Automatic differentiation for applied mathematicians

Is PyTorch the right tool for you?

Jean Feydy

February 2018

Écoles Normales Supérieures de Paris et Paris-Saclay

Overview

What is backpropagation?

A time-efficient algorithm to compute gradients.

Overview

What is backpropagation?

A time-efficient algorithm to compute gradients.

How can we use it?

`PyTorch` provides a simple syntax, transparent CPU/GPU support.

Overview

What is backpropagation?

A time-efficient algorithm to compute gradients.

How can we use it?

`PyTorch` provides a simple syntax, transparent CPU/GPU support.

Where is it useful?

Automatic tuning of variables (optimal control)
or of your transform's parameters (neural networks).

Overview

What is backpropagation?

A time-efficient algorithm to compute gradients.

How can we use it?

`PyTorch` provides a simple syntax, transparent CPU/GPU support.

Where is it useful?

Automatic tuning of variables (optimal control)
or of your transform's parameters (neural networks).

Bonus: you can extend it easily.

Link with your homebrew CUDA routines!

How do we compute a gradient?

Let $f : \mathbb{R}^n \to \mathbb{R}$ be a smooth function. Then:

$$\nabla f(x_0) = \begin{pmatrix} \partial_{x^1} f(x_0) \\ \partial_{x^2} f(x_0) \\ \vdots \\ \partial_{x^n} f(x_0) \end{pmatrix} \simeq \frac{1}{\delta t} \begin{pmatrix} f(x_0 + \delta t \cdot (1, 0, \dots, 0)) - f(x_0) \\ f(x_0 + \delta t \cdot (0, 1, \dots, 0)) - f(x_0) \\ \vdots \\ f(x_0 + \delta t \cdot (0, 0, \dots, 1)) - f(x_0) \end{pmatrix}.$$

How do we compute a gradient?

Let $f : \mathbb{R}^n \to \mathbb{R}$ be a smooth function. Then:

$$\nabla f(x_0) = \begin{pmatrix} \partial_{x^1} f(x_0) \\ \partial_{x^2} f(x_0) \\ \vdots \\ \partial_{x^n} f(x_0) \end{pmatrix} \simeq \frac{1}{\delta t} \begin{pmatrix} f(x_0 + \delta t \cdot (1, 0, \ldots, 0)) - f(x_0) \\ f(x_0 + \delta t \cdot (0, 1, \ldots, 0)) - f(x_0) \\ \vdots \\ f(x_0 + \delta t \cdot (0, 0, \ldots, 1)) - f(x_0) \end{pmatrix}.$$

$\implies$ costs **(n+1) evaluations of** f, which is poor.

What is a gradient?

Let $f: x \in \mathbb{R}^n \mapsto y \in \mathbb{R}$ be smooth,

What is a gradient?

Let $f: x \in \mathbb{R}^n \mapsto y \in \mathbb{R}$ be smooth,

So that $\mathrm{d}f(x) : \mathrm{d}x \in \mathbb{R}^n \mapsto \mathrm{d}y \in \mathbb{R}$ is linear:

What is a gradient?

$$\begin{array}{rcl} \text{Let} & f: x \in \mathbb{R}^n \mapsto y \in \mathbb{R} & \text{be smooth,} \\ \text{So that} & \mathrm{d}f(x) : \mathrm{d}x \in \mathbb{R}^n \mapsto \mathrm{d}y \in \mathbb{R} & \text{is linear:} \end{array}$$

$$\mathrm{d}f(x).\mathrm{d}x = \begin{pmatrix} \partial_1 f(x) & \cdots & \partial_n f(x) \end{pmatrix} \cdot \begin{pmatrix} \mathrm{d}x_1 \\ \vdots \\ \mathrm{d}x_n \end{pmatrix} = \begin{pmatrix} \mathrm{d}y \end{pmatrix}$$

What is a gradient?

Let $f : x \in \mathbb{R}^n \mapsto y \in \mathbb{R}$ be smooth,

So that $\mathrm{d}f(x) : \mathrm{d}x \in \mathbb{R}^n \mapsto \mathrm{d}y \in \mathbb{R}$ is linear:

$$\mathrm{d}f(x).\mathrm{d}x = \begin{pmatrix} \partial_1 f(x) & \cdots & \partial_n f(x) \end{pmatrix} \cdot \begin{pmatrix} \mathrm{d}x_1 \\ \vdots \\ \mathrm{d}x_n \end{pmatrix} = \begin{pmatrix} \mathrm{d}y \end{pmatrix}$$

We define $\partial \mathbf{f}(\mathbf{x}) = (\mathrm{d}\mathbf{f}(\mathbf{x}))^\star$

What is a gradient?

Let $f: x \in \mathbb{R}^n \mapsto y \in \mathbb{R}$ be smooth,

So that $\mathrm{d}f(x) : \mathrm{d}x \in \mathbb{R}^n \mapsto \mathrm{d}y \in \mathbb{R}$ is linear:

$$\mathrm{d}f(x).\mathrm{d}x = \begin{pmatrix} \partial_1 f(x) & \cdots & \partial_n f(x) \end{pmatrix} \cdot \begin{pmatrix} \mathrm{d}x_1 \\ \vdots \\ \mathrm{d}x_n \end{pmatrix} = \begin{pmatrix} \mathrm{d}y \end{pmatrix}$$

We define $\partial \mathbf{f}(\mathbf{x}) = (\mathrm{d}\mathbf{f}(\mathbf{x}))^\star \simeq (\mathrm{d}\mathbf{f}(\mathbf{x}))^\top$

What is a gradient?

Let $f: x \in \mathbb{R}^n \mapsto y \in \mathbb{R}$ be smooth,

So that $\mathrm{d}f(x) : \mathrm{d}x \in \mathbb{R}^n \mapsto \mathrm{d}y \in \mathbb{R}$ is linear:

$$\mathrm{d}f(x).\mathrm{d}x = \begin{pmatrix} \partial_1 f(x) & \cdots & \partial_n f(x) \end{pmatrix} \cdot \begin{pmatrix} \mathrm{d}x_1 \\ \vdots \\ \mathrm{d}x_n \end{pmatrix} = \begin{pmatrix} \mathrm{d}y \end{pmatrix}$$

We define $\partial \mathbf{f}(\mathbf{x}) = (\mathrm{d}\mathbf{f}(\mathbf{x}))^\star \simeq (\mathrm{d}\mathbf{f}(\mathbf{x}))^\top$

i.e. $\partial f(x) : \mathrm{d}y^\star \in \mathbb{R} \mapsto \mathrm{d}x^\star \in \mathbb{R}^n$.

What is a gradient?

Let $f : x \in \mathbb{R}^n \mapsto y \in \mathbb{R}$ be smooth,

So that $\mathrm{d}f(x) : \mathrm{d}x \in \mathbb{R}^n \mapsto \mathrm{d}y \in \mathbb{R}$ is linear:

$$\mathrm{d}f(x).\mathrm{d}x = \begin{pmatrix} \partial_1 f(x) & \cdots & \partial_n f(x) \end{pmatrix} \cdot \begin{pmatrix} \mathrm{d}x_1 \\ \vdots \\ \mathrm{d}x_n \end{pmatrix} = \begin{pmatrix} \mathrm{d}y \end{pmatrix}$$

We define $\partial \mathbf{f}(\mathbf{x}) = (\mathrm{d}\mathbf{f}(\mathbf{x}))^\star \simeq (\mathrm{d}\mathbf{f}(\mathbf{x}))^\top$

i.e. $\partial f(x) : \mathrm{d}y^\star \in \mathbb{R} \mapsto \mathrm{d}x^\star \in \mathbb{R}^n$.

$$\partial f(x).\mathrm{d}y^\star = \begin{pmatrix} \partial_1 f(x) \\ \vdots \\ \partial_n f(x) \end{pmatrix} \cdot \begin{pmatrix} \mathrm{d}y^\star \end{pmatrix} = \begin{pmatrix} \mathrm{d}x^\star \end{pmatrix} \quad \text{so that} \quad \nabla f(x) = \partial f(x).1$$

Autodiff is simple – no magic!

This definition lets us **compose gradients:**

Autodiff is simple – no magic!

This definition lets us **compose gradients:**

$$f \;=\; h \circ g$$

Autodiff is simple – no magic!

This definition lets us **compose gradients:**

$$
\begin{aligned}
f &= h \circ g \\
\mathrm{d}f(x) &= \mathrm{d}h(g(x)) \circ \mathrm{d}g(x)
\end{aligned}
$$

Autodiff is simple – no magic!

This definition lets us **compose gradients:**

$$
\begin{aligned}
f &= h \circ g \\
\mathrm{d}f(x) &= \mathrm{d}h(g(x)) \circ \mathrm{d}g(x) \\
\partial f(x) &= (\mathrm{d}h(g(x)) \circ \mathrm{d}g(x))^{\mathsf{T}} = \partial g(x) \circ \partial h(g(x))
\end{aligned}
$$

Autodiff is simple – no magic!

This definition lets us **compose gradients:**

$$
\begin{aligned}
f &= h \circ g \\
\mathrm{d}f(x) &= \mathrm{d}h(g(x)) \circ \mathrm{d}g(x) \\
\partial f(x) &= (\mathrm{d}h(g(x)) \circ \mathrm{d}g(x))^{\top} = \partial g(x) \circ \partial h(g(x)) \\
\nabla f(x) &= \partial f(x).1 = \partial g(x).(\partial h(g(x)).1)
\end{aligned}
$$

Autodiff is simple – no magic!

This definition lets us **compose gradients:**

$$
\begin{aligned}
f &= h \circ g \\
\mathrm{d}f(x) &= \mathrm{d}h(g(x)) \circ \mathrm{d}g(x) \\
\partial f(x) &= (\mathrm{d}h(g(x)) \circ \mathrm{d}g(x))^{\top} = \partial g(x) \circ \partial h(g(x)) \\
\nabla f(x) &= \partial f(x).1 = \partial g(x).(\partial h(g(x)).1)
\end{aligned}
$$

$$x \xrightarrow{\textbf{input}} x$$

Autodiff is simple – no magic!

This definition lets us **compose gradients:**

$$
\begin{aligned}
f &= h \circ g \\
\mathrm{d}f(x) &= \mathrm{d}h(g(x)) \circ \mathrm{d}g(x) \\
\partial f(x) &= (\mathrm{d}h(g(x)) \circ \mathrm{d}g(x))^{\top} = \partial g(x) \circ \partial h(g(x)) \\
\nabla f(x) &= \partial f(x).1 = \partial g(x).(\partial h(g(x)).1)
\end{aligned}
$$

$$x \xrightarrow{\textbf{input}} x \xrightarrow{\quad g \quad} g(x)$$

Autodiff is simple – no magic!

This definition lets us **compose gradients:**

$$
\begin{aligned}
f &= h \circ g \\
\mathrm{d}f(x) &= \mathrm{d}h(g(x)) \circ \mathrm{d}g(x) \\
\partial f(x) &= (\mathrm{d}h(g(x)) \circ \mathrm{d}g(x))^{\top} = \partial g(x) \circ \partial h(g(x)) \\
\nabla f(x) &= \partial f(x).1 = \partial g(x).(\partial h(g(x)).1)
\end{aligned}
$$

Autodiff is simple – no magic!

This definition lets us **compose gradients:**

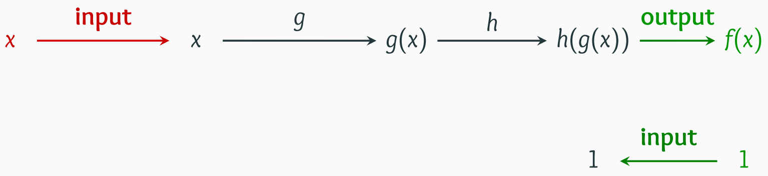
$$
\begin{aligned}
f &= h \circ g \\
\mathrm{d}f(x) &= \mathrm{d}h(g(x)) \circ \mathrm{d}g(x) \\
\partial f(x) &= (\mathrm{d}h(g(x)) \circ \mathrm{d}g(x))^{\top} = \partial g(x) \circ \partial h(g(x)) \\
\nabla f(x) &= \partial f(x).1 = \partial g(x).(\partial h(g(x)).1)
\end{aligned}
$$

$$x \xrightarrow{\textbf{input}} x \xrightarrow{\quad g \quad} g(x) \xrightarrow{\quad h \quad} h(g(x)) \xrightarrow{\textbf{output}} f(x)$$

Autodiff is simple – no magic!

This definition lets us **compose gradients:**

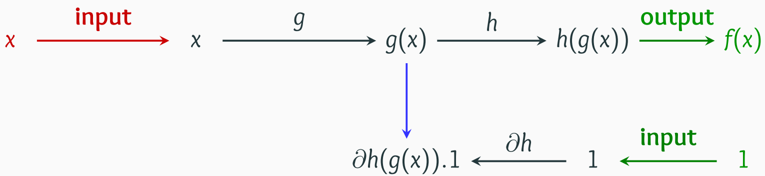
$$
\begin{aligned}
f &= h \circ g \\
\mathrm{d}f(x) &= \mathrm{d}h(g(x)) \circ \mathrm{d}g(x) \\
\partial f(x) &= (\mathrm{d}h(g(x)) \circ \mathrm{d}g(x))^\top = \partial g(x) \circ \partial h(g(x)) \\
\nabla f(x) &= \partial f(x).1 = \partial g(x).(\partial h(g(x)).1)
\end{aligned}
$$



Autodiff is simple – no magic!

This definition lets us **compose gradients:**

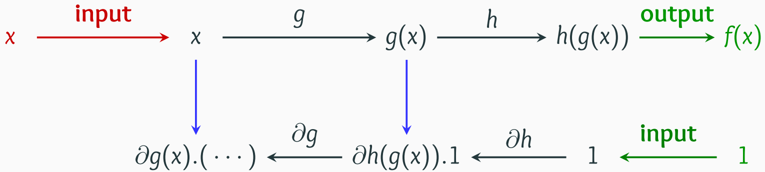
$$\begin{aligned} f &= h \circ g \\ \mathrm{d}f(x) &= \mathrm{d}h(g(x)) \circ \mathrm{d}g(x) \\ \partial f(x) &= (\mathrm{d}h(g(x)) \circ \mathrm{d}g(x))^\intercal = \partial g(x) \circ \partial h(g(x)) \\ \nabla f(x) &= \partial f(x).1 = \partial g(x).(\partial h(g(x)).1) \end{aligned}$$



Autodiff is simple – no magic!

This definition lets us **compose gradients:**

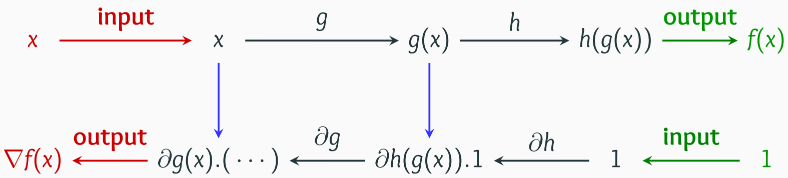
$$
\begin{aligned}
f &= h \circ g \\
\mathrm{d}f(x) &= \mathrm{d}h(g(x)) \circ \mathrm{d}g(x) \\
\partial f(x) &= (\mathrm{d}h(g(x)) \circ \mathrm{d}g(x))^\top = \partial g(x) \circ \partial h(g(x)) \\
\nabla f(x) &= \partial f(x).1 = \partial g(x).(\partial h(g(x)).1)
\end{aligned}
$$



Autodiff is simple – no magic!

This definition lets us **compose gradients:**

$$\begin{aligned}
f &= h \circ g \\
\mathrm{d}f(x) &= \mathrm{d}h(g(x)) \circ \mathrm{d}g(x) \\
\partial f(x) &= (\mathrm{d}h(g(x)) \circ \mathrm{d}g(x))^{\mathsf{T}} = \partial g(x) \circ \partial h(g(x)) \\
\nabla f(x) &= \partial f(x).1 = \partial g(x).(\partial h(g(x)).1)
\end{aligned}$$



What do you need to compute a gradient?

Backpropagating through a computational graph requires:

$$\begin{aligned} f_i \quad : \quad E_{i-1} &\longrightarrow E_i \\ x &\longmapsto f_i(x) \end{aligned} \tag{1}$$

and

$$\begin{aligned} \partial_x f_i \quad : \quad E_{i-1} \times E_i &\longrightarrow E_{i-1} \\ (x_0, a) &\longmapsto \partial_x f_i(x_0) \cdot a \end{aligned} \tag{2}$$

encoded as **computer programs.**

What do you need to compute a gradient?

Backpropagating through a computational graph requires:

$$f_i \quad : \quad \begin{array}{ccc} E_{i-1} & \rightarrow & E_i \\ x & \mapsto & f_i(x) \end{array} \qquad (1)$$

and

$$\partial_x f_i \quad : \quad \begin{array}{ccc} E_{i-1} \times E_i & \rightarrow & E_{i-1} \\ (x_0, a) & \mapsto & \partial_x f_i(x_0) \cdot a \end{array} \qquad (2)$$

encoded as **computer programs.**

This is what PyTorch is all about.

Autodiff is easy to use

PyTorch :

- Straightforward replacement of Matlab/Numpy.

Autodiff is easy to use

PyTorch :

- Straightforward replacement of Matlab/Numpy.
- Operators $f : x \mapsto f(x)$ are bundled with their **adjoints** $\partial f : (x, \mathrm{d}y^\star) \mapsto \partial f(x).\mathrm{d}y^\star = \mathrm{d}x^\star$.

Autodiff is easy to use

PyTorch :

- Straightforward replacement of Matlab/Numpy.
- Operators $f : x \mapsto f(x)$ are bundled with their **adjoints** $\partial f : (\, x, \mathrm{d}y^\star) \mapsto \partial f(x).\mathrm{d}y^\star = \mathrm{d}x^\star$.
- Seamless **GPU** support.

