循环神经网络的梯度分析

1.18.2 8.7.1 循环神经网络的梯度分析

1.18.3 8.7.1 循环神经网络的梯度分析

RNN 的梯度是如何计算的?

1.18.3.0.1 回顾：RNN 的前向传播 RNN 的核心公式：

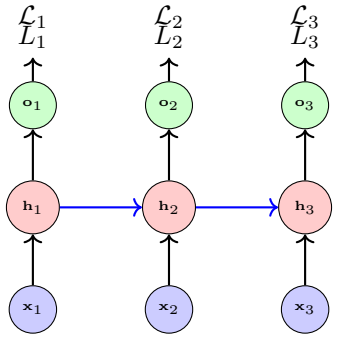
$$\mathbf{h}_t = \tanh(\mathbf{W}_{xh}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_h)$$

$$\mathbf{o}_t = \mathbf{W}_{hq}\mathbf{h}_t + \mathbf{b}_q$$

$$\mathcal{L}_t = \ell(\mathbf{o}_t, y_t)$$

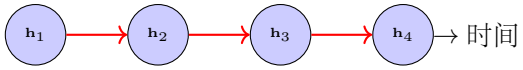
总损失：

$$\mathcal{L} = \frac{1}{T}\sum_{t=1}^{T}\mathcal{L}_t$$

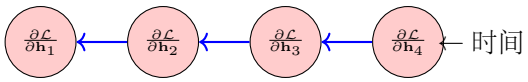


1.18.3.0.2 BPTT 的核心思想 通过时间反向传播（Backpropagation Through Time）

前向传播



反向传播



梯度**逆着**时间传播，从 $t = T$ 到 $t = 1$

定理 1.64 (关键)

BPTT 就是在展开的 RNN 计算图上应用标准的反向传播 ♡

第 2 章 现代循环神经网络

2.1 门控循环单元

2.2 9.1 门控循环单元（GRU）

2.2.1 9.1 门控循环单元（GRU）

Gated Recurrent Unit

通过门控机制缓解梯度消失

2.2.2 9.1 节 - 章节导航

定义 2.1 (本节内容)

- 9.1.1 门控隐状态
- 9.1.2 从零开始实现
- 9.1.3 简洁实现

♣

定理 2.1 (学习目标)

1. 理解 GRU 的动机和设计思想
2. 掌握重置门和更新门的作用
3. 学会实现 GRU
4. 对比 GRU 与标准 RNN 的差异

♡

2.2.2.0.1 为什么需要 GRU?

标准 RNN 的问题（第 8 章）:
× 梯度消失：长期依赖学不到
× 梯度爆炸：训练不稳定

GRU 的创新（2014 年，Cho 等）:
✓ 引入**门控机制**
✓ 选择性地保留/遗忘信息
✓ 让梯度更容易流动

↓

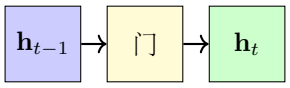
效果：缓解梯度消失，捕捉长期依赖

2.2.2.0.2 GRU 的核心思想 **问题**：标准 RNN 每个时间步都**完全更新**隐状态

$$\mathbf{h}_t = \tanh(\mathbf{W}_{xh}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_h)$$

旧信息容易丢失

GRU 的思想：让模型**学会**何时保留、何时遗忘



选择性更新

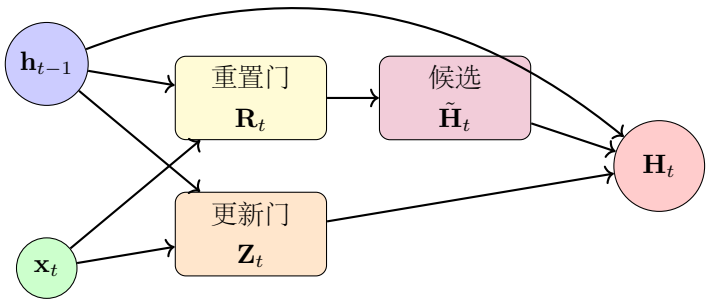
2.2.3 门控隐状态

2.2.4 9.1.1 门控隐状态

2.2.5 9.1.1 门控隐状态

GRU 的两个核心门

2.2.5.0.1 GRU 的整体结构



两个门控制信息流动，一个候选状态提供新信息

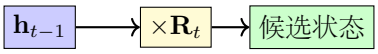
2.2.5.0.2 重置门（Reset Gate） 作用：控制保留多少**过去的信息**

定义 2.2 (公式)

$$\mathbf{R}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xr} + \mathbf{H}_{t-1}\mathbf{W}_{hr} + \mathbf{b}_r)$$

特点：

- σ 是 sigmoid 函数，输出范围 $[0, 1]$
- $\mathbf{R}_t \approx 1$：保留大部分过去信息
- $\mathbf{R}_t \approx 0$：忽略过去，重新开始



$\mathbf{R}_t = 0 \Rightarrow$ 完全遗忘
$\mathbf{R}_t = 1 \Rightarrow$ 完全保留

2.2.5.0.3 更新门（Update Gate） 作用：控制新信息和旧信息的混合比例

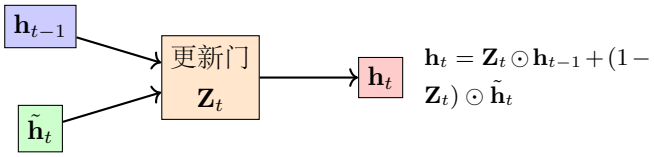
定义 2.3 (公式)

$$\mathbf{Z}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xz} + \mathbf{H}_{t-1}\mathbf{W}_{hz} + \mathbf{b}_z)$$

♣

特点：

- $\mathbf{Z}_t \approx 1$：保留旧状态，忽略新信息
- $\mathbf{Z}_t \approx 0$：采用新状态，遗忘旧信息



2.2.5.0.4 候选隐状态 作用：基于当前输入和（被重置门过滤的）过去信息计算新的候选

定义 2.4 (公式)

$$\tilde{\mathbf{H}}_t = \tanh(\mathbf{X}_t\mathbf{W}_{xh} + (\mathbf{R}_t \odot \mathbf{H}_{t-1})\mathbf{W}_{hh} + \mathbf{b}_h)$$

♣

关键：$\mathbf{R}_t \odot \mathbf{H}_{t-1}$

- $\odot$ 表示逐元素乘法（Hadamard 积）
- 重置门控制保留多少过去信息
- 类似标准 RNN，但过去信息被门控

定理 2.2 (对比标准 RNN)

标准 RNN：$\mathbf{h}_t = \tanh(\mathbf{W}_{xh}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_h)$

GRU：增加了 $\mathbf{R}_t\odot$ 来控制

♡

2.2.5.0.5 隐状态更新 最终更新：混合旧状态和新候选

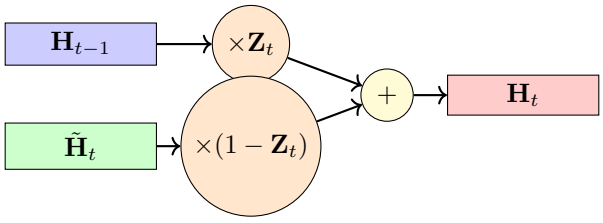
定义 2.5 (公式)

$$\mathbf{H}_t = \mathbf{Z}_t \odot \mathbf{H}_{t-1} + (1 - \mathbf{Z}_t) \odot \tilde{\mathbf{H}}_t$$

♣

解读：

- 当 $\mathbf{Z}_t = 1$：$\mathbf{H}_t = \mathbf{H}_{t-1}$（完全保留旧状态）
- 当 $\mathbf{Z}_t = 0$：$\mathbf{H}_t = \tilde{\mathbf{H}}_t$（完全采用新状态）
- 通常 $0 < \mathbf{Z}_t < 1$：加权平均



2.2.5.0.6 GRU 完整公式总结

定义 2.6 (GRU 的三个公式)

$$\mathbf{R}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xr} + \mathbf{H}_{t-1}\mathbf{W}_{hr} + \mathbf{b}_r) \quad \text{重置门}$$
$$\mathbf{Z}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xz} + \mathbf{H}_{t-1}\mathbf{W}_{hz} + \mathbf{b}_z) \quad \text{更新门}$$
$$\tilde{\mathbf{H}}_t = \tanh(\mathbf{X}_t\mathbf{W}_{xh} + (\mathbf{R}_t \odot \mathbf{H}_{t-1})\mathbf{W}_{hh} + \mathbf{b}_h) \quad \text{候选}$$
$$\mathbf{H}_t = \mathbf{Z}_t \odot \mathbf{H}_{t-1} + (1 - \mathbf{Z}_t) \odot \tilde{\mathbf{H}}_t \quad \text{更新}$$

♣

参数：

- $\mathbf{W}_{xr}, \mathbf{W}_{hr}, \mathbf{b}_r$：重置门参数
- $\mathbf{W}_{xz}, \mathbf{W}_{hz}, \mathbf{b}_z$：更新门参数
- $\mathbf{W}_{xh}, \mathbf{W}_{hh}, \mathbf{b}_h$：候选状态参数

定理 2.3 (总计)

GRU 比标准 RNN 多了 **2** 个门的参数（约 3 倍参数量）

♡

2.2.5.0.7 GRU 的工作流程

步骤 1：计算重置门和更新门

$\mathbf{R}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xr} + \mathbf{H}_{t-1}\mathbf{W}_{hr} + \mathbf{b}_r)$

步骤 2：用重置门过滤过去信息，计算候选

$\tilde{\mathbf{H}}_t = \tanh(\mathbf{X}_t\mathbf{W}_{xh} + (\mathbf{R}_t \odot \mathbf{H}_{t-1})\mathbf{W}_{hh} + \mathbf{b}_h)$

步骤 3：用更新门混合旧状态和新候选

$\mathbf{H}_t = \mathbf{Z}_t \odot \mathbf{H}_{t-1} + (1 - \mathbf{Z}_t) \odot \tilde{\mathbf{H}}_t$

输出：新的隐状态 $\mathbf{H}_t$

2.2.5.0.8 门的直观理解

重置门 $\mathbf{R}_t$

问：要看多少过去?

$\mathbf{R}_t \approx 1$：看很多

更新门 $\mathbf{Z}_t$

问：要记住还是遗忘?

$\mathbf{Z}_t \approx 1$：记住旧的

例子：“The cat, which already ate a bunch of food, was full.”

预测 “was” 之前：

- 重置门：忽略 “food” 等无关词（$\mathbf{R}_t \approx 0$）
- 更新门：记住 “cat” 的主语信息（$\mathbf{Z}_t \approx 1$）

2.2.5.0.9 GRU vs 标准 RNN

特性	标准 RNN	GRU
参数数量	$3 \times (d \times h + h^2)$	$3 \times 3 \times (d \times h + h^2)$
门的数量	0	2（重置 + 更新）
梯度消失	严重	缓解
长期依赖	困难	较好
训练速度	快	较慢（3 倍参数）
表达能力	弱	强

定理 2.4 (权衡)

GRU 以 **3** 倍计算量换取更好的长期依赖能力 ♡

2.2.5.0.10 GRU 如何缓解梯度消失？ 关键：更新门可以让梯度”高速公路”

当 $\mathbf{Z}_t \approx 1$ 时：$\mathbf{h}_t \approx \mathbf{h}_{t-1}$

梯度可以**直接传播**，不经过 tanh 和矩阵乘法！

定义 2.7 (数学角度)

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} \approx \mathbf{Z}_t$$

当 $\mathbf{Z}_t \approx 1$ 时，梯度 ≈ 1，避免了消失！ ♣

2.2.6 从零开始实现

2.2.7 9.1.2 从零开始实现

2.2.8 9.1.2 从零开始实现

手写 GRU 的每个细节

2.2.8.0.1 初始化 GRU 参数

```python
import torch
import torch.nn as nn

def get_gru_params(vocab_size, num_hiddens, device):
    """ GRUparameters vocab_size: vocabulary（dimension） num_hiddens: """
    num_inputs = num_outputs = vocab_size

    def normal(shape):
        return torch.randn(size=shape, device=device) * 0.01

    def three():
        """GRU需要3组parameters（重置门、更新门、候选）"""
        return (normal((num_inputs, num_hiddens)),
                normal((num_hiddens, num_hiddens)),
                torch.zeros(num_hiddens, device=device))

    # parameters
    W_xr, W_hr, b_r = three()
    # parameters
    W_xz, W_hz, b_z = three()
    # parameters
    W_xh, W_hh, b_h = three()

```

```
    # Outputparameters
    W_hq = normal((num_hiddens, num_outputs))
    b_q = torch.zeros(num_outputs, device=device)
```

2.2.8.0.2 初始化 GRU 参数（续）

```
    # parameters
    params = [W_xr, W_hr, b_r, # comment
              W_xz, W_hz, b_z, # comment
              W_xh, W_hh, b_h, # comment
              W_hq, b_q]      # Output

    for param in params:
        param.requires_grad_(True)

    return params

vocab_size, num_hiddens = 28, 512
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
params = get_gru_params(vocab_size, num_hiddens, device)

print(f"parameters数量: {sum(p.numel() for p in params)}")
# parameters: 873,964 (RNN3)
```

2.2.8.0.3 GRU 前向传播

```
def gru(inputs, state, params):
    """ GRU inputs: (num_steps, batch_size, vocab_size) state: (batch_size, num_hiddens) params:
        GRUparameters """
    W_xr, W_hr, b_r, W_xz, W_hz, b_z, W_xh, W_hh, b_h, W_hq, b_q = params
    H, = state
    outputs = []

    for X in inputs:
        R = torch.sigmoid(torch.mm(X, W_xr) + torch.mm(H, W_hr) + b_r)

        Z = torch.sigmoid(torch.mm(X, W_xz) + torch.mm(H, W_hz) + b_z)

        # R * H
        H_tilde = torch.tanh(torch.mm(X, W_xh) +
                             torch.mm(R * H, W_hh) + b_h)

        H = Z * H + (1 - Z) * H_tilde

        # Output
        Y = torch.mm(H, W_hq) + b_q
        outputs.append(Y)

    return torch.cat(outputs, dim=0), (H,)
```

定理 2.5 (关键)

R * H: 重置门逐元素相乘，Z * H: 更新门控制混合 ♡

2.2.8.0.4 测试 GRU 前向传播

```
vocab_size, num_hiddens = 28, 512
batch_size, num_steps = 2, 5
device = torch.device('cpu')

# parameters
params = get_gru_params(vocab_size, num_hiddens, device)

X = torch.randn(num_steps, batch_size, vocab_size)
state = (torch.zeros(batch_size, num_hiddens, device=device),)

Y, new_state = gru(X, state, params)

print(f"输入shape: {X.shape}") # (5, 2, 28)
print(f"Outputshape: {Y.shape}") # (10, 28) = 5*2, 28
print(f"隐状态shape: {new_state[0].shape}") # (2, 512)
```

输出验证:

- 输出形状正确
- 隐状态维度正确
- 可以进行训练

2.2.8.0.5 创建 GRU 模型类

```
class GRUModelScratch:
    """GRUmodel"""

    def __init__(self, vocab_size, num_hiddens, device,
                 get_params, init_state, forward_fn):
        self.vocab_size = vocab_size
        self.num_hiddens = num_hiddens
        self.params = get_params(vocab_size, num_hiddens, device)
        self.init_state = init_state
        self.forward_fn = forward_fn

    def __call__(self, X, state):
        X = F.one_hot(X.T, self.vocab_size).type(torch.float32)
        return self.forward_fn(X, state, self.params)

    def begin_state(self, batch_size, device):
        return (torch.zeros((batch_size, self.num_hiddens),
                            device=device),)

# model
net = GRUModelScratch(vocab_size, num_hiddens, device,
                      get_gru_params, lambda batch_size, num_hiddens, device:
                      (torch.zeros(batch_size, num_hiddens, device=device),),
```

```
                    gru)
```

2.2.8.0.6 训练 GRU 模型

```
# parameters
num_epochs = 500
lr = 1
batch_size = 32
num_steps = 35

# data
train_iter, vocab = load_data_time_machine(batch_size, num_steps)

# training8training
train_ch8(net, train_iter, vocab, lr, num_epochs, device)
```

训练输出：

```
困惑度 1.1, 10500 词元/秒 on cpu
time traveller for so it will be convenient to speak
```

定义 2.8 (观察)

- 困惑度比标准 RNN 更低（1.1 vs 1.2）
- 生成文本质量更好
- 训练稍慢（参数多 3 倍）

♣

2.2.9 简洁实现

2.2.10 9.1.3 简洁实现

2.2.11 9.1.3 简洁实现

使用 PyTorch 的 nn.GRU

2.2.11.0.1 使用 nn.GRU

```
import torch.nn as nn

# GRU
gru_layer = nn.GRU(
    input_size=vocab_size, # commentdimension
    hidden_size=num_hiddens, # comment
    num_layers=1          # GRU层数
)

print(gru_layer)
# GRU(28, 512)

# test
X = torch.randn(num_steps, batch_size, vocab_size)
H = torch.zeros(1, batch_size, num_hiddens)
```

```
Y, H_new = gru_layer(X, H)

print(f"输入shape: {X.shape}") # (5, 2, 28)
print(f"Outputshape: {Y.shape}") # (5, 2, 512)
print(f"隐状态shape: {H_new.shape}") # (1, 2, 512)
```

