RNN

> 2013551 雷贺奥

实验要求

- 掌握RNN原理
- 学会使用PyTorch搭建循环神经网络来训练名字识别
- 学会使用PyTorch搭建LSTM网络来训练名字识别

作者复现了原始版本的RNN，实现了调库的Lstm，最后也写出了自己的Lstm,效果甚至要优于调库实现的Lstm。

原始版本RNN

网络结构

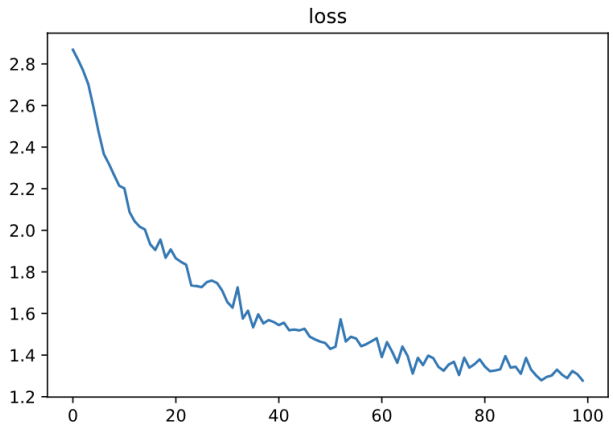
```
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()
        self.hidden_size = hidden_size
        self.i2h = nn.Linear(input_size + hidden_size, hidden_size)
        self.i2o = nn.Linear(input_size + hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        combined = torch.cat((input, hidden), 1)
        hidden = self.i2h(combined)
        output = self.i2o(combined)
        output = self.softmax(output)
        return output, hidden
```

```
    def initHidden(self):
        return torch.zeros(1, self.hidden_size)
```

使用print()函数打印网络结构的结果如下：

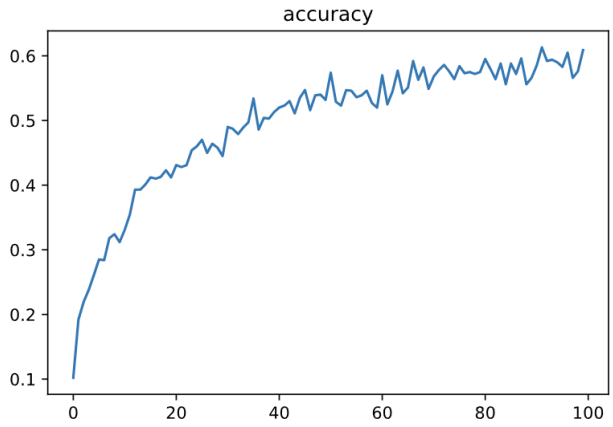
```
RNN(
  (i2h): Linear(in_features=185, out_features=128, bias=True)
  (i2o): Linear(in_features=185, out_features=18, bias=True)
  (softmax): LogSoftmax(dim=1)
)
```

