循环神经网络实战

PYTORCH 版

Dezeming Family

Dezeming

Art. No 0
ISBN 000–00–0000–00–0
Edition 0.0

Cover design by Dezeming Family

Published by Dezeming
Printed in China

目录

前言及简介

DezemingFamily 系列书和小册子因为是电子书，所以可以很方便地进行修改和重新发布。如果您获得了 *DezemingFamily* 的系列书，可以从我们的网站 *[https://dezeming.top/]* 找到最新版。对书的内容建议和出现的错误欢迎在网站留言。

0.1 本书前言

本书的意义同前面几本，为提供一个的简洁的 RNN 实战代码。

循环神经网络会把一些输出和中间层结果进行存储，作为下一次的输入，因此这种网络具有一定的记忆功能。可以想象到，对于同样的输入，可能由于前面的输入不同从而得到不同的输出，例如下面两句话：I love you. 和 I do not love you. 网络可以根据前面的信息判断当前的句子描述的是爱你还是不爱你。

在计算机图形学的实时渲染中（尽管用神经网络几乎无法实现“实时”），RNN 可以将前面几帧的图像作为依据，总结渲染物体的特征，结合 AutoEncoder 对当前帧的图像进行去噪。这种方法就是大名鼎鼎的 RAE。不过在本书中，我们并不会涉及这么复杂的例子。

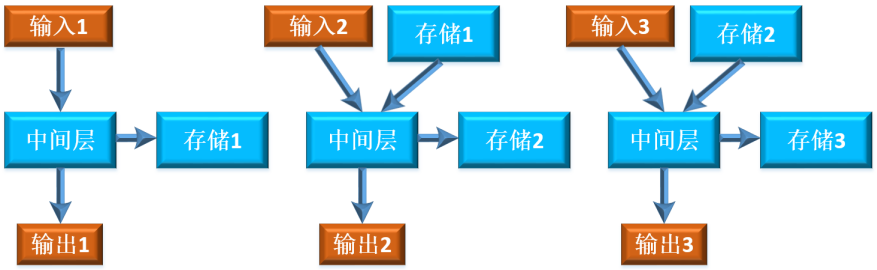
本书简单介绍 RNN 的基本原理，然后重点介绍 RNN 中最重要的一类：LSTM。我们学习如何搭建一个 LSTM 并运用到一个小项目中。

