theano

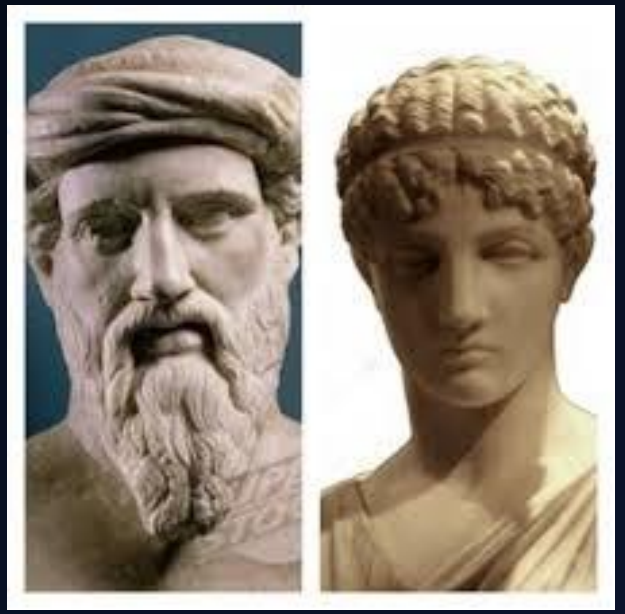
MARIA NAKLICKA 127313
OLIWIA MASIAN 127324

Krótka historia

- 2007- initial release
- 28 września 2017- ogłoszenie Montreal Institute for Learning Algorithms (MILA) o wstrzymaniu prac po wypuszczeniu wersji 1.0
- Theano 1.0.0 (15 listopada, 2017)
- 1.0.4 (15 stycznia 2019) - maintenace release

Theano

- kompilator matematycznych wyrażeń symbolicznych
- Biblioteka Pythonowa
- Wsparcie dla CPU i GPU
- Współpraca z NumPy
- Licencja BSD
- Nazwa na cześć greckiej matematyczki (mogła być żoną Pitagorasa)



Biblioteki wykorzystujące Theano:

- Keras
- Blocks
- Lasagne
- PyMC 3
- sklearn-theano
- Platoon

Code sample

```
1   import theano
2   from theano import tensor
3
4   # declare two symbolic floating-point scalars
5   a = tensor.dscalar()
6   b = tensor.dscalar()
7
8   # create a simple expression
9   c = a + b
10
11  # convert the expression into a callable object that takes (a,b)
12  # values as input and computes a value for c
13  f = theano.function([a,b], c)
14
15  # bind 1.5 to 'a', 2.5 to 'b', and evaluate 'c'
16  assert 4.0 == f(1.5, 2.5)
```

Wyrażenia symboliczne

Theano – język, kompilator, biblioteka

1. Zdefiniuj wyrażenie symboliczne.
2. Skompiluj funkcję, która oblicza wartości *(theano.function)* .
3. Wywołaj funkcję dla konkretnych wartości.

Definicja zmiennych wejściowych

- Symbolicznie, silnie typowane, nie muszą mieć zdefiniowanych wymiarów
- tensor as T – popularna konwencja

```
1   import theano
2   from theano import tensor as T
3   x = T.vector('x')
4   y = T.vector('y')
```

Zmienne współdzielone

- pomocne np. przy tworzeniu funkcji z wewnętrznym stanem, który zmienia się przy każdorazowym wywołaniu

```
1   import numpy as np
2   np.random.seed(42)
3   W_val = np.random.randn(4, 3)
4   b_val = np.ones(3)
5   W = theano.shared(W_val)
6   b = theano.shared(b_val)
7   W.name = 'W'
8   b.name = 'b'
```

