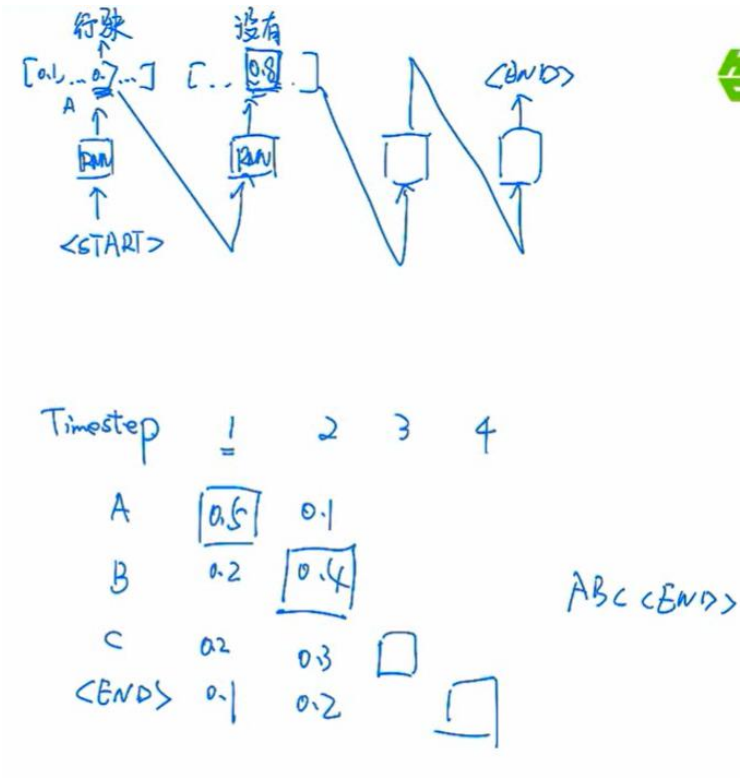
NLG 过程的优化

一、项目的 inference

1. Greedy Search

预测过程中，首先应用到的是贪婪搜索，基本原理是：将预测的结果概率每次取最大的作为生成词，并输入到下一次输入。具体如下图：



Greedy search 原理图

```
In [3]: data

Out[3]: array([[0.1, 0.2, 0.3, 0.4, 0.5],
               [0.5, 0.4, 0.3, 0.2, 0.1],
               [0.1, 0.2, 0.3, 0.4, 0.5],
               [0.5, 0.4, 0.3, 0.2, 0.1],
               [0.1, 0.2, 0.3, 0.4, 0.5],
               [0.5, 0.4, 0.3, 0.2, 0.1],
               [0.1, 0.2, 0.3, 0.4, 0.5],
               [0.5, 0.4, 0.3, 0.2, 0.1],
               [0.1, 0.2, 0.3, 0.4, 0.5],
               [0.5, 0.4, 0.3, 0.2, 0.1]])

In [5]: # greedy decoder
        def greedy_decoder(data):
            # index for largest probability each row
            return [np.argmax(s) for s in data]
```