Autodiff is easy to use

PyTorch :

- Straightforward replacement of Matlab/Numpy.
- Operators $f : x \mapsto f(x)$ are bundled with their **adjoints** $\partial f : (x, \mathrm{d}y^\star) \mapsto \partial f(x).\mathrm{d}y^\star = \mathrm{d}x^\star$.
- Seamless **GPU** support.

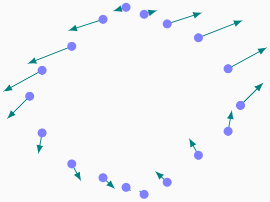
Autodiff is easy to use

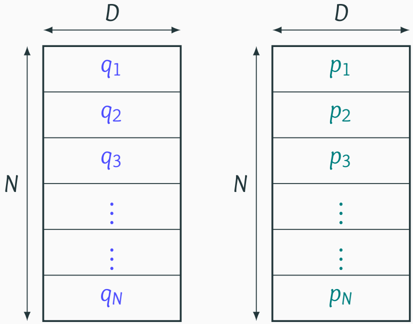
PyTorch :

- Straightforward replacement of Matlab/Numpy.
- Operators $f : x \mapsto f(x)$ are bundled with their **adjoints** $\partial f : (\, x, \mathrm{d}y^\star) \mapsto \partial f(x).\mathrm{d}y^\star = \mathrm{d}x^\star$.
- Seamless **GPU** support.

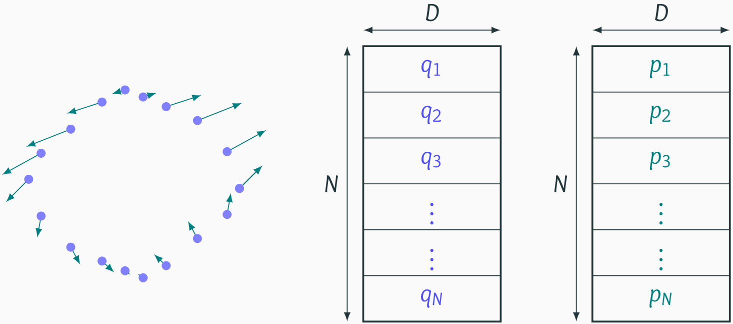
Let's see how it goes **in practice**!

A typical formula: the kernel square norm





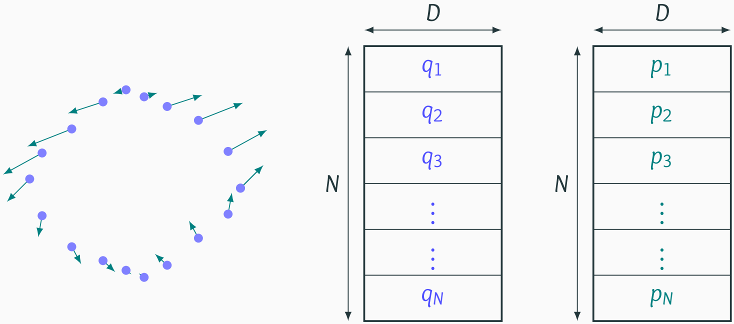
A typical formula: the kernel square norm



In shape analysis, algorithms often rely on the **kernel dot product:**

$$H(q, p) \;=\; \frac{1}{2} \sum_{i,j} \exp(-\tfrac{1}{\sigma^2} \|q_i - q_j\|^2) \, \langle p_i, p_j \rangle_2$$

A typical formula: the kernel square norm



In shape analysis, algorithms often rely on the **kernel dot product:**

$$H(q,p) \;=\; \frac{1}{2}\sum_{i,j} \exp(-\tfrac{1}{\sigma^2}\|q_i - q_j\|^2)\,\langle p_i, p_j\rangle_2$$

$$=\; \frac{1}{2}\sum_{i} \Big\langle p_i, \sum_{j} k(q_i - q_j)\, p_j\Big\rangle_2 \;=\; \frac{1}{2}\langle p, K_q p\rangle_2\,.$$

Numpy, in practice

```python
import numpy as np  # standard library
N = 5000 ; D = 3    # cloud of 5,000 points in 3D
q = np.random.rand(N,D)
p = np.random.rand(N,D)
s = 1.
```

Numpy, in practice

```
import numpy as np  # standard library
N = 5000 ; D = 3    # cloud of 5,000 points in 3D
q = np.random.rand(N,D)
p = np.random.rand(N,D)
s = 1.
# Re-indexing:
q_i  = q[:,np.newaxis,:] # shape (N,D) -> (N,1,D)
q_j  = q[np.newaxis,:,:] # shape (N,D) -> (1,N,D)
```

Numpy, in practice

```
import numpy as np  # standard library
N = 5000 ; D = 3    # cloud of 5,000 points in 3D
q = np.random.rand(N,D)
p = np.random.rand(N,D)
s = 1.
# Re-indexing:
q_i  = q[:,np.newaxis,:] # shape (N,D) -> (N,1,D)
q_j  = q[np.newaxis,:,:] # shape (N,D) -> (1,N,D)

# Actual computations:
sqd  = np.sum( (q_i - q_j)**2 , 2 ) # |q_i-q_j|^2
K_qq = np.exp( - sqd / s**2 )
v    = K_qq @ p # matrix mult. (N,N)@(N,D) = (N,D)
```

Numpy, in practice

```
import numpy as np  # standard library
N = 5000 ; D = 3    # cloud of 5,000 points in 3D
q = np.random.rand(N,D)
p = np.random.rand(N,D)
s = 1.
# Re-indexing:
q_i  = q[:,np.newaxis,:] # shape (N,D) -> (N,1,D)
q_j  = q[np.newaxis,:,:] # shape (N,D) -> (1,N,D)

# Actual computations:
sqd  = np.sum( (q_i - q_j)**2 , 2 ) # |q_i-q_j|^2
K_qq = np.exp( - sqd / s**2 )
v    = K_qq @ p # matrix mult. (N,N)@(N,D) = (N,D)

# Finally, output the kernel norm H(q,p): .5*<p,v>
H    = .5 * np.inner( p.ravel(), v.ravel() )
```

Numpy, in practice

```
import numpy as np  # standard library
N = 5000 ; D = 3    # cloud of 5,000 points in 3D
q = np.random.rand(N,D)
p = np.random.rand(N,D)
s = 1.
# Re-indexing:
q_i  = q[:,np.newaxis,:] # shape (N,D) -> (N,1,D)
q_j  = q[np.newaxis,:,:] # shape (N,D) -> (1,N,D)

# Actual computations:
sqd  = np.sum( (q_i - q_j)**2 , 2 ) # |q_i-q_j|^2
K_qq = np.exp( - sqd / s**2 )
v    = K_qq @ p # matrix mult. (N,N)@(N,D) = (N,D)

# Finally, output the kernel norm H(q,p): .5*<p,v>
H    = .5 * np.inner( p.ravel(), v.ravel() )
```

```
H = 6029309.1348486
Elapsed time: 3.01s
```

PyTorch, in practice

```python
import torch           # GPU + autodiff library
from   torch.autograd import grad

# With PyTorch, using the GPU is that simple:
use_gpu  = torch.cuda.is_available()
dtype    = torch.cuda.FloatTensor if use_gpu \
           else torch.FloatTensor
```

PyTorch, in practice

```
import torch              # GPU + autodiff library
from   torch.autograd import grad

# With PyTorch, using the GPU is that simple:
use_gpu  = torch.cuda.is_available()
dtype    = torch.cuda.FloatTensor if use_gpu \
           else torch.FloatTensor

# Store arbitrary arrays on the CPU or GPU:
q = torch.from_numpy( q ).type(dtype)
p = torch.from_numpy( p ).type(dtype)
s = torch.Tensor(  [1.] ).type(dtype)
```

PyTorch, in practice

```
import torch              # GPU + autodiff library
from   torch.autograd import grad

# With PyTorch, using the GPU is that simple:
use_gpu  = torch.cuda.is_available()
dtype    = torch.cuda.FloatTensor if use_gpu \
           else torch.FloatTensor
# Store arbitrary arrays on the CPU or GPU:
q = torch.from_numpy( q ).type(dtype)
p = torch.from_numpy( p ).type(dtype)
s = torch.Tensor(  [1.] ).type(dtype)
# Tell PyTorch to track the variables "q" and "p"
q.requires_grad = True
p.requires_grad = True
```

PyTorch, in practice

```
# Re-indexing:
q_i  = q[:,None,:] # shape (N,D) -> (N,1,D)
q_j  = q[None,:,:] # shape (N,D) -> (1,N,D)
```

PyTorch, in practice

```
# Re-indexing:
q_i  = q[:,None,:] # shape (N,D) -> (N,1,D)
q_j  = q[None,:,:] # shape (N,D) -> (1,N,D)
# Actual computations:
sqd  = torch.sum( (q_i - q_j)**2 , 2 ) # |q_i-q_j|^2
```

PyTorch, in practice

```
# Re-indexing:
q_i  = q[:,None,:] # shape (N,D) -> (N,1,D)
q_j  = q[None,:,:] # shape (N,D) -> (1,N,D)
# Actual computations:
sqd  = torch.sum( (q_i - q_j)**2 , 2 ) # |q_i-q_j|^2
K_qq = torch.exp( - sqd / s**2 )       # Gaussian kernel
```

PyTorch, in practice

```
# Re-indexing:
q_i  = q[:,None,:] # shape (N,D) -> (N,1,D)
q_j  = q[None,:,:] # shape (N,D) -> (1,N,D)
# Actual computations:
sqd  = torch.sum( (q_i - q_j)**2 , 2 ) # |q_i-q_j|^2
K_qq = torch.exp( - sqd / s**2 )       # Gaussian kernel
v    = K_qq @ p # matrix mult. (N,N)@(N,D) = (N,D)
```

PyTorch, in practice

```
# Re-indexing:
q_i  = q[:,None,:] # shape (N,D) -> (N,1,D)
q_j  = q[None,:,:] # shape (N,D) -> (1,N,D)
# Actual computations:
sqd  = torch.sum( (q_i - q_j)**2 , 2 ) # |q_i-q_j|^2
K_qq = torch.exp( - sqd / s**2 )       # Gaussian kernel
v    = K_qq @ p # matrix mult. (N,N)@(N,D) = (N,D)

# Output the kernel norm H(q,p): .5*<p,v>
H    = .5 * torch.dot( p.view(-1), v.view(-1) )
```

PyTorch, in practice

```
# Re-indexing:
q_i  = q[:,None,:] # shape (N,D) -> (N,1,D)
q_j  = q[None,:,:] # shape (N,D) -> (1,N,D)
# Actual computations:
sqd  = torch.sum( (q_i - q_j)**2 , 2 ) # |q_i-q_j|^2
K_qq = torch.exp( - sqd / s**2 )       # Gaussian kernel
v    = K_qq @ p # matrix mult. (N,N)@(N,D) = (N,D)

# Output the kernel norm H(q,p): .5*<p,v>
H    = .5 * torch.dot( p.view(-1), v.view(-1) )
```

```
H = 6029309.0
Elapsed time: 0.31s
```

PyTorch, in practice

```
# Re-indexing:
q_i  = q[:,None,:] # shape (N,D) -> (N,1,D)
q_j  = q[None,:,:] # shape (N,D) -> (1,N,D)
# Actual computations:
sqd  = torch.sum( (q_i - q_j)**2 , 2 ) # |q_i-q_j|^2
K_qq = torch.exp( - sqd / s**2 )       # Gaussian kernel
v    = K_qq @ p # matrix mult. (N,N)@(N,D) = (N,D)

# Output the kernel norm H(q,p): .5*<p,v>
H    = .5 * torch.dot( p.view(-1), v.view(-1) )
```

```
H = 6029309.0
Elapsed time: 0.31s
```

```
# Automatic differentiation is straightforward:
[dq,dp] = grad( H, [q,p] )
```

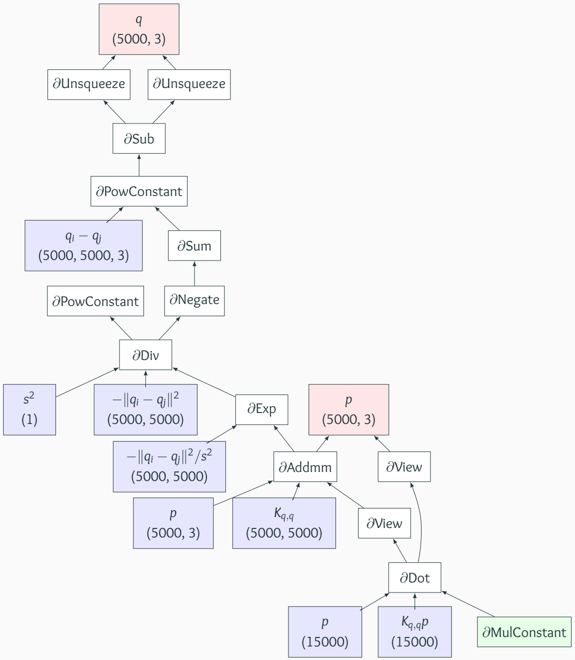
PyTorch, in practice

```
# Re-indexing:
q_i  = q[:,None,:] # shape (N,D) -> (N,1,D)
q_j  = q[None,:,:] # shape (N,D) -> (1,N,D)
# Actual computations:
sqd  = torch.sum( (q_i - q_j)**2 , 2 ) # |q_i-q_j|^2
K_qq = torch.exp( - sqd / s**2 )       # Gaussian kernel
v    = K_qq @ p # matrix mult. (N,N)@(N,D) = (N,D)

# Output the kernel norm H(q,p): .5*<p,v>
H    = .5 * torch.dot( p.view(-1), v.view(-1) )
```

```
H = 6029309.0
Elapsed time: 0.31s
```

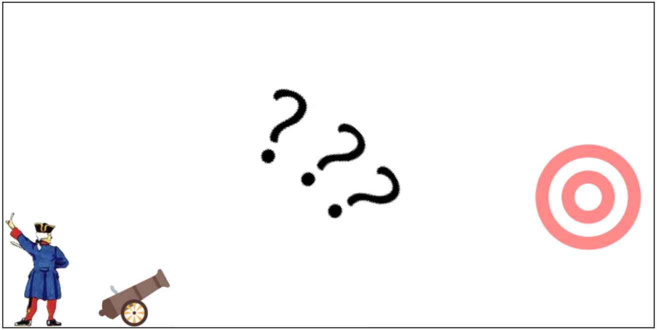
```
# Automatic differentiation is straightforward:
[dq,dp] = grad( H, [q,p] )
```

```
dq.shape = q.shape ; dp.shape = p.shape
Elapsed time: 0.03s
```


q
(5000, 3)
∂Unsqueeze
∂Unsqueeze
∂Sub
∂PowConstant
$q_i - q_j$
(5000, 5000, 3)
∂Sum
∂PowConstant
∂Negate
∂Div
s^2
(1)
$-\|q_i - q_j\|^2$
(5000, 5000)
∂Exp
p
(5000, 3)
$-\|q_i - q_j\|^2/s^2$
(5000, 5000)
∂Addmm
∂View
p
(5000, 3)
$K_{q,q}$
(5000, 5000)
∂View
∂Dot
p
(15000)
$K_{q,q}p$
(15000)
∂MulConstant

Using PyTorch for Optimal Control

Ballistic 101



Ballistic 101

Take two locations in the plane $\mathbb{R}^2$:

$$x_0 = \begin{pmatrix} 0 \\ .5 \end{pmatrix} \qquad \text{and} \qquad \widetilde{x} = \begin{pmatrix} 7 \\ 2 \end{pmatrix}.$$

Ballistic 101

Take two locations in the plane $\mathbb{R}^2$:

$$x_0 = \begin{pmatrix} 0 \\ .5 \end{pmatrix} \qquad \text{and} \qquad \widetilde{x} = \begin{pmatrix} 7 \\ 2 \end{pmatrix}.$$

Assume that the trajectory x_t follows Newton's laws of motion:

$$\ddot{x}_t = \begin{pmatrix} 0 \\ -g \end{pmatrix}.$$

Ballistic 101

Take two locations in the plane $\mathbb{R}^2$:

$$x_0 = \begin{pmatrix} 0 \\ .5 \end{pmatrix} \qquad \text{and} \qquad \widetilde{x} = \begin{pmatrix} 7 \\ 2 \end{pmatrix}.$$

Assume that the trajectory x_t follows Newton's laws of motion:

$$\ddot{x}_t = \begin{pmatrix} 0 \\ -g \end{pmatrix}.$$

Optimal Control problem: find a momentum $P \in \mathbb{R}^2$ such that

$$m\,\dot{x}_0 = P \quad \Longrightarrow \quad x_1 \simeq \widetilde{x}.$$

PyTorch allows you to work with the proper equations!

Using the position-momentum coordinates

$$q_t = x_t, \qquad p_t = m\,v_t,$$

PyTorch allows you to work with the proper equations!

Using the position-momentum coordinates

$$q_t \;=\; x_t, \qquad p_t \;=\; m\,v_t,$$

write down the expression of the mechanical energy

$$E_{\text{mec}}(x, v) \;=\; mg \cdot x[2] \;+\; \tfrac{1}{2}m\,\|v\|^2,$$
$$E_{\text{mec}}(q, p) \;=\; mg \cdot q[2] \;+\; \tfrac{1}{2m}\,\|p\|^2.$$

PyTorch allows you to work with the proper equations!

