HW3-P1：实现RNN各组件

2024年8月14日 星期三 15:54

1

0 概述

作业目标

- 如何从头开始编写实现一个 RNN 的代码
  - 如何实现 RNN 单元，并使用 RNN 单元构建一个简单的 RNN 分类器
  - 如何实现 GRU 单元，并使用 GRU 单元构建一个字符预测器
  - 如何实现 CTC 损失函数的前向和反向传播（CTC 损失是一种特定于递归神经网络的准则，其功能类似于我们为 HW2P2 使用的交叉熵损失）
- 如何解码模型输出的概率以获得可理解的输出
  - 如何实现贪婪搜索（Greedy Search）
  - 如何实现束搜索（Beam Search）

文件结构

```
HW3P1
├── mytorch
│   ├── nn
│   │   ├── linear.py
│   │   ├── activation.py
│   │   ├── loss.py
│   │   ├── gru_cell.py
│   │   └── rnn_cell.py
├── models
│   ├── char_predictor.py
│   └── rnn_classifier.py
├── MCQ
│   └── mcq.py
├── CTC
│   ├── CTC.py
│   └── CTCDecoding.py
├── autograder
│   ├── data
│   ├── test_ctc_decoding_toy.py
│   ├── test_ctc_decoding.py
│   ├── test_ctc_toy.py
│   ├── test_ctc.py
│   ├── test_gru_toy.py
│   ├── test_gru.py
│   ├── test_mc.py
│   ├── test_rnn_toy.py
│   ├── test_rnn.py
│   ├── test.py
│   ├── toy_runner.py
│   └── runner.py
└── create_tarball.sh
```

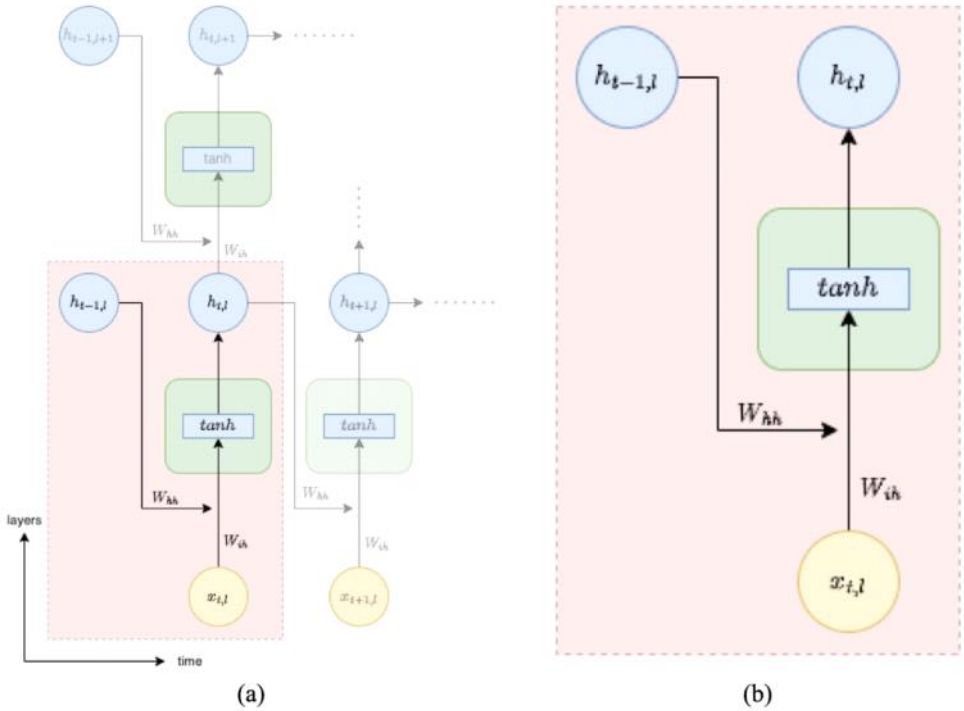
测试代码方法

- 单模块测试：python3 autograder/runner.py test_name，其中 test_name 可以是 rnn, gru, ctc, beam_search，根据你要测试的部分选择相应的名称。
- 完整测试：python3 autograder/runner.py

TASK	SCORE
Multiple Choice Questions	5
RNN Forward	5
RNN Backward	5
RNN Classifier	10
GRU Forward	5
GRU Backward	15
GRU Inference	10
Extend Sequence with Blank	5
Posterior Probability	5
CTC Forward	10
CTC Backward	5
Greedy Search	5
Beam Search	15
TOTAL SCORE	100

1 RNN

RNN结构示意图如下，注意：RNN是将当前输入 x_t 和前一时间步的隐藏状态 h_{t-1} 分别先通过不同的权重矩阵线性变换，然后将结果相加，再经过激活函数。这个跟LSTM先拼接再经过线性变换和激活函数不同。



1.1 RNN 前向传播

对于RNN单元，前向传播公式为：

$$h_{t,l} = tanh(W_{ih} \cdot x_t + b_{ih} + W_{hh} \cdot h_{t-1,l} + b_{hh})$$

- 其中：
  - W_{ih}：输入到隐藏状态的权重矩阵，维度为 $H_{out} \times H_{in}$。
  - W_{hh}：隐藏状态到隐藏状态的权重矩阵，维度为 $H_{out} \times H_{out}$。
  - b_{ih}：输入到隐藏状态的偏置，维度为 H_{out}。
  - b_{hh}：隐藏状态到隐藏状态的偏置，维度为 H_{out}。
  - x_t：当前时间步的输入，维度为 $N \times H_{in}$。
  - $h_{t-1,l}$：前一个时间步的隐藏状态，维度为 $N \times H_{out}$。
  - $h_{t,l}$：当前时间步的隐藏状态，维度为 $N \times H_{out}$。

根据这个公式，前向传播的步骤如下：

1. **计算当前输入的线性变换**：

   $$W_{ih} \cdot x_t + b_{ih}$$

   这里的 $W_{ih} \cdot x_t$ 表示矩阵乘法，而 b_{ih} 是一个偏置向量。

2. **计算前一隐藏状态的线性变换**：

   $$W_{hh} \cdot h_{t-1,l} + b_{hh}$$

   这里的 $W_{hh} \cdot h_{t-1,l}$ 表示矩阵乘法，而 b_{hh} 是另一个偏置向量。

3. **将两个线性变换结果相加并应用激活函数**：

   $$h_{t,l} = tanh(W_{ih} \cdot x_t + b_{ih} + W_{hh} \cdot h_{t-1,l} + b_{hh})$$

1.2 RNN 反向传播

假设损失函数 L 对于 $h_{t,l}$ 的梯度已经通过反向传播传递过来，梯度是 $\delta = \frac{\partial L}{\partial h_{t,l}}$。（代码中是delta）

我们要计算的梯度包括：

1. $\frac{\partial L}{\partial W_{ih}}$
2. $\frac{\partial L}{\partial W_{hh}}$
3. $\frac{\partial L}{\partial b_{ih}}$
4. $\frac{\partial L}{\partial b_{hh}}$
5. $\frac{\partial L}{\partial x_t}$

6. $\frac{\partial L}{\partial h_{t-1,l}}$

1. 通过激活函数的梯度

因为隐藏状态 $h_{t,l}$ 是通过激活函数 $tanh$ 计算的，我们需要通过链式法则计算损失函数相对于 $z_{t,l}$ 的梯度，其中 $z_{t,l}$ 是激活函数 $tanh$ 的输入：

$$z_{t,l} = W_{ih} \cdot x_t + b_{ih} + W_{hh} \cdot h_{t-1,l} + b_{hh}$$

因为 $h_{t,l} = tanh(z_{t,l})$，所以

$$\frac{\partial h_{t,l}}{\partial z_{t,l}} = 1 - tanh^2(z_{t,l}) = 1 - h_{t,l}^2$$

根据链式法则，我们可以计算损失函数对 $z_{t,l}$ 的梯度：

$$\frac{\partial L}{\partial z_{t,l}} = \frac{\partial L}{\partial h_{t,l}} \cdot \frac{\partial h_{t,l}}{\partial z_{t,l}} = \delta \cdot (1 - h_{t,l}^2)$$

2. 计算相对于权重和偏置的梯度

接下来，我们使用链式法则计算相对于权重 W_{ih}、W_{hh} 以及偏置 b_{ih}、b_{hh} 的梯度。

- **对** W_{ih} 的梯度：

$$\frac{\partial L}{\partial W_{ih}} = \frac{\partial L}{\partial z_{t,l}} \cdot \frac{\partial z_{t,l}}{\partial W_{ih}} = (\delta \cdot (1 - h_{t,l}^2)) \cdot x_t^T$$

- **对** W_{hh} 的梯度：

$$\frac{\partial L}{\partial W_{hh}} = \frac{\partial L}{\partial z_{t,l}} \cdot \frac{\partial z_{t,l}}{\partial W_{hh}} = (\delta \cdot (1 - h_{t,l}^2)) \cdot h_{t-1,l}^T$$

- **对** b_{ih} 的梯度：

$$\frac{\partial L}{\partial b_{ih}} = \frac{\partial L}{\partial z_{t,l}} \cdot \frac{\partial z_{t,l}}{\partial b_{ih}} = \delta \cdot (1 - h_{t,l}^2)$$

- **对** b_{hh} 的梯度：

$$\frac{\partial L}{\partial b_{hh}} = \frac{\partial L}{\partial z_{t,l}} \cdot \frac{\partial z_{t,l}}{\partial b_{hh}} = \delta \cdot (1 - h_{t,l}^2)$$

3. 计算相对于前一时间步隐藏状态和输入的梯度

最后，我们还需要计算损失函数相对于当前时间步输入 x_t 和前一时间步隐藏状态 $h_{t-1,l}$ 的梯度：

- **对** x_t 的梯度：

$$\frac{\partial L}{\partial x_t} = \frac{\partial L}{\partial z_{t,l}} \cdot W_{ih}^T = (\delta \cdot (1 - h_{t,l}^2)) \cdot W_{ih}^T$$

- **对** $h_{t-1,l}$ 的梯度：

$$\frac{\partial L}{\partial h_{t-1,l}} = \frac{\partial L}{\partial z_{t,l}} \cdot W_{hh}^T = (\delta \cdot (1 - h_{t,l}^2)) \cdot W_{hh}^T$$

为什么计算梯度时，结果中基本都有转置T符号?

- 转置的符号出现是因为我们在计算梯度时需要保持矩阵的维度一致性。矩阵微积分中的链式法则要求在求导过程中，维度必须匹配。如果不使用转置，维度将不匹配，从而无法正确计算梯度。

特别注意：RNN 结构中的自循环特性

RNN 的关键特性在于其时间步的循环依赖，即当前时间步的隐藏状态依赖于前一个时间步的隐藏状态。在前向传播时，RNN 会随着时间步（通常记为 `t`）逐步展开，每个时间步使用相同的参数（如权重 `W_ih` 和 `W_hh`），进行更新隐藏状态。

因此，当进行反向传播时，我们需要逐步从最后一个时间步回溯到第一个时间步，并计算每个时间步的梯度。这些梯度是相对于同一组参数的，所以在反向传播中，我们会逐步累积每个时间步的梯度。

具体操作流程，假设我们有 T 个时间步：

1. **前向传播**：
   - 从时间步 `1` 到 `T`，RNN 使用相同的参数集 `W_ih`，`W_hh`，逐步计算每个时间步的隐藏状态。
- **反向传播**：
  - 反向传播从时间步 `T` 回溯到 `1`，在每个时间步计算对应的梯度。
  - 对于每个时间步 `t`，我们计算相对于参数 `W_ih`，`W_hh` 的梯度，并使用 `+=` 操作将其累积起来。

