1.Sparse

희소행렬 (sparse matrix): 많은 원소들의 값이 0 인 행렬.
A=

$$\begin{bmatrix} 15 & 0 & 0 & 22 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix}$$

행렬의 원소들을 모두 기억장치에 저장하면 기억공간이 많이 필요하다. 특히 희소행렬의 경우 0 인 원소들의 개수가 0 이 아닌 원소들의 개수보다 많을 때, 0 인 원소들을 모두 기억시킨다면 기억장치의 낭비가 심하게 된다.
그렇기 때문에, 희소행렬 중 0 이 아닌 원소들만 새로운 행렬로서 기억시킴으로써 기억공간의 낭비를 방지한다

예를들어 다음과 같은 2차원 배열이 있다고 할 때

		column					
		0	1	2	3	4	5
row	0	15	0	0	22	0	-15
	1	0	11	3	0	0	0
	2	0	0	0	-6	0	0
	3	0	0	0	0	0	0
	4	91	0	0	0	0	0
	5	0	0	28	0	0	0

불필요하게 0을 저장하는 공간이 많아지기 때문에

	row	col	value
a [0]	6	6	8
[1]	0	0	15
[2]	0	3	22
[3]	0	5	-15
[4]	1	1	11
[5]	1	2	3
[6]	2	3	-6
[7]	4	0	91
[8]	5	2	28

다음과 같이 표현하는 것이다.

이는 구조체를 이용해서 배열을 만들어 주면 간단하게 1차원 배열로도 값을 표현해 낼 수 있다.

[0]번 인덱스에는 원래의 2차원 배열이 몇행 몇열로 이루어 져있고 값이 몇개인지가 저장되고,

[1]번 인덱스 부터 row값을 기준으로 0이 아닌 값들의 위치와 값이 저장된다.

Theano의 sparse패키지는 효율적인 알고리즘을 제공하지만 모든 매트릭스 케이스에 추천되어지는 건 아니다. sparsity proportion(행렬에서 엘리먼트 0의 개수/ 행렬의 모든 엘리먼트 개수) 가 낮은 경우엔 고려가 필요, 왜냐하면 실제 데이터도 저장되면서 동시에 행렬의 모든 요소들의 위치도 저장해야 하기 때문이다.
이것은 오히려 일반 행렬을 최적화 알고리즘을 이용해 돌리는 것보다 계산시간이 더 걸릴 수도 있다.

Theano의 sparse 라이브러리는 Scipy의 Sparse 라이브러리에 기반을 두고 만들어 졌기 때문에,관련 자세한 문서는 Scipy Sparce를 참조하면 된다.

희소행렬은 인접배열로 저장되지 않기 때문에, 메모리에 희소행렬을 올리는데 몇가지 방법이 있다.
이것은 소위 행렬의 "format"이라고 불리우는 것에 의해 디자인 된다. (희소행렬의 자료구조)

format은 csc,csr 두가지가 있고, 이는 거의 유사하나, 열(column)을 기준으로 행(row)기준으로 하느냐에 따른 차이가 있다.

위 함수의 속성값은 data, indices, indptr, shape 이며, data는 1차원의 ndarray로 sparse matrix의 non-zero 요소들을 포함한다.
indices 와 indptr 은 sparse matrix에서의 데이터의 위치를 저장하기 위해 사용한다.
shape은 행렬의 형태(nxm)을 의미한다.

그렇다면 csr,csc 포맷 이 둘 중 어떤걸 써야 하는가?

→ If shape[0] > shape[1], use csr format. Otherwise, use csc.

(행렬의 행(row)가 열(column)보다 클 경우엔 csr 포맷, 반대의 경우엔 csc 포맷선택)

1)Properties and Construction

```
#########################
#1)Properties and Construction
#########################

import theano
import numpy as np
import scipy.sparse as sp
from theano import sparse

a= sparse.all_dtypes

print a

## csc --> csr로 변환 (전치)
x = sparse.csc_matrix(name='x', dtype='int64')
data, indices, indptr, shape = sparse.csm_properties(x) # sparse 변수들은 직접 엑세스 할수 없기 때문에 csm_property함수를 사용해 접근한다.
y = sparse.CSR(data, indices, indptr, shape) # sparse 행렬을 재구성 하기위해 쓰이는 함수는 csc/csr 이다 (CSR로 전치)
f = theano.function([x], y)
a = sp.csc_matrix(np.asarray([[0, 1, 1], [0, 0, 0], [1, 0, 0]]))
print '1) a.toarray:'
print a.toarray()
print '2) f(a).toarray():'
print f(a).toarray()  #전치 된 결과
```

```
[[0 1 1]
 [0 0 0]
 [1 0 0]]
>>> print f(a).toarray()
[[0 0 1]
 [1 0 0]
 [1 0 0]]
```