1. 循环神经网络

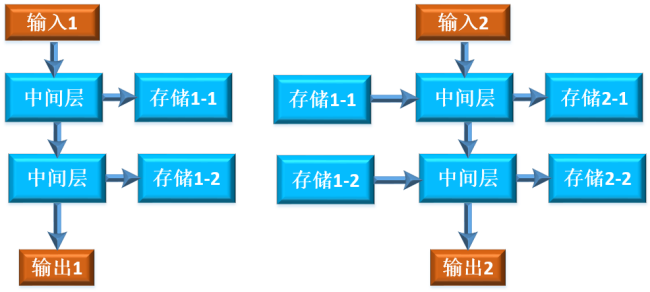
本章介绍循环神经网络的基本概念和一些简单的知识介绍。

1.1 基本的 RNN 结构

最简单的 RNN 结构如下（注意这是同一个神经网络，只是输入了三次）：

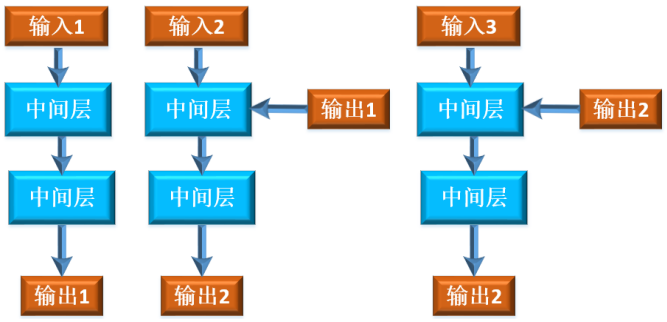


上面这个神经网络只有一个隐含层，扩展到多个隐含层表示如下：

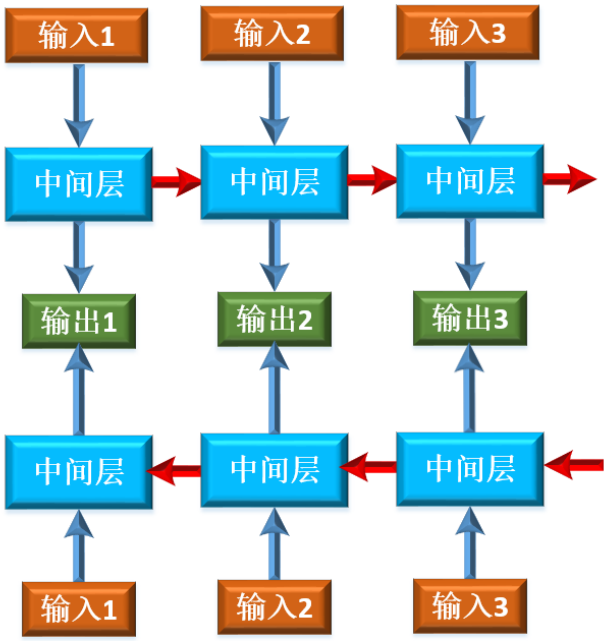


这种把中间层用来当做记忆的网络叫做 Elman 网络。

有的网络会把上一帧的输出值输入到当前帧的隐含层，叫做 Jordan 网络。由于输出是带有目标性的，所以我们能知道给 RNN 网络输入了什么东西：



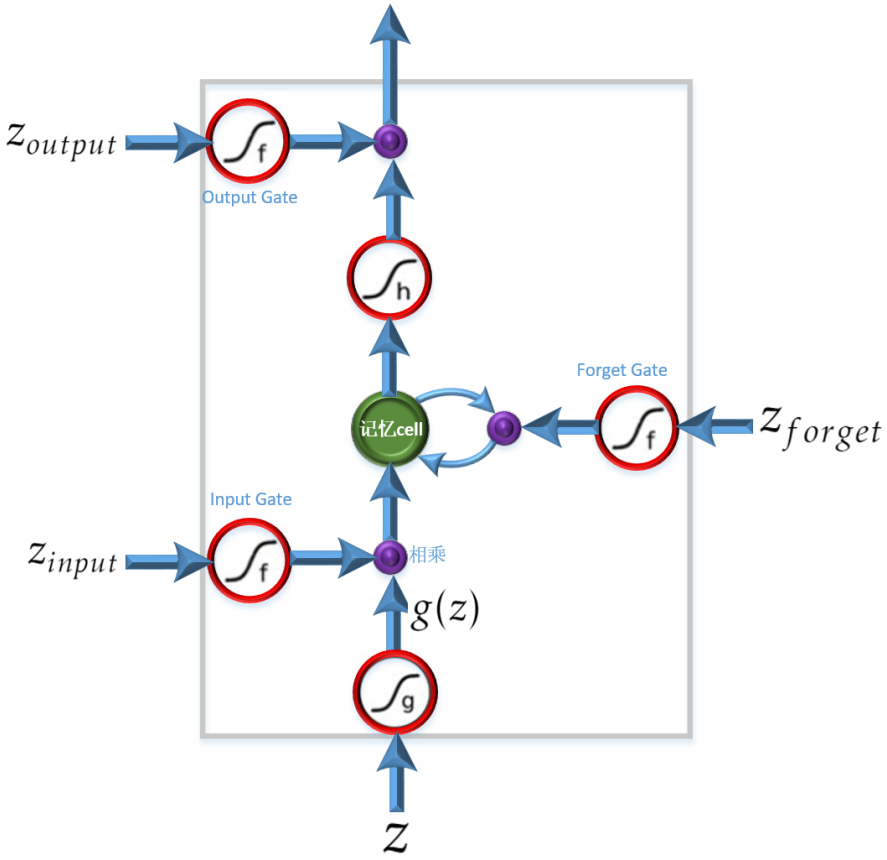
双向 RNN 就是训练一个网络，把数据从前往后训练保存一组参数，以及从后往前训练保存一组参数，比如对于这个网络（其实实际上就是建造两个网络），一个训练过程是依次输入 I + love + you；另一个训练过程是 you + love + I。我们把前后两个网络的结果一次作为输出：