由于所有时间步共享相同的参数，所以我们需要将每个时间步的梯度累积在一起，最后一次性更新参数。这就是为什么在反向传播的过程中需要使用 `+=` 操作。

1.3 RNN 代码实现

```
import numpy as np
from nn.activation import *

class RNNCell(object):
    """RNN 单元类"""

    def __init__(self, input_size, hidden_size):
        """
        初始化 RNNCell 类

        参数:
        ----
        input_size: 输入的维度大小
        hidden_size: 隐藏层的维度大小
        """

        self.input_size = input_size
        self.hidden_size = hidden_size

        # 激活函数为 Tanh
        self.activation = Tanh()

        # 隐藏层维度和输入维度
        h = self.hidden_size
        d = self.input_size
```

```
        # 初始化权重和偏置
        self.W_ih = np.random.randn(h, d)  # 输入到隐藏层的权重矩阵
        self.W_hh = np.random.randn(h, h)  # 隐藏层到隐藏层的权重矩阵
        self.b_ih = np.random.randn(h)     # 输入到隐藏层的偏置
        self.b_hh = np.random.randn(h)     # 隐藏层到隐藏层的偏置

        # 初始化梯度
        self.dW_ih = np.zeros((h, d))
        self.dW_hh = np.zeros((h, h))
        self.db_ih = np.zeros(h)
        self.db_hh = np.zeros(h)

    def init_weights(self, W_ih, W_hh, b_ih, b_hh):
        """
        初始化权重和偏置

        参数:
        ----
        W_ih: 输入到隐藏层的权重矩阵
        W_hh: 隐藏层到隐藏层的权重矩阵
        b_ih: 输入到隐藏层的偏置
        b_hh: 隐藏层到隐藏层的偏置
        """
        self.W_ih = W_ih
        self.W_hh = W_hh
        self.b_ih = b_ih
        self.b_hh = b_hh

    def zero_grad(self):
        """
        将所有的梯度初始化为零
        """
        d = self.input_size
        h = self.hidden_size
        self.dW_ih = np.zeros((h, d))
        self.dW_hh = np.zeros((h, h))
        self.db_ih = np.zeros(h)
        self.db_hh = np.zeros(h)

    def __call__(self, x, h_prev_t):
        """
        实现类的可调用性，直接调用 forward 方法

        参数:
        ----
        x: 当前时间步的输入
        h_prev_t: 上一个时间步的隐藏状态

        返回:
```

```
        ----
        当前时间步的隐藏状态
        """
        return self.forward(x, h_prev_t)

    def forward(self, x, h_prev_t):
        """
        RNN 单元的前向传播（单个时间步）

        参数:
        ----
        x: (batch_size, input_size) 当前时间步的输入
        h_prev_t: (batch_size, hidden_size) 上一个时间步的隐藏状态

        返回:
        ----
        h_t: (batch_size, hidden_size) 当前时间步的隐藏状态
        """

        # 计算隐藏状态，公式 h_t = tanh(W_ih * x + b_ih + W_hh * h_prev_t + b_hh)
        z = (np.matmul(self.W_ih, np.transpose(x)) + np.expand_dims(self.b_ih, axis=1)
+
             np.matmul(self.W_hh, np.transpose(h_prev_t)) + np.expand_dims(self.b_hh,
axis=1))
        h_t = self.activation.forward(np.transpose(z))  # 对 z 应用tanh激活函数
        return h_t

    def backward(self, delta, h_t, h_prev_l, h_prev_t):
        """
        RNN 单元的反向传播（单个时间步）

        参数:
        ----
        delta: (batch_size, hidden_size) 当前隐藏层的梯度
        h_t: (batch_size, hidden_size) 当前时间步和当前层的隐藏状态
        h_prev_l: (batch_size, input_size) 当前时间步和前一层的隐藏状态
        h_prev_t: (batch_size, hidden_size) 上一个时间步和当前层的隐藏状态

        返回:
        ----
        dx: (batch_size, input_size) 相对于当前时间步和前一层的输入梯度
        dh_prev_t: (batch_size, hidden_size) 相对于上一个时间步和当前层的输入梯度
        """
        batch_size = delta.shape[0]
        # 0) 反向传播通过 tanh 激活函数。
        dz = self.activation.backward(delta, state=h_t)

        # 1) 计算权重和偏置的梯度
        # 使用 += 的是因为RNN结构在前向传播时会自循环很多次，假设自循环5次，
```

```
        # 那么反向传播时就需要调用5次backward，梯度值累加5次后才去更新一次参数。
        self.dW_ih += ((dz.T @ h_prev_l)/ batch_size)
        self.dW_hh += ((dz.T @ h_prev_t) / batch_size)
        self.db_ih += (np.sum(dz, axis=0) / batch_size)
        self.db_hh += (np.sum(dz, axis=0) / batch_size)

        # 2) 计算 dx 和 dh_prev_t
        dx = dz @ self.W_ih
        dh_prev_t = dz @ self.W_hh

        # 3) 返回 dx 和 dh_prev_t
        return dx, dh_prev_t
```

2 使用 RNN 构建简单的音素分类器

本小节是实现一个多层次、时间序列长度可变的RNN音素分类器。

2.1 前向传播

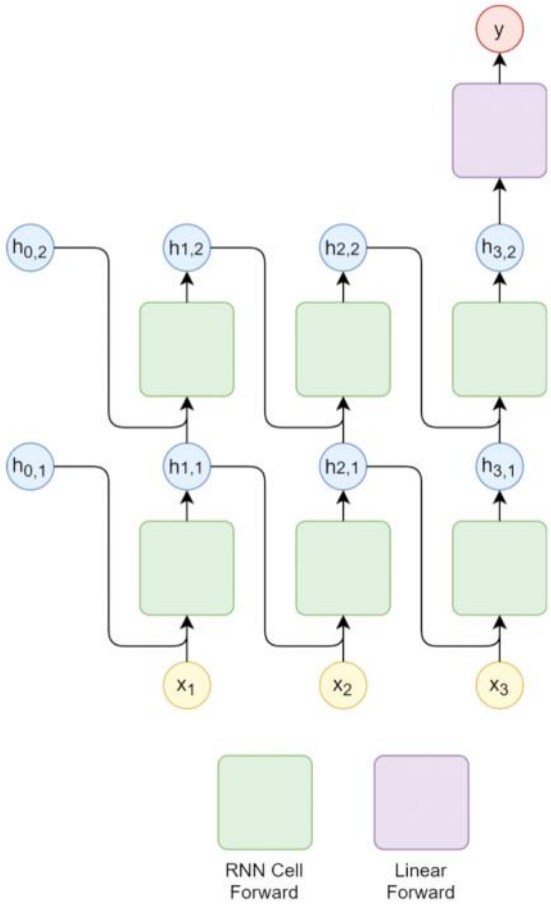


Figure 4: The forward computation flow for the RNN.

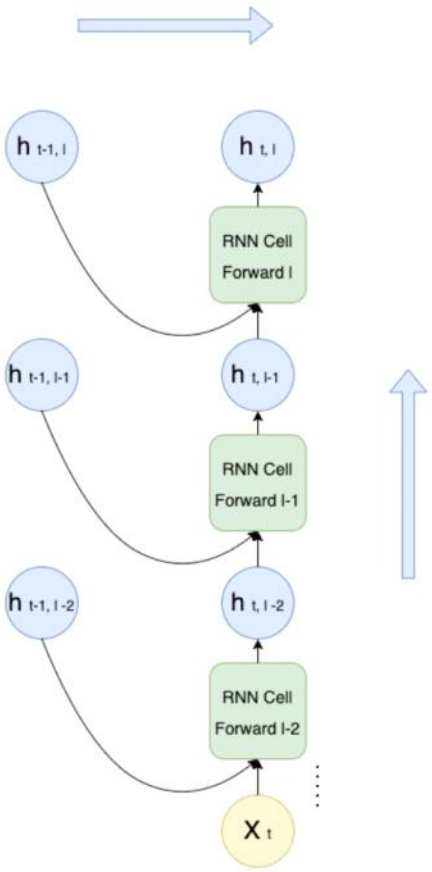


Figure 5: The forward computation flow for the RNN at time step t.

2.2 反向传播

这道题可能是本次作业中在概念上最难的一道题。不过，如果你按照这个伪代码去做，并试着理解其中的含义，你就能顺利完成作业。

输入:

- `delta`: 损失函数相对于最后一个时间步输出的梯度

输出:

- `dx`: 损失函数相对于输入序列的梯度

伪代码

```text
1. for t ← seq_len - 1 to 0 do
2.     for l ← num_layers - 1 to 0 do
3.         // 根据当前层次，从 hiddens 或 x 中获取 h_prev_l
           （注意，hiddens 中包含一个额外的初始隐藏状态）
4.         // 使用 dh 和 hiddens 来获取反向传播方法的其他参数
           （注意，hiddens 中包含一个额外的初始隐藏状态）
5.         // 使用 RNN 单元的反向传播结果更新 dh
6.         if l ≠ 0 then
7.             // 如果当前不是第 1 层，你需要将 dx 添加到 l-1 层的 dh 中
8.         end
9.     end
10. end
11. // 将 dh 除以 batch size 进行归一化，因为初始隐藏状态也被视为网络的参数
12. return dh;  // 返回损失函数相对于初始隐藏状态的梯度
```

1. **时间步逆序遍历**:
   - 算法从时间步 `seq_len-1` 开始，逐步回溯到第一个时间步 `0`。这样做是因为我们需要从最后的时间步开始逐步向前进行反向传播。
2. **层次逆序遍历**:
   - 在每个时间步中，算法从最深层（`num_layers-1`）开始，逐层向上进行反向传播。这是因为在 RNN 中，较深层的梯度依赖于上一层的输出。
3. **获取前一时间步的隐藏状态**:
   - 对于每一层，在反向传播时，我们需要获取前一时间步的隐藏状态 `h_prev_l`。如果是最底层（第0层），`h_prev_l` 取自输入 `x`；如果是其他层，`h_prev_l` 取自 `hiddens` 中保存的前一层的隐藏状态。
4. **反向传播更新梯度**:
   - 使用当前层的 `dh` 和 `hiddens` 来计算反向传播所需的其他参数，并更新当前层的 `dh`。
5. **梯度累积**:
   - 如果当前层不是最底层（第0层），我们需要将从当前层得到的 `dx` 累积到上一层的 `dh` 中。这一步是为了将误差传播到前面的层次。
6. **归一化梯度**:
   - 由于初始隐藏状态被视为网络的参数，因此我们需要对最终的 `dh` 进行归一化处理，具体方法是将其除以批次大小（`batch size`）。

7. **返回初始隐藏状态的梯度**:
    - 最后，算法返回的是损失函数相对于初始隐藏状态的梯度。这个梯度可以用于进一步的反向传播，或者作为训练的结果用于参数更新。

2.3 代码

```
import numpy as np
import sys

# 添加自定义模块路径
sys.path.append("mytorch")
from rnn_cell import *  # 导入自定义的 RNNCell 类
from nn.linear import *  # 导入自定义的 Linear 类

class RNNPhonemeClassifier(object):
    """RNN 音素分类器类"""

    def __init__(self, input_size, hidden_size, output_size, num_layers=2):
        """
        初始化分类器

        参数:
        ---------
        input_size: int
            输入特征的维度
        hidden_size: int
            隐藏层单元的数量（隐藏状态的维度）
        output_size: int
            输出特征的维度
        num_layers: int, 默认值为2
            RNN 层的数量
        """
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers

        # 初始化 RNN 层。第一层的输入维度是 input_size，后续层的输入维度是 hidden_size
        self.rnn = [
            RNNCell(input_size, hidden_size) if i == 0
                else RNNCell(hidden_size, hidden_size)
                    for i in range(num_layers)
        ]
        self.output_layer = Linear(hidden_size, output_size)  # 初始化输出层

        # 存储每个时间步的隐藏状态，形状为 [(seq_len+1) * (num_layers, batch_size,
hidden_size)]
        self.hiddens = []
```

```
    def init_weights(self, rnn_weights, linear_weights):
        """
        初始化权重

        参数:
        ---------
        rnn_weights: list
            包含多个 RNN 层权重的列表，每一层包括 [W_ih, W_hh, b_ih, b_hh]

        linear_weights: list
            包含线性层权重的列表 [W, b]
        """
        # 初始化每一层 RNN 的权重
        for i, rnn_cell in enumerate(self.rnn):
            rnn_cell.init_weights(*rnn_weights[i]) # * 用于将一个列表或元组中的元素解包为函数的多个参数。
        # 初始化输出层的权重
        self.output_layer.W = linear_weights[0]
        self.output_layer.b = linear_weights[1].reshape(-1, 1)

    def __call__(self, x, h_0=None):
        return self.forward(x, h_0)

    def forward(self, x, h_0=None):
        """
        RNN 前向传播，多层、多时间步。

        参数:
        ---------
        x: np.array
            输入数据，形状为 (batch_size, seq_len, input_size)

        h_0: np.array，可选
            初始隐藏状态，形状为 (num_layers, batch_size, hidden_size)。如果未指定，则默认为全零

        返回:
        ---------
        logits: np.array
            输出的 logits，形状为 (batch_size, output_size)
        """
        # 获取批次大小和序列长度，初始化隐藏状态向量
        batch_size, seq_len = x.shape[0], x.shape[1]
        if h_0 is None:
            hidden = np.zeros((self.num_layers, batch_size, self.hidden_size), dtype=float)
        else:
            hidden = h_0
```