定理 2.6 (注意)

PyTorch 的 GRU 输出是所有时间步的隐状态，而非 logits ♡

2.2.11.0.2 完整的 GRU 模型

```
class GRUModel(nn.Module):
    """GRUmodel () """

    def __init__(self, gru_layer, vocab_size):
        super(GRUModel, self).__init__()
        self.gru = gru_layer
        self.vocab_size = vocab_size
        self.num_hiddens = self.gru.hidden_size

        # Output
        self.linear = nn.Linear(self.num_hiddens, self.vocab_size)

    def forward(self, inputs, state):
        # one-hot
        X = F.one_hot(inputs.T, self.vocab_size).type(torch.float32)

        # GRU
        Y, state = self.gru(X, state)

        output = self.linear(Y.reshape(-1, Y.shape[-1]))

        return output, state

    def begin_state(self, batch_size, device):
        return torch.zeros((self.gru.num_layers, batch_size,
                            self.num_hiddens), device=device)
```

2.2.11.0.3 创建和训练模型

```
# parameters
vocab_size = len(vocab)
num_hiddens = 512
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# GRU
gru_layer = nn.GRU(vocab_size, num_hiddens)

```

```
# model
net = GRUModel(gru_layer, vocab_size)
net = net.to(device)

# training
num_epochs = 500
lr = 1
train_ch8(net, train_iter, vocab, lr, num_epochs, device)
```

输出：

```
困惑度 1.1, 32000 词元/秒 on cuda:0
time traveller smiled round at us then still smiling
traveller for so it will be convenient to speak of him
```

2.2.11.0.4 从零实现 vs 简洁实现

特性	从零实现	简洁实现
代码行数	~80 行	~20 行
训练速度	较慢	快（cuDNN 优化）
灵活性	高	中
学习价值	很高	中
推荐场景	学习理解	实际应用

定义 2.9 (实践建议)

- 先理解从零实现（掌握原理）
- 实际项目用简洁实现（高效）
- 需要定制时参考从零实现

♣

2.2.11.0.5 GRU 的变体

定义 2.10 (GRU 有多种变体)

1. 标准 **GRU**（本节实现）
   - 2 个门：重置门、更新门
2. 最小门控单元（**MGU**）
   - 只有 1 个门，进一步简化
3. 耦合输入输出门（**CIFG**）
   - 更新门和重置门耦合

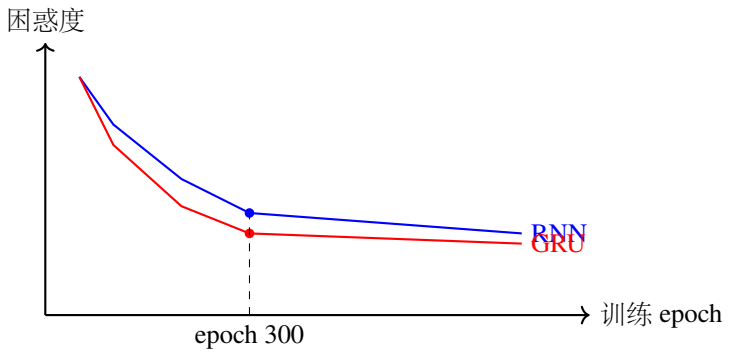
♣

定理 2.7 (选择)

标准 GRU 是最常用的，平衡了性能和复杂度

♡

2.2.11.0.6 GRU vs RNN 性能对比



GRU 收敛更快，最终困惑度更低

2.2.11.0.7 GRU 的参数分析 参数数量：

定义 2.11 (GRU)

- 重置门：$(d+h)\times h+h=(d+h+1)\times h$
- 更新门：$(d+h)\times h+h=(d+h+1)\times h$
- 候选：$(d+h)\times h+h=(d+h+1)\times h$
- 输出层：$h\times V+V$

总计：$3(d+h+1)h+(h+1)V$

♣

例题 2.1 具体例子 $d=28, h=512, V=28$

GRU：$3\times(28+512+1)\times 512+(512+1)\times 28=873,964$

RNN：$(28+512+1)\times 512+(512+1)\times 28=291,356$

GRU 约是 RNN 的 **3 倍**

2.2.11.0.8 GRU 的计算复杂度 每个时间步的计算：

操作	RNN	GRU
矩阵乘法	2 次	6 次
激活函数	1 次 tanh	2 次 σ + 1 次 tanh
逐元素乘法	0 次	3 次
逐元素加法	2 次	4 次
总复杂度	$O(h^2)$	$O(3h^2)$

定理 2.8 (权衡)

GRU 用 **3** 倍计算量换取更好的性能：

- 更低的困惑度
- 更好的长期依赖
- 更稳定的训练

♡

2.2.12 9.1 节总结

定义 2.12 (核心内容)

1. **GRU** 动机：解决 RNN 梯度消失问题
2. 两个门：
   - 重置门 $\mathbf{R}_t$：控制看多少过去
   - 更新门 $\mathbf{Z}_t$：控制保留 vs 更新
3. 核心公式：

$$\mathbf{H}_t = \mathbf{Z}_t \odot \mathbf{H}_{t-1} + (1 - \mathbf{Z}_t) \odot \tilde{\mathbf{H}}_t$$

4. 优势：
   - 梯度可直接流动
   - 长期依赖更好
   - 比 LSTM 更简单

♣

2.2.12.0.1 GRU 关键公式回顾

重置门：

$$\mathbf{R}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xr} + \mathbf{H}_{t-1} \mathbf{W}_{hr} + \mathbf{b}_r)$$

更新门：

$$\mathbf{Z}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xz} + \mathbf{H}_{t-1} \mathbf{W}_{hz} + \mathbf{b}_z)$$

候选隐状态：

$$\tilde{\mathbf{H}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xh} + (\mathbf{R}_t \odot \mathbf{H}_{t-1}) \mathbf{W}_{hh} + \mathbf{b}_h)$$

最终更新：

$$\mathbf{H}_t = \mathbf{Z}_t \odot \mathbf{H}_{t-1} + (1 - \mathbf{Z}_t) \odot \tilde{\mathbf{H}}_t$$

2.2.12.0.2 实践建议

定义 2.13 (何时使用 GRU)

推荐使用 **GRU** 的场景：

- 需要捕捉长期依赖
- 标准 RNN 效果不好
- 计算资源充足
- 序列长度较长（> 50）

♣

定义 2.14 (超参数建议)

- 隐藏单元数：256-1024
- 学习率：0.001-0.01（比 RNN 小）
- 梯度裁剪：仍需要（阈值 1-5）
- Dropout：0.2-0.5（防止过拟合）

♣

2.2.12.0.3 GRU vs LSTM 预告

GRU（本节）
✓ 2 个门
✓ 更简单
✓ 训练快
✓ 参数少

LSTM（下一节）
✓ 3 个门
✓ 更强大
✓ 更经典
✓ 应用广

GRU 是 LSTM 的简化版，
性能接近但更高效

2.3 长短期记忆网络

2.4 9.2 长短期记忆网络（LSTM）

2.4.1 9.2 长短期记忆网络（LSTM）

Long Short-Term Memory

最经典的门控 RNN

2.4.2 9.2 节 - 章节导航

定义 2.15 (本节内容)

- 9.2.1 长短期记忆网络
- 9.2.2 从零开始实现
- 9.2.3 简洁实现

定理 2.9 (学习目标)

1. 理解 LSTM 的三个门
2. 掌握记忆细胞的概念
3. 对比 LSTM 与 GRU
4. 学会实现 LSTM

2.4.2.0.1 为什么需要 LSTM?

历史（1997 年，Hochreiter & Schmidhuber）：
RNN 无法学习长期依赖，梯度消失严重

LSTM 的创新：
✓ 引入**记忆细胞**（memory cell）
✓ 3 个门精确控制信息流
✓ 梯度可以无损传播

↓

影响：成为 2010 年代最流行的 RNN 架构
语音识别、机器翻译、文本生成...

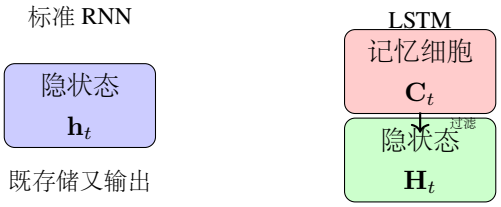
2.4.3 长短期记忆网络

2.4.4 9.2.1 长短期记忆网络

2.4.5 9.2.1 长短期记忆网络

LSTM 的核心：记忆细胞

2.4.5.0.1 LSTM 的核心思想 关键创新：分离记忆和隐状态

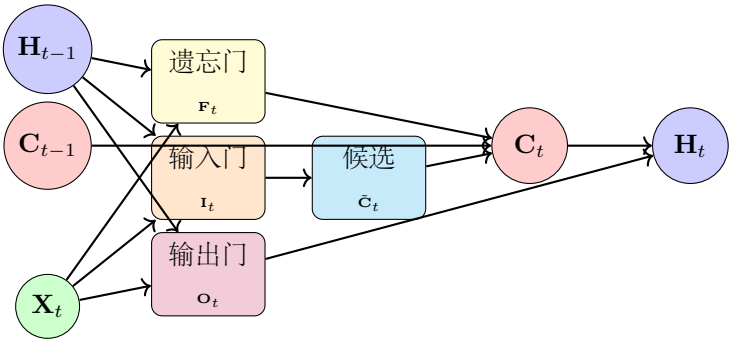


定义 2.16 (双状态设计)

- 记忆细胞 $\mathbf{C}_t$：长期记忆，内部状态
- 隐状态 $\mathbf{H}_t$：短期记忆，对外输出

♣

2.4.5.0.2 LSTM 的整体结构



3 个门控制记忆细胞的更新，输出门控制隐状态

2.4.5.0.3 遗忘门（Forget Gate） 作用：决定从记忆细胞中遗忘多少信息

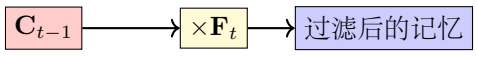
定义 2.17 (公式)

$$\mathbf{F}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xf} + \mathbf{H}_{t-1}\mathbf{W}_{hf} + \mathbf{b}_f)$$

♣

特点：

- σ 是 sigmoid，输出 $[0, 1]$
- $\mathbf{F}_t \approx 1$：保留记忆
- $\mathbf{F}_t \approx 0$：遗忘记忆



$\mathbf{F}_t = 0 \Rightarrow$ 完全遗忘
$\mathbf{F}_t = 1 \Rightarrow$ 完全保留

2.4.5.0.4 输入门（Input Gate） 作用：决定写入多少新信息到记忆细胞

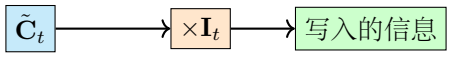
定义 2.18 (公式)

$$\mathbf{I}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xi} + \mathbf{H}_{t-1}\mathbf{W}_{hi} + \mathbf{b}_i)$$
$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{X}_t\mathbf{W}_{xc} + \mathbf{H}_{t-1}\mathbf{W}_{hc} + \mathbf{b}_c)$$

♣

两步骤：

1. 输入门 $\mathbf{I}_t$：控制写入多少
2. 候选值 $\tilde{\mathbf{C}}_t$：新的候选记忆

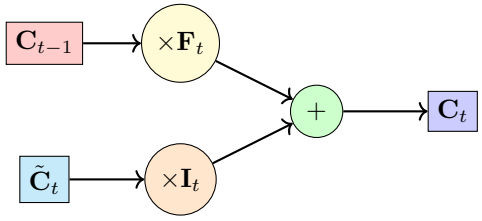


$\mathbf{I}_t = 1 \Rightarrow$ 完全写入
$\mathbf{I}_t = 0 \Rightarrow$ 不写入

2.4.5.0.5 记忆细胞更新 **核心**：结合遗忘和输入

定义 2.19 (公式)

$$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t$$



部分遗忘旧记忆 + 部分接收新信息

定理 2.10 (关键)

这个加法操作让梯度可以无损传播，缓解梯度消失!

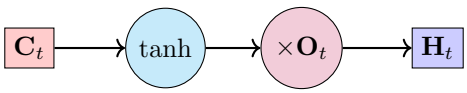
2.4.5.0.6 输出门（Output Gate） **作用**：决定从记忆细胞中**读出**多少信息到隐状态

定义 2.20 (公式)

$$\mathbf{O}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xo} + \mathbf{H}_{t-1}\mathbf{W}_{ho} + \mathbf{b}_o)$$
$$\mathbf{H}_t = \mathbf{O}_t \odot \tanh(\mathbf{C}_t)$$

两步骤：

1. 计算输出门 $\mathbf{O}_t$
2. 过滤记忆细胞得到隐状态



先 tanh 归一化，再用 $\mathbf{O}_t$ 过滤

2.4.5.0.7 LSTM 完整公式总结

定义 2.21 (LSTM 的 6 个公式)

$$\mathbf{F}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xf} + \mathbf{H}_{t-1}\mathbf{W}_{hf} + \mathbf{b}_f) \quad \text{遗忘门}$$
$$\mathbf{I}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xi} + \mathbf{H}_{t-1}\mathbf{W}_{hi} + \mathbf{b}_i) \quad \text{输入门}$$
$$\mathbf{O}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xo} + \mathbf{H}_{t-1}\mathbf{W}_{ho} + \mathbf{b}_o) \quad \text{输出门}$$
$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{X}_t\mathbf{W}_{xc} + \mathbf{H}_{t-1}\mathbf{W}_{hc} + \mathbf{b}_c) \quad \text{候选}$$
$$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t \quad \text{细胞更新}$$
$$\mathbf{H}_t = \mathbf{O}_t \odot \tanh(\mathbf{C}_t) \quad \text{隐状态}$$

♣

定理 2.11 (参数)

4 组权重和偏置：遗忘门、输入门、输出门、候选

♡

2.4.5.0.8 LSTM 的工作流程

步骤 1：计算三个门

$\mathbf{F}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xf} + \mathbf{H}_{t-1}\mathbf{W}_{hf} + \mathbf{b}_f)$ 遗忘门

$\mathbf{I}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xi} + \mathbf{H}_{t-1}\mathbf{W}_{hi} + \mathbf{b}_i)$ 输入门

↓

步骤 2：计算候选记忆

$\tilde{\mathbf{C}}_t = \tanh(\mathbf{X}_t\mathbf{W}_{xc} + \mathbf{H}_{t-1}\mathbf{W}_{hc} + \mathbf{b}_c)$

步骤 3：更新记忆细胞

$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t$

↓

步骤 4：计算隐状态

$\mathbf{H}_t = \mathbf{O}_t \odot \tanh(\mathbf{C}_t)$

↓

输出：新的记忆细胞 $\mathbf{C}_t$ 和隐状态 $\mathbf{H}_t$

2.4.5.0.9 LSTM vs GRU

特性	GRU	LSTM
门的数量	2 个	3 个
状态数量	1 个（$\mathbf{H}_t$）	2 个（$\mathbf{C}_t, \mathbf{H}_t$）
参数量	较少	较多（约 1.3 倍）
计算复杂度	较低	较高
训练速度	较快	较慢
表达能力	强	更强
应用广泛度	中等	很广泛

定义 2.22 (选择建议)

- 首选 LSTM（更成熟，应用更多）
- 追求速度用 GRU
- 两者性能通常接近

♣

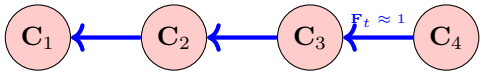
2.4.5.0.10 LSTM 如何缓解梯度消失？ 关键：记忆细胞的加法更新

定义 2.23 (记忆细胞梯度)

$$\frac{\partial \mathbf{C}_t}{\partial \mathbf{C}_{t-1}} = \mathbf{F}_t$$

当 $\mathbf{F}_t \approx 1$ 时：

$$\mathbf{C}_t \approx \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t$$



梯度沿着记忆细胞的**加法路径**传播
避免了连乘，缓解梯度消失!