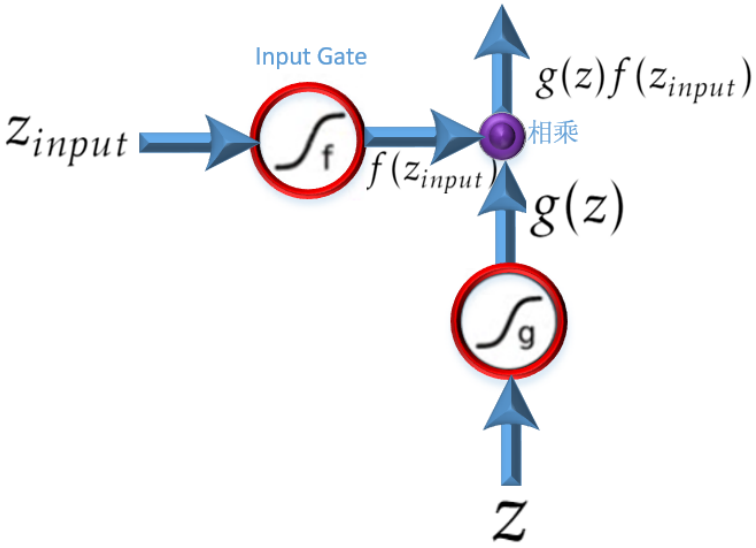
但是具体实现过程肯定还是有不少细节在里面的，甚至是最简单的 RNN，我们也需要考虑如何把输出或者记忆部分用来作为输入。

1.2 LSTM

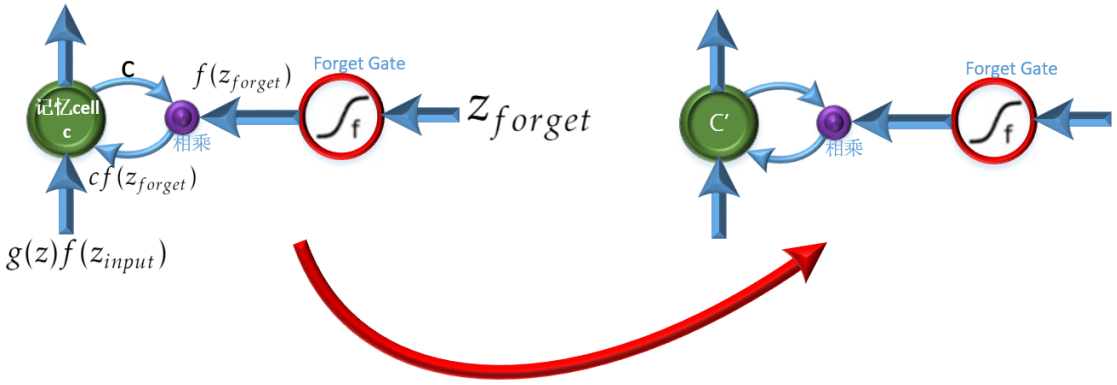
LSTM 是一个比较复杂的网络，也非常著名。这个网络全名为 long short-term memory 网络：



