简明条件随机场CRF介绍 | 附带纯Keras实现

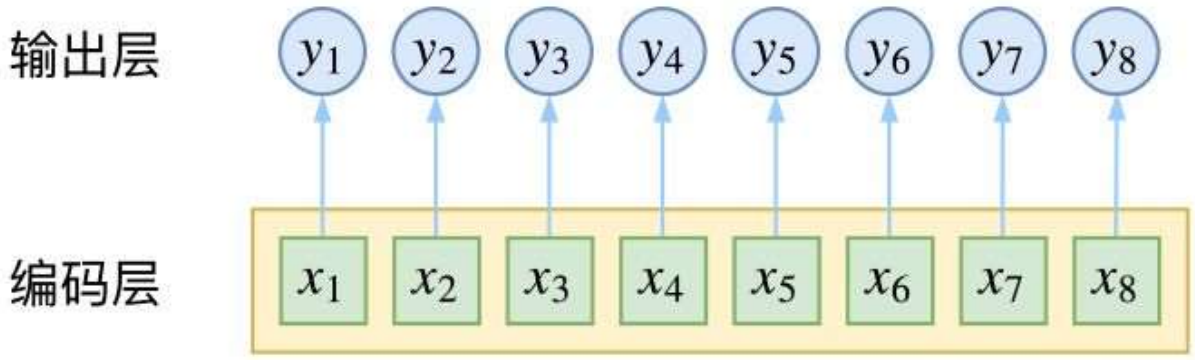
本文是对 CRF 基本原理的一个简明的介绍。当然，“简明”是相对而言中，要想真的弄清楚 CRF，免不了要提及一些公式，如果只关心调用的读者，可以直接移到文末。

图示

按照之前的思路，我们依旧来对比一下普通的逐帧 softmax 和 CRF 的异同。

逐帧softmax

CRF 主要用于序列标注问题，可以简单理解为是**给序列中的每一帧都进行分类**，既然是分类，很自然想到将这个序列用 CNN 或者 RNN 进行编码后，接一个全连接层用 softmax 激活，如下图所示：



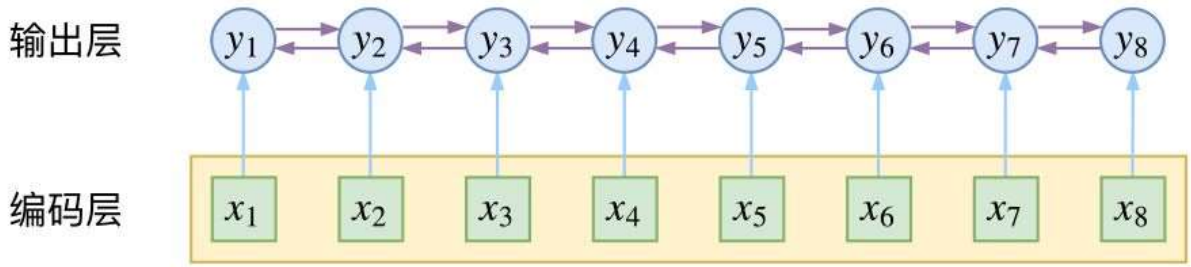
▲ 逐帧softmax并没有直接考虑输出的上下文关联

条件随机场

然而，当我们设计标签时，比如用 s、b、m、e 的 4 个标签来做字标注法的分词，目标输出序列本身会带有一些上下文关联，比如 s 后面就不能接 m 和 e，等等。逐标签 softmax 并没有考虑这种输出层面的上下文关联，所以它意味着把这些关联放到了编码层面，希望模型能自