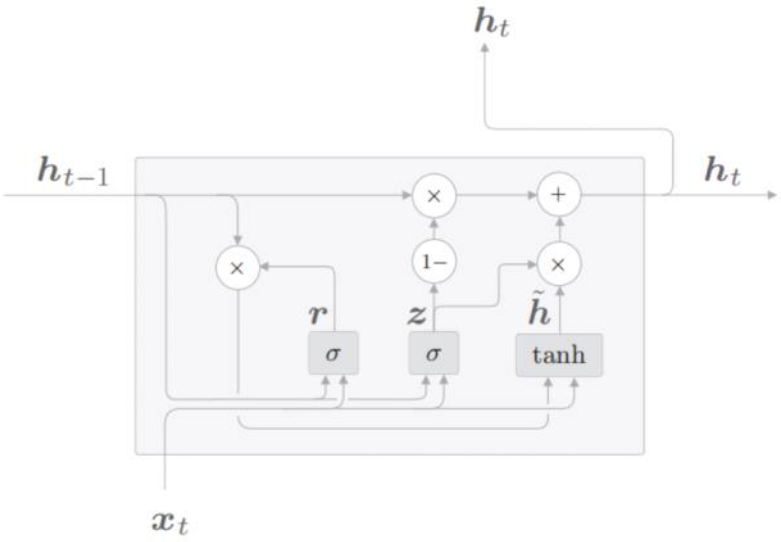
```
        # 保存输入数据，并将初始隐藏状态添加到 hiddens 列表中
        self.x = x
        self.hiddens.append(hidden.copy())
        logits = None

        # 遍历序列中的每个时间步
        for i in range(seq_len):
            hidden_prev = hidden  # 保留前一个时间步的隐藏状态
            # 遍历每一层 RNN
            for l, rnn_cell in enumerate(self.rnn):
                if l == 0:
                    hidden[l,:,:] = rnn_cell(x[:, i, :], hidden_prev[l,:,:])  # 第1层使用输入数据
                else:
                    hidden[l,:,:] = rnn_cell(hidden[l-1,:,:], hidden_prev[l,:,:])  # 其余层使用上一层的隐藏状态
            self.hiddens.append(hidden.copy())  # 将当前时间步的隐藏状态添加到 hiddens 列表中

        # 最后一个时间步的隐藏状态通过线性层计算输出
        logits = self.output_layer(hidden[-1,:,:])

        return logits  # 返回 logits 作为模型输出



    def backward(self, delta):
        """
        RNN 反向传播，通过时间的反向传播（BPTT）。

        参数:
        ---------
        delta: np.array
            损失函数相对于最后一个时间步输出的梯度，形状为 (batch_size, hidden_size)

        返回:
        ---------
        dh_0: np.array
            损失函数相对于初始隐藏状态的梯度，形状为 (num_layers, batch_size, hidden_size)
        """
        # 初始化
        batch_size, seq_len = self.x.shape[0], self.x.shape[1]
        dh = np.zeros((self.num_layers, batch_size, self.hidden_size), dtype=float)

        # 从最后的输出层开始反向传播
        # dh[-1] 取出了 dh 的最后一层，形状为 (batch_size, self.hidden_size)
        dh[-1] = self.output_layer.backward(delta)


        """
```

```
        * 注意：
        * 更详细的伪代码可能会在讲义幻灯片中出现，且在文档中有可视化的描述。
        * 小心由于实现决策导致的 off by 1 错误（例如索引偏移一位的错误）。

        伪代码：
        * 以时间顺序的逆序遍历（从 seq_len-1 到 0）
            * 以层次的逆序遍历（从 num_layers-1 到 0）
                * 根据当前层次从 hiddens 或 x 中获取 h_prev_l
                    （注意，hiddens 中包含一个额外的初始隐藏状态）
                * 使用 dh 和 hiddens 来获取反向传播方法的其他参数
                    （注意，hiddens 中包含一个额外的初始隐藏状态）
                * 使用 RNN 单元的反向传播结果更新 dh
                * 如果当前不是第一层，则需要将 dx 添加到第 l-1 层的梯度中。

        * 由于初始隐藏状态也被视为网络的参数，因此需要将 dh 除以 batch_size 进行归一化（即除以批次大小）。

        提示：在某些地方可能需要使用 `+=` 操作。思考后再编写代码。
        """

        # 反向遍历时间步，从后向前
        for i in range(seq_len-1, -1, -1):
            # 反向遍历层次，从最后一层向前
            for l in range(self.num_layers-1, -1, -1):
                if l == 0:
                    h_prev_l = self.x[:, i, :]  # 如果是第1层，使用输入数据作为 h_prev_l
                else:
                    h_prev_l = self.hiddens[i+1][l-1,:,:]  # 其余层使用上一层的隐藏状态
                h_t = self.hiddens[i+1][l,:,:]  # 当前时间步的隐藏状态
                h_prev_t = self.hiddens[i][l,:,:]  # 前一个时间步的隐藏状态
                dx, dh_prev_t = self.rnn[l].backward(dh[l,:,:], h_t, h_prev_l, h_prev_t)
# 执行 RNN 单元的反向传播
                dh[l,:,:] = dh_prev_t  # 更新当前层的梯度
                if l > 0:
                    dh[l-1,:,:] += dx  # 如果不是第1层，将 dx 加到上一层的梯度中

        return dh / batch_size  # 返回相对于初始隐藏状态的平均梯度
```

3 GRU

在标准的 RNN 中，矩阵的长乘积会导致长期梯度消失（即降为零）或爆炸（即趋于无穷大）。为解决这一问题，最早提出的方法之一是 LSTM（长短期记忆网络）。GRU（门控递归单元）是 LSTM 的一种变体，参数较少，性能相当，计算速度明显更快。GRU 可用于多种任务，如光学字符识别和使用对话文本对光谱图进行语音识别。在本节中，您将基本了解 GRU 单元的前向和后向传递是如何工作的。

3.1 GRU前向传播



GRU计算图，σ节点和tanh节点有专用的权重，节点内部进行仿射变换（“1−”节点输入x，输出1 − x）

$$
\begin{aligned}
\mathbf{r}_t &= \sigma(\mathbf{W}_{rx} \cdot \mathbf{x}_t + \mathbf{b}_{rx} + \mathbf{W}_{rh} \cdot \mathbf{h}_{t-1} + \mathbf{b}_{rh}) \\
\mathbf{z}_t &= \sigma(\mathbf{W}_{zx} \cdot \mathbf{x}_t + \mathbf{b}_{zx} + \mathbf{W}_{zh} \cdot \mathbf{h}_{t-1} + \mathbf{b}_{zh}) \\
\mathbf{n}_t &= tanh(\mathbf{W}_{nx} \cdot \mathbf{x}_t + \mathbf{b}_{nx} + \mathbf{r}_t \odot (\mathbf{W}_{nh} \cdot \mathbf{h}_{t-1} + \mathbf{b}_{nh})) \\
\mathbf{h}_t &= (1 - \mathbf{z}_t) \odot \mathbf{n}_t + \mathbf{z}_t \odot \mathbf{h}_{t-1}
\end{aligned}
$$

- 更新门 z 决定了隐藏状态如何在当前时间步和前一时间步之间进行平衡。
- 重置门 r 决定了前一时间步的隐藏状态在计算候选隐藏状态时的影响。

3.2 GRU反向传播

目标是计算相对于各个参数的梯度，包括：

- **相对于输入** x_t 的梯度 `dx`
- **相对于前一时间步隐藏状态** h_{t-1} 的梯度 `dh_prev_t`
- **相对于各权重和偏置** 的梯度： `dWrx` , `dWzx` , `dWnx` , `dWrh` , `dWzh` , `dWnh` , `dbrx` , `dbzx` , `dbnx` , `dbrh` , `dbzh` , `dbnh`

反向传播的具体步骤

反向传播从损失函数相对于最后时间步的输出 $\delta = \frac{\partial L}{\partial h_t}$ 开始。我们从最后的隐藏状态 h_t 开始向前计算梯度。

1 对 h_t 的梯度

从公式 $h_t = (1 - z_t) \odot \tilde{h}_t + z_t \odot h_{t-1}$ 开始：

- 对 z_t 的梯度：

$$\frac{\partial L}{\partial z_t} = \delta \odot (h_{t-1} - \tilde{h}_t)$$

- 对 $\tilde{h}_t$ 的梯度：

$$\frac{\partial L}{\partial \tilde{h}_t} = \delta \odot (1 - z_t)$$

- 对 h_{t-1} 的梯度：

$$\frac{\partial L}{\partial h_{t-1}} = \delta \odot z_t$$

2 对 $\tilde{h}_t$ 的梯度

从公式 $\tilde{h}_t = tanh(W_{nx} \cdot x_t + b_{nx} + r_t \odot (W_{nh} \cdot h_{t-1} + b_{nh}))$ 开始：

- 对 W_{nx} 和 b_{nx} 的梯度：

$$\frac{\partial L}{\partial W_{nx}} = \frac{\partial L}{\partial \tilde{h}_t} \cdot \tanh'(\tilde{h}_t) \cdot x_t^T$$

$$\frac{\partial L}{\partial b_{nx}} = \frac{\partial L}{\partial \tilde{h}_t} \cdot \tanh'(\tilde{h}_t)$$

- 对 r_t 的梯度：

$$\frac{\partial L}{\partial r_t} = \frac{\partial L}{\partial \tilde{h}_t} \cdot \tanh'(\tilde{h}_t) \odot (W_{nh} \cdot h_{t-1} + b_{nh})$$

- 对 W_{nh} 和 b_{nh} 的梯度：

$$\frac{\partial L}{\partial W_{nh}} = \frac{\partial L}{\partial \tilde{h}_t} \cdot \tanh'(\tilde{h}_t) \cdot r_t \cdot h_{t-1}^T$$

$$\frac{\partial L}{\partial b_{nh}} = \frac{\partial L}{\partial \tilde{h}_t} \cdot \tanh'(\tilde{h}_t) \cdot r_t$$

3 对 z_t 和 r_t 的梯度

- 对 W_{zx} 和 b_{zx} 的梯度：

$$\frac{\partial L}{\partial W_{zx}} = \frac{\partial L}{\partial z_t} \cdot \sigma'(z_t) \cdot x_t^T$$

$$\frac{\partial L}{\partial b_{zx}} = \frac{\partial L}{\partial z_t} \cdot \sigma'(z_t)$$

- 对 W_{zh} 和 b_{zh} 的梯度：

$$\frac{\partial L}{\partial W_{zh}} = \frac{\partial L}{\partial z_t} \cdot \sigma'(z_t) \cdot h_{t-1}^T$$

$$\frac{\partial L}{\partial b_{zh}} = \frac{\partial L}{\partial z_t} \cdot \sigma'(z_t)$$

- 对 W_{rx} 和 b_{rx} 的梯度：

$$\frac{\partial L}{\partial W_{rx}} = \frac{\partial L}{\partial r_t} \cdot \sigma'(r_t) \cdot x_t^T$$

$$\frac{\partial L}{\partial b_{rx}} = \frac{\partial L}{\partial r_t} \cdot \sigma'(r_t)$$

- 对 W_{rh} 和 b_{rh} 的梯度：

$$\frac{\partial L}{\partial W_{rh}} = \frac{\partial L}{\partial r_t} \cdot \sigma'(r_t) \cdot h_{t-1}^T$$

$$\frac{\partial L}{\partial b_{rh}} = \frac{\partial L}{\partial r_t} \cdot \sigma'(r_t)$$

4. 计算最终的输入梯度

- 对 x_t 的梯度：

$$\frac{\partial L}{\partial x_t} = \text{结合所有涉及到 } x_t \text{ 的梯度""\""}$$

- 对 h_{t-1} 的梯度：

$$\frac{\partial L}{\partial h_{t-1}} = \text{结合所有涉及到 } h_{t-1} \text{ 的梯度""\""}$$

5. 注意事项

- **梯度累加**: 在整个反向传播过程中，我们通常需要将各个时间步累积的梯度加总，来更新最终的梯度。
- **矩阵运算一致性**: 确保所有矩阵运算中的形状和维度是一致的，以避免计算错误。

3.3 GRU代码

```
import numpy as np
from nn.activation import *

class GRUCell(object):
    """GRU 单元类。"""

    def __init__(self, input_size, hidden_size):
        """
        初始化 GRU 单元的参数。

        参数:
        ---------
        input_size: int
            输入的维度大小。
        hidden_size: int
            隐藏层的维度大小。
        """
        self.d = input_size  # 输入的维度大小
        self.h = hidden_size  # 隐藏层的维度大小
        h = self.h
        d = self.d
        self.x_t = 0  # 初始化输入

        # GRU 单元的权重矩阵，使用随机数初始化
        self.Wrx = np.random.randn(h, d)  # 重置门的输入到隐藏层的权重
        self.Wzx = np.random.randn(h, d)  # 更新门的输入到隐藏层的权重
        self.Wnx = np.random.randn(h, d)  # 候选隐藏状态的输入到隐藏层的权重

        self.Wrh = np.random.randn(h, h)  # 重置门的隐藏层到隐藏层的权重
        self.Wzh = np.random.randn(h, h)  # 更新门的隐藏层到隐藏层的权重
        self.Wnh = np.random.randn(h, h)  # 候选隐藏状态的隐藏层到隐藏层的权重
```

```
        # GRU 单元的偏置，使用随机数初始化
        self.brx = np.random.randn(h)
        self.bzx = np.random.randn(h)
        self.bnx = np.random.randn(h)

        self.brh = np.random.randn(h)
        self.bzh = np.random.randn(h)
        self.bnh = np.random.randn(h)

        # 初始化各个权重和偏置的梯度为0
        self.dWrx = np.zeros((h, d))
        self.dWzx = np.zeros((h, d))
        self.dWnx = np.zeros((h, d))

        self.dWrh = np.zeros((h, h))
        self.dWzh = np.zeros((h, h))
        self.dWnh = np.zeros((h, h))

        self.dbrx = np.zeros((h))
        self.dbzx = np.zeros((h))
        self.dbnx = np.zeros((h))

        self.dbrh = np.zeros((h))
        self.dbzh = np.zeros((h))
        self.dbnh = np.zeros((h))

        # 激活函数
        self.r_act = Sigmoid()  # 重置门的激活函数
        self.z_act = Sigmoid()  # 更新门的激活函数
        self.h_act = Tanh()     # 候选隐藏状态的激活函数

        # 定义其他变量，用于存储前向传播的结果以便反向传播时使用

    def init_weights(self, Wrx, Wzx, Wnx, Wrh, Wzh, Wnh, brx, bzx, bnx, brh, bzh, bnh):
        """
        初始化 GRU 单元的权重和偏置。

        参数:
        ---------
        Wrx, Wzx, Wnx, Wrh, Wzh, Wnh: np.array
            对应的权重矩阵。
        brx, bzx, bnx, brh, bzh, bnh: np.array
            对应的偏置向量。
        """
        self.Wrx = Wrx
        self.Wzx = Wzx
        self.Wnx = Wnx
        self.Wrh = Wrh
        self.Wzh = Wzh
        self.Wnh = Wnh
```

```
        self.brx = brx
        self.bzx = bzx
        self.bnx = bnx
        self.brh = brh
        self.bzh = bzh
        self.bnh = bnh

    def __call__(self, x, h_prev_t):
        # 使对象实例可以像函数一样调用，实际调用的是 forward 方法
        return self.forward(x, h_prev_t)

    def forward(self, x, h_prev_t):
        """
        GRU 单元的前向传播。

        参数:
        ---------
        x: np.array, 形状为 (input_dim)
            当前时间步的输入。
        h_prev_t: np.array, 形状为 (hidden_dim)
            前一个时间步的隐藏状态。

        返回:
        ---------
        h_t: np.array, 形状为 (hidden_dim)
            当前时间步的隐藏状态。
        """
        self.x = x  # 存储输入
        self.hidden = h_prev_t  # 存储前一时间步的隐藏状态

        # 计算重置门
        self.r = self.r_act.forward(self.Wrx @ x + self.brx + self.Wrh @ h_prev_t +
self.brh)
        # 计算更新门
        self.z = self.z_act.forward(self.Wzx @ x + self.bzx + self.Wzh @ h_prev_t +
self.bzh)
        # 计算候选隐藏状态
        self.n = self.h_act.forward(self.Wnx @ x + self.bnx + self.r * (self.Wnh @
h_prev_t + self.bnh))
        # 计算当前时间步的隐藏状态
        h_t = (1 - self.z) * self.n + self.z * h_prev_t

        # 确保输入和隐藏状态的维度正确
        assert self.x.shape == (self.d,)
        assert self.hidden.shape == (self.h,)

        assert self.r.shape == (self.h,)
        assert self.z.shape == (self.h,)
        assert self.n.shape == (self.h,)
        assert h_t.shape == (self.h,)  # h_t 是 GRU 单元的最终输出
```

```
        self.h_t = h_t  # 存储当前时间步的隐藏状态
        return h_t  # 返回当前时间步的隐藏状态

    def backward(self, delta):
        """
        GRU 单元的反向传播。

        计算各参数的梯度，并返回相对于输入 xt 和 ht 的梯度。

        参数:
        ---------
        delta: np.array, 形状为 (hidden_dim)
            损失函数相对于当前时间步的输出以及下一个时间步的梯度之和。

        返回:
        ---------
        dx: np.array, 形状为 (input_dim)
            损失函数相对于输入 x 的梯度。

        dh_prev_t: np.array, 形状为 (hidden_dim)
            损失函数相对于前一时间步隐藏状态 h 的梯度。
        """
        # 计算更新门 z 的梯度
        dz = delta * (-self.n + self.hidden)
        # 计算候选隐藏状态 n 的梯度
        dn = delta * (1 - self.z)

        # 计算 tanh 激活函数的反向传播
        dtanh = self.h_act.backward(dn, state=self.n)
        self.dWnx = np.expand_dims(dtanh, axis=1) @ np.expand_dims(self.x, axis=1).T  # 计算 Wnx 的梯度
        self.dbnx = dtanh  # 计算 bnx 的梯度
        dr = dtanh * (self.Wnh @ self.hidden + self.bnh)  # 计算重置门 r 的梯度
        self.dWnh = np.expand_dims(dtanh, axis=1) * np.expand_dims(self.r, axis=1) @ np.expand_dims(self.hidden, axis=1).T  # 计算 Wnh 的梯度
        self.dbnh = dtanh * self.r  # 计算 bnh 的梯度

        # 计算 sigmoid 激活函数的反向传播
        dsigz = self.z_act.backward(dz)
        self.dWzx = np.expand_dims(dsigz, axis=1) @ np.expand_dims(self.x, axis=1).T  # 计算 Wzx 的梯度
        self.dbzx = dsigz  # 计算 bzx 的梯度
        self.dWzh = np.expand_dims(dsigz, axis=1) * np.expand_dims(self.hidden, axis=1).T  # 计算 Wzh 的梯度
        self.dbzh = dsigz  # 计算 bzh 的梯度

        # 计算 sigmoid 激活函数的反向传播（重置门 r）
        dsigr = self.r_act.backward(dr)
```

```
        self.dWrx = np.expand_dims(dsigr, axis=1) @ np.expand_dims(self.x, axis=1).T  # 计算 Wrx 的梯度
        self.dbrx = dsigr  # 计算 brx 的梯度
        self.dWrh = np.expand_dims(dsigr, axis=1) * np.expand_dims(self.hidden, axis=1).T  # 计算 Wrh 的梯度
        self.dbrh = dsigr  # 计算 brh 的梯度

        # 计算输入 x 的梯度
        dx = np.squeeze((np.expand_dims(dtanh, axis=1).T @ self.Wnx + np.expand_dims(dsigz, axis=1).T @ self.Wzx + np.expand_dims(dsigr, axis=1).T @ self.Wrx).T)
        # 计算前一时间步隐藏状态 h 的梯度
        dh_prev_t = delta * self.z + dtanh * self.r @ self.Wnh + dsigz @ self.Wzh + dsigr @ self.Wrh

        # 确保梯度的维度正确
        assert dx.shape == (self.d,)
        assert dh_prev_t.shape == (self.h,)

        return dx, dh_prev_t  # 返回输入 x 和前一时间步隐藏状态 h 的梯度
```