loss图



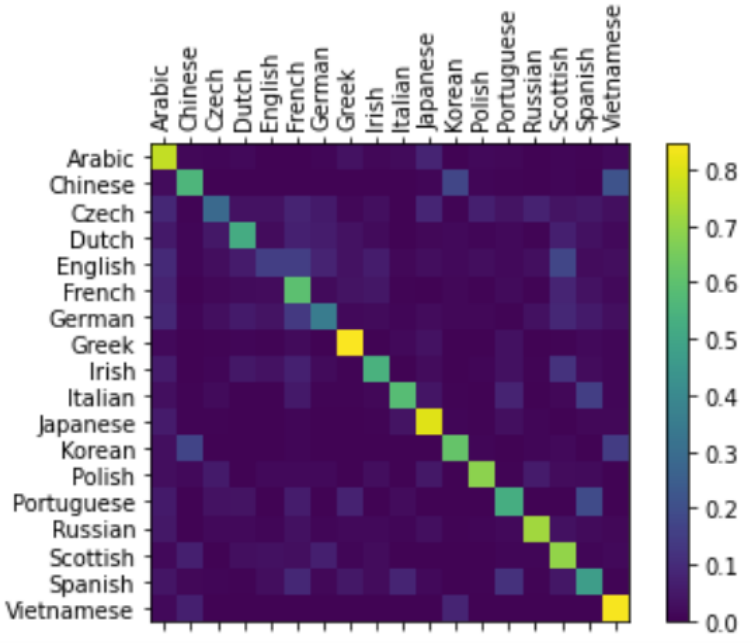
RNN的loss最后稳定在1.2左右，效果良好。

准确度图



RNN的准确率最后稳定在60%左右，效果良好。但还有巨大的提升空间，将使用Lstm进行优化。

预测矩阵图



可以看出，对角线十分清晰，预测效果良好。

Lstm 库

网络结构

作者先使用pytorch提供的Lstm库，**在最后一段展示自己手写的Lstm网络。**

```
class Lstm_official(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(Lstm_official,self).__init__()
        self.lstm=nn.LSTM(input_size,hidden_size,2)
        self.linear=nn.Sequential(
            nn.Linear(hidden_size,output_size),
            nn.LogSoftmax(dim=1)
        )
    def forward(self,input):
        x,(h_n,c_n)=self.lstm(input)
        # 只使用最后一次的输出
        x=self.linear(x[-1])
        return x
```

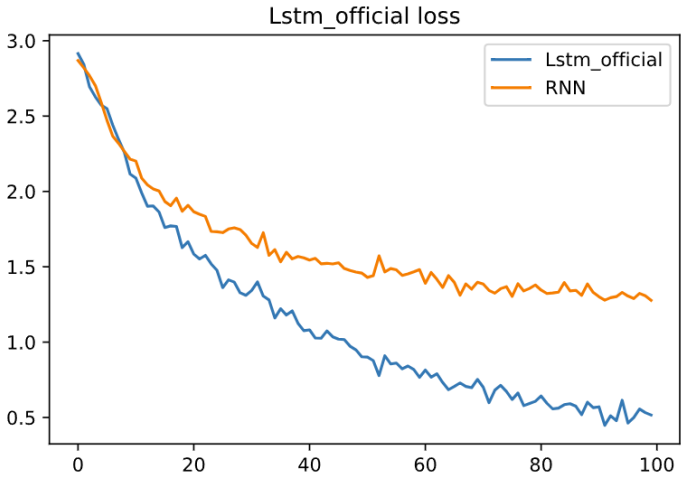
使用print()函数打印网络结构的结果如下：

```
Lstm_official(
  (lstm): LSTM(57, 128, num_layers=2)
  (linear): Sequential(
    (0): Linear(in_features=128, out_features=18, bias=True)
    (1): LogSoftmax(dim=1)
  )
)
```

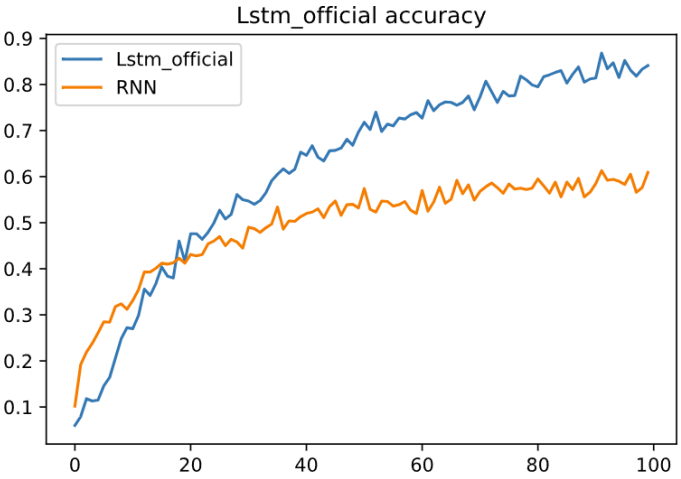
注意：调库实现Lstm，需要更改源代码中的train()函数，此时，将直接全部输入sequence。

```
def train(category_tensor, line_tensor):
    Lstm.zero_grad()
    #此时直接全部输入sequence
    output = Lstm(line_tensor)
    loss = criterion(output, category_tensor)
    loss.backward()
    optimizer.step()
    return output, loss.item()
```