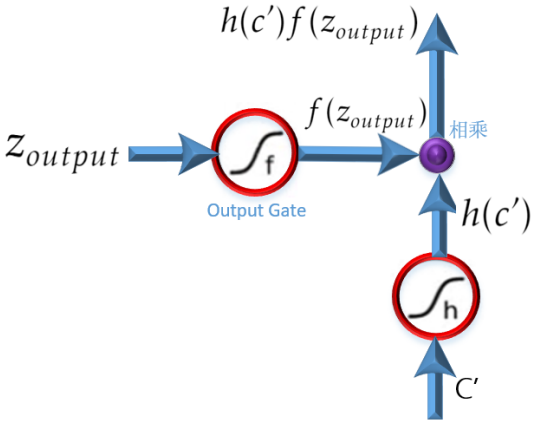
一个门控制输入，一个门控制记忆是否保留，一个门控制输出。上图的激活函数 f 表示门被打开的程度，是一个 sigmoid 函数（注意输出值在 0-1 之间）。输入值激活以后与控制输入的门的输出值相乘，得到 $g(z)f(z_{input})$：



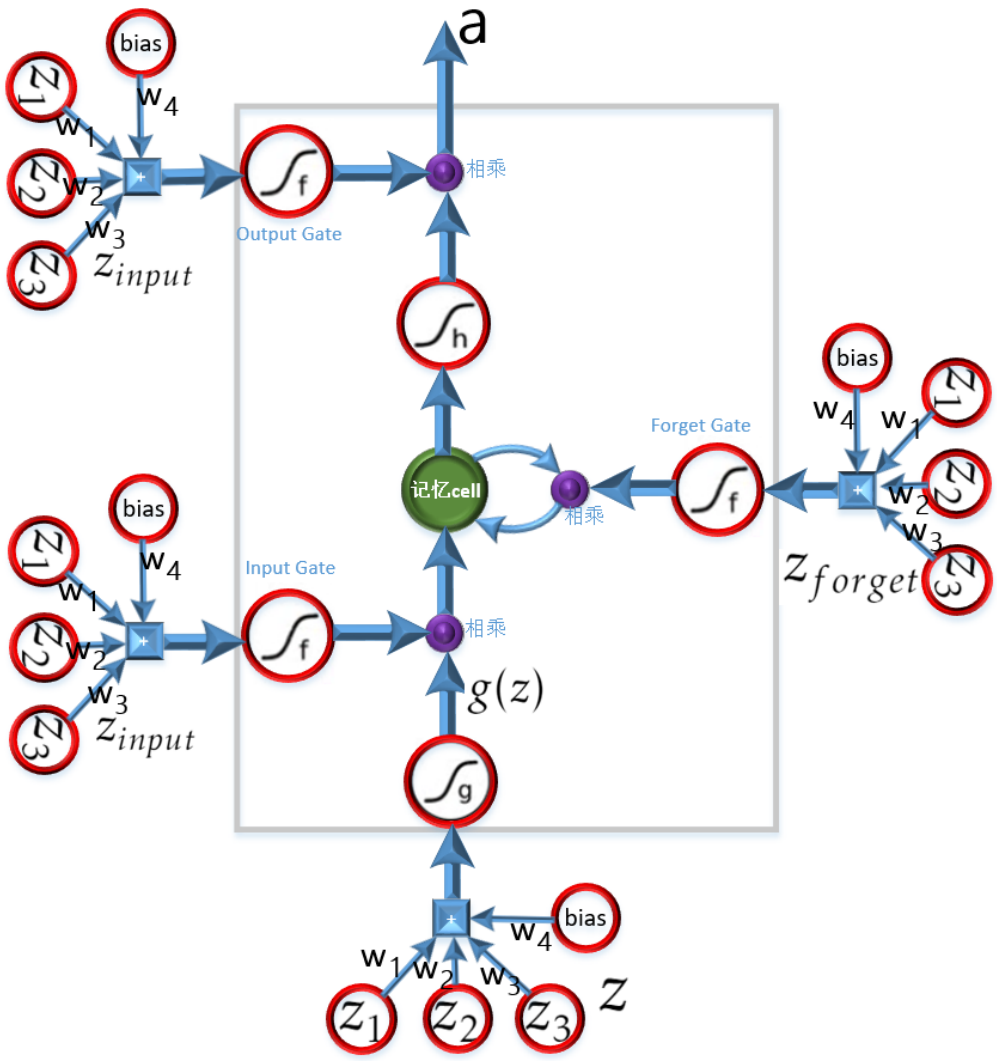
z_{forget} 通过 forget 门，与存储的记忆值相乘，得到 $cf(z_{forget})$，然后该值与记忆 cell 的值相加，得到 $c' = g(z)f(z_{input}) + cf(z_{forget})$，更新为新的存储值：