4 使用 GRU 构建一个字符预测器

4.1 任务概述

在 `models/char_predictor.py` 文件中，使用之前实现的 `GRUCell` 和一个线性层（`Linear` layer）来组成一个神经网络。这个神经网络将对输入序列进行展开（unroll），以每个时间步生成一个 logits 输出。

与之前 RNN 音素分类器任务的区别

1. **仅进行推理（前向传播）**：在这个网络中，只需要进行推理，不涉及训练过程。
2. **只有一层**：网络结构仅包含一层 GRUCell 和一层线性层。

因此，在 `CharacterPredictor` 类中的前向传播方法可以非常简洁，通常只需要2-3行代码即可完成。而推理函数 `inference` 也可以在不到10行代码内完成。

4.2 任务要求

你需要完成以下步骤：

1. **初始化 CharacterPredictor 类**：在类的 `__init__` 方法中初始化 GRU 单元和线性层。
    - `input_dim`：GRUCell 的输入维度。
    - `hidden_dim`：GRUCell 的隐藏层维度，该维度的输出将传递到线性层。

    - `num_classes`：类别的数量，也就是线性层的输出维度（对应输出的 logits）。
- **实现前向传播**：
    - `forward` 方法应该处理从 GRUCell 到线性层的前向传播。
    - 线性层（在代码中引用为 `self.projection`）仅仅是一个从隐藏状态到输出状态的线性变换。
- **完成推理函数 `inference`**：
    - 输入：
        - `net`： `CharacterPredictor` 类的一个实例。
        - `inputs`：一个包含输入序列（形状为 `(seq_len, feature_dim)`）的张量。
    - 输出：
        - `logits`：形状为 `(seq_len, num_classes)` 的 logits 序列。

4.3 实现步骤

- **初始化 `CharacterPredictor` 类**：
    - 在 `__init__` 方法中，创建一个 GRUCell 实例和一个 Linear 层实例，并将它们与 `input_dim`，`hidden_dim` 和 `num_classes` 连接起来。
- **实现 `forward` 方法**：
    - 接受一个输入序列和一个初始隐藏状态。
    - 使用 GRUCell 处理输入序列，并将输出传递到线性层以生成 logits。
- **实现 `inference` 方法**：
    - 传入一个 `CharacterPredictor` 类的实例和输入序列。
    - 调用 `forward` 方法并展开（unroll）整个序列，将每个时间步的输出（logits）收集起来，最终返回完整的 logits 序列。

4.4 总结

- 该任务主要是将之前实现的 GRUCell 与线性层组合起来，以完成字符预测任务的推理部分。
- 推理部分仅需进行前向传播，不涉及训练，重点是理解如何通过 GRUCell 处理序列数据，并将其输出传递到线性层以获得最终的预测结果。

4.5 代码

```
import numpy as np
import sys

# 添加自定义模块路径
sys.path.append("mytorch")
from gru_cell import *  # 导入自定义的 GRUCell 类
```

```
from nn.linear import *  # 导入自定义的 Linear 类

class CharacterPredictor(object):
    """CharacterPredictor 类。

    这个类实现了一个神经网络，可以处理输入序列的一个时间步。
    你只需要实现这个类的 forward 方法。
    这个类主要用于测试你实现的 GRU Cell 在作为 GRU 使用时是否正确。
    """

    def __init__(self, input_dim, hidden_dim, num_classes):
        super(CharacterPredictor, self).__init__()
        """网络由一个 GRU Cell 和一个线性层组成。"""
        self.gru = GRUCell(input_dim, hidden_dim)  # 初始化 GRU Cell，输入维度为
input_dim，隐藏层维度为 hidden_dim
        self.projection = Linear(hidden_dim, num_classes)  # 初始化线性层，输入维度为
hidden_dim，输出维度为 num_classes
        self.num_classes = num_classes  # 保存类别数
        self.hidden_dim = hidden_dim  # 保存隐藏层维度
        self.projection.W = np.random.rand(num_classes, hidden_dim)  # 随机初始化线性层的
权重

    def init_rnn_weights(
        self, Wrx, Wzx, Wnx, Wrh, Wzh, Wnh, brx, bzx, bnx, brh, bzh, bnh
    ):
        """初始化 GRU 的权重，不需要修改这个函数。"""
        self.gru.init_weights(
            Wrx, Wzx, Wnx, Wrh, Wzh, Wnh, brx, bzx, bnx, brh, bzh, bnh
        )

    def __call__(self, x, h):
        # 使对象实例可以像函数一样调用，实际调用的是 forward 方法
        return self.forward(x, h)

    def forward(self, x, h):
        """CharacterPredictor 的前向传播。

        处理输入序列的一个时间步。

        参数
        -----
        x: np.array，形状为 (feature_dim)
            当前时间步的输入。

        h: np.array，形状为 (hidden_dim)
            前一个时间步的隐藏状态。

        返回
        -------
```

```
        logits: np.array, 形状为 (num_classes)
            当前时间步的 logits 输出。

        hnext: np.array, 形状为 (hidden_dim)
            当前时间步的隐藏状态。

        """
        hnext = self.gru(x, h)  # 通过 GRU Cell 计算当前时间步的隐藏状态
        # self.projection 期望输入的形状为 (batch_size, input_dimension)，因此需要将 hnext
重塑为 (1, -1)
        logits = self.projection(hnext.reshape(1,-1))  # 通过线性层计算 logits
        logits = logits.reshape(-1,)  # 将 logits 重塑为 (num_classes)
        return logits, hnext  # 返回 logits 和当前时间步的隐藏状态 hnext


def inference(net, inputs):
    """CharacterPredictor 的推理函数。

    使用上面定义的类的一个实例，对输入序列进行推理，生成每个时间步的 logits 输出。

    参数
    -----
    net:
        CharacterPredictor 类的一个实例。

    inputs: np.array, 形状为 (seq_len, feature_dim)
            输入序列，每个时间步的输入维度为 feature_dim。

    返回
    -------
    logits: np.array, 形状为 (seq_len, num_classes)
            每个时间步的 logits 输出。

    """
    seq_len, feature_dim = inputs.shape  # 获取序列长度和输入维度
    logits = np.zeros((seq_len, net.num_classes))  # 初始化 logits 输出矩阵
    h = np.zeros(net.hidden_dim)  # 初始化隐藏状态为全零
    for i in range(seq_len):  # 遍历序列中的每个时间步
        logits[i, :], h = net(inputs[i, :], h)  # 对每个时间步进行推理，并更新隐藏状态

    return logits  # 返回所有时间步的 logits 输出
```

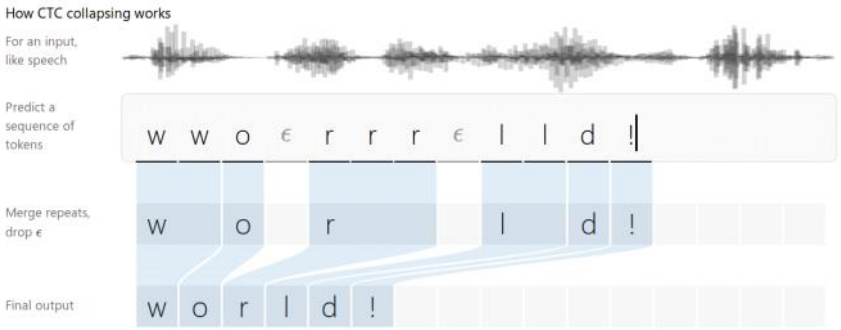
5 CTC

5.1 什么是CTC

连接时序分类（Connectionist Temporal Classification，CTC）是一种专为序列标注任务设计的损失函数和解码算法，主要用于解决输入序列和目标序列长度不对齐的问题。CTC尤其适用于在没有明确对齐标注的情况下训练循环神经网络（RNNs），如LSTM或GRU，常用于语音识别、手写字符识别等领域。

5.2 问题背景

在许多序列学习任务中，输入序列的长度与输出序列的长度通常是不相等的。例如，在语音识别中，一个较长的音频输入片段可能需要映射到一个较短的文本序列。传统的监督学习方法需要明确的对齐标注，即在每个时间步明确标注输出对应的目标值，这在很多实际应用中是难以获得的。（具体可去看HW1P2-音素标签）



