RNN的python实现

原创 小云数儿 云南高校数据化运营管理工程中心 2019-05-12

目录

编辑: 沉观
校对: 沉观
版本: python3

1.准备工作

我们写一个模块，定义了softmax函数，和Adam优化器来更新参数，到时我们好导入它，也可以使我们的代码更美观，代码如下：

```
import numpy as np

import sys
sys.path.append("D:/programmingsoftware/Anaconda3/Lib")
import rnn_utils # 导入我们写好的模块
def rnn_cell_forward(xt, a_prev, parameters):
    """
    实现RNN单元的单步前向传播
    参数：
        xt -- 时间步“t”输入的数据，维度为（n_x, m）
        a_prev -- 时间步“t - 1”的隐藏隐藏状态，维度为（n_a, m）
        parameters -- 字典，包含了以下内容：
                        Wax -- 矩阵，输入乘以权重，维度为（n_a, n_x）
                        Waa -- 矩阵，隐藏状态乘以权重，维度为（n_a, n_a）
                        Wya -- 矩阵，隐藏状态与输出相关的权重矩阵，维度为（n_y,
                        ba  -- 偏置，维度为（n_a, 1）
                        by  -- 偏置，隐藏状态与输出相关的偏置，维度为（n_y, 1）
    返回：
```

```
        a_next -- 下一个隐藏状态，维度为（n_a， m）
        yt_pred -- 在时间步“t”的预测，维度为（n_y， m）
        cache -- 反向传播需要的元组，包含了(a_next, a_prev, xt, parameters)
    """
    # 从“parameters”获取参数
    Wax = parameters["Wax"]
    Waa = parameters["Waa"]
    Wya = parameters["Wya"]
    ba = parameters["ba"]
    by = parameters["by"]

    # 使用上面的公式计算下一个激活值
    a_next = np.tanh(np.dot(Waa, a_prev) + np.dot(Wax, xt) + ba)

    # 使用上面的公式计算当前单元的输出
    yt_pred = rnn_utils.softmax(np.dot(Wya, a_next) + by)

    # 保存反向传播需要的值
    cache = (a_next, a_prev, xt, parameters)

    return a_next, yt_pred, cache

np.random.seed(1)
xt = np.random.randn(3,10)
a_prev = np.random.randn(5,10)
Waa = np.random.randn(5,5)
Wax = np.random.randn(5,3)
Wya = np.random.randn(2,5)
ba = np.random.randn(5,1)
by = np.random.randn(2,1)
parameters = {"Waa": Waa, "Wax": Wax, "Wya": Wya, "ba": ba, "by": by}

a_next, yt_pred, cache = rnn_cell_forward(xt, a_prev, parameters)
print("a_next[4] = ", a_next[4])
print("a_next.shape = ", a_next.shape)
print("yt_pred[1] =", yt_pred[1])
print("yt_pred.shape = ", yt_pred.shape)
```

结果如下：

```
a_next[4] =  [ 0.59584544  0.18141802  0.61311866  0.99808218  0.85016201  0
 -0.18887155  0.99815551  0.6531151   0.82872037]
a_next.shape =  (5, 10)
yt_pred[1] = [0.9888161  0.01682021 0.21140899 0.36817467 0.98988387 0.88945
 0.36920224 0.9966312  0.9982559  0.17746526]
yt_pred.shape =  (2, 10)
```