然后 c' 经过激活以后，与控制输出的门值相乘，得到输出：



注意这里的 z_{input}、z_{output} 和 z_{forget} 和 z 都可以是同样的输入向量，只是用来控制不同的门和作为输入。结构如下，注意网络的权重都是可以训练得到的：

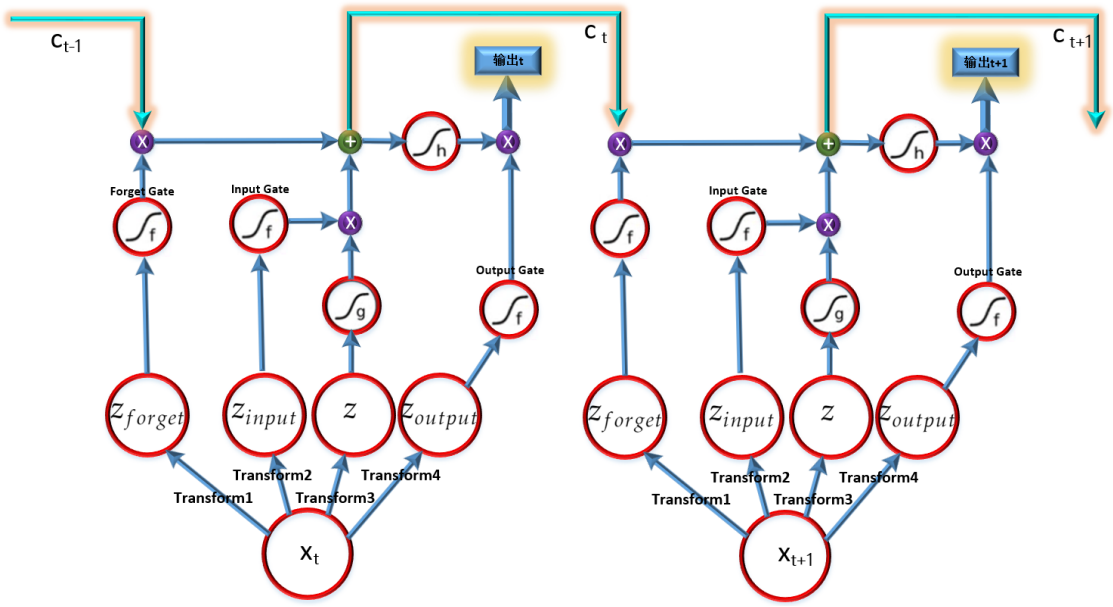


也可以将输入 x 进行不同的变换，得到不同的 z_{input}、z_{output} 和 z_{forget} 和 z 作为输入。我们可以把这一个单元作为这个网络的神经元，因此它一个神经元需要的参数是别的一般的网络的四倍。

1.3 LSTM 的流程

这个网络这么复杂，因此很多人可能会感觉它并不能很好的工作，或者感觉这玩意就是“根据意识胡乱捏造出来的”。其实我也是这么感觉的，但很多时序相关场景这种网络结构却出乎意料地好用。