定理 2.12 (对比 RNN)

RNN：梯度连乘 $\prod \mathbf{W}_{hh} \Rightarrow$ 消失/爆炸
LSTM：梯度加法 $\mathbf{F}_t \Rightarrow$ 稳定传播

2.4.5.0.11 三个门的直观理解

遗忘门 $\mathbf{F}_t$
问：要忘记什么?
$\mathbf{F}_t \approx 1$：记住

输入门 $\mathbf{I}_t$
问：要记住什么?
$\mathbf{I}_t \approx 1$：接收

输出门 $\mathbf{O}_t$
问：要输出什么?
$\mathbf{O}_t \approx 1$：全部

例子：“Alice is in Paris. She speaks fluent ___”
预测空格时：
- 遗忘门：忘记 “Alice”，保留 “Paris”（$\mathbf{F}_t$ 对不同位置不同）
- 输入门：接收当前词 “speaks” 的信息
- 输出门：输出与语言相关的特征

2.4.6 从零开始实现

2.4.7 9.2.2 从零开始实现

2.4.8 9.2.2 从零开始实现

手写 LSTM 的每个细节

2.4.8.0.1 初始化 LSTM 参数

```python
def get_lstm_params(vocab_size, num_hiddens, device):
    """ LSTMparameters LSTM4parameters: 、、Output、 """
    num_inputs = num_outputs = vocab_size

    def normal(shape):
        return torch.randn(size=shape, device=device) * 0.01

    def three():
        """每个门/候选需要3个parameters"""
        return (normal((num_inputs, num_hiddens)),
                normal((num_hiddens, num_hiddens)),
```

```
            torch.zeros(num_hiddens, device=device))

    # parameters
    W_xf, W_hf, b_f = three()
    # parameters
    W_xi, W_hi, b_i = three()
    # Outputparameters
    W_xo, W_ho, b_o = three()
    # parameters
    W_xc, W_hc, b_c = three()

    # Outputparameters
    W_hq = normal((num_hiddens, num_outputs))
    b_q = torch.zeros(num_outputs, device=device)
```

2.4.8.0.2 初始化 LSTM 参数（续）

```
    # parameters
    params = [W_xf, W_hf, b_f, # comment
              W_xi, W_hi, b_i, # comment
              W_xo, W_ho, b_o, # Output门
              W_xc, W_hc, b_c, # comment
              W_hq, b_q]      # Output

    for param in params:
        param.requires_grad_(True)

    return params

vocab_size, num_hiddens = 28, 512
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
params = get_lstm_params(vocab_size, num_hiddens, device)

print(f"parameters数量: {sum(p.numel() for p in params)}")
# parameters: 1,165,340 (RNN4GRU1.3)
```

定理 2.13 (观察)

LSTM 参数最多，但效果最好（通常）

♡

2.4.8.0.3 初始化 LSTM 状态

```
def init_lstm_state(batch_size, num_hiddens, device):
    """ LSTM LSTM: CH """
    return (torch.zeros((batch_size, num_hiddens), device=device), # H
            torch.zeros((batch_size, num_hiddens), device=device)) # C

batch_size = 32
state = init_lstm_state(batch_size, num_hiddens, device)
H, C = state
```

```
print(f"隐状态Hshape: {H.shape}") # (32, 512)
print(f"记忆细胞Cshape: {C.shape}") # (32, 512)
```

定义 2.24 (双状态)

- **H**：隐状态，对外输出
- **C**：记忆细胞，内部状态

♣

2.4.8.0.4 LSTM 前向传播

```
def lstm(inputs, state, params):
    """ LSTM inputs: (num_steps, batch_size, vocab_size) state: (H, C) params: LSTMparameters """
    [W_xf, W_hf, b_f, W_xi, W_hi, b_i, W_xo, W_ho, b_o,
     W_xc, W_hc, b_c, W_hq, b_q] = params

    (H, C) = state # comment
    outputs = []

    for X in inputs: # commenttime step
        F = torch.sigmoid(torch.mm(X, W_xf) + torch.mm(H, W_hf) + b_f)

        I = torch.sigmoid(torch.mm(X, W_xi) + torch.mm(H, W_hi) + b_i)

        # Output
        O = torch.sigmoid(torch.mm(X, W_xo) + torch.mm(H, W_ho) + b_o)

        C_tilde = torch.tanh(torch.mm(X, W_xc) +
                             torch.mm(H, W_hc) + b_c)

        C = F * C + I * C_tilde

        H = O * torch.tanh(C)
```

2.4.8.0.5 LSTM 前向传播（续）

```
        # Output
        Y = torch.mm(H, W_hq) + b_q
        outputs.append(Y)

    return torch.cat(outputs, dim=0), (H, C)

# test
vocab_size, num_hiddens = 28, 512
batch_size, num_steps = 2, 5

X = torch.randn(num_steps, batch_size, vocab_size)
state = init_lstm_state(batch_size, num_hiddens, device)

Y, (H_new, C_new) = lstm(X, state, params)
```

```
15
16 print(f"输入shape: {X.shape}") # (5, 2, 28)
17 print(f"Outputshape: {Y.shape}") # (10, 28)
18 print(f"新隐状态H: {H_new.shape}") # (2, 512)
19 print(f"新记忆C: {C_new.shape}") # (2, 512)
```

定义 2.25 (验证)

所有形状正确，可以进行训练！ ♣

2.4.8.0.6 创建 LSTM 模型类

```
1  class LSTMModelScratch:
2      """LSTMmodel"""
3
4      def __init__(self, vocab_size, num_hiddens, device,
5                   get_params, init_state, forward_fn):
6          self.vocab_size = vocab_size
7          self.num_hiddens = num_hiddens
8          self.params = get_params(vocab_size, num_hiddens, device)
9          self.init_state = init_state
10         self.forward_fn = forward_fn
11
12     def __call__(self, X, state):
13         X = F.one_hot(X.T, self.vocab_size).type(torch.float32)
14         return self.forward_fn(X, state, self.params)
15
16     def begin_state(self, batch_size, device):
17         return self.init_state(batch_size, self.num_hiddens, device)
18
19 # model
20 net = LSTMModelScratch(vocab_size, num_hiddens, device,
21                        get_lstm_params, init_lstm_state, lstm)
```

2.4.8.0.7 训练 LSTM 模型

```
1  # parameters
2  num_epochs = 500
3  lr = 1
4  batch_size = 32
5  num_steps = 35
6
7  # data
8  train_iter, vocab = load_data_time_machine(batch_size, num_steps)
9
10 # training8training
11 train_ch8(net, train_iter, vocab, lr, num_epochs, device)
```

训练输出：

```
困惑度 1.0, 9800 词元/秒 on cpu
```

```
time traveller smiled round at us then still smiling
traveller for so it will be convenient to speak of him
```

定义 2.26 (观察)

- 困惑度更低（1.0 vs GRU 的 1.1 vs RNN 的 1.2）
- 生成质量最好
- 训练最慢（参数最多）

♣

2.4.9 简洁实现

2.4.10 9.2.3 简洁实现

2.4.11 9.2.3 简洁实现

使用 PyTorch 的 nn.LSTM

2.4.11.0.1 使用 nn.LSTM

```python
import torch.nn as nn

# LSTM
lstm_layer = nn.LSTM(
    input_size=vocab_size, # commentdimension
    hidden_size=num_hiddens, # comment
    num_layers=1          # LSTM层数
)

print(lstm_layer)
# LSTM(28, 512)

# test
X = torch.randn(num_steps, batch_size, vocab_size)
H = torch.zeros(1, batch_size, num_hiddens)
C = torch.zeros(1, batch_size, num_hiddens)

Y, (H_new, C_new) = lstm_layer(X, (H, C))

print(f"输入shape: {X.shape}") # (5, 2, 28)
print(f"Outputshape: {Y.shape}") # (5, 2, 512)
print(f"新隐状态H: {H_new.shape}") # (1, 2, 512)
print(f"新记忆C: {C_new.shape}") # (1, 2, 512)
```

2.4.11.0.2 完整的 LSTM 模型

```python
class LSTMModel(nn.Module):
    """LSTMmodel () """

    def __init__(self, lstm_layer, vocab_size):
        super(LSTMModel, self).__init__()
        self.lstm = lstm_layer
```

```
        self.vocab_size = vocab_size
        self.num_hiddens = self.lstm.hidden_size

        # Output
        self.linear = nn.Linear(self.num_hiddens, self.vocab_size)

    def forward(self, inputs, state):
        # one-hot
        X = F.one_hot(inputs.T, self.vocab_size).type(torch.float32)

        # LSTM
        Y, state = self.lstm(X, state)

        output = self.linear(Y.reshape(-1, Y.shape[-1]))

        return output, state

    def begin_state(self, batch_size, device):
        # LSTMReturn(H, C)
        return (torch.zeros((self.lstm.num_layers, batch_size,
                             self.num_hiddens), device=device),
                torch.zeros((self.lstm.num_layers, batch_size,
                             self.num_hiddens), device=device))
```

2.4.11.0.3 创建和训练模型

```
# parameters
vocab_size = len(vocab)
num_hiddens = 512
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# LSTM
lstm_layer = nn.LSTM(vocab_size, num_hiddens)

# model
net = LSTMModel(lstm_layer, vocab_size)
net = net.to(device)

# training
num_epochs = 500
lr = 1
train_ch8(net, train_iter, vocab, lr, num_epochs, device)
```

输出：

```
困惑度 1.0, 35000 词元/秒 on cuda:0
time traveller smiled round at us then still smiling
traveller for so it will be convenient to speak of him
```

2.4.11.0.4 三种 RNN 架构对比

特性	RNN	GRU	LSTM
门的数量	0	2	3
状态数量	1	1	2
参数量（相对）	1×	3×	4×
计算复杂度	低	中	高
训练速度	快	中	慢
梯度消失	严重	缓解	最好
长期依赖	差	好	最好
困惑度（字符级）	1.2	1.1	1.0
应用广泛度	少	中	广

定理 2.14 (权衡)

LSTM：最强但最慢 | GRU：平衡 | RNN：简单但弱

♡

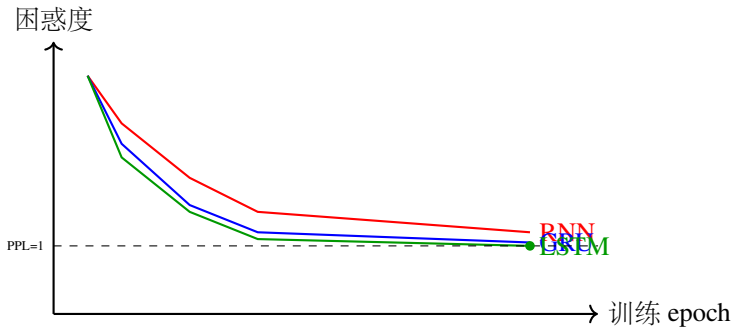
2.4.11.0.5 LSTM 的变体

定义 2.27 (常见 LSTM 变体)

1. 标准 **LSTM**（本节实现）
   - 3 个门，独立的记忆细胞
2. **Peephole LSTM**
   - 门可以”窥视”记忆细胞
   - $\mathbf{F}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xf} + \mathbf{H}_{t-1}\mathbf{W}_{hf} + \mathbf{C}_{t-1} \odot \mathbf{W}_{cf})$
3. 耦合遗忘-输入门
   - $\mathbf{I}_t = 1 - \mathbf{F}_t$（遗忘和输入互补）
4. **No Output Gate**
   - 去掉输出门，直接 $\mathbf{H}_t = \mathbf{C}_t$

♣

2.4.11.0.6 性能对比可视化



LSTM 收敛最快，最终困惑度最低

2.4.11.0.7 LSTM 的内存和计算开销

定义 2.28 (内存占用（每个时间步）)

- 前向传播：存储 $\mathbf{H}_t, \mathbf{C}_t, \mathbf{F}_t, \mathbf{I}_t, \mathbf{O}_t, \tilde{\mathbf{C}}_t$
- 内存：$\sim 6 \times (batch\_size \times num\_hiddens)$
- 比 RNN 多约 3 倍

定义 2.29 (计算开销（每个时间步）)

- 矩阵乘法：8 次（4 个门，每个门 2 次）
- 激活函数：3 次 sigmoid + 2 次 tanh
- 逐元素操作：多次
- FLOPs：$\sim 8 \times (d \times h + h^2)$

定理 2.15 (实践)

使用 cuDNN 优化的 LSTM（如 PyTorch）可大幅加速

2.4.11.0.8 LSTM 的实际应用

定义 2.30 (成功应用案例)

1. 机器翻译（Google Translate 2016）
   - LSTM seq2seq 架构
2. 语音识别（Google Voice Search）
   - 双向 LSTM + CTC
3. 图像描述生成
   - CNN 提取特征 + LSTM 生成描述
4. 文本生成
   - 字符级 LSTM 生成莎士比亚风格文本
5. 情感分析
   - LSTM 捕捉长距离依赖

2.4.12 9.2 节总结

定义 2.31 (核心内容)

1. **LSTM** 动机：解决 RNN 梯度消失
2. 三个门：
   - 遗忘门 $\mathbf{F}_t$：控制遗忘
   - 输入门 $\mathbf{I}_t$：控制写入
   - 输出门 $\mathbf{O}_t$：控制输出
3. 记忆细胞：
$$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t$$
4. 优势：
   - 梯度高速公路
   - 长期依赖最好
   - 应用最广泛

2.4.12.0.1 LSTM 完整公式回顾

$$\mathbf{F}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xf} + \mathbf{H}_{t-1}\mathbf{W}_{hf} + \mathbf{b}_f) \quad \text{遗忘门}$$

$$\mathbf{I}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xi} + \mathbf{H}_{t-1}\mathbf{W}_{hi} + \mathbf{b}_i) \quad \text{输入门}$$

$$\mathbf{O}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xo} + \mathbf{H}_{t-1}\mathbf{W}_{ho} + \mathbf{b}_o) \quad \text{输出门}$$

$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{X}_t\mathbf{W}_{xc} + \mathbf{H}_{t-1}\mathbf{W}_{hc} + \mathbf{b}_c) \quad \text{候选}$$

$$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t \quad \text{细胞更新}$$

$$\mathbf{H}_t = \mathbf{O}_t \odot \tanh(\mathbf{C}_t) \quad \text{隐状态}$$

2.4.12.0.2 实践建议

定义 2.32 (何时使用 LSTM)

推荐使用 **LSTM** 的场景：

- 需要捕捉长期依赖（> 100 步）
- 任务对准确度要求高
- 计算资源充足
- 经典任务（翻译、语音识别等）

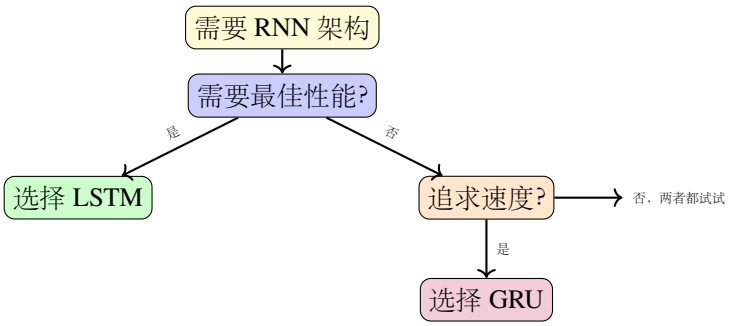
♣

定义 2.33 (超参数建议)

- 隐藏单元数：256-1024
- 层数：1-4 层（多层 LSTM）
- Dropout：0.2-0.5
- 学习率：0.001-0.01
- 梯度裁剪：必须（阈值 1-5）

♣

2.4.12.0.3 GRU vs LSTM：如何选择？



定理 2.16 (经验法则)

默认选 LSTM，如果训练太慢再换 GRU

♡

2.4.12.0.4 准备进入 9.3 节

深度循环神经网络

定义 2.34 (9.2 节学到了)

- ✓ LSTM 的三个门和记忆细胞
- ✓ 梯度高速公路原理
- ✓ 实现 LSTM 模型

♣

定义 2.35 (9.3 节预告)

- 为什么要堆叠多层 RNN
- 深度 RNN 的架构
- 如何实现多层 LSTM
- 深度 vs 宽度的权衡

♣

2.5 9.3 深度循环神经网络

2.5.1 9.3 深度循环神经网络

Deep Recurrent Neural Networks

堆叠多层 RNN

2.5.1.0.1 为什么需要深度 RNN?