loss图

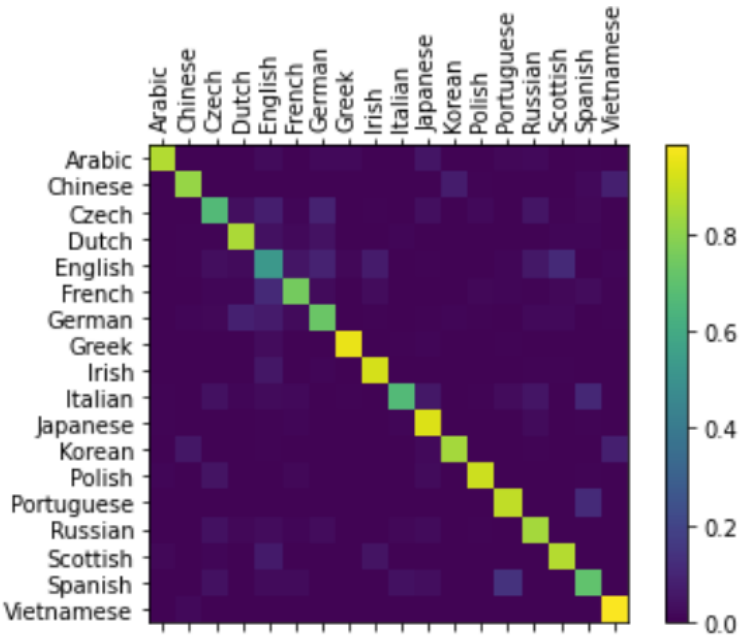


准确度



Lstm的准确率最后稳定在87%左右，效果良好，同时相较于RNN是巨大的提升。

预测矩阵图



Lstm 对角线相较于RNN更加清晰，证明Lstm模型效果更好。

Lstm 优于RNN的原因

RNN梯度消失或梯度爆炸

RNN在处理的Sequence长度很长时会产生梯度爆炸或消失，所以循环神经网络（RNN）实际上只能学习到短期的依赖关系。下面证明产生梯度爆炸或消失的原因：（**来源：智能计算系统课程PPT**）

损失函数对W的偏导为:

$$\frac{\partial L}{\partial W}=\sum_{t=1}^{\tau}\frac{\partial L^{(t)}}{\partial W}=\sum_{t=1}^{\tau}\sum_{k=1}^{t}\frac{\partial L^{(t)}}{\partial \hat{\boldsymbol{y}}^{(t)}}\frac{\partial \hat{\boldsymbol{y}}^{(t)}}{\partial \boldsymbol{o}^{(t)}}\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{h}^{(t)}}\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(k)}}\frac{\partial \boldsymbol{h}^{(k)}}{\partial W}$$

因为:

$$\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(k)}}=\prod_{i=k+1}^{t}\frac{\partial \boldsymbol{h}^{(i)}}{\partial \boldsymbol{h}^{(i-1)}}$$

