RNN入门与实践

原创 叶虎 机器学习算法工程师 2017-10-13

作者：叶虎

编辑：黄俊嘉

引言

递归神经网络(Recurrent Neural Network, RNN)是神经网络家族的重要成员，而且也是深度学习领域中的得力干将，因为深度学习广泛应用的领域如语音识别，机器翻译等都有RNN的身影。与经典的神经网络不同，RNN主要解决的是样本数据为序列的建模问题，如语音序列，语言序列。因为对于序列数据来说，大部分情况下序列的每个元素并不是相互独立，其存在依赖关系，而RNN特别适合这类建模问题。本文会介绍RNN的原理及应用，并动手实现一个RNN预测模型。

RNN原理

RNN处理的是序列建模问题。给定一个长度为T输入序列{$x_0,x_1...,x_t,....,x_T$}，这里 表示的是序列在t时刻的输入特征向量，这里的t时刻并不一定真的指的是时间，只是用来表明这是一个序列输入问题。现在要得到每个时刻的隐含特征{$h_0,h_1...,h_t,....,h_T$} ，这些隐含特征用于后面层的特征输入。如果采用传统的神经网络模型，只需要计算：

$$h_t = f(Ux_t + b)$$

其中f为非线性激活函数。但是这样明显忽略了这是一个序列输入问题，即丢失了序列中各个元素的依赖关系。对于RNN模型来说，其在计算t时刻的特征时，不仅考虑当前时刻的输入特征x_T ，而且引入前一个时刻的隐含特征h_{t-1} ，其计算过程如下：

$$h_t = f(Ux_t + Wh_{t-1} + b)$$

显然这样可以捕捉到序列中依赖关系，可以认为是一个h_{t-1}记忆特征，其提取了前面t-1个时刻的输入特征，有时候又称h_{t-1}为旧状态，而h_t为新状态。因此，RNN模型特别适合序列问题。从结构上看，RNN可以看成有环的神经网络模型，如图1所示。不过可以将其展开成普通的神经网络模型，准确地说展开成T个普通的神经网络模型。但是这T个神经网络不是割立的，其所使用参数是一样的，即权重共享。这样每一个时刻，RNN执行的是相同的计算过程，只不过其输入不一样而已。所以本质上，RNN也只不过多个普通的神经网络通过权值共享连接而成。

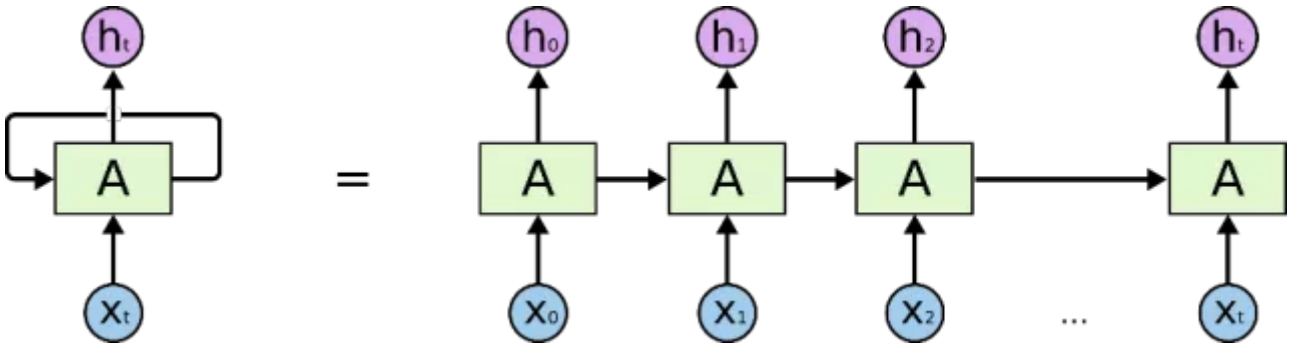

图1 RNN模型及展开简图

（来源：http://colah.github.io/posts/2015-08-Understanding-LSTMs/）

还有一点，RNN可以提取一组特征{h_0,h_1...,h_t,....,h_T}，但是并不是所有的特征都会送入后面的层，如果你只是需要根据输入序列进行分类，可能你仅需要最后时刻的特征h_T。这和具体的应用场景相关。