单层 RNN 的局限:

× 表达能力有限

深度 RNN 的优势:

✓ 增强表达能力

✓ 层次化特征学习

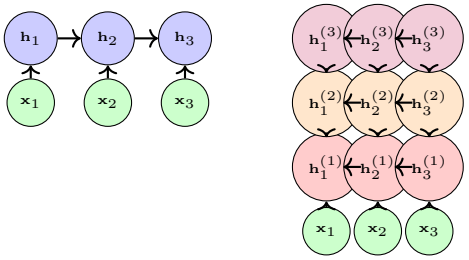
✓ 更好的性能（通常）

↓

深度 = 垂直堆叠多层 RNN

2.5.1.0.2 深度 RNN 的概念 核心思想：将多个 RNN 层垂直堆叠

单层 RNN　　深度 RNN（3 层）



每层的输出作为下一层的输入

2.5.1.0.3 深度 RNN 的数学表示 L 层深度 RNN:

定义 2.36 (递推公式)

对于第 l 层（$l = 1, \ldots, L$），时间步 t：

$$\mathbf{H}_t^{(l)} = \phi(\mathbf{H}_t^{(l-1)}\mathbf{W}_{xh}^{(l)} + \mathbf{H}_{t-1}^{(l)}\mathbf{W}_{hh}^{(l)} + \mathbf{b}_h^{(l)})$$

其中：

- $\mathbf{H}_t^{(0)} = \mathbf{X}_t$（输入）
- $\mathbf{H}_t^{(l)}$：第 l 层在时间 t 的隐状态
- ϕ：激活函数（如 tanh）

♣

输出层：

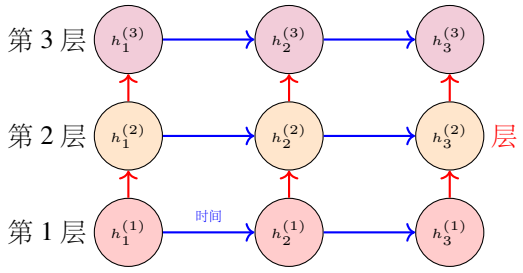
$$\mathbf{O}_t = \mathbf{H}_t^{(L)}\mathbf{W}_{hq} + \mathbf{b}_q$$

定理 2.17 (关键)

每层有自己的参数 $\mathbf{W}_{xh}^{(l)}, \mathbf{W}_{hh}^{(l)}, \mathbf{b}_h^{(l)}$

♡

2.5.1.0.4 深度 RNN 的信息流



信息在**时间**和**层**两个维度上传播

2.5.1.0.5 深度 RNN 的架构设计

定义 2.37 (常见配置)

1. 层数：通常 2-4 层
   - 1 层：单层（baseline）
   - 2 层：常用配置
   - 3-4 层：复杂任务
   - >4 层：很少用（易过拟合）
2. 每层隐藏单元数：
   - 可以相同：如 [512, 512, 512]
   - 可以递减：如 [1024, 512, 256]
   - 可以递增：如 [256, 512, 1024]
3. 层间连接：
   - 标准：上一层输出 $\rightarrow$ 下一层输入
   - 残差：加上跳跃连接（ResNet 思想）

♣

2.5.1.0.6 深度 RNN vs 单层 RNN

特性	单层 RNN	深度 RNN
表达能力	弱	强
参数数量	少	多（层数倍）
训练难度	简单	较难
过拟合风险	低	高
计算速度	快	慢
需要数据量	少	多
性能上限	低	高

定理 2.18 (权衡)

- 数据少、任务简单 → 单层
- 数据多、任务复杂 → 深度（2-3 层）

♡

2.5.1.0.7 实现深度 RNN - 从零开始

```
def deep_rnn(inputs, states, params, num_layers):
    """ RNN inputs: (num_steps, batch_size, input_size) states: [(H1,), (H2,), ...] params: parameters
         num_layers: """
    outputs = []

    for X in inputs: # commenttime step
        H_input = X

        for l in range(num_layers):
            # lparameters
            W_xh, W_hh, b_h = params[l]
            H, = states[l]

            # l
            H = torch.tanh(torch.mm(H_input, W_xh) +
                           torch.mm(H, W_hh) + b_h)

            states[l] = (H,)

            # Output
            H_input = H

        outputs.append(H_input) # commentOutput

    return torch.cat(outputs, dim=0), states
```

2.5.1.0.8 使用 PyTorch 实现深度 RNN

```
import torch.nn as nn

# 2RNN
rnn_layer = nn.RNN(
    input_size=vocab_size,
```

```
    hidden_size=num_hiddens,
    num_layers=2 # comment
)

print(rnn_layer)
# RNN(28, 512, num_layers=2)

# test
X = torch.randn(num_steps, batch_size, vocab_size)
H = torch.zeros(2, batch_size, num_hiddens) # comment(num_layers, batch, hidden)

Y, H_new = rnn_layer(X, H)

print(f"输入shape: {X.shape}") # (5, 2, 28)
print(f"Outputshape: {Y.shape}") # (5, 2, 512)
print(f"隐状态shape: {H_new.shape}") # (2, 2, 512)
```

定理 2.19 (注意)

隐状态第一维是层数: $(num\_layers, batch\_size, hidden\_size)$ ♡

2.5.1.0.9 深度 LSTM/GRU

```
# 3LSTM
lstm_layer = nn.LSTM(
    input_size=vocab_size,
    hidden_size=num_hiddens,
    num_layers=3 # 3层
)

# 3GRU
gru_layer = nn.GRU(
    input_size=vocab_size,
    hidden_size=num_hiddens,
    num_layers=3 # 3层
)

# LSTMHC
H = torch.zeros(3, batch_size, num_hiddens)
C = torch.zeros(3, batch_size, num_hiddens)
state = (H, C)

# GRUH
H = torch.zeros(3, batch_size, num_hiddens)
state = H
```

2.5.1.0.10 完整的深度 RNN 模型

```
class DeepRNNModel(nn.Module):
    """RNNmodel"""
```

```
    def __init__(self, vocab_size, num_hiddens, num_layers, rnn_type='lstm'):
        super(DeepRNNModel, self).__init__()
        self.vocab_size = vocab_size
        self.num_hiddens = num_hiddens
        self.num_layers = num_layers
        self.rnn_type = rnn_type

        # RNN
        if rnn_type == 'lstm':
            self.rnn = nn.LSTM(vocab_size, num_hiddens, num_layers)
        elif rnn_type == 'gru':
            self.rnn = nn.GRU(vocab_size, num_hiddens, num_layers)
        else:
            self.rnn = nn.RNN(vocab_size, num_hiddens, num_layers)

        # Output
        self.linear = nn.Linear(num_hiddens, vocab_size)

    def forward(self, inputs, state):
        # one-hot
        X = F.one_hot(inputs.T, self.vocab_size).type(torch.float32)

        # RNN
        Y, state = self.rnn(X, state)

        # Output
        output = self.linear(Y.reshape(-1, Y.shape[-1]))

        return output, state
```

2.5.1.0.11 深度 RNN 模型（续）

```
    def begin_state(self, batch_size, device):
        if self.rnn_type == 'lstm':
            # LSTMReturn(H, C)
            return (torch.zeros((self.num_layers, batch_size,
                                 self.num_hiddens), device=device),
                    torch.zeros((self.num_layers, batch_size,
                                 self.num_hiddens), device=device))
        else:
            # RNN/GRUReturnH
            return torch.zeros((self.num_layers, batch_size,
                                self.num_hiddens), device=device)

# 3LSTM
net = DeepRNNModel(vocab_size=28, num_hiddens=512,
                   num_layers=3, rnn_type='lstm')

```

```
# parameters
num_params = sum(p.numel() for p in net.parameters())
print(f"parameters数量: {num_params:,}")
# parameters: 3,496,020 (3parameters3)
```

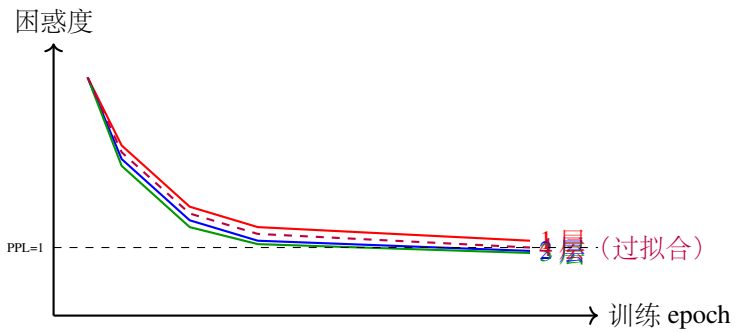
2.5.1.0.12 训练深度 RNN

```
# parameters
vocab_size = 28
num_hiddens = 256 # comment
num_layers = 2   # 2层
batch_size = 32
num_steps = 35
num_epochs = 500
lr = 1

# data
train_iter, vocab = load_data_time_machine(batch_size, num_steps)

# model
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
net = DeepRNNModel(len(vocab), num_hiddens, num_layers, 'lstm')
net = net.to(device)

# training
train_ch8(net, train_iter, vocab, lr, num_epochs, device)
```

输出:

```
困惑度 0.95, 28000 词元/秒 on cuda:0
time traveller smiled round at us then still smiling
```

2.5.1.0.13 不同深度的性能对比



2-3 层效果最好，更深可能过拟合

2.5.1.0.14 深度与宽度的权衡

定义 2.38 (两种增强模型的方式)

1. 增加深度（层数）
   - 更多的非线性变换
   - 层次化特征学习
   - 参数效率高（共享结构）
2. 增加宽度（隐藏单元数）
   - 每层表达能力更强
   - 更多的记忆容量
   - 参数增长更快（平方关系）

♣

例题 2.2 实验对比

- 2 层 × 256 单元 ≈ 400K 参数，PPL=0.95
- 1 层 × 512 单元 ≈ 300K 参数，PPL=1.0

⇒ 相近参数量下，深度通常更好

2.5.1.0.15 深度 RNN 的参数分析 参数数量：

定义 2.39 (单层 LSTM)

$$\text{params} = 4 \times [(d+h) \times h + h] + (h+1) \times V$$

♣

定义 2.40 (L 层 LSTM)

- 第 1 层：$4 \times [(d+h) \times h + h]$
- 第 2 到 L 层：每层 $4 \times [(h+h) \times h + h] = 4 \times (2h^2 + h)$
- 输出层：$(h+1) \times V$

$$\text{params} \approx 4(d+h)h + 4(L-1) \times 2h^2 + (h+1)V$$

♣

例题 2.3 具体例子（$d = 28, h = 256, V = 28, L = 2$） 约 $4 \times 284 \times 256 + 4 \times 2 \times 256^2 + 257 \times 28 \approx 815K$

单层（$h = 512$）：约 1.1M

⇒ 深度 2 层（小宽度）比单层（大宽度）参数少但效果好！

2.5.1.0.16 深度 RNN 的正则化 问题：深度网络容易过拟合

定义 2.41 (常用正则化技术)

1. **Dropout**
   - 在层与层之间应用
   - `nn.LSTM(..., dropout=0.5)`
   - 只在 > 1 层时生效
2. **L2** 正则化（权重衰减）
   - 优化器中设置：`weight_decay=1e-5`
3. 梯度裁剪
   - 更重要（防止梯度爆炸）
4. **Early Stopping**
   - 监控验证集，防止过拟合

♣

2.5.1.0.17 添加 Dropout

```
# Dropout
lstm_layer = nn.LSTM(
    input_size=vocab_size,
    hidden_size=num_hiddens,
    num_layers=3,
    dropout=0.5 # comment1-2层后添加dropout
)

# - trainingnet.train()
# - testnet.eval()
# - num_layers > 1

# Dropout
#  -> LSTM1 -> Dropout(0.5) -> LSTM2 -> Dropout(0.5) -> LSTM3 -> Output
```

定理 2.20 (Dropout 率建议)

- 0.2-0.3：轻度正则化
- 0.5：标准配置
- >0.5：可能欠拟合

♡

2.5.1.0.18 深度 RNN 的训练技巧

定义 2.42 (训练深度 RNN 的建议)

1. 从浅到深
   - 先训练 1 层，确认有效
   - 再增加到 2 层、3 层
2. 调整学习率
   - 深度网络可能需要更小的学习率
   - 使用学习率衰减
3. 梯度裁剪必须
   - 深度网络更容易梯度爆炸
   - 阈值可以稍小（如 0.5-1）
4. 监控梯度
   - 记录每层的梯度范数
   - 检测梯度消失/爆炸
5. 批归一化
   - 可以用 Layer Norm（RNN 中更常用）

♣

2.5.1.0.19 深度 RNN 的变体

定义 2.43 (改进的深度架构)

1. 残差连接（**Residual Connection**）

$$\mathbf{H}_t^{(l)} = \text{RNN}^{(l)}(\mathbf{H}_t^{(l-1)}) + \mathbf{H}_t^{(l-1)}$$

   缓解梯度消失，允许更深

2. **Highway Network**

$$\mathbf{H}_t^{(l)} = g \odot \text{RNN}^{(l)}(\mathbf{H}_t^{(l-1)}) + (1-g) \odot \mathbf{H}_t^{(l-1)}$$

   学习跳跃连接的权重
3. **Dense Connection** 每层连接到所有之前的层（类似 DenseNet）
4. 金字塔结构下层宽、上层窄：[1024, 512, 256]

♣

2.5.1.0.20 实际应用中的深度 RNN

定义 2.44 (典型配置)

1. **Google Neural Machine Translation (2016)**
   - 8 层 LSTM 编码器 + 8 层 LSTM 解码器
   - 残差连接
   - 注意力机制
2. **Google Speech Recognition**
   - 5 层双向 LSTM
   - Dropout + 批归一化
3. **OpenAI GPT-2 (Transformer 前)**
   - 研究表明：2-3 层 LSTM + 注意力 ≈ 浅层 Transformer

♣

定理 2.21 (趋势)

现在更多用 Transformer，但深度 RNN 在某些任务仍有优势

♡

2.5.1.0.21 深度 RNN vs Transformer

特性	深度 RNN	Transformer
并行化	困难（顺序处理）	容易
训练速度	慢	快
长期依赖	LSTM 可以	更好
参数效率	高	低
实时推理	快（逐步）	慢（全局）
内存占用	低	高
流式处理	✓支持	× 不支持

定义 2.45 (深度 RNN 仍适用的场景)

- 实时流式任务（语音识别）
- 资源受限环境（移动设备）
- 在线学习（持续更新）
- 小数据集任务

♣

2.5.1.0.22 深度 RNN 关键要点

1. 层数选择
1 层：基线 | 2 层：标准 | 3 层：复杂任务 | > 3 层：谨慎

2. 正则化必须
Dropout（0.5）+ 梯度裁剪 + Early Stopping

3. 深度 vs 宽度
相近参数量下，深度（2 层 256）> 宽度（1 层 512）

4. 训练技巧
从浅到深、监控梯度、调整学习率

2.5.1.0.23 实践建议

定义 2.46 (如何选择深度)

1. 数据量小（<**10K** 样本）
   - 用 1 层，避免过拟合
2. 数据量中等（**10K-100K**）
   - 用 2 层，加 Dropout
3. 数据量大（>**100K**）
   - 尝试 3 层
   - 监控验证集性能
4. 调参建议
   - 先固定 1 层，调好隐藏单元数
   - 再逐步增加层数
   - 每次增加层数后重新调学习率

♣

2.5.1.0.24 完整示例：2 层 LSTM 语言模型

```
import torch
import torch.nn as nn

class TwoLayerLSTM(nn.Module):
    def __init__(self, vocab_size, embed_size=256, hidden_size=512):
        super().__init__()
        self.vocab_size = vocab_size

        # one-hot
        self.embedding = nn.Embedding(vocab_size, embed_size)

        # 2LSTM
        self.lstm = nn.LSTM(embed_size, hidden_size,
                            num_layers=2, dropout=0.5)

        # Output
        self.fc = nn.Linear(hidden_size, vocab_size)

    def forward(self, x, state):
        # x: (batch, seq_len)
        x = self.embedding(x.T) # (seq_len, batch, embed)
        out, state = self.lstm(x, state)
        out = self.fc(out.reshape(-1, out.shape[-1]))
```

```
        return out, state

    def init_state(self, batch_size, device):
        h = torch.zeros(2, batch_size, 512, device=device)
        c = torch.zeros(2, batch_size, 512, device=device)
        return (h, c)
```

2.5.1.0.25 常见错误与解决

定义 2.47 (问题 1：深度网络不收敛)

可能原因：

- 学习率太大 → 减小 10 倍
- 梯度爆炸 → 减小裁剪阈值
- 初始化不好 → 使用 Xavier/He 初始化

♣

定义 2.48 (问题 2：验证集性能不提升)

可能原因：

- 过拟合 → 增大 Dropout
- 层数太多 → 减少到 2 层
- 数据太少 → 数据增强或减少模型复杂度

♣

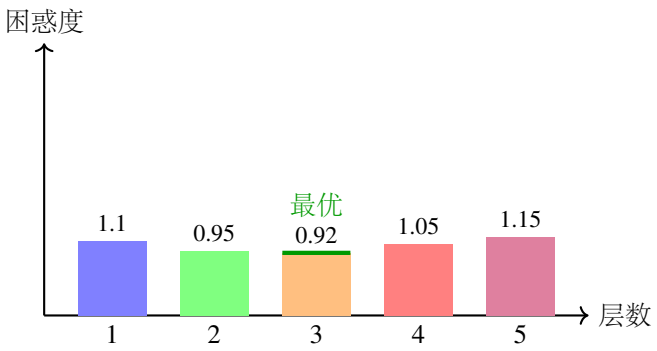
定义 2.49 (问题 3：训练很慢)

解决方案：

- 减少隐藏单元数
- 减少层数
- 使用 cuDNN（PyTorch 自动）
- 减小批量大小（换取速度）

♣

2.5.1.0.26 层数实验对比



3 层最优，更深反而性能下降（过拟合）

2.6 9.4 双向循环神经网络

2.6.1 9.4 双向循环神经网络

Bidirectional RNN

同时利用过去和未来的信息

2.6.2 9.4 节 - 章节导航

定义 2.50 (本节内容)

- 9.4.1 双向循环神经网络的动机
- 9.4.2 双向循环神经网络的定义
- 9.4.3 双向循环神经网络的实现
- 9.4.4 应用与限制

♣

定理 2.22 (学习目标)

1. 理解为什么需要双向 RNN
2. 掌握双向 RNN 的结构
3. 学会实现双向 RNN
4. 了解适用场景

♡

2.6.2.0.1 为什么需要双向 RNN?