代码图

贪婪搜索的缺点：局部最优

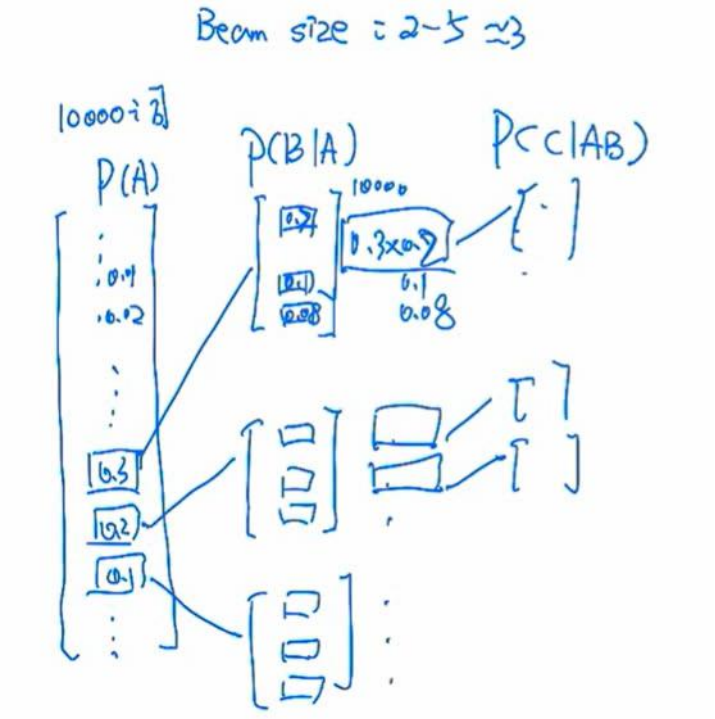
2. Beam search

为了避免局部最优，每一次不取最大的，而是取概率前 k 个，这也是 beam search

的思想。k 的选取很重要，以下有几种方式处理。

2.1 beam_search 原理

假设 beam_size=3，<START>输入后得到概率结果，选取前 3 个，分别将这 3 个作为输入，又分别选出 3 个，此时得到 3*3 个,再从这 9 个选出前 3 个；以此类推。时间复杂度为：O(beam*vocab_size*seq_len)



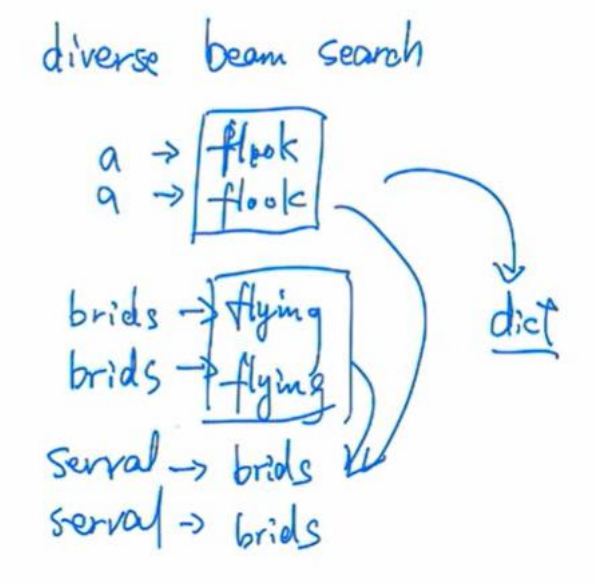
Beam search 原理图

```
In [ ]: # beam search
def beam_search_decoder(data, k):
    sequences = [[list(), 1.0]]
#     print('sequences is ', sequences)
    # walk over each step in sequence
    for row in data:
#         print('row is', row)
        all_candidates = list()
        # expand each current candidate
        for i in range(len(sequences)):
#             print('sequences is', sequences)
#             print('sequences[i] is ', sequences[i])
            seq, score = sequences[i]
            for j in range(len(row)):
                candidate = [seq + [j], score * -np.log(row[j])]
#                 print('score is ',score)
#                 print('row[j] is ', row[j])
                print('candidate is ', candidate)
                all_candidates.append(candidate)
        # order all candidates by score
#             print('all_candidates is ', all_candidates)
        ordered = sorted(all_candidates, key=lambda tup:tup[1])
        # select k best
        sequences = ordered[:k]
    return sequences
```

代码图

缺点：生成文本的多样性比较差（生成序列前面变化不大，只有结尾有些变化，例：40404098、40404079）

-->可以参考 diverse beam search 改进，原理：如下图，a 输入得到 flook，记录起来并进行惩罚，使下一个生成不会生成它。



Diverse beam search 原理图

2.2 代码的设计

理论主要设计有三个部分组成：存储类、每一步的前 k、外部的循环；实际还有注意外部数据集应 copy beam_size 份。

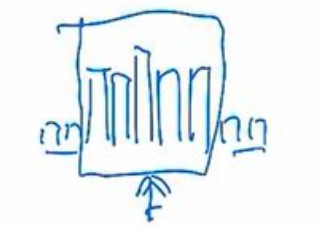
二、Beam Search 的改进

1. 随机采样

Beam_search 依旧避免不了全局最优，我们可以采取随机采样，一般 beam_size 是从 vocab_size 中选前 k 个，改进：先从 vocab_size 随机选 1000 个，再从 1000 个选前 k 个。

2. Top-k

Top-k 先定义一个超参 k1（数量），从 vocab_size 选出前 k1 个，再从 k1 个中进行随机采样 k 个。



3. Top-p (Nucleus sampling)

Top-p 则先定义一个参数 p=0.8(阈值)，对 vocab_size 的概率进行排序，从大到小累加，超过阈值就停止，然后再从中随机选 k 个；可以与 top-k 结合使用。