Using the position-momentum coordinates

$$q_t = x_t, \qquad p_t = m\,v_t,$$

write down the expression of the mechanical energy

$$E_{\text{mec}}(x, v) = mg \cdot x[2] + \tfrac{1}{2} m \, \|v\|^2,$$
$$E_{\text{mec}}(q, p) = mg \cdot q[2] + \tfrac{1}{2m} \, \|p\|^2.$$

Then (Hamilton, 1833; Pontryagin, 1956):

$$\begin{cases} \dot{q}_t = v_t = +\tfrac{1}{m} p_t = +\tfrac{\partial E_{\text{mec}}}{\partial p}(q_t, p_t) \\ \dot{p}_t = m\,\dot{v}_t = (0, -mg) = -\tfrac{\partial E_{\text{mec}}}{\partial q}(q_t, p_t) \end{cases}$$

Setting the parameters of our model

```python
import torch                # GPU + autodiff library
from   torch                import Tensor
from   torch.autograd import grad
```

Setting the parameters of our model

```
import torch            # GPU + autodiff library
from   torch            import Tensor
from   torch.autograd import grad

# Set the parameters of our model:
g      = Tensor( [ 9.81], requires_grad = True )
m      = Tensor( [ 15. ], requires_grad = True )
source = Tensor( [0.,.5], requires_grad = True )
target = Tensor( [7.,2.], requires_grad = True )
```

Defining a cost to optimize

```
def cost(m, g, P) :
  "Cost associated to a simple ballistic problem."
  def Emec(q,p) :
    "Particle of mass m in a gravitational field g."
    return m*g*q[1] + (p**2).sum() / (2*m)
```

Defining a cost to optimize

```
def cost(m, g, P) :
  "Cost associated to a simple ballistic problem."
  def Emec(q,p) :
    "Particle of mass m in a gravitational field g."
    return m*g*q[1] + (p**2).sum() / (2*m)

  # Initial condition:
  qt = source ; pt = P
```

Defining a cost to optimize

```
def cost(m, g, P) :
  "Cost associated to a simple ballistic problem."
  def Emec(q,p) :
    "Particle of mass m in a gravitational field g."
    return m*g*q[1] + (p**2).sum() / (2*m)

  # Initial condition:
  qt = source ; pt = P

  # Simple Euler scheme:
  for it in range(10) :
    [dq,dp] = grad(Emec(qt,pt), [qt,pt], create_graph=True)
```

Defining a cost to optimize

```python
def cost(m, g, P) :
  "Cost associated to a simple ballistic problem."
  def Emec(q,p) :
    "Particle of mass m in a gravitational field g."
    return m*g*q[1] + (p**2).sum() / (2*m)

  # Initial condition:
  qt = source ; pt = P

  # Simple Euler scheme:
  for it in range(10) :
    [dq,dp] = grad(Emec(qt,pt), [qt,pt], create_graph=True)
    qt = qt + .1 * dp
    pt = pt - .1 * dq
```

Defining a cost to optimize

```python
def cost(m, g, P) :
  "Cost associated to a simple ballistic problem."
  def Emec(q,p) :
    "Particle of mass m in a gravitational field g."
    return m*g*q[1] + (p**2).sum() / (2*m)

  # Initial condition:
  qt = source ; pt = P

  # Simple Euler scheme:
  for it in range(10) :
    [dq,dp] = grad(Emec(qt,pt), [qt,pt], create_graph=True)
    qt = qt + .1 * dp
    pt = pt - .1 * dq

  # Return the squared distance to the target:
  return ((qt - target)**2).sum()
```

Solving the control problem through gradient descent

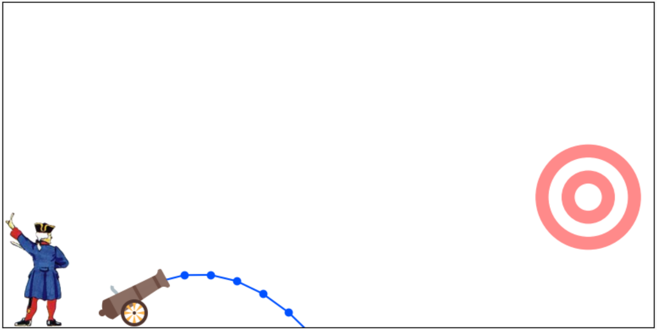
```
P = Tensor( [60., 30.], requires_grad = True )
```

Solving the control problem through gradient descent

```
P = Tensor( [60., 30.], requires_grad = True )
lr = 5.
for it in range(100) :
  [dP] = grad( cost(m,g,P), [P])
  P.data -= lr * dP.data
```

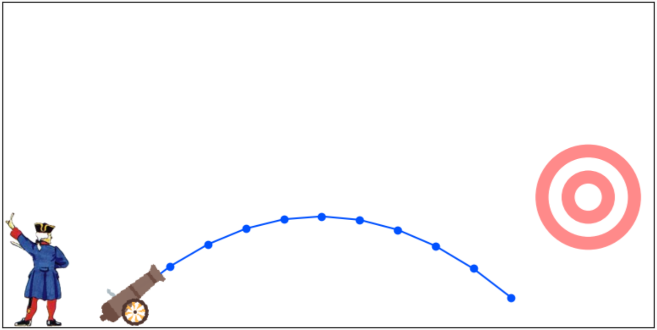
Solving the control problem through gradient descent

```
P = Tensor( [60., 30.], requires_grad = True )
lr = 5.
for it in range(100) :
  [dP] = grad( cost(m,g,P), [P])
  P.data -= lr * dP.data
```



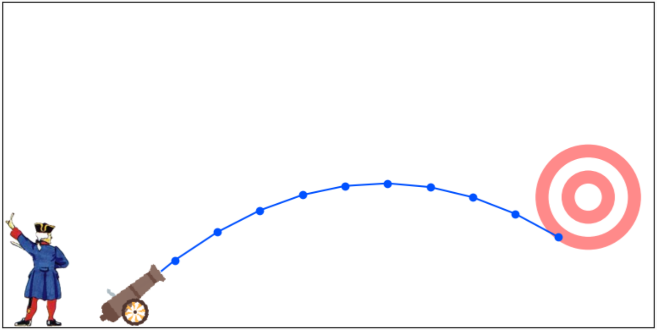
Solving the control problem through gradient descent

```
P = Tensor( [60., 30.], requires_grad = True )
lr = 5.
for it in range(100) :
  [dP] = grad( cost(m,g,P), [P])
  P.data -= lr * dP.data
```



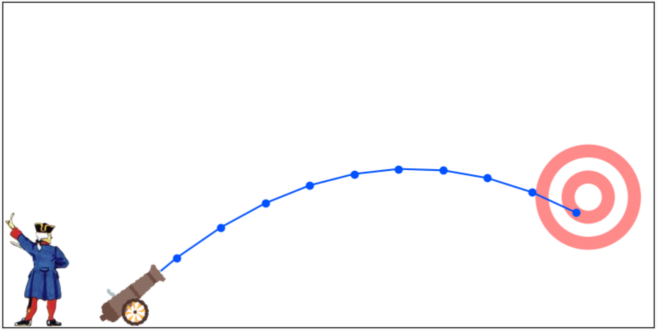
Solving the control problem through gradient descent

```
P = Tensor( [60., 30.], requires_grad = True )
lr = 5.
for it in range(100) :
  [dP] = grad( cost(m,g,P), [P])
  P.data -= lr * dP.data
```



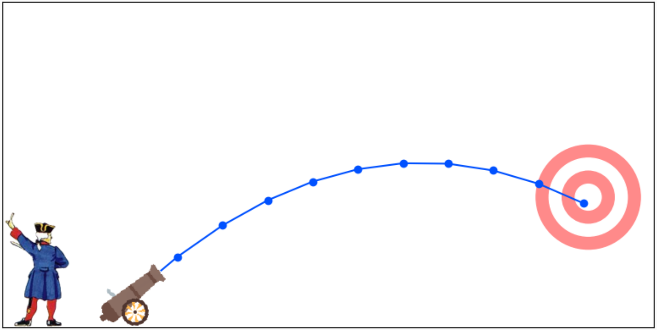
Solving the control problem through gradient descent

```
P = Tensor( [60., 30.], requires_grad = True )
lr = 5.
for it in range(100) :
  [dP] = grad( cost(m,g,P), [P])
  P.data -= lr * dP.data
```



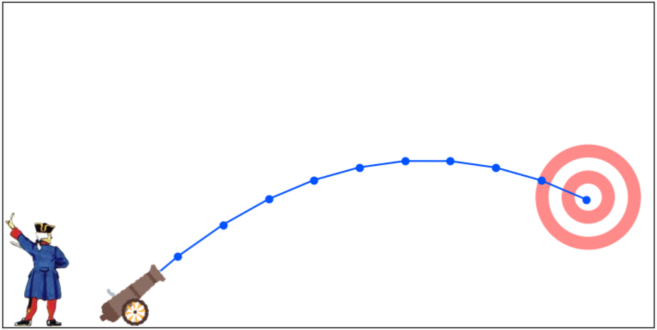
Solving the control problem through gradient descent

```
P = Tensor( [60., 30.], requires_grad = True )
lr = 5.
for it in range(100) :
  [dP] = grad( cost(m,g,P), [P])
  P.data -= lr * dP.data
```



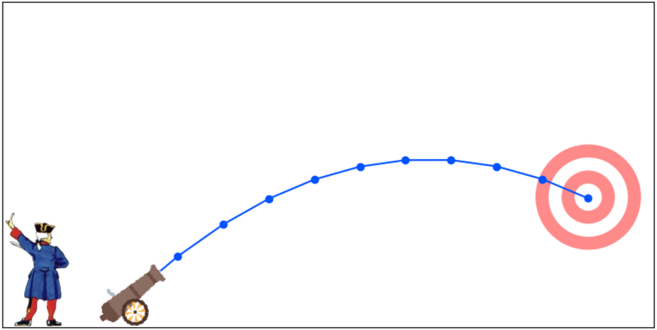
Solving the control problem through gradient descent

```
P = Tensor( [60., 30.], requires_grad = True )
lr = 5.
for it in range(100) :
  [dP] = grad( cost(m,g,P), [P])
  P.data -= lr * dP.data
```



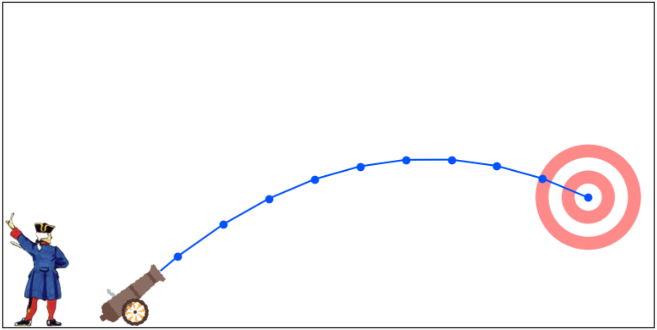
Solving the control problem through gradient descent

```
P = Tensor( [60., 30.], requires_grad = True )
lr = 5.
for it in range(100) :
  [dP] = grad( cost(m,g,P), [P])
  P.data -= lr * dP.data
```



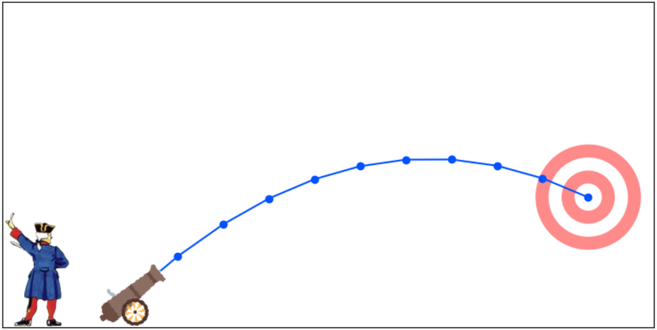
Solving the control problem through gradient descent

```
P = Tensor( [60., 30.], requires_grad = True )
lr = 5.
for it in range(100) :
  [dP] = grad( cost(m,g,P), [P])
  P.data -= lr * dP.data
```



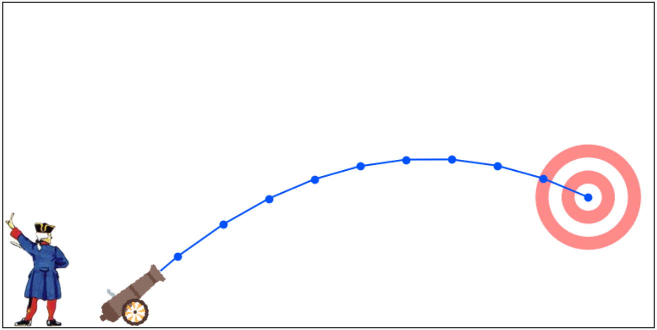
Solving the control problem through gradient descent

```
P = Tensor( [60., 30.], requires_grad = True )
lr = 5.
for it in range(100) :
  [dP] = grad( cost(m,g,P), [P])
  P.data -= lr * dP.data
```



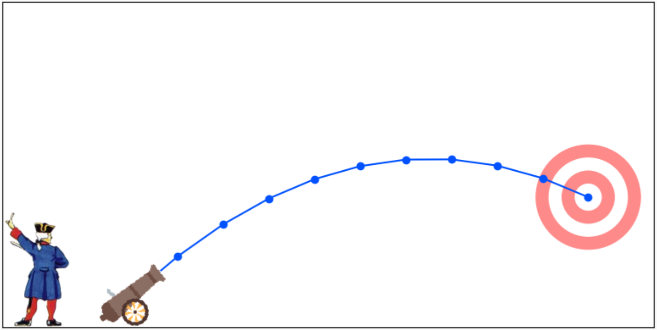
Solving the control problem through gradient descent

```
P = Tensor( [60., 30.], requires_grad = True )
lr = 5.
for it in range(100) :
  [dP] = grad( cost(m,g,P), [P])
  P.data -= lr * dP.data
```



Solving the control problem through gradient descent

```
P = Tensor( [60., 30.], requires_grad = True )
lr = 5.
for it in range(100) :
  [dP] = grad( cost(m,g,P), [P])
  P.data -= lr * dP.data
```



Putting randomness into our model

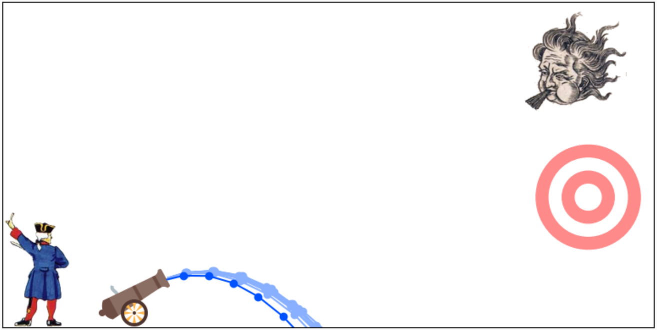
```python
def cost(m, g, P) :
  "Cost associated to a simple ballistic problem."
  def Emec(q,p) :
    "Particle of mass m in a gravitational field g."
    return m*g*q[1] + (p**2).sum() / (2*m)

  # Initial condition:
  qt = source ; pt = P
  # Simple Euler scheme:
  for it in range(10) :
    [dq,dp] = grad(Emec(qt,pt), [qt,pt], create_graph=True)
    dq += qt[1] * 20 * torch.randn(2)
    qt = qt + .1 * dp
    pt = pt - .1 * dq

  # Return the squared distance to the target:
  return ((qt - target)**2).sum()
```

Optimizing a noisy command

```
P = Tensor( [60., 30.], requires_grad = True )
lr = 5.
for it in range(100) :
  [dP] = grad( cost(m,g,P), [P])
  P.data -= lr * dP.data
```

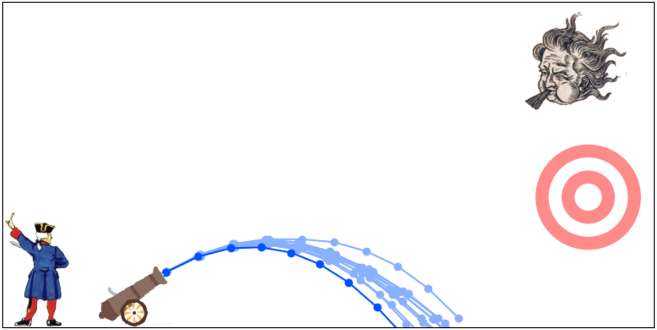
Optimizing a noisy command

```
P = Tensor( [60., 30.], requires_grad = True )
lr = 5.
for it in range(100) :
  [dP] = grad( cost(m,g,P), [P])
  P.data -= lr * dP.data
```



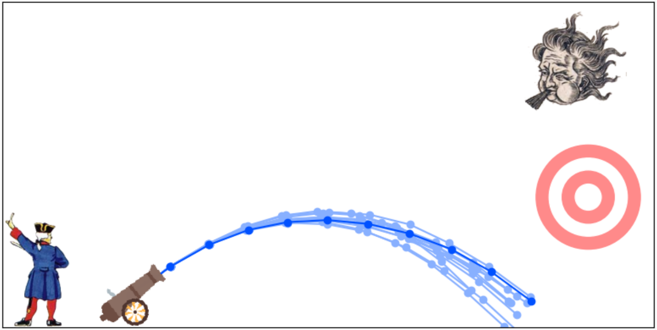
Optimizing a noisy command

```
P = Tensor( [60., 30.], requires_grad = True )
lr = 5.
for it in range(100) :
  [dP] = grad( cost(m,g,P), [P])
  P.data -= lr * dP.data
```



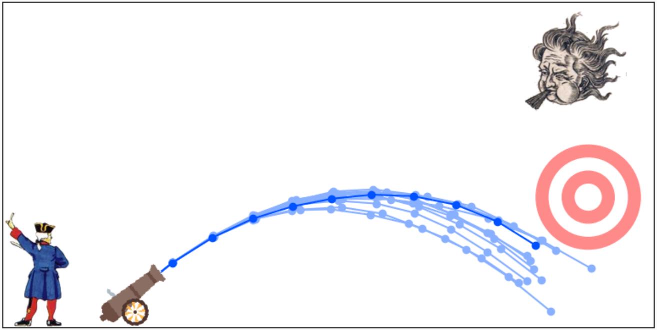
Optimizing a noisy command

```
P = Tensor( [60., 30.], requires_grad = True )
lr = 5.
for it in range(100) :
  [dP] = grad( cost(m,g,P), [P])
  P.data -= lr * dP.data
```



