UNIVERSITY OF CAMBRIDGE

Accelerating Function Minimisation with PyTorch

Bojan Nikolic

ADASS 2018, College Park, MD

arXiv:1805.07439

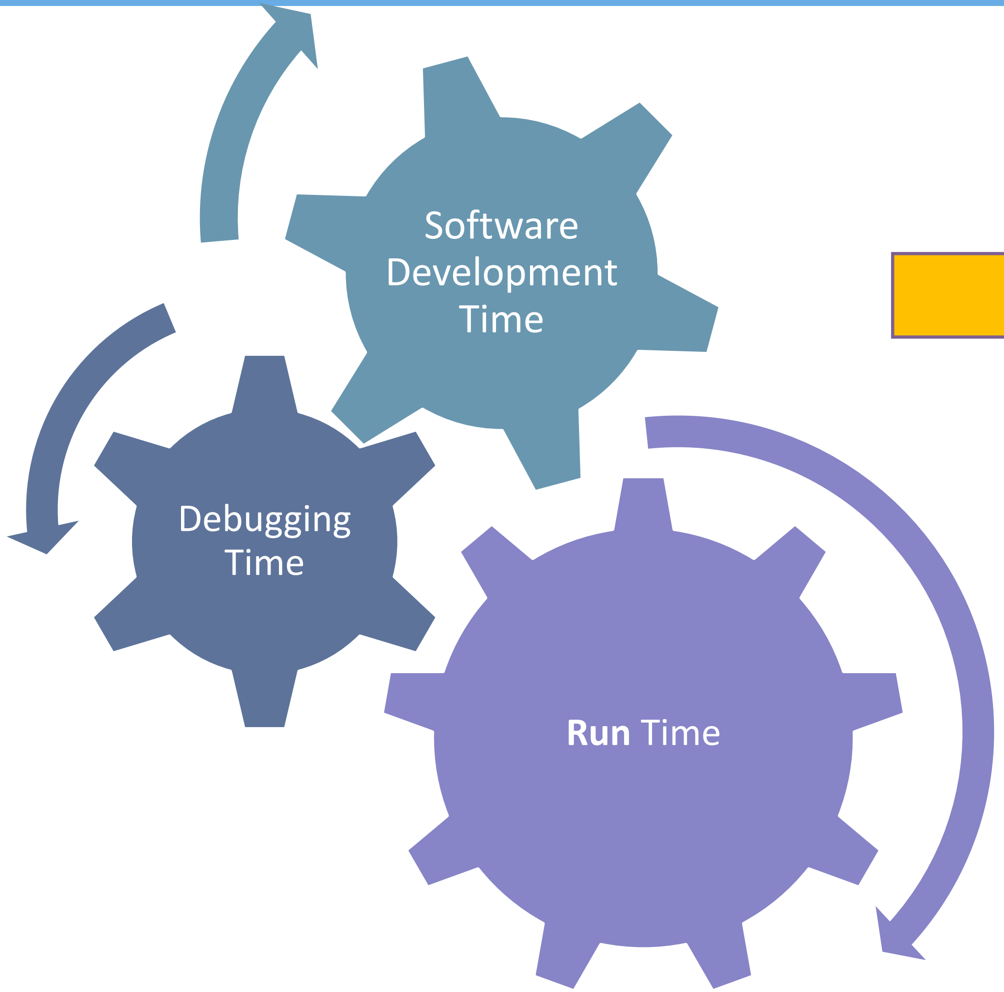
Acknowledgement

H2020-Astronomy ESFRI and Research Infrastructure Cluster (Grant Agreement number: 653477).

H2020-Astronomy ESFRI and Research Infrastructure Cluster (Grant Agreement number: 653477).

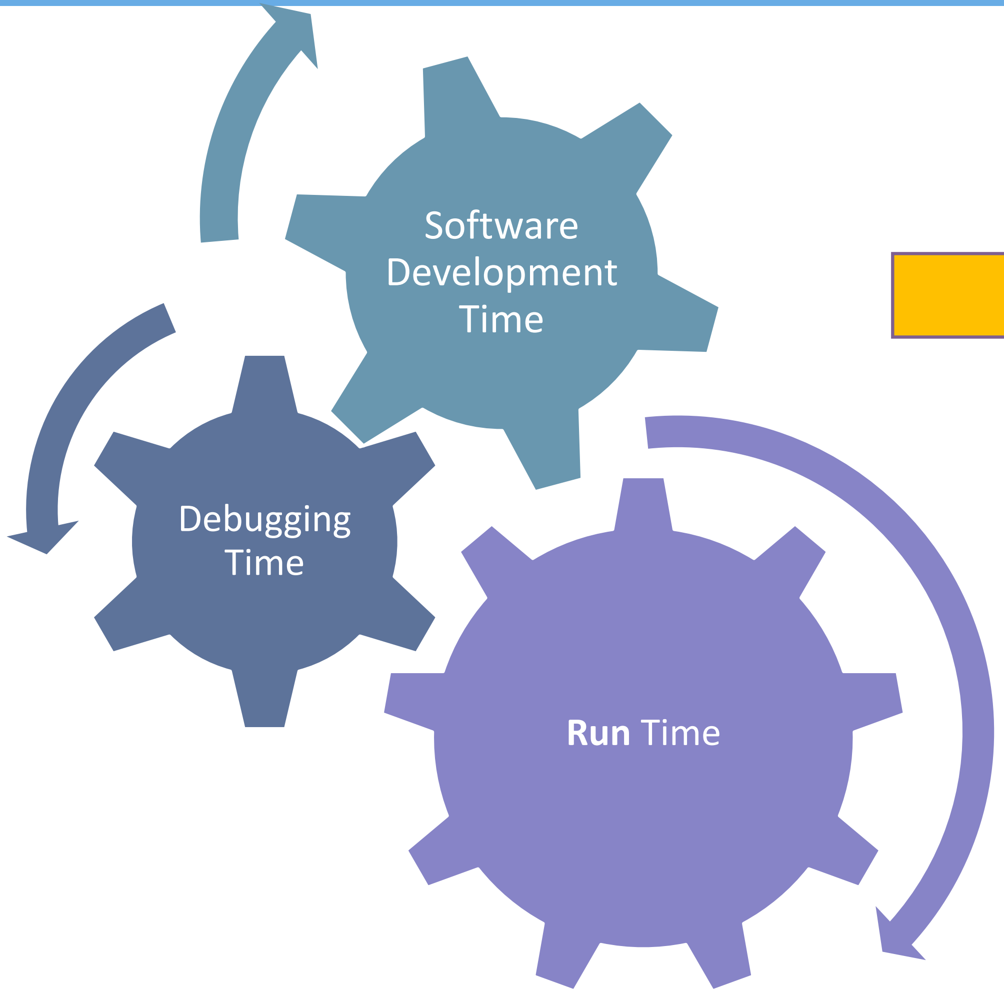
Introduction

Objective: accelerate a common data analysis step



"Faster" to good science

Objective: accelerate a common data analysis step



“Faster” to good science
AND
Simple enough not to be a distraction for a PhD Student