根据推导可知序列损失函数对U和W的偏导为:

$$\frac{\partial L}{\partial W}=\sum_t\sum_{k=1}^{k=t}\frac{\partial L^{(t)}}{\partial \hat{\boldsymbol{y}}^{(t)}}\frac{\partial \hat{\boldsymbol{y}}^{(t)}}{\partial \boldsymbol{o}^{(t)}}\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{h}^{(t)}}\left(\prod_{i=k+1}^{t}W^{\top}\operatorname{diag}\left(1-\left(\boldsymbol{h}^{(i)}\right)^2\right)\right)\frac{\partial \boldsymbol{h}^{(k)}}{\partial W}$$

$$\frac{\partial L}{\partial U}=\sum_t\sum_{k=1}^{k=t}\frac{\partial L^{(t)}}{\partial \hat{\boldsymbol{y}}^{(t)}}\frac{\partial \hat{\boldsymbol{y}}^{(t)}}{\partial \boldsymbol{o}^{(t)}}\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{h}^{(t)}}\left(\prod_{i=k+1}^{t}W^{\top}\operatorname{diag}\left(1-\left(\boldsymbol{h}^{(i)}\right)^2\right)\right)\frac{\partial \boldsymbol{h}^{(k)}}{\partial U}$$

令$\gamma=\left\|\prod_{i=k+1}^{t}W^{\top}\operatorname{diag}\left(1-\left(\boldsymbol{h}^{(i)}\right)^2\right)\right\|_2$

当Sequence长度很长时，$t>>k$，就会产生爆炸或消失。

$$r\begin{cases}\to\infty,\frac{\partial L}{\partial U}\to\infty,\frac{\partial L}{\partial W}\to\infty\\ \to 0,\frac{\partial L}{\partial W}\to 0,\frac{\partial L}{\partial U}\to 0\end{cases}$$

Lstm门

LSTM通过“门”（gate）来控制丢弃或者增加信息，从而实现遗忘或记忆的功能，从而使得Lstm可以能学习到长期的依赖关系。

- 遗忘门 f_t 控制上一个时刻的内部状态 c_{t-1} 需要遗忘多少信息

  公式如下：

  $f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right)$

  本质上是以上一单元的输出 h_{t-1} 和本单元的输入 x_t为输入的sigmoid函数，为 c_{t-1}中的每一项产生一个在[0,1]内的值，来控制上一单元状态被遗忘的程度。

- 输入门i_t 控制当前时刻的候选状态 $\tilde{c}_t$有多少信息需要保存

  公式如下：

  $$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] + b_i\right)$$
  $$\tilde{C}_t = \tanh\left(W_C \cdot [h_{t-1}, x_t] + b_c\right)$$

  输入门和一个tanh函数配合控制有哪些新信息被加入。

- 输出门o_t 控制当前时刻的内部状态c_t有多少信息需要输出给外部状态 h_t

  公式如下：

  $$o_t = o(W_o[h_{t-1}, x_t]) + b_o$$
  $$h_t = O_t * \tan h(C_t)$$

  输出门用来控制当前的单元状态有多少被过滤掉。先将单元状态激活，输出门为其中每一项产生一个在[0,1]内的值，控制单元状态被过滤的程度。

LSTM 网络中，记忆单元 c可以在某个时刻捕捉到某个关键信息，并有能力将此关键信息保存一定的时间间隔。记忆单元 c 中保存信息的生命周期要长于短期记忆h，所以LSTM 可以更好的学习到长期依赖。

自己实现的Lstm

网络结构

作者自己实现的为两层Lstm，在myLstm中嵌套两个myLstmbase。

- myLstmbase: 就是将公式，用代码的形式写出，公式如下：

  $$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right)$$
  $$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] + b_i\right)$$
  $$\tilde{C}_t = \tanh\left(W_C \cdot [h_{t-1}, x_t] + b_c\right)$$
  $$o_t = o(W_o[h_{t-1}, x_t]) + b_o$$
  $$h_t = O_t * \tan h(C_t)$$