标准 RNN 的局限：
只能利用**过去**的信息，看不到**未来**

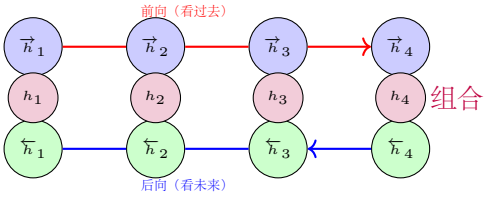
例子：填空题
“I am __ because I will go to Paris tomorrow.”
预测空格需要看到 “Paris”（未来信息）才知道是 “excited”

双向 RNN：同时从前向后和从后向前处理

2.6.2.0.2 双向 RNN 的核心思想 标准 RNN：只看过去

只看左边（过去）

双向 RNN：同时看过去和未来



2.7 未命名节

2.7.1 双向 RNN 的动机

2.7.2 9.4.1 双向循环神经网络的动机

2.7.3 9.4.1 双向循环神经网络的动机

为什么需要看” 未来”?

2.7.3.0.1 自然语言的双向性 观察：理解一个词常需要上下文

例题 2.4 例子 1：词性标注 “The **bank** is closed.” → bank = 银行（名词）

“I will **bank** on you.” → bank = 依靠（动词）

需要看后面的词才能确定词性

例题 2.5 例子 2：命名实体识别 “**Apple** released a new phone.” → Apple = 公司

“I ate an **apple**.” → apple = 水果

需要看上下文才能判断

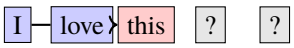
例题 2.6 例子 3：情感分析 “The movie was not bad.” → 正面（需要看到 “not”）

只看 “bad” 会误判为负面

2.7.3.0.2 单向 vs 双向对比

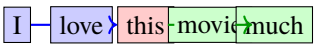
句子：“I love this movie very much”

单向 RNN 预测 “this”



只能用 “I love” → 信息不足

双向 RNN 预测 “this”



用 “I love” + “movie much” → 信息充分

2.7.3.0.3 双向 RNN 适用的任务

定义 2.51 (完全适合（离线任务）)

- 文本分类：情感分析、主题分类
- 序列标注：词性标注、命名实体识别
- 机器翻译：编码器部分
- 问答系统：理解问题和文档
- 文本摘要：理解全文

♣

定理 2.23 (不适合（在线任务）)

- 语言模型：预测下一个词（不能看未来）
- 实时语音识别：需要即时输出
- 在线翻译：解码器部分（逐词生成）

♡

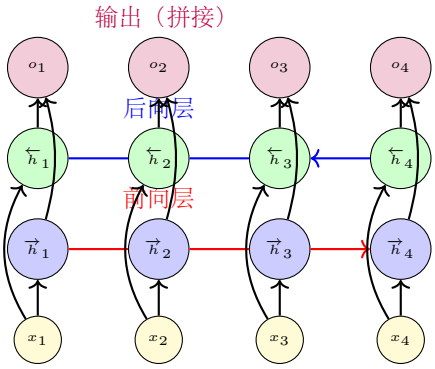
2.7.4 双向 RNN 的定义

2.7.5 9.4.2 双向循环神经网络的定义

2.7.6 9.4.2 双向循环神经网络的定义

双向 RNN 的数学表示

2.7.6.0.1 双向 RNN 的结构



每个时间步的输出结合了前向和后向信息

2.7.6.0.2 双向 RNN 的数学公式

定义 2.52 (前向层（从左到右）)

$$\overrightarrow{\mathbf{H}}_t = \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(f)} + \overrightarrow{\mathbf{H}}_{t-1} \mathbf{W}_{hh}^{(f)} + \mathbf{b}_h^{(f)})$$

定义 2.53 (后向层（从右到左）)

$$\overleftarrow{\mathbf{H}}_t = \phi(\mathbf{X}_t \mathbf{W}_{xh}^{(b)} + \overleftarrow{\mathbf{H}}_{t+1} \mathbf{W}_{hh}^{(b)} + \mathbf{b}_h^{(b)})$$

定义 2.54 (输出（拼接）)

$$\mathbf{H}_t = [\overrightarrow{\mathbf{H}}_t; \overleftarrow{\mathbf{H}}_t]$$

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hq} + \mathbf{b}_q$$

定理 2.24 (关键)

- $\overrightarrow{\mathbf{H}}_t$：前向隐状态，维度 h
- $\overleftarrow{\mathbf{H}}_t$：后向隐状态，维度 h
- $\mathbf{H}_t$：拼接后，维度 $2h$

2.7.6.0.3 双向 RNN 的计算流程

步骤 1：前向传播（$t = 1 \to T$）

$$\overrightarrow{\mathbf{H}}_1 = \phi(\mathbf{X}_1 \mathbf{W}_{xh}^{(f)} + \mathbf{b}_h^{(f)})$$

步骤 2：后向传播（$t = T \to 1$）

$$\overleftarrow{\mathbf{H}}_T = \phi(\mathbf{X}_T \mathbf{W}_{xh}^{(b)} + \mathbf{b}_h^{(b)})$$

$$\overleftarrow{\mathbf{H}}_{T-1} = \phi(\mathbf{X}_{T-1} \mathbf{W}_{xh}^{(b)} + \overleftarrow{\mathbf{H}}_T \mathbf{W}_{hh}^{(b)} + \mathbf{b}_h^{(b)})$$

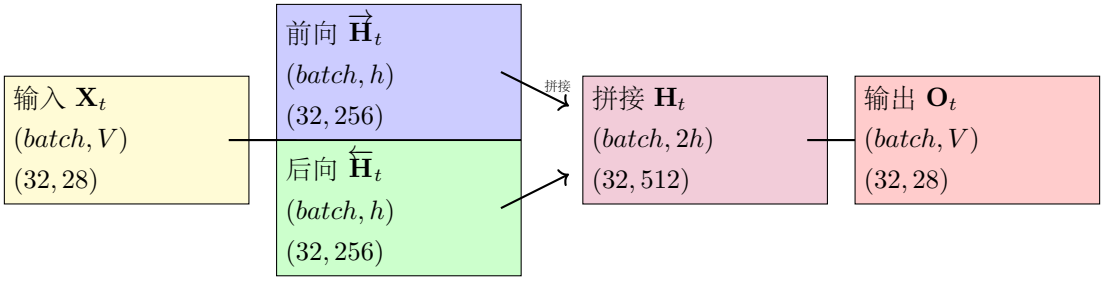
步骤 3：拼接隐状态

$$\mathbf{H}_t = [\overrightarrow{\mathbf{H}}_t; \overleftarrow{\mathbf{H}}_t] \quad \text{for } t = 1, \ldots, T$$

步骤 4：计算输出

$$\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{hq} + \mathbf{b}_q$$

2.7.6.0.4 维度变化分析 假设：隐藏单元数 $h = 256$，词表大小 $V = 28$



定理 2.25 (关键)

拼接后维度翻倍：$h \to 2h$ ♡

2.7.6.0.5 双向 RNN 的参数分析 参数数量：

定义 2.55 (单向 RNN)

$$\text{params} = (d + h) \times h + h + h \times V + V$$ ♣

定义 2.56 (双向 RNN)

- 前向层：$(d + h) \times h + h$
- 后向层：$(d + h) \times h + h$
- 输出层：$2h \times V + V$ （注意：$2h$）

$$\text{params} = 2 \times [(d + h) \times h + h] + 2h \times V + V$$ ♣

例题 2.7 具体例子（$d = 28, h = 256, V = 28$） 单向：$(28 + 256) \times 256 + 256 + 256 \times 28 + 28 \approx 80K$

双向：$2 \times (284 \times 256 + 256) + 512 \times 28 + 28 \approx 160K$

双向参数约是单向的 **2 倍**

2.7.7 双向 RNN 的实现

2.7.8 9.4.3 双向循环神经网络的实现

2.7.9 9.4.3 双向循环神经网络的实现

用 PyTorch 实现双向 RNN

2.7.9.0.1 使用 PyTorch 创建双向 RNN

```
import torch.nn as nn

# RNN
rnn_layer = nn.RNN(
    input_size=vocab_size,
    hidden_size=num_hiddens,
    num_layers=1,
    bidirectional=True # commentTrue
)

print(rnn_layer)
# RNN(28, 256, bidirectional=True)

# test
X = torch.randn(num_steps, batch_size, vocab_size)
H = torch.zeros(2, batch_size, num_hiddens) # comment2 = 2个方向

Y, H_new = rnn_layer(X, H)

print(f"输入shape: {X.shape}") # (5, 32, 28)
print(f"Outputshape: {Y.shape}") # (5, 32, 512) = 2*256
print(f"隐状态shape: {H_new.shape}") # (2, 32, 256)
```

定理 2.26 (注意)

- 输出维度：$2 \times hidden\_size$
- 隐状态第一维：$2 \times num\_layers$（2 个方向）

♡

2.7.9.0.2 双向 LSTM/GRU

```
# LSTM
lstm_layer = nn.LSTM(
    input_size=vocab_size,
    hidden_size=num_hiddens,
    num_layers=1,
    bidirectional=True
)

# LSTMHC
H = torch.zeros(2, batch_size, num_hiddens) # 2个方向
C = torch.zeros(2, batch_size, num_hiddens)
state = (H, C)

Y, (H_new, C_new) = lstm_layer(X, state)
print(f"Outputshape: {Y.shape}") # (num_steps, batch, 2*hidden)

# GRU
gru_layer = nn.GRU(
    input_size=vocab_size,
```

```
    hidden_size=num_hiddens,
    bidirectional=True
)
```

2.7.9.0.3 完整的双向 RNN 模型

```
class BiRNNModel(nn.Module):
    """RNNmodel"""

    def __init__(self, vocab_size, num_hiddens, rnn_type='lstm'):
        super(BiRNNModel, self).__init__()
        self.vocab_size = vocab_size
        self.num_hiddens = num_hiddens
        self.rnn_type = rnn_type

        # RNN
        if rnn_type == 'lstm':
            self.rnn = nn.LSTM(vocab_size, num_hiddens, bidirectional=True)
        elif rnn_type == 'gru':
            self.rnn = nn.GRU(vocab_size, num_hiddens, bidirectional=True)
        else:
            self.rnn = nn.RNN(vocab_size, num_hiddens, bidirectional=True)

        # Outputdimension2*num_hiddens
        self.linear = nn.Linear(2 * num_hiddens, vocab_size)

    def forward(self, inputs, state):
        # one-hot
        X = F.one_hot(inputs.T, self.vocab_size).type(torch.float32)

        # RNN
        Y, state = self.rnn(X, state)

        # Output
        output = self.linear(Y.reshape(-1, Y.shape[-1]))

        return output, state
```

2.7.9.0.4 完整的双向 RNN 模型（续）

```
    def begin_state(self, batch_size, device):
        # 2
        if self.rnn_type == 'lstm':
            # LSTMReturn(H, C)
            return (torch.zeros((2, batch_size, self.num_hiddens),
                                device=device),
                    torch.zeros((2, batch_size, self.num_hiddens),
                                device=device))
        else:
            # RNN/GRUReturnH
```

```
            return torch.zeros((2, batch_size, self.num_hiddens),
                               device=device)

# LSTMmodel
net = BiRNNModel(vocab_size=28, num_hiddens=256, rnn_type='lstm')

# parameters
num_params = sum(p.numel() for p in net.parameters())
print(f"parameters数量: {num_params:,}")
# parameters: 1,067,036 (2)
```

2.7.9.0.5 多层双向 RNN

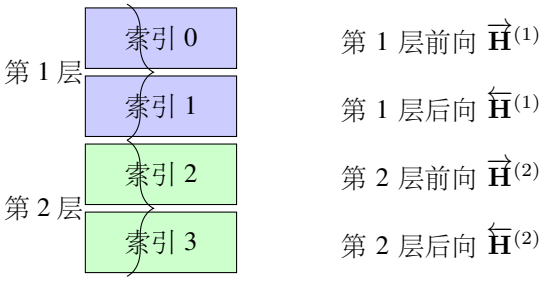
```
# 2LSTM
lstm_layer = nn.LSTM(
    input_size=vocab_size,
    hidden_size=num_hiddens,
    num_layers=2,       # 2层
    bidirectional=True, # comment
    dropout=0.5         # commentdropout
)

# 2 * 2 = 4
H = torch.zeros(4, batch_size, num_hiddens)
C = torch.zeros(4, batch_size, num_hiddens)
state = (H, C)

Y, (H_new, C_new) = lstm_layer(X, state)
print(f"Outputshape: {Y.shape}") # (5, 32, 512)
print(f"Hshape: {H_new.shape}") # (4, 32, 256)
print(f"Cshape: {C_new.shape}") # (4, 32, 256)
```

定理 2.27 (注意)

隐状态第一维 = $num\_layers \times 2$（方向数）

♡

2.7.9.0.6 双向 RNN 的隐状态结构 2 层双向 LSTM 的隐状态排列：



定义 2.57 (规律)

对于 L 层双向 RNN，隐状态顺序：
$[\overrightarrow{H}^{(1)}, \overleftarrow{H}^{(1)}, \overrightarrow{H}^{(2)}, \overleftarrow{H}^{(2)}, \ldots, \overrightarrow{H}^{(L)}, \overleftarrow{H}^{(L)}]$

♣

2.7.10 应用与限制

2.7.11 9.4.4 应用与限制

2.7.12 9.4.4 应用与限制

何时使用双向 RNN?

2.7.12.0.1 双向 RNN 的应用场景

定义 2.58 (完美适用（批处理任务）)

1. 序列标注
   - 词性标注（POS Tagging）
   - 命名实体识别（NER）
   - 语义角色标注
2. 文本分类
   - 情感分析
   - 主题分类
   - 垃圾邮件检测
3. 机器翻译编码器
   - 理解源语言句子
   - 提供丰富的上下文表示
4. 问答系统
   - 理解问题和文档
   - 提取答案

♣

2.7.12.0.2 双向 RNN 的限制

定理 2.28 (不适用（在线任务）)

1. 语言模型
   - 预测下一个词时不能看未来
   - 必须用单向 RNN
2. 实时语音识别
   - 需要即时输出
   - 不能等待整个句子
3. 在线翻译（解码器）
   - 逐词生成，看不到未来
4. 流式处理
   - 数据流式到达
   - 无法回看

♡

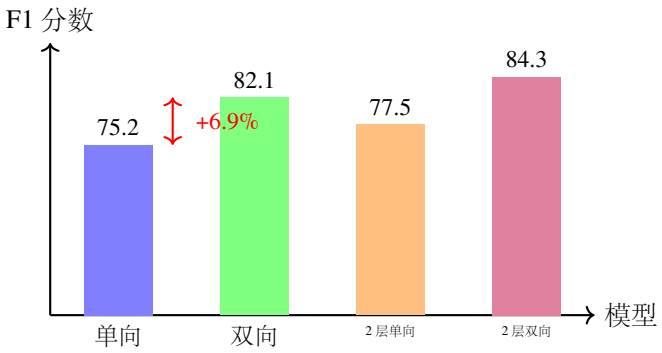
2.7.12.0.3 双向 RNN 的代价

优点：
✓ 性能更好
✓ 更丰富的上下文
✓ 适合大多数 NLP 任务

代价：
× 参数翻倍
× 计算翻倍
× 内存翻倍
× 不能在线推理

权衡：如果任务允许批处理，双向通常值得（性能提升 > 计算成本）

2.7.12.0.4 单向 vs 双向性能对比 任务：命名实体识别（NER）



定义 2.59 (观察)

双向 RNN 通常比单向提升 5-10 个百分点

♣

2.7.12.0.5 实际应用案例

定义 2.60 (经典应用)

1. **Google BERT (2018)**
   - 实际是双向 Transformer（概念类似）
   - 革命性地利用了双向信息
2. **ELMo (2018)**
   - 双向 LSTM 语言模型
   - 前向和后向独立训练再组合
3. **BiLSTM-CRF (NER 标准架构)**
   - 双向 LSTM 编码
   - CRF 层解码
   - NER 任务的标准选择
4. 机器翻译的编码器
   - 几乎都用双向 RNN
   - 充分理解源语言