2.1RNN单步前向传播

```
def rnn_forward(x, a0, parameters):
    """
    实现循环神经网络的前向传播
    参数：
        x -- 输入的全部数据，维度为(n_x, m, T_x)
        a0 -- 初始化隐藏状态，维度为 (n_a, m)
        parameters -- 字典，包含了以下内容:
                        Wax -- 矩阵，输入乘以权重，维度为（n_a, n_x）
                        Waa -- 矩阵，隐藏状态乘以权重，维度为（n_a, n_a）
                        Wya -- 矩阵，隐藏状态与输出相关的权重矩阵，维度为（n_y,
                        ba  -- 偏置，维度为（n_a, 1）
                        by  -- 偏置，隐藏状态与输出相关的偏置，维度为（n_y, 1）
    返回：
        a -- 所有时间步的隐藏状态，维度为(n_a, m, T_x)
        y_pred -- 所有时间步的预测，维度为(n_y, m, T_x)
        caches -- 为反向传播的保存的元组，维度为（【列表类型】cache, x)）
    """
    # 初始化“caches”，它将以列表类型包含所有的cache
    caches = []

    # 获取 x 与 Wya 的维度信息
    n_x, m, T_x = x.shape
    n_y, n_a = parameters["Wya"].shape

    # 使用0来初始化“a” 与“y”
    a = np.zeros([n_a, m, T_x])
    y_pred = np.zeros([n_y, m, T_x])

    # 初始化“next”
    a_next = a0
```

```
    # 遍历所有时间步
    for t in range(T_x):
        ## 1.使用rnn_cell_forward函数来更新“next”隐藏状态与cache。
        a_next, yt_pred, cache = rnn_cell_forward(x[:, :, t], a_next, param

        ## 2.使用 a 来保存“next”隐藏状态（第 t ）个位置。
        a[:, :, t] = a_next

        ## 3.使用 y 来保存预测值。
        y_pred[:, :, t] = yt_pred

        ## 4.把cache保存到“caches”列表中。
        caches.append(cache)

    # 保存反向传播所需要的参数
    caches = (caches, x)

    return a, y_pred, caches

np.random.seed(1)
x = np.random.randn(3,10,4)
a0 = np.random.randn(5,10)
Waa = np.random.randn(5,5)
Wax = np.random.randn(5,3)
Wya = np.random.randn(2,5)
ba = np.random.randn(5,1)
by = np.random.randn(2,1)
parameters = {"Waa": Waa, "Wax": Wax, "Wya": Wya, "ba": ba, "by": by}

a, y_pred, caches = rnn_forward(x, a0, parameters)
print("a[4][1] = ", a[4][1])
print("a.shape = ", a.shape)
print("y_pred[1][3] =", y_pred[1][3])
print("y_pred.shape = ", y_pred.shape)
print("caches[1][1][3] =", caches[1][1][3])
print("len(caches) = ", len(caches))
```

结果如下：

```
a[4][1] =  [-0.99999375  0.77911235 -0.99861469 -0.99833267]
a.shape =  (5, 10, 4)
y_pred[1][3] = [0.79560373 0.86224861 0.11118257 0.81515947]
y_pred.shape =  (2, 10, 4)
caches[1][1][3] = [-1.1425182  -0.34934272 -0.20889423  0.58662319]
len(caches) =  2
```

2.2RNN前向传播

```
def rnn_forward(x, a0, parameters):
    """
    实现循环神经网络的前向传播
    参数：
        x -- 输入的全部数据，维度为(n_x, m, T_x)
        a0 -- 初始化隐藏状态，维度为 (n_a, m)
        parameters -- 字典，包含了以下内容:
                        Wax -- 矩阵，输入乘以权重，维度为（n_a, n_x）
                        Waa -- 矩阵，隐藏状态乘以权重，维度为（n_a, n_a）
                        Wya -- 矩阵，隐藏状态与输出相关的权重矩阵，维度为（n_y,
                        ba  -- 偏置，维度为（n_a, 1）
                        by  -- 偏置，隐藏状态与输出相关的偏置，维度为（n_y, 1）
    返回：
        a -- 所有时间步的隐藏状态，维度为(n_a, m, T_x)
        y_pred -- 所有时间步的预测，维度为(n_y, m, T_x)
        caches -- 为反向传播的保存的元组，维度为（【列表类型】cache, x)）
    """
    # 初始化“caches”，它将以列表类型包含所有的cache
    caches = []

    # 获取 x 与 Wya 的维度信息
    n_x, m, T_x = x.shape
    n_y, n_a = parameters["Wya"].shape

    # 使用0来初始化“a” 与“y”
    a = np.zeros([n_a, m, T_x])
    y_pred = np.zeros([n_y, m, T_x])

    # 初始化“next”
    a_next = a0
    # 遍历所有时间步
```

```
    for t in range(T_x):
        ## 1.使用rnn_cell_forward函数来更新“next”隐藏状态与cache。
        a_next, yt_pred, cache = rnn_cell_forward(x[:, :, t], a_next, param

        ## 2.使用 a 来保存“next”隐藏状态（第 t ）个位置。
        a[:, :, t] = a_next

        ## 3.使用 y 来保存预测值。
        y_pred[:, :, t] = yt_pred

        ## 4.把cache保存到“caches”列表中。
        caches.append(cache)

    # 保存反向传播所需要的参数
    caches = (caches, x)

    return a, y_pred, caches

np.random.seed(1)
x = np.random.randn(3,10,4)
a0 = np.random.randn(5,10)
Waa = np.random.randn(5,5)
Wax = np.random.randn(5,3)
Wya = np.random.randn(2,5)
ba = np.random.randn(5,1)
by = np.random.randn(2,1)
parameters = {"Waa": Waa, "Wax": Wax, "Wya": Wya, "ba": ba, "by": by}

a, y_pred, caches = rnn_forward(x, a0, parameters)
print("a[4][1] = ", a[4][1])
print("a.shape = ", a.shape)
print("y_pred[1][3] =", y_pred[1][3])
print("y_pred.shape = ", y_pred.shape)
print("caches[1][1][3] =", caches[1][1][3])
print("len(caches) = ", len(caches))
```