己学到这些内容，但有时候会“强模型所难”。

而 CRF 则更直接一点，它**将输出层面的关联分离了出来**，这使得模型在学习上更为“从容”：



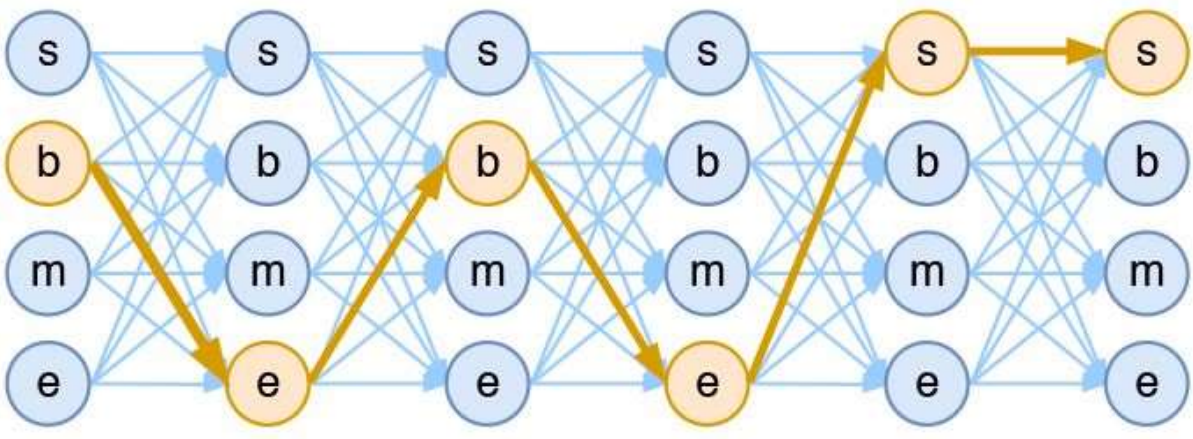
▲ CRF在输出端显式地考虑了上下文关联

数学

当然，如果仅仅是引入输出的关联，还不仅仅是 CRF 的全部，CRF 的真正精巧的地方，是它以路径为单位，考虑的是路径的概率。

模型概要

假如一个输入有 n 帧，每一帧的标签有 k 中可能性，那么理论上就有k^n中不同的输入。我们可以将它用如下的网络图进行简单的可视化。在下图中，每个点代表一个标签的可能性，点之间的连线表示标签之间的关联，而每一种标注结果，都对应着图上的一条完整的路径。



▲ 4tag分词模型中输出网络图

而在序列标注任务中，我们的正确答案是一般是唯一的。比如“今天天气不错”，如果对应的分词结果是“今天/天气/不/错”，那么目标输出序列就是 bebess，除此之外别的路径都不符合要求。

换言之，在序列标注任务中，我们的研究的基本单位应该是路径，我们要做的事情，是从 k^n 条路径选出正确的一条，那就意味着，如果将它视为一个分类问题，那么将是 k^n 类中选一类的分类问题。

这就是逐帧 softmax 和 CRF 的根本不同了：**前者将序列标注看成是 n 个 k 分类问题，后者将序列标注看成是 1 个 k^n 分类问题。**

具体来讲，在 CRF 的序列标注问题中，我们要计算的是条件概率：

$$P(y_1, \dots, y_n|x_1, \dots, x_n) = P(y_1, \dots, y_n|\boldsymbol{x}), \quad \boldsymbol{x} = (x_1, \dots, x_n) \tag{1}$$

为了得到这个概率的估计，CRF 做了两个假设：

假设一：该分布是指数族分布。

这个假设意味着存在函数 f(y1,...,yn;x)，使得：

$$P(y_1, \dots, y_n|\boldsymbol{x}) = \frac{1}{Z(\boldsymbol{x})} \exp\Big(f(y_1, \dots, y_n; \boldsymbol{x})\Big) \tag{2}$$

其中 Z(x) 是归一化因子，因为这个是条件分布，所以归一化因子跟 x 有关。这个 f 函数可以视为一个打分函数，打分函数取指数并归一化后就得到概率分布。

假设二：输出之间的关联仅发生在相邻位置，并且关联是指数加性的。

这个假设意味着 f(y1,...,yn;x) 可以更进一步简化为：

$$f(y_1,\dots,y_n;\boldsymbol{x}) = h(y_1;\boldsymbol{x}) + g(y_1,y_2;\boldsymbol{x}) + h(y_2;\boldsymbol{x}) + \dots + g(y_{n-1},y_n;\boldsymbol{x}) + h(y_n;\boldsymbol{x}) \tag{3}$$

这也就是说，现在我们只需要对每一个标签和每一个相邻标签对分别打分，然后将所有打分结果求和得到总分。

线性链CRF

尽管已经做了大量简化，但一般来说，(3) 式所表示的概率模型还是过于复杂，难以求解。于是考虑到当前深度学习模型中，RNN 或者层叠 CNN 等模型已经能够比较充分捕捉各个 y 与输出 x 的联系，因此，我们不妨考虑函数 g 跟 x 无关，那么：

$$f(y_1,\dots,y_n;\boldsymbol{x}) = h(y_1;\boldsymbol{x}) + g(y_1,y_2) + h(y_2;\boldsymbol{x}) + \dots + g(y_{n-1},y_n) + h(y_n;\boldsymbol{x}) \tag{4}$$