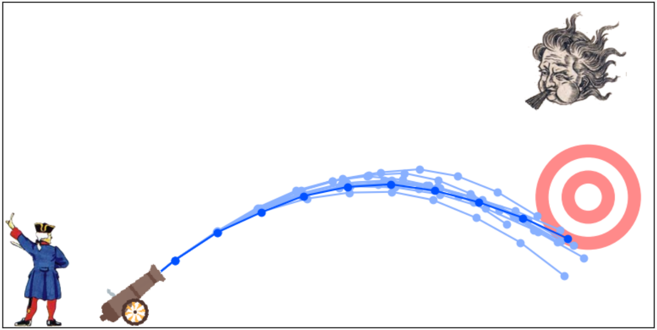
Optimizing a noisy command

```
P = Tensor( [60., 30.], requires_grad = True )
lr = 5.
for it in range(100) :
  [dP] = grad( cost(m,g,P), [P])
  P.data -= lr * dP.data
```



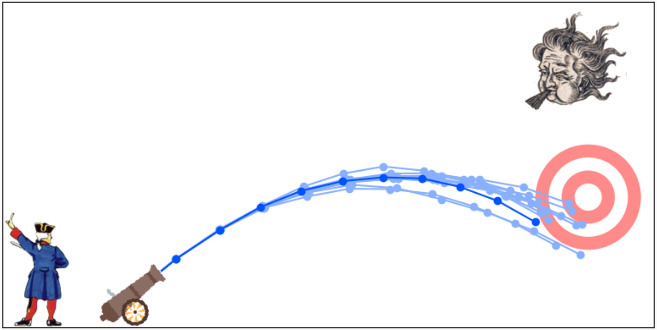
Optimizing a noisy command

```
P = Tensor( [60., 30.], requires_grad = True )
lr = 5.
for it in range(100) :
  [dP] = grad( cost(m,g,P), [P])
  P.data -= lr * dP.data
```



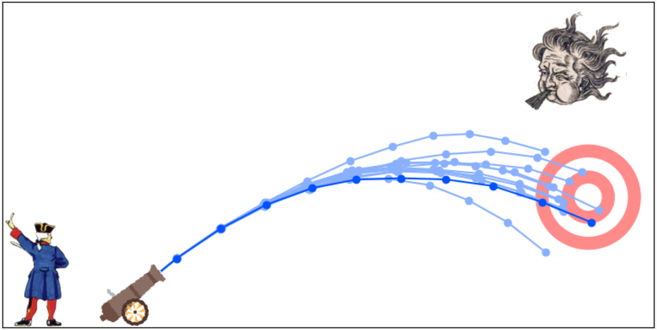
Optimizing a noisy command

```
P = Tensor( [60., 30.], requires_grad = True )
lr = 5.
for it in range(100) :
  [dP] = grad( cost(m,g,P), [P])
  P.data -= lr * dP.data
```



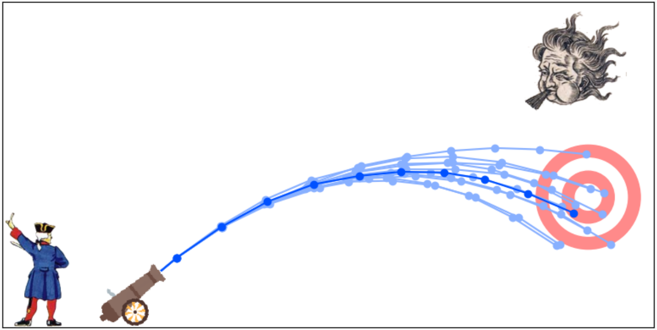
Optimizing a noisy command

```
P = Tensor( [60., 30.], requires_grad = True )
lr = 5.
for it in range(100) :
  [dP] = grad( cost(m,g,P), [P])
  P.data -= lr * dP.data
```



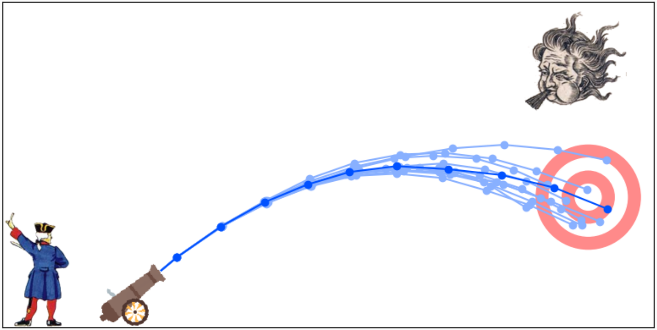
Optimizing a noisy command

```
P = Tensor( [60., 30.], requires_grad = True )
lr = 5.
for it in range(100) :
  [dP] = grad( cost(m,g,P), [P])
  P.data -= lr * dP.data
```



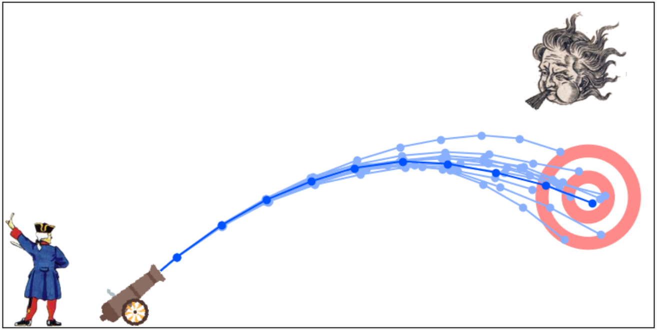
Optimizing a noisy command

```
P = Tensor( [60., 30.], requires_grad = True )
lr = 5.
for it in range(100) :
  [dP] = grad( cost(m,g,P), [P])
  P.data -= lr * dP.data
```



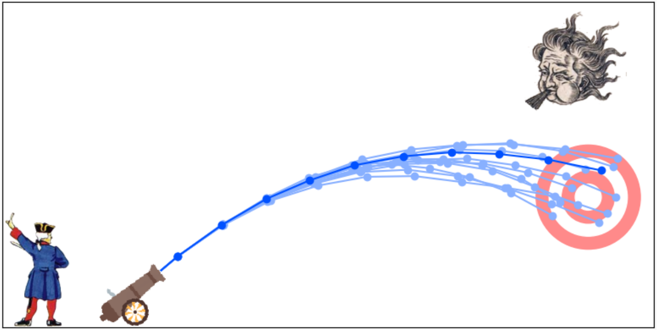
Optimizing a noisy command

```
P = Tensor( [60., 30.], requires_grad = True )
lr = 5.
for it in range(100) :
  [dP] = grad( cost(m,g,P), [P])
  P.data -= lr * dP.data
```



Optimizing a noisy command

```
P = Tensor( [60., 30.], requires_grad = True )
lr = 5.
for it in range(100) :
  [dP] = grad( cost(m,g,P), [P])
  P.data -= lr * dP.data
```

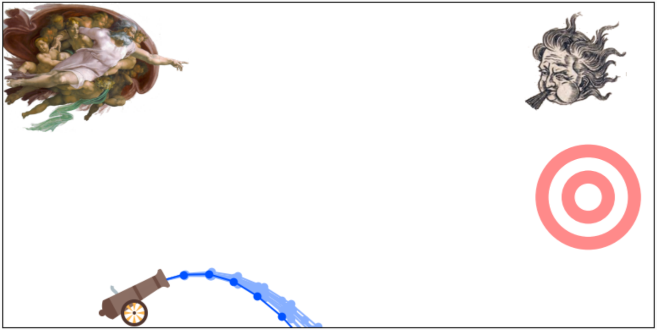


Optimizing wrt. the gravitational field

```
g = Tensor( [ 9.81], requires_grad = True )
lr = .1
for it in range(100) :
  [dg] = grad( cost(m,g,P), [g])
  g.data -= lr * dg.data
```

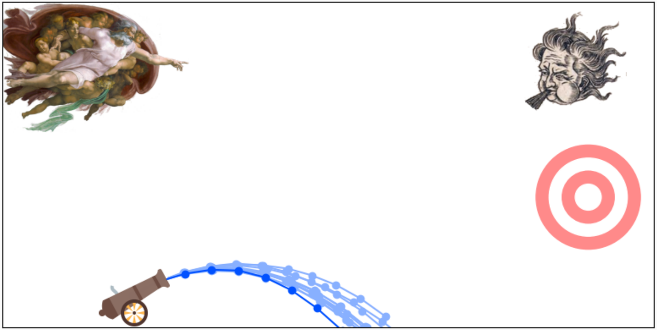
Optimizing wrt. the gravitational field

```
g = Tensor( [ 9.81], requires_grad = True )
lr = .1
for it in range(100) :
  [dg] = grad( cost(m,g,P), [g])
  g.data -= lr * dg.data
```



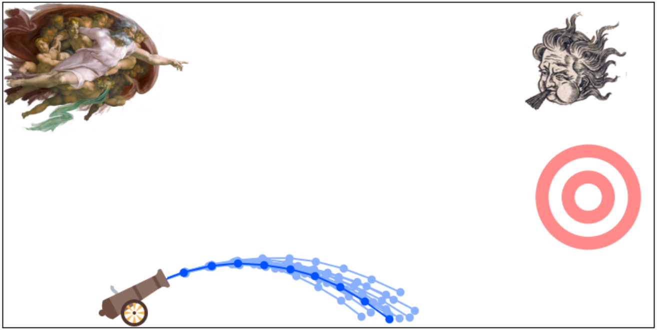
Optimizing wrt. the gravitational field

```
g = Tensor( [ 9.81], requires_grad = True )
lr = .1
for it in range(100) :
  [dg] = grad( cost(m,g,P), [g])
  g.data -= lr * dg.data
```



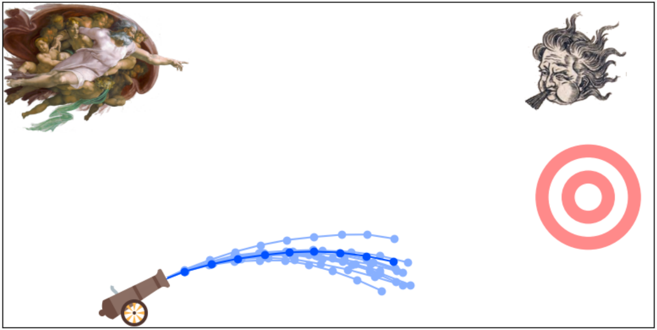
Optimizing wrt. the gravitational field

```
g = Tensor( [ 9.81], requires_grad = True )
lr = .1
for it in range(100) :
  [dg] = grad( cost(m,g,P), [g])
  g.data -= lr * dg.data
```



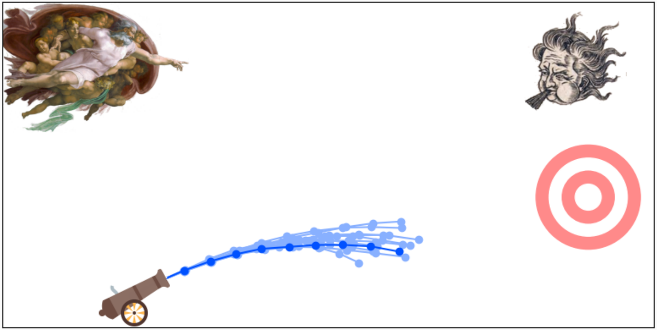
Optimizing wrt. the gravitational field

```
g = Tensor( [ 9.81], requires_grad = True )
lr = .1
for it in range(100) :
  [dg] = grad( cost(m,g,P), [g])
  g.data -= lr * dg.data
```



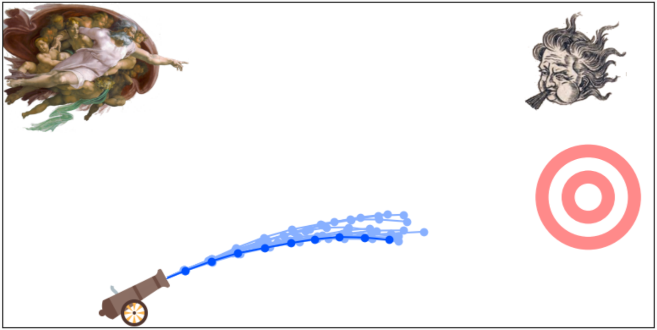
Optimizing wrt. the gravitational field

```
g = Tensor( [ 9.81], requires_grad = True )
lr = .1
for it in range(100) :
  [dg] = grad( cost(m,g,P), [g])
  g.data -= lr * dg.data
```



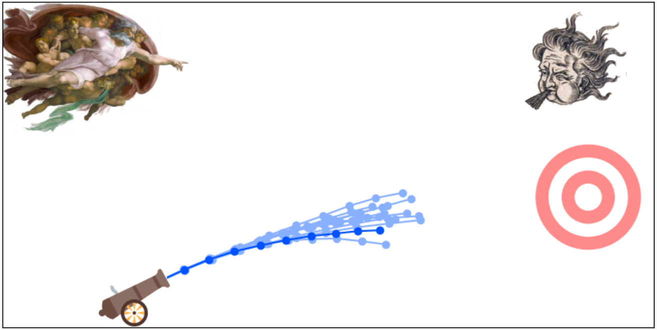
Optimizing wrt. the gravitational field

```
g = Tensor( [ 9.81], requires_grad = True )
lr = .1
for it in range(100) :
  [dg] = grad( cost(m,g,P), [g])
  g.data -= lr * dg.data
```



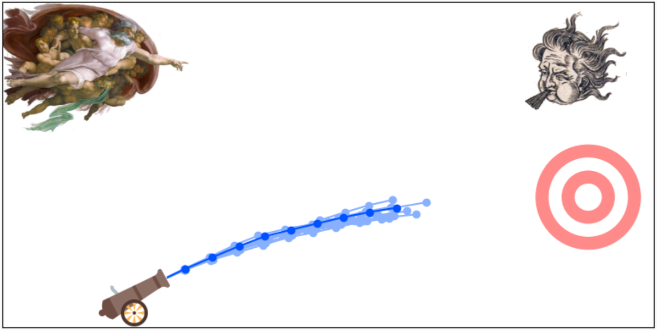
Optimizing wrt. the gravitational field

```
g = Tensor( [ 9.81], requires_grad = True )
lr = .1
for it in range(100) :
  [dg] = grad( cost(m,g,P), [g])
  g.data -= lr * dg.data
```



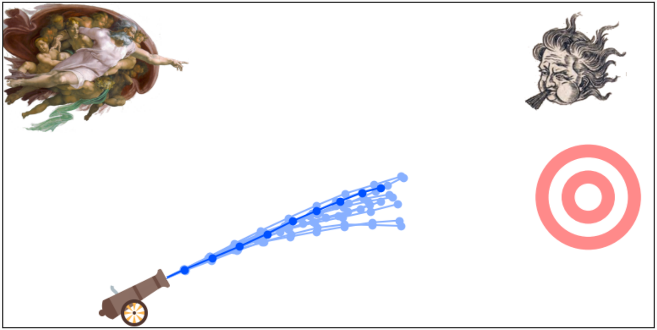
Optimizing wrt. the gravitational field

```
g = Tensor( [ 9.81], requires_grad = True )
lr = .1
for it in range(100) :
  [dg] = grad( cost(m,g,P), [g])
  g.data -= lr * dg.data
```



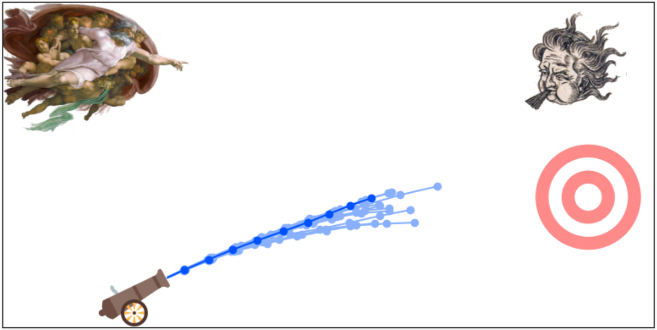
Optimizing wrt. the gravitational field

```
g = Tensor( [ 9.81], requires_grad = True )
lr = .1
for it in range(100) :
  [dg] = grad( cost(m,g,P), [g])
  g.data -= lr * dg.data
```



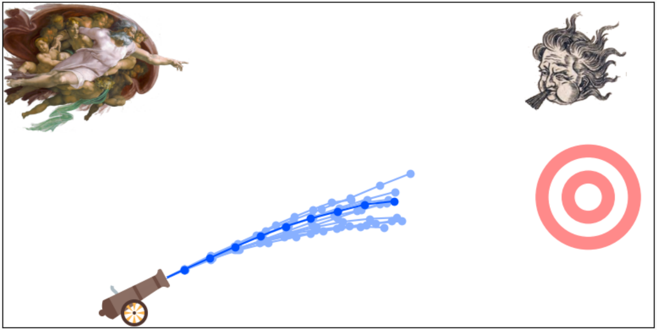
Optimizing wrt. the gravitational field

```
g = Tensor( [ 9.81], requires_grad = True )
lr = .1
for it in range(100) :
  [dg] = grad( cost(m,g,P), [g])
  g.data -= lr * dg.data
```



Optimizing wrt. the gravitational field

```
g = Tensor( [ 9.81], requires_grad = True )
lr = .1
for it in range(100) :
  [dg] = grad( cost(m,g,P), [g])
  g.data -= lr * dg.data
```



Recap on the basic usage of PyTorch

PyTorch is a simple replacement for numpy:

- Use `torch.Tensor` instead of `numpy.array`.
- Back-and-forth: `from_numpy(...)` and `.numpy()`.
- GPU backend: `.cuda()` and `.cpu()`.
- `Tensor(...)` = (data, graph history) container: check éA.data and `A.grad_fn`.
- Compute gradients with `grad(A, [...])`.
- High-order derivatives are supported!