结果如下：

```
a[4][1] =  [-0.99999375  0.77911235 -0.99861469 -0.99833267]
```

```
a.shape =  (5, 10, 4)
y_pred[1][3] = [0.79560373 0.86224861 0.11118257 0.81515947]
y_pred.shape =  (2, 10, 4)
caches[1][1][3] = [-1.1425182  -0.34934272 -0.20889423  0.58662319]
len(caches) =  2
```

3.1RNN反向传播

```
def rnn_cell_backward(da_next, cache):
    """
    实现基本的RNN单元的单步反向传播
    参数：
        da_next -- 关于下一个隐藏状态的损失的梯度。
        cache -- 字典类型，rnn_step_forward()的输出
    返回：
        gradients -- 字典，包含了以下参数：
                        dx -- 输入数据的梯度，维度为(n_x, m)
                        da_prev -- 上一隐藏层的隐藏状态，维度为(n_a, m)
                        dWax -- 输入到隐藏状态的权重的梯度，维度为(n_a, n_x)
                        dWaa -- 隐藏状态到隐藏状态的权重的梯度，维度为(n_a, n_a)
                        dba -- 偏置向量的梯度，维度为(n_a, 1)
    """
    # 获取cache 的值
    a_next, a_prev, xt, parameters = cache
    # 从 parameters 中获取参数
    Wax = parameters["Wax"]
    Waa = parameters["Waa"]
    Wya = parameters["Wya"]
    ba = parameters["ba"]
    by = parameters["by"]

    # 计算tanh相对于a_next的梯度.
    dtanh = (1 - np.square(a_next)) * da_next
    # 计算关于Wax损失的梯度
    dxt = np.dot(Wax.T, dtanh)
    dWax = np.dot(dtanh, xt.T)

    # 计算关于Waa损失的梯度
    da_prev = np.dot(Waa.T, dtanh)
    dWaa = np.dot(dtanh, a_prev.T)
```

```
    # 计算关于b损失的梯度
    dba = np.sum(dtanh, keepdims=True, axis=-1)

    # 保存这些梯度到字典内
    gradients = {"dxt": dxt, "da_prev": da_prev, "dWax": dWax, "dWaa": dWaa

    return gradients


np.random.seed(1)
xt = np.random.randn(3,10)
a_prev = np.random.randn(5,10)
Wax = np.random.randn(5,3)
Waa = np.random.randn(5,5)
Wya = np.random.randn(2,5)
b = np.random.randn(5,1)
by = np.random.randn(2,1)
parameters = {"Wax": Wax, "Waa": Waa, "Wya": Wya, "ba": ba, "by": by}

a_next, yt, cache = rnn_cell_forward(xt, a_prev, parameters)

da_next = np.random.randn(5,10)
gradients = rnn_cell_backward(da_next, cache)
print("gradients[\"dxt\"][1][2] =", gradients["dxt"][1][2])
print("gradients[\"dxt\"].shape =", gradients["dxt"].shape)
print("gradients[\"da_prev\"][2][3] =", gradients["da_prev"][2][3])
print("gradients[\"da_prev\"].shape =", gradients["da_prev"].shape)
print("gradients[\"dWax\"][3][1] =", gradients["dWax"][3][1])
print("gradients[\"dWax\"].shape =", gradients["dWax"].shape)
print("gradients[\"dWaa\"][1][2] =", gradients["dWaa"][1][2])
print("gradients[\"dWaa\"].shape =", gradients["dWaa"].shape)
print("gradients[\"dba\"][4] =", gradients["dba"][4])
print("gradients[\"dba\"].shape =", gradients["dba"].shape)
```

结果如下：

```
gradients["dxt"][1][2] = -1.3872130506020925
gradients["dxt"].shape = (3, 10)
```

```
gradients["da_prev"][2][3] = -0.15239949377395473
gradients["da_prev"].shape = (5, 10)
gradients["dWax"][3][1] = 0.41077282493545836
gradients["dWax"].shape = (5, 3)
gradients["dWaa"][1][2] = 1.1503450668497135
gradients["dWaa"].shape = (5, 5)
gradients["dba"][4] = [0.20023491]
gradients["dba"].shape = (5, 1)
```