暂时不理解没关系，看下面这个举例。

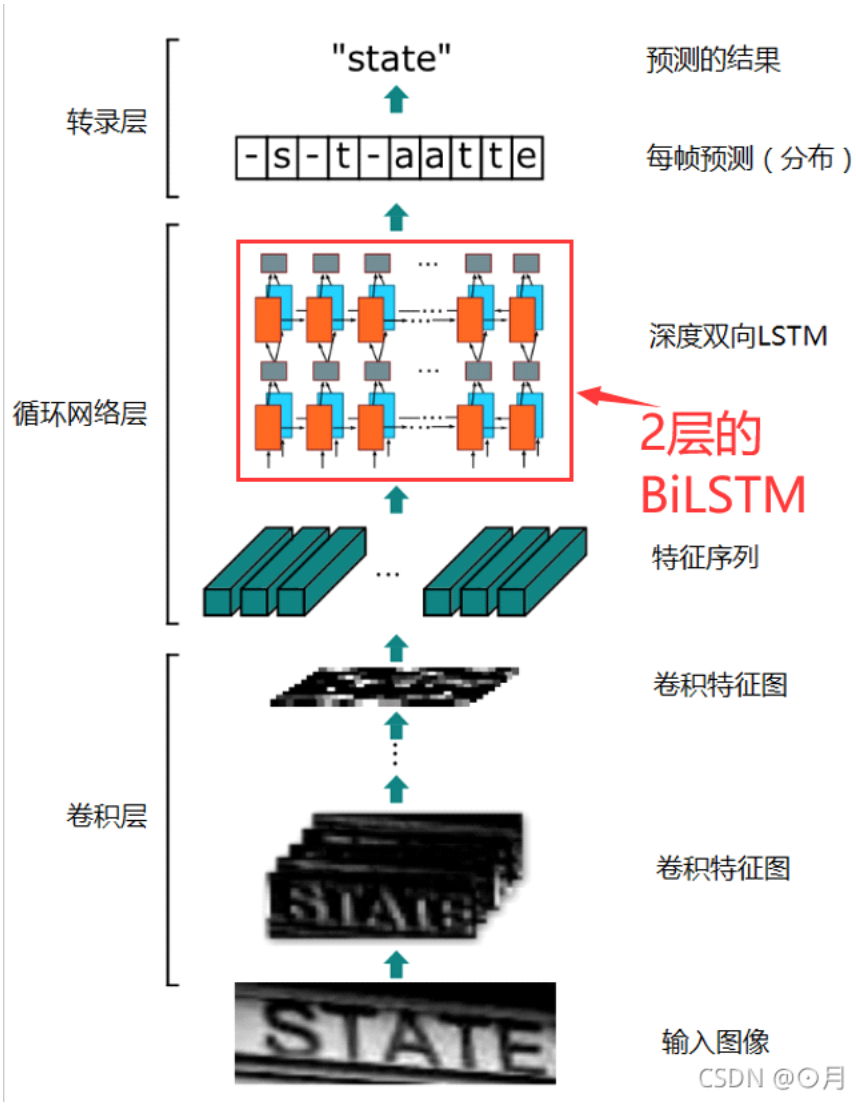
5.3 举例解释CTC的功能（CRNN算法）

以下主要参考与合并：

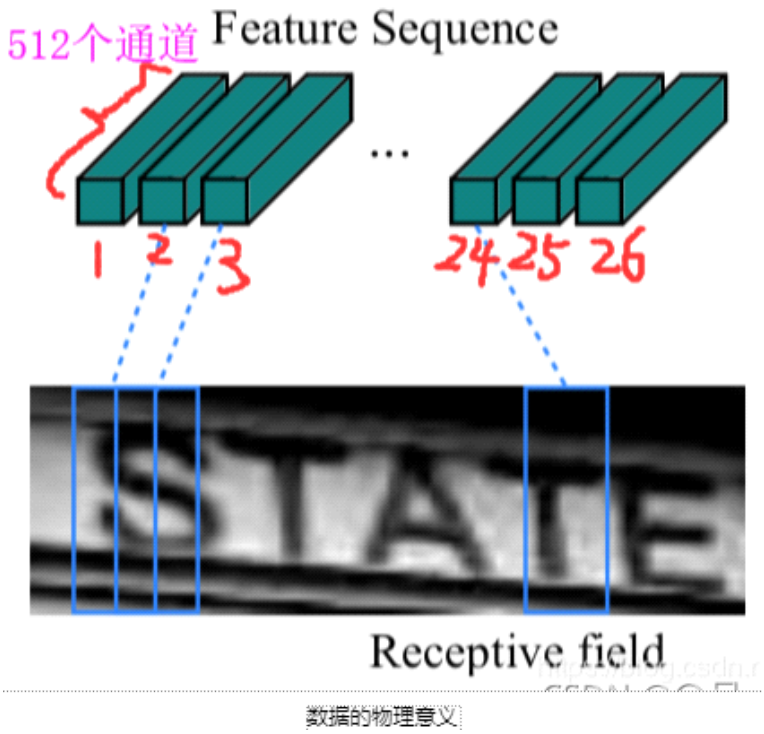
- 《CTCLoss如何使用》https://www.cnblogs.com/chenkui164/p/16359288.html
  - 有非常完备的举例，以及严谨表示。
- 《一文读懂CRNN+CTC文字识别》https://zhuanlan.zhihu.com/p/43534801
  - 有详细、严谨的表述。

基于CTC的最经典算法是CRNN（Convolutional Recurrent Neural Network）。它主要分为三个部分：图像特征提取模块CNN、图像上下文信息提取模型RNN（双向LSTM）、解码模块CTC。

架构图如下：



关于CRNN算法CTC层前面层都干了些什么，见《一幅图真正理解LSTM、BiLSTM》：https://blog.csdn.net/weixin_42118657/article/details/120022112



数据的物理意义

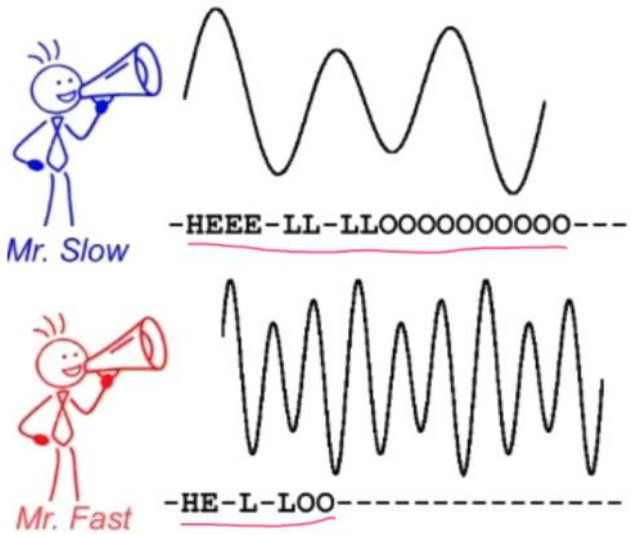
- **(26, 512)**：假设BiLSTM出来后的维度是（26, 512），26是时间步数，也即上述图片的切片数，512代表每个切片图

片中所有信息浓缩在512维向量中。

- **（26, 27）**：BiLSTM出来的结果后续会进入一个全连接层，维度由（26,512）变为（26,27）。27维度的含义是字典中字符类别个数，英文字符预测就是26个字母，另外还要加一个空白类别“-”（用来代表当前图像切片位置没有字符），总共就是27个类别（如果想让CRNN预测阿拉伯数字+英文，那标签类别个数就是26+10+1=37）。
- **（26，27）**：接着，（26,27）中第二个维度的数据会通过一个softmax，将27维向量转变成概率值，现在维度（26，27）中每个值的物理意义是，26个切片图，每个切片所属每个字母类别的概率值。

好停下来，使用传统softmax交叉熵损失，我们就能对上述（26，27）前向传播的结果进行损失计算了，只要取27维度中概率最高的那个，然后假设每个图片的标签精确到每个字母在第几个切片位置，这样预测的字母就能和标签进行损失计算。但是，标记这种对齐样本非常麻烦（有点类似目标检测标注与语义分割标注之间的差距），工作量很大。

当然这种问题同样存在于语音识别领域。例如有人说话快，有人说话慢，那么如何进行语音帧对齐，是一直以来困扰语音识别的巨大难题。



CTC的目的就是提出一种不需要对齐的Loss计算方法，用于训练网络，被广泛应用于文本行识别和语音识别中。

关于CTC loss的计算过程，公式介绍和推导就写了，具体先去看这篇极好的文章：https://www.cnblogs.com/chenkui164/p/16359288.html。里面也包含具体的例子说明，我把里面的例子说明浓缩整理一下：

回到CRNN字符识别问题，现在假设CRNN只对图片进行5等分切片，把写有“cat”的图片输入CRNN，比如：

上述这张图片经过CRNN后，最后输出维度会是（5, 27），5表示图片从左到右5等份切割，27表示26个字母+空白符号“-”。

假设这个（5, 27）维度数据如下：（刚开始训练CRNN时，每个类别预测的概率几乎都差不多）

y	t=1	t=2	t=3	t=4	t=5
k=0 (-)	0.031953	0.044296	0.038297	0.038320	0.027464
k=1 (A)	0.026221	0.030363	0.031878	0.027295	0.029824
k=2 (B)	0.040555	0.025838	0.023487	0.041529	0.028116
k=3 (C)	0.029333	0.045889	0.031872	0.023184	0.029338
k=4 (D)	0.023595	0.053792	0.022519	0.039882	0.025342

k=5 (E)	0.048014	0.028887	0.020526	0.041302	0.045833
k=6 (F)	0.028770	0.040735	0.045488	0.044244	0.032191
k=7 (G)	0.035127	0.032281	0.034032	0.051973	0.041613
k=8 (H)	0.044897	0.047910	0.049222	0.056956	0.048665
k=9 (I)	0.032323	0.044911	0.038994	0.046017	0.040002
k=10 (J)	0.047130	0.024608	0.034797	0.038146	0.041496
k=11 (K)	0.033491	0.049294	0.043909	0.053962	0.037901
k=12 (L)	0.044700	0.056019	0.046794	0.038094	0.027488
k=13 (M)	0.045632	0.034822	0.052229	0.021692	0.039653
k=14 (N)	0.035123	0.050406	0.019438	0.024067	0.056986
k=15 (O)	0.023015	0.037482	0.046163	0.050536	0.058191
k=16 (P)	0.031419	0.024302	0.035848	0.034614	0.031820
k=17 (Q)	0.034497	0.025424	0.052284	0.049642	0.029912
k=18 (R)	0.029572	0.031274	0.032931	0.026295	0.042725
k=19 (S)	0.027484	0.044015	0.031383	0.037050	0.046068
k=20 (T)	0.051330	0.047532	0.043297	0.040039	0.036849
k=21 (U)	0.034691	0.045869	0.024400	0.022020	0.029838
k=22 (V)	0.054835	0.028627	0.031971	0.039436	0.062661
k=23 (W)	0.033373	0.035513	0.047827	0.030642	0.026361
k=24 (X)	0.048700	0.022777	0.034515	0.022410	0.026991
k=25 (Y)	0.033561	0.023278	0.045237	0.034797	0.027990
k=26 (Z)	0.050657	0.023858	0.040665	0.025854	0.028682

t表示每个切片全部27个类别的概率，总共5个切片。

在5个切片的所有排列组合中，能凑出“CAT”预测结果的总共有28种组合（先不管算法如何把这28种排列算出来），如下：

	t=1	t=2	t=3	t=4	t=5
π 1	-	-	C	A	T
π 2	-	C	-	A	T
π 3	-	C	C	A	T
π 4	-	C	A	-	T
π 5	-	C	A	A	T
π 6	-	C	A	T	-
π 7	-	C	A	T	T
π 8	C	-	-	A	T
π 9	C	-	A	-	T
π 10	C	-	A	A	T
π 11	C	-	A	T	-
π 12	C	-	A	T	T
π 13	C	C	-	A	T
π 14	C	C	C	A	T
π 15	C	C	A	-	T

π 16	C	C	A	A	T
π 17	C	C	A	T	–
π 18	C	C	A	T	T
π 19	C	A	–	–	T
π 20	C	A	–	T	–
π 21	C	A	–	T	T
π 22	C	A	A	–	T
π 23	C	A	A	A	T
π 24	C	A	A	T	–
π 25	C	A	A	T	T
π 26	C	A	T	–	–
π 27	C	A	T	T	–
π 28	C	A	T	T	T

上述注意：对于相邻切片预测为同一个字符的，需要合并为1个字符，而如果两个相同字符中间有预测为“-”的空白符，那这2个相同字符就不合并，如下：

举例说明，当 $T = 12$ 时：

$$B(\pi_1) = B(--stta-t--e) = state$$

$$B(\pi_2) = B(sst-aaa-tee-) = state$$

$$B(\pi_3) = B(--sttaa-tee-) = state$$

$$B(\pi_4) = B(sst-aa-t---e) = state$$

对于字符间有blank符号的则不合并：

$$B(\pi_5) = B(-\ s\ t\ a-a\ t\ t\ e-\ \ e-) = staatee$$

上述预测为“CAT”的排列总共有28种组合，我们逐一查询并计算出每种组合情况下的概率值，然后累加这28种组合的总概率值。比如，查询"- - C A T"这种情况的概率值如下（上述概率表标红的）：

$$y^1_{k=\pi^1_1} = y^1_0 = 0.031953$$

$$y^1_{k=\pi^2_1} = y^2_0 = 0.044296$$

$$y^1_{k=\pi^3_1} = y^3_3 = 0.031872$$

$$y^1_{k=\pi^4_1} = y^4_1 = 0.027295$$

$$y^1_{k=\pi^5_1} = y^5_{20} = 0.036849$$

因此路径π_1的概率$p(\pi_1|x)$的计算如下

$$\begin{aligned} p(\pi_1|x) &= \prod_{t=1}^{T} y^t_{k=\pi^t_1} \\ &= y^1_{k=\pi^1_1} \times y^2_{k=\pi^2_1} \times y^3_{k=\pi^3_1} \times \ldots \times y^T_{k=\pi^T_1} \\ &= y^1_0 \times y^2_0 \times y^3_3 \times y^4_1 \times y^5_{20} \\ &= 0.031953 \times 0.044296 \times 0.031872 \times 0.027295 \times 0.036849 \\ &= 4.5373e^{-8} \end{aligned}$$

同理可计算

$$p(\pi_1|x) = 4.5374e^{-8}, p(\pi_2|x) = 5.6482e^{-8}, p(\pi_3|x) = 4.7006e^{-8}, p(\pi_4|x) = 6.6003e^{-8}$$

$$p(\pi_5|x) = 4.7014e^{-8}, p(\pi_6|x) = 5.1401e^{-8}, p(\pi_7|x) = 6.8965e^{-8}, p(\pi_8|x) = 5.0050e^{-8}$$

$$p(\pi_9|x) = 5.8487e^{-8}, p(\pi_{10}|x) = 4.1660e^{-8}, p(\pi_{11}|x) = 4.5547e^{-8}, p(\pi_{12}|x) = 6.1111e^{-8}$$

$$p(\pi_{13}|x) = 5.1850e^{-8}, p(\pi_{14}|x) = 4.3151e^{-8}, p(\pi_{15}|x) = 6.0590e^{-8}, p(\pi_{16}|x) = 4.3158e^{-8}$$

$$p(\pi_{17}|x) = 4.7185e^{-8}, p(\pi_{18}|x) = 6.3309e^{-8}, p(\pi_{19}|x) = 4.8163e^{-8}, p(\pi_{20}|x) = 3.7508e^{-8}$$

$$p(\pi_{21}|x) = 5.0324e^{-8}, p(\pi_{22}|x) = 4.0090e^{-8}, p(\pi_{23}|x) = 2.8556e^{-8}, p(\pi_{24}|x) = 3.1220e^{-8}$$

$$p(\pi_{25}|x) = 4.1889e^{-8}, p(\pi_{26}|x) = 4.0583e^{-8}, p(\pi_{27}|x) = 4.2404e^{-8}, p(\pi_{28}|x) = 5.6894e^{-8}$$