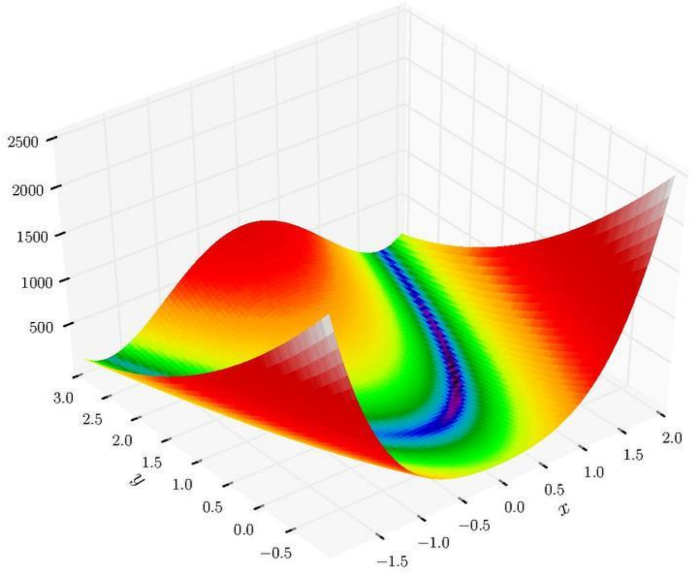
Function minimisation

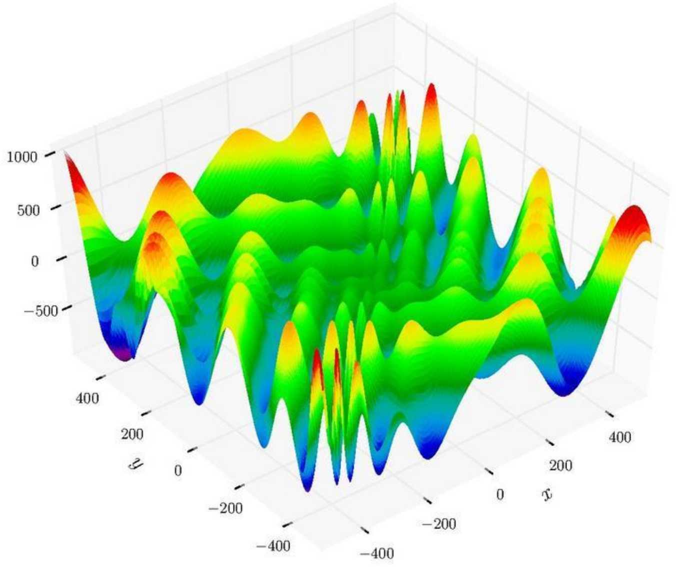
Find:

$$\underset{\vec{x} \in R^n}{\operatorname{argmin}} \; f(\vec{x})$$

$f(\vec{x})$ single valued function

Example algorithms: Nelder-Mead downhill simplex, gradient descent, Broyden Fletcher Goldfarb Shanno (BFGS), MCMC etc. Function gradient often used





Graphics: wikipedia

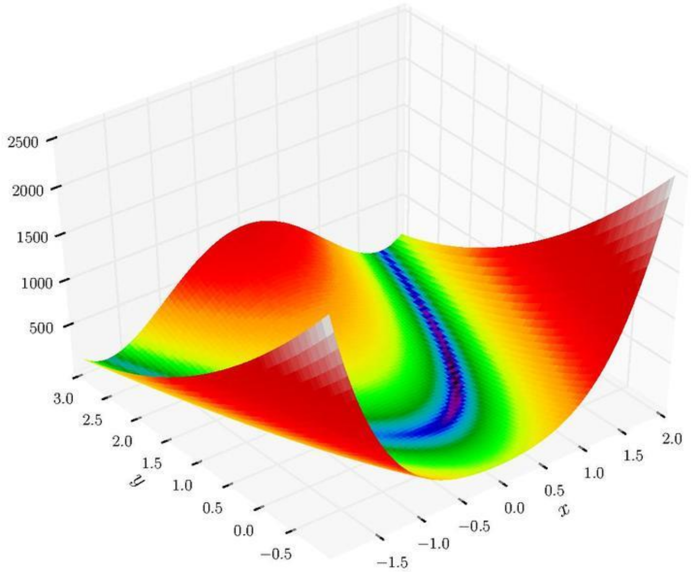
Function minimisation

Find:

$$\underset{\vec{x} \in R^n}{\operatorname{argmin}} f(\vec{x})$$

$f(\vec{x})$ single valued function

Example algorithms: Nelder-Mead downh implex, gradient descent, Broyden Fletcher Goldfarb Shanno (BFGS), MCMC etc. Fun gradient often used



Graphics: wikipedia

Maximum Likelihood
Maximum A-Posteriori
System Design Optimisation
Observing strategy
Optimisation

Strategy

Calculate:

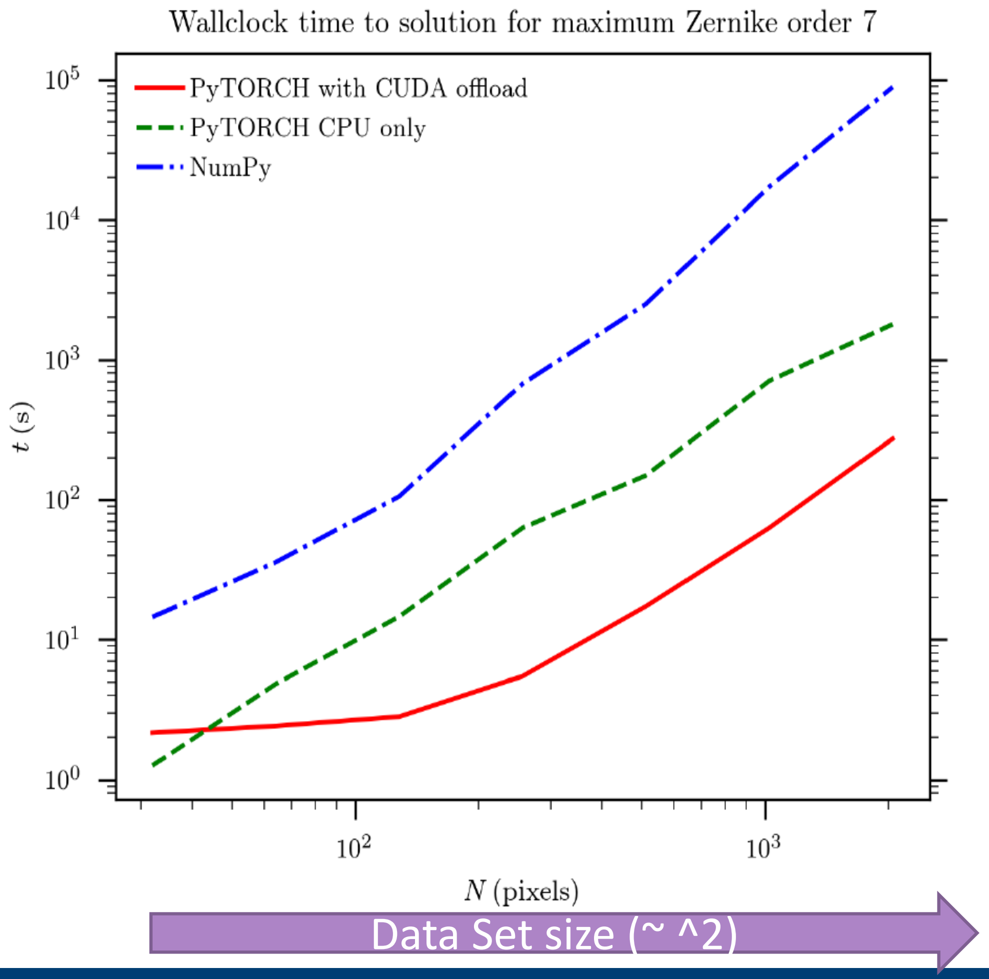
$$f(\boldsymbol{x}) \text{ and } \nabla f(\boldsymbol{x}) = \frac{\partial f}{\partial x_i}$$

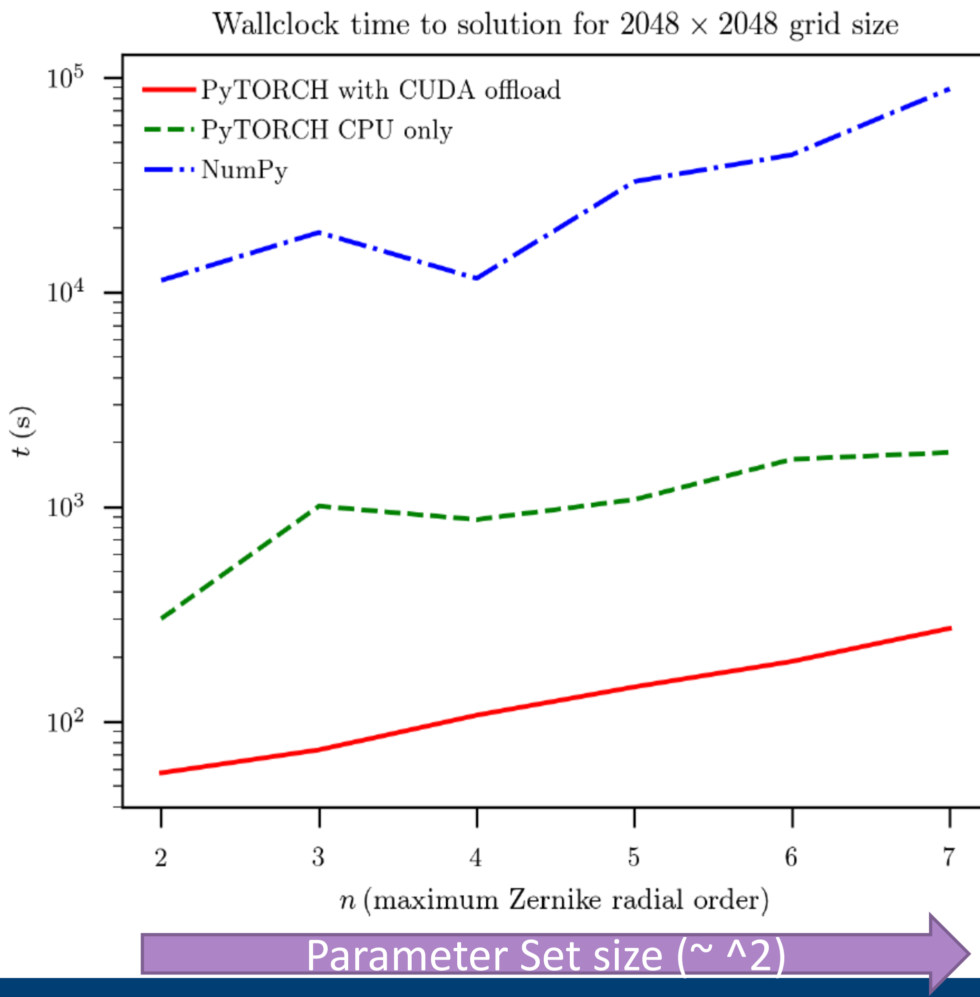
quickly & <u>easily</u>

- Best results for $f(\boldsymbol{x})$ that takes lots of data, uses arrays and has few iterations
- Will say nothing about the minimisation algorithms themselves!
- Useful for function minimisation/maximisation but presumably in other areas too