- myLstm: 简单的将两个myLstm顺序连接即可，输出接上线性连接层,最后再接上LogSoftmax，得出概率。

```
class myLstmbase(nn.Module):
    def __init__(self,input_sz,hidden_sz):
        super().__init__()
        self.input_size=input_sz
        self.hidden_size=hidden_sz
        #输入参数
        self.U_i=nn.Parameter(torch.Tensor(input_sz,hidden_sz))
```

```
        self.V_i = nn.Parameter(torch.Tensor(hidden_sz,hidden_sz))
        self.b_i = nn.Parameter(torch.Tensor(hidden_sz))

        #遗忘门参数
        self.U_f = nn.Parameter(torch.Tensor(input_sz, hidden_sz))
        self.V_f = nn.Parameter(torch.Tensor(hidden_sz, hidden_sz))
        self.b_f = nn.Parameter(torch.Tensor(hidden_sz))

        #记忆门参数
        self.U_c = nn.Parameter(torch.Tensor(input_sz, hidden_sz))
        self.V_c = nn.Parameter(torch.Tensor(hidden_sz, hidden_sz))
        self.b_c = nn.Parameter(torch.Tensor(hidden_sz))

        #输出门参数
        self.U_o = nn.Parameter(torch.Tensor(input_sz, hidden_sz))
        self.V_o = nn.Parameter(torch.Tensor(hidden_sz, hidden_sz))
        self.b_o = nn.Parameter(torch.Tensor(hidden_sz))

        self.init_weights()

    # 初始化，采用正态分布
    def init_weights(self):
        stdv = 1.0 / math.sqrt(self.hidden_size)
        for weight in self.parameters():
            weight.data.uniform_(-stdv, stdv)
    # 前向
    def forward(self,x,h_t,c_t):
        bs,seq_size = x.size()
        #计算
        x_t = x
        i_t = torch.sigmoid(x_t @ self.U_i + h_t @ self.V_i + self.b_i)
        f_t = torch.sigmoid(x_t @ self.U_f + h_t @ self.V_f + self.b_f)
        g_t = torch.tanh(x_t @ self.U_c + h_t @ self.V_c + self.b_c)
        o_t = torch.sigmoid(x_t @ self.U_o + h_t @ self.V_o + self.b_o)
        c_t = f_t * c_t + i_t * g_t
        h_t = o_t * torch.tanh(c_t)

        hidden_seq=h_t
        return hidden_seq, (h_t, c_t)

# 两层lstm
class myLstm(nn.Module):
    def __init__(self,input_sz,hidden_sz,output_size):
        super().__init__()
        self.input_size=input_sz
        self.hidden_size=hidden_sz
        # 第一层lstm
        self.lstm1 = myLstmbase(input_sz,hidden_sz)
        # 第二层lstm
        self.lstm2 = myLstmbase(input_sz,hidden_sz)
        # linear
        self.linear=nn.Sequential(
            nn.Linear(hidden_sz,output_size),
            nn.LogSoftmax(dim=1)
        )
    def forward(self,x,h_t,c_t):
        # layer1
        hidden_seq,(temp_h_t,temp_c_t) = self.lstm1(x,h_t,c_t)
```

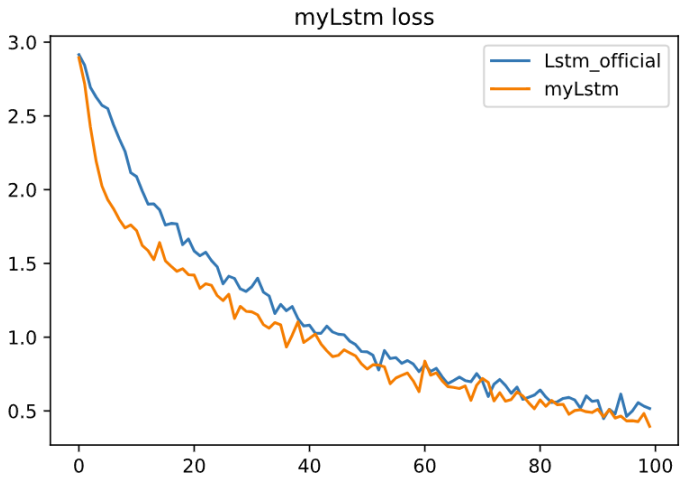
```
        # layer2
        hidden_seq,(temp_h_t,temp_c_t) = self.lstm2(x,temp_h_t,temp_c_t)
        # result
        result = self.linear(hidden_seq)
        return result,(temp_h_t,temp_c_t)
```

