THE HELMHOLTZ ANALYTICS TOOLKIT (HEAT) - A SCIENTIFIC BIG DATA LIBRARY FOR HPC -


EXTREME DATA WORKSHOP 2018
HEAT

- HeAT = Helmholtz Analytics Toolkit
- Early alpha phase (project start: May 2018)
- Data Analytics framework for transparent distributed computation
- Build on PyTorch
- Developed in the open: https://github.com/helmholtz-analytics and https://pypi.org/project/heat
- Liberally licensed: MIT
## HELMHOLTZ ANALYTICS FRAMEWORK (HAF)

### HAF – Goals and Objectives

<table>
<thead>
<tr>
<th>Scientific big data analytics</th>
<th>- methodologies and tools for problems of highest data and compute complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Helmholtz Analytics Framework</strong></td>
<td>- foster data science in Helmholtz</td>
</tr>
<tr>
<td></td>
<td>- develop and exploit the Helmholtz Data Federation (HDF)</td>
</tr>
<tr>
<td><strong>Use case driven co-design between</strong></td>
<td>- domain scientists</td>
</tr>
<tr>
<td></td>
<td>- data experts</td>
</tr>
<tr>
<td></td>
<td>- infrastructure professionals</td>
</tr>
<tr>
<td><strong>Create data analytics techniques</strong></td>
<td>- in a systematic manner</td>
</tr>
<tr>
<td></td>
<td>- domain-specific as well as generalizable and standardized</td>
</tr>
</tbody>
</table>
HAF USE CASES

Earth System Modelling

Research with Photons

Neuroscience

Aeronautics and Aerodynamics

Structural Biology

Mitglied der Helmholtz-Gemeinschaft

00. Monat 2017

Seite 4
HAF USE CASE METHODS

Clustering           kmeans (UC4, UC5, UC6, UC7, UC8), mean shift clustering (UC5)
Uncertainty quantification Ensemble methods (UC1, UC2, UC3, UC4)
Dimension Reduction autoencoder (UC5, UC7), reduced order models (UC7)
Feature learning     image descriptors (UC5), autoencoder (UC5, UC7)
Data Assimilation    Kalman filter (UC1), 4Dvar(UC2), particle filter/smooth(UC1, UC2)
Classification/Regression Random forest, CNN (UC5), SVM (UC2, UC3, UC5), Lasso/Ridge regression (UC5)
Modelling            fiber tractography (UC5), point processes (UC6)
Optimization techniques L-BFGS (UC5), simulated annealing (UC5)
Hyperparameter optimization differential evolution (UC5), grid search (UC5)
Interpolation        Radial basis function (UC7), Kriging (UC7)
Data Mining          Frequent item set mining (UC5)
BIG DATA/DL LIBRARIES

Big Data
- Hadoop
- DISCO
- Mahout
- Spark
- Apache Storm
- DASK

Deep Learning
- PyTorch
- TensorFlow
- dy/net
- PaddlePaddle
- CNTK
- ArrayFire
- H2O.ai
- BigDL
- mxnet
- Chainer
- Keras
**MAP-REDUCE ALGORITHM**

\[
\text{MAP} : K \times V \rightarrow (L \times W)^*\\
\text{REDUCE} : L \times W^* \rightarrow X^*
\]
Computations are defined as directed acyclic graphs (DAGs)

Graph is defined in high-level language (Python, C++, Go)

Graph is compiled (e.g. CNTK) or built at run-time (e.g. PyTorch)

Graph is executed on CPU/GPU

Graph can be computed distributed but required manually assignments
<table>
<thead>
<tr>
<th></th>
<th>GPU</th>
<th>Distributed</th>
<th>MPI</th>
<th>AD</th>
<th>ML</th>
<th>Linear Algebra</th>
<th>ND-Tensors</th>
<th>Dynamic Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>BigDL</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Spark</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Hadoop</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Disco</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>H2O</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Mahout</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Intel Daal</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Dask</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>TensorFlow</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>PyTorch</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Arrayfire</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>MXNet</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>PaddlePaddle</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>CNTK</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>DyNet</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Chainer</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
RUNTIME COMPARISON

Batch size of 32 x 224 x 224

ResNet architecture neural network

Run on single NVIDIA K80 GPU@JURECA

Similar results for other ML methods (e.g. k-means)

Choose Pytorch as a base technology
NumPy

Runs on:

Data structure

ND-Tensor

Operations

- Elementwise operations
- Slicing
- Matrix operations
- Reduction

shape: (4, 3, 2)
Numpy Array

N dimensional homogeneous collection of scalars of the same data type
**ELEMWISE OPERATIONS**