2) Structured Operation

```
##########################
#2) Structured Operation
##########################
# 0을 제외한 나머지 엘리먼트에 똑같이 어떤 값의 연산을 처리할 때 사용한다

x = sparse.csc_matrix(name='x', dtype='float32')
y = sparse.structured_add(x, 2)
f = theano.function([x], y)
a = sp.csc_matrix(np.asarray([[0, 1, -1], [0, -2, 1], [3, 0, 0]], dtype='float32')) #[0, 0, -1]
print '[Structured Operation](1) a.toarray:'
print a.toarray()
print '[Structured Operation](2) f(a).toarray():'
print f(a).toarray()
print '###### shape test #####'
print a.shape[0] # row 의 크기
print a.shape[1] # column 크기

print a.data
print a.indices
print a.indptr
```

```
[[ 0.  0. -1.]
 [ 0. -2.  1.]
 [ 3.  0.  0.]]
>>> print f(a).toarray()
[[ 0.  0.  1.]
 [ 0.  0.  3.]
 [ 5.  0.  0.]]
```

2.Using the GPU

Theano는 CPU와 GPU를 사용한 연산을 유연하게 오갈 수 있다.

GPU 사용을 위해서는 Nvidia's GPU-programming toolchain (CUDA)설치가 필요

GPGPU(General Purpose GPU)

고정된 형태의 GPU를 사용하던 방식에서 프로그램 가능한 보다 유연한 형태의 GPU로 넘어오면서 3D그래픽 분야를 위해서 사용되던 GPU에 대한 새로운 접근이 시도되었다. 이는 GPU가 행렬과 벡터 연산에 높은계산 성능을 보이는것에서 기인한것으로 그래픽 연산뿐 아니라 일반 Computing 영역에서 GPU를 황용하고자 하는 체제를 말한다.

수천개의 코어로 이루어진 그래픽카드를 다양하게 사용하기위해 사용하는 GPGPU는 병렬처리에 적합한 연산에서 뛰어난 속도를 낼 수 있다는 잇점이 있어서 좋지만,GPU병렬 프로세싱을 위한 프로그래밍 작성을 해야한다. 이에 GPGPU 프로그래밍 작성을 위해 다양한 방법들이 나오고 있다.

CUDA:

NVIDIA에서 사용하는 GPU개발을 위한 툴이다. 이는 병렬 컴퓨팅 플랫폼으로 대량의 GPU코어로 컴퓨팅 속도를 크게 향상 시킬 수 있는 프로그래밍 모델이다. 이는 C Language기반으로 공유 메모리를 사용하여 빠른 연산이 가능하다.그러나 NVIDIA제조사에 특화되어 G8X GPU로 구성된 Geforce 9 시리즈 급 이상만 지원하는 한계가 있다.

OpenCL :

크로노스 그룹에서 유지 및 관리되고 있는 OpenCL 은 애플, AMD, IBM, 인텔, 그리고 NVIDIA 에서 개발 한 개방형 범용 병렬 컴퓨팅 프레임워크로, OpenGL 과 완벽하게 연동 되는 OpenCL 은 모바일 임베디드 분야에서도 사용이 가능하지만, 제조사인 NVIDIA 와 인텔의 지원 부족으로 인해 이기종 플랫 폼 구현이 제한된다는 한계점이 있다. 또한, C99 스탠 다드 헤더를 사용할 수 없다는 제한이 있다

****Device= gpu 변경: 홈폴더에있는 .theanorc 파일을 수정한다.(추천)**

```
[global]
floatX = float32
device = gpu

[nvcc]
fastmath = True
```