本节参考 [1] 将流程提取出来，让读者能够更清晰地看到 LSTM 两个神经元之间的传递过程（做图做了将近半个小时）。



这个网络应该如何训练呢？首先我们需要定义一个损失函数，然后我们需要使用随着时间变化的 BP 方法（BPTT）。由于 RNN 的训练比较复杂，我当初花了很久时间看论文和一些技术分析（当时用的 keras）才勉强理解，但遗憾的是这对我在模型训练中并没有什么直接的帮助，因此这个实战书中我就不进行讲解了，同时也建议如果不是深入研究 NLP 的或者网络原理的人可以不用很详细的了解。我们只需要知道在实际训练中应该怎么定义损失函数和标签就可以了。

RNN 在实际训练中损失率可能会不断突变，不像卷积网络等的训练中正确率会不断抖动中稳步上式，损失率不断抖动中稳步下降。在 RNN 中会很有可能跳到 Loss 很大的位置，在论文 [2] 中有一些比较详细的分析。简单来说，RNN 更容易出现梯度消失和梯度爆炸的现象，大家也可以去 [3] 看看解释，个人认为解释的还算详细。

LSTM 可以控制梯度消失的问题，每个时间点，memory 里面的值是和输入相加的，一旦对 memory 有作用时，每次都会更新值（当 forget gate 打开时，梯度就并不会消失）。[4] 中有更详细的分析说明，大家可以去参考一下。以后有时间我也会写一下关于神经网络梯度消失和梯度爆炸的原因，以及为什么在 RNN 中梯度消失和梯度爆炸极其严重的问题。另外，LSTM 的简化版 Gated Recurrent Unit（GRU）会比 LSTM 更简单，大家可以自行搜寻一些原理和使用方法。

2. LSTM 实战

本章介绍如何搭建 *LSTM* 网络，并完成一个小项目的实战。

2.1 我们的任务和数据

本节代码见 chapter2-1.py。

一个最简单的 LSTM 程序不会涉及 NLP 的基本知识，不需要 word embedding 等词汇编码的方法，我们就用 LSTM 来做手写数字辨识，了解 LSTM 网络怎么搭建即可。

可能有人会很奇怪，为什么 LSTM 可以做手写数字辨识呢？不同的图片之间有什么时序关系吗？当然没有。但同一张图片的上下行之间却有时序关系，因此我们可以将每张图像一行一行地输入，用 LSTM 总结行与行之间的关系，用来做手写数字辨识。

我们还是按照以前的方法，将数据导入：

```
1 from torch.utils.data import Dataset
2 from torch.utils.data import DataLoader
3 from torchvision import datasets
4 from torchvision.transforms import ToTensor, Lambda
5 #数据集 28*28=784
6 training_data = datasets.MNIST(root='./myData', train=True, transform=
      ToTensor(), download=True)
7 test_data = datasets.MNIST(root='./myData', train=False, transform=
      ToTensor(), download=True)
8 train_dataloader = DataLoader(training_data, batch_size=64, shuffle=
      True)
9 test_dataloader = DataLoader(test_data, batch_size=64, shuffle=True)
```

以及用前面几本书的方法可以显示一下图片检查正确性：

```
import matplotlib.pyplot as plt
figure = plt.figure()
img, label = training_data[100]
plt.title(label)
#squeeze函数把为1的维度去掉
plt.imshow(img.squeeze(), cmap="gray")
plt.show()
print(img.shape)
```

2.2 LSTM 网络结构

本节代码见 chapter2-1.py。

我们先把 LSTM 的结构定义一下，然后再解释各个部分：

```
import torch.nn as nn
class LSTM_Net(nn.Module):
    def __init__(self):
        super(LSTM_Net, self).__init__()
        self.RNNNet = nn.LSTM(
            input_size=28,
            hidden_size=64,
            num_layers=1,
            bias = True,
            batch_first=True,
        )
        self.linear = nn.Sequential(
            nn.Linear(64, 32),
            nn.ReLU(),
            nn.Linear(32, 10)
        )
    def forward(self, x):
        r_out, (h_n, h_c) = self.RNNNet(x, None)
        out = self.linear(r_out[:, -1, :])
        return out
```

