Language Model

训练语言模型，我们需要计算单词的概率， 以及给定前面几个单词后出现某个单词的条件概率。 这些概率本质上就是语言模型的参数。

$P(\text{deep}, \text{learning}, \text{is}, \text{fun}) = P(\text{deep})P(\text{learning} \mid \text{deep})P(\text{is} \mid \text{deep}, \text{learning})P(\text{fun} \mid \text{deep}, \text{learning}, \text{is}).$

拉普拉斯平滑

在所有计数中添加一个小常量。 用n表示训练集中的单词总数，用m表示唯一单词的数量。 此解决方案有助于处理单元素问题.

$$\begin{aligned}
\hat{P}(x) & = \frac{n(x) + \epsilon_1/m}{n + \epsilon_1}, \\
\hat{P}(x' \mid x) & = \frac{n(x, x') + \epsilon_2 \hat{P}(x')}{n(x) + \epsilon_2}, \\
\hat{P}(x'' \mid x,x') & = \frac{n(x, x',x'') + \epsilon_3 \hat{P}(x'')}{n(x, x') + \epsilon_3}.
\end{aligned}$$

以上模型（高情商：过时）

现在LLM不适用了，ε太大了近似1/m，m太小了，n太大了导致概率又小于1

Library function

Read_time_machine：将每一行的非字母字符替换为空格，去除首尾空格，转为小写。上一节定义的。

Vocab：将数据集种单词映射为唯一索引。上一节定义的。

Code

```python
import random
import torch
from d2l import torch as d2l

tokens = d2l.tokenize(d2l.read_time_machine())
corpus = [token for line in tokens for token in line]
vocab = d2l.Vocab(corpus)
print(vocab.token_freqs[:10])

freqs = [freq for token, freq in vocab.token_freqs]
d2l.plot(freqs, xlabel='token: x', ylabel='frequency: n(x)',
         xscale='log', yscale='log')
d2l.plt.show()

# 二元语法
bigram_tokens = [pair for pair in zip(corpus[:-1], corpus[1:])]
bigram_vocab = d2l.Vocab(bigram_tokens)
print(bigram_vocab.token_freqs[:10])

# 三元语法
trigram_tokens = [triple for triple in zip(
    corpus[:-2], corpus[1:-1], corpus[2:])]
trigram_vocab = d2l.Vocab(trigram_tokens)
```

```python
print(trigram_vocab.token_freqs[:10])

bigram_freqs = [freq for token, freq in bigram_vocab.token_freqs]
trigram_freqs = [freq for token, freq in trigram_vocab.token_freqs]
d2l.plot([freqs, bigram_freqs, trigram_freqs], xlabel='token: x',
         ylabel='frequency: n(x)', xscale='log', yscale='log',
         legend=['unigram', 'bigram', 'trigram']
         )
d2l.plt.show()

# 沐神的代码太优秀了，自己照着一步一步自己推才能理解，不讲解。
def seq_data_iter_random(corpus, batch_size, num_steps):
    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 corpus[pos:pos + num_steps]

    num_batches = num_subseqs // batch_size
    for i in range(0, batch_size * num_batches, batch_size):
        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)

my_seq = list(range(35))
for X, Y in seq_data_iter_random(my_seq, batch_size=2, num_steps=5):
```