从零实现深度循环神经网络

荣耀学院

2023 年 03 月

第一版序言

从零实现循环神经网络，并解释每一步。

本书可以视为https://kunniaa.com《深度学习之 RNN》的后续。

目 录

第一章　实现一个最简单的循环神经网络 rnn

最简单的循环神经网络 rnn，一个输入层，一个隐层，一个输出层。输入样本只有一个特征，输出目标值只有一个特征，隐层神经元的状态量也只有一个特征。

假设输入样本是 x_t，t 表示第 t 时刻。

那么，隐层神经元的输出是：

$$h_t = tanh(x_t \cdot w_{ih} + h_{t-1} \cdot w_{hh} + b_{ih})$$

其中，$tanh$ 表示 $tanh$ 函数：

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

w_{ih} 表示输入样本和隐层神经元之间的权重矩阵，因为输入样本和隐层神经元都是单特征，因此 w_{ih} 是一个实数变量。

h_{t-1} 表示隐层神经元上一状态的值。

w_{hh} 表示隐层神经元记录当前状态和上一状态权重矩阵，因为隐层神经元都是单特征，因此 w_{hh} 是一个实数变量。

b_{ih} 表示输入样本和隐层神经元之间之间的偏差 bias。

理论上而言，有两个 bias：输入样本和隐层神经元之间的偏差 bias 是 b_ih，隐层神经元当前值和隐层神经元上一时刻值的偏差 bias 是 b_hh，也就是说，h_t 写为：

$$h_t = tanh(x_t \cdot w_{ih} + b_{ih} + h_{t-1} \cdot w_{hh} + b_{hh})$$

实际上，可以将两个偏差 bias 合并成一个，将 b_{hh} 省略掉，因为计算在本质上是一样的。有些框架没有合并，大概率是为了理解方便。

输出层的输出 o_t：

$$o_t = h_t \cdot w_{ho} + b_{ho}$$

其中，w_{ho} 是隐层输出值和输出层神经元的权重矩阵，因为隐层神经元和输出层神经元都是单特征，因此 w_{ho} 是一个实数变量。b_{ih} 表示输入样本和隐层神经元之间之间的偏差 bias。

注意，输出层的输出，不需要 tanh 函数了。

代码如下：

```
import torch
from torch import autograd

# 最简单的循环神经网络rnn

# 学习速率
eta = 0.01

# 只学习一个样本
x = torch.tensor([.04])
y = torch.tensor([1.])

# 注意，权重初始化最好在(0,1)之间。
w_ih = torch.tensor([0.1], requires_grad=True)
b_ih = torch.tensor([0.2], requires_grad=True)
w_hh = torch.tensor([0.3], requires_grad=True)
w_ho = torch.tensor([0.4], requires_grad=True)
b_ho = torch.tensor([0.5], requires_grad=True)

# 初始化h_{t-1}
h_t_1 = torch.randn(1)

i = 1
while i < 300:
    h = torch.tanh(x*w_ih + b_ih + h_t_1*w_hh)
    o = torch.tanh(h*w_ho + b_ho)
    err = torch.pow(o-y, 2)
    # 更新隐层神经元状态值
    h_t_1 = h
    print('err = ', err)
    # 计算权重和偏差的梯度
    w_ih_g, b_ih_g, w_hh_g, w_ho_g, b_ho_g = \
        autograd.grad(err, [w_ih, b_ih, w_hh, w_ho, b_ho])

    # 更新权重和偏差
    w_ih = w_ih - eta*w_ih_g
    b_ih = b_ih - eta*b_ih_g
    w_hh = w_hh - eta*w_hh_g
    w_ho = w_ho - eta*w_ho_g
    b_ho = b_ho - eta*b_ho_g

    i += 1
```

输出结果:

```
err =  tensor([0.2531], grad_fn=<PowBackward0>)
err =  tensor([0.2128], grad_fn=<PowBackward0>)
err =  tensor([0.1990], grad_fn=<PowBackward0>)
...
err =  tensor([0.0159], grad_fn=<PowBackward0>)
err =  tensor([0.0159], grad_fn=<PowBackward0>)
err =  tensor([0.0158], grad_fn=<PowBackward0>)
```

第二章　多输入多输出的循环神经网络 rnn

多输入多输出的循环神经网络 rnn，麻烦的地方是一些细节。

假设一个样本是 $x_t \in R^{1\times d}$，d 是样本的特征数。

那么，隐层神经元的输出是:

$$h_t = tanh(x_t \cdot w_{ih} + h_{t-1} \cdot w_{hh} + b_{ih})$$

其中，$h_t \in R^{1\times p}$，p 是隐层神经元状态变量的特征数，注意，p 不需要跟 d 一样，可以比 d 大，可以比 d 小，也可以相等。

$w_{ih} \in R^{d\times p}$，因为 x_t 的特征数是 d，且隐层神经元状态变量的特征数是 p，所以权重 w_{ih} 的维数一定是 $d \times p$，否则无法相乘。