♣

2.7.12.0.6 双向 RNN 的变体

定义 2.61 (改进的双向架构)

1. 深度双向 **RNN**
   - 垂直堆叠多层双向 RNN
   - 每层都是双向
2. 单向 + 双向混合

- 底层：双向（捕捉上下文）
- 顶层：单向（任务特定）

3. 不对称双向
   - 前向层：标准 LSTM
   - 后向层：简化版（节省计算）
4. 延迟后向
   - 前向实时处理
   - 后向延迟几步（平衡实时性和效果）

2.8 9.5 机器翻译与数据集

2.8.1 9.5 机器翻译与数据集

Machine Translation

序列到序列学习的经典应用

2.8.2 9.5 节 - 章节导航

定义 2.62 (本节内容)

- 9.5.1 机器翻译简介
- 9.5.2 数据集下载与预处理
- 9.5.3 词表构建
- 9.5.4 数据加载器

定理 2.29 (学习目标)

1. 理解机器翻译的挑战
2. 掌握双语数据集的处理
3. 学会构建源语言和目标语言词表
4. 实现序列对的批量加载

2.9 未命名节

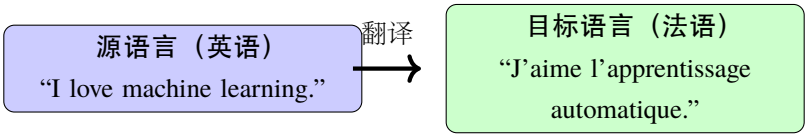
2.9.1 机器翻译简介

2.9.2 9.5.1 机器翻译简介

2.9.3 9.5.1 机器翻译简介

将一种语言翻译成另一种语言

2.9.3.0.1 什么是机器翻译？ 定义：自动将文本从源语言翻译到目标语言

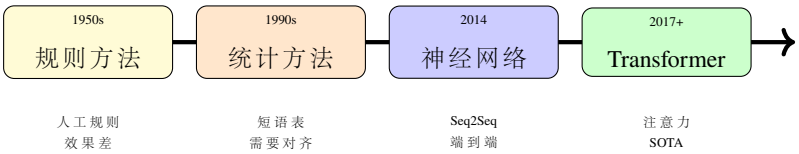


定义 2.63 (挑战)

- 语言结构差异（词序、语法）
- 一对多映射（一词多义）
- 上下文依赖
- 习语和文化差异

♣

2.9.3.0.2 机器翻译的历史

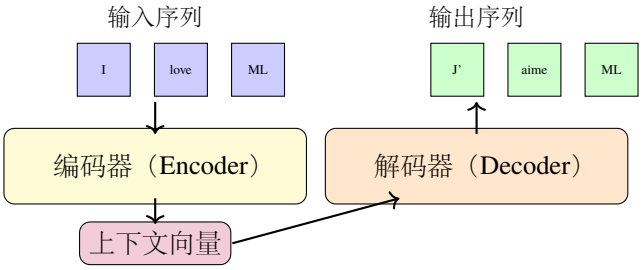


定理 2.30 (革命性进展)

2014 年：神经机器翻译（NMT）出现，性能超越传统方法

♡

2.9.3.0.3 序列到序列（Seq2Seq）框架 核心思想：输入序列 → 输出序列



编码器将源语言压缩成向量，解码器从向量生成目标语言

2.9.3.0.4 机器翻译的特点 与语言模型的区别：

- 语言模型：
  $P(x_1, \ldots, x_T)$
- 机器翻译：
  $P(y_1, \ldots, y_{T'} \mid x_1, \ldots, x_T)$
- 条件生成任务

挑战：

- 源序列和目标序列**长度不同**
- 需要**对齐**
- 一对多映射
- 长距离依赖

例题 2.8 例子 **英语**：“I love you”（3 个词）
法语：“Je t’aime”（2 个词）
德语：“Ich liebe dich”（3 个词）
长度不同！

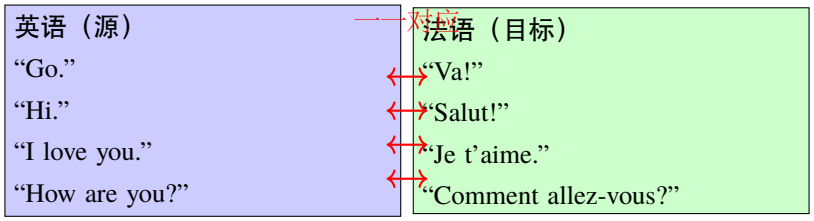
2.9.4 数据集下载与预处理

2.9.5 9.5.2 数据集下载与预处理

2.9.6 9.5.2 数据集下载与预处理

准备双语平行语料

2.9.6.0.1 平行语料（Parallel Corpus） 定义： 一一对应的源语言和目标语言句子对



定义 2.64 (来源)

- 官方文档翻译（联合国、欧盟）
- 电影字幕
- 书籍翻译
- 网页翻译

♣

2.9.6.0.2 本节使用的数据集 数据集： 英语-法语翻译对

定义 2.65 (数据集信息)

- 来源：Tatoeba 项目
- 语言对：英语（en）↔ 法语（fr）
- 规模：约 20 万句对
- 特点：短句为主，适合教学
- 格式：制表符分隔的文本文件

♣

例题 2.9 数据样例

```
Go.    Va!
Hi.    Salut!
Run!   Cours!
I see. Je vois.
```

2.9.6.0.3 下载和读取数据

```
import os
import torch

def download_extract(name, cache_dir='./data'):
    """data"""
    # URL
    return os.path.join(cache_dir, name)

def read_data_nmt():
```

```
    """读取英语-法语data集"""
    data_dir = download_extract('fra-eng')
    with open(os.path.join(data_dir, 'fra.txt'), 'r',
              encoding='utf-8') as f:
        return f.read()

raw_text = read_data_nmt()
print(raw_text[:75])
# Go.   Va!
# Hi.   Salut!
# Run!  Cours!
```

2.9.6.0.4 数据预处理

```
def preprocess_nmt(text):
    """data"""

    def no_space(char, prev_char):
        """判断是否需要在字符前加空格"""
        return char in set(',.!?') and prev_char != ' '

    text = text.replace('\u202f', ' ').replace('\xa0', ' ').lower()

    out = [' ' + char if i > 0 and no_space(char, text[i - 1])
           else char for i, char in enumerate(text)]

    return ''.join(out)

text = preprocess_nmt(raw_text)
print(text[:80])
# go .   va !
# hi .   salut !
# run !  cours !
```

定理 2.31 (预处理目的)

- 标准化格式（小写）
- 分离标点符号（便于词元化）

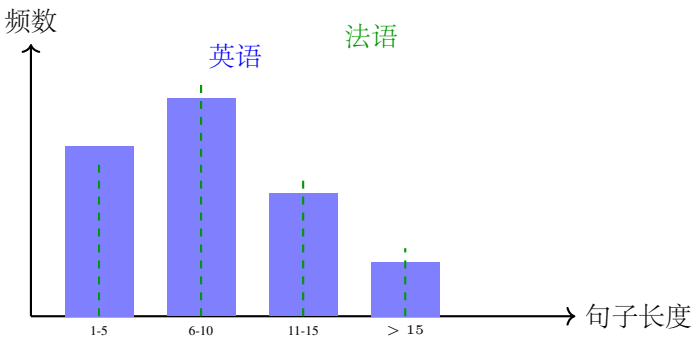
♡

2.9.6.0.5 词元化（Tokenization）

```
def tokenize_nmt(text, num_examples=None):
    """ tokensdata Return: tokens """
    source, target = [], []

    for i, line in enumerate(text.split('\n')):
        if num_examples and i >= num_examples:
            break

        parts = line.split('\t')
```

```
        if len(parts) == 2:
            source.append(parts[0].split(' '))
            target.append(parts[1].split(' '))

    return source, target

# tokens
source, target = tokenize_nmt(text)
print(source[:3])
# [['go', '.'], ['hi', '.'], ['run', '!']]
print(target[:3])
# [['va', '!'], ['salut', '!'], ['cours', '!']]
```

2.9.6.0.6 数据统计



定义 2.66 (观察)

- 大多数句子在 10 个词以内
- 英语和法语长度分布相似
- 适合限制最大长度

2.9.6.0.7 过滤和截断

```
def truncate_pad(line, num_steps, padding_token):
    """sequencelength"""
    if len(line) > num_steps:
        return line[:num_steps] # comment
    return line + [padding_token] * (num_steps - len(line)) # comment

def filter_examples(source, target, max_len=10):
    """过滤过长的句子"""
    source_filtered, target_filtered = [], []

    for src, tgt in zip(source, target):
        # lengthmax_len
        if len(src) <= max_len and len(tgt) <= max_len:
            source_filtered.append(src)
            target_filtered.append(tgt)

    return source_filtered, target_filtered
```

```
18
19 source, target = filter_examples(source, target, max_len=10)
20 print(f"过滤后句子对数量: {len(source)}")
21 # : 152,777
```

2.9.7 词表构建

2.9.8 9.5.3 词表构建

2.9.9 9.5.3 词表构建

为源语言和目标语言分别构建词表

2.9.9.0.1 双语词表 关键：源语言和目标语言需要独立的词表

英语词表

```
'<pad>': 0,
'<bos>': 1,
'<eos>': 2,
'i': 3,
'love': 4,
...
```

法语词表

```
'<pad>': 0,
'<bos>': 1,
'<eos>': 2,
'je': 3,
'aime': 4,
```

两个词表独立，同一个索引对应不同的词

定理 2.32 (特殊词元)

- <pad>：填充
- <bos>：句子开始（Begin of Sequence）
- <eos>：句子结束（End of Sequence）

♡

2.9.9.0.2 为什么需要特殊词元?

定义 2.67 (<pad>（填充）)

批量处理时，需要将不同长度的句子填充到相同长度

```
"Go ."        -> [Go, ., <eos>, <pad>, <pad>]
"I love you" -> [I, love, you, ., <eos>]
```

♣

定义 2.68 (<bos>（开始）)

解码器的初始输入，告诉模型”开始生成”

输入：[<bos>] -> 输出："Je"
输入：[<bos>, Je] -> 输出："t'"

♣

定义 2.69 (<eos>（结束）)

标记句子结束，模型知道何时停止生成

生成序列：[Je, t', aime, ., <eos>] -> 停止

♣

2.9.9.0.3 构建词表

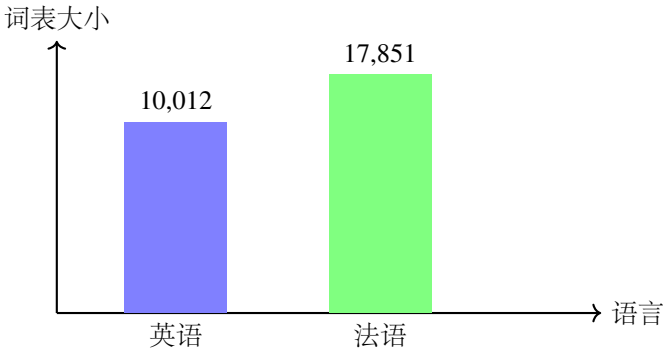
```
class Vocab:
    """vocabulary"""
    def __init__(self, tokens=None, min_freq=0, reserved_tokens=None):
        if tokens is None:
            tokens = []
        if reserved_tokens is None:
            reserved_tokens = []

        counter = count_corpus(tokens)
        self._token_freqs = sorted(counter.items(),
                                   key=lambda x: x[1],
                                   reverse=True)

        # tokens +
        self.idx_to_token = ['<unk>'] + reserved_tokens
        self.token_to_idx = {token: idx
                             for idx, token in enumerate(self.idx_to_token)}

        for token, freq in self._token_freqs:
            if freq < min_freq:
                break
            if token not in self.token_to_idx:
                self.idx_to_token.append(token)
                self.token_to_idx[token] = len(self.idx_to_token) - 1
```

2.9.9.0.4 构建双语词表

```
def build_array_nmt(lines, vocab, num_steps):
    """sequence"""
    # <eos>
    lines = [vocab[l] + [vocab['<eos>']] for l in lines]

    # length
    array = torch.tensor([
        truncate_pad(l, num_steps, vocab['<pad>'])
        for l in lines
    ])

    # lengthpadding
    valid_len = (array != vocab['<pad>']).type(torch.int32).sum(1)

    return array, valid_len

# vocabulary
src_vocab = Vocab(source, min_freq=2,
                  reserved_tokens=['<pad>', '<bos>', '<eos>'])
tgt_vocab = Vocab(target, min_freq=2,
                  reserved_tokens=['<pad>', '<bos>', '<eos>'])
```

```
22
23 print(f"英语vocabulary大小: {len(src_vocab)}") # comment10,000
24 print(f"法语vocabulary大小: {len(tgt_vocab)}") # comment17,000
```

2.9.9.0.5 词表大小对比



法语词汇更丰富（动词变位、名词性别等）

定理 2.33 (观察)

法语词表比英语大约 70%，因为法语形态变化更丰富

♡

2.9.10 数据加载器

2.9.11 9.5.4 数据加载器

2.9.12 9.5.4 数据加载器

批量加载句子对

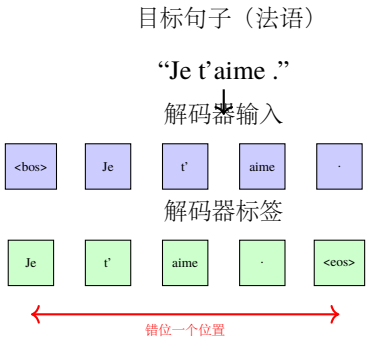
2.9.12.0.1 数据加载的挑战

定义 2.70 (序列对的特殊性)

1. 两个序列
   - 源序列（英语）
   - 目标序列（法语）
2. 长度不同
   - 源序列和目标序列长度可能不同
   - 需要分别填充
3. 解码器需要两个目标
   - 输入：$[\text{<bos>}, y_1, \ldots, y_{T-1}]$
   - 标签：$[y_1, \ldots, y_{T-1}, \text{<eos>}]$

♣

2.9.12.0.2 解码器的输入输出



定理 2.34 (Teacher Forcing)

训练时，解码器的每一步都用真实的前一个词作为输入 ♡

2.9.12.0.3 数据加载函数

```
def load_data_nmt(batch_size, num_steps, num_examples=600):
    """ data Return: dataiterationvocabulary """
    text = preprocess_nmt(read_data_nmt())
    source, target = tokenize_nmt(text, num_examples)

    # vocabulary
    src_vocab = Vocab(source, min_freq=2,
                      reserved_tokens=['<pad>', '<bos>', '<eos>'])
    tgt_vocab = Vocab(target, min_freq=2,
                      reserved_tokens=['<pad>', '<bos>', '<eos>'])

    src_array, src_valid_len = build_array_nmt(source, src_vocab, num_steps)
    tgt_array, tgt_valid_len = build_array_nmt(target, tgt_vocab, num_steps)

    # data
    data_arrays = (src_array, src_valid_len, tgt_array, tgt_valid_len)
    data_iter = load_array(data_arrays, batch_size)

    return data_iter, src_vocab, tgt_vocab
```

2.9.12.0.4 使用数据加载器

```
# data
train_iter, src_vocab, tgt_vocab = load_data_nmt(
    batch_size=2,
    num_steps=8,
    num_examples=600
)

# batch
for X, X_valid_len, Y, Y_valid_len in train_iter:
    print('X:', X.type(torch.int32))
    print('X的有效length:', X_valid_len)
    print('Y:', Y.type(torch.int32))
    print('Y的有效length:', Y_valid_len)
```

```
    break

# Output
# X: tensor([[ 47,  4,   3,   3,   3, 1388,  5,   2],
#            [ 33,  81,  4,   3,   3,   5,   2,   0]])
# Xlength: tensor([8, 7])
# Y: tensor([[ 91,  4,  3,  3,  14,  5,  2,  0],
#            [ 24, 115,  4,  3,  3,  5,  2,  0]])
# Ylength: tensor([7, 7])
```

2.9.12.0.5 批次数据的结构 一个批次包含：

源序列 X: $(batch\_size, num\_steps)$ 编码器的输入，已填充到 num_steps
源序列有效长度 X_valid_len: $(batch\_size,)$ 每个序列的实际长度（不含 padding）
目标序列 Y: $(batch\_size, num\_steps)$ 解码器的输入和标签，已填充
目标序列有效长度 Y_valid_len: $(batch\_size,)$ 每个序列的实际长度

2.9.13 9.5 节总结

定义 2.71 (核心内容)

1. 机器翻译：序列到序列任务
   - 编码器-解码器架构
   - 条件生成：$P(y_{1:T'} \mid x_{1:T})$
2. 数据预处理：
   - 标准化、词元化
   - 过滤过长句子
   - 填充到固定长度
3. 词表构建：
   - 源语言和目标语言分别构建
   - 特殊词元：<pad>, <bos>, <eos>
4. 数据加载：
   - 批量加载句子对
   - 记录有效长度