Wyrażenia symboliczne

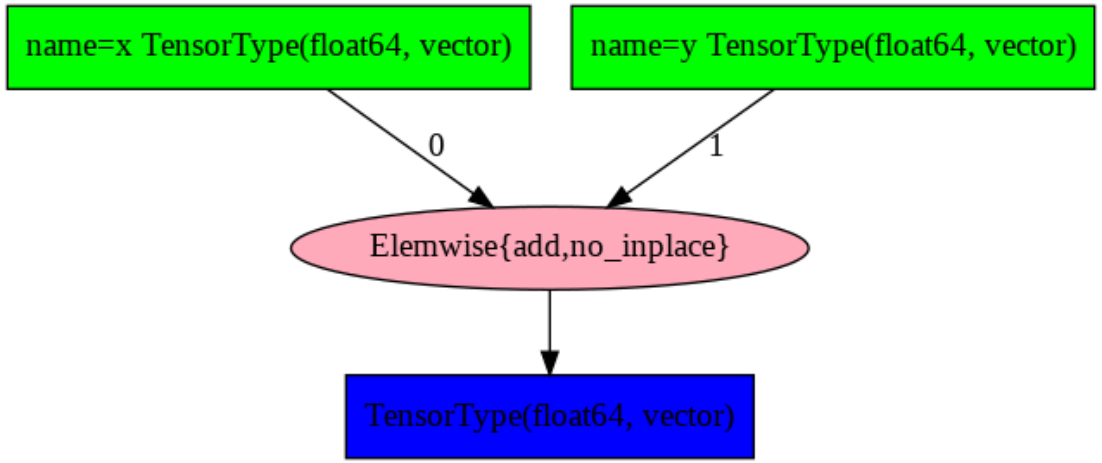
```
[5]  import theano
     from theano import tensor as T
     x = T.vector('x')
     y = T.vector('y')
```

```
[6]  c= x+y
     print(c)
```

```
Elemwise{add,no_inplace}.0
```

```
[7]  from theano.printing import pydotprint
     from IPython.display import Image, SVG
```

```
[8]  Image(pydotprint(c, format='png', compact=False, return_image=True))
```



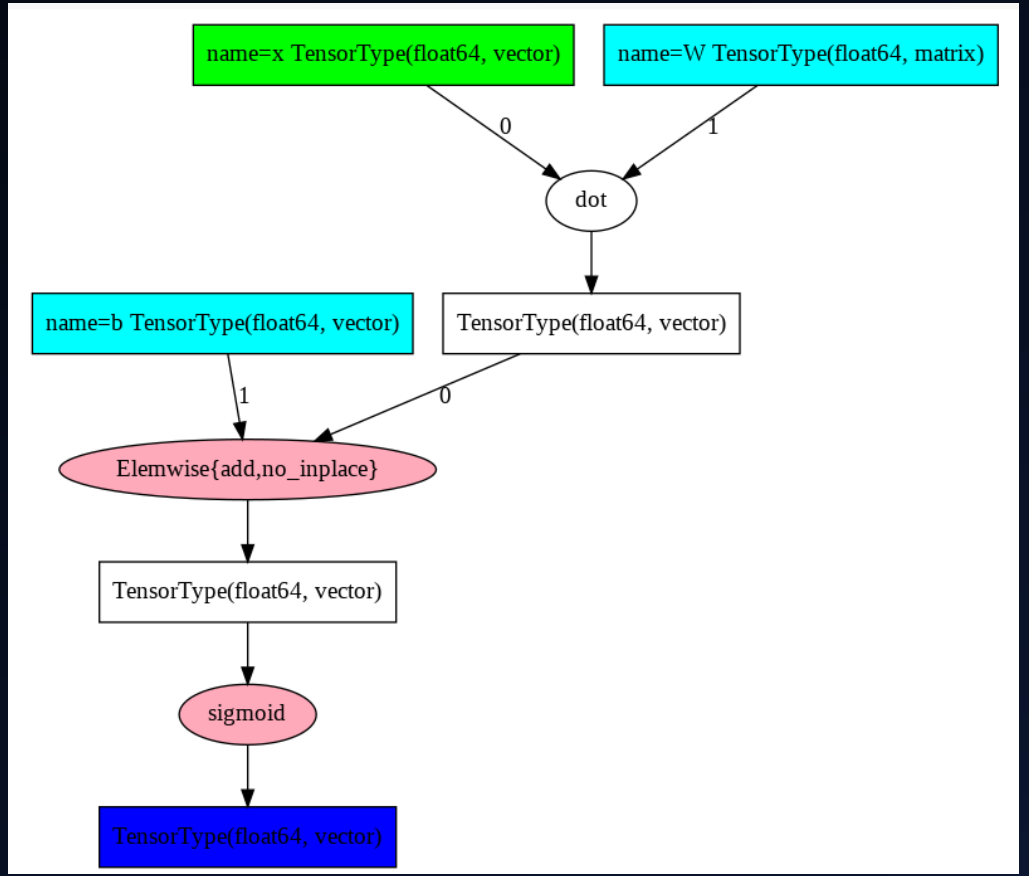
Wyrażenia symboliczne

```
dot = T.dot(x, W)
out = T.nnet.sigmoid(dot + b)

C = ((out - y) ** 2).sum()
C.name = 'C'
```

```
from theano.printing import pydotprint
from IPython.display import Image, SVG
```