3.2RNN反向传播

```
def rnn_backward(da, caches):
    """
    在整个输入数据序列上实现RNN的反向传播
    参数：
        da -- 所有隐藏状态的梯度，维度为(n_a, m, T_x)
        caches -- 包含向前传播的信息的元组
    返回：
        gradients -- 包含了梯度的字典：
                        dx -- 关于输入数据的梯度，维度为(n_x, m, T_x)
                        da0 -- 关于初始化隐藏状态的梯度，维度为(n_a, m)
                        dWax -- 关于输入权重的梯度，维度为(n_a, n_x)
                        dWaa -- 关于隐藏状态的权值的梯度，维度为(n_a, n_a)
                        dba -- 关于偏置的梯度，维度为(n_a, 1)
    """
    # 从caches中获取第一个cache（t=1）的值
    caches, x = caches
    a1, a0, x1, parameters = caches[0]

    # 获取da与x1的维度信息
    n_a, m, T_x = da.shape
    n_x, m = x1.shape

    # 初始化梯度
    dx = np.zeros([n_x, m, T_x])
    dWax = np.zeros([n_a, n_x])
    dWaa = np.zeros([n_a, n_a])
    dba = np.zeros([n_a, 1])
    da0 = np.zeros([n_a, m])
    da_prevt = np.zeros([n_a, m])
```

```
    # 处理所有时间步
    for t in reversed(range(T_x)):
        # 计算时间步“t”时的梯度
        gradients = rnn_cell_backward(da[:, :, t] + da_prevt, caches[t])

        # 从梯度中获取导数
        dxt, da_prevt, dWaxt, dWaat, dbat = gradients["dxt"], gradients["da
            "dWaa"], gradients["dba"]

        # 通过在时间步t添加它们的导数来增加关于全局导数的参数
        dx[:, :, t] = dxt
        dWax += dWaxt
        dWaa += dWaat
        dba += dbat

    # 将 da0设置为a的梯度，该梯度已通过所有时间步骤进行反向传播
    da0 = da_prevt

    # 保存这些梯度到字典内
    gradients = {"dx": dx, "da0": da0, "dWax": dWax, "dWaa": dWaa, "dba": d

    return gradients

np.random.seed(1)
x = np.random.randn(3,10,4)
a0 = np.random.randn(5,10)
Wax = np.random.randn(5,3)
Waa = np.random.randn(5,5)
Wya = np.random.randn(2,5)
ba = np.random.randn(5,1)
by = np.random.randn(2,1)
parameters = {"Wax": Wax, "Waa": Waa, "Wya": Wya, "ba": ba, "by": by}
a, y, caches = rnn_forward(x, a0, parameters)
da = np.random.randn(5, 10, 4)
gradients = rnn_backward(da, caches)

print("gradients[\"dx\"][1][2] =", gradients["dx"][1][2])
print("gradients[\"dx\"].shape =", gradients["dx"].shape)
print("gradients[\"da0\"][2][3] =", gradients["da0"][2][3])
```

```
print("gradients[\"da0\"].shape =", gradients["da0"].shape)
print("gradients[\"dWax\"][3][1] =", gradients["dWax"][3][1])
print("gradients[\"dWax\"].shape =", gradients["dWax"].shape)
print("gradients[\"dWaa\"][1][2] =", gradients["dWaa"][1][2])
print("gradients[\"dWaa\"].shape =", gradients["dWaa"].shape)
print("gradients[\"dba\"][4] =", gradients["dba"][4])
print("gradients[\"dba\"].shape =", gradients["dba"].shape)
```

结果如下:

```
gradients["dx"][1][2] = [-2.07101689 -0.59255627  0.02466855  0.01483317]
gradients["dx"].shape = (3, 10, 4)
gradients["da0"][2][3] = -0.31494237512664996
gradients["da0"].shape = (5, 10)
gradients["dWax"][3][1] = 11.264104496527777
gradients["dWax"].shape = (5, 3)
gradients["dWaa"][1][2] = 2.303333126579893
gradients["dWaa"].shape = (5, 5)
gradients["dba"][4] = [-0.74747722]
gradients["dba"].shape = (5, 1)
```

思考——学而不思则罔

掌握了RNN的基础知识和代码，现在可以去尝试下使用RNN来搭建些好玩的应用，如模仿莎士比亚的风格等等，快去试试吧!

理解编程语言，探索数据奥秘
每日练习|干货分享|新闻资讯|公益平台。
每天学习一点点，你将会见到全新的自己。

长按识别二维码关注