计算总概率$p(l|x)$

$p(l|x)$是所有满足$\mathcal{B}(\pi) = l$的路径概率之和。

$$\begin{aligned} p(l|x) &= \sum_{\pi \in \mathcal{B}^{-1}(l)} p(\pi|x) \\ &= p(\pi_1|x) + p(\pi_2|x) + p(\pi_1|x) + \ldots + p(\pi_{28}|x) \\ &= 4.5374e^{-8} + 5.6482e^{-8} + 4.7006e^{-8} + \ldots + 5.6894e^{-8} \\ &= 1.366e^{-6} \end{aligned}$$

计算损失函数CTCLoss

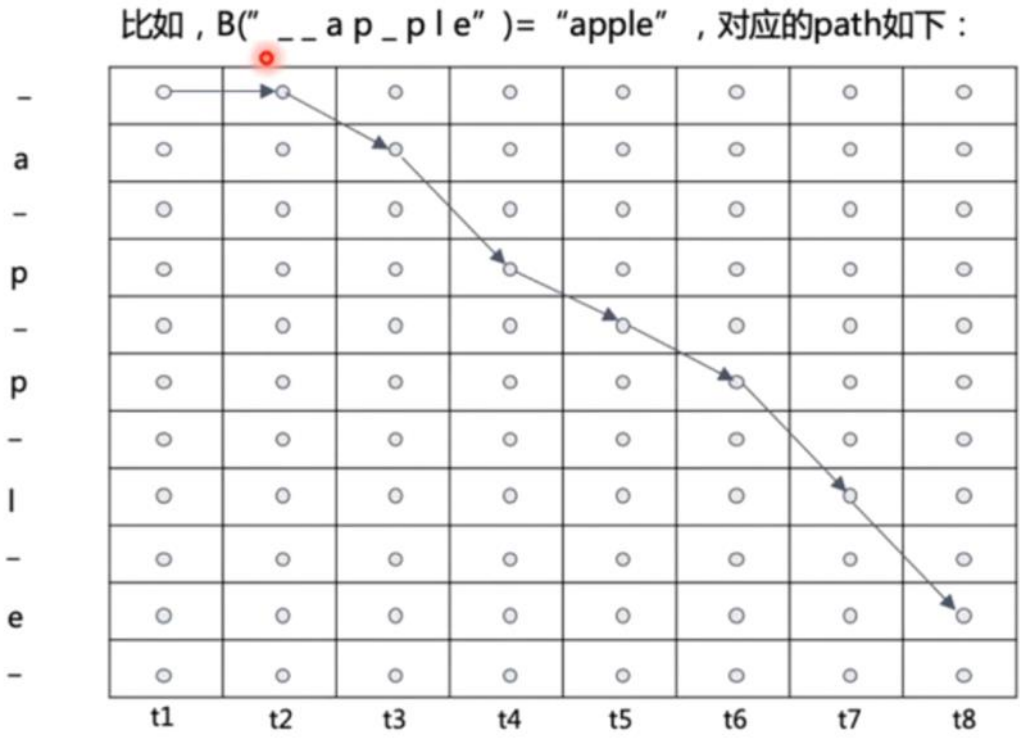
由于例子中只给了1样本，所以下面的损失函数CTCLoss也就只有这一个样本的损失。

$$\begin{aligned} O^{ML}(S, \mathcal{N}_w) &= -ln(S, \mathcal{N}_w) \\ &= -\sum_{x,z \in S} ln(p(z|x)) \\ &= -ln(p(z|x)) \\ &= -ln(1.366e^{-6}) \\ &= 13.5036 \end{aligned}$$

如此，就算出了本张图片经过CRNN后的损失值，后续就是反向传播更新所有参数。

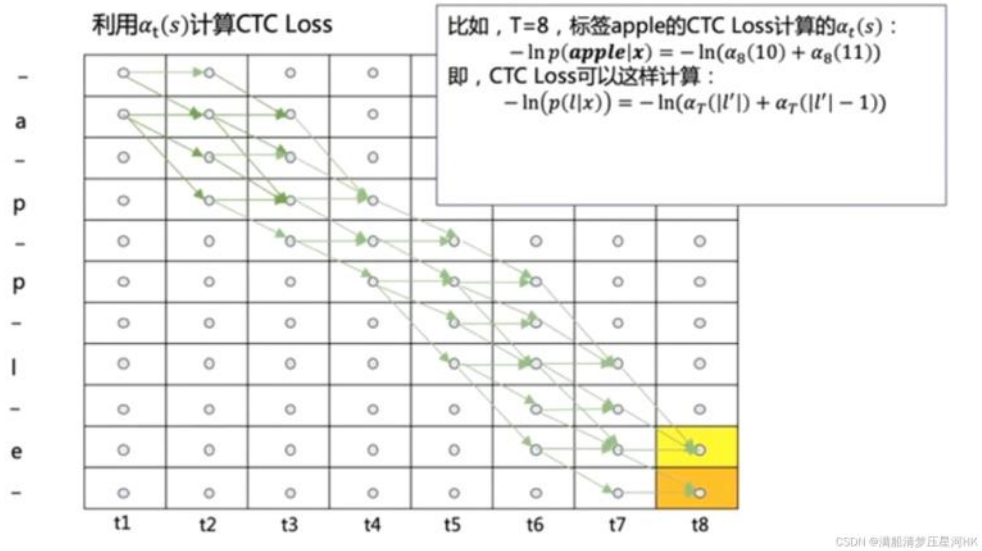
在上述方案中，面对具体问题，使用代码实现时，有个难点需要解决，就是如何找到所有可能的可以得到正确预测答案的组合。比如在CRNN实际的算法中，对图像会有26个切片，如果暴力穷举，就完全无法实用。这个问题怎么解决？——用动态规划方法。

如果刷题刷的比较多的朋友，肯定就一眼看出来了，这就是一个动态规划的网格类问题。具体做法：将gt每个字符之间加一个空白符，作为dp的行，dp的列为各个序列，如下图：



$$P(\text{" \_ \_ a p \_ p l e"}) = y_-^1 y_-^2 y_a^3 y_p^4 y_-^5 y_p^6 y_l^7 y_e^8$$

$$p(\mathbf{z}|\mathbf{x}) = \alpha_T(|\mathbf{z}'|) + \alpha_T(|\mathbf{z}'| - 1)$$



我们只需要定义规则和状态转移方程，每步相乘，即可求出每个可以得到apple的字符概率，再相加求到总概率。（上述参考：https://blog.csdn.net/qq_38253797/article/details/124894754）

更严谨而言，CTC中用的“前向后向算法”（Forward-Backward Algorithm），本质也是一种动态规划，因为它通过递归关系存储和复用中间计算结果，以避免重复计算。这一过程类似于隐马尔可夫模型（HMM）中的前向后向算法。具体来说，前向后向算法分为两部分：

- 前向概率（Forward Probability）：计算从起始点到某个时间步的所有可能路径的累积概率。
- 后向概率（Backward Probability）：计算从某个时间步到终点的所有可能路径的累积概率。

通过前向概率和后向概率的组合，CTC可以高效地计算出给定目标序列的所有可能路径的总概率。具体详解见：https://zhuanlan.zhihu.com/p/43534801。

5.4 CTC实现代码

5.4.1 类 `CTC` 的定义

- `__init__` 方法：初始化 CTC 类，设置空白标签的索引，默认为 0。
- `extend_target_with_blank` 方法：这个方法用于将目标序列扩展，插入空白标签。它还创建了一个跳跃连接数组，用来在计算向前和向后概率时优化计算路径。
- `get_forward_probs` 方法：计算向前概率。这个方法逐时间步更新并累积到达每个扩展符号位置的概率。
- `get_backward_probs` 方法：计算向后概率。这个方法从序列的末尾开始反向逐步计算到达每个扩展符号位置的概率。
- `get_posterior_probs` 方法：利用计算出的向前和向后概率来计算后验概率，这是每个符号位置上观测到的数据与模型预测的一致程度的衡量。