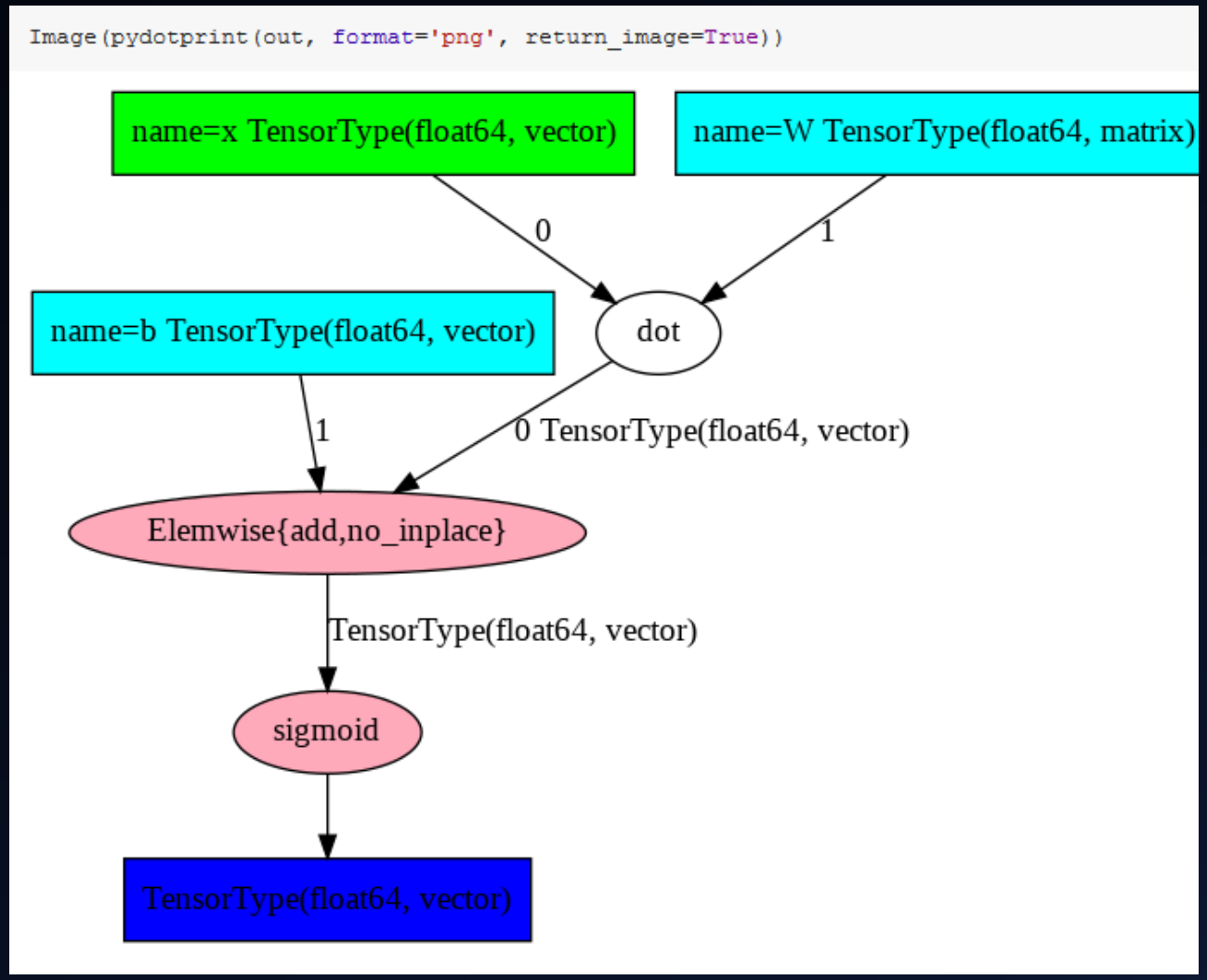
```
Image(pydotprint(out, format='png', compact=False, return_image=True))
```