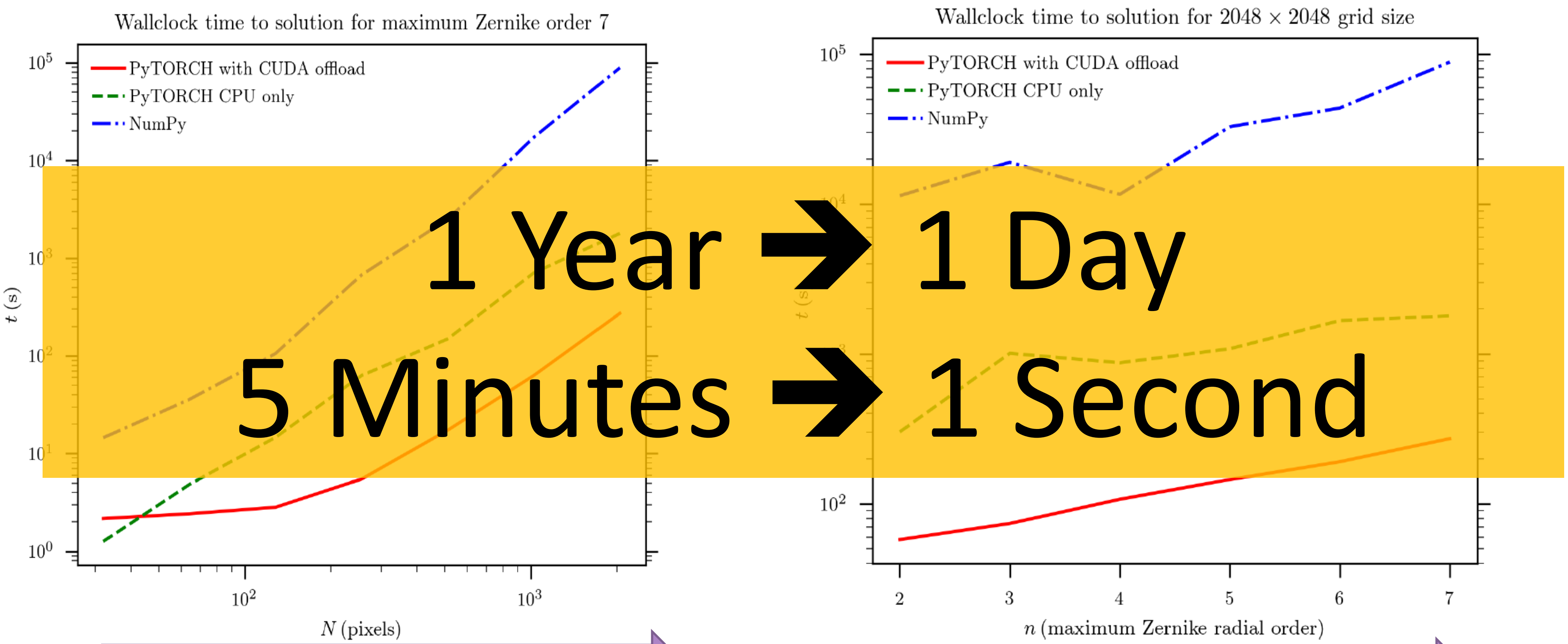
Results

Performance comparison





Performance comparison



1 Year ➔ 1 Day

5 Minutes ➔ 1 Second

Data Set size (~ ^2)

Parameter Set size (~ ^2)

NumPy vs PyTorch code comparison

NumPy:

```
def gauss(x0, y0,
          amp,
          sigma, rho, diff,
          a):
    dx=a[...,0]-x0
    dy= a[...,1]-y0
    r=numpy.hypot(dx, dy)
    R2= (r**2 +
         rho*(dx*dy)+
         diff*(dx**2-dy**2))
    E=numpy.exp(-1.0/
         (2*sigma**2)*R2)
    return amp*E
```

PyTorch:

```
import torch as T
def hypot(x, y):
    return T.sqrt(x**2 + y**2)

def gauss(x0, y0,
          amp,
          sigma, rho, diff,
          a):
    dx=a[...,0]-x0
    dy= a[...,1]-y0
    r=hypot(dx, dy)
    R2=(r**2 +
        OO(rho*(dx*dy))+
        OO(diff*(dx**2-dy**2)))
    E=T.exp(-1.0/(2*sigma**2)*R2)
    return OO(amp*E)
```

NumPy vs PyTorch code comparison

NumPy:

```
def gauss(x0, y0,
          amp,
          sigma, rho, diff,
          a):
    dx=a[...,0]-x0
    dy= a[...,1]-y0
    r=numpy.hypot(dx, dy)
    R2= (r**2 +
         rho*(dx*dy)+
         diff*(dx**2-dy**2))
    E=numpy.exp(-1.0/
         (2*sigma**2)*R2)
    return amp*E
```

PyTorch:

```
import torch as T
def hypot(x, y):
    return T.sqrt(x**2 + y**2)

def gauss(x0, y0,
          amp,
          sigma, rho, diff,
          a):
    dx=a[...,0]-x0
    dy= a[...,1]-y0
    r=hypot(dx, dy)
    R2=(r**2 +
        OO(rho*(dx*dy))+
        OO(diff*(dx**2-dy**2)))
    E=T.exp(-1.0/(2*sigma**2)*R2)
    return OO(amp*E)
```

A PhD student has been inoculated

….

