Introduction to: Deep Learning: du neurone artificiel aux réseaux profonds

Diane Lingrand

UNIVERSITÉ
CÔTE D'AZUR

SI 4

2024 - 2025

Pour communiquer

- le **slack** du cours `#si4-deep-learning`, pour les questions d'ordre général pouvant intéresser les autres étudiants du cours.
  - Tout le monde peut répondre aux questions posées, pas seulement les enseignants !
  - Cela permet également de corriger les fausses vérités lors des révisions.
- la page **moodle** du cours
  - les supports de cours, les sujets de TP, les rendus et autres documents
- les discussions privées sur **slack** ou par **mail** avec les enseignants :
  - `prenom.nom@univ-cotedazur.fr`

Outline

Nouveau cours à la rentrée 2024

- Volonté d'un socle commun pour comprendre les modèles fondation
  - S7 : du neurone artificiel aux réseaux de neurones profonds
  - S8 : des transformers aux modèles fondation
- Réseaux de neurones
  - approximateur universel de fonction
- apprentissage de la librairie `pytorch`

Outline

Cadre d'apprentissage simplifié

- Apprentissage supervisé : nécessite des données (X) ET des labels (y)
  - Régression :
    - estimation d'une ou plusieurs valeurs
  - Classification :
    - attribution d'une classe à chaque donnée

Loss function for a regression algorithm

- mse : mean square error

$$\mathcal{L}_{mse} = \frac{1}{n}\sum_{i=0}^{n}(y^i_{pred} - y^i_{truth})^2$$

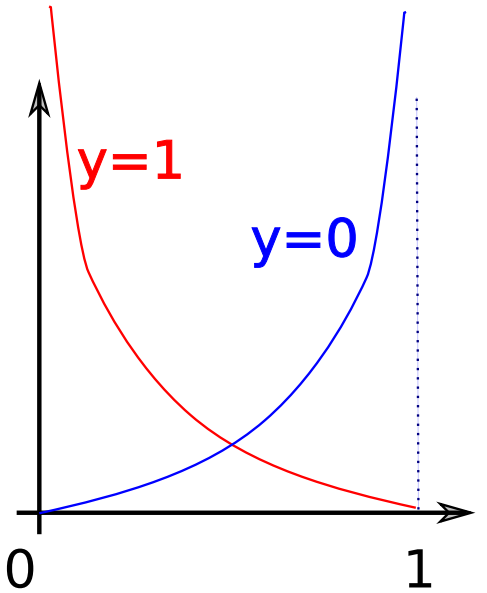
- rmse : root mean square error

$$E = \sqrt{\frac{1}{n}\sum_{i=0}^{n}(y^i_{pred} - y^i_{truth})^2}$$

Loss function for a classification algorithm

- special case of binary cross-entropy :

$$J = -\frac{1}{m}\sum_{i=1}^{m}(y^i \log(y^i_{pred}) + (1-y^i)\log(1-y^i_{pred}))$$



Loss function for a classification algorithm

- categorical cross-entropy :

$$J = -\frac{1}{m}\sum_{i=1}^{m} y_c^i \log(y_{pred}^i)$$

  - categorical encoding of labels
    - e.g. classification into 3 classes ('cat', 'dog', 'bird')
    - label of cat : $[1, 0, 0]$, label of bird : $[0, 0, 1]$
  - normalized output of neural networks :

$$\sum_i y_{pred}^i = 1$$

  - Example : output for a sample from the class 'bird' j is $y_{pred}^j = [0.1, 0.2, 0.7]$
    - we verify that $0.1 + 0.2 + 0.7 = 1$
    - $y_c^j = [0, 0, 1]$
    - contribution to the loss : $0 * \log 0.1 + 0 * \log 0.2 + 1 * \log 0.7 = -0.36$

Minimisation of an error function

- How to guess where the minimum is ?
  - You can ask for values of $f(\theta)$ for some θ

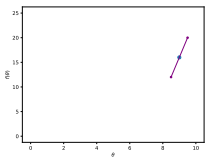
Minimisation of an error function

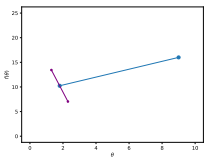
- How to guess where the minimum is ?
  - You can ask for values of $f(\theta)$ for some θ
  - Random guess
  - Exhaustive search with steps
  - Gradient descent (if differentiable)

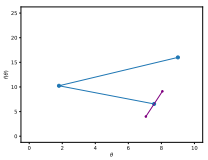
Outline

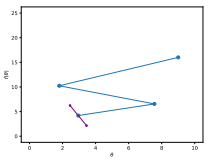
Gradient descent : idea

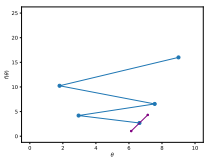
- random initialisation of parameter θ
- iterations
  - increase/decrease θ proportionally to the opposite of the tangent


init.

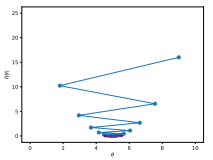

iter. 1


iter. 2


iter. 3


iter. 4

. . .


iter. 20

Gradient descent : very simple code example

```
def f(x):
    return (x-5)*(x-5)
def df(x):
    return 2*(x-5)

theta = 9 # initialisation
alpha = 0.3
limite = 20; epsilon = 0.1
cpt = 0

while (abs(df(theta)) > epsilon) and (cpt < limite):
    theta -= alpha*df(theta)
    cpt += 1

print('min of f for theta = ', theta, 'with f(theta)=', f(theta))
```

Derivative ($f : \mathbb{R} \rightarrow \mathbb{R}$)

- Derivative of univariate function : $f'(x) = \frac{df}{dx}(x)$
  - derivatives of usual functions are well-known :

$f(x)$	λx	x^n	e^x	$sin(x)$	$cos(x)$	$ln(x)$	...
$f'(x)$	λ	nx^{n-1}	e^x	$cos(x)$	$-sin(x)$	$\frac{1}{x}$	...