Graf obliczeń

- kompaktowy

```
Image(pydotprint(out, format='png', return_image=True))
```



Jeszcze raz o zmiennych współdzielonych

- Updates- para (shared-variable, new expression)

```
from theano import shared, function
state = shared(0)
inc = T.iscalar('inc')
accumulator = function([inc], state, updates=[(state, state+inc)])
```

```
print(state.get_value())
```

```
0
```

```
accumulator(5)
```

```
array(0)
```

```
print(state.get_value())
```

```
5
```

```
accumulator(9)
```

```
array(5)
```

```
print(state.get_value())
```

```
14
```

Obliczanie gradientu

- Theano jest w stanie samodzielnie tworzyć symboliczne grafy obliczeń gradientów
- theano.grad – aplikuje regułę łańcuchową
- Niech C(x) = f(g(x))
- dC_dW = theano.grad(C, W)
- dC_db = theano.grad(C, b)
- albo dC_dW, dC_db = theano.grad(C, [W, b])

Obliczanie gradientu

- dC_dW i dC_db są wyrażeniami symbolicznymi, nie przyjmują konkretnych wartości na tym etapie
- Są częścią tego samego grafu obliczeń
- Mogą być użyte do sformułowania nowego wyrażenia - np. aktualizacja wag

```
upd_W = W - 0.1 * dC_dW
upd_b = b - 0.1 * dC_db
```

Kompilacja funkcji

- Kompilowanie funkcji
- Optymalizacja grafu
- Wizualizacja grafu

Kompilacja funkcji

- Pierwszym argumentem są wartości wejściowe
- Drugi argument to wyrażenie, które chcemy obliczyć

```
predict = theano.function([x], out)
x_val = np.random.rand(4)
print(predict(x_val))
# -> array([ 0.9421594 ,  0.73722395,  0.67606977])

monitor = theano.function([x, y], [out, C])
y_val = np.random.uniform(size=3)
print(monitor(x_val, y_val))
# -> [array([ 0.9421594 ,  0.73722395,  0.67606977]),
#     array(0.6191236997823024)]

error = theano.function([out, y], C)
print(error([0.942, 0.737, 0.676], y_val))
# -> array(0.6002210054210885)
```

Aktualizacja zmiennych współdzielonych

- Funkcje mogą służyć do obliczania nowych wartości zmiennych współdzielonych i aktualizowania ich

```
train = theano.function([x, y], C,
                        updates=[(W, upd_W),
                                 (b, upd_b)])
```