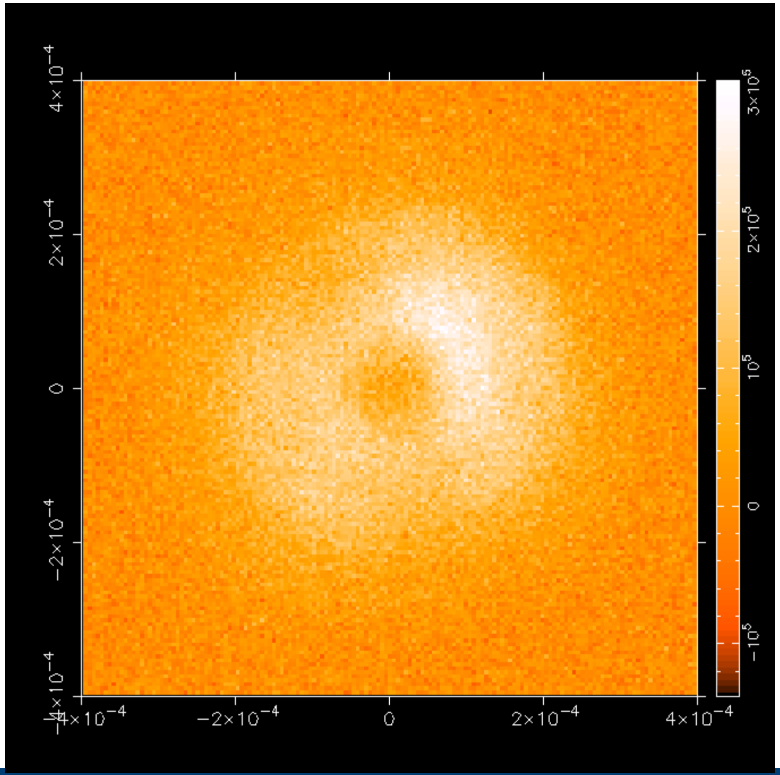
And is still able to do science!

What?

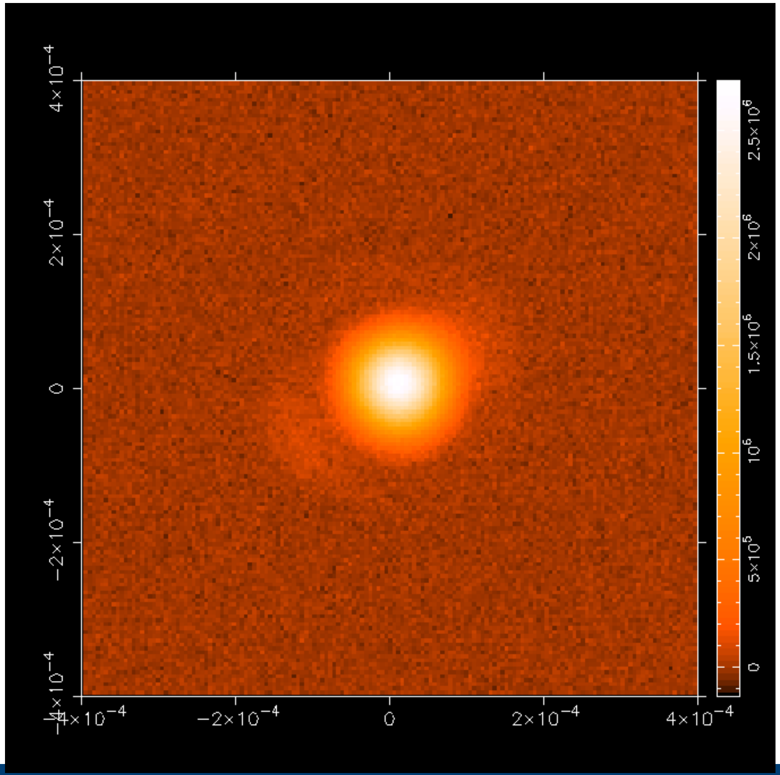
OOF Holography – a technique to measuring telescopes

These are effectively the measured PSFs of the telescope

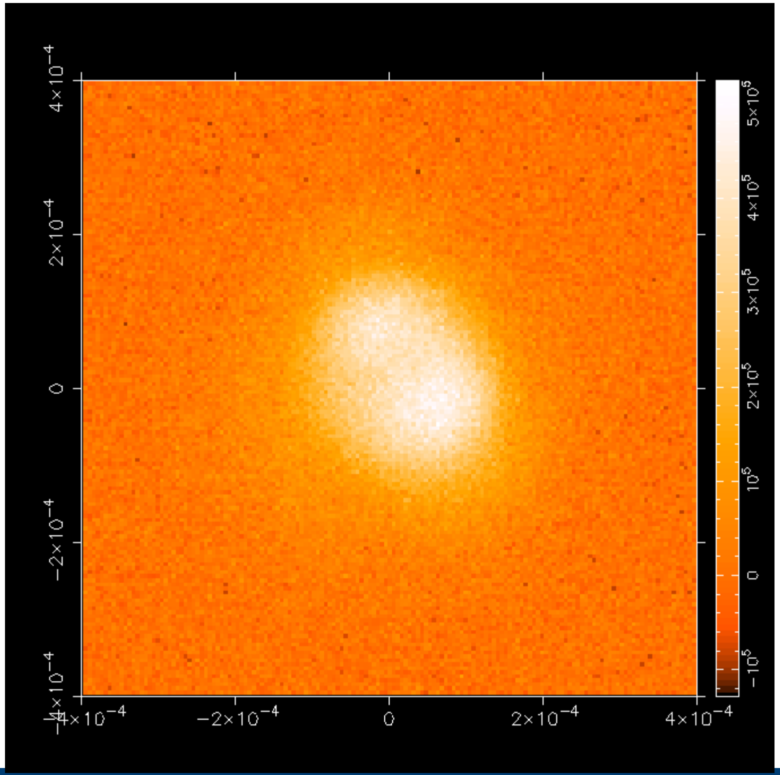
-3mm defocus



In-focus



+3mm defocus



Likelihood function

TRADITIONAL

$$P(y|\hat{y}) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{(y-\hat{y})^2}{2\sigma^2}}$$

Likelihood for a set of observations

$$P(\{y_i\}) = \prod_i P(y_i|\hat{y}_i)$$

Log-likelihood is equivalent to **least-squares** problem

<u>Not efficient</u> to reduce likelihood to a single valued function!

"ROBUST"

E.g. Cauchy distribution:

$$P(y|\hat{y}) = \frac{1}{\pi\gamma} \frac{\gamma^2}{(y-\hat{y})^2+\gamma^2}$$

Captures the possibility of outliers (glitches in read-out, short term pointing instability in telescope, atmospheric disturbance)

Log-likelihood **does not** factor into least-squares

Likelihood function

TRADITIONAL

$$P(y|\hat{y}) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{(y-\hat{y})^2}{2\sigma^2}}$$

Likelihood for a set of observations

$$P(\{y_i\}) = \prod_i P(y_i|\hat{y}_i)$$

Log-likelihood is equivalent to **least-squares** problem

<u>Not efficient</u> to reduce likelihood to a single valued function!