1) Testing Theano with GPU

```
#1)Testing Theano with GPU
from theano import function, config, shared, sandbox
import theano.tensor as T
import numpy
import time

vlen = 10 * 30 * 768  # 10 x #cores x # threads per core
iters = 1000

rng = numpy.random.RandomState(22) #랜덤 넘버 생성 ( 22는 일종의 번호로 randomstate 변수를 생성할 때 번호 지정해야
                                   #여러개의 randomstate 변수를 사용할 때 모두 다른  random한 값을 출력)
#rjk = numpy.random.RandomState(22)
x = shared(numpy.asarray(rng.rand(vlen), config.floatX)) # 랜덤값이 입력값 ,데이터 타입은 floatX(디폴트로 'float32'가)
                                                         # Shared 함수는  input x를 GPU 메모리로 올라가게 한다.
f = function([], T.exp(x))
print(f.maker.fgraph.toposort())
t0 = time.time()
for i in xrange(iters):
    r = f()
t1 = time.time()
print("Looping %d times took %f seconds" % (iters, t1 - t0))   # exp()연산 처리에 걸린 시간
print("Result is %s" % (r,))
if numpy.any([isinstance(x.op, T.Elemwise) for x in f.maker.fgraph.toposort()]):
    print('Used the cpu')
else:
    print('Used the gpu')
```

```
$ THEANO_FLAGS=mode=FAST_RUN,device=cpu,floatX=float32 python check1.py
[Elemwise{exp,no_inplace}(<TensorType(float32, vector)>)]
Looping 1000 times took 3.06635117531 seconds
Result is [ 1.23178029  1.61879337  1.52278066 ...,  2.20771813  2.29967761
  1.62323284]
Used the cpu

$ THEANO_FLAGS=mode=FAST_RUN,device=gpu,floatX=float32 python check1.py
Using gpu device 0: GeForce GTX 580
[GpuElemwise{exp,no_inplace}(<CudaNdarrayType(float32, vector)>), HostFromGpu(GpuElemwise{exp,no_inplace}.0)]
Looping 1000 times took 0.638810873032 seconds
Result is [ 1.23178029  1.61879349  1.52278066 ...,  2.20771813  2.29967761
  1.62323296]
Used the gpu
```

2)Returning a Handle to Device-Allocated Data

```
#2) Returning a Handle to Device-Allocated Data(error)

from theano import function, config, shared, sandbox
import theano.sandbox.cuda.basic_ops
import theano.tensor as T
import numpy
import time
vlen = 10 * 30 * 768  # 10 x #cores x # threads per core
iters = 1000

rng = numpy.random.RandomState(22)
x = shared(numpy.asarray(rng.rand(vlen), 'float32'))
f = function([], sandbox.cuda.basic_ops.gpu_from_host(T.exp(x)))
#gpu_from_host?
# GPU를 이용한 연산속도를 향상시키기 위해,  gpu_from_host 연산을 사용할수 있는데,
#이것은 인풋을 host 에서 GPU로 복사시키는 연산이다.
# 이 연산을 사용함으로써 , T.exp(x) 는 GPU 버전의 exp() 대체된다.

print(f.maker.fgraph.toposort())
t0 = time.time()
for i in xrange(iters):
    r = f()
t1 = time.time()
print("Looping %d times took %f seconds" % (iters, t1 - t0))
print("Result is %s" % (r,))
print("Numpy result is %s" % (numpy.asarray(r),))
if numpy.any([isinstance(x.op, T.Elemwise) for x in f.maker.fgraph.toposort()]):
    print('Used the cpu')
else:
    print('Used the gpu')
```

```
$ THEANO_FLAGS=mode=FAST_RUN,device=gpu,floatX=float32 python check2.py
Using gpu device 0: GeForce GTX 580
[GpuElemwise{exp,no_inplace}(<CudaNdarrayType(float32, vector)>)]
Looping 1000 times took 0.34898686409 seconds
Result is <CudaNdarray object at 0x6a7a5f0>
Numpy result is [ 1.23178029  1.61879349  1.52278066 ...,  2.20771813  2.29967761
  1.62323296]
Used the gpu
```

첫번째 예제의 GPU연산 속도에 비해 50% 속도 개선이 되었다.(결과 배열을 host로 복사시키는 것을 안 함으로써)
→ 이제 각 함수에 의해 리턴되는 결과가 Numpy array가 아닌 CudaNdarray로 변환

3)Running the GPU at Full Speed

```
f = function([],
        Out(sandbox.cuda.basic_ops.gpu_from_host(T.exp(x)),
            borrow=True))
```

맥시멈 속도를 내기 위해서, out인스턴스를 borrow=true플래그와 함께 사용하는게 필요하다.

이는 Theano가 output을 우리에게 리턴하기 위해 복사하지 않는걸 의미한다.

이는 Theano가 pre-allocate memory(내부에서 사용을 하기위해 미리 할당된 메모리-사용중인 버퍼처럼)

하기 때문인데, 이것은 각 함수호출시 결과와 관련된 버퍼를 새롭게 할당된 메모리로 복사시킨다.

연속적인 함수 호출에도 이전에 계산된 결과가 overwrite되지 않게 한다.

What Can Be Accelerated on the GPU

The performance characteristics will change as we continue to optimize our implementations, and vary from device to device, but to give a rough idea of what to expect right now:

- Only computations with *float32* data-type can be accelerated. Better support for *float64* is expected in upcoming hardware but *float64* computations are still relatively slow (Jan 2010).
- Matrix multiplication, convolution, and large element-wise operations can be accelerated a lot (5-50x) when arguments are large enough to keep 30 processors busy.
- Indexing, dimension-shuffling and constant-time reshaping will be equally fast on GPU as on CPU.
- Summation over rows/columns of tensors can be a little slower on the GPU than on the CPU.
- Copying of large quantities of data to and from a device is relatively slow, and often cancels most of the advantage of one or two accelerated functions on that data. Getting GPU performance largely hinges on making data transfer to the device pay off.