这时候 g 实际上就是一个有限的、待训练的参数矩阵而已，而单标签的打分函数 h(yi;x) 我们可以通过 RNN 或者 CNN 来建模。因此，该模型是可以建立的，其中概率分布变为：

$$P(y_1,\dots,y_n|\boldsymbol{x}) = \frac{1}{Z(\boldsymbol{x})}\exp\left(h(y_1;\boldsymbol{x}) + \sum_{k=1}^{n-1} g(y_k,y_{k+1}) + h(y_{k+1};\boldsymbol{x})\right) \tag{5}$$

这就是线性链 CRF 的概念。

归一化因子

为了训练 CRF 模型，我们用最大似然方法，也就是用：

$$-\log P(y_1, \ldots, y_n|\boldsymbol{x}) \tag{6}$$

作为损失函数，可以算出它等于：

$$-\left(h(y_1;\boldsymbol{x}) + \sum_{k=1}^{n-1} g(y_k, y_{k+1}) + h(y_{k+1};\boldsymbol{x})\right) + \log Z(\boldsymbol{x}) \tag{7}$$

其中第一项是原来概率式的**分子**的对数，它目标的序列的打分，虽然它看上去挺迂回的，但是并不难计算。真正的难度在于**分母**的对数 logZ(x) 这一项。

归一化因子，在物理上也叫配分函数，在这里它需要我们对所有可能的路径的打分进行指数求和，而我们前面已经说到，这样的路径数是指数量级的（k^n），因此直接来算几乎是不可能的。

事实上，**归一化因子难算，几乎是所有概率图模型的公共难题。**幸运的是，在 CRF 模型中，由于我们只考虑了临近标签的联系（马尔可夫假设），因此我们可以递归地算出归一化因子，这使得原来是指数级的计算量降低为线性级别。

具体来说，我们将计算到时刻 t 的归一化因子记为 Zt，并将它分为 k 个部分：

$$Z_t = Z_t^{(1)} + Z_t^{(2)} + \cdots + Z_t^{(k)} \tag{8}$$

其中

$Z_t^{(1)}, \ldots, Z_t^{(k)}$

分别是截止到当前时刻 t 中、以标签 1,...,k 为终点的所有路径的得分指数和。那么，我们可以递归地计算：

$$\begin{aligned} Z_{t+1}^{(1)} &= \left(Z_t^{(1)} G_{11} + Z_t^{(2)} G_{21} + \cdots + Z_t^{(k)} G_{k1}\right) h_{t+1}(1|\boldsymbol{x}) \\ Z_{t+1}^{(2)} &= \left(Z_t^{(1)} G_{12} + Z_t^{(2)} G_{22} + \cdots + Z_t^{(k)} G_{k2}\right) h_{t+1}(2|\boldsymbol{x}) \\ &\vdots \\ Z_{t+1}^{(k)} &= \left(Z_t^{(1)} G_{1k} + Z_t^{(2)} G_{2k} + \cdots + Z_t^{(k)} G_{kk}\right) h_{t+1}(k|\boldsymbol{x}) \end{aligned} \tag{9}$$

它可以简单写为矩阵形式：

$$\boldsymbol{Z}_{t+1} = \boldsymbol{Z}_t \boldsymbol{G} \otimes H(y_{t+1}|\boldsymbol{x}) \tag{10}$$