Recap on the basic usage of PyTorch

PyTorch is a simple replacement for numpy:

- Use `torch.Tensor` instead of `numpy.array`.
- Back-and-forth: `from_numpy(...)` and `.numpy()`.
- GPU backend: `.cuda()` and `.cpu()`.
- `Tensor(...)` = (data, graph history) container: check é`A.data` and `A.grad_fn`.
- Compute gradients with `grad(A, [...])`.
- High-order derivatives are supported!

Recap on the basic usage of PyTorch

PyTorch is a simple replacement for numpy:

- Use `torch.Tensor` instead of `numpy.array`.
- Back-and-forth: `from_numpy(...)` and `.numpy()`.
- GPU backend: `.cuda()` and `.cpu()`.
- `Tensor(...)` = (data, graph history) container: check éA`.data` and `A.grad_fn`.
- Compute gradients with `grad(A, [...])`.
- High-order derivatives are supported!

Recap on the basic usage of PyTorch

PyTorch is a simple replacement for numpy:

- Use `torch.Tensor` instead of `numpy.array`.
- Back-and-forth: `from_numpy(...)` and `.numpy()`.
- GPU backend: `.cuda()` and `.cpu()`.
- `Tensor(...)` = (data, graph history) container: check é`A.data` and `A.grad_fn`.
- Compute gradients with `grad(A, [...])`.
- High-order derivatives are supported!

Recap on the basic usage of PyTorch

PyTorch is a simple replacement for numpy:

- Use `torch.Tensor` instead of `numpy.array`.
- Back-and-forth: `from_numpy(...)` and `.numpy()`.
- GPU backend: `.cuda()` and `.cpu()`.
- `Tensor(...)` = (data, graph history) container:
  check éA.data and `A.grad_fn`.
- Compute gradients with `grad(A, [...])`.
- High-order derivatives are supported!

Recap on the basic usage of PyTorch

PyTorch is a simple replacement for numpy:

- Use `torch.Tensor` instead of `numpy.array`.
- Back-and-forth: `from_numpy(...)` and `.numpy()`.
- GPU backend: `.cuda()` and `.cpu()`.
- `Tensor(...)` = (data, graph history) container: check `éA.data` and `A.grad_fn`.
- Compute gradients with `grad(A, [...])`.
- High-order derivatives are supported!

Convolutional “neural” networks: optimizing a multiscale transform

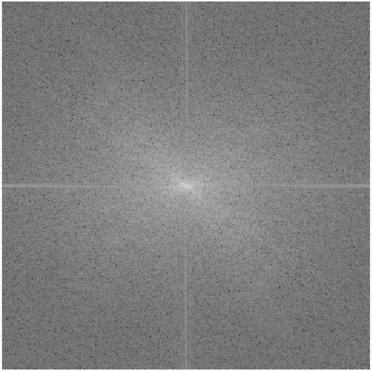
The (Discrete) Fourier Transform

Given a signal f, compute the coefficients

$$\widehat{a}(\omega) \;=\; \langle e_\omega, a \rangle_{L^2}, \qquad\qquad \text{where} \quad e_\omega : x \mapsto e^{i\omega\cdot x}.$$

Given a signal f, compute the coefficients

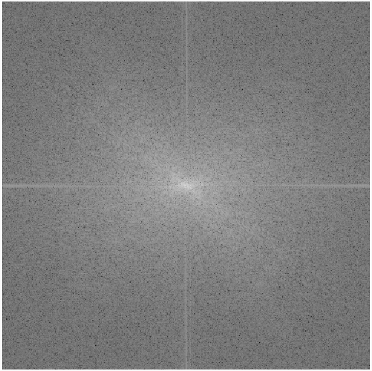
$$\widehat{a}(\omega) \;=\; \langle e_\omega, a\rangle_{L^2}, \qquad \text{where} \quad e_\omega : x \mapsto e^{i\omega\cdot x}.$$



$f(x)$ and $\log(|\widehat{f}(\omega)|)$.

The (Discrete) Fourier Transform

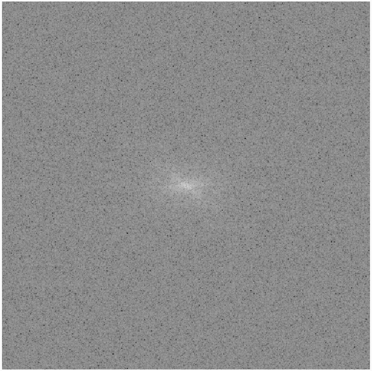
This transform allows us to apply **Gaussian blur**, unsharp filters or **Wiener denoising**.



Original image.

The (Discrete) Fourier Transform

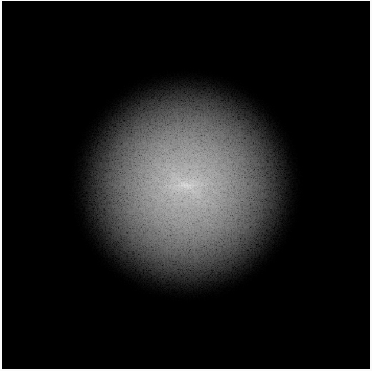
This transform allows us to apply **Gaussian blur**, unsharp filters or **Wiener denoising**.



With a Gaussian white noise.

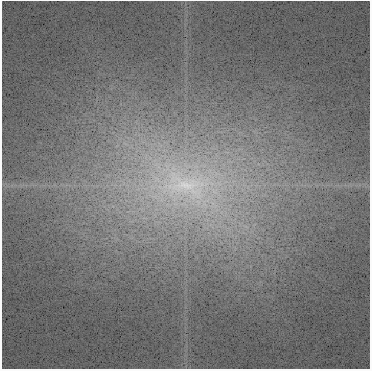
The (Discrete) Fourier Transform

This transform allows us to apply **Gaussian blur**, unsharp filters or **Wiener denoising**.



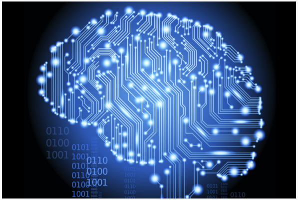
Blurred with a Gaussian filter.

This transform allows us to apply **Gaussian blur**, unsharp filters or **Wiener denoising**.



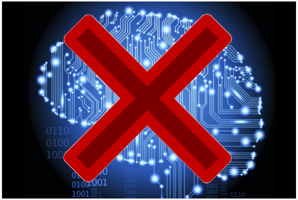
Denoised with a Wiener filter.

Going beyond linear signal processing



Super-clever algorithms...

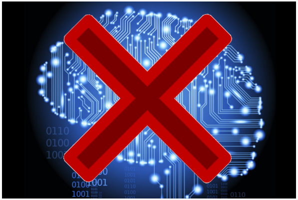
Going beyond linear signal processing



Super-clever algorithms...

Do not scale well – at all.

Going beyond linear signal processing



Super-clever algorithms...

Do not scale well – at all.

As of 2018, we can only implement **basic** algorithms on clever **representations.** We strive to find relevant mappings

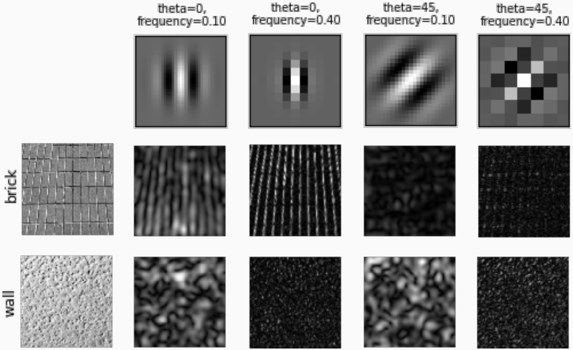
$$F: \ a \in \mathbb{R}^{W \times H} \ \mapsto \ b \in \mathbb{R}^{N}.$$

Wavelet transforms: Fourier++

Compute linear features by enforcing two **priors**:

- Features should be localized and **translation**-covariant:

$$b_{i,x,y} = (\varphi_i \star a)(x, y).$$



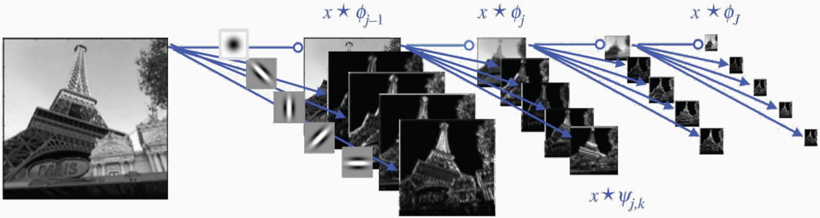
Gabor filter responses, from the scikit-learn doc.

Wavelet transforms: Fourier++

- **Multiscale** prior: features are built in cascade from finer scales,

$$b_{(i_1,\dots,i_k),\, x,y} \;=\; (\psi_{i_k} \star \cdots \star \varphi_{i_1} \star a)(x,y),$$

  with filters of (geometrically) increasing radii – this is algorithmically enforced through the **subsampling** of feature maps.



Understanding Deep Convolutional Networks (Mallat, 2016).

A real-life application: JPEG 2000

Standard format in **cinemas**:

- **Subsample** the coarse scales.
- Only store the **large** coefficients.

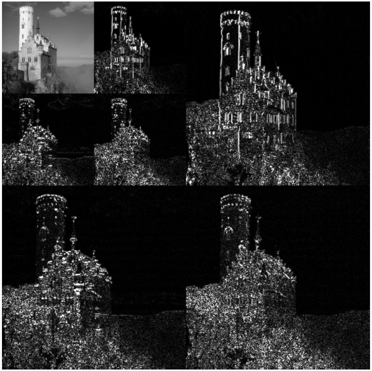


Image by Allessio Damato, from Wikipedia.

How do we choose the convolution filters?

Fast Wavelet Transform (Mallat, 1989): Given a lowpass and a highpass filter of size k, compute a multiscale decomposition of a signal of size n in $O(k \cdot n)$ operations.

How do we choose the convolution filters?

Fast Wavelet Transform (Mallat, 1989): Given a lowpass and a highpass filter of size k, compute a multiscale decomposition of a signal of size n in $O(k \cdot n)$ operations.

Daubechies filters (1992): For a given index p, there exists a pair of filters of size $2\,p$ generating an **orthogonal** transform that perfectly factorizes **locally polynomial** signals of degree $< p$.

How do we choose the convolution filters?

Fast Wavelet Transform (Mallat, 1989): Given a lowpass and a highpass filter of size k, compute a multiscale decomposition of a signal of size n in $O(k \cdot n)$ operations.

Daubechies filters (1992): For a given index p, there exists a pair of filters of size $2p$ generating an **orthogonal** transform that perfectly factorizes **locally polynomial** signals of degree $< p$.

Here is the *Db*2 pair:

	-1	0	+1	2
lowpass	-.129	.224	.837	.483
highpass	-.483	.837	-.224	-.129

Scattering Transform: |Fourier|++

We use a wavelet transform:

$$
\begin{aligned}
F_{\text{wav}}(a) \quad : \quad \mathbb{R}^{W\times H} \quad &\to \quad \mathbb{R}^{N\times W\times H} \\
a \quad &\mapsto \quad \big(\; \psi_1 \star \; \varphi_1 \star a \;\; (\cdot, \cdot), \\
& \qquad\;\; \psi_1 \star \; \varphi_2 \star a \;\; (\cdot, \cdot), \\
& \qquad\qquad\quad \cdots \qquad\qquad\quad \big)
\end{aligned}
$$

We use a scattering transform:

$$
\begin{aligned}
F_{\text{scat}}(a) \quad : \quad \mathbb{R}^{W\times H} &\to \mathbb{R}_+^{N\times W\times H} \\
a &\mapsto \big(\, |\psi_1 \star |\varphi_1 \star a||(\,\cdot\,,\,\cdot\,), \\
&\qquad |\psi_1 \star |\varphi_2 \star a||(\,\cdot\,,\,\cdot\,), \\
&\qquad \cdots \quad \big)
\end{aligned}
$$

Scattering Transform: |Fourier|++

We use scattering momenta:

$$
\begin{aligned}
F^1_{\text{scat}}(a) \quad : \quad \mathbb{R}^{W\times H} \quad &\to \quad \mathbb{R}^N_+ \\
a \quad &\mapsto \quad \big(\, \|\psi_1 \star |\varphi_1 \star a|\|_1, \\
&\qquad \|\psi_1 \star |\varphi_2 \star a|\|_1, \\
&\qquad \cdots \quad \big)
\end{aligned}
$$

Scattering Transform: |Fourier|++

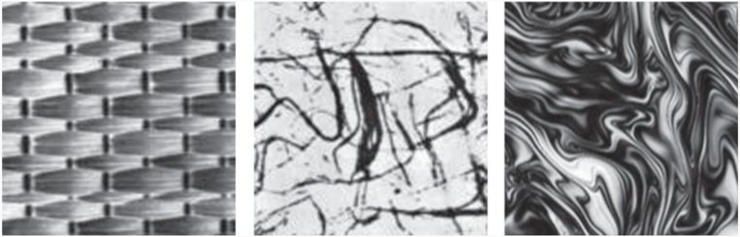
We use scattering momenta:

$$
\begin{array}{rcccc}
F^1_{\text{scat}}(a) & : & \mathbb{R}^{W\times H} & \to & \mathbb{R}^N_+ \\
 & & a & \mapsto & (\ \|\psi_1 \star |\varphi_1 \star a|\|_1, \\
 & & & & \|\psi_1 \star |\varphi_2 \star a|\|_1, \\
 & & & & \cdots \qquad)
\end{array}
$$

Texture synthesis: an optimal control problem.
Given an image Y and a transform F, find, by gradient descent from a random starting point, a synthetized image X such that

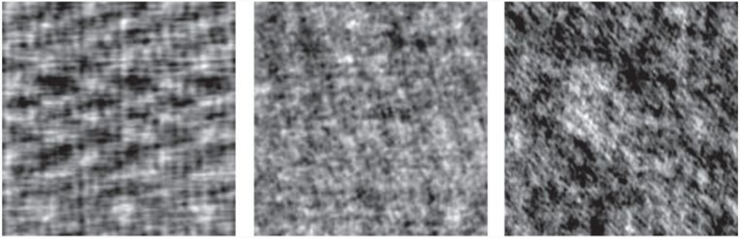
$$F(X) \simeq F(Y).$$

Using scattering momenta to characterize textured patches



Understanding Deep Convolutional Networks (Mallat, 2016).
Texture synthesis: Original patches.

Using scattering momenta to characterize textured patches



Understanding Deep Convolutional Networks (Mallat, 2016).
Texture synthesis: Synthetized from covariance momenta.

Using scattering momenta to characterize textured patches

Understanding Deep Convolutional Networks (Mallat, 2016).
Texture synthesis: Synthetized from scattering momenta.

Using scattering momenta to characterize textured patches

Understanding Deep Convolutional Networks (Mallat, 2016):
Trying to synthetize a photo using scattering momenta...

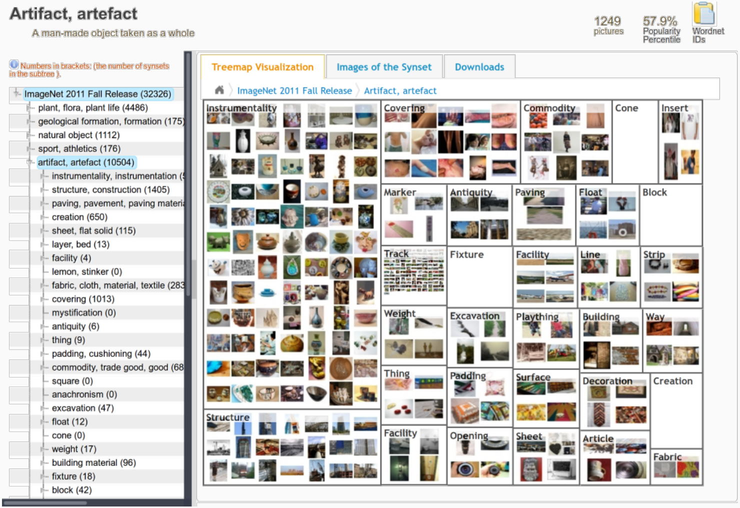
Pure maths can only take you so far.

Pure maths can only take you so far.

Thankfully, you can now go beyond explicit formulas.

Problem: classification of web-like images

ImageNet: 100,000+ classes, with 1,000+ samples per class.



Problem: classification of web-like images

Let's restrict ourselves to a subset of C classes.

The dataset is seen as a collection

$$(x_i, y_i) \in \mathbb{R}^{W\times H} \times [\![1, C]\!] \quad \simeq \quad (x_i, \delta_{y_i}) \in \mathbb{R}^{W\times H} \times [0, 1]^C,$$

Problem: classification of web-like images

Let's restrict ourselves to a subset of C classes.
The dataset is seen as a collection

$$(x_i, y_i) \in \mathbb{R}^{W\times H} \times [\![1, C]\!] \simeq (x_i, \delta_{y_i}) \in \mathbb{R}^{W\times H} \times [0,1]^C,$$

and we try to learn a **sensible classifier**

$$F_w : \mathbb{R}^{W\times H} \rightarrow [0,1]^C$$

such that for all index i,

$$F_w(x_i) \simeq \delta_{y_i}.$$

Multiscale transform $F_{\text{feat}} : \mathbb{R}^{W \times H} \to \mathbb{R}^N$ combined with a classifier

$$F_{\text{class}} : \; x \in \mathbb{R}^N \;\; \to \;\; \text{Softmax}(M_2(M_1 x)_+),$$

with M_1 an N-by-H matrix, M_2 an H-by-C matrix and

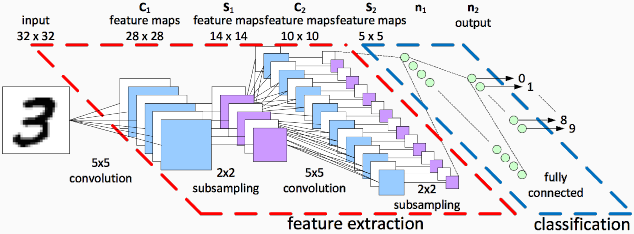
$$\text{Softmax} : \; x_i \in \mathbb{R}^C \;\; \mapsto \;\; \left(\frac{\exp(x_i)}{\sum_j \exp(x_j)} \right)_i \in [0,1]^C.$$

Multiscale transform $F_{\text{feat}} : \mathbb{R}^{W \times H} \to \mathbb{R}^N$ combined with a classifier

$$F_{\text{class}} : \; x \in \mathbb{R}^N \;\; \to \;\; \text{Softmax}(M_2(M_1 x)_+),$$

with M_1 an N-by-H matrix, M_2 an H-by-C matrix and

$$\text{Softmax} : \; x_i \in \mathbb{R}^C \;\; \mapsto \;\; \left(\frac{\exp(x_i)}{\sum_j \exp(x_j)} \right)_i \in [0,1]^C.$$



Speed sign detection and recognition by convolutional neural networks, Peeman et al. (2011).