- Wszystkie outputy są obliczane **przed** aktualizacją

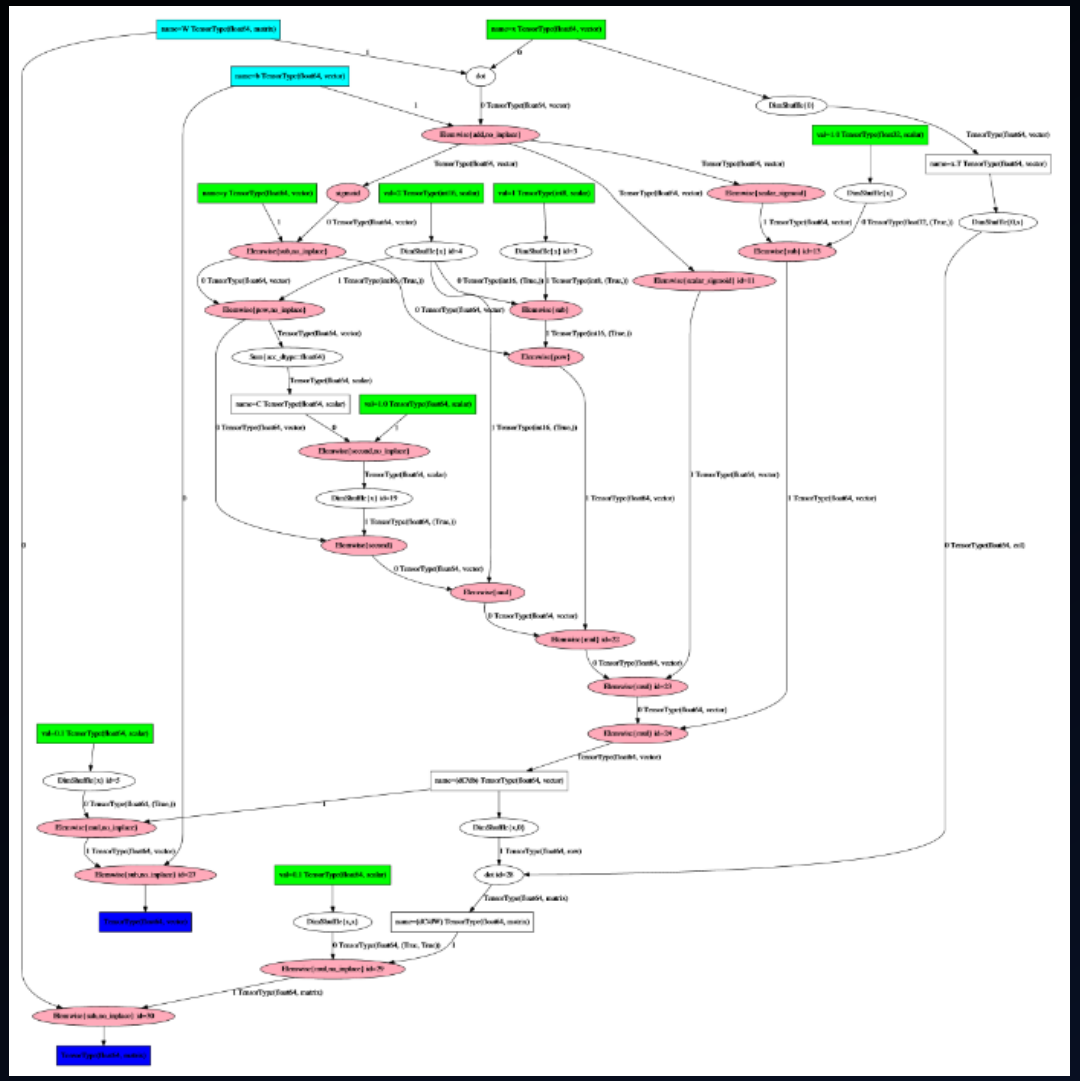
```
print(b.get_value())
# -> [ 1.  1.  1.]
train(x_val, y_val)
print(b.get_value())
# -> [ 0.9967082   0.99340064  0.97245715]
```

