## 循环神经网络实验报告

姓名：杨鑫 学号：2011028

**一、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

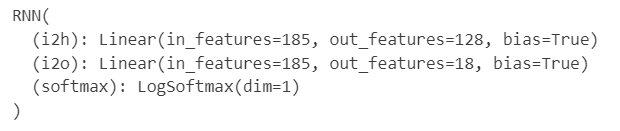
**二、报告内容**

* 老师提供的原始版本RNN网络结构（可用print(net)打印，复制文字或截图皆可）、在名字识别验证集上的训练loss曲线、准确度曲线图以及预测矩阵图
* 个人实现的LSTM网络结构在上述验证集上的训练loss曲线、准确度曲线图以及预测矩阵图
* 解释为什么LSTM网络的性能优于RNN网络（重点部分）
* 格式不限

**三、实验过程**

**1、原始RNN**

打印原始RNN的网络结构：

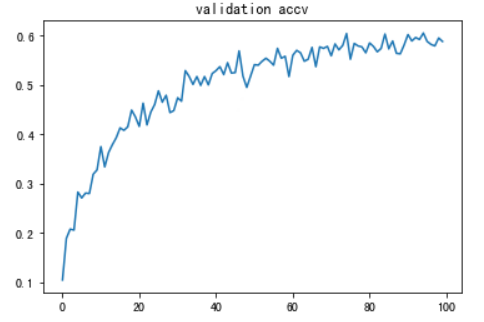


这是一个简单的循环神经网络（RNN）结构，包含三个层：

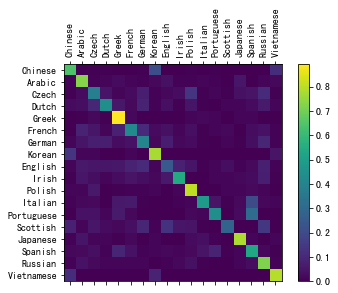
* 输入层到隐藏层的线性变换层（i2h），输入特征维度为185，隐藏层维度为128。
* 输入层到输出层的线性变换层（i2o），输入特征维度为185，输出维度为18。
* 输出层的softmax函数，用于将输出转化为概率分布。

设置学习率为0.005(经过测试发现0.005的时候训练效果最好)，设置迭代次数为100000，在CPU上训练的时间消耗为2m25s。训练loss曲线、准确度曲线图如下：