Convolutional Neural Networks are trainable multiscale transforms

The full transform is **parameterized** by:

a set of convolution filters
+ a few matrices in the classifier
= **a large vector w.**

Convolutional Neural Networks are trainable multiscale transforms

The full transform is **parameterized** by:

a set of convolution filters
+ a few matrices in the classifier
= **a large vector *w*.**

Let's optimize these weights using stochastic gradient descent!

Convolutional Neural Networks are trainable multiscale transforms

The full transform is **parameterized** by:

a set of convolution filters

+ a few matrices in the classifier

= **a large vector w.**

Let's optimize these weights using stochastic gradient descent!

```
for it in range(1,000,000):
  I       = random set of 10 indices
  cost    = ∑_{i∈I} ||F_w(x_i) − y_i||_KL
  dw      = grad( cost, [w] )[0]
  w.data -= .001 * dw.data
```

Convolutional Neural Networks are trainable multiscale transforms

The full transform is **parameterized** by:

a set of convolution filters

\+ a few matrices in the classifier

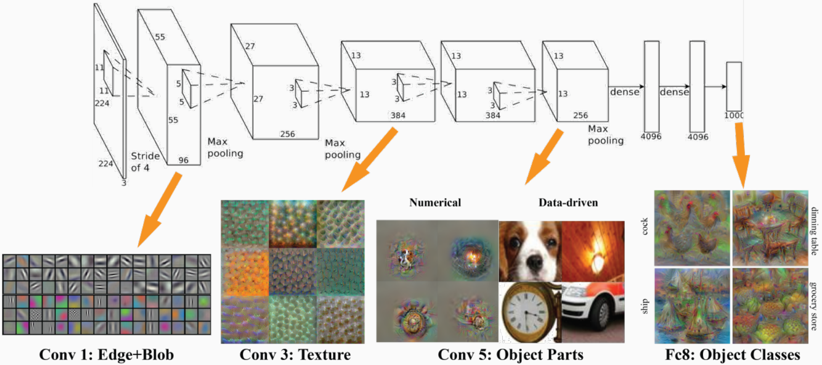
= **a large vector w.**

Let's optimize these weights using stochastic gradient descent!

```
for it in range(1,000,000):
  I        = random set of 10 indices
  cost     = ∑_{i∈I} ||F_w(x_i) − y_i||_KL
  dw       = grad( cost, [w] )[0]
  w.data -= .001 * dw.data
```

(You'd better own a good GPU!)

Convolutional Neural Networks : Texture + Structure



Hopeful CNN visualization, from `vision03.csail.mit.edu/cnn_art/`.

Convolutional Neural Networks : a good compromise

Wavelets $\simeq$ JPEG2000 :

- Super cheap.

Convolutional Neural Networks : a good compromise

Wavelets $\simeq$ JPEG2000 :

- Super cheap.
- Lot of structure.

Convolutional Neural Networks : a good compromise

Wavelets $\simeq$ JPEG2000 :

- Super cheap.
- Lot of structure.
- Encode a **multiscale** prior on images.

Convolutional Neural Networks : a good compromise

Wavelets $\simeq$ JPEG2000 :

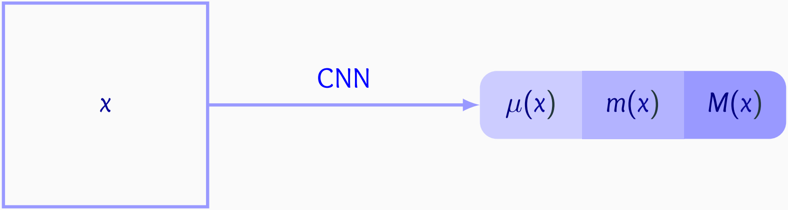
- Super cheap.
- Lot of structure.
- Encode a **multiscale** prior on images.

Convolutional Neural Networks : a good compromise

Wavelets $\simeq$ JPEG2000 :

- Super cheap.
- Lot of structure.
- Encode a **multiscale** prior on images.

By **tuning its coefficients** on a database of labeled images, we get a **CNN** $\simeq$ "JPEG 2020" that is adapted to the problem.

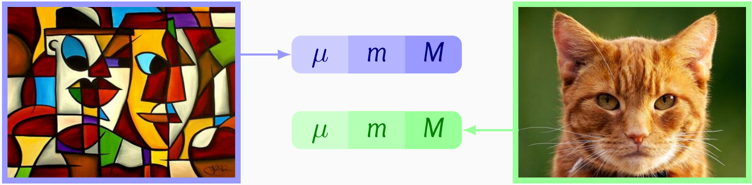


Deep Art : an emblematic application (Nikulin, Novak (2016))

Deep Art : an emblematic application (Nikulin, Novak (2016))

Deep Art : an emblematic application (Nikulin, Novak (2016))

Deep Art : an emblematic application (Nikulin, Novak (2016))

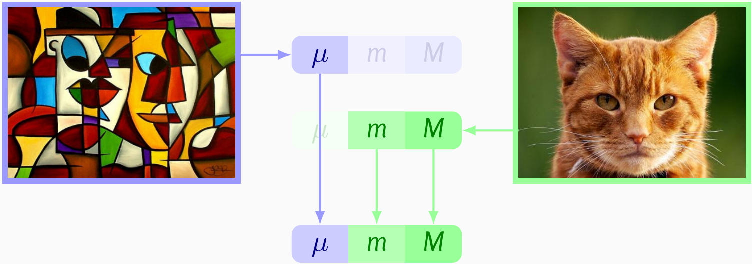


Deep Art : an emblematic application (Nikulin, Novak (2016))

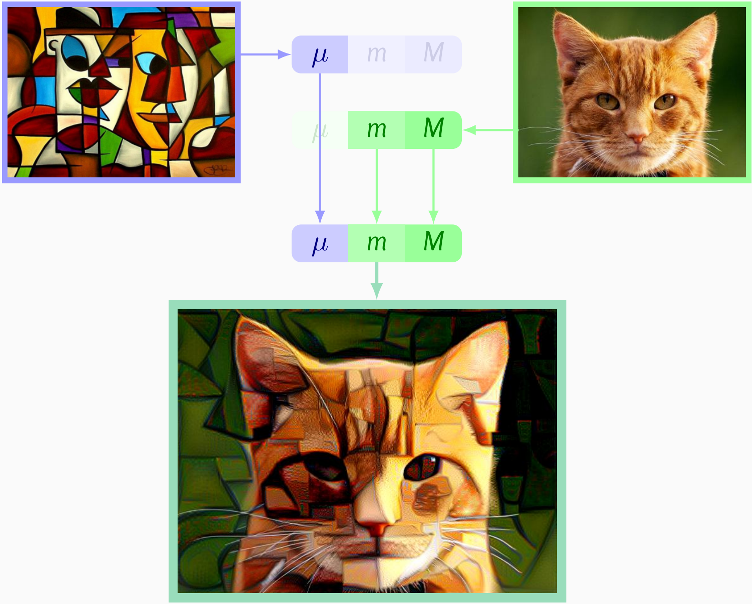
μ m M

μ m M

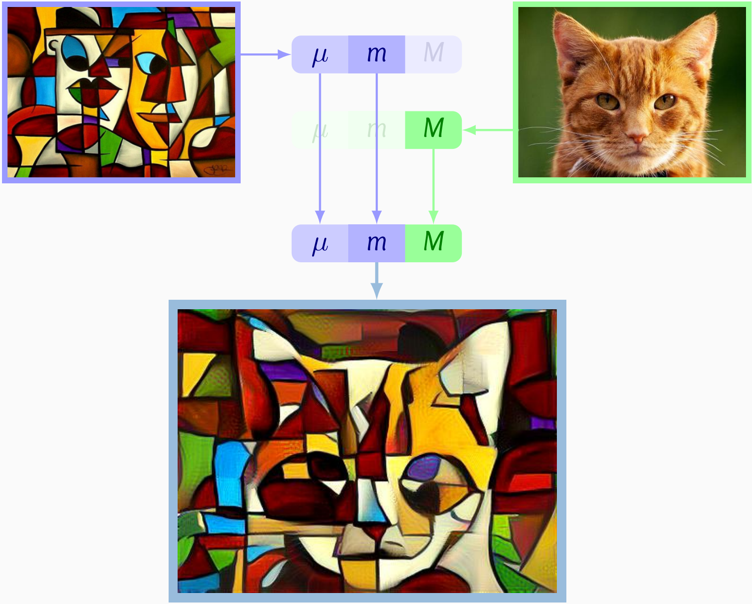
Deep Art : an emblematic application (Nikulin, Novak (2016))



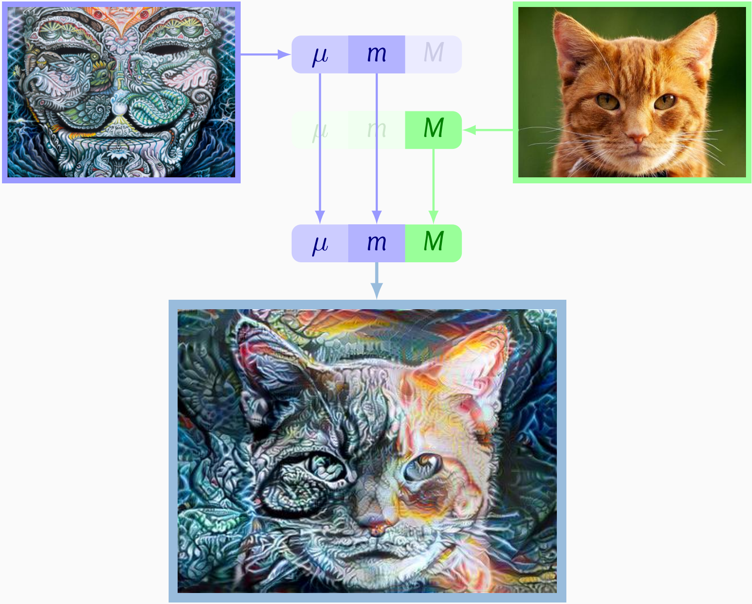
Deep Art : an emblematic application (Nikulin, Novak (2016))



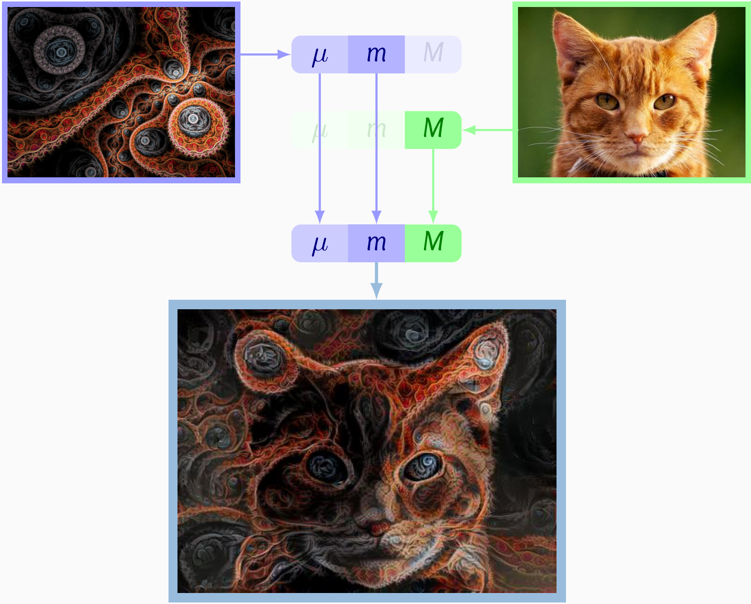
Deep Art : an emblematic application (Nikulin, Novak (2016))



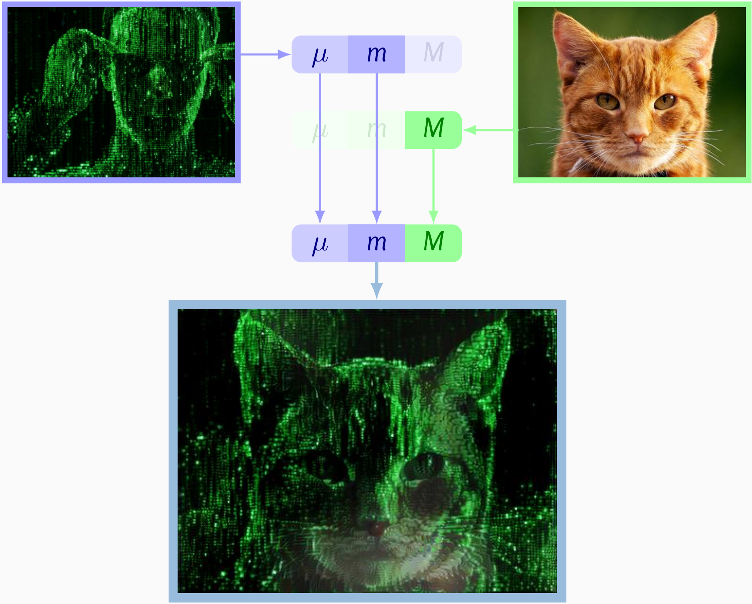
Deep Art : an emblematic application (Nikulin, Novak (2016))



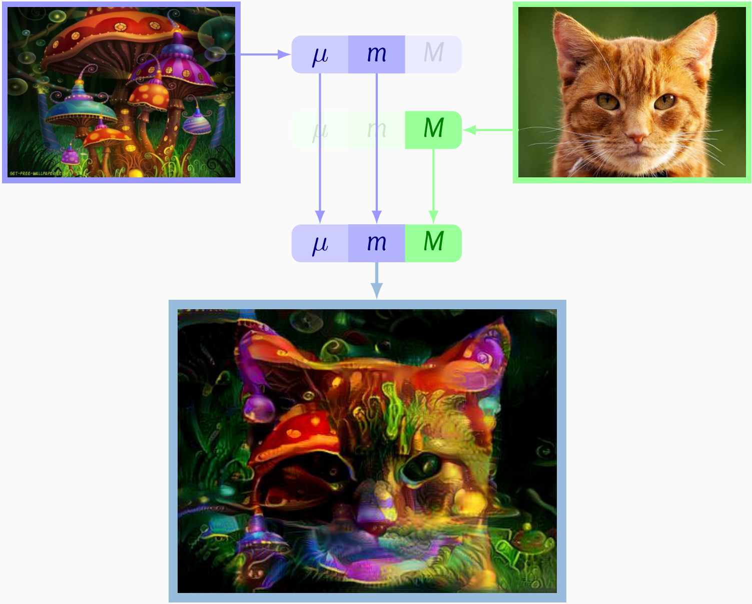
Deep Art : an emblematic application (Nikulin, Novak (2016))



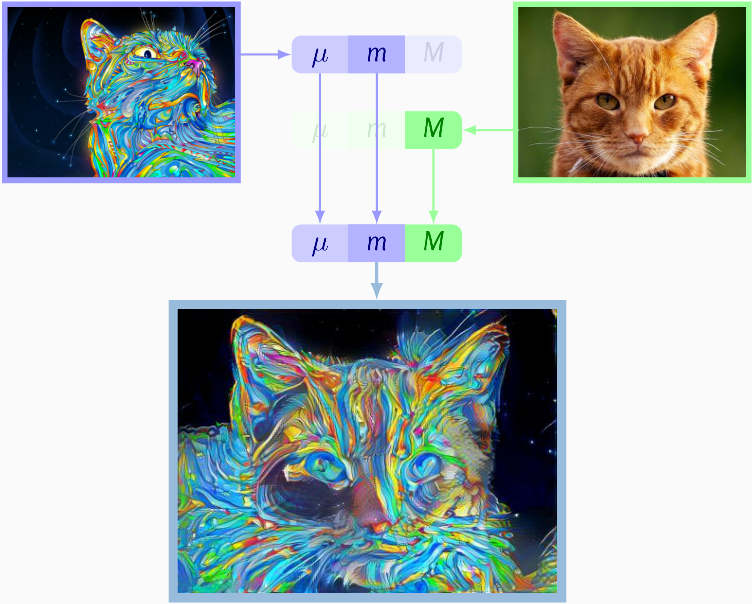
Deep Art : an emblematic application (Nikulin, Novak (2016))