RNN训练

RNN模型像其他神经网络模型一样也是采用梯度下降法训练，相应的也需要计算梯度。计算梯度也是采用BP算法，但是由于RNN的特殊性，其对应的BP算法又称为BPTT(Backpropagation Through Time)。BPTT的背后含义是梯度还要在时间层进行反向传播，这很好理解，比如h_t的梯度h_{t-1},....,h_0还要对做贡献。这从数学公式上可以看出来的，本质上还是链式规则。但是你可能知道梯度消失的问题，在RNN模型中其同样存在。梯度消失的问题在RNN上表现为h_t的梯度传播距离可能有限，这带来的一个直接后果是：RNN对长依赖序列问题(long-term dependencies)无效。这使得经典的RNN模型的应用很受限，所以才会出现RNN的变种如LSTM，它们可以很好地解决这类问题。

RNN应用

RNN主要应用在输入为序列数据的业务场景。其中一个很重要的领域是自然语言处理，如语言模型及机器翻译等。RNN还可以对具有周期性特征的数据建立预测模型。对于序列问题，可以用下图来说明RNN的应用：

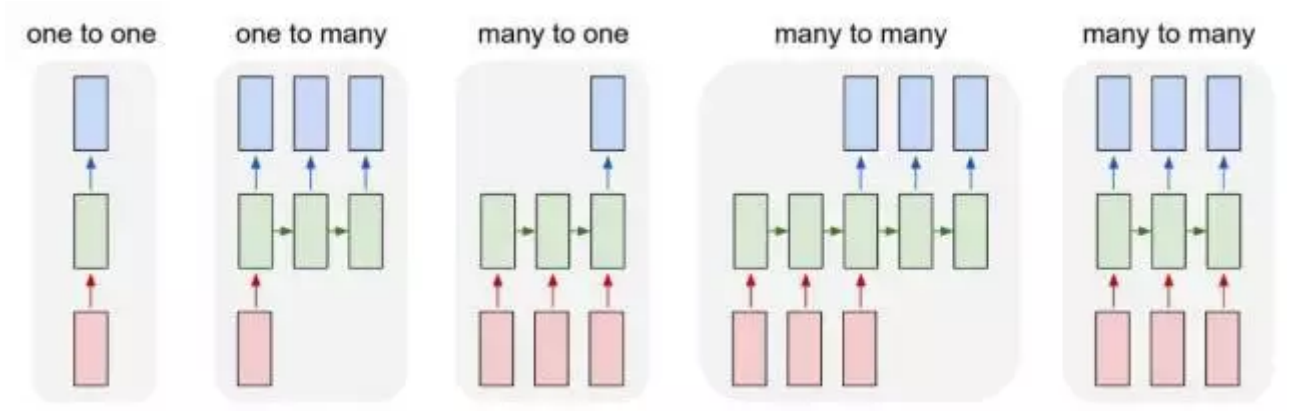

图2 RNN处理序列问题（来源：cs231n）

其中one to one是典型的神经网络的应用，给定一个输入，预测一个输出。而其他的情形都需要应用RNN模型。one to many的一个例子是图像标注(Image Captioning)，输入一个图片，得到对图片的语言描述，这是一个序列输出。对于many to one，其应用实例如情感分类(Sentiment Classification)，给定一句话判断其情感，其中输入是序列。第一种Many to many的典型应用场景是机器翻译，比如一句英文，输出一句中文，这时输入与输出都是序列。第二种many to many可以应用在视频分类问题(Video classification on frame level)，输入一段视频，对每一帧图片分类。因此，可见RNN模型广泛应用在各种业务场景中。

RNN实践

最后我们使用RNN模型实现一个简单的二进制加法器。任何一个整数都可以用一个二进制串来表示，给定两个二进制串，我们希望生成表示其和的二进制串。一个二进制串可以看成一个序列，这可以用RNN来搭建模型。先上图来说明：