其中

$Z_t = [Z_t^{(1)}, \ldots, Z_t^{(k)}]$

，而 G 是对 g(yi,yj) 各个元素取指数后的矩阵，即

$G = e^{g(yi,yj)}$

；而

$H(y_{t+1}|x)$

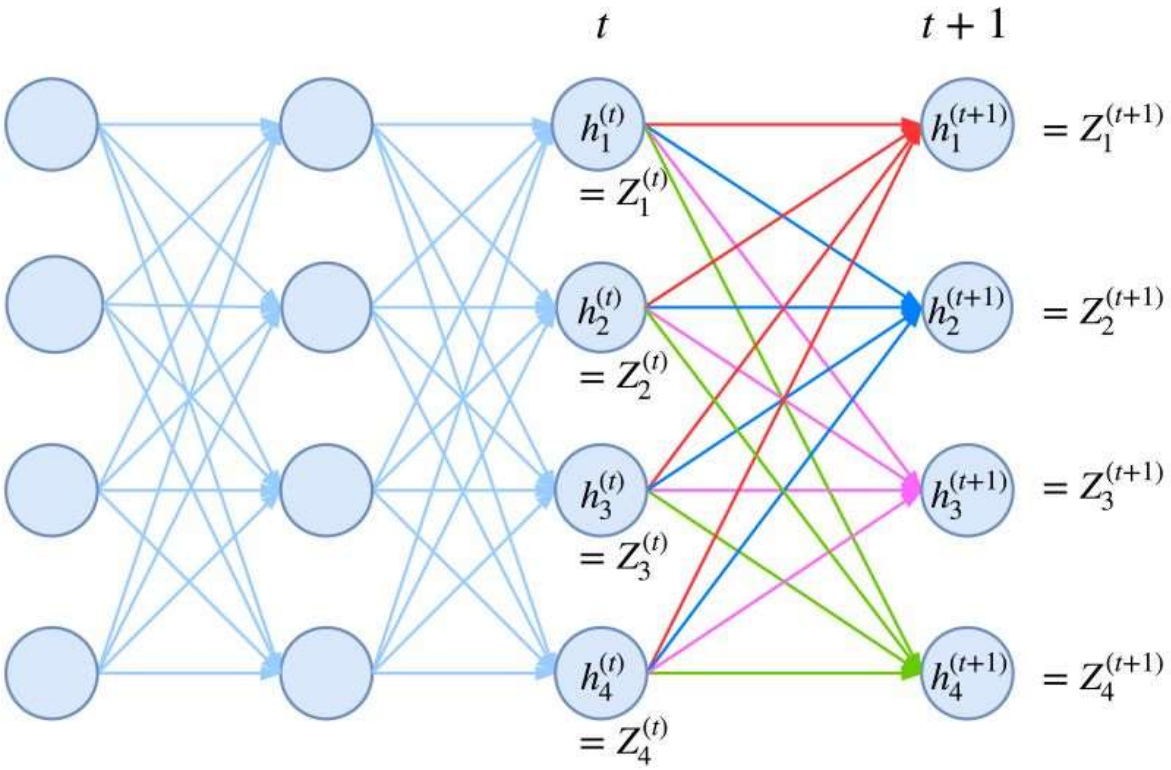
是编码模型

$h(y_{t+1}|x)$

（RNN、CNN等）对位置 t+1 的各个标签的打分的指数，即

$H(y_{t+1}|x) = e^{h(y_{t+1}|x)}$

，也是一个向量。式 (10) 中，ZtG 这一步是矩阵乘法，得到一个向量，而 ⊗ 是两个向量的逐位对应相乘。



▲ 归一化因子的递归计算图示。从t到t+1时刻的计算，包括转移概率和j+1节点本身的概率

如果不熟悉的读者，可能一下子比较难接受 (10) 式。读者可以把 n=1,n=2,n=3 时的归一化因子写出来，试着找它们的递归关系，慢慢地就可以理解 (10) 式了。

动态规划

写出损失函数 −logP(y1,…,yn|x) 后，就可以完成模型的训练了，因为目前的深度学习框架都已经带有自动求导的功能，只要我们能写出可导的 loss，就可以帮我们完成优化过程了。

那么剩下的最后一步，就是模型训练完成后，如何根据输入找出最优路径来。跟前面一样，这也是一个从 k^n 条路径中选最优的问题，而同样地，因为马尔可夫假设的存在，它可以转化为一个动态规划问题，用 viterbi 算法解决，计算量正比于 n。

动态规划在本博客已经出现了多次了，**它的递归思想就是：一条最优路径切成两段，那么每一段都是一条（局部）最优路径。**在本博客右端的搜索框键入“动态规划”，就可以得到很多相关介绍了，所以不再重复了。

实现

经过调试，基于 Keras 框架下，笔者得到了一个线性链 CRF 的简明实现，**这也许是最简短的 CRF 实现了。**这里分享最终的实现并介绍实现要点。

实现要点

前面我们已经说明了，实现 CRF 的困难之处是 −logP(y1,…,yn|x) 的计算，而本质困难是归一化因子部分 Z(x) 的计算，得益于马尔科夫假设，我们得到了递归的 (9) 式或 (10) 式，它们应该已经是一般情况下计算 Z(x) 的计算了。

那么怎么在深度学习框架中实现这种递归计算呢？要注意，从计算图的视角看，这是通过递归的方法定义一个图，而且这个图的长度还不固定。这对于 PyTorch这样的动态图框架应该是不为难的，但是对于TensorFlow或者基于 TensorFlow 的 Keras 就很难操作了（它们是静态图框架）。

不过，并非没有可能，**我们可以用封装好的 RNN 函数来计算。**我们知道，RNN 本质上就是在递归计算：

$$\boldsymbol{h}_{t+1} = f(\boldsymbol{x}_{t+1}, \boldsymbol{h}_t) \tag{11}$$

新版本的 TensorFlow 和 Keras 都已经允许我们自定义 RNN 细胞，这就意味着函数 f 可以自行定义，而后端自动帮我们完成递归计算。**于是我们只需要设计一个 RNN，使得我们要计算的 Z 对应于 RNN 的隐藏向量。**