♣

2.9.13.0.1 关键要点回顾

1. 平行语料
源语言和目标语言一一对应的句子对

2. 特殊词元
<bos> 开始 | <eos> 结束 | <pad> 填充

3. 双词表
源语言和目标语言独立的词表

4. Teacher Forcing
解码器训练时用真实词作为输入

2.10 编码器-解码器架构

2.11 9.6 编码器-解码器架构

2.11.1 9.6 编码器-解码器架构

Encoder-Decoder Architecture

序列到序列模型的核心框架

2.11.2 9.6 节 - 章节导航

定义 2.72 (本节内容)

- 9.6.1 编码器
- 9.6.2 解码器
- 9.6.3 编码器-解码器模型
- 9.6.4 损失函数与训练

定理 2.35 (学习目标)

1. 理解编码器-解码器架构
2. 实现 RNN 编码器
3. 实现 RNN 解码器
4. 掌握序列到序列模型的训练

2.11.2.0.1 编码器-解码器架构概览

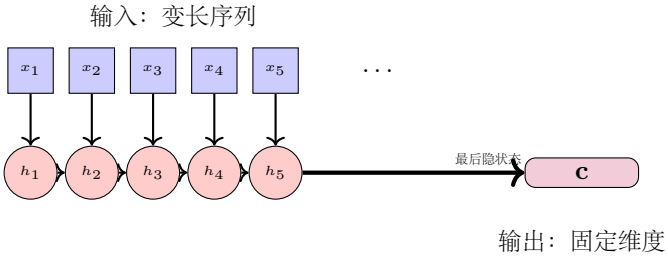
2.11.3 编码器

2.11.4 9.6.1 编码器

2.11.5 9.6.1 编码器

将源序列编码为上下文向量

2.11.5.0.1 编码器的任务 目标：将变长的源序列压缩成固定长度的向量



定义 2.73 (关键点)

- 上下文向量 $\mathbf{c} = \mathbf{h}_T$（最后一个隐状态）
- 维度固定（如 512 维）
- 压缩了整个源序列的信息

♣

2.11.5.0.2 编码器的抽象接口

```
class Encoder(nn.Module):
    """ () """

    def __init__(self, **kwargs):
        super(Encoder, self).__init__(**kwargs)

    def forward(self, X, *args):
        """
        前向传播
        X: 输入sequence
        Return: 上下文vector（可以是隐状态、Output等）
        """
        raise NotImplementedError
```

定理 2.36 (设计模式)

定义抽象基类，具体的编码器（RNN、LSTM、Transformer 等）继承并实现

♡

2.11.5.0.3 RNN 编码器实现

```
class Seq2SeqEncoder(Encoder):
    """RNN"""

    def __init__(self, vocab_size, embed_size, num_hiddens,
                 num_layers, dropout=0, **kwargs):
        super(Seq2SeqEncoder, self).__init__(**kwargs)
```

```
        self.embedding = nn.Embedding(vocab_size, embed_size)

        # RNNLSTMGRU
        self.rnn = nn.GRU(embed_size, num_hiddens, num_layers,
                          dropout=dropout)

    def forward(self, X, *args):
        """
        X: (batch_size, num_steps) 输入sequence（词索引）
        Return: (Output, 隐状态)
        """
        # (batch, steps) -> (batch, steps, embed)
        X = self.embedding(X)

        # (batch, steps, embed) -> (steps, batch, embed)
        X = X.permute(1, 0, 2)

        # RNN
        output, state = self.rnn(X)

        return output, state
```

2.11.5.0.4 编码器的组件

1. 嵌入层（Embedding）
词索引 → 稠密向量

2. RNN 层（GRU/LSTM）
处理嵌入序列

3. 输出
- output: 所有时间步的隐状态
- state: 最终隐状态（上下文向量）

2.11.5.0.5 测试编码器

```
vocab_size = 10 # commentvocabulary大小
embed_size = 8 # commentdimension
num_hiddens = 16 # comment
num_layers = 2 # comment
batch_size = 4
num_steps = 7

encoder = Seq2SeqEncoder(vocab_size, embed_size, num_hiddens, num_layers)

# test
X = torch.zeros((batch_size, num_steps), dtype=torch.long)
output, state = encoder(X)

```

```
print(f"Outputshape: {output.shape}") # (7, 4, 16)
print(f"状态shape: {state.shape}") # (2, 4, 16)
```

定义 2.74 (输出解释)

- output: $(num\_steps, batch, hidden)$ 所有隐状态
- state: $(num\_layers, batch, hidden)$ 最终状态

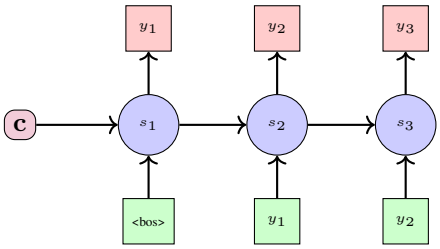
2.11.6 解码器

2.11.7 9.6.2 解码器

2.11.8 9.6.2 解码器

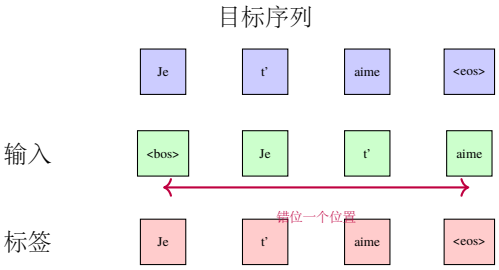
从上下文向量生成目标序列

2.11.8.0.1 解码器的任务 **目标**：基于上下文向量，逐步生成目标序列



每步基于上下文和前一个词预测下一个词

2.11.8.0.2 解码器的输入 **训练时（Teacher Forcing）**：



定理 2.37 (Teacher Forcing)

训练时，每一步都用真实的前一个词作为输入（而非预测的词）

2.11.8.0.3 解码器的抽象接口

```
class Decoder(nn.Module):
    """Simple function"""

    def __init__(self, **kwargs):
        super(Decoder, self).__init__(**kwargs)

    def init_state(self, enc_outputs, *args):
```

```
        """
        根据编码器Output初始化解码器状态
        enc_outputs: 编码器的Output
        """
        raise NotImplementedError

    def forward(self, X, state):
        """
        前向传播
        X: 目标sequence（解码器输入）
        state: 解码器状态
        """
        raise NotImplementedError
```

2.11.8.0.4 RNN 解码器实现

```
class Seq2SeqDecoder(Decoder):
    """RNN"""

    def __init__(self, vocab_size, embed_size, num_hiddens,
                 num_layers, dropout=0, **kwargs):
        super(Seq2SeqDecoder, self).__init__(**kwargs)

        self.embedding = nn.Embedding(vocab_size, embed_size)

        # RNN = embed + context
        self.rnn = nn.GRU(embed_size + num_hiddens, num_hiddens,
                          num_layers, dropout=dropout)

        # Output
        self.dense = nn.Linear(num_hiddens, vocab_size)

    def init_state(self, enc_outputs, *args):
        """用编码器的最终状态初始化"""
        return enc_outputs[1] # enc_outputs = (output, state)
```

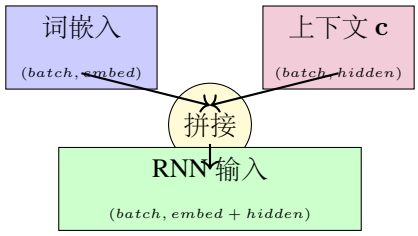
2.11.8.0.5 RNN 解码器前向传播

```
    def forward(self, X, state):
        """ X: (batch_size, num_steps) sequence state: ()  """
        X = self.embedding(X).permute(1, 0, 2)
        # X: (num_steps, batch, embed_size)

        # vector
        context = state[-1].repeat(X.shape[0], 1, 1)
        # context: (num_steps, batch, num_hiddens)

        X_and_context = torch.cat((X, context), 2)
        # (num_steps, batch, embed + hidden)

```

```
        # RNN
        output, state = self.rnn(X_and_context, state)

        # Output
        output = self.dense(output).permute(1, 0, 2)
        # (batch, num_steps, vocab_size)

        return output, state
```

2.11.8.0.6 解码器的输入结构

每个时间步的输入



定理 2.38 (关键)

每个时间步都将词嵌入和上下文向量拼接作为 RNN 输入

2.11.8.0.7 测试解码器

```
decoder = Seq2SeqDecoder(vocab_size, embed_size, num_hiddens, num_layers)

state = decoder.init_state(encoder(X))

# sequence
Y = torch.zeros((batch_size, num_steps), dtype=torch.long)

output, state = decoder(Y, state)

print(f"Outputshape: {output.shape}") # (4, 7, 10)
print(f"状态shape: {state.shape}") # (2, 4, 16)
```

定义 2.75 (输出解释)

- output: $(batch, num\_steps, vocab\_size)$ 每步的预测分布
- state: $(num\_layers, batch, hidden)$ 最终状态

2.11.9 编码器-解码器模型

2.11.10 9.6.3 编码器-解码器模型

2.11.11 9.6.3 编码器-解码器模型

组合编码器和解码器

2.11.11.0.1 EncoderDecoder 类

```
class EncoderDecoder(nn.Module):
    """-"""

    def __init__(self, encoder, decoder, **kwargs):
        super(EncoderDecoder, self).__init__(**kwargs)
        self.encoder = encoder
        self.decoder = decoder

    def forward(self, enc_X, dec_X, *args):
        """
        enc_X: 编码器输入（源sequence）
        dec_X: 解码器输入（目标sequence，training时）
        """
        enc_outputs = self.encoder(enc_X, *args)

        dec_state = self.decoder.init_state(enc_outputs, *args)

        return self.decoder(dec_X, dec_state)
```

2.11.11.0.2 创建完整模型

```
# parameters
src_vocab_size = 10 # commentvocabulary
tgt_vocab_size = 10 # commentvocabulary
embed_size = 8
num_hiddens = 16
num_layers = 2

encoder = Seq2SeqEncoder(src_vocab_size, embed_size,
                         num_hiddens, num_layers)
decoder = Seq2SeqDecoder(tgt_vocab_size, embed_size,
                         num_hiddens, num_layers)

# model
net = EncoderDecoder(encoder, decoder)

# test
enc_X = torch.zeros((batch_size, num_steps), dtype=torch.long)
dec_X = torch.zeros((batch_size, num_steps), dtype=torch.long)
output, state = net(enc_X, dec_X)

print(f"Outputshape: {output.shape}") # (4, 7, 10)
```

2.11.12 损失函数与训练

2.11.13 9.6.4 损失函数与训练

2.11.14 9.6.4 损失函数与训练

训练序列到序列模型

2.11.14.0.1 序列到序列的损失 目标：最大化 $P(y_1, \ldots, y_{T'} \mid x_1, \ldots, x_T)$

定义 2.76 (交叉熵损失)

$$\mathcal{L} = -\frac{1}{T'} \sum_{t=1}^{T'} \log P(y_t \mid y_1, \ldots, y_{t-1}, \mathbf{c})$$

♣

问题：不同序列长度不同，如何处理 padding?

定理 2.39 (Masked Loss)

只计算有效位置的损失，忽略 padding 部分

$$\mathcal{L} = -\frac{1}{\sum_t \mathbb{1}[y_t \neq \text{<pad>}]} \sum_{t: y_t \neq \text{<pad>}} \log P(y_t \mid \cdots)$$

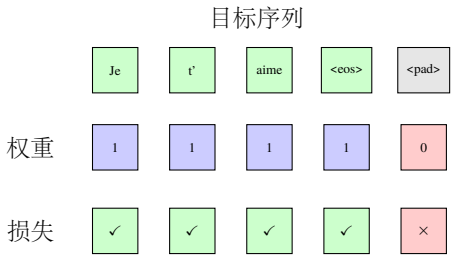
♡

2.11.14.0.2 带 Mask 的损失函数

```python
def sequence_mask(X, valid_len, value=0):
    """ sequencepadding X: (batch, steps, ...) valid_len: (batch,) sequencelength """
    maxlen = X.size(1)
    mask = torch.arange((maxlen), dtype=torch.float32,
                        device=X.device)[None, :] < valid_len[:, None]
    X[~mask] = value
    return X

def masked_softmax_cross_entropy(pred, label, valid_len):
    """
    带mask的交叉熵
    pred: (batch, steps, vocab_size)
    label: (batch, steps)
    valid_len: (batch,) 有效length
    """
    # 1padding0
    weights = torch.ones_like(label)
    weights = sequence_mask(weights, valid_len)

    weights = weights.reshape(-1)
    pred = pred.reshape(-1, pred.shape[-1])
    label = label.reshape(-1)

    # loss
    unweighted_loss = nn.CrossEntropyLoss(reduction='none')(pred, label)

```

```
    weighted_loss = (unweighted_loss * weights).sum() / weights.sum()
    return weighted_loss
```

2.11.14.0.3 Mask 损失的可视化



只计算有效位置的损失

2.11.14.0.4 训练函数

```
def train_seq2seq(net, data_iter, lr, num_epochs, device):
    """trainingseq2seqmodel"""
    net.to(device)
    optimizer = torch.optim.Adam(net.parameters(), lr=lr)

    for epoch in range(num_epochs):
        net.train()
        total_loss = 0
        num_batches = 0

        for batch in data_iter:
            X, X_valid_len, Y, Y_valid_len = [x.to(device) for x in batch]

            # bos
            bos = torch.tensor([tgt_vocab['<bos>']] * Y.shape[0],
                               device=device).reshape(-1, 1)
            dec_input = torch.cat([bos, Y[:, :-1]], 1) # Teacher forcing

            Y_hat, _ = net(X, dec_input)

            # loss
            l = masked_softmax_cross_entropy(Y_hat, Y, Y_valid_len)

            optimizer.zero_grad()
            l.backward()
            grad_clipping(net, 1)
            optimizer.step()

            total_loss += l.item()
            num_batches += 1

        print(f'epoch {epoch + 1}, loss {total_loss / num_batches:.3f}')
```

2.11.14.0.5 训练过程详解

步骤 1：准备解码器输入
目标序列：[Je, t', aime, <eos>]

步骤 2：前向传播
编码器处理源序列 → 解码器生成预测

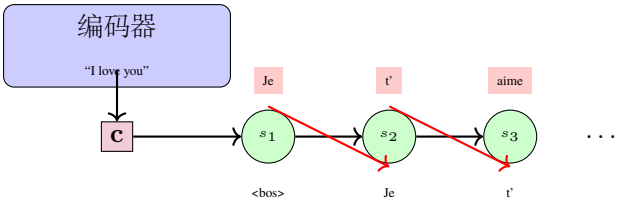
步骤 3：计算 Masked 损失
只在有效位置计算交叉熵（忽略 padding）

步骤 4：反向传播与更新
梯度裁剪 → 优化器更新

2.11.14.0.6 预测（推理）

```
def predict_seq2seq(net, src_sentence, src_vocab, tgt_vocab,
                num_steps, device):
    """ trainingmodel src_sentence:  ()  """
    net.eval()

    src_tokens = src_vocab[src_sentence.lower().split(' ')] + [
        src_vocab['<eos>']]
    enc_valid_len = torch.tensor([len(src_tokens)], device=device)

    src_tokens = truncate_pad(src_tokens, num_steps, src_vocab['<pad>'])
    enc_X = torch.unsqueeze(
        torch.tensor(src_tokens, dtype=torch.long, device=device), dim=0)

    enc_outputs = net.encoder(enc_X, enc_valid_len)
    dec_state = net.decoder.init_state(enc_outputs, enc_valid_len)

    dec_X = torch.unsqueeze(
        torch.tensor([tgt_vocab['<bos>']], dtype=torch.long,
                 device=device), dim=0)
    output_seq = []

    for _ in range(num_steps):
        Y, dec_state = net.decoder(dec_X, dec_state)
        dec_X = Y.argmax(dim=2) # comment
        pred = dec_X.squeeze(dim=0).type(torch.int32).item()

        if pred == tgt_vocab['<eos>']:
            break
        output_seq.append(pred)

    return ' '.join(tgt_vocab.to_tokens(output_seq))
```

2.11.14.0.7 推理过程（贪心解码）



每步选概率最大的词，作为下一步输入

定理 2.40 (贪心解码)

每步选择概率最大的词，简单但可能次优（下节介绍束搜索）

♡

2.11.14.0.8 BLEU 评估

```
def bleu(pred_seq, label_seq, k):
    """BLEU"""
    pred_tokens, label_tokens = pred_seq.split(' '), label_seq.split(' ')
    len_pred, len_label = len(pred_tokens), len(label_tokens)

    score = math.exp(min(0, 1 - len_label / len_pred))

    for n in range(1, k + 1):
        num_matches, label_subs = 0, collections.defaultdict(int)

        for i in range(len_label - n + 1):
            label_subs[' '.join(label_tokens[i: i + n])] += 1

        for i in range(len_pred - n + 1):
            if label_subs[' '.join(pred_tokens[i: i + n])] > 0:
                num_matches += 1
                label_subs[' '.join(pred_tokens[i: i + n])] -= 1

        score *= math.pow(num_matches / (len_pred - n + 1),
                          math.pow(0.5, n))

    return score

bleu('je t aime .', 'je t aime .', k=2) # 1.0 (完美匹配)
bleu('je t aime', 'je t aime .', k=2) # 0.658 (length惩罚)
```

2.11.14.0.9 BLEU 分数（BiLingual Evaluation Understudy） 评估翻译质量的标准指标

定义 2.77 (核心思想)

- 统计 n-gram 重叠
- 长度惩罚
- 综合评分

♣

例题 2.10 例子 **参考**：“Je t’aime .”