  - derivatives of functions that are composed by other functions :
    - $(f + g)'(x) = f'(x) + g'(x)$
    - $(f * g)'(x) = f'(x) * g(x) + f(x) * g'(x)$
    - $(fog)'(x) = f'(g(x)) * g'(x)$
    - also written as $\frac{d(fog)}{dx} = \frac{df}{dg} * \frac{dg}{dx}$ (**chain rule**)
    - can be applied recursively

Derivative ($f : \mathbb{R} \to \mathbb{R}$)

- Derivative of univariate function : $f'(x) = \frac{df}{dx}(x)$
  - derivatives of usual functions are well-known :

$f(x)$	λx	x^n	e^x	$sin(x)$	$cos(x)$	$ln(x)$	...
$f'(x)$	λ	nx^{n-1}	e^x	$cos(x)$	$-sin(x)$	$\frac{1}{x}$	...

  - derivatives of functions that are composed by other functions :
    - $(f+g)'(x) = f'(x) + g'(x)$
    - $(f*g)'(x) = f'(x)*g(x) + f(x)*g'(x)$
    - $(fog)'(x) = f'(g(x))*g'(x)$
    - also written as $\frac{d(fog)}{dx} = \frac{df}{dg} * \frac{dg}{dx}$ (**chain rule**)
    - can be applied recursively
- Example : $s(x) = \frac{1}{1+e^{-x}}$
  - composition of : negation, exponential, addition to a scalar and inverse
  - $s'(x) = -\frac{d(1+e^{-x})/dx}{(1+e^{-x})^2} = \frac{e^{-x}}{(1+e^{-x})^2}$
    - simplified as : $s'(x) = \frac{e^{-x}+1-1}{(1+e^{-x})^2} = s(x) - s^2(x) = s(x)(1-s(x))$

Gradient ($f : \mathbb{R}^n \rightarrow \mathbb{R}$)

- Derivatives of multivariate functions : $f(x_0, x_1, ..., x_n) = f(\mathbf{x})$
  - consider the derivative with respect to each of the variables :
    - $\frac{\partial f}{\partial x_0}, \frac{\partial f}{\partial x_1}, \ldots \frac{\partial f}{\partial x_n}$
  - rearranged in a vector : this is the **gradient vector**
    $\nabla f(\mathbf{x}) = \frac{df}{d\mathbf{x}}(\mathbf{x}) = [\frac{\partial f}{\partial x_0}(\mathbf{x}), \frac{\partial f}{\partial x_1}(\mathbf{x}) ... \frac{\partial f}{\partial x_n}(\mathbf{x})]^T$

Gradient ($f : \mathbb{R}^n \rightarrow \mathbb{R}$)

- Derivatives of multivariate functions : $f(x_0, x_1, ..., x_n) = f(\mathbf{x})$
  - consider the derivative with respect to each of the variables :
    - $\frac{\partial f}{\partial x_0}, \frac{\partial f}{\partial x_1}, \dots \frac{\partial f}{\partial x_n}$
  - rearranged in a vector : this is the **gradient vector**
    $\nabla f(\mathbf{x}) = \frac{df}{d\mathbf{x}}(\mathbf{x}) = [\frac{\partial f}{\partial x_0}(\mathbf{x}), \frac{\partial f}{\partial x_1}(\mathbf{x}) ... \frac{\partial f}{\partial x_n}(\mathbf{x})]^T$
- Example 1 : $f(x_0, x_1) = ax_0 + bx_1$
  - Gradient : $\nabla f(\mathbf{x}) = \nabla f(x_0, x_1) = [a\ b]^T$

Gradient ($f : \mathbb{R}^n \rightarrow \mathbb{R}$)

- Derivatives of multivariate functions : $f(x_0, x_1, ..., x_n) = f(\mathbf{x})$
  - consider the derivative with respect to each of the variables :
    - $\frac{\partial f}{\partial x_0}, \frac{\partial f}{\partial x_1}, \dots \frac{\partial f}{\partial x_n}$
  - rearranged in a vector : this is the **gradient vector**
    $\nabla f(\mathbf{x}) = \frac{df}{d\mathbf{x}}(\mathbf{x}) = [\frac{\partial f}{\partial x_0}(\mathbf{x}), \frac{\partial f}{\partial x_1}(\mathbf{x}) ... \frac{\partial f}{\partial x_n}(\mathbf{x})]^T$
- Example 1 : $f(x_0, x_1) = ax_0 + bx_1$
  - Gradient : $\nabla f(\mathbf{x}) = \nabla f(x_0, x_1) = [a\ b]^T$
- Example 2 : $f(x_0, x_1) = s(ax_0 + bx_1)$
  - Gradient :
    - let us denote $z = ax_0 + bx_1$:
    - $\nabla f(\mathbf{x}) = \nabla f(x_0, x_1) = [\frac{ds}{dz} * \frac{\partial z}{\partial x_0}\ ,\ \frac{ds}{dz} * \frac{\partial z}{\partial x_1}]^T = s'(ax_0 + bx_1)[a\ ,\ b]^T$