"ROBUST"

E.g. Cauchy distribution:

$$P(y|\hat{y}) = \frac{1}{\pi\gamma} \frac{\gamma^2}{(y-\hat{y})^2+\gamma^2}$$

Captures the possibility of outliers (glitches in read-out, short term pointing instability in telescope, atmospheric disturbance)

Log-likelihood **does not** factor into least-squares

Why PyTorch?

Machine Learning as Function Minimisation

$\mathcal{D}_{\text{train}}$: $\{(X_i, \widehat{Y}_i)\}$, the training data set

$M(X;\ \Theta) \rightarrow Y$: Predictor

X : An observation (e.g., pixelated image)

Y : Prediction/classification/etc

Θ : Predictor parameters (to be learned) – e.g., weights, biases of a neural network

$\mathcal{L}(\Theta; M, \mathcal{D}_{\text{train}}) = \sum_i L\left(\widehat{Y}_i - M(X;\ \Theta)\right)$: The total "loss" function

L : individual loss function, could be $L_{1,}$ L_2 or something more tailored

"Learning" is (approximately) minimising $\mathcal{L}$ with respect to Θ

PyTorch

PyTorch

Tensors and Dynamic neural networks in Python with strong GPU acceleration

Installation:

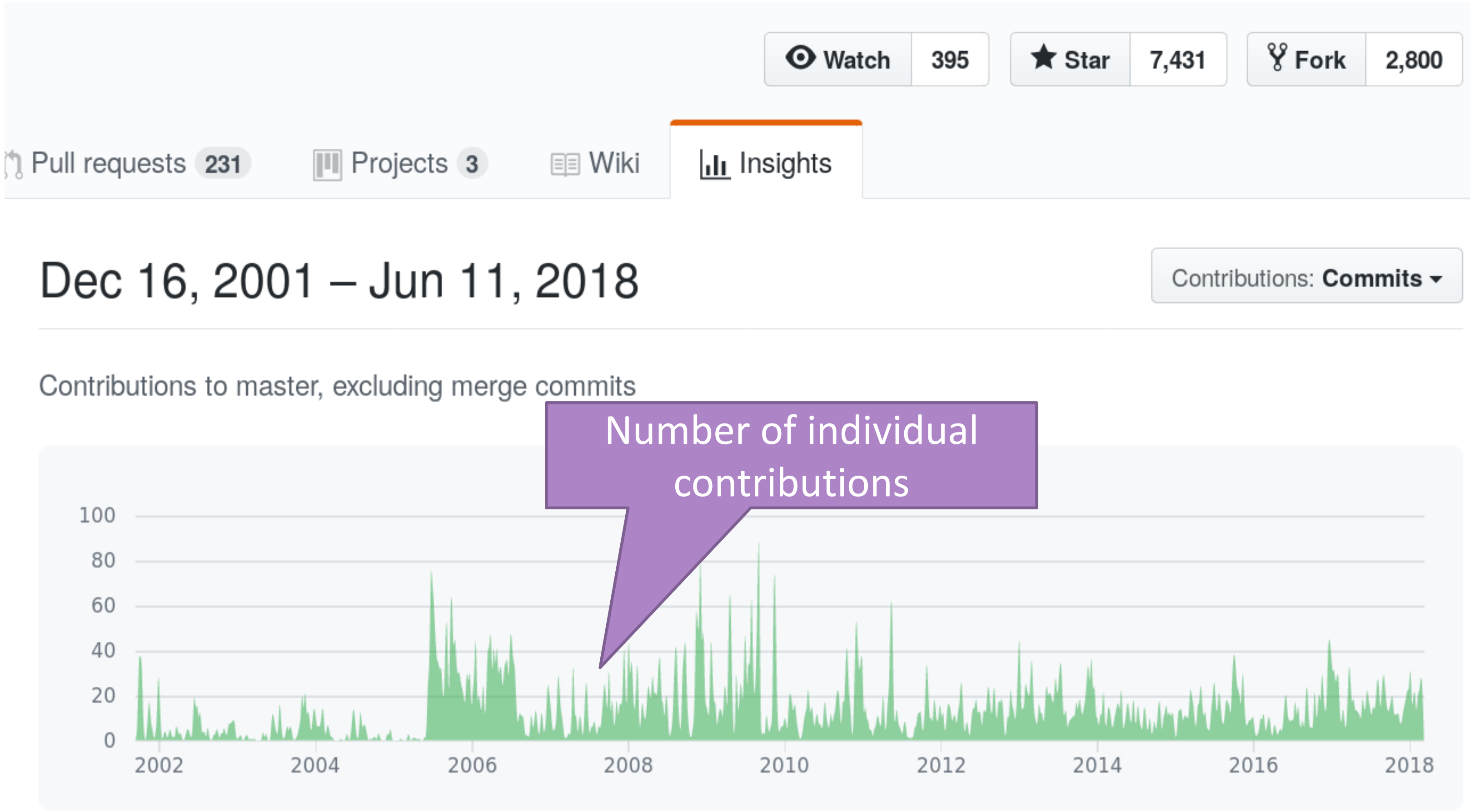
conda install pytorch cuda91 –c pytorch

Automatic differentiation
Trivially easy to offload to GPUs:

NumPy Contributions

Plot on GitHub of contribution frequency over lifetime of the project

NumPy is the main workhorse of numerical data analysis in Python. It is evolution of a library starting in 1996 (numeric, numarrays, etc)



PyTorch Contributions

Plot on GitHub of contribution frequency over lifetime of the project

Sustained, intense contribution

Not usually seen in community-led sw

How?

What can ML software offer?