预测 1：“Je t’aime .” ⇒ BLEU = 1.0

预测 2：“Je aime .” ⇒ BLEU ≈ 0.7
预测 3：“aime Je .” ⇒ BLEU ≈ 0.4

定理 2.41 (范围)
BLEU $\in [0,1]$，越高越好

2.11.15 9.6 节总结

定义 2.78 (核心内容)
1. 编码器：将源序列压缩为上下文向量
$$\mathbf{c} = \text{Encoder}(x_1, \ldots, x_T)$$
2. 解码器：从上下文生成目标序列
$$P(y_t \mid y_{1:t-1}, \mathbf{c})$$
3. 训练：Teacher Forcing + Masked Loss
4. 推理：贪心解码（每步选最大概率）
5. 评估：BLEU 分数

2.11.15.0.1 关键要点回顾

1. 上下文向量
编码器最后隐状态，压缩整个源序列信息

2. Teacher Forcing
训练时用真实词作为解码器输入

3. Masked Loss
只计算有效位置损失，忽略 padding

4. 贪心解码
推理时每步选概率最大的词

2.11.15.0.2 编码器-解码器架构的局限

定理 2.42 (核心问题：信息瓶颈)
将整个源序列压缩到固定长度的向量
- 源序列很长时，信息损失严重
- 解码器看不到源序列细节
- 所有解码步骤共享同一个上下文

例题 2.11 例子 源句子：“The agreement on the European Economic Area was signed in August 1992.”
上下文向量（512 维）很难完整保存这么多信息！

定义 2.79 (解决方案)
注意力机制（Attention）：让解码器动态关注源序列的不同部分

2.11.15.0.3 实践建议

定义 2.80 (超参数选择)

- 嵌入维度：128-512
- 隐藏单元：256-1024
- 层数：1-4 层
- **Dropout**：0.1-0.5
- 学习率：0.001（Adam）
- 梯度裁剪：阈值 1-5

♣

定义 2.81 (训练技巧)

- 从小数据集开始验证
- 监控训练集和验证集 BLEU
- 使用学习率衰减
- 早停（验证集 BLEU 不再提升）

♣

2.11.15.0.4 常见问题与解决

定义 2.82 (问题 1：生成重复)

现象："Je Je Je ..."

原因：模型陷入局部最优

解决：

- 使用束搜索（下一节）
- 增加 Dropout
- 使用 Coverage 机制

♣

定义 2.83 (问题 2：翻译不完整)

现象：提前输出 `<eos>`

原因：长度惩罚不够

解决：

- 调整 BLEU 中的长度惩罚
- 限制最小输出长度

♣

2.12 9.7 序列到序列学习

2.12.1 9.7 序列到序列学习

Sequence to Sequence Learning

完整实现机器翻译系统

2.12.2 9.7 节 - 章节导航

定义 2.84 (本节内容)

- 9.7.1 训练 Seq2Seq 模型
- 9.7.2 预测与评估
- 9.7.3 性能分析

- 9.7.4 改进方向

定理 2.43 (学习目标)

1. 在真实数据集上训练 Seq2Seq
2. 评估翻译质量
3. 分析模型性能
4. 了解改进方法

2.13 未命名节

2.13.1 训练 Seq2Seq 模型

2.13.2 9.7.1 训练 Seq2Seq 模型

2.13.3 9.7.1 训练 Seq2Seq 模型

英语到法语的翻译

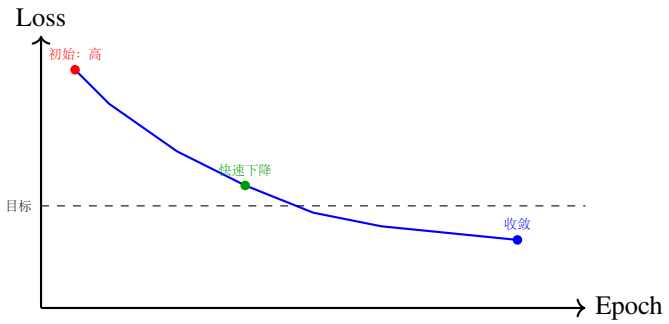
2.13.3.0.1 完整的训练流程

```
# data
batch_size = 64
num_steps = 10
lr = 0.005
num_epochs = 300

train_iter, src_vocab, tgt_vocab = load_data_nmt(
    batch_size, num_steps)

# modelparameters
embed_size = 32
num_hiddens = 32
num_layers = 2
dropout = 0.1

# model
encoder = Seq2SeqEncoder(len(src_vocab), embed_size,
                         num_hiddens, num_layers, dropout)
decoder = Seq2SeqDecoder(len(tgt_vocab), embed_size,
                         num_hiddens, num_layers, dropout)
net = EncoderDecoder(encoder, decoder)

# training
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
train_seq2seq(net, train_iter, lr, num_epochs, tgt_vocab, device)
```

2.13.3.0.2 训练循环详解

```
def train_seq2seq(net, data_iter, lr, num_epochs, tgt_vocab, device,
                  save_path='seq2seq.pth'):
    """trainingseq2seqmodel"""
    net.to(device)
    optimizer = torch.optim.Adam(net.parameters(), lr=lr)
    animator = Animator(xlabel='epoch', ylabel='loss',
                        xlim=[10, num_epochs])

    for epoch in range(num_epochs):
        timer = Timer()
        metric = Accumulator(2) # loss和, tokens数

        for batch in data_iter:
            optimizer.zero_grad()
            X, X_valid_len, Y, Y_valid_len = [x.to(device) for x in batch]

            # Y<bos>
            bos = torch.tensor([tgt_vocab['<bos>']] * Y.shape[0],
                               device=device).reshape(-1, 1)
            dec_input = torch.cat([bos, Y[:, :-1]], 1)

            Y_hat, _ = net(X, dec_input)

            # loss
            l = masked_softmax_cross_entropy(Y_hat, Y, Y_valid_len)
            l.sum().backward()

            grad_clipping(net, 1)

            optimizer.step()

            with torch.no_grad():
                metric.add(l.sum(), Y_valid_len.sum())

        if (epoch + 1) % 10 == 0:
            animator.add(epoch + 1, (metric[0] / metric[1],))

    print(f'loss {metric[0] / metric[1]:.3f}, '
          f'{metric[1] / timer.stop():.1f} tokens/sec on {str(device)}')

    # model
    torch.save(net.state_dict(), save_path)
```

2.13.3.0.3 训练过程可视化



典型训练曲线：初期快速下降，后期缓慢收敛

定义 2.85 (训练输出示例)

```
loss 0.023, 14523.3 tokens/sec on cuda:0
```

♣

2.13.4 预测与评估

2.13.5 9.7.2 预测与评估

2.13.6 9.7.2 预测与评估

用训练好的模型翻译

2.13.6.0.1 预测函数

```
def predict_seq2seq(net, src_sentence, src_vocab, tgt_vocab,
                num_steps, device, save_attention_weights=False):
    """prediction"""
    net.eval()
    src_tokens = src_vocab[src_sentence.lower().split(' ')] + [
        src_vocab['<eos>']]
    enc_valid_len = torch.tensor([len(src_tokens)], device=device)

    # length
    src_tokens = truncate_pad(src_tokens, num_steps, src_vocab['<pad>'])
    enc_X = torch.unsqueeze(
        torch.tensor(src_tokens, dtype=torch.long, device=device),
        dim=0)

    enc_outputs = net.encoder(enc_X, enc_valid_len)
    dec_state = net.decoder.init_state(enc_outputs, enc_valid_len)

    dec_X = torch.unsqueeze(
        torch.tensor([tgt_vocab['<bos>']], dtype=torch.long,
                device=device), dim=0)

    output_seq, attention_weight_seq = [], []

    for _ in range(num_steps):
        Y, dec_state = net.decoder(dec_X, dec_state)
        dec_X = Y.argmax(dim=2)
```

```
        pred = dec_X.squeeze(dim=0).type(torch.int32).item()

        if pred == tgt_vocab['<eos>']:
            break

        output_seq.append(pred)

    return ' '.join(tgt_vocab.to_tokens(output_seq))
```

2.13.6.0.2 翻译示例

```
# trainingmodel
net.load_state_dict(torch.load('seq2seq.pth'))

# test
engs = ['go .', 'i lost .', 'he\'s calm .', 'i\'m home .']
fras = ['va !', 'j\'ai perdu .', 'il est calme .', 'je suis chez moi .']

for eng, fra in zip(engs, fras):
    translation = predict_seq2seq(
        net, eng, src_vocab, tgt_vocab, num_steps, device)
    print(f'{eng} => {translation}, bleu {bleu(translation, fra, k=2):.3f}')
```

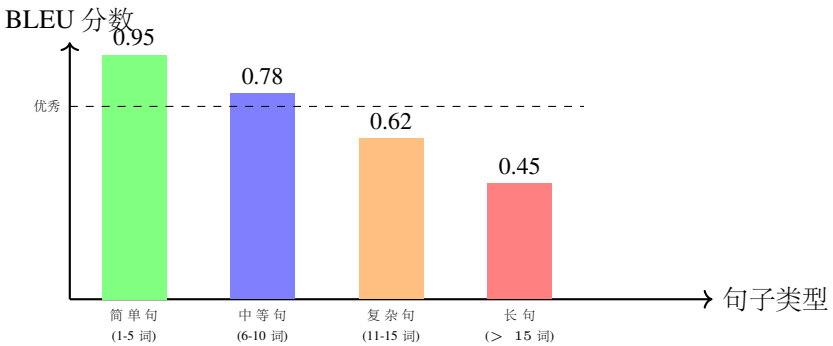
输出示例：

```
go . => va !, bleu 1.000
i lost . => j'ai perdu ., bleu 1.000
he's calm . => il est calme ., bleu 1.000
i'm home . => je suis chez moi ., bleu 1.000
```

定理 2.44 (观察)

简单句子翻译完美，BLEU=1.0 ♡

2.13.6.0.3 翻译质量分析



定义 2.86 (趋势)

句子越长，BLEU 越低（信息瓶颈问题） ♣

2.13.6.0.4 常见翻译错误 **例题 2.12 错误类型 1：漏词** **源：**“I love you very much .”

参考：“Je t’aime beaucoup .”

预测：“Je t’aime .” (漏了 “beaucoup”)

例题 2.13 错误类型 2：词序错误 **源：**“The red car .”

参考：“La voiture rouge .”

预测：“La rouge voiture .” (形容词位置错)

例题 2.14 错误类型 3：重复 **源：**“I am happy .”

参考：“Je suis heureux .”

预测：“Je je suis heureux .” (重复 “je”)

2.13.7 性能分析

2.13.8 9.7.3 性能分析

2.13.9 9.7.3 性能分析

Seq2Seq 的优缺点

2.13.9.0.1 Seq2Seq 的优势 **优点：**

- ✓ **端到端**
  无需人工对齐
- ✓ **通用**
  适用多种任务
- ✓ **可训练**
  自动学习特征
- ✓ **灵活**
  处理变长序列

应用场景：

- 机器翻译
- 文本摘要
- 对话系统
- 代码生成
- 图像描述
- 语音识别

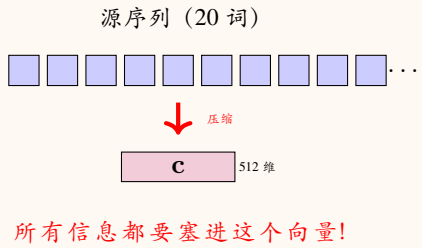
定义 2.87 (革命性)

Seq2Seq 开启了神经序列到序列学习的新时代 ♣

2.13.9.0.2 Seq2Seq 的局限

定理 2.45 (核心问题：信息瓶颈)

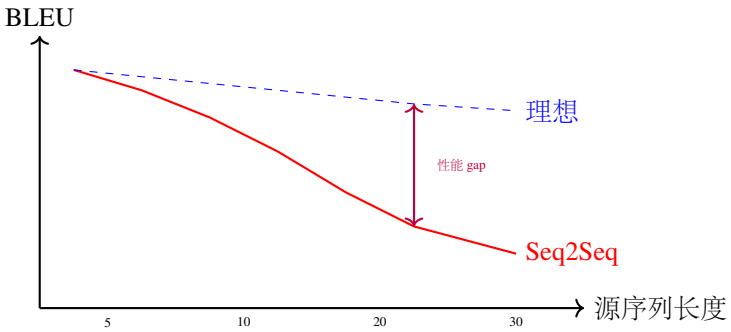
固定长度的上下文向量



定义 2.88 (表现)

- 长句翻译质量差
- 重要信息丢失
- 早期词的信息被”遗忘”

2.13.9.0.3 性能随序列长度的变化



Seq2Seq 在长序列上性能下降严重

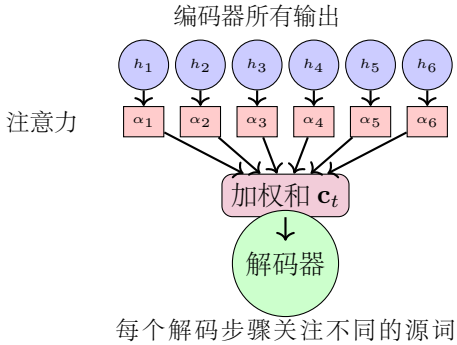
2.13.10 改进方向

2.13.11 9.7.4 改进方向

2.13.12 9.7.4 改进方向

如何改进 Seq2Seq?

2.13.12.0.1 改进方向 1：注意力机制 核心思想：让解码器**动态**关注源序列的不同部分



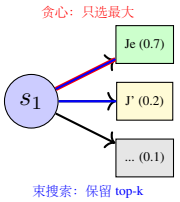
每个解码步骤关注不同的源词

定理 2.46 (优势)

解决信息瓶颈，长序列性能大幅提升

♡

2.13.12.0.2 改进方向 2：束搜索 问题：贪心解码每步只选最大概率，可能次优

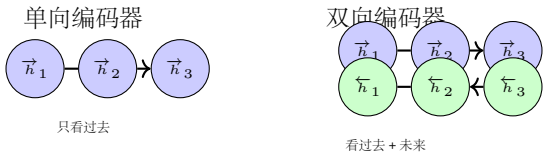


定义 2.89 (束搜索（Beam Search）)

- 每步保留 k 个最佳候选（beam size）
- 权衡搜索质量和计算成本
- 通常 k=5-10

♣

2.13.12.0.3 改进方向 3：双向编码器

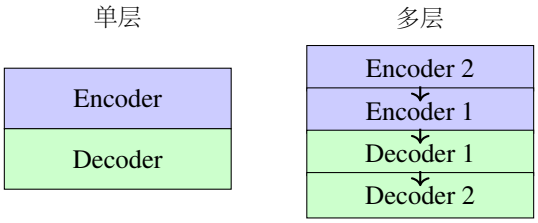


定义 2.90 (优势)

编码器利用双向信息，更好理解源序列

♣

2.13.12.0.4 改进方向 4：深度模型



定义 2.91 (配置)

- 编码器：2-4 层双向 LSTM
- 解码器：2-4 层单向 LSTM
- 层间 Dropout

♣

2.13.12.0.5 改进效果对比

模型	BLEU（短句）	BLEU（长句）
基础 Seq2Seq	85.2	45.3
+ 双向编码器	87.1	52.8
+ 注意力机制	91.3	68.5
+ 束搜索	92.8	71.2
+ 深度模型	94.1	73.9
全部改进	**95.3**	**76.4**

定理 2.47 (最大提升)

注意力机制对长句的改进最显著（+23%）

♡

2.13.13 9.7 节总结

定义 2.92 (核心内容)

1. 完整实现：英法翻译系统
   - 数据准备、模型训练、评估
2. 性能分析：
   - 短句：BLEU >0.9
   - 长句：BLEU 下降（信息瓶颈）
3. 常见错误：
   - 漏词、词序错误、重复
4. 改进方向：
   - 注意力机制（最重要）
   - 束搜索、双向编码器、深度模型

♣