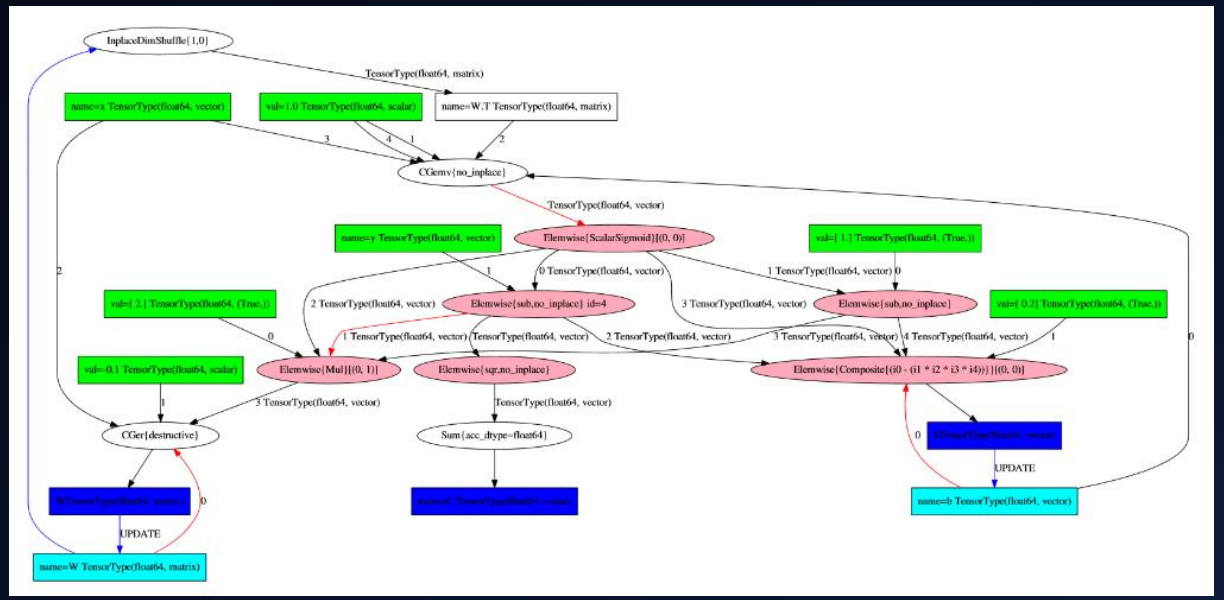
Optymalizacja grafu

- Optymalizacja polega na zastąpieniu części grafu innymi wierzchołkami
  - typy wierzchołków muszą się zgadzać
  - wartości muszą być identyczne
- Cele optymalizacji:
  - połączenie równoważnych obliczeń
  - uproszczenie wyrażeń (np. x/x = 1)
  - stabilność numeryczna
  - użycie efektywnych konstrukcji (np. operacje w miejscu)
  - transfer do obliczeń na GPU

Włączanie/wyłączanie optymalizacji

- Przetarg między szybkością kompilacji, czasem wykonania i detekcją błędów. Istnieje kilka predefiniowanych trybów:
  - **mode = 'FAST_RUN'** - tryb domyślny. Możliwie najszybszy runtime. Pozwala na wykonanie obliczeń na GPU.
  - **mode = 'FAST_COMPILE'** - minimalizacja kosztów uruchomienia, prędkość zbliżona do NumPy
  - **optimizer= 'fast_compile'** - umożliwia użycie GPU, ale ogranicza optymalizację grafu
  - **mode = 'DEBUG_MODE'** - sprawdza wszystko, bardzo wolny
  - Można włączać i wyłączać poszczególne optymalizacje
  - Można to robić globalnie lub dla każdej funkcji z osobna




InplaceDimShuffle{1,0}
TensorType(float64, matrix)
name=x TensorType(float64, vector)
val=1.0 TensorType(float64, scalar)
name=W.T TensorType(float64, matrix)
CGemv{no_inplace}
TensorType(float64, vector)
name=y TensorType(float64, vector)
Elemwise{ScalarSigmoid}{(0, 0)}
val=[1.] TensorType(float64, (True,))
val=[2.] TensorType(float64, (True,))
Elemwise{sub,no_inplace} id=4
Elemwise{sub,no_inplace}
val=[0.2] TensorType(float64, (True,))
val=-0.1 TensorType(float64, scalar)
Elemwise{Mul}{(0, 1)}
Elemwise{sqr,no_inplace}
Elemwise{Composite{(i0 - (i1 * i2 * i3 * i4))}}{(0, 0)}
CGer{destructive}
Sum{acc_dtype=float64}
UPDATE
UPDATE
name=W TensorType(float64, matrix)
name=b TensorType(float64, vector)

debugprint

```
debugprint(out)

sigmoid [id A] ''
 |Elemwise{add,no_inplace} [id B] ''
   |dot [id C] ''
   | |x [id D]
   | |W [id E]
   |b [id F]
```

```
debugprint(predict)

Elemwise{ScalarSigmoid}[(0, 0)] [id A] ''   2
 |CGemv{no_inplace} [id B] ''   1
   |b [id C]
   |TensorConstant{1.0} [id D]
   |InplaceDimShuffle{1,0} [id E] 'W.T'   0
   | |W [id F]
   |x [id G]
   |TensorConstant{1.0} [id D]
```

Optymalizacja wykonania kodu

- Można napisać kod w C++/CUDA, który zwraca wartości wyjściowe
- Możliwe jest dynamiczne generowanie kodu
- Kod jest kompilowany do modułu Pythonowego, zapamiętywany w cache i importowany do programu

Wykorzystanie GPU

- Zamysłem autorów Theano było transparentne użycie GPU.
- Backend GPU został dodany w wersji 0.9
  - więcej typów danych (nie tylko float32 – double, int)
  - łatwiejsza interakcja z GPU arrays z poziomu Pythona
- Żeby umożliwić wykorzystanie GPU należy ustawić **device flag** na 'cuda'
  - wszystkie zmienne współdzielone zostana utworzone w pamięci GPU
  - możliwa optymalizacja poprzez wykonanie niektórych operacji na GPU
  - jeśli zależy nam na szybkości - konieczne użycie float32

Flagi konfiguracyjne

- Możliwe sposoby ustawienia flag:
  - w pliku konfiguracyjnym .theanorc

    [global]

    device = cuda0

    floatX = float32
  - w konsoli:

    THEANO_FLAGS=device=cuda0,floatX=float32
  - z poziomu Pythona:

    theano.config.floatX = 'float32'

    theano.config.device nie może być zmienione po imporcie Theano

Pętle – operacja scan

- Ogólna forma rekurencji
- reduce, map, reduce and accumulate
- Sekwencja na wejściu, funkcja obliczana w każdym kroku czasowym
- Funkcja jest w stanie widzieć efekt wywołania poprzednich k kroków
- Gradient obliczany w kolejnych krokach -> wsteczna propagacja w czasie (rozwijanie w kolejnych krokach czasowych)

scan

Zalety wykorzystania operacji scan:

- Liczba iteracji jest elementem grafu obliczeń
- Minimalizacja transferów GPU
- Szybsza niż używanie pętli Pythonowej pętli for, wewnątrz której następuje wywołanie skompilowanej funkcji Theano
- Możliwość oszczędności pamięci (w przypadku wykrycia mniejszej ilości potrzebnej pamięci)

Scan – przykład 1

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

# symboliczny opis wyniku
result, updates = theano.scan(fn=lambda prior_result, A: prior_result * A,
                              outputs_info=T.ones_like(A),
                              non_sequences=A,
                              n_steps=k)

# Interesuje nas tylko wynik końcowy A**k, jednak funkcja scan zapewnia
# wynik w każdym kroku czasowym (od A**1 do A**k)
# Dla oszczędności pamięci zaznaczymy, że na wyjściu chcemy mieć tylko wynik z ostatniego kroku.

final_result = result[-1]

# skompilowana funkcja, która zwraca A**k
power = theano.function(inputs=[A, k], outputs=final_result, updates=updates)

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

```
[ 0.  1.  4.  9. 16. 25. 36. 49. 64. 81.]
```

Scan – przykład 2

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

# defining the tensor variables
X = T.matrix("X")
W = T.matrix("W")
b_sym = T.vector("b_sym")

results, updates = theano.scan(lambda v: T.tanh(T.dot(v, W) + b_sym), sequences=X)
compute_elementwise = theano.function(inputs=[X, W, b_sym], outputs=results)

# test values
x = np.eye(2, dtype=theano.config.floatX)
w = np.ones((2, 2), dtype=theano.config.floatX)
b = np.ones((2), dtype=theano.config.floatX)
b[1] = 2

print(compute_elementwise(x, w, b))

# comparison with numpy
print(np.tanh(x.dot(w) + b))
```

```
[[0.96402758 0.99505475]
 [0.96402758 0.99505475]]
[[0.96402758 0.99505475]
 [0.96402758 0.99505475]]
```

Wizualizacja, debugging, narzędzia diagnostyczne

Definicja funkcji jest czymś oddzielnym niż jej *wykonanie*. Może to stanowić problem w rozumieniu i inspekcji programu. Theano dostarcza narzędzi, które mają pomóc w radzeniu sobie z tą rozbieżnością poprzez:

- Informacje o błędach
- Informacje podczas wykonywania kodu programu
- Monitorowanie NaN lub przepełnienia zmiennych
- Detekcje powszechnych źródeł spowolnienia
- Testy podczas budowania grafu obliczeń

Rozszerzanie Theano

- Tworzenie operacji z poziomu Pythona:
  - łatwe używanie Pythona w połączeniu z wyspecjalizowanymi bibliotekami (np. PyCuda)
  - możliwe zwiększenie kosztów dla runtime'u
  - przykład: Konwolucja 3D z użyciem szybkiej transformaty Fouriera na GPU
- Tworzenie operacji za pomocą kodu w C i CUDA:
  - używanie C-API dla Pythona lub NumPy
  - brak kosztów wywołania funkcji
  - kod w C inline lub w osobnym pliku
- Dodatkowa optymalizacja:
  - przeprowadzenie dodatkowych optymalizacji grafu
  - zastąpienie części grafu nowo utworzona operacją

Źródła

- http://deeplearning.net/software/theano/index.html
- https://github.com/lamblin/bayareadlschool
- https://www.youtube.com/watch?v=OU8I1oJ9HhI&fbclid=IwAR0gCphclaVCaUiPudb_FiRS8uf3ZFyaxExCEGjoaWq-Y5hnFiDseRdwGF0

Dziękujemy za uwagę!