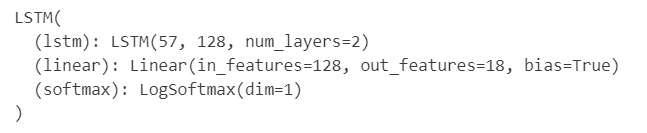


得到的RNN预测矩阵图如下：



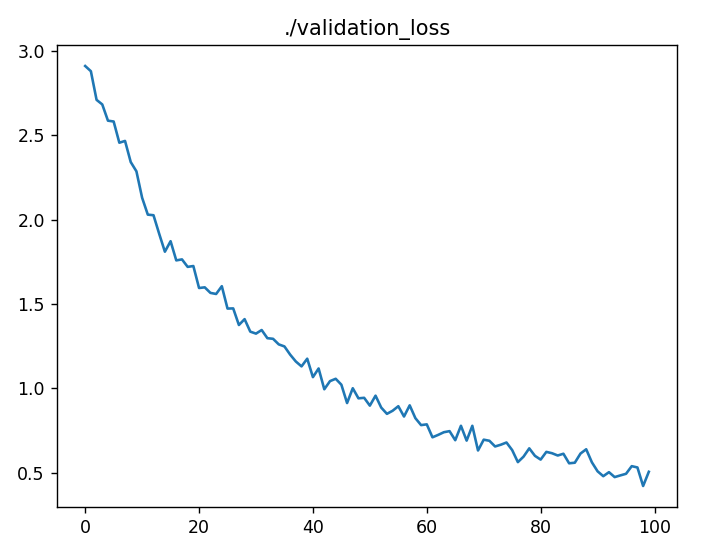
**2、利用Pytorch搭建LSTM**

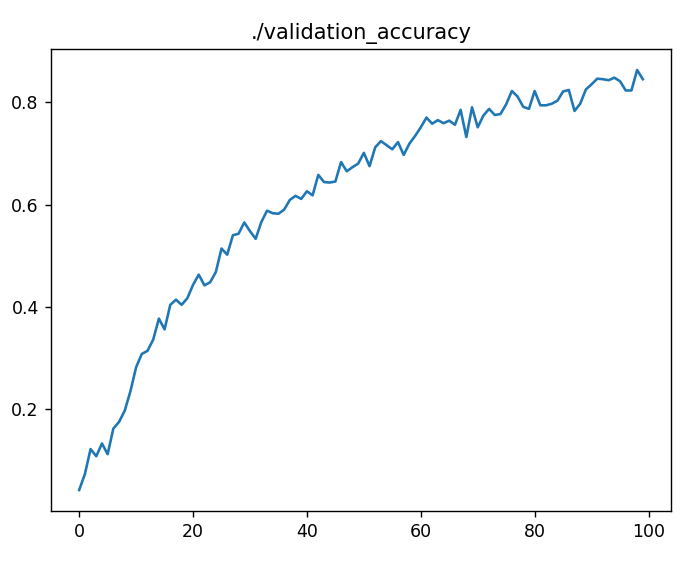
利用Pytorch搭建的LSTM网络结构如下：

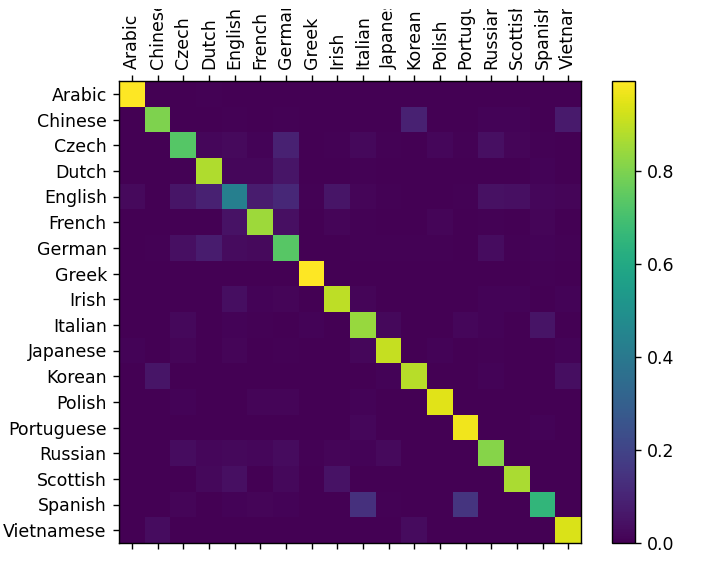


搭建了一个两层的LSTM网络，输入维度为57，隐藏层维度为128，输出维度为18。其中，LSTM层的输入维度为57，这个值是根据输入数据的特征维度而定的。在这个例子中，57是指数据集中所有字符的数量（大小写字母、标点符号、空格等），因此输入向量的长度为57。LSTM层的隐藏层维度为128，意味着LSTM层有128个隐藏单元。最后，输出层是一个线性层，将LSTM层的输出映射为18个类别（即18个国家）。输出通过LogSoftmax函数进行归一化，以便获得每个类别的概率分布。

设置学习率为0.08(经过测试发现0.08的时候训练效果最好)，设置迭代次数为100000，在CPU上训练的时间消耗为3m34s。训练loss曲线、准确度曲线图如下：