5.4.2 类 `CTCLoss` 的定义

- `__init__` 方法：初始化 CTC 损失计算类，同样设置空白标签索引。
- `forward` 方法：计算给定批次数据的 CTC 损失。该方法对于每个样本：
    1. 截断目标序列和对数概率序列至实际长度。
    2. 使用 `extend_target_with_blank` 方法扩展目标序列。
    3. 计算向前和向后概率。
    4. 计算后验概率。
    5. 根据后验概率计算期望的发散度，并更新总损失。
1. `backward` 方法：计算损失函数相对于输入概率的梯度。这是通过对每个时间步的每个符号位置使用后验概率和原始概率的比值来进行的。

```
import numpy as np

class CTC(object):
```

```
    def __init__(self, BLANK=0):
        """
        初始化实例变量

        参数:
        ------
        BLANK (int, 可选): 空白标签的索引。默认值为0。
        """
        self.BLANK = BLANK

    def extend_target_with_blank(self, target):
        """用空白符扩展目标序列。

        输入:
        -----
        target: (np.array, 维度 = (target_len,))
                目标输出，包含目标音素的索引
        示例: [1,4,4,7]

        返回:
        ------
        extSymbols: (np.array, 维度 = (2 * target_len + 1,))
                    扩展了空白符的目标序列
        示例: [0,1,0,4,0,4,0,7,0]

        skipConnect: (np.array, 维度 = (2 * target_len + 1,))
                    跳跃连接
        示例: [0,0,0,1,0,0,0,1,0]
        """

        # 初始化扩展符号序列，首先添加一个空白符
        extended_symbols = [self.BLANK]
        for symbol in target:
            extended_symbols.append(symbol)  # 添加目标符号
            extended_symbols.append(self.BLANK)  # 添加空白符

        N = len(extended_symbols)  # 扩展后的序列长度
        skip_connect = np.zeros(N)  # 初始化跳跃连接数组，默认值为0

        for i, sy in enumerate(target):
            if i != 0:
                # 如果当前符号与前一个符号不同，设置跳跃连接为1
                if target[i] != target[i-1]:
                    skip_connect[2*i+1] = 1

        extended_symbols = np.array(extended_symbols).reshape((N,))  # 转换为numpy数组
        skip_connect = np.array(skip_connect).reshape((N,))  # 转换为numpy数组

        return extended_symbols, skip_connect
```

```
def get_forward_probs(self, logits, extended_symbols, skip_connect):
    """计算前向概率。

    输入:
    -----
    logits: (np.array, 维度 = (input_len, len(Symbols)))
            预测的（对数）概率

            在时间步t获取符号i的对数概率:
            p(t,s(i)) = logits[t, extended_symbols[i]]

    extSymbols: (np.array, 维度 = (2 * target_len + 1,))
                扩展了空白符的标签序列

    skipConnect: (np.array, 维度 = (2 * target_len + 1,))
                跳跃连接

    返回:
    ------
    alpha: (np.array, 维度 = (input_len, 2 * target_len + 1))
            前向概率
    """

    S, T = len(extended_symbols), len(logits)  # S为扩展后的序列长度，T为输入序列长度
    alpha = np.zeros(shape=(T, S))  # 初始化前向概率矩阵

    # 初始时间步的前向概率
    alpha[0, 0] = logits[0, extended_symbols[0]]  # 第一个符号的前向概率
    alpha[0, 1] = logits[0, extended_symbols[1]]  # 第二个符号的前向概率

    for t in range(1, T):
        # 计算时间步t的第一个符号的前向概率
        alpha[t, 0] = alpha[t-1, 0] * logits[t, extended_symbols[0]]
        for r in range(1, S):
            # 当前符号的前向概率等于前一个符号和当前符号前缀的概率之和
            alpha[t, r] = alpha[t-1, r] + alpha[t-1, r-1]
            # 如果可以跳跃连接，增加跳跃连接的前向概率
            if (r > 1 and skip_connect[r]):
                alpha[t, r] += alpha[t-1, r-2]
            # 乘以当前时间步的对数概率
            alpha[t, r] *= logits[t, extended_symbols[r]]

    return alpha

def get_backward_probs(self, logits, extended_symbols, skip_connect):
    """计算后向概率。

    输入:
```

```
        -----
        logits: (np.array, 维度 = (input_len, len(symbols)))
                预测的（对数）概率

                在时间步t获取符号i的对数概率:
                p(t,s(i)) = logits[t,extended_symbols[i]]

        extSymbols: (np.array, 维度 = (2 * target_len + 1,))
                    扩展了空白符的标签序列

        skipConnect: (np.array, 维度 = (2 * target_len + 1,))
                    跳跃连接

        返回:
        ------
        beta: (np.array, 维度 = (input_len, 2 * target_len + 1))
                后向概率

        """

        S, T = len(extended_symbols), len(logits)  # S为扩展后的序列长度，T为输入序列长度
        betahat = np.zeros(shape=(T, S))  # 初始化后向概率的中间值矩阵
        beta = np.zeros(shape=(T, S))  # 初始化后向概率矩阵

        # 最后一个时间步的初始后向概率
        betahat[T-1, S-1] = logits[T-1, extended_symbols[S-1]]
        betahat[T-1, S-2] = logits[T-1, extended_symbols[S-2]]

        for t in range(T-2, -1, -1):
            # 计算时间步t的最后一个符号的后向概率
            betahat[t, S-1] = betahat[t+1, S-1] * logits[t, extended_symbols[S-1]]
            for r in range(S-2, -1, -1):
                # 当前符号的后向概率等于后一个符号和当前符号后缀的概率之和
                betahat[t, r] = betahat[t+1, r] + betahat[t+1, r+1]
                # 如果可以跳跃连接，增加跳跃连接的后向概率
                if (r <= S-3 and skip_connect[r+2]):
                    betahat[t, r] += betahat[t+1, r+2]
                # 乘以当前时间步的对数概率
                betahat[t, r] *= logits[t, extended_symbols[r]]

        # 归一化后向概率
        for t in range(T-1, -1, -1):
            for r in range(S-1, -1, -1):
                beta[t, r] = betahat[t, r] / logits[t, extended_symbols[r]]

        return beta

    def get_posterior_probs(self, alpha, beta):
        """计算后验概率。
```

```
        输入:
        -----
        alpha: (np.array, 维度 = (input_len, 2 * target_len + 1))
                前向概率

        beta: (np.array, 维度 = (input_len, 2 * target_len + 1))
                后向概率

        返回:
        ------
        gamma: (np.array, 维度 = (input_len, 2 * target_len + 1))
                后验概率
        """

        [T, S] = alpha.shape  # 获取时间步长度和扩展序列长度
        gamma = np.zeros(shape=(T, S))  # 初始化后验概率矩阵
        sumgamma = np.zeros((T,))  # 初始化后验概率的归一化因子

        for t in range(T):
            # 计算时间步t的后验概率
            for r in range(S):
                gamma[t, r] = alpha[t, r] * beta[t, r]
                sumgamma[t] += gamma[t, r]

            # 对时间步t的后验概率进行归一化
            for r in range(S):
                gamma[t, r] = gamma[t, r] / sumgamma[t]

        return gamma


class CTCLoss(object):

    def __init__(self, BLANK=0):
        """
        初始化实例变量

        参数:
        -----------
        BLANK (int, 可选): 空白标签的索引。默认值为0。

        """
        super(CTCLoss, self).__init__()  # 调用父类的初始化函数

        self.BLANK = BLANK
        self.gammas = []  # 存储后验概率
        self.ctc = CTC()  # 初始化CTC对象

    def __call__(self, logits, target, input_lengths, target_lengths):
        # 调用forward函数计算CTC损失
```

```
        return self.forward(logits, target, input_lengths, target_lengths)

    def forward(self, logits, target, input_lengths, target_lengths):
        """CTC 损失前向计算

        计算CTC损失，通过计算前向、后向和后验概率，然后计算目标和预测对数概率之间的平均损失。

        输入:
        -----
        logits [np.array, 维度=(seq_length, batch_size, len(symbols))]:
            RNN/GRU输出的对数概率（输出序列）

        target [np.array, 维度=(batch_size, padded_target_len)]:
            目标序列

        input_lengths [np.array, 维度=(batch_size,)]:
            输入序列的长度

        target_lengths [np.array, 维度=(batch_size,)]:
            目标序列的长度

        返回:
        -------
        loss [float]:
            后验概率和目标之间的平均散度
        """

        self.logits = logits
        self.target = target
        self.input_lengths = input_lengths
        self.target_lengths = target_lengths

        B, _ = target.shape  # 获取批次大小
        total_loss = np.zeros(B)  # 初始化总损失
        self.extended_symbols = []  # 初始化扩展符号列表

        for batch_itr in range(B):
            # -------------------------------------------->
            # 计算单个批次的CTC损失
            # 过程:
            #     将目标序列截断为目标长度
            #     将logits截断为输入长度
            #     用空白符扩展目标序列
            #     计算前向概率
            #     计算后向概率
            #     使用总概率函数计算后验概率
            #     计算每个批次的期望散度并存储在total_loss中
            #     对所有批次取平均并返回最终结果
            # <--------------------------------------------
```

```
            ctc = CTC()
            target_trunc = target[batch_itr, :target_lengths[batch_itr]]  # 截断目标序列到目标长
度
            logits_trunc = logits[:input_lengths[batch_itr], batch_itr, :]  # 截断logits到输入长
度
            extended_symbols, skip_connect = ctc.extend_target_with_blank(target_trunc)  # 扩展
目标序列
            alpha = ctc.get_forward_probs(logits_trunc, extended_symbols, skip_connect)  # 计算
前向概率
            beta = ctc.get_backward_probs(logits_trunc, extended_symbols, skip_connect)  # 计算
后向概率
            gamma = ctc.get_posterior_probs(alpha, beta)  # 计算后验概率

            div = 0
            S, T = len(extended_symbols), len(logits_trunc)
            for t in range(T):
                for r in range(S):
                    # 计算后验概率和对数概率之间的散度
                    div -= gamma[t, r] * np.log(logits_trunc[t, extended_symbols[r]])

            total_loss += div  # 累加散度

        total_loss /= B  # 对所有批次取平均

        return total_loss

    def backward(self):
        """
        CTC损失反向计算

        计算相对于参数的梯度，并返回相对于输入（xt和ht）的导数。

        输入:
        -----
        logits [np.array, 维度=(seqlength, batch_size, len(Symbols))]:
            RNN/GRU输出的对数概率（输出序列）

        target [np.array, 维度=(batch_size, padded_target_len)]:
            目标序列

        input_lengths [np.array, 维度=(batch_size,)]:
            输入序列的长度

        target_lengths [np.array, 维度=(batch_size,)]:
            目标序列的长度

        返回:
        -------
```

```
        dY [np.array, 维度=(seq_length, batch_size, len(extended_symbols))]:
            散度相对于输入符号在每个时间步的导数
        """

        T, B, C = self.logits.shape  # 获取时间步、批次大小和符号数
        dY = np.full_like(self.logits, 0)  # 初始化导数矩阵

        for batch_itr in range(B):
            # ------------------------------------------>
            # 计算单个批次的CTC导数
            # 过程:
            #     将目标序列截断为目标长度
            #     将logits截断为输入长度
            #     用空白符扩展目标序列
            #     计算散度的导数并存储在dY中
            # <------------------------------------------

            ctc = CTC()
            target_trunc = self.target[batch_itr, :self.target_lengths[batch_itr]]  # 截断目标
序列到目标长度
            logits_trunc = self.logits[:self.input_lengths[batch_itr], batch_itr, :]  # 截断
logits到输入长度
            extended_symbols, skip_connect = ctc.extend_target_with_blank(target_trunc)  # 扩展
目标序列
            alpha = ctc.get_forward_probs(logits_trunc, extended_symbols, skip_connect)  # 计算
前向概率
            beta = ctc.get_backward_probs(logits_trunc, extended_symbols, skip_connect)  # 计算
后向概率
            gamma = ctc.get_posterior_probs(alpha, beta)  # 计算后验概率

            S, T = len(extended_symbols), len(logits_trunc)
            for t in range(T):
                for r in range(S):
                    # 计算散度相对于输入符号的导数
                    dY[t, batch_itr, extended_symbols[r]] -= gamma[t, r] / logits_trunc[t,
extended_symbols[r]]

        return dY
```

6 CTC解码：贪婪搜索和束搜索

还有最后一个问题：当模型训练好后，如何取预测值?

6.1 贪婪搜索

在CRNN中，通常使用贪心搜索方法直接选取每个时间步上模型预测概率最高的字符作为输出，然后通过CTC解码中的规则去除重复字符和空白符号，最终得到预测结果。

在讲义中的示意图和代码如下：

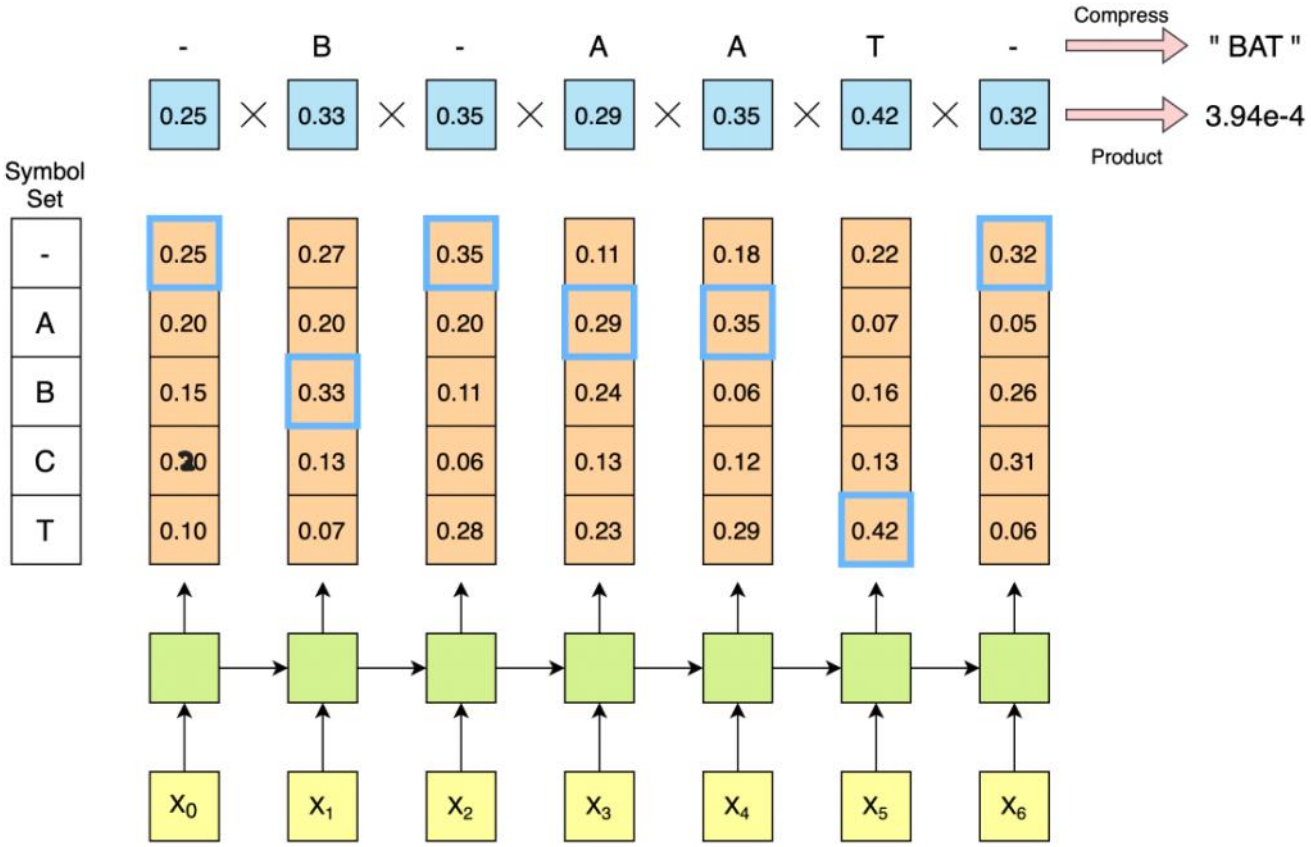


Figure 18: Greedy Search

贪婪搜索代码

```
class GreedySearchDecoder(object):

    def __init__(self, symbol_set):
        """
        初始化实例变量

        参数:
        -----------

        symbol_set [list[str]]:
            所有的符号（不包括空白符号的词汇表）

        """

        self.symbol_set = symbol_set
```

```
    def decode(self, y_probs):
        """
        执行贪婪搜索解码

        输入:
        -----

        y_probs [np.array, 维度=(len(symbols) + 1, seq_length, batch_size)]
            符号的概率分布，注意对于第一个部分的批次大小始终为1，
            但如果你计划将实现用于第二部分，则需要考虑批次大小

        返回:
        -------

        decoded_path [str]:
            压缩后的符号序列，即去除空白符号或重复符号后的序列

        path_prob [float]:
            贪婪路径的前向概率

        """

        decoded_path = []  # 用于存储解码后的路径
        blank = 0  # 空白符号的索引
        path_prob = 1  # 初始化路径概率为1

        # 1. 遍历序列长度 - len(y_probs[0])
        # 2. 遍历符号概率
        # 3. 通过与当前最大概率相乘来更新路径概率
        # 4. 选择最可能的符号并附加到解码路径
        # 5. 压缩序列（在循环内或循环外完成）

        symbols_len, seq_len, batch_size = y_probs.shape  # 获取符号长度、序列长度和批次大小
        self.symbol_set = ["-"] + self.symbol_set  # 将空白符号添加到符号集合的开头
        for batch_itr in range(batch_size):

            path = " "  # 初始化路径为空格
            path_prob = 1  # 初始化路径概率为1
            for i in range(seq_len):
                max_idx = np.argmax(y_probs[:, i, batch_itr])  # 找到当前时间步概率最大的符号的索
引

                path_prob *= y_probs[max_idx, i, batch_itr]  # 更新路径概率
                if path[-1] != self.symbol_set[max_idx]:  # 如果当前符号与前一个符号不同
                    path += self.symbol_set[max_idx]  # 将符号添加到路径中

            path = path.replace('-', '')  # 移除路径中的空白符号
            decoded_path.append(path[1:])  # 将路径添加到解码后的路径列表中
```

```
    return path[1:], path_prob  # 返回解码后的路径和路径概率
```

6.2 束搜索

但是，在生成式任务和语音识别任务中，当前时间步的预测结果通常需要作为下一时间步的输入或影响下一时间步的预测。在这些情况下，如果仅仅依赖贪心搜索，即每次只选择概率最高的结果，可能会忽略上下文信息，从而导致整体结果不理想。

如果算力无限，最好办法是把每种结果都试一次，看最终整个的预测结果哪个最好，但是这种指数级计算难以实现，折中的方法是“束搜索”（Beam Search），即每个时间步的预测，取概率最高的k种组合，然后这K个结果独立的去用于下一帧的预测，然后再从衍生的多种组合中，**依旧只取概率最高的K种组合**，依次类推，到最后一个时间步，取概率最高的组合即可。

在讲义中描述了一个用于理解光束搜索（Beam Search）的示例。在这里，我们在一个仅由{-，A，B}组成的词汇表上执行光束搜索，其中“-”是空白符号。光束宽度（beam width）为3，并且我们执行了三个解码步骤。表格展示了每个时间步的各个类别的输出概率。图展示了束搜索的流程。

6.2.1 Example

Fig. 19 depicts a toy problem for understanding Beam Search. Here, we are performing Beam Search over a vocabulary of {-, A, B}, where " - " is the BLANK character. The **beam width is 3** and we perform three decoding steps. Table 8 shows the output probabilities for each token at each decoding step.