Acceleration	NUMA	Non-Uniform Interconnect	Distributed Memory Cluster Scaling
High Input Data Throughput	High Quality Programming Interfaces	Automatic Differentiation	Task Segmentation
Working Memory Management	Efficient large scale minimisers	+++	

What can ML software offer -- example

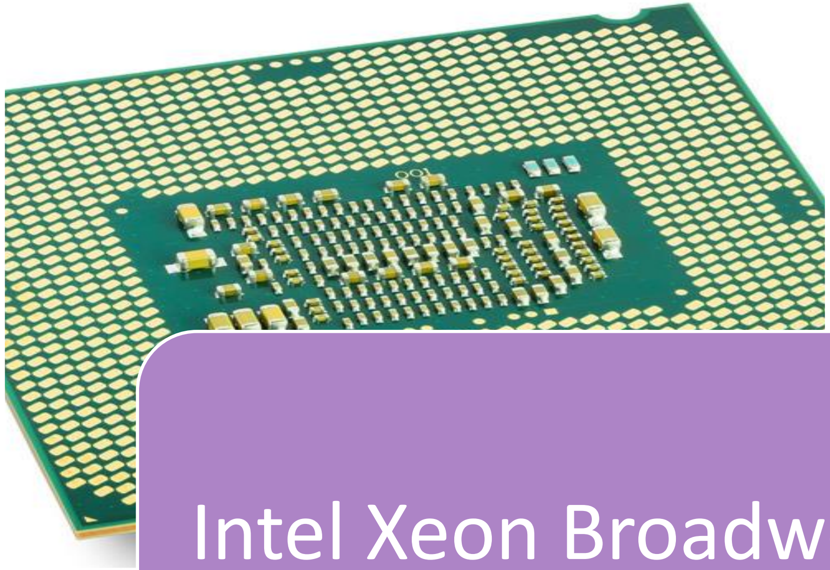
Acceleration	NUMA	Non-Uniform Interconnect	Distributed Memory Cluster Scaling
High Input Data Throughput	High Quality Programming Interfaces	Automatic Differentiation	Task Segmentation
Working Memory Management	Efficient large scale minimisers	+++	

Why (what is) acceleration ?

Intel 8087 (Wikipedia, by Rautakorbi)

One Floating Point Unit needs around 20k transistors

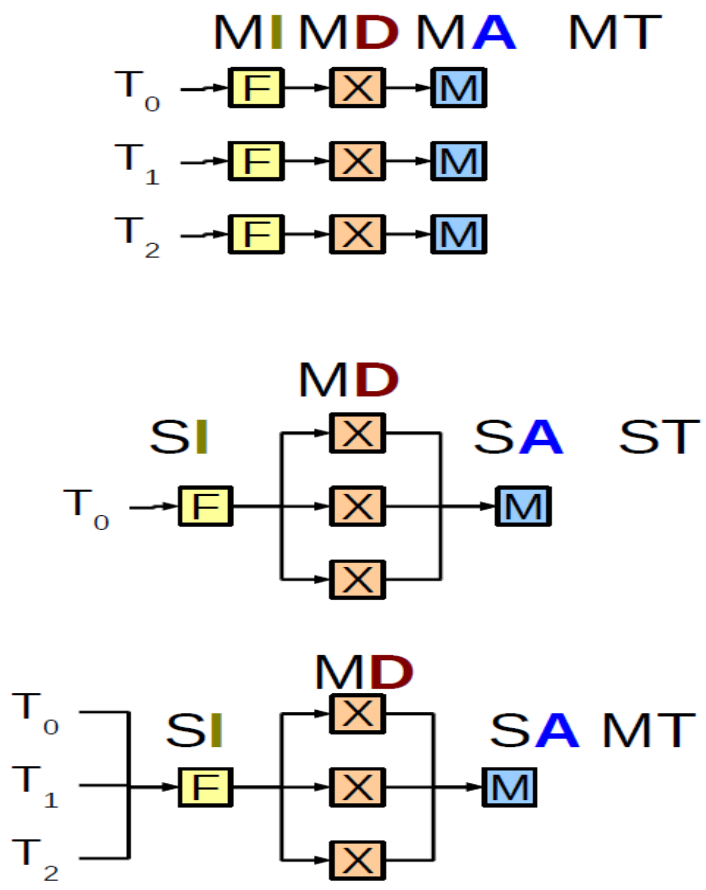
Complete FP Co-proc 45k transistors



Intel Xeon Broadwell E5 V4 – 7 billion transistors

GPU Acceleration

- Multi-core
  - MIMD(MAMT)
- Short-vector SIMD
  - SIMD(SAST)
- GPU
  - SI(MDSA)MT



From very nice slide deck by Sylvain Collange (2011)

Automatic Differentiation

$$h = H(x) \qquad f = F(G(H(x)))$$

$$g = G(h) \qquad h_0 = H(x_0)$$

$$f = F(g) \qquad g_0 = G(h_0)$$

$$\xrightarrow{x} H \xrightarrow{h} G \xrightarrow{g} F \xrightarrow{f}$$

Symbolic Differentiation

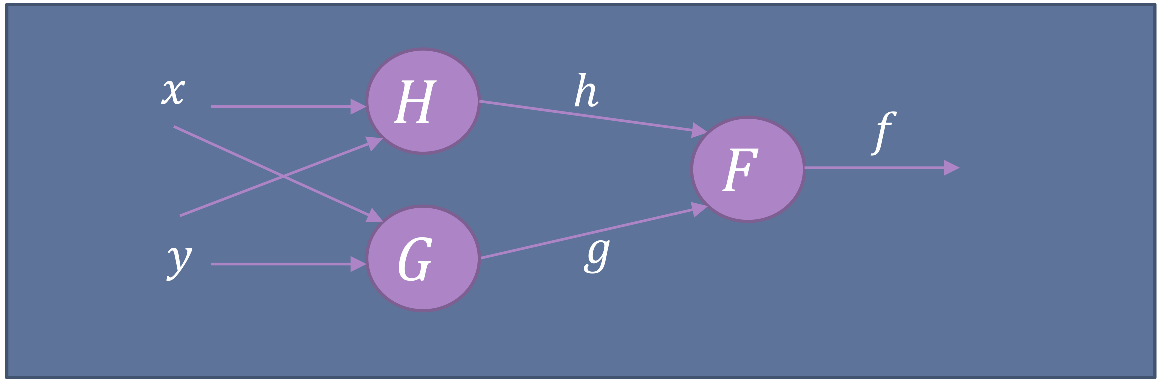
- $\left.\frac{df}{dx}\right|_{x_0} = \left.\left(\frac{dF}{dg}\frac{dG}{dh}\frac{dH}{dx}\right)\right|_{x_0}$

Automatic differentiation

- $\left.\frac{df}{dx}\right|_{x_0} = \left.\frac{dF}{dg}\right|_{g_0} \left.\frac{dG}{dh}\right|_{h_0} \left.\frac{dH}{dx}\right|_{x_0}$