### Multiply by a scalar

```python
>>> a = array((1,2))
>>> a*3.
array([3., 6.])
```

### Element by element addition

```python
>>> a = array([1,2])
>>> b = array([3,4])
>>> a + b
array([4, 6])
```

### Operator function

```python
>>> add(a,b)
array([4, 6])
```

### In-place Operator function

```python
# Overwrite contents of a.
# Saves array creation
# overhead.
>>> add(a,b,a) # a += b
array([4, 6])
>>> a
array([4, 6])
```
COMPARISON AND LOGICAL OPERATIONS

<table>
<thead>
<tr>
<th>equal</th>
<th>(==)</th>
</tr>
</thead>
<tbody>
<tr>
<td>greater_equal</td>
<td>(&gt;=)</td>
</tr>
<tr>
<td>logical_and</td>
<td></td>
</tr>
<tr>
<td>logical_or</td>
<td></td>
</tr>
<tr>
<td>logical_not</td>
<td></td>
</tr>
<tr>
<td>not_equal</td>
<td>(!=)</td>
</tr>
<tr>
<td>greater</td>
<td>(&gt;)</td>
</tr>
<tr>
<td>less</td>
<td>(&lt;)</td>
</tr>
<tr>
<td>less_equal</td>
<td>(&lt;=)</td>
</tr>
<tr>
<td>logical_xor</td>
<td></td>
</tr>
</tbody>
</table>

2-D Example

```python
>>> a = array(((1,2,3,4),(2,3,4,5)))
>>> b = array(((1,2,5,4),(1,3,4,5)))
>>> a == b
array([[True, True, False, True],
       [False, True, True, True]])

# functional equivalent
>>> equal(a,b)
array([[True, True, False, True],
       [False, True, True, True]])
```
ARRAY SLICING

Slicing works much like standard Python slicing

```python
>>> a[0:3:5]
array([3, 4])

>>> a[4:, 4:]
array([[44, 45],
       [54, 55]])

>>> a[:, 2]
array([[2, 12, 22, 32, 42, 52]])
```

Strides are also possible

```python
>>> a[2::2, ::2]
array([[20, 22, 24],
       [40, 42, 44]])
```
FANCY INDEXING

```python
>>> a[(0,1,2,3,4), (1,2,3,4,5)]
array([ 1, 12, 23, 34, 45])

>>> a[3:, [0, 2, 5]]
array([[30, 32, 35],
       [40, 42, 45],
       [50, 52, 55]])

>>> mask = array([1,0,1,0,0,1],
                 dtype=bool)

>>> a[mask, 2]
array([2,22,52])
```
ARRAY BROACASTING
### REDUCTION METHODS

#### Sum Function

```python
>>> a = array([[1,2,3],
             [4,5,6]], float)

# sum() defaults to adding up all the values in an array
>>> sum(a)
21.

# supply the keyword axis to sum along the 0th axis
>>> sum(a, axis=0)
daarray([5., 7., 9.])

# supply the keyword axis to sum along the last axis
>>> sum(a, axis=-1)
daarray([6, 15.])
```

#### Sum Array Method

```python
# a.sum() defaults to adding up all values in an array
>>> a.sum()
21.

# supply an axis argument to sum along a specific axis
>>> a.sum(axis=0)
daarray([5., 7., 9.])

# product along columns
>>> a.prod(axis=0)
daarray([4, 10, 18])

# functional form
>>> prod(a, axis=0)
daarray([4, 10, 18])
```

#### Product
Matrix Operations

Matrix Multiplication

```python
>>> a = [[1, 0],
      [0, 1]]
>>> b = [[4, 1],
      [2, 2]]
# Matrix product of a and b
>>> np.matmul(a, b)
array([[4, 1],
      [2, 2]])
```

Eigenvalue decomposition

```python
>>> w, v = np.linalg.eig(np.diag((1, 2, 3)))
>>> w; v
array([ 1.,  2.,  3.])
array([[ 1.,  0.,  0.],
      [ 0.,  1.,  0.],
      [ 0.,  0.,  1.]])
```

Determinant

```python
>>> a = np.array([[1, 2],
                [3, 4]])
# compute determinant
>>> np.linalg.det(a)
-2.0
```

Matrix inversion

```python
a = np.array([[1., 2.],
              [3., 4.]])
>>> ainv = np.linalg.inv(np.matrix(a))
>>> ainv
matrix([[[-2. ,  1. ],
          [ 1.5, -0.5]]])
```
What is PyTorch?

Python based scientific computing package targeted at two sets of audiences:

- A replacement for NumPy to use the power of GPUs
- A deep learning research platform that provides maximum flexibility and speed

Automated differentiation (DA)
**Runs on:**

- CPU
- GPU

**Data structure:**

ND-Tensor

**Operations:**

- Elementwise operations
- Slicing
- Matrix operations
- Reduction
- Automatic differentiation
# Let us run this cell only if CUDA is available
# We will use `torch.device` objects to move tensors in and out of GPU

```python
if torch.cuda.is_available():
    device = torch.device("cuda")  # a CUDA device object
    y = torch.ones_like(x, device=device)  # directly create a tensor on GPU
    x = x.to(device)  # or just use strings `.to("cuda")`
    z = x + y
    print(z)
    print(z.to("cpu", torch.double))  # `.to` can also change dtype together!
```
PYTORCH: AUTOMATIC DIFFERENTIATION

How can we do the same as above with PyTorch's autograd package?