Vocabulary	P(symbol) @ T=1	P(symbol) @ T=2	P(symbol) @ T=3
-	0.49	0.38	0.02
A	0.03	0.44	0.40
B	0.47	0.18	0.58

Table 8: Probabilities of each symbol in the vocabulary over three consecutive decoding steps

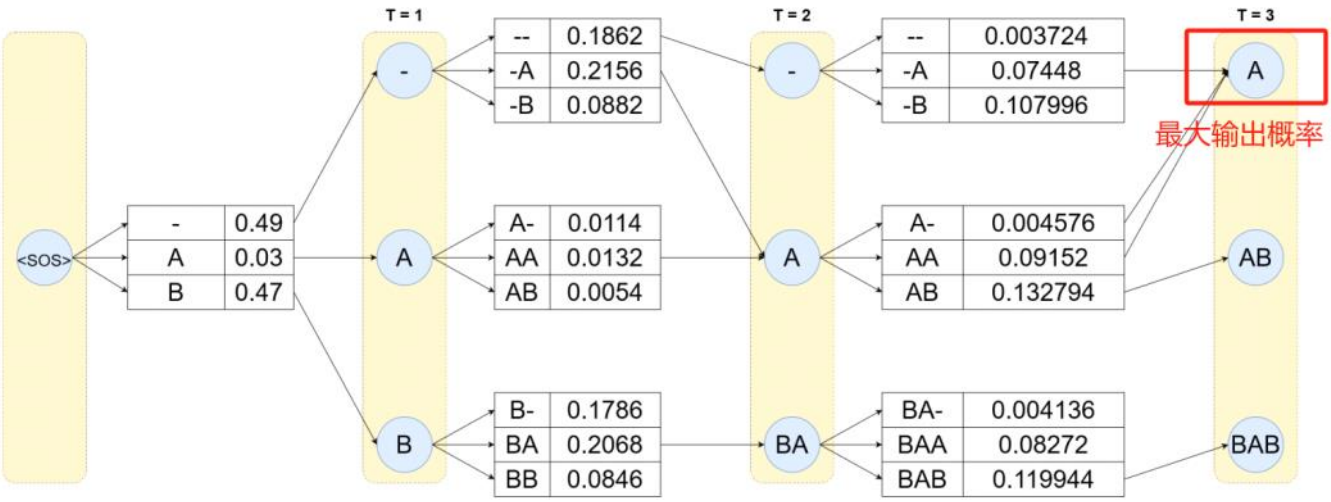


Figure 19: Beam Search over a vocabulary / symbols set = {-, A, B} with beam width (k) = 3. (" - " == BLANK). The blue shaded nodes indicate the compressed decoded sequence at given time step, and the expansion tables show the probability (right column) and symbols (left column) at each time step pre-pended with the current decoded sequence

上面注意几点：

- 每通过一个时间步，就有新的3*3种排列，必须先去重、合并概率，然后从去重合并后的所有结果中，取概率最高的3种作为下一个时间步。
- 去重合并的规则：比如上图T=2中的A，就是AA和-A合并来的。注意，A-不能合并为A，因为如果A-下一个时间步如果再接个A，预测结果就是A-A即AA，而如果A-合并为A，A下一个时间步后面再接个A，结果就是A。
- 最终遍历完所有时间步后，取概率最高的情况，即上图的T=3时刻的A。

束搜索（Beam Search）算法的伪代码和设计思路

1. **函数定义**：
   - 你需要实现一个名为 `BeamSearch` 的函数，它接收三个参数：
     - `SymbolSets`：这是符号集，包含所有可能生成的符号（类似于字母表）。
     - `y_probs`：这是一个张量，包含每个时间步上每个符号的概率。
     - `BeamWidth`：这个参数限制了在每个时间步上保留的部分序列（或路径）的数量。
- **初始化**：
  - 在光束搜索的初始化阶段，使用一个包含空白符号的路径开始搜索。在每个时间步，迭代当前的最佳路径，并通过每个可能的新符号扩展路径，同时计算新路径的分数。
- **路径扩展**：
  - 在扩展路径时，你将遇到三种场景：
    1. 新符号与路径上的最后一个符号相同。
    2. 路径的最后一个符号是空白符号。
    3. 路径的最后一个符号与新符号不同且不是空白符号。
  1. 扩展路径后，根据 `BeamWidth` 的设定，保留得分最高的路径。
2. **后处理**：
   - 在所有时间步结束后，如果路径的末尾有空白符号，将其移除。然后将数字符号序列转换为字符串序列，汇总路径的分数，最终选择分数最高的路径作为最佳路径。
- **返回值**：
  - 该函数最终应返回最高得分的路径以及所有最终路径的得分。

额外说明：

- **批量处理**：注意，提供的 `y_probs` 是为批量大小为1的情况设计的，但你需要考虑如何将该函数适应更大的批量大小。
- **扩展和优化**：你可以在这个类中实现其他方法，只要你从解码方法中返回期望的变量即可。此外，你可以尝试更高效的实现方法，其中之一已经通过伪代码给出。

输入：`SymbolSets`，`y_probs`，`BeamWidth`

输出：`BestPath`，`MergedPathScores`

```
初始化：
BestPaths ← 一个字典，将空白符路径映射到分数1.0
TempBestPaths ← 一个空字典
对 y_probs 中的每个时间步执行
    提取当前符号的概率；
    对 BestPaths 中的路径和分数进行迭代，限制为 BeamWidth；
        对当前符号概率中的每个新符号进行迭代；
            根据路径的最后一个符号，确定新路径；
            在 TempBestPaths 中更新新路径的分数；
        结束
    结束
    用 TempBestPaths 更新 BestPaths；
    清空 TempBestPaths；
结束
// 在所有时间步结束时，如果路径末尾是空白符，则移除它
初始化 MergedPathScores 为一个空字典；
BestPath ← 空白符路径
BestScore ← 0
对 BestPaths 中的每个路径和分数进行迭代
    移除路径末尾的空白符；
    在 MergedPathScores 中更新翻译后的路径分数；
    如果 score > BestScore，则
        更新 BestPath 和 BestScore；
    结束
结束
返回 BestPath 和 BestScore；
```

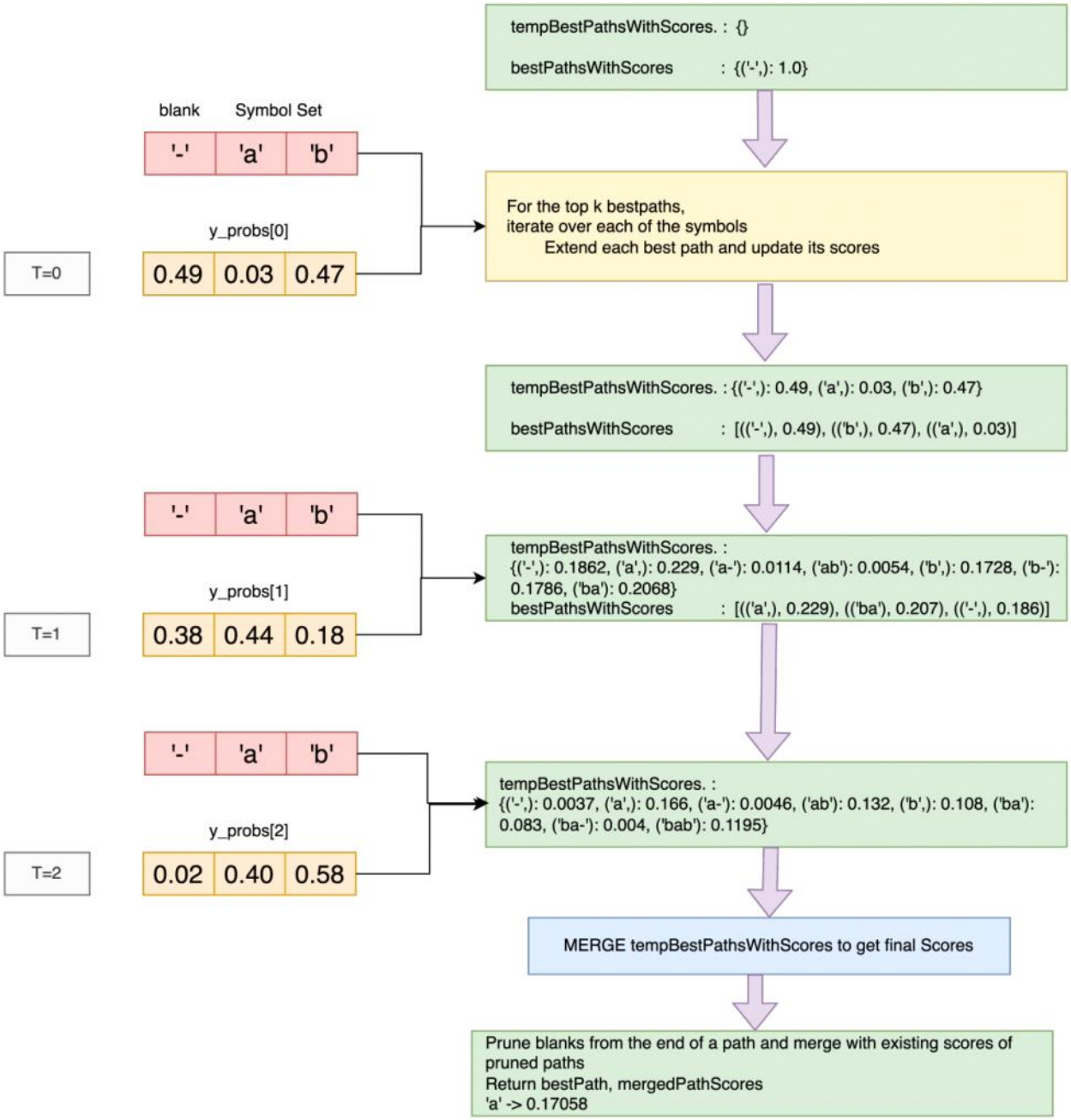


Figure 20: Efficient Beam Search procedure

束搜索代码

```
class BeamSearchDecoder(object):

    def __init__(self, symbol_set, beam_width):
        """
        初始化实例变量

        参数:
        ----------

        symbol_set [list[str]]:
            所有的符号（不包括空白符号的词汇表）
```

```
        beam_width [int]:
            光束宽度，用于选择扩展的前k个假设

        """

        self.symbol_set = symbol_set
        self.beam_width = beam_width

    def decode(self, y_probs):
        """
        执行光束搜索解码

        输入:
        -----

        y_probs [np.array, 维度=(len(symbols) + 1, seq_length, batch_size)]
            符号的概率分布，注意对于第一个部分的批次大小始终为1，
            但如果你计划将实现用于第二部分，则需要考虑批次大小

        返回:
        -------

        forward_path [str]:
            拥有最佳路径分数（前向概率）的符号序列

        merged_path_scores [dict]:
            所有最终合并路径及其分数

        """
        self.symbol_set = ['-'] + self.symbol_set  # 将空白符号添加到符号集合的开头
        symbols_len, seq_len, batch_size = y_probs.shape  # 获取符号长度、序列长度和批次大小
        bestPaths = dict()  # 存储当前最佳路径
        tempBestPaths = dict()  # 存储临时的最佳路径
        bestPaths['-'] = 1  # 初始化最佳路径为空白符号，分数为1

        # 遍历序列长度
        for t in range(seq_len):
            sym_probs = y_probs[:, t]  # 获取当前时间步的符号概率分布
            tempBestPaths = dict()  # 重置临时路径

            # 遍历当前的最佳路径
            for path, score in bestPaths.items():

                # 遍历所有符号
                for r, prob in enumerate(sym_probs):
                    new_path = path  # 初始化新路径为当前路径

                    # 更新新路径
```

```
                    if path[-1] == '-':  # 如果当前路径的最后一个符号是空白符号
                        new_path = new_path[:-1] + self.symbol_set[r]  # 替换最后一个符号为当前符号
                    elif (path[-1] != self.symbol_set[r]) and not (t == seq_len-1 and self.symbol_set[r] == '-'):
                        new_path += self.symbol_set[r]  # 如果当前符号与最后一个符号不同，且不是空白符号，则附加当前符号

                    # 在临时路径中更新概率
                    if new_path in tempBestPaths:
                        tempBestPaths[new_path] += prob * score  # 累加路径概率
                    else:
                        tempBestPaths[new_path] = prob * score  # 初始化路径概率


            # 获取前k个最佳路径并重置最佳路径
            if len(tempBestPaths) >= self.beam_width:
                bestPaths = dict(sorted(tempBestPaths.items(), key=lambda x: x[1], reverse=True)[:self.beam_width])

        # 获取得分最高的路径
        bestPath = max(bestPaths, key=bestPaths.get)
        finalPaths = dict()
        for path, score in tempBestPaths.items():
            if path[-1] == '-':
                finalPaths[path[:-1]] = score  # 移除路径末尾的空白符号
            else:
                finalPaths[path] = score  # 保持路径原样
        return bestPath, finalPaths  # 返回最佳路径和所有最终合并的路径及其分数
```