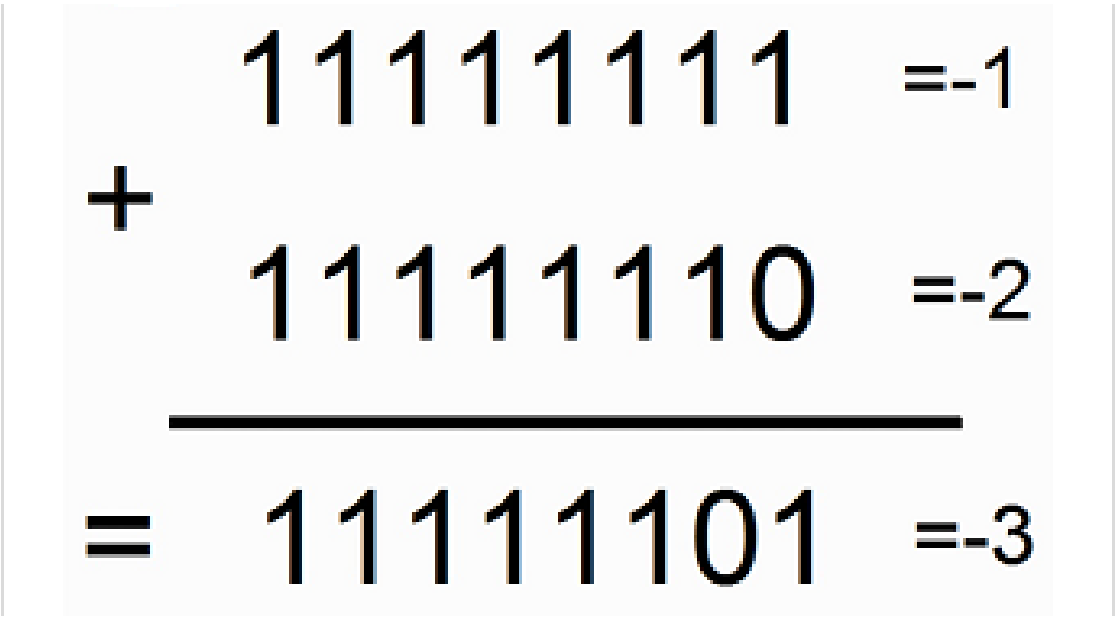

图3 二进制加法器（来源：angelfire.com）

二进制器加法器从左向右开始计算，通过两个运算数对应位上二进制数来得到新的二进制数。但是你要考虑运算溢出的问题，图上彩色方框中的1表示的是运算溢出后的“携带位”，你需要将其传递给下一位的运算。好吧，这是序列依赖关系，到了RNN发挥作用了。你就想象着上一个时刻的隐含特征保存这个“携带位”信息就可以了，这样当前时刻的运算就可以捕获到前面运算溢出得到的“携带位”。不过，这里的时刻指的是位置。