使用print()打印网络结构

由于myLstmbase是公式的乘法，所以可能使用Print函数无法有效打印，但是可以在后面的训练过程中验证正确性。

```
myLstm(
  (lstm1): myLstmbase()
  (lstm2): myLstmbase()
  (linear): Sequential(
    (0): Linear(in_features=128, out_features=18, bias=True)
    (1): LogSoftmax(dim=1)
  )
)
```

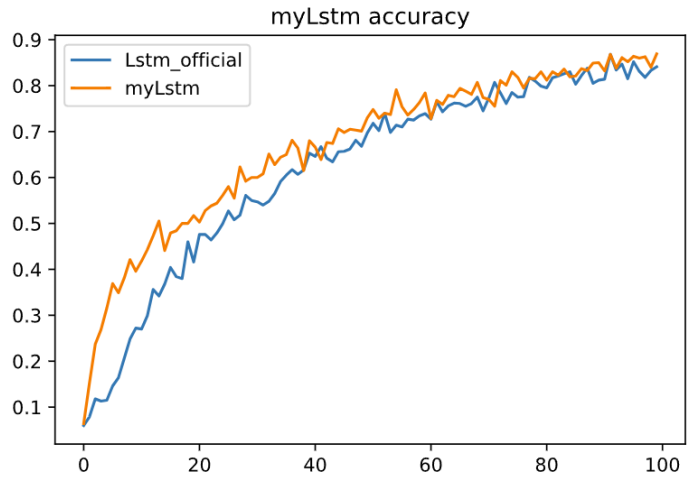
loss图



从图中可以看出，自己实现的Lstm与调库实现的Lstm的Loss曲线趋势基本相同，最终，自己实现的lstm甚至优于调库实现的Lstm。

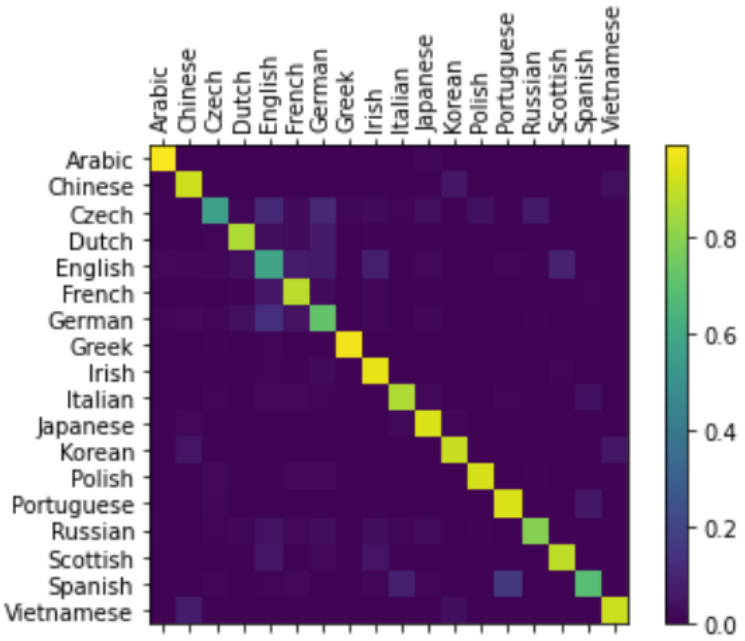
但是，自己实现的Lstm训练时间很长，远大于调库实现的Lstm。

准确度



从图中可以看出，自己实现的Lstm与调库实现的Lstm的accuracy曲线趋势基本相同，最终，自己实现的lstm甚至优于调库实现的Lstm。

预测矩阵图



Lstm 对角线清晰。