Tips for Improving Performance on GPU

- Consider adding `floatX=float32` to your `.theanorc` file if you plan to do a lot of GPU work.
- Use the Theano flag `allow_gc=False`. See *GPU Async capabilities*
- Prefer constructors like `matrix`, `vector` and `scalar` to `dmatrix`, `dvector` and `dscalar` because the former will give you *float32* variables when `floatX=float32`.
- Ensure that your output variables have a *float32* dtype and not *float64*. The more *float32* variables are in your graph, the more work the GPU can do for you.
- Minimize tranfers to the GPU device by using `shared` *float32* variables to store frequently-accessed data (see `shared()`). When using the GPU, *float32* tensor `shared` variables are stored on the GPU by default to eliminate transfer time for GPU ops using those variables.
- If you aren't happy with the performance you see, try building your functions with `mode='ProfileMode'`. This should print some timing information at program termination. Is time being used sensibly? If an op or Apply is taking more time than its share, then if you know something about GPU programming, have a look at how it's implemented in theano.sandbox.cuda. Check the line similar to *Spent Xs(X%) in cpu op, Xs(X%) in gpu op and Xs(X%) in transfer op*. This can tell you if not enough of your graph is on the GPU or if there is too much memory transfer.
- Use nvcc options. nvcc supports those options to speed up some computations: *-ftz=true* to flush denormals values to zeros., *-prec-div=false* and *-prec-sqrt=false* options to speed up division and square root operation by being less precise. You can enable all of them with the *nvcc.flags=-use_fast_math* Theano flag or you can enable them individually as in this example: *nvcc.flags=-ftz=true -prec-div=false*.

4) Changing the Value of Shared Variables

**theano.shared(value=W_values, name='W', borrow=True)

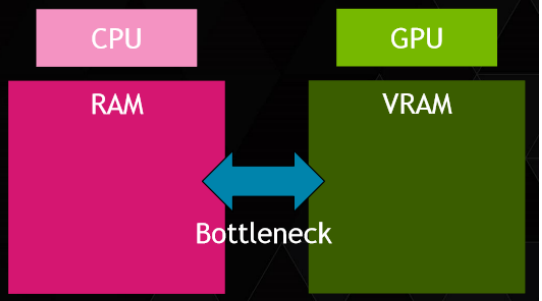
: 기본적으로 GPU 연산은 데이터를 주메모리에서 GPU용 메모리인 VRAM으로 옮긴 후 처리된다. 그리고 결과를 확인하려면 다시 VRAM에서 주메모리로 가져와야한다. 이 부분이 시간 많이 소요된다. 따라서 메모리간 이동을 최소화 하기 위해서 GPU에 데이터를 올리는 함수이다

*Shared variable 은 일종의 글로벌 변수처럼 생각해도 된다. (외부 함수에서 이 변수를 건드리려면 get_value 와 set_value 를 사용)

GPU 연산 관련 문법: SHARED

- 기능: VRAM과 RAM 사이의 데이터 전송

- shared_var = theano.shared(numpy_array)
- numpy_array = shared_var.get_value()



GPU 연산 관련 문법: GIVENS

- 기능: Symbolic 변수에 Shared 데이터를 대입

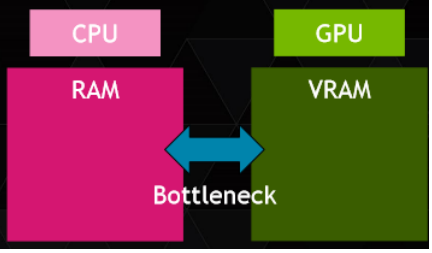
[예제] y = 2*x 일때, x에 10을 대입 계산하는 두 가지 구현 방법

- 방법1)
  - compute = theano.function([x], 2*x);
  - compute(10)

← 실행시 RAM→VRAM→GPU연산

- 방법2)
  - x_value = theano.shared(10)
  - compute = theano.function([], 2*x, givens=(x, x_value))
  - compute()

← 실행시 VRAM→GPU연산



GPU 연산 관련 문법: UPDATES

- 기능: GPU연산 결과를 이용해 Shared 데이터를 수정

- x_val = theano.shared(0)
- increase = theano.function([], x_val, updates=(x_val, x_val+1))
- Increase()

← 실행시 RAM을 거치지 않고, GPU내에서 계속 x_val을 1씩 증가시킴