4. 惩罚重复

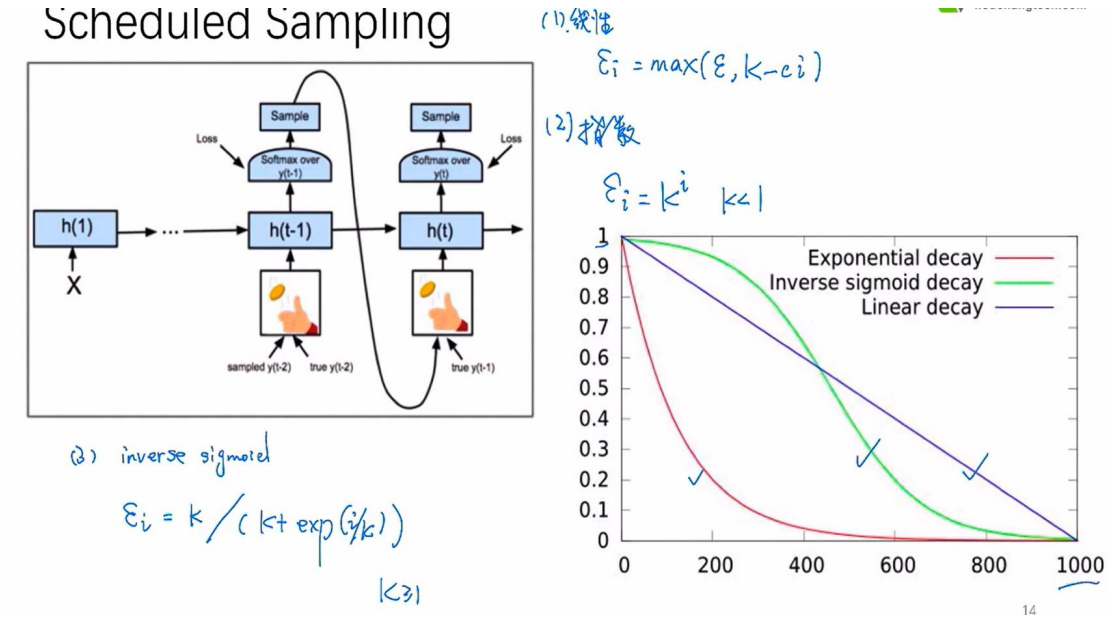
对于已经生成的词进行惩罚，避免生成重复的值，具体详见 transformer/generation_tf_utils 框架。

三、生成问题的训练策略

1、解决 Exposure bias 问题？

- Scheduled Sampling（可参考 openNMT-tf）

  定义 θ in [0,1]，使得 decoder_input= θ *target_label+(1-θ)*predict_label，其中 target_label 为摘要得 label，predict_label 为预测生成的 label；θ 的控制有以下几种方式，根据 epoch 线性衰减、指数衰减、inverse sigmoid 衰减，如下图：