这就是 CRF 实现中最精致的部分了。

至于剩下的，是一些细节性的，包括：

1. 为了防止溢出，我们通常要取对数，但由于归一化因子是指数求和，所以实际上是 $log(\Sigma_{i=1}^{k} e^{ai})$ 这样的格式，它的计算技巧是：

$$\log\left(\sum_{i=1}^{k} e^{a_i}\right) = A + \log\left(\sum_{i=1}^{k} e^{a_i - A}\right), \quad A = \max\{a_1, \ldots, a_k\}$$

TensorFlow 和 Keras 中都已经封装好了对应的 logsumexp 函数了，直接调用即可；

2. 对于分子（也就是目标序列的得分）的计算技巧，在代码中已经做了注释，主要是通过用“目标序列”点乘“预测序列”来实现取出目标得分；

3. 关于变长输入的 padding 部分如何进行 mask？我觉得在这方面 Keras 做得并不是很好。

为了简单实现这种 mask，我的做法是引入多一个标签，比如原来是 s、b、m、e 四个标签做分词，然后引入第五个标签，比如 x，将 padding 部分的标签都设为 x，然后可以直接在 CRF 损失计算时忽略第五个标签的存在，具体实现请看代码。

代码速览

纯 Keras 实现的 CRF 层，欢迎使用。

```
# -*- coding:utf-8 -*-

from              import
                    as


      CRF
    """纯Keras实现CRF层
    CRF层本质上是一个带训练参数的loss计算层，因此CRF层只用来训练模型，
```

```
    而预测则需要另外建立模型。
    """
    def __init__
        """ignore_last_label：定义要不要忽略最后一个标签，起到mask的效果
        """
        self.ignore_last_label = 1 if                    else 0
        super(CRF, self).__init__(**kwargs)
    def build
                                    -1
                                        'crf_trans'

                                                'glorot_uniform'
                                              True
    def log_norm_step
        """递归计算归一化因子
        要点：1、递归计算；2、用logsumexp避免溢出。
        技巧：通过expand_dims来对齐张量。
        """
        states = K.expand_dims(states[0    2  # (batch_size, output_dim, 1)
        trans = K.expand_dims(self.trans, 0  # (1, output_dim, output_dim)
        output = K.logsumexp(states+trans, 1  # (batch_size, output_dim)
        return
    def path_score
        """计算目标路径的相对概率（还没有归一化）
        要点：逐标签得分，加上转移概率得分。
        技巧：用“预测”点乘“目标”的方法抽取出目标路径的得分。
        """
        point_score = K.sum(K.sum(inputs*labels, 2    1           True  # 逐标
签得分
        labels1 = K.expand_dims(labels[:, :-1    3
                                        1    2
                                  # 两个错位labels，负责从转移矩阵中抽取目标转移
得分
        trans = K.expand_dims(K.expand_dims(self.trans, 0    0
                                              2 3    1           True
        return                              # 两部分得分之和
    def call                  # CRF本身不改变输出，它只是一个loss
        return
    def loss                            # 目标y_pred需要是one hot形式
        mask = 1          1  -1  if                          else None
        y_true,y_pred =
y_true[:,:,:self.num_labels],y_pred[:,:,:self.num_labels]
        init_states = [y_pred[:,0    # 初始状态
        log_norm,_,_ = K.rnn(self.log_norm_step, y_pred[:,1
           # 计算Z向量（对数）
        log_norm = K.logsumexp(log_norm, 1           True  # 计算Z（对数）
        path_score = self.path_score(y_pred, y_true) # 计算分子（对数）
        return                           # 即log(分子/分母)
    def accuracy                            # 训练过程中显示逐帧准确率的函数，排除了
mask的影响
        mask = 1            -1  if                          else None
        y_true,y_pred =
```

```
y_true[:,:,:self.num_labels],y_pred[:,:,:self.num_labels]
        isequal = K.equal(K.argmax(y_true, 2                    2
                                  'float32'
        if          None
            return
        else
            return
```

除去注释和 accuracy 的代码，真正的 CRF 的代码量也就 30 行左右，可以说跟哪个框架比较都称得上是简明的 CRF 实现了。

用纯 Keras 实现一些复杂的模型，是一件颇有意思的事情。目前仅在 TensorFlow 后端测试通过，理论上兼容 Theano、CNTK 后端，但可能要自行微调。

使用案例

我的 Github 中还附带了一个使用 CNN+CRF 实现的中文分词的例子，用的是 Bakeoff 2005 语料，例子是一个完整的分词实现，包括 viterbi 算法、分词输出等。

Github地址：https://github.com/bojone/crf/