Theano /

scan - Looping in Theano

介绍

- recurrence的通用形式，用来looping
- reduction和map(在leading dimenstions上loop)是scan的特例
- 沿着一些输入序列scan一个函数，产生每个time-step的输出
- 这个函数可以看过你的函数的previous K time-steps
- sum()可以通过在a list上扫描z+x(i)函数来计算，给定初始状态z=0
- 通过a for loop可以表示为scan()操作，scan是最接近Theano looping的
- 使用scan来for loops的优点
  - 迭代的数目 作为 symbolic graph的一部分
  - 沿着sequential steps来计算gradients

定义

- def scan(
- fn, #描述scan中的一步操作。fn能够构建出（描述一个迭代步骤的输出）的变量。它期望输入（表示输入序列的所有slices和以前的输出值，以及其他作为'non_sequences'的参数）的theano变量。scan传递变量到fn的次序是：sequences，output_info, non_sequences. sequences和output_info的时间篇可通过taps设定。
- sequences=None, #list of Theano variables or dictionaries描述scan迭代的序列。
- outputs_info=None, #list of Theano variables or dictionaries描述递归计算的输出的初始状态。
- non_sequences=None, #在每一步传递到fn的参数列表
- n_steps=None, #int或theano scalar，迭代步数
- truncate_gradient=-1, #在迭代中使用truncated BPTT，或classical BPTT
- go_backwards=False, #
- mode=None,
- name=None,
- profile=False):
- :return: #tuple of the form (outputs, updates）。``outputs``是表示scan输出的Theano variable或是Theano variables列表(与``outputs_info``次序相同).

计算A^k

```
result = 1
for i in xrange(k):
  result = result * A
```

- 其中需要处理三件事：初始值赋给result，结果的累加在result，和不变量A。
- 不变量作为non_sequences传递给scan。初始化是在outputs_info,累加是自动发生的。

代码为

```
k = T.iscalar("k")
A = T.vector("A")

# Symbolic description of the result
result, updates = theano.scan(fn=lambda prior_result, A: prior_result * A,
                              outputs_info=T.ones_like(A),
                              non_sequences=A,
                              n_steps=k)

# We only care about A**k, but scan has provided us with A**1 through A**k.
# Discard the values that we don't care about. Scan is smart enough to
# notice this and not waste memory saving them.
final_result = result[-1]

# compiled function that returns A**k
power = theano.function(inputs=[A,k], outputs=final_result, updates=updates)

print power(range(10),2)
print power(range(10),4)
```

- 首先构建一个function(使用lambda expression)，提供prior_result和A，返回prior_result*A。参数的次序是通过scan确定：fn的prior call的输出（或初始值）是第一个参数，后面是所有的non_sequences。
- 下一步初始化输出为与A相同的shape和dtype的tensor，用1填充（T.ones_like(A)）. 我们把A作为non_sequence参数送到scan，设置步骤数k在lambda表达式上进行迭代。
- scan返回一个元组，包括结果result和更新的词典(本例为empty)。注意：结果不是一个matrix,而是一个3D tensor包括每一步的值A**k。我们想要最后的值(k步后）,所有我们编译一个function来返回它。注意，这有一个优化，在编译时间会检测你只使用result的最后一个值，确保scan不存储所有使用过的中间值。所有不用担心A和k的值大的问题。

```
theano.config.warn.subtensor_merge_bug = False

def inner_fct(prior_result, B):
    return prior_result * B

result, updates = theano.scan(fn=inner_fct,
                              outputs_info=T.ones_like(A),
                              non_sequences=A, n_steps=k)
```

沿着tensor的第一维迭代：计算多项式polynomial

- 除了looping一定数目，scan可以沿着tensor的首维进行迭代（类似python的for x in a_list)
- 提供给scan的需要looping的tensor(s) 使用sequence参数

下面是从coefficients列表中计算多项式

```
coefficients = theano.tensor.vector("coefficients")
x = T.scalar("x")

max_coefficients_supported = 10000

# Generate the components of the polynomial
components, updates = theano.scan(fn=lambda coefficient, power, free_variable: coefficient * (free_variable ** power),
                                  outputs_info=None,
                                  sequences=[coefficients, theano.tensor.arange(max_coefficients_supported)],
                                  non_sequences=x)
# Sum them up
polynomial = components.sum()

# Compile a function
calculate_polynomial = theano.function(inputs=[coefficients, x], outputs=polynomial)

# Test
test_coefficients = numpy.asarray([1, 0, 2], dtype=numpy.float32)
test_value = 3
print calculate_polynomial(test_coefficients, test_value)
print 1.0 * (3 ** 0) + 0.0 * (3 ** 1) + 2.0 * (3 ** 2)
```

需要注意：

- 通过首先产生每个coefficients，然后在最后相加来得到多项式（也可以沿着来累计，然后取最后最后一个值，这样内存更有效，这只是个例子）。
- 第二，没有结果的累加，我们可以设置outputs_info为None。这表明scan不需要传递先验值（prior result）到fn。函数参数的通用次序到fn为：sequences (if any), prior result(s) (if needed), non-sequences (if any)
- 第三，这儿有一个trick来模拟python的enumerate:简单的包含到theano.tensor.arange到sequences。
- 第四，给定不同长度的多个序列，scan将截断到最短的。这样传递一个长的arange是安全的，这是通常我们需要做的，因为arange必须在创建时必须配置长度。

简单叠加到标量（scalar)，ditching lambda

- 虽然这个几乎可以自解释，它有一个缺点需要注意：初始输出状态是output_info提供的，必须是每次迭代产生的输出变量相似的形状，并且它必须不包含后者的implicit downcast

```
import numpy as np
import theano
import theano.tensor as T

up_to = T.iscalar("up_to")

# define a named function, rather than using lambda
def accumulate_by_adding(arange_val, sum_to_date):
    return sum_to_date + arange_val
seq = T.arange(up_to)

# An unauthorized implicit downcast from the dtype of 'seq', to that of
# 'T.as_tensor_variable(0)' which is of dtype 'int8' by default would occur
# if this instruction were to be used instead of the next one:
# outputs_info = T.as_tensor_variable(0)

outputs_info = T.as_tensor_variable(np.asarray(0, seq.dtype))
scan_result, scan_updates = theano.scan(fn=accumulate_by_adding,
                                        outputs_info=outputs_info,
                                        sequences=seq)
triangular_sequence = theano.function(inputs=[up_to], outputs=scan_result)

# test
some_num = 15
print triangular_sequence(some_num)
print [n * (n + 1) // 2 for n in xrange(some_num)]
```

References

- Loop http://deeplearning.net/software/theano/tutorial/loop.html
- http://deeplearning.net/software/theano/library/scan.html
- source: theano/scan_module/scan.py

Page last modified on December 17, 2013, at 06:33 AM