那么，现在开始设计这个RNN模型，首先肯定的这是many to many的例子。假定二进制串长度为L，那么时间步长为L，而且每个时刻的输入特征的维度是2。利用RNN模型，我们可以得到每个时刻的隐含特征，这个特征维度大小可以自定义，这里我们取16。将隐含特征送入输出层，得到预测结果，我们希望预测输出的维度是1，并且值限制在0和1这两个数。此时可以使用sigmoid激活函数，将值限制在[0,1]范围内，这个值大于0.5取1，反之取0。Python实现的代码如下：

```
import numpy as np
# sigmoid
```

```
# sigmoid
def sigmoid(x):
    return 1.0 / (1.0 + np.exp(-x))
# sigmoid导数
def sigmoid_derivative(output):
    return output * (1.0 - output)
# 生成整数与二进制数转化字典
int2binary = {}
binary_dim = 8
largest_number = pow(2, binary_dim)
binary = np.unpackbits(np.array([range(largest_number)], dtype=np.uint8).T,
                       axis=1)
for i in range(largest_number):
    int2binary[i] = binary[i]

# 模型参数
input_dim = 2
hidden_dim = 16
output_dim = 1
learing_rate = 1e-1

# 初始化模型参数
# 模型: h(t) = sigmoid(Ux + Vh(t-1)) -> output(t) = sigmoid(Wh(t))
U = np.random.randn(input_dim, hidden_dim)
V = np.random.randn(hidden_dim, hidden_dim)
W = np.random.randn(hidden_dim, output_dim)

# 初始化参数梯度
dU = np.zeros_like(U)
dV = np.zeros_like(V)
dW = np.zeros_like(W)

iterations = 20000
# 训练过程: 不使用batch
for i in range(iterations):
    # 生成一个简单的加法问题 （a+b = c), a, b 除以2防止c溢出
      a_int = np.random.randint(largest_number / 2)
    a = int2binary[a_int]
    b_int = np.random.randint(largest_number / 2)
    b = int2binary[b_int]

    c_int = a_int + b_int
    c = int2binary[c_int]

    d = np.zeros_like(c)
    # 训练样本
      X = np.array([a, b]).T
    y = np.array([c]).T

    loss = 0  # 损失函数

      hs = []  # 保存每个时间步长下的隐含特征
      hs.append(np.zeros((1, hidden_dim)))  # 初始化0时刻特征为0
```

```
os = []  # 保存每个时间步长的预测值

  # forward过程
  for t in range(binary_dim):
    # 当前时刻特征
       xt = X[binary_dim - t - 1]
    # 隐含层
       ht = sigmoid(xt.dot(U) + hs[-1].dot(V))
    # 输出层
       ot = sigmoid(ht.dot(W))
    # 存储结果
       hs.append(ht)
    os.append(ot)
    # 计算loss，采用L1
    loss += np.abs(ot - y[binary_dim - t - 1])[0][0]
    # 预测值
       d[binary_dim - t - 1] = np.round(ot)[0][0]

# backward过程
  future_d_ht = np.zeros((1, hidden_dim))  # 从上一个时刻传递的梯度
  for t in reversed(range(binary_dim)):
    xt = X[binary_dim - t - 1].reshape(1, -1)
    ht = hs[t+1]
    ht_prev = hs[t]
    ot = os[t]
    # d_loss/d_ot
    d_ot = ot - y[binary_dim - t - 1]
    d_ot_output = sigmoid_derivative(ot) * d_ot
    dW += ht.T.dot(d_ot_output)
    d_ht = d_ot_output.dot(W.T) + future_d_ht  # 别忘来了上一时刻传入的梯度
       d_ht_output = sigmoid_derivative(ht) * d_ht
    dU += xt.T.dot(d_ht_output)
    dV += ht_prev.T.dot(d_ht_output)

    # 更新future_d_ht
    future_d_ht = d_ht_output.dot(V.T)

# SGD更新参数
  U -= learing_rate * dU
V -= learing_rate * dV
W -= learing_rate * dW

# 重置梯度
  dU *= 0
dV *= 0
dW *= 0

# 输出loss和预测结果
  if (i % 1000 == 0):
    print("loss:" + str(loss))
    print("Pred:" + str(d))
    print("True:" + str(c))
```

```
        out = 0
        for index, x in enumerate(reversed(d)):
            out += x * pow(2, index)
        print(str(a_int) + " + " + str(b_int) + " = " + str(out))
        print("------------")
```

最后经过一定训练步长之后，得到的二进制加法器效果还是非常好的：

```
------------
loss:0.144199050347
Pred:[0 1 0 0 1 1 0 1]
True:[0 1 0 0 1 1 0 1]
3 + 74 = 77
------------
loss:0.0942641561631
Pred:[0 0 1 1 0 1 0 0]
True:[0 0 1 1 0 1 0 0]
20 + 32 = 52
------------
loss:0.0818225933734
Pred:[0 0 0 1 1 1 1 1]
True:[0 0 0 1 1 1 1 1]
5 + 26 = 31
------------
```

总结

本文简单介绍了RNN的原理以及应用场景，并给出了一个RNN的纯Python实例，后序大家可以学习更复杂的应用实例，也可以深入了解RNN的变种如LSTM等模型。

参考资料

1. Recurrent Neural Networks Tutorial, Part 1 – Introduction to RNNs: http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/.
2. Understanding LSTM Networks: http://colah.github.io/posts/2015-08-Understanding-LSTMs/.
3. Anyone Can Learn To Code an LSTM-RNN in Python (Part 1: RNN): http://iamtrask.github.io/2015/11/15/anyone-can-code-lstm/.