衰减方式图

2、调参的一些建议：

（1）参数初始化：

不宜过大/过小，要适中；

Xavier 初始化，适用于 tanh 激活函数，原理：Xavier 会对每一层进行动态 m_dim 的缩放，保证符合高斯分布（对 relu 激活函数不友好）

Kaiming 初始化，适用于 relu 激活函数（relu 会将一半的值置为 0，影响了分布）

（2）Batch_size 的调参：2**n

Batch_size 越小，更新越快，找最优值越细致（针对性），陷入局部；batch_size 越大，梯度越精确，容易跳出局部区域。

Batch_size 先调到足够大，然后逐渐调小

（3）优化器

Adam or adamw 可以很快跳出局部最优，sgd 与动量很容易陷入局部

先 adam 使达到全局最优附近（经验），再 sgd 或动量

（4）

四、领域的数据集

- CNN dailymail 数据集
- 新浪微博摘要数据集
- nlpcc(中文数据集)
- 单文档摘要

- Gigaword
- LCSTS
- Newsroom
- Xsum

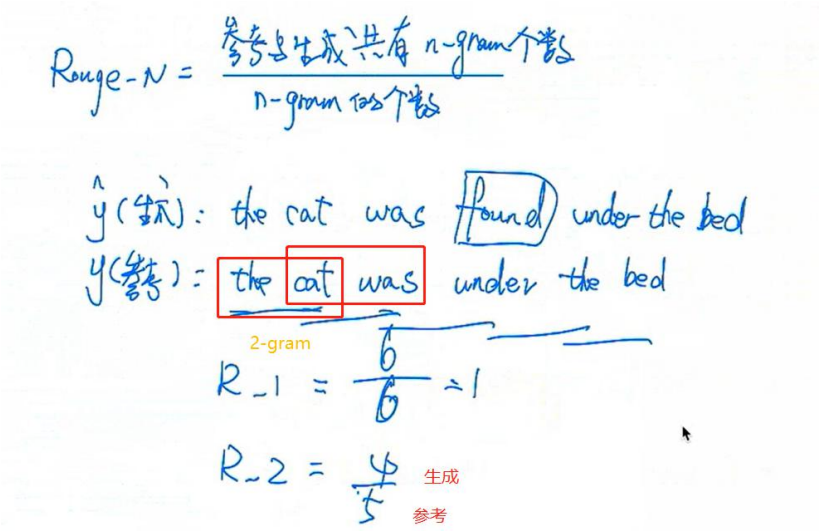
五、数据集刷榜网站

- paperwithcode.com

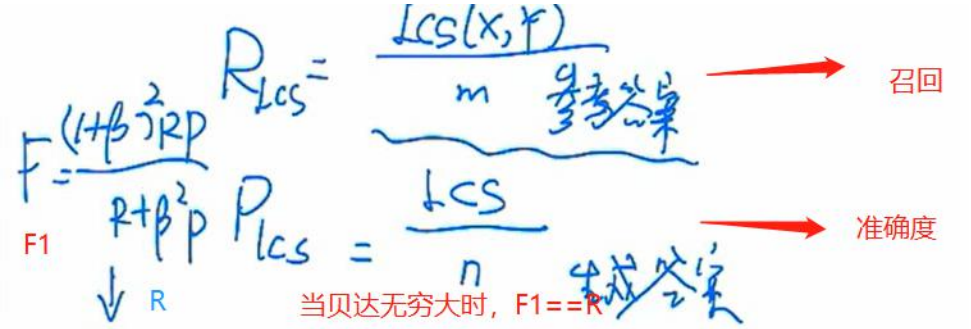
六、评估指标-ROUGE

rouge 全称 recall-oriented understand for gisting evalution；N 是 n-gram 的意思，L 是 lcs,全称 longest common subsequence.最长公共子序列

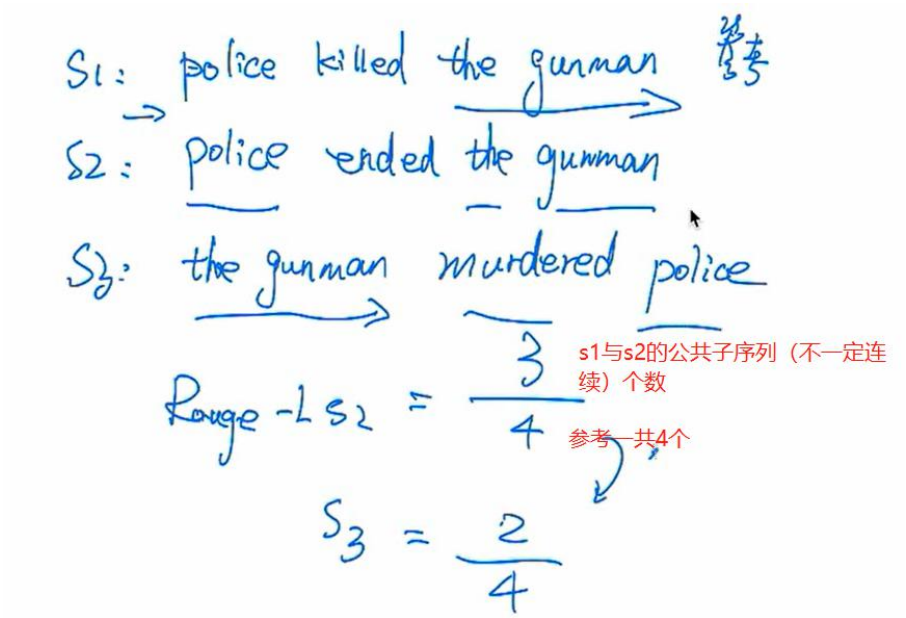
- Rouge-N



- ROUGE-L
  - 公式



  - 例子


S1: police killed the gunman
S2: police ended the gunman
S3: the gunman murdered police
Rouge-L s2 = 3/4
S3 = 2/4
s1与s2的公共子序列（不一定连续）个数
参考一共4个

-