Jacobian ($f : \mathbb{R}^n \rightarrow \mathbb{R}^m$)

- In short :
  - if $n = 1 = m$: derivatives
  - if $n \geq 1$ and $m = 1$: gradient
- if $n \geq 1$ and $m \geq 1$: Jacobian
  - array or matrix composed of gradients for each m component of the function output :

$$\mathbf{J} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

Gradient

- If we build a mathematical function :
  - using closed-form expressions
  - composed of elementary functions that are differentiable
    - and we can compute their gradients
    - the composition is differentiable
- we can compute it's gradient using the chain rule
  - it could be painful but if you are methodic, it's not a big deal !
  - it could be done automatically by .... a computer !
- Gradient computation :
  - numerical differentiation
    - based on : $f'(x) = \lim_{h \to 0} \frac{f(x+h)-f(x)}{h}$
  - symbolic differentiation
    - based on chain rules
    - only closed form expressions (no cond., no loops, no rec...)
  - automatic differentiation
    - from code/algorithm to gradients

Gradient by Numerical differentiation

- Approximation subject to :
  - Round-off error
  - Truncation error

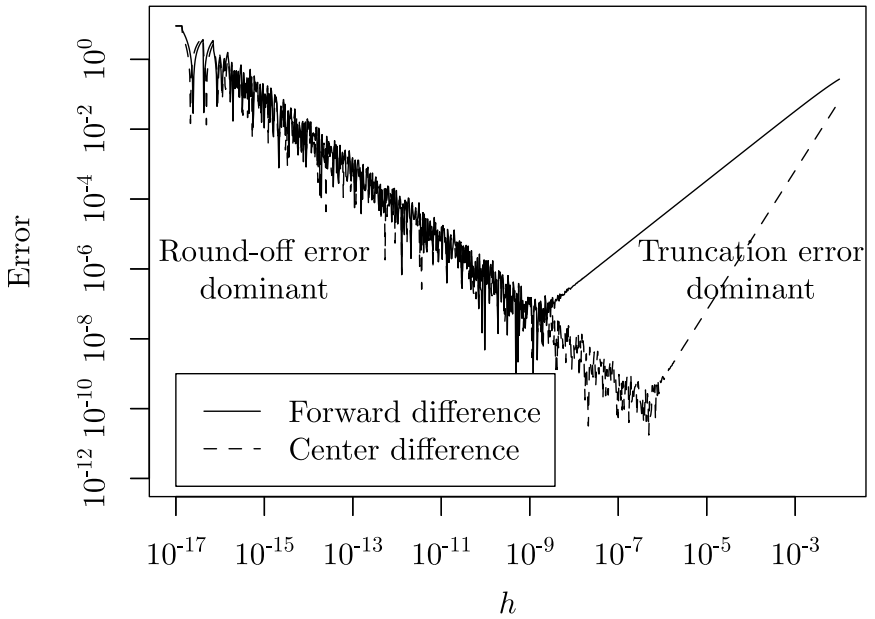


Figure 3: Error in the forward (Eq. 1) and center difference (Eq. 2) approximations as a function of step size h, for the derivative of the truncated logistic map $f(x) = 64x(1-x)(1-2x)^2(1-8x+8x^2)^2$. Plotted errors are computed using $E_{\text{forward}}(h, x_0) = \left|\frac{f(x_0+h)-f(x_0)}{h} - \frac{d}{dx}f(x)\big|_{x_0}\right|$ and $E_{\text{center}}(h, x_0) = \left|\frac{f(x_0+h)-f(x_0-h)}{2h} - \frac{d}{dx}f(x)\big|_{x_0}\right|$ at $x_0 = 0.2$.

from [Baydin etal 2018]

Gradient by Symbolic differentiation

- Many tools : Maple, SymPy ...
- Requires closed-form expressions
  - no condition, no loop, no recursion ...
- May leed to expression swell
  - ex : derivative of a product : $(uv)' = uv' + u'v$
  - problem for functions used in neural networks

Table 1: Iterations of the logistic map $l_{n+1} = 4l_n(1 - l_n)$, $l_1 = x$ and the corresponding derivatives of l_n with respect to x, illustrating expression swell.

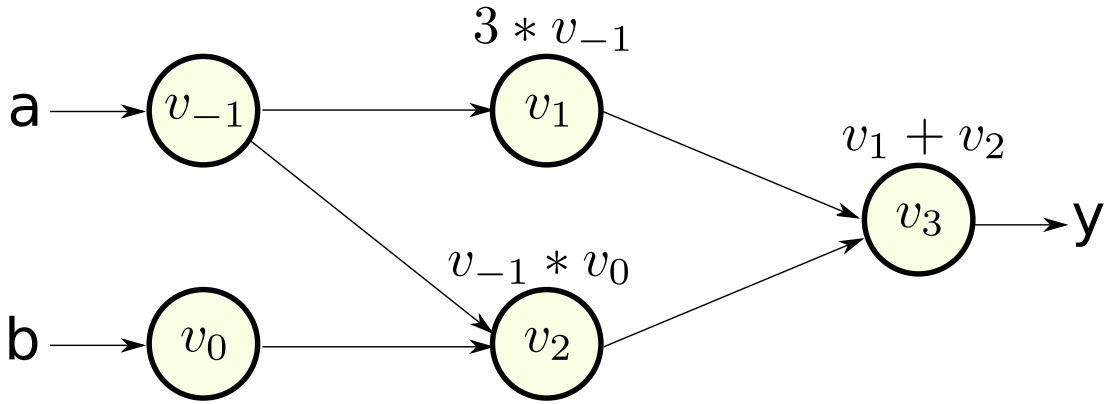
n	l_n	$\frac{d}{dx}l_n$	$\frac{d}{dx}l_n$ (Simplified form)
1	x	1	1
2	$4x(1-x)$	$4(1-x)-4x$	$4-8x$
3	$16x(1-x)(1-2x)^2$	$16(1-x)(1-2x)^2-16x(1-2x)^2-64x(1-x)(1-2x)$	$16(1-10x+24x^2-16x^3)$
4	$64x(1-x)(1-2x)^2(1-8x+8x^2)^2$	$128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2)+64(1-x)(1-2x)^2(1-8x+8x^2)^2-64x(1-2x)^2(1-8x+8x^2)^2-256x(1-x)(1-2x)(1-8x+8x^2)^2$	$64(1-42x+504x^2-2640x^3+7040x^4-9984x^5+7168x^6-2048x^7)$