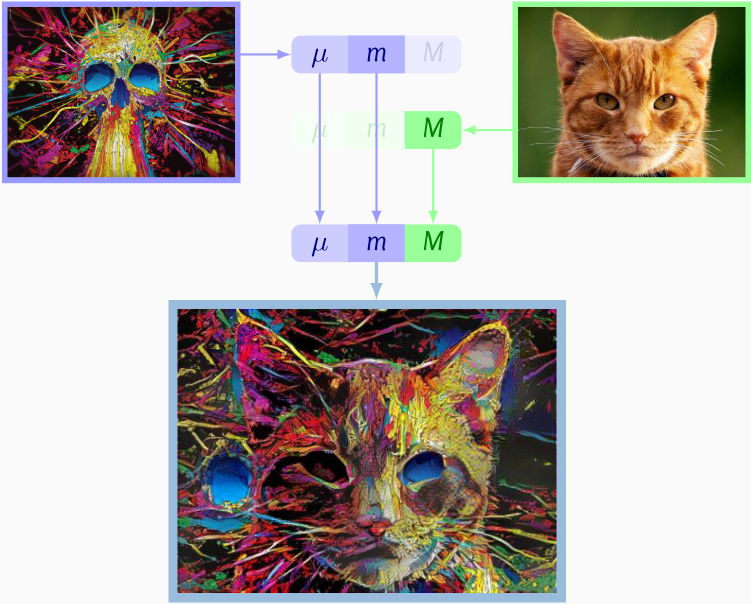
Deep Art : an emblematic application (Nikulin, Novak (2016))



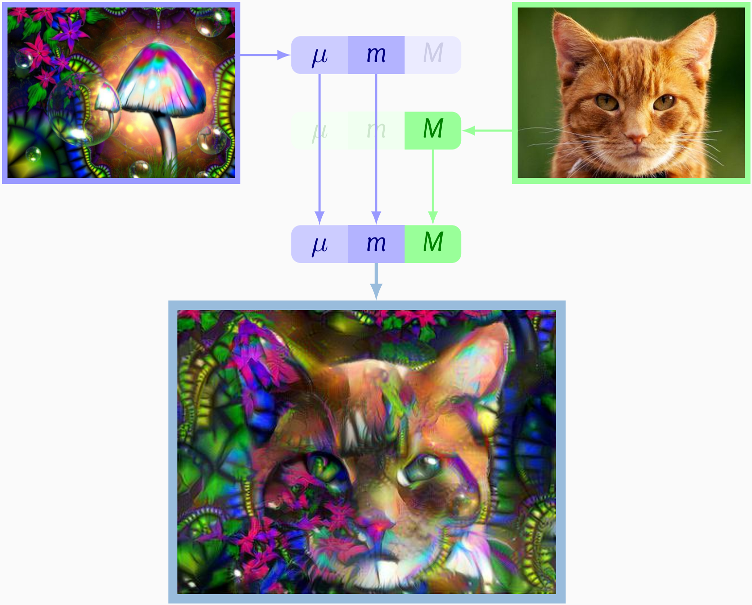
Deep Art : an emblematic application (Nikulin, Novak (2016))



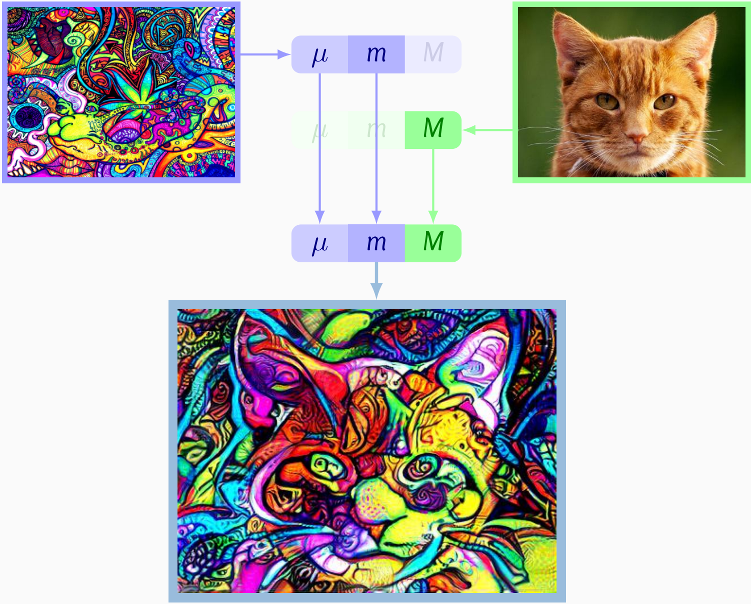
Deep Art : an emblematic application (Nikulin, Novak (2016))



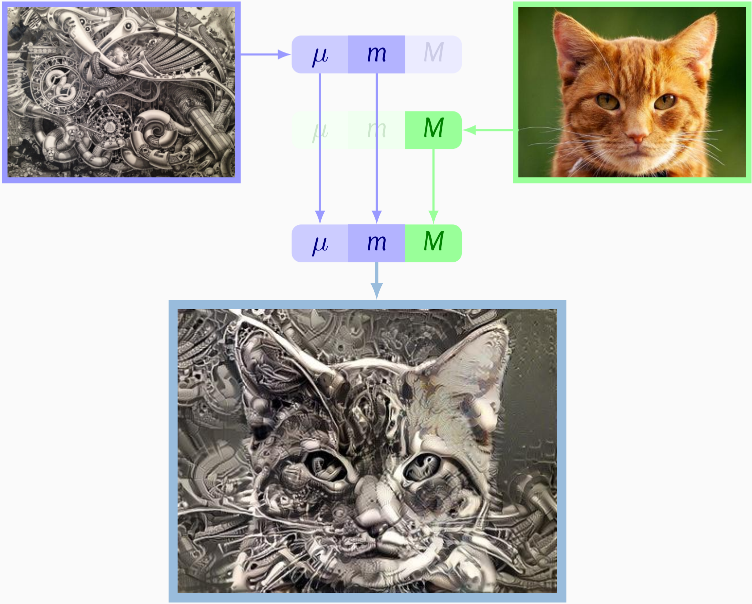
Deep Art : an emblematic application (Nikulin, Novak (2016))



Deep Art : an emblematic application (Nikulin, Novak (2016))



Deep Art : an emblematic application (Nikulin, Novak (2016))



Major autodiff frameworks in python

Major autodiff frameworks in python

- **Theano** (MILA, 2008-2017):

Major autodiff frameworks in python

- **Theano** (MILA, 2008-2017):
    - Turns a computational graph into C++ code.

Major autodiff frameworks in python

- **Theano** (MILA, 2008-2017):
    - Turns a computational graph into C++ code.
    - Relies on g++ to provide a linked executable.

Major autodiff frameworks in python

- **Theano** (MILA, 2008-2017):
    - Turns a computational graph into C++ code.
    - Relies on g++ to provide a linked executable.

- **TensorFlow** (Google, 2015-):

Major autodiff frameworks in python

- **Theano** (MILA, 2008-2017):
    - Turns a computational graph into C++ code.
    - Relies on g++ to provide a linked executable.

- **TensorFlow** (Google, 2015-):
    - Operations are pre-compiled.

Major autodiff frameworks in python

- **Theano** (MILA, 2008-2017):
    - Turns a computational graph into C++ code.
    - Relies on g++ to provide a linked executable.

- **TensorFlow** (Google, 2015-):
    - Operations are pre-compiled.
    - Strong emphasis put on large-scale deployment.

Major autodiff frameworks in python

- **Theano** (MILA, 2008-2017):
  - Turns a computational graph into C++ code.
  - Relies on g++ to provide a linked executable.
- **TensorFlow** (Google, 2015-):
  - Operations are pre-compiled.
  - Strong emphasis put on large-scale deployment.
- **PyTorch** (Facebook, 2016-):

Major autodiff frameworks in python

- **Theano** (MILA, 2008-2017):
    - Turns a computational graph into C++ code.
    - Relies on g++ to provide a linked executable.
- **TensorFlow** (Google, 2015-):
    - Operations are pre-compiled.
    - Strong emphasis put on large-scale deployment.
- **PyTorch** (Facebook, 2016-):
    - Straitghtforward numpy replacement.

Major autodiff frameworks in python

- **Theano** (MILA, 2008-2017):
  - Turns a computational graph into C++ code.
  - Relies on g++ to provide a linked executable.
- **TensorFlow** (Google, 2015-):
  - Operations are pre-compiled.
  - Strong emphasis put on large-scale deployment.
- **PyTorch** (Facebook, 2016-):
  - Straitghtforward numpy replacement.
  - Strong emphasis put on flexibility.

Extending PyTorch

Computing the Hamiltonian

```
# Actual computations.
q_i  = q.unsqueeze[:,None,:] # shape (N,D) -> (N,1,D)
q_j  = q.unsqueeze[None,:,:] # shape (N,D) -> (1,N,D)
```

Computing the Hamiltonian

```
# Actual computations.
q_i  = q.unsqueeze[:,None,:] # shape (N,D) -> (N,1,D)
q_j  = q.unsqueeze[None,:,:] # shape (N,D) -> (1,N,D)
sqd  = torch.sum( (q_i - q_j)**2 , 2 ) # |q_i-q_j|^2
```

Computing the Hamiltonian

```
# Actual computations.
q_i  = q.unsqueeze[:,None,:] # shape (N,D) -> (N,1,D)
q_j  = q.unsqueeze[None,:,:] # shape (N,D) -> (1,N,D)
sqd  = torch.sum( (q_i - q_j)**2 , 2 ) # |q_i-q_j|^2
K_qq = torch.exp( - sqd / (s**2) )     # Gaussian kernel
```

Computing the Hamiltonian

```
# Actual computations.
q_i  = q.unsqueeze[:,None,:] # shape (N,D) -> (N,1,D)
q_j  = q.unsqueeze[None,:,:] # shape (N,D) -> (1,N,D)
sqd  = torch.sum( (q_i - q_j)**2 , 2 ) # |q_i-q_j|^2
K_qq = torch.exp( - sqd / (s**2) )     # Gaussian kernel
v    = K_qq @ p # matrix mult. (N,N)@(N,D) = (N,D)
```

Computing the Hamiltonian

```
# Actual computations.
q_i  = q.unsqueeze[:,None,:] # shape (N,D) -> (N,1,D)
q_j  = q.unsqueeze[None,:,:] # shape (N,D) -> (1,N,D)
sqd  = torch.sum( (q_i - q_j)**2 , 2 ) # |q_i-q_j|^2
K_qq = torch.exp( - sqd / (s**2) )     # Gaussian kernel
v    = K_qq @ p # matrix mult. (N,N)@(N,D) = (N,D)

# Finally, compute the Hamiltonian H(q,p): .5*<p,v>
H    = .5 * torch.dot( p.view(-1), v.view(-1) )
```

Computing the Hamiltonian

```
# Actual computations.
q_i  = q.unsqueeze[:,None,:] # shape (N,D) -> (N,1,D)
q_j  = q.unsqueeze[None,:,:] # shape (N,D) -> (1,N,D)
sqd  = torch.sum( (q_i - q_j)**2 , 2 ) # |q_i-q_j|^2
K_qq = torch.exp( - sqd / (s**2) )     # Gaussian kernel
v    = K_qq @ p # matrix mult. (N,N)@(N,D) = (N,D)

# Finally, compute the Hamiltonian H(q,p): .5*<p,v>
H    = .5 * torch.dot( p.view(-1), v.view(-1) )

# Automatic differentiation is straitghtforward
[dq,dp] = grad( H, [q,p], 1.)
```

Computing the Hamiltonian

```
# Actual computations.
q_i  = q.unsqueeze[:,None,:] # shape (N,D) -> (N,1,D)
q_j  = q.unsqueeze[None,:,:] # shape (N,D) -> (1,N,D)
sqd  = torch.sum( (q_i - q_j)**2 , 2 ) # |q_i-q_j|^2
K_qq = torch.exp( - sqd / (s**2) )     # Gaussian kernel
v    = K_qq @ p # matrix mult. (N,N)@(N,D) = (N,D)

# Finally, compute the Hamiltonian H(q,p): .5*<p,v>
H    = .5 * torch.dot( p.view(-1), v.view(-1) )

# Automatic differentiation is straitghtforward
[dq,dp] = grad( H, [q,p], 1.)
```

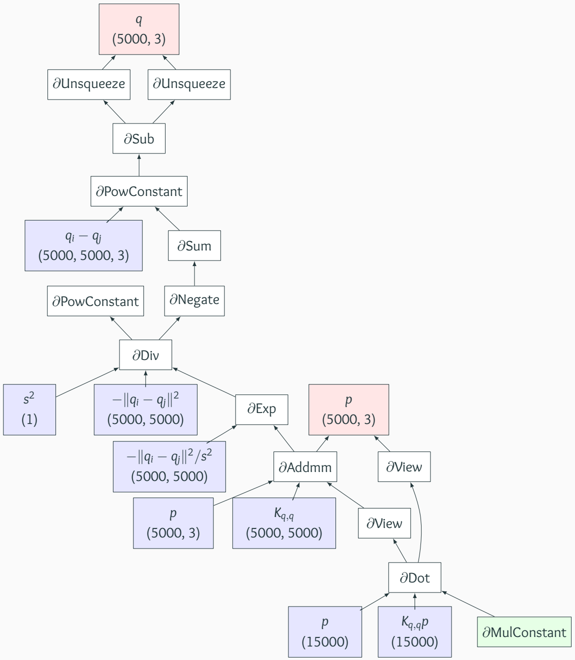
RuntimeError: cuda runtime error (2) : out of memory at /opt/conda/.../THCStorage.cu:66

Computing the Hamiltonian

```
# Actual computations.
q_i  = q.unsqueeze[:,None,:] # shape (N,D) -> (N,1,D)
q_j  = q.unsqueeze[None,:,:] # shape (N,D) -> (1,N,D)
sqd  = torch.sum( (q_i - q_j)**2 , 2 ) # |q_i-q_j|^2
K_qq = torch.exp( - sqd / (s**2) )     # Gaussian kernel
v    = K_qq @ p # matrix mult. (N,N)@(N,D) = (N,D)

# Finally, compute the Hamiltonian H(q,p): .5*<p,v>
H    = .5 * torch.dot( p.view(-1), v.view(-1) )

# Automatic differentiation is straitghtforward
[dq,dp] = grad( H, [q,p], 1.)
```

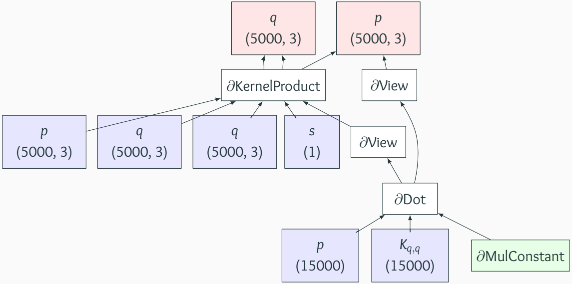
RuntimeError: cuda runtime error (2) : out of memory at /opt/conda/.../THCStorage.cu:66

```
# Display -- see next figure.
make_dot(H, {'q':q, 'p':p, 's':s}).render(view=True)
```


q
(5000, 3)
∂Unsqueeze
∂Unsqueeze
∂Sub
∂PowConstant
$q_i - q_j$
(5000, 5000, 3)
∂Sum
∂PowConstant
∂Negate
∂Div
s^2
(1)
$-\|q_i - q_j\|^2$
(5000, 5000)
∂Exp
p
(5000, 3)
$-\|q_i - q_j\|^2/s^2$
(5000, 5000)
∂Addmm
∂View
p
(5000, 3)
$K_{q,q}$
(5000, 5000)
∂View
∂Dot
p
(15000)
$K_{q,q}p$
(15000)
∂MulConstant

The KeOps library

```
# Compute the kernel convolution
v = kernelproduct(s, q, q, p, "gaussian")
# Then, compute the Hamiltonian H(q,p): .5*<p,v>
H = .5 * torch.dot( p.view(-1), v.view(-1) )
```



Define custom operators

```
class KernelProduct(torch.autograd.Function):
```

Define custom operators

```
class KernelProduct(torch.autograd.Function):

@staticmethod
def forward(ctx, s, q1, q2, p, kernel_type):
```

Define custom operators

```
class KernelProduct(torch.autograd.Function):

@staticmethod
def forward(ctx, s, q1, q2, p, kernel_type):
  # save everything to compute the gradient
  ctx.save_for_backward( s, q1, q2, p )
```

Define custom operators

```
class KernelProduct(torch.autograd.Function):

@staticmethod
def forward(ctx, s, q1, q2, p, kernel_type):
  # save everything to compute the gradient
  ctx.save_for_backward( s, q1, q2, p )
  # init gamma, the output of the convolution K_(q1,q2) @ p
  gamma  = torch.zeros(
           q1.size()[0] * p.size()[1] ).type(dtype)
```

Define custom operators

```
class KernelProduct(torch.autograd.Function):

@staticmethod
def forward(ctx, s, q1, q2, p, kernel_type):
  # save everything to compute the gradient
  ctx.save_for_backward( s, q1, q2, p )
  # init gamma, the output of the convolution K_(q1,q2) @ p
  gamma  = torch.zeros(
           q1.size()[0] * p.size()[1] ).type(dtype)
  # Inplace CUDA routine on the raw float arrays,
  # loaded from .dll/.so files by the "pybind11" module
  cudaconv.cuda_conv(   q1, q2, p, gamma, s,
                        kernel = kernel_type)
```

Define custom operators

```
class KernelProduct(torch.autograd.Function):

@staticmethod
def forward(ctx, s, q1, q2, p, kernel_type):
  # save everything to compute the gradient
  ctx.save_for_backward( s, q1, q2, p )
  # init gamma, the output of the convolution K_(q1,q2) @ p
  gamma  = torch.zeros(
           q1.size()[0] * p.size()[1] ).type(dtype)
  # Inplace CUDA routine on the raw float arrays,
  # loaded from .dll/.so files by the "pybind11" module
  cudaconv.cuda_conv(   q1, q2, p, gamma, s,
                        kernel = kernel_type)
  gamma  = gamma.view( q1.size()[0], p.size()[1] )
  return gamma
```

Define custom operators

```python
@staticmethod
def backward(ctx, a):
  (s, q1, q2, p) = ctx.saved_variables
```

Define custom operators

```python
@staticmethod
def backward(ctx, a):
  (s, q1, q2, p) = ctx.saved_variables
  # In order to get second derivatives, we encapsulated the
  # cudagradconv.cuda_gradconv routine in another
  # torch.autograd.Function object:
  kernelproductgrad_x = KernelProductGrad_x().apply
```

Define custom operators

```python
@staticmethod
def backward(ctx, a):
  (s, q1, q2, p) = ctx.saved_variables
  # In order to get second derivatives, we encapsulated the
  # cudagradconv.cuda_gradconv routine in another
  # torch.autograd.Function object:
  kernelproductgrad_x = KernelProductGrad_x().apply
  # Call the CUDA routines
  # ...
  grad_x = kernelproductgrad_x( ... )
  # ...
```

Define custom operators

```python
@staticmethod
def backward(ctx, a):
  (s, q1, q2, p) = ctx.saved_variables
  # In order to get second derivatives, we encapsulated the
  # cudagradconv.cuda_gradconv routine in another
  # torch.autograd.Function object:
  kernelproductgrad_x = KernelProductGrad_x().apply
  # Call the CUDA routines
  # ...
  grad_x = kernelproductgrad_x( ... )
  # ...
  return (grad_s, grad_q1, grad_q2, grad_p, None)
```

$\Longrightarrow$ You can do it!