$x_t \cdot w_{ih}$ 的结果是一个 $1 \times p$ 的向量。

h_{t-1} 的维数跟 h_t 是一样的，也是 $1 \times p$。

$h_{t-1} \cdot w_{hh}$ 的维数跟 h_t 一致，因为它们是隐层神经元前一时刻跟后一时刻的值，维数必须一致，所以 w_{hh} 的维数必须是 $p \times p$ 的。

b_{ih} 的维数，跟着 $x_t \cdot w_{ih}$ 走的，因此也是 $1 \times p$。比如说，如果 $xw + b$ 的 x 是 1×1 维，那 w 和 b 也是 1×1 维，如果 x 是 1×2 维，那么 w 是 2×2 维，b 是 1×2 维。

因此，$x_t \cdot w_{ih} + h_{t-1} \cdot w_{hh} + b_{ih}$ 的结果是 $1 \times p$ 维的。

$tanh(x_t \cdot w_{ih} + h_{t-1} \cdot w_{hh} + b_{ih})$，对 $1 \times p$ 维结果的每个成员都进行 $tanh$ 函数计算，得到隐层输出的最终结果。

这是用一个样本推导整个过程的情况。

如果把多个样本组合成一个二维矩阵作为输入，略有不同，此时，设样本

集为 $X \in R^{n\times d}$，X 是：

$$\begin{bmatrix} x_1 \\ ... \\ x_t \\ ... \\ x_n \end{bmatrix}$$

根据 $h_t = tanh(x_t \cdot w_{ih} + h_{t-1} \cdot w_{hh} + b_{ih})$ 计算，$x_t \cdot w_{ih} \in R^{n\times p}$，$h_{t-1} \cdot w_{hh} \in R^{1\times p}$，$b_{ih} \in R^{1\times p}$，似乎维数不同，不能相加。但是，从本质考虑，$X$ 是 n 个样本堆起来的结果，因此只要把 $x_t \cdot w_{ih} + h_{t-1} \cdot w_{hh}$ 加起来得到一个 $1 \times p$ 的向量，把这个向量从上到下逐个加到 $x_t \cdot w_{ih}$ 结果的每一行上即可。这个做法，学术上有个好听的名字，叫“广播机制”。

得到的 $n \times p$ 矩阵，每个成员都进行 $tanh$ 函数计算，就得到了隐层神经元的输出。

输出层的计算，相对而言就简单了，单样本输入的情况下：

$$o_t = h_t \cdot w_{ho} + b_{ho}$$

其中，$o_t \in R^{1\times q}$，$h_t \in R^{1\times p}$，$w_{ho} \in R^{p\times q}$，$b_{ho} \in R^{1\times q}$。

q 是输出值的特征数，w_{ho} 是隐层跟输出层的权重矩阵，b_{ho} 是隐层跟输出层的偏差 bias 向量。

如果输入是样本集，根据广播机制不难推导，原则是一样的。

代码如下：

```python
import torch
from torch import autograd

# 学习速率
eta = 0.01

# 只学习一个样本
x = torch.tensor([[4., 5., 6.]])
y = torch.tensor([100.,101., 102.,103. ])

w_ih = torch.randn([3, 2], requires_grad=True)
b_ih = torch.randn([1, 2], requires_grad=True)
w_hh = torch.randn([2, 2], requires_grad=True)
w_ho = torch.randn([2, 4], requires_grad=True)
b_ho = torch.randn([1, 4], requires_grad=True)

# 初始化h_{t-1}
h_t_1 = torch.randn([1, 2])

i = 1
while i < 300:
    h = torch.tanh(torch.mm(x, w_ih) + torch.mm(h_t_1, w_hh) + b_ih)
    o = torch.mm(h, w_ho)+b_ho
    err = torch.pow(o-y, 2).sum()
    print('err = ', err)
    # 更新隐层神经元状态值
    h_t_1 = h

    # 计算梯度
    w_ih_g, b_ih_g, w_hh_g, w_ho_g, b_ho_g = autograd.grad(err, [w_ih, b_ih,
                                                    w_hh, w_ho, b_ho])

    #更新权重矩阵和偏差
    w_ih = w_ih - eta*w_ih_g
    b_ih = b_ih - eta*b_ih_g
    w_hh = w_hh - eta*w_hh_g
    w_ho = w_ho - eta*w_ho_g
    b_ho = b_ho - eta*b_ho_g

    i += 1
```

输出结果:

```
err =  tensor(40127.1875, grad_fn=<SumBackward0>)
err =  tensor(35458.7344, grad_fn=<SumBackward0>)
err =  tensor(31331.3711, grad_fn=<SumBackward0>)
err =  tensor(27684.4414, grad_fn=<SumBackward0>)
...
err =  tensor(3.3528e-08, grad_fn=<SumBackward0>)
err =  tensor(3.3528e-08, grad_fn=<SumBackward0>)
err =  tensor(3.3528e-08, grad_fn=<SumBackward0>)
err =  tensor(3.3528e-08, grad_fn=<SumBackward0>)
err =  tensor(3.3528e-08, grad_fn=<SumBackward0>)
```