Reverse-Mode Automatic Differentiation

$$h = H(x, y)$$
$$g = G(x, y)$$
$$f = F(h, g)$$



$$\left.\frac{\partial f}{\partial x}\right|_{x_0,y_0} = \left.\frac{\partial F}{\partial h}\right|_{h_0,g_0} \left.\frac{\partial h}{\partial x}\right|_{x_0,y_0} + \left.\frac{\partial F}{\partial g}\right|_{h_0,g_0} \left.\frac{\partial g}{\partial x}\right|_{x_0,y_0}$$

$$\left.\frac{\partial f}{\partial y}\right|_{x_0,y_0} = \left.\frac{\partial F}{\partial h}\right|_{h_0,g_0} \left.\frac{\partial h}{\partial y}\right|_{x_0,y_0} + \left.\frac{\partial F}{\partial g}\right|_{h_0,g_0} \left.\frac{\partial g}{\partial y}\right|_{x_0,y_0}$$

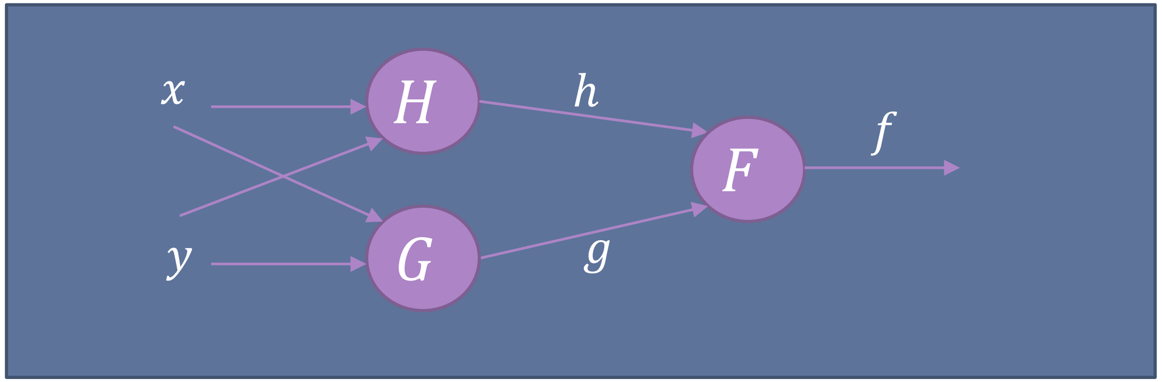
Standard Chain Rule

Reverse-Mode Automatic Differentiation

$$h = H(x, y)$$

$$g = G(x, y)$$

$$f = F(h, g)$$



$$\left.\frac{\partial f}{\partial x}\right|_{x_0,y_0} = \left.\frac{\partial F}{\partial h}\right|_{h_0,g_0} \left.\frac{\partial h}{\partial x}\right|_{x_0,y_0} + \left.\frac{\partial F}{\partial g}\right|_{h_0,g_0} \left.\frac{\partial g}{\partial x}\right|_{x_0,y_0}$$

$$\left.\frac{\partial f}{\partial y}\right|_{x_0,y_0} = \left.\frac{\partial F}{\partial h}\right|_{h_0,g_0} \left.\frac{\partial h}{\partial y}\right|_{x_0,y_0} + \left.\frac{\partial F}{\partial g}\right|_{h_0,g_0} \left.\frac{\partial g}{\partial y}\right|_{x_0,y_0}$$

Standard Chain Rule

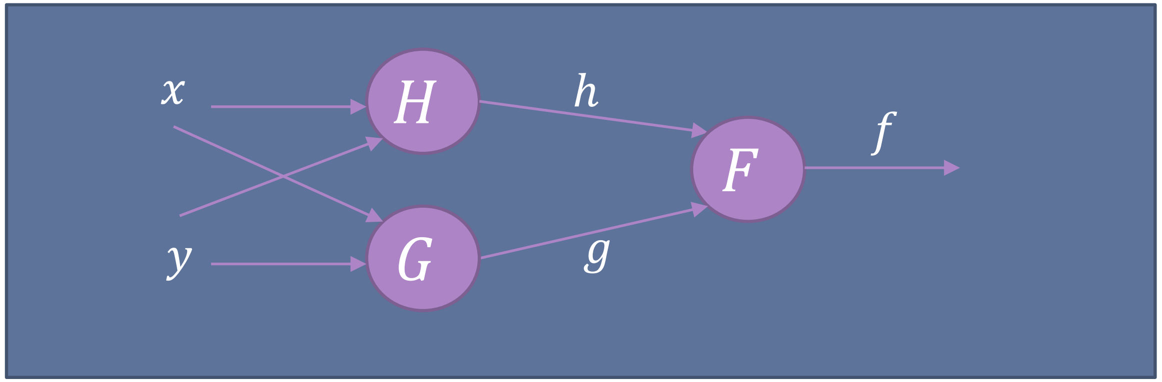
Huge efficiency gain by re-using these (scalar) values

Why reverse?

$$h = H(x, y)$$

$$g = G(x, y)$$

$$f = F(h, g)$$



$$\left.\frac{\partial f}{\partial x}\right|_{x_0,y_0} = \left.\frac{\partial h}{\partial x}\right|_{x_0,y_0} \left.\frac{\partial F}{\partial h}\right|_{h_0,g_0} + \left.\frac{\partial g}{\partial x}\right|_{x_0,y_0} \left.\frac{\partial F}{\partial g}\right|_{h_0,g_0}$$

$$\left.\frac{\partial f}{\partial y}\right|_{x_0,y_0} = \left.\frac{\partial h}{\partial y}\right|_{x_0,y_0} \left.\frac{\partial F}{\partial h}\right|_{h_0,g_0} + \left.\frac{\partial g}{\partial y}\right|_{x_0,y_0} \left.\frac{\partial F}{\partial g}\right|_{h_0,g_0}$$

Standard Chain Rule

Need to evaluate gradient in reverse order compared to program flow

Summary

Summary

>100x performance improvement in minimising functions

Small, contained, software effort needed

- Perfect integration with standard Python environment

Out-of-box support for GPUs and multi-threaded CPUs

Easy to use (& install!)

More details: arXiv:1805.07439