KeOps:
Online Map-Reduce Operators,
with autodiff,
without memory overflows.

KeOps:
Online Map-Reduce Operators,
with autodiff,
without memory overflows.

`www.kernel-operations.io`

$\Longrightarrow$ `pip install pykeops` $\Longleftarrow$

(Thank you Benjamin!)

What we provide

For $i = 1, \ldots, N$, you want to compute:

$$a_i \;=\; \text{Reduction}_{j=1,\ldots,M} \left[F\,(\,p^1, p^2, \ldots, x_i^1, x_i^2, \ldots, y_j^1, y_j^2, \ldots) \right],$$

What we provide

For $i = 1, \ldots, N$, you want to compute:

$$a_i \;=\; \text{Reduction}_{j=1,\ldots,M} \left[F\,(\,p^1, p^2, \ldots, x_i^1, x_i^2, \ldots, y_j^1, y_j^2, \ldots) \right],$$

with :

- "**Reduction**" : Sum, Max, ArgMin, LogSumExp, ...

What we provide

For $i = 1, \ldots, N$, you want to compute:

$$a_i \;=\; \text{Reduction}_{j=1,\ldots,M} \Big[F\,(\,p^1, p^2, \ldots, x_i^1, x_i^2, \ldots, y_j^1, y_j^2, \ldots) \Big] \,,$$

with :

- "**Reduction**" : Sum, Max, ArgMin, LogSumExp, ...
- A vector-valued **formula**: F.

What we provide

For $i = 1, \ldots, N$, you want to compute:

$$a_i = \text{Reduction}_{j=1,\ldots,M} \Big[F\left(p^1, p^2, \ldots, x_i^1, x_i^2, \ldots, y_j^1, y_j^2, \ldots \right) \Big],$$

with :

- "**Reduction**" : Sum, Max, ArgMin, LogSumExp, ...
- A vector-valued **formula**: F.
- Vector **parameters**: p^1, p^2, ...

What we provide

For $i = 1, \ldots, N$, you want to compute:

$$a_i \;=\; \text{Reduction}_{j=1,\ldots,M} \Big[F\,(\,p^1, p^2, \ldots, x_i^1, x_i^2, \ldots, y_j^1, y_j^2, \ldots) \Big]\,,$$

with :

- "**Reduction**" : Sum, Max, ArgMin, LogSumExp, ...
- A vector-valued **formula**: F.
- Vector **parameters**: p^1, p^2, ...
- Vector **x-variables**, indexed by i : x_i^1, x_i^2, ...

What we provide

For $i = 1, \ldots, N$, you want to compute:

$$a_i \;=\; \text{Reduction}_{j=1,\ldots,M} \left[F\,(\,p^1, p^2, \ldots, x_i^1, x_i^2, \ldots, y_j^1, y_j^2, \ldots) \right],$$

with :

- "**Reduction**" : Sum, Max, ArgMin, LogSumExp, ...
- A vector-valued **formula**: F.
- Vector **parameters**: p^1, p^2, ...
- Vector **x-variables**, indexed by i : x_i^1, x_i^2, ...
- Vector **y-variables**, indexed by j : y_j^1, y_j^2, ...

What we provide

For $i = 1, \ldots, N$, you want to compute:

$$a_i \;=\; \text{Reduction}_{j=1,\ldots,M} \Big[F\,(\,p^1, p^2, \ldots, x_i^1, x_i^2, \ldots, y_j^1, y_j^2, \ldots) \Big]\,,$$

with :

- "**Reduction**" : Sum, Max, ArgMin, LogSumExp, ...
- A vector-valued **formula**: F.
- Vector **parameters**: p^1, p^2, ...
- Vector **x-variables**, indexed by i : x_i^1, x_i^2, ...
- Vector **y-variables**, indexed by j : y_j^1, y_j^2, ...

What we provide

For $i = 1, \ldots, N$, you want to compute:

$$a_i \;=\; \text{Reduction}_{j=1,\ldots,M} \Big[F\big(p^1, p^2, \ldots, x_i^1, x_i^2, \ldots, y_j^1, y_j^2, \ldots \big) \Big],$$

with :

- "**Reduction**" : Sum, Max, ArgMin, LogSumExp, ...
- A vector-valued **formula**: F.
- Vector **parameters**: p^1, p^2, ...
- Vector **x-variables**, indexed by i : x_i^1, x_i^2, ...
- Vector **y-variables**, indexed by j : y_j^1, y_j^2, ...

With **KeOps** you will get:

- **Linear** memory footprint.

What we provide

For $i = 1, \ldots, N$, you want to compute:

$$a_i = \text{Reduction}_{j=1,\ldots,M} \Big[F\,(p^1, p^2, \ldots, x_i^1, x_i^2, \ldots, y_j^1, y_j^2, \ldots) \Big],$$

with :

- "**Reduction**" : Sum, Max, ArgMin, LogSumExp, ...
- A vector-valued **formula**: F.
- Vector **parameters**: $p^1, p^2, \ldots$
- Vector **x-variables**, indexed by i : $x_i^1, x_i^2, \ldots$
- Vector **y-variables**, indexed by j : $y_j^1, y_j^2, \ldots$

With **KeOps** you will get:

- **Linear** memory footprint.
- High order **derivatives** – thank you Joan!

What we provide

For $i = 1, \ldots, N$, you want to compute:

$$a_i = \text{Reduction}_{j=1,\ldots,M} \Big[F\big(p^1, p^2, \ldots, x_i^1, x_i^2, \ldots, y_j^1, y_j^2, \ldots \big) \Big],$$

with :

- "**Reduction**" : Sum, Max, ArgMin, LogSumExp, ...
- A vector-valued **formula**: F.
- Vector **parameters**: $p^1, p^2, \ldots$
- Vector **x-variables**, indexed by i : $x_i^1, x_i^2, \ldots$
- Vector **y-variables**, indexed by j : $y_j^1, y_j^2, \ldots$

With **KeOps** you will get:

- **Linear** memory footprint.
- High order **derivatives** – thank you Joan!
- Support for **block-sparse** (=cluster-aware) reductions.

KeOps' low-level interface: `generic_sum`

With x_i, y_j points in $\mathbb{R}^3$ and b_j a 2D-signal:

$$a_i = \sum_{j=1}^{M} \exp\left(-\frac{\|x_i - y_j\|^2}{\sigma^2}\right) \cdot b_j$$

KeOps' low-level interface: `generic_sum`

With x_i, y_j points in $\mathbb{R}^3$ and b_j a 2D-signal:

$$a_i = \sum_{j=1}^{M} \exp\left(-\frac{\|x_i - y_j\|^2}{\sigma^2}\right) \cdot b_j$$

```
from pykeops.torch import generic_sum

gaussian_conv = generic_sum(
  "Exp(-G*SqDist(X,Y)) * B", # Custom formula
  "A = Vx(2)", # Output,  2D, indexed by i
  "G = Pm(1)", # 1st arg, 1D, parameter
  "X = Vx(3)", # 2nd arg, 3D, indexed by i
  "Y = Vy(3)", # 3rd arg, 3D, indexed by j
  "B = Vy(2)") # 4th arg, 2D, indexed by j
```

KeOps' low-level interface: `generic_sum`

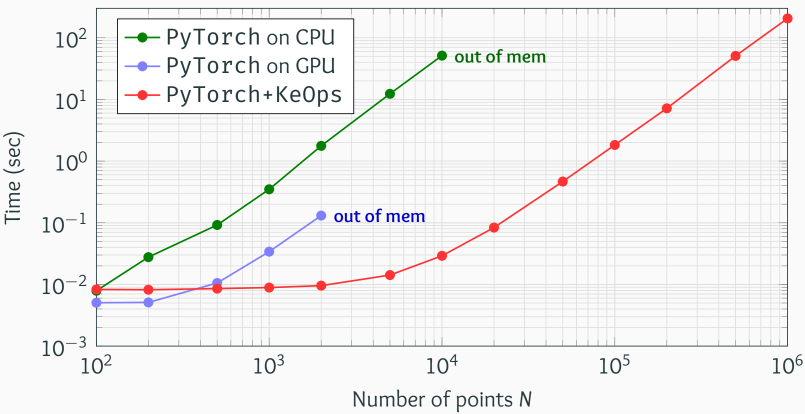
With x_i, y_j points in $\mathbb{R}^3$ and b_j a 2D-signal:

$$a_i = \sum_{j=1}^{M} \exp\left(-\frac{\|x_i - y_j\|^2}{\sigma^2}\right) \cdot b_j$$

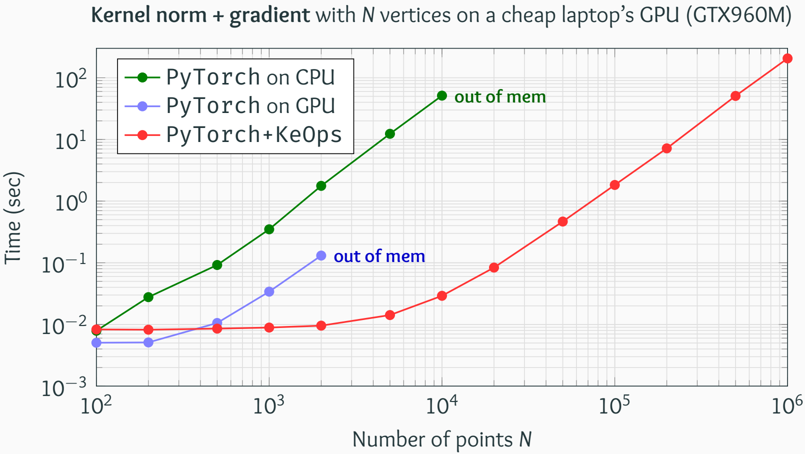
```python
from pykeops.torch import generic_sum

gaussian_conv = generic_sum(
  "Exp(-G*SqDist(X,Y)) * B", # Custom formula
  "A = Vx(2)", # Output,  2D, indexed by i
  "G = Pm(1)", # 1st arg, 1D, parameter
  "X = Vx(3)", # 2nd arg, 3D, indexed by i
  "Y = Vy(3)", # 3rd arg, 3D, indexed by j
  "B = Vy(2)") # 4th arg, 2D, indexed by j

# Simply apply your routine to CPU/GPU torch tensors!
a = gaussian_conv( 1/sigma**2, x, y, b )
```

It works!

Kernel norm + gradient with *N* vertices on a cheap laptop's GPU (GTX960M)



It works!



+ You can go further and use **multiscale**, FMM-like information.

Recap of today's presentation

Key points:

- Gradients are **cheap**.

Recap of today's presentation

Key points:

- Gradients are **cheap**.
- PyTorch is the perfect framework for researchers as it's both **simple** and flexible.

Recap of today's presentation

Key points:

- Gradients are **cheap**.
- PyTorch is the perfect framework for researchers as it's both **simple** and flexible.
- It generalizes **regression** to arbitrary models, without hassle.

Recap of today's presentation

Key points:

- Gradients are **cheap**.
- PyTorch is the perfect framework for researchers as it's both **simple** and flexible.
- It generalizes **regression** to arbitrary models, without hassle.
- Multiscale image analysis has gone through a revolution over the past six years.

What about your field?

Going further

PyTorch Tutorials

PYTORCH

0.3.0.post4

Search docs

BEGINNER TUTORIALS

- Deep Learning with PyTorch: A 60 Minute Blitz
- PyTorch for former Torch users
- **Learning PyTorch with Examples**
  - Tensors
  - Autograd
  - *nn* module
  - Examples
- Transfer Learning tutorial
- Data Loading and Processing Tutorial
- Deep Learning for NLP with Pytorch

INTERMEDIATE TUTORIALS

- Classifying Names with a Character-Level RNN
- Generating Names with a Character-Level RNN

Docs » Learning PyTorch with Examples   View page source

Learning PyTorch with Examples

Author: Justin Johnson

This tutorial introduces the fundamental concepts of PyTorch through self-contained examples.

At its core, PyTorch provides two main features:

- An n-dimensional Tensor, similar to numpy but can run on GPUs
- Automatic differentiation for building and training neural networks

We will use a fully-connected ReLU network as our running example. The network will have a single hidden layer, and will be trained with gradient descent to fit random data by minimizing the Euclidean distance between the network output and the true output.

Note

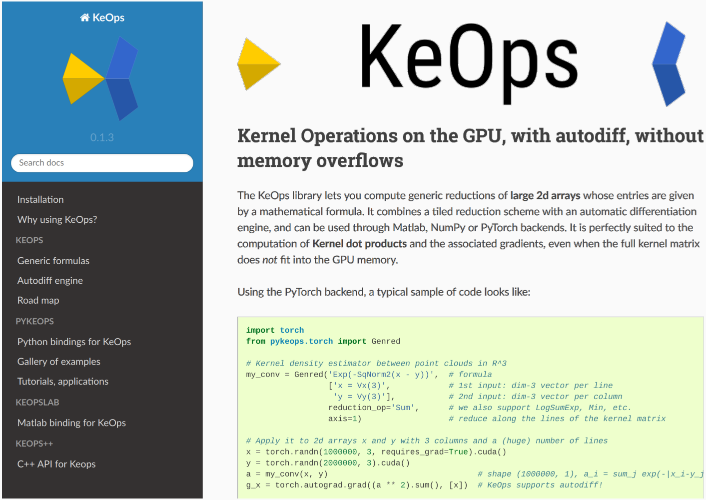
You can browse the individual examples at the end of this page.

Table of Contents

- Tensors
  - Warm-up: numpy
  - PyTorch: Tensors
- Autograd

`pytorch.org`

Going further



KeOps

0.1.3

Search docs

Installation
Why using KeOps?
KEOPS
Generic formulas
Autodiff engine
Road map
PYKEOPS
Python bindings for KeOps
Gallery of examples
Tutorials, applications
KEOPSLAB
Matlab binding for KeOps
KEOPS++
C++ API for Keops

KeOps

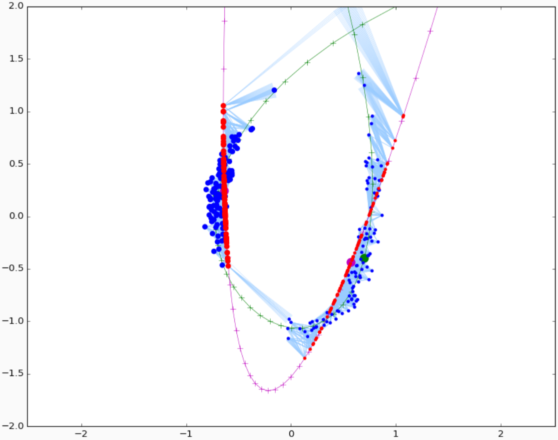
Kernel Operations on the GPU, with autodiff, without memory overflows

The KeOps library lets you compute generic reductions of **large 2d arrays** whose entries are given by a mathematical formula. It combines a tiled reduction scheme with an automatic differentiation engine, and can be used through Matlab, NumPy or PyTorch backends. It is perfectly suited to the computation of **Kernel dot products** and the associated gradients, even when the full kernel matrix does *not* fit into the GPU memory.

Using the PyTorch backend, a typical sample of code looks like:

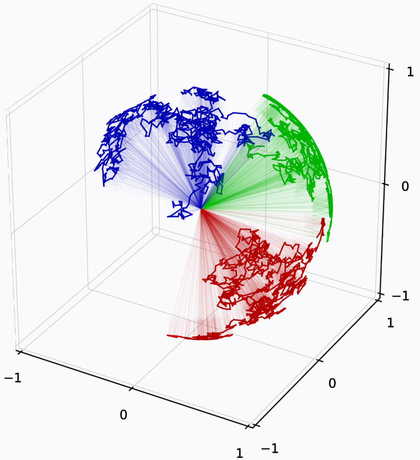
```
import torch
from pykeops.torch import Genred

# Kernel density estimator between point clouds in R^3
my_conv = Genred('Exp(-SqNorm2(x - y))',  # formula
                 ['x = Vx(3)',            # 1st input: dim-3 vector per line
                  'y = Vy(3)'],           # 2nd input: dim-3 vector per column
                 reduction_op='Sum',      # we also support LogSumExp, Min, etc.
                 axis=1)                  # reduce along the lines of the kernel matrix

# Apply it to 2d arrays x and y with 3 columns and a (huge) number of lines
x = torch.randn(1000000, 3, requires_grad=True).cuda()
y = torch.randn(2000000, 3).cuda()
a = my_conv(x, y)                          # shape (1000000, 1), a_i = sum_j exp(-|x_i-y_j
g_x = torch.autograd.grad((a ** 2).sum(), [x])  # KeOps supports autodiff!
```

www.kernel-operations.io



www.math.ens.fr/~feydy/Teaching/

Going further



Differential geometry and stochastic dynamics with Deep Learning numerics, Kühnel, Arnaudon, Sommer (2017)

Thank you for your attention.