首先解释一下 nn.LSTM：

- input_size 表示输入特征的维数（可以理解为是 embedding 后词向量的维度，我们的输入维度是 28）；

- hidden_size 表示 LSTM 中隐含层一层中 LSTM 神经元的个数；num_layers 表示循环神经网络的层数；
- bias 表示是否用偏置，默认使用偏置；
- batch_first 表示是否输入数据第一维是 batchsize，默认为 false，我们定义为 True，注意我们的输入数据 shape 应该是（batch_size,seq_length,embedding_dim），注意 embedding_dim 就是 input_size，seq_length 并不需要定义，因为网络不会限制输入数据的长度（比如一个句子有多少个单词，网络并不会严格限制这件事）；
- 还有一些没有出现在 nn.LSTM 上面代码的参数，例如 dropout=0 表示默认不使用 dropout；
- bidirectional=false 表示默认不使用双向 LSTM。

至于 self.RNNNet 函数的输入：

- 输入到 self.RNNNet 的格式是 (input,(h_0,c_0))，因为我们前面定义了 batch_first=True，所以 input 的 shape 就是 (batch_size,seq_length,input_size)；
- h_0 和 c_0 都是张量，shape 都是 (batch_size,num_layers*num_directions,hidden_size)，它表示在这个 batch_size 每个输入要对应的 RNN 的 LSTM 单元初始状态，如果不提供的话初始状态就是 0，其中 h_0 表示初始隐藏状态，c_0 表示初始细胞状态
- bidirectional=True 时 num_directions=1，否则就是 0

self.RNNNet 函数的返回值：

- h_0 表示最后一个单词的隐藏状态，c_0 表示句子最后一个单词的细胞状态，注意它们都与句子的长度无关

我们需要明确的是 LSTM 中隐藏状态其实就是输出，细胞状态才是 LSTM 中隐藏的的中间层，因此 output[-1] 与 h_n 是相等的（因为都是输出）。

注意如果网络的 nn.LSTM 定义的输入中 batch_first=False，上面的所有 batch_size 这一维都会与第二维的进行交换。

r_out[:, -1, :] 的意义：

- 将第二维只保留最后一个元素。即原来的输出 shape 是 (batch_size,seq_length,input_size)，现在是 (batch_size,input_size)。我们只需要把序列中最后一个输入得到的输出送给后面的线性结构即可。

2.3 开始训练网络

本节代码见 chapter2-1.py。

首先定义一些网络要训练前的参数：

```
rnn = LSTM_Net().cuda()
print(rnn)
import torch
optimizer = torch.optim.Adam(rnn.parameters(), lr=0.01)
loss_function = nn.CrossEntropyLoss()
```

然后就是训练网络的循环：

```
epochs = 100
for epoch in range(epochs):
    rnn.train()
    for step, (x, y) in enumerate(train_dataloader):
        # 训练网络
    rnn.eval()
    with torch.no_grad():
        # 测试网络
```

训练网络的代码如下，前一个 28 表示一共 28 行（seq_length），后一个 28 表示每个 sequence 是 28 维的向量（embedding_dim）。view 函数相当于 resize，把 [64,1,28,28] 的 shape 变换为到 [64,28,28]。

```
  x = x.view(-1, 28, 28).cuda()
  #上面的view函数也可以使用squeeze函数来代替，去掉为1的维度
  #x = x.squeeze().cuda()
  y = y.cuda()
  output = rnn(x)
  loss = loss_function(output, y)
  optimizer.zero_grad()
  loss.backward()
  optimizer.step()
```