from [Baydin etal 2018]

Gradient by Automatic differentiation

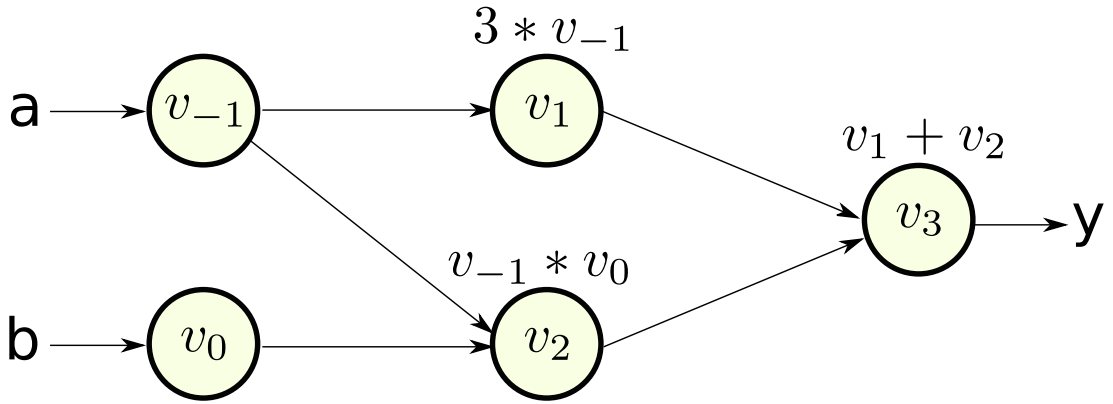
- computed from the code itself, using chain rules
- evaluation of numerical values
- mainly 2 modes :
  - forward mode
    - replace intermediate variable v_i by $(v_i, \dot{v}_i)$
    - implementation using operators overloading or source code transformation
    - usually used when few inputs and many outputs
  - reverse mode
    - forward pass for v_i values evaluations
    - backward pass for evaluation of adjoint values $\bar{v}_i = \frac{\partial f}{\partial x}$
    - with a tree : $\bar{v}_i = \sum_{j \ child \ of \ i} \bar{v}_j \frac{\partial v_j}{\partial v_i}$
    - prefered when many inputs and few outputs : machine learning applications
- this is what `torch.autograd` in `pytorch` or `tf.GradientTape` in `tensorflow` are doing for you
  - video for automatic differentiation : `https://www.youtube.com/watch?v=wG_nF1awSSY`
  - reference paper : [Baydin etal 2018]

An example of reverse mode



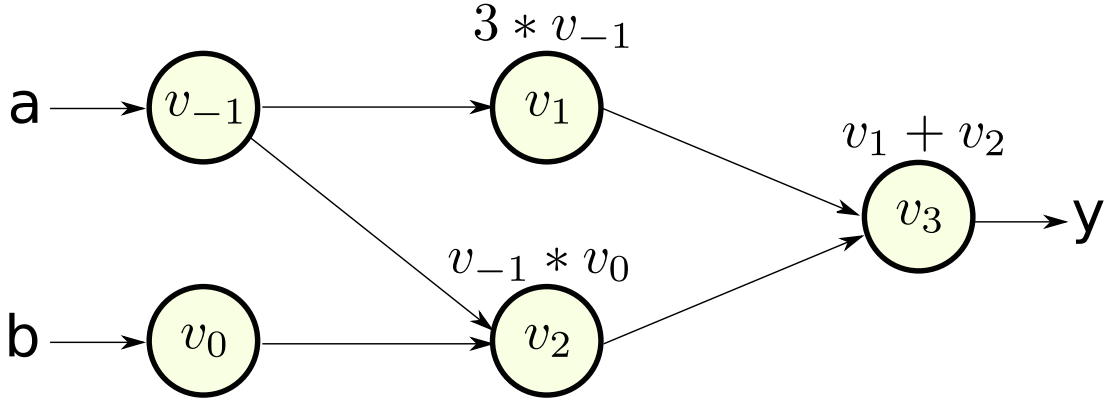
- function $y = 3 * a + a * b$
- Forward pass :
  - computation of all v_i
- Reverse pass :
  - computation of adjoints : $\bar{v}_i = \dfrac{\partial y}{\partial v_i} = \sum_{j \text{ child of } i} \bar{v}_j \dfrac{\partial v_j}{\partial v_i}$
    - $\bar{v}_3 =$

An example of reverse mode



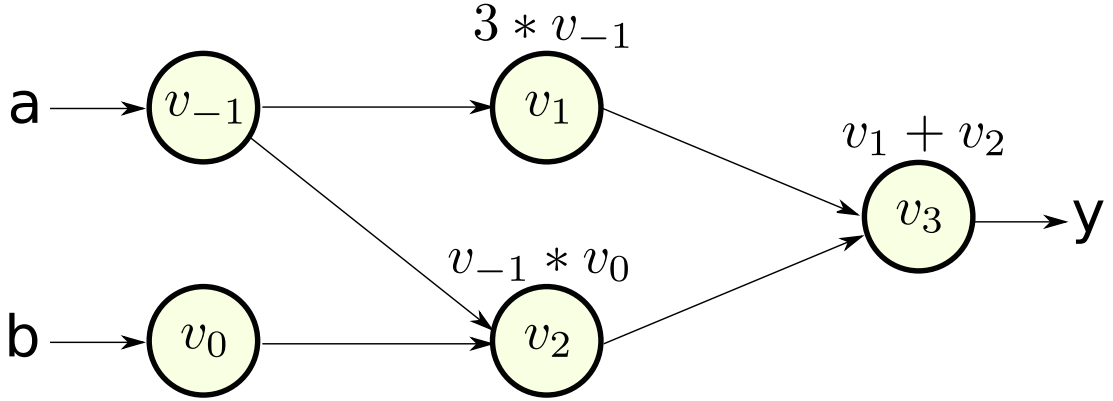
- function $y = 3 * a + a * b$
- Forward pass :
  - computation of all v_i
- Reverse pass :
  - computation of adjoints : $\bar{v}_i = \frac{\partial y}{\partial v_i} = \sum_{j \text{ child of } i} \bar{v}_j \frac{\partial v_j}{\partial v_i}$
    - $\bar{v}_3 = 1$
    - $\bar{v}_2 =$

An example of reverse mode



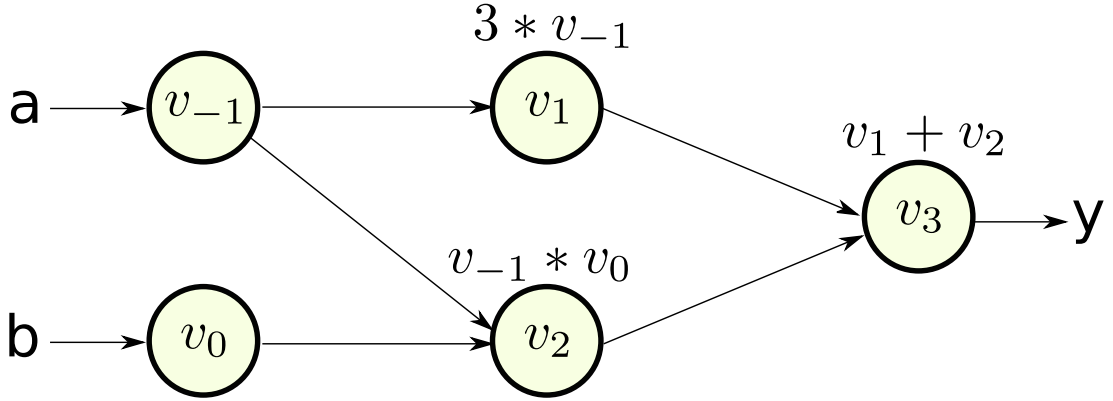
- function $y = 3 * a + a * b$
- Forward pass :
  - computation of all v_i
- Reverse pass :
  - computation of adjoints : $\bar{v}_i = \frac{\partial y}{\partial v_i} = \sum_{j \ child \ of \ i} \bar{v}_j \frac{\partial v_j}{\partial v_i}$
    - $\bar{v}_3 = 1$
    - $\bar{v}_2 = \frac{\partial y}{\partial v_2} = \bar{v}_3 \frac{\partial v_3}{\partial v_2} = 1$
    - $\bar{v}_0 =$

An example of reverse mode



- function $y = 3 * a + a * b$
- Forward pass :
  - computation of all v_i
- Reverse pass :
  - computation of adjoints : $\bar{v}_i = \frac{\partial y}{\partial v_i} = \sum_{j \text{ child of } i} \bar{v}_j \frac{\partial v_j}{\partial v_i}$
    - $\bar{v}_3 = 1$
    - $\bar{v}_2 = \frac{\partial y}{\partial v_2} = \bar{v}_3 \frac{\partial v_3}{\partial v_2} = 1$
    - $\bar{v}_0 = \frac{\partial y}{\partial v_0} = \bar{v}_2 \frac{\partial v_2}{\partial v_0} = v_{-1} = a$
    - $\bar{v}_{-1} =$

An example of reverse mode



- function $y = 3 * a + a * b$
- Forward pass :
  - computation of all v_i
- Reverse pass :
  - computation of adjoints : $\bar{v}_i = \frac{\partial y}{\partial v_i} = \sum_{j \text{ child of } i} \bar{v}_j \frac{\partial v_j}{\partial v_i}$
    - $\bar{v}_3 = 1$
    - $\bar{v}_2 = \frac{\partial y}{\partial v_2} = \bar{v}_3 \frac{\partial v_3}{\partial v_2} = 1$
    - $\bar{v}_0 = \frac{\partial y}{\partial v_0} = \bar{v}_2 \frac{\partial v_2}{\partial v_0} = v_{-1} = a$
    - $\bar{v}_{-1} = \frac{\partial y}{\partial v_{-1}} = \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}} + \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}} = v_0 + 3 = b + 3$

Outline

Variables

- tensors
  - dimensions (`shape`, `ndim`)
    - e.g. shape of `(10000,28,28)` with `ndim` of 3
  - type or `dtype` :
    - `pytorch.float32`, `pytorch.float64`, `pytorch.float16`
    - `pytorch.uint8`, `pytorch.int8` ...
  - device where the tensor is allocated
    - cpu, cuda ...
  - gradients : if you need to compute gradients with respect to this variable

Variables : scalar, vectors, matrices

1.1 scalar values

```
a = torch.tensor(2)
```

```
print('nb dim = ', a.dim(), '\n shape = ',a.shape, '\n dtype = ', a.dtype, '\n␣
 ↪value = ', a.item())
```

```
nb dim =  0
 shape =  torch.Size([])
 dtype =  torch.int64
 value =  2
```

1.2 vectors

```
b = torch.tensor([2.1, 7.5, 3])
```

```
print('nb dim = ', b.dim(), '\n shape = ',b.shape, '\n dtype = ', b.dtype)
```

```
nb dim =  1
 shape =  torch.Size([3])
 dtype =  torch.float32
```

1.3 matrix or 2d-array

```
c = torch.tensor([[2, 7, 3],[1,4,9]])
```

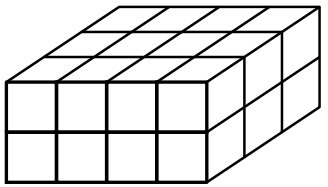
```
print('nb dim = ', c.dim(), '\n shape = ',c.shape, '\n dtype = ', c.dtype)
```

```
nb dim =  2
 shape =  torch.Size([2, 3])
 dtype =  torch.int64
```

1.4 tensor or nd-array ($n > 2$)

```
d = torch.tensor([[[2, 7, 3, 6],
                   [1, 4, 9, 11]],
                  [[5, 4, 0, 6],
                   [10, 8, 1, 3]],
                  [[3, 6, 2, 8],
                   [9, 1, 2, 7]]])
print(d)
```

```
tensor([[[ 2,  7,  3,  6],
         [ 1,  4,  9, 11]],

        [[ 5,  4,  0,  6],
         [10,  8,  1,  3]],

        [[ 3,  6,  2,  8],
         [ 9,  1,  2,  7]]])
```



```
print('nb dim = ', d.dim(), '\n shape = ',d.shape, '\n dtype = ', d.dtype)
```

```
nb dim =  3
 shape =  torch.Size([3, 2, 4])
 dtype =  torch.int64
```

Variables : devices

2 Variables: device

```
# default device
print("d is stored on device: ", d.device)
```

```
d is stored on device:  cpu
```

```
# making the code device agnostic
device = "cuda" if torch.cuda.is_available() else "cpu"
device
```

```
'cuda'
```

```
torch.cuda.device_count()
```

```
1
```

```
# move the tensor d to the device
d_gpu = d.to(device)
```

```
print("d_gpu is stored on device: ", d_gpu.device)
```

```
d_gpu is stored on device:  cuda:0
```

Variables : creation

```
# zeros
z0 = torch.zeros(2,3)
print(z0)
zc = torch.zeros_like(c)
print(zc)
```

```
tensor([[0., 0., 0.],
        [0., 0., 0.]])
tensor([[0, 0, 0],
        [0, 0, 0]])
```

```
# ones
z1 = torch.ones(3,4)
print(z1)
```

```
tensor([[1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]])
```

```
# linear
zl = torch.linspace(0,1,5)
print(zl)
```

```
tensor([0.0000, 0.2500, 0.5000, 0.7500, 1.0000])
```

```
# random
zr = torch.randint(0,10,(3,10))
print(zr)
```

```
tensor([[5, 5, 9, 4, 7, 0, 7, 9, 9, 2],
        [6, 1, 8, 4, 2, 0, 1, 2, 8, 7],
        [8, 4, 1, 8, 2, 3, 2, 2, 1, 9]])
```

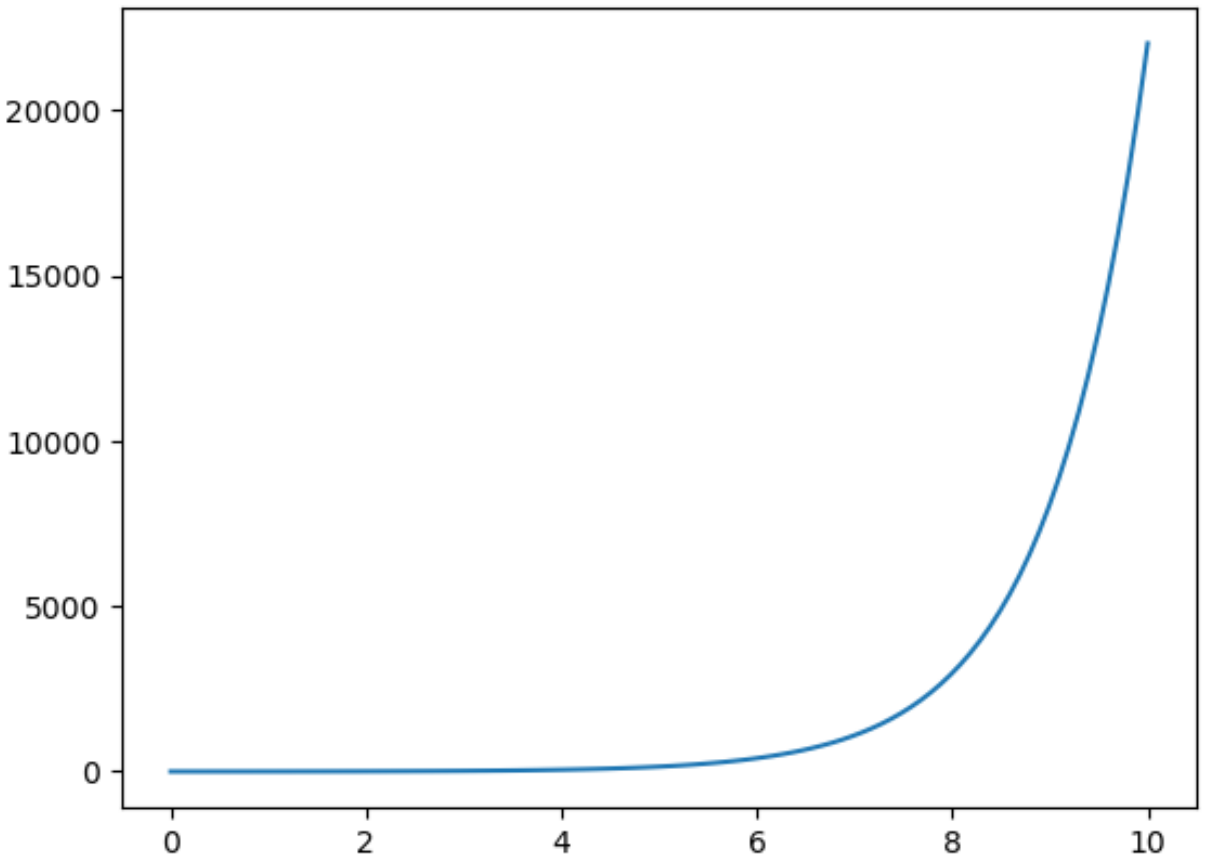
```
# from and to numpy array
anArray = np.array([1,5,2,7])
aTensor = torch.from_numpy(anArray)
backToNumpy = aTensor.numpy()
print(anArray,aTensor,backToNumpy, sep='\n')
```

```
[1 5 2 7]
tensor([1, 5, 2, 7])
[1 5 2 7]
```

Variables : operations

- Have a look on the list of Math operations
  - sin, cos, exp, log, erf, mul, neg, sign, sqrt ...

```
# back to first week in SI3 'données numériques'
x = torch.linspace(0,10,100)
y = torch.exp(x)
plt.plot(x,y)
```



Pytorch Autograd : 2 examples

- simple function of 2 variables a and b :
  - $y = 3 * a + a * b$
  - $\frac{\partial y}{\partial a} =$

Pytorch Autograd : 2 examples

- simple function of 2 variables a and b :
  - $y = 3 * a + a * b$
  - $\frac{\partial y}{\partial a} = 3 + b$
  - $\frac{\partial y}{\partial b} = a$

```
a = torch.tensor(10.0, requires_grad=True)
b = torch.tensor(20.0, requires_grad=True)
y = 3*a + a*b # forward: computes y value and the tree
y.backward() # reverse mode
print(a.grad) # dy/da
print(b.grad) # dy/db
# tree is released
```

```
tensor(23.)
tensor(10.)
```

Pytorch Autograd : 2 examples

- simple function of 2 variables a and b :
  - $y = 3 * a + a * b$
  - $\frac{\partial y}{\partial a} = 3 + b$
  - $\frac{\partial y}{\partial b} = a$

```
a = torch.tensor(10.0, requires_grad=True)
b = torch.tensor(20.0, requires_grad=True)
y = 3*a + a*b # forward: computes y value and the tree
y.backward() # reverse mode
print(a.grad) # dy/da
print(b.grad) # dy/db
# tree is released
```

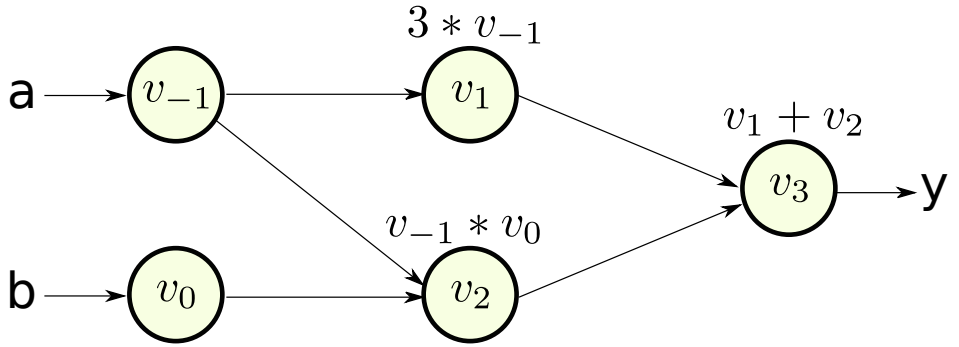
```
tensor(23.)
tensor(10.)
```

- a more complex function

```
a = torch.tensor(1.0, requires_grad=True)
b = torch.tensor(2.0, requires_grad=True)
y = 10.0 / (1 + torch.exp(3*a + a*b))
y.backward() # reverse mode
print(a.grad) # dy/da
print(b.grad) # dy/db
```

```
tensor(-0.3324)
tensor(-0.0665)
```

Pytorch Autograd : tree



- Tree construction with forward pass :
  - $v_{-1} = \text{a} = 10$, $v_0 = \text{b} = 20$
  - $v_1 = 30$, $v_2 = 200$, $y = v_3 = 220$
  - all ops (and reverse grad) are stored when `requires_grad` is `True`
- Reverse pass :
  - $\bar{v}_3 = 1$
  - $\bar{v}_2 = 1$, $\bar{v}_1 = 1$
  - $\bar{v}_0 = \text{a} = 10$, $\bar{v}_{-1} = \text{b} + 3 = 23$
- In pytorch :
  - trees are dynamic and are released after the `backward` call
  - gradients are accumulated
- An in-depth explanation of tree construction in pytorch :

  `https://www.youtube.com/watch?v=MswxJw-8PvE&t=1s`

Pytorch Autograd : release of tree

```
a = torch.tensor(1.0, requires_grad=True)
b = torch.tensor(2.0, requires_grad=True)
y = 10.0 / (1 + torch.exp(3*a + a*b))
y.backward() # reverse mode
print(a.grad) # dy/da
print(b.grad) # dy/db
y.backward() # tree is released
print(a.grad) # dy/da
print(b.grad) # dy/db
```

```
tensor(-0.3324)
tensor(-0.0665)
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Cell In[32], line 7
      5 print(a.grad) # dy/da
      6 print(b.grad) # dy/db
----> 7 y.backward() # tree is released
      8 print(a.grad) # dy/da
      9 print(b.grad)
```

```
RuntimeError: Trying to backward through the graph a second time (or directly␣
↪access saved tensors after they have already been freed). Saved intermediate␣
↪values of the graph are freed when you call .backward() or autograd.grad().␣
↪Specify retain_graph=True if you need to backward through the graph a second␣
↪time or if you need to access saved tensors after calling backward.
```

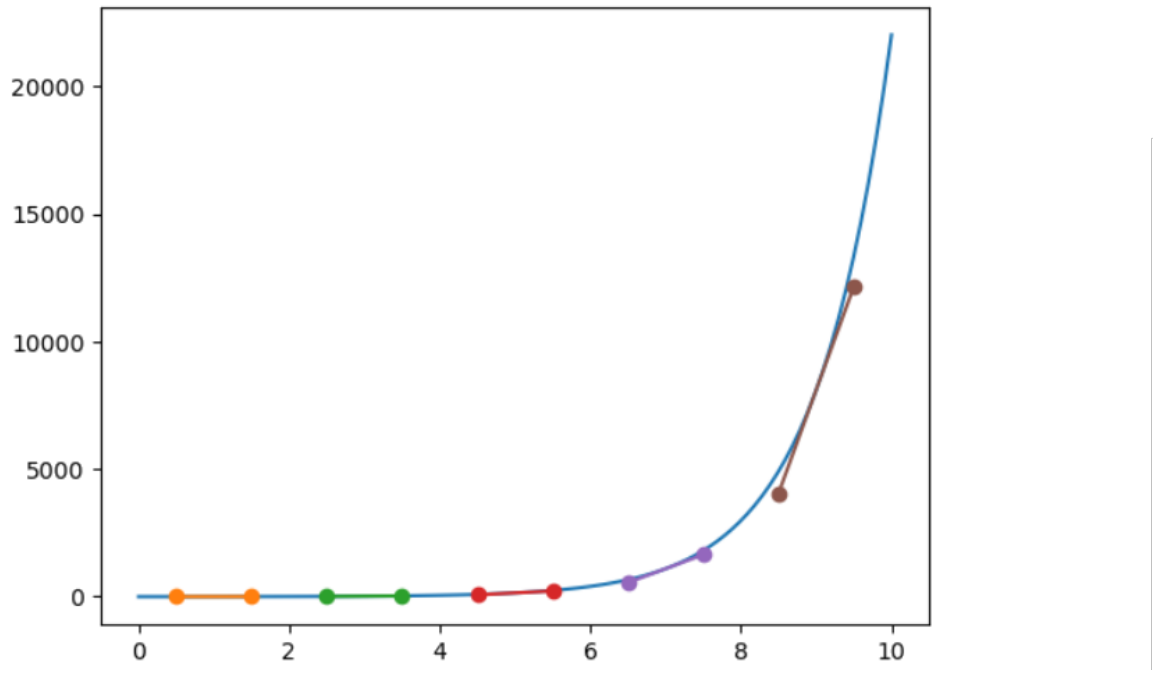
Pytorch Autograd : gradient accumulation

```
a = torch.tensor(1.0, requires_grad=True)
b = torch.tensor(2.0, requires_grad=True)
y = 10.0 / (1 + torch.exp(3*a + a*b))
y.backward(retain_graph=True) # reverse mode
print(a.grad) # dy/da
print(b.grad) # dy/db
y.backward()
print(a.grad) # dy/da
print(b.grad) # dy/db
```

```
tensor(-0.3324)
tensor(-0.0665)
tensor(-0.6648)
tensor(-0.1330)
```

Pytorch Autograd : back to the first lab in SI3

```
# plot exp fn
x = torch.linspace(0.0,10.0,100)
y = torch.exp(x)
plt.plot(x,y)

# plot tangent at each xi
xi = torch.arange(1.0,11.0,2.0)
xi.requires_grad_()
yi = torch.exp(xi)

# we explicitely compute the derivatives
yi.backward(torch.FloatTensor(np.ones(len(xi))))
dyi = xi.grad

x1 = xi-0.5
x2 = xi+0.5
y1 = dyi*(x1-xi) + dyi
y2 = dyi*(x2-xi) + dyi
plt.plot([x1.detach(), x2.detach()], [y1.detach(), y2.detach()], marker = 'o')
```



Simple gradient descent