第三章　一点评述

前两章的例子，让网络只学习一个样本。学习多个样本，跟学习一个样本是一样的。

设计网络并不复杂，最重要的，是搞清楚矩阵计算的维数。维数不对，会导致两个问题：其一，无法运行，其二，变量初始化很茫然。

剩下的事情，交给 autograd。

第四章 pytorch 的 rnn 怎么用

先上一个最简单的代码

```
import torch
import torch.nn as nn

# 输入样本特征数
input_size = 3
# 隐层神经元状态量特征数
hidden_size = 2
# 隐层数量
num_layers = 1

# rnn实例化
rnn = nn.RNN(input_size, hidden_size, num_layers)

# 序列长度
seq_size = 4
# batch的长度
batch_size = 5

# 随机生成输入样本
input = torch.randn(seq_size, batch_size, input_size)

# 随机初始化h_{t-1}
num_directions = 2 if rnn.bidirectional else 1
h_t_1 = torch.randn(rnn.num_layers * num_directions, batch_size, hidden_size)

# 计算rnn输出
output, hn = rnn(input, h_t_1)
```

到 rnn 实例化为止，都是很好的理解的，网络结构的参数有不少，无法缺省的就这么几个：样本特征数，隐层神经元状态特征数，有几个隐层。注意，这个 rnn 不包括输出层，因此输出层特征数就不给出来了。当然，你也可以理解为：如果这个 rnn 做为一个多层网络的一层，那么样本特征数实际上是前一层输出值的特征数。

但 seq_size 和 $batch\_size$ 是什么？这个问题，我问 chatgpt 了，它的回答是这样的（侵删）：

假设有一个包含 100 个音频文件的数据集，每个音频文件的长度为 3 秒，每秒采样率为 16000，每个样本有 2 个特征。那么这个数据集的 shape 就是 (100, 48000, 2)，其中 100 表示 batch 大小，48000 表示 seq 长度，2 表示每个时间步的特征数。

在 pytorch 的 rnn 模型，输入数据的 shape 应该是 (seq_len, $batch\_size$, $input\_size$)。在上述例子中，需要将数据 reshape 为 (48000, 100, 2)。这样，模型每次输入的就是 100 个长度为 48000 的音频文件，每个时间步有 2 个特征。

pytorch 使用 (seq_len, $batch\_size$, $input\_size$) 的输入格式，主要是为了方便序列操作。其一，方便序列操作，在处理序列数据时，通常需要在时间维度上进行操作，例如 rnn 的循环过程，使用 (seq_len, $batch\_size$, $input\_size$) 的格式，可以更方便地进行时间维度上的操作；其二，方便分离和合并，在某些情况下，需要分离和合并数据集中的 batch 和 seq_len 维度，使用 (seq_len, $batch\_size$, $input\_size$) 的格式，可以方便地使用 pytorch 的 split 和 cat 函数来进行分离和合并；其三，方便 GPU 计算，在 GPU 上计算时，pytorch 默认使用 $batch\_size$ 作为第一维度，这样可以更好地利用 GPU 的并行计算能力，但对于序列数据，时间维度往往更重要，使用 (seq_len, $batch\_size$, $input\_size$) 的格式可以更好地利用 GPU 的并行计算能力。

再往下，难解的地方，是 t_{t-1} 的初始化，这个有点复杂。首先，隐层神经元状态量的特征数是必须要有的，这没啥好说的。那为啥要有 $batch\_size$ 呢？因为，数据可能会并行处理，按照 batch 分配出去，因此每个 batch 都可能分配到不同的地方，因此都需要各自对应的隐层神经元，因此需要分别初始化。同时，pytorch 的 rnn，即可能是单向的，也可能是双向的，双向 rnn 的权重和偏差 bias 都有两套，因此隐层神经元的状态变量也是两套。所以，t_{t-1} 的初始化跟单双向、batch 数量、隐层神经元状态量的特征数都相关，pytorch 要求按照这个顺序进行初始化。

更进一步地，pytorch 的 rnn 对象有 8 个参数，依次是：

$input\_size$：样本 x 的特征的数量。

$hidden\_size$：隐层的每个神经元，会记录一个状态值，这个状态值有几个特征。注意，状态值的特征数量可以跟 $input\_size$ 不一样。

num_layers：有几个隐层。

nonlinearity：用 tanh 还是 relu。

bias：是不是用 bias，如果不用，就没有 b_ih 和 b_ho，默认当然用啊。

$batch\_first$：默认不用，此时，(seq, batch, feature)。如果用 (batch, seq, feature)。这个是为了兼容那些不习惯前者的人。

dropout：是否加一个 dropout 层，强化泛化能力。

bidirectional：双向 rnn，默认不是。

第五章　未完待续