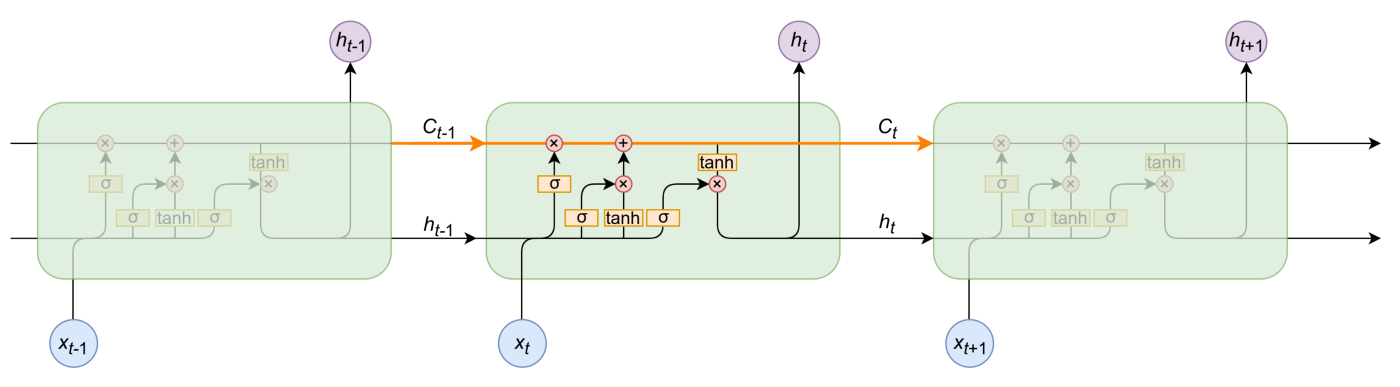


可以看到经过LSTM训练后的模型的准确率显著提高。

**3、手动实现LSTM**

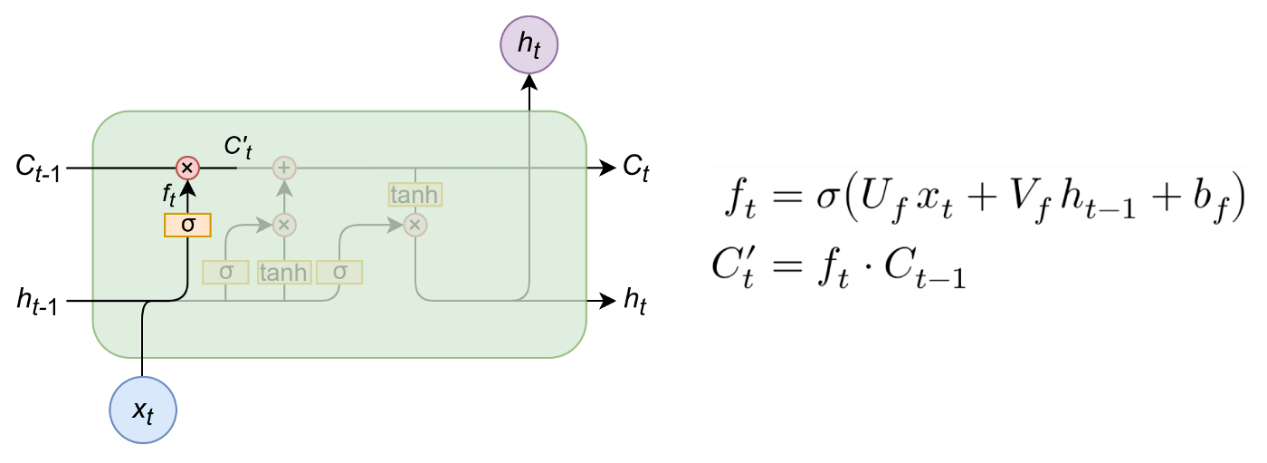
在动手实现LSTM之前需要了解一下LSTM的工作原理，LSTM是RNN网络中最有趣的结构之一，不仅仅使得模型可以从长序列中学习，还创建了长短期记忆模块，模块中所记忆的数值在需要时可以得到更改。

**LSTM的结构如下**：



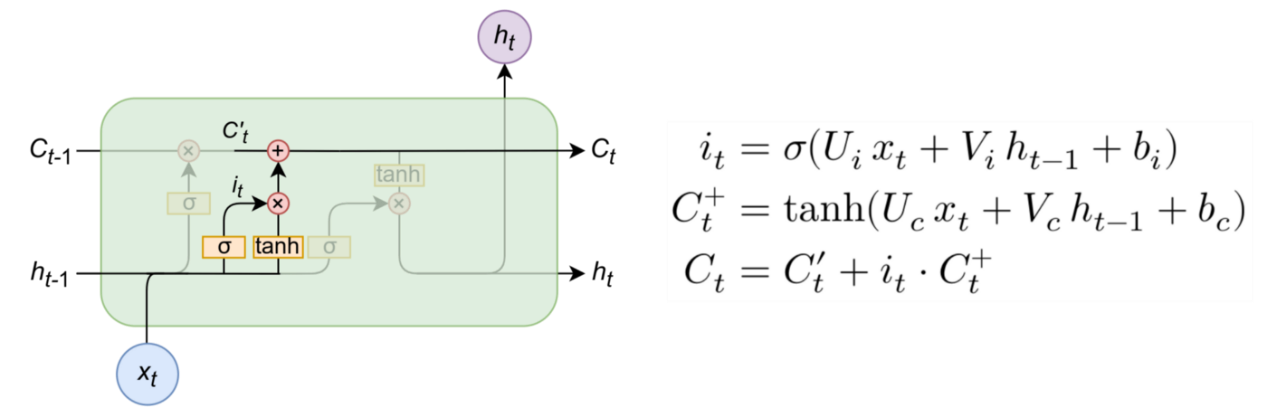
LSTM由门控单元组成，一种可由数学运算后控制信息是否传递或保持的计算结构，这因为这种结构，使得模型可以决定是在长期还是短期记忆中进行输出。

**遗忘门**：



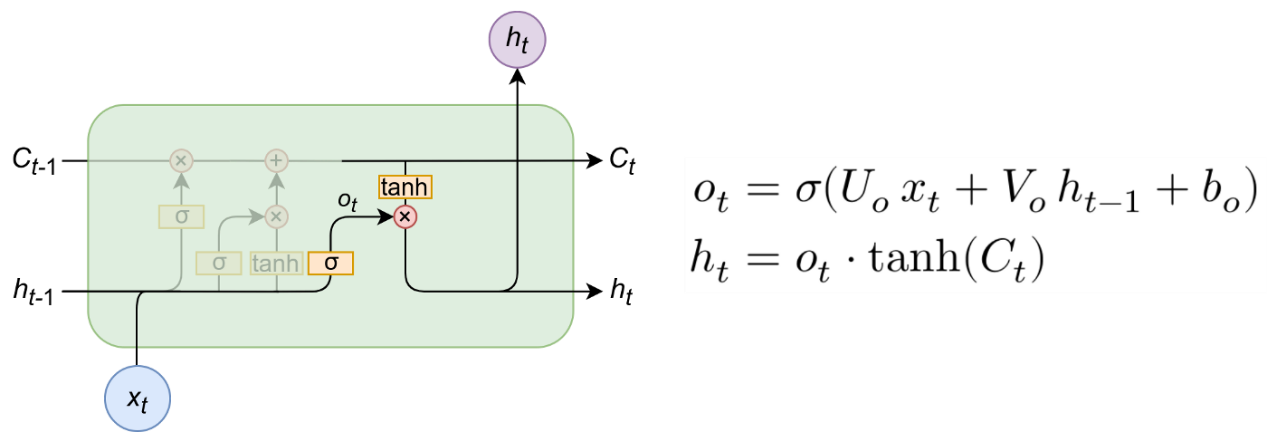
遗忘单元可以将**输入信息和隐藏信息进行信息整合**，并进行信息更替，更替步骤如右图公式，其中**与乘上权重矩阵后，加上偏置项后，经过激活函数**，此时输出值为位于[0,1]之间,并将上一个时间步的与激活函数输出值相乘，更新为

**输入门**：



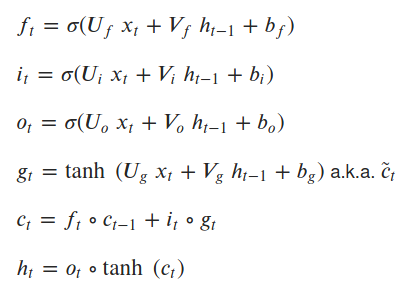
当有输入进入时，**输入门会结合输入信息与隐藏信息进行整合**，并对信息进行更替过程与过程类似，中间公式使用了tanh函数，可以将输出缩放到[-1,1]之间，再更新

**输出门：**



输出门也会对输出过程进行控制，与输入门不同的是，输出门使用tannh激活函数。

因此最后LSTM的公式如下：



因此使用pytorch实现的代码如下：

|  |
| --- |
| class myLSTM(nn.Module):      def \_\_init\_\_(self, input\_size, hidden\_size, output\_size):          super(myLSTM, self).\_\_init\_\_()          self.lstm = CustomLSTM(input\_size, hidden\_size)          self.linear = nn.Linear(hidden\_size, output\_size)          self.softmax = nn.LogSoftmax(dim=1)      def forward(self, x):          out, (h\_n, c\_n) = self.lstm(x)          out = (self.linear(out[-1]))          out = self.softmax(out)          return out  class CustomLSTM(nn.Module):      def \_\_init\_\_(self, input\_dim, hidden\_dim):          super(CustomLSTM, self).\_\_init\_\_()          self.input\_dim = input\_dim          self.hidden\_dim = hidden\_dim          """input gate"""          self.linear\_i\_x = nn.Linear(self.input\_dim, self.hidden\_dim)          self.linear\_i\_h = nn.Linear(self.hidden\_dim, self.hidden\_dim)          self.linear\_i\_c = nn.Linear(self.hidden\_dim, self.hidden\_dim)          self.i\_sigmod = nn.Sigmoid()          """forget gate"""          self.linear\_f\_x = nn.Linear(self.input\_dim, self.hidden\_dim)          self.linear\_f\_h = nn.Linear(self.hidden\_dim, self.hidden\_dim)          self.linear\_f\_c = nn.Linear(self.hidden\_dim, self.hidden\_dim)          self.f\_sigmod = nn.Sigmoid()          """cell memeory"""          self.linear\_c\_x = nn.Linear(self.input\_dim, self.hidden\_dim)          self.linear\_c\_h = nn.Linear(self.hidden\_dim, self.hidden\_dim)          self.c\_tanh = nn.Tanh()          """output gate"""          self.linear\_o\_x = nn.Linear(self.input\_dim, self.hidden\_dim)          self.linear\_o\_h = nn.Linear(self.hidden\_dim, self.hidden\_dim)          self.linear\_o\_c = nn.Linear(self.hidden\_dim, self.hidden\_dim)          self.o\_sigmod = nn.Sigmoid()          """hidden memory"""          self.h\_tanh = nn.Tanh()      def input\_gate(self, x, h, c):          return self.i\_sigmod(self.linear\_i\_x(x) + self.linear\_i\_h(h) + self.linear\_i\_c(c))      def forget\_gate(self, x, h, c):          return self.f\_sigmod(self.linear\_f\_x(x) + self.linear\_f\_h(h) + self.linear\_f\_c(c))      def cell\_memory(self, i, f, x, h, c):          return f \* c + i \* self.c\_tanh(self.linear\_c\_x(x) + self.linear\_c\_h(h))      def output\_gate(self, x, h, c\_next):          o = self.o\_sigmod(self.linear\_o\_x(              x) + self.linear\_o\_h(h) + self.linear\_o\_c(c\_next))          return o \* self.h\_tanh(c\_next)      def hidden\_memory(self, c\_next, o):          return o \* self.h\_tanh(c\_next)      def init\_hidden\_cell(self, x):          """initial hidden and cell"""          h\_0 = x.data.new(x.size(0), self.hidden\_dim).zero\_()          c\_0 = x.data.new(x.size(0), self.hidden\_dim).zero\_()          return (h\_0, c\_0)      def forward(self, x, memory=None):          \_, seq\_sz, \_ = x.size()          hidden\_seq = []          if memory is not None:              h, c = memory          else:              h, c = self.init\_hidden\_cell(x)          for t in range(seq\_sz):              x\_t = x[:, t, :]              i = self.input\_gate(x\_t, h, c)  # (x.size(0), hidden\_dim)              f = self.forget\_gate(x\_t, h, c)              c = self.cell\_memory(i, f, x\_t, h, c)              o = self.output\_gate(x\_t, h, c)              h = self.hidden\_memory(c, o)              hidden\_seq.append(h.unsqueeze(0))          hidden\_seq = torch.cat(hidden\_seq, dim=0)          hidden\_seq = hidden\_seq.transpose(0, 1).contiguous()          return hidden\_seq, (h, c) |

经过调参发现在隐藏层层数为32、学习率为0.035的时候效果最好，其中validation loss和validation accv以及预测矩阵图如下：