First, it should be obvious that we have to represent our original function in Python as such:

\[ y = 5x^4 + 3x^3 + 7x^2 + 9x - 5 \]

```python
import torch

x = torch.autograd.Variable(torch.Tensor([2]), requires_grad=True)
y = 5*x**4 + 3*x**3 + 7*x**2 + 9*x - 5

y.backward()

x.grad
```
Runs on:

Data structure

Operations

- Elementwise operations
- Slicing
- Matrix operations
- Reduction
- Automatic differentiation
Example:

```python
import heat as ht
# construct a range tensor
>>> range_data = ht.arange(6, split=1)

Server#1 PyTorch Tensor#1
Server#2 PyTorch Tensor#2
Server#3 PyTorch Tensor#3
```

```
>>> range_data.mean()
2.5
>>> range_data.argmax()
5
```
EXAMPLE: CLUSTERING
EXAMPLE: K-MEANS

- Core idea: *k clusters* around centroids

- Iterative minimization
  - \( \arg \min_C \sum_{i=1}^k \sum_{x \in C_i} \| x - \hat{x} \|^2 \)
  - Other metrics possible

- Algorithm sketch
  - Choose *k* centroids
  - For each points calculate distance to centroids
  - Assign point to closest centroid
  - Estimate new centroids as mean of points
  - Repeat until convergence

KK: This has to be updated!!!!!
EXAMPLE: K-MEANS

Numpy vs. HeAT

```python
- distances = ((data - centroids) ** 2).sum(axis=1, keepdims=True)
- matching_centroids = np.expand_dims(distances.argmin(axis=2, axis=2)
+ distances = ((data - centroids) ** 2).sum(axis=1)
+ matching_centroids = distances.argmin(axis=2)
```
EXAMPLE: K-MEANS

Numpy vs. HeAT

```python
- distances = ((data - centroids) ** 2).sum(axis=1, keepdims=True)
- matching_centroids = np.expand_dims(distances.argmin(axis=2), axis=2)
+ distances = ((data - centroids) ** 2).sum(axis=1)
+ matching_centroids = distances.argmin(axis=2)
```
EXAMPLE: K-MEANS

Numpy vs. HeAT

```python
- selection = (matching_centroids == i).astype(np.int64)
- new_centroids[:, :, i:i+1] = ((data * selection).sum(axis=0, keepdims=True) / 
+ selection.sum(axis=0).clip(1.0, sys.maxsize))
```
NUMPY VS. HEAT

```python
>>> -1.9 +1.9
import sys

-1.8

import numpy as np

+import heat as ht

-"-class KMeansHT:
   -class KMeans:
       def __init__(self, n_clusters, max_iter=1000, tol=1e-4, random_state=42):
           # TODO: document me
           # TODO: extend the parameter list
           # -17.12 +17.12
           # TODO: document me
           # TODO: extend me with further initialization methods
           # zero-centered uniform random distribution in [-1, 1]
           np.random.seed(seed)

           return np.random.uniform(low=-1.0, high=1.0, size=(1, dimensions, k))

           +ht.random.set_gseed(seed)

           +ht.random.uniform(low=-1.0, high=1.0, size=(1, dimensions, k))

       def fit(self, data):
           # TODO: document me

           data = np.expand_dims(data, axis=2)

           data = data.expand_dims(axis=2)

           # initialize the centroids randomly
           centroids = self.initialize_centroids(self.n_clusters, data.shape[1], self.random_state)

           # for epoch in range(self.max_iter):
           #     calculate the distance matrix and determine the closest centroid

           distances = ((data - centroids) ** 2).sum(axis=1, keepdims=True)

           matching_centroids = np.expand_dims(distances.argmin(axis=1), axis=2)

           +distances = ((data - centroids) ** 2).sum(axis=1)

           +matching_centroids = distances.argmin(axis=2)

           # update the centroids

           for i in range(self.n_clusters):

               selection = (matching_centroids == i).astype(np.int64)

               new_centroids[:, i, i+1] = ((data * selection).sum(axis=0, keepdims=True) /

               +selection = (matching_centroids == i).astype(ht.int64)

               +new_centroids[:, i, i+1] = ((data * selection).sum(axis=0) /

               selection.sum(axis=0).clip(1.0, sys.maxsize))
```

Important from a users perspective:

- Aiming for numpy-compatibility
- Ideally, you just need to replace np by ht
SUMMARY

BLABLA