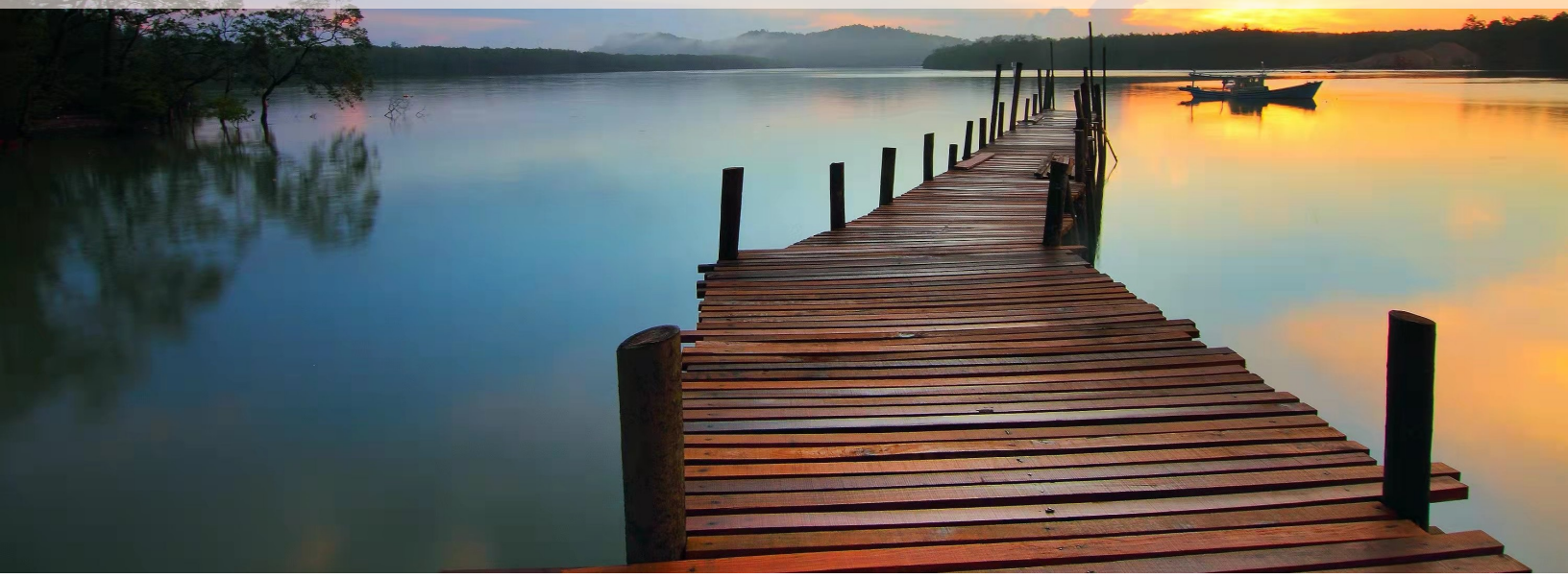
测试网络的代码如下：

```
  test_loss = 0
  correct = 0
  num_batches = len(test_dataloader)
  size = len(test_dataloader.dataset)
  for X, y in test_dataloader:
      X = X.view(-1, 28, 28).cuda()
      y = y.cuda()
      pred = rnn(X)
      test_loss += loss_function(pred, y).item()
```

```
        correct += (pred.argmax(1) == y).type(torch.float).sum().item()
    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100 * correct):>0.1f}%, Avg loss:
        {test_loss:>8f} \n")
```

现在我们的网络就可以开始用来训练了。在我的训练过程中看到，一个 epoch 以后，正确率就能达到百分之 95 以上，然而接下来的训练中，正确率会在 97% 到 98% 之间跳动。

Bibliography



[1] https://www.bilibili.com/video/BV1JE411g7XF?p=20

[2] On the difficulty of training Recurrent Neural Networks https://arxiv.org/abs/1211.5063

[3] https://zhuanlan.zhihu.com/p/28687529

[4] https://zhuanlan.zhihu.com/p/28749444

[5] https://blog.csdn.net/m0_45478865/article/details/104455978

[6] https://blog.csdn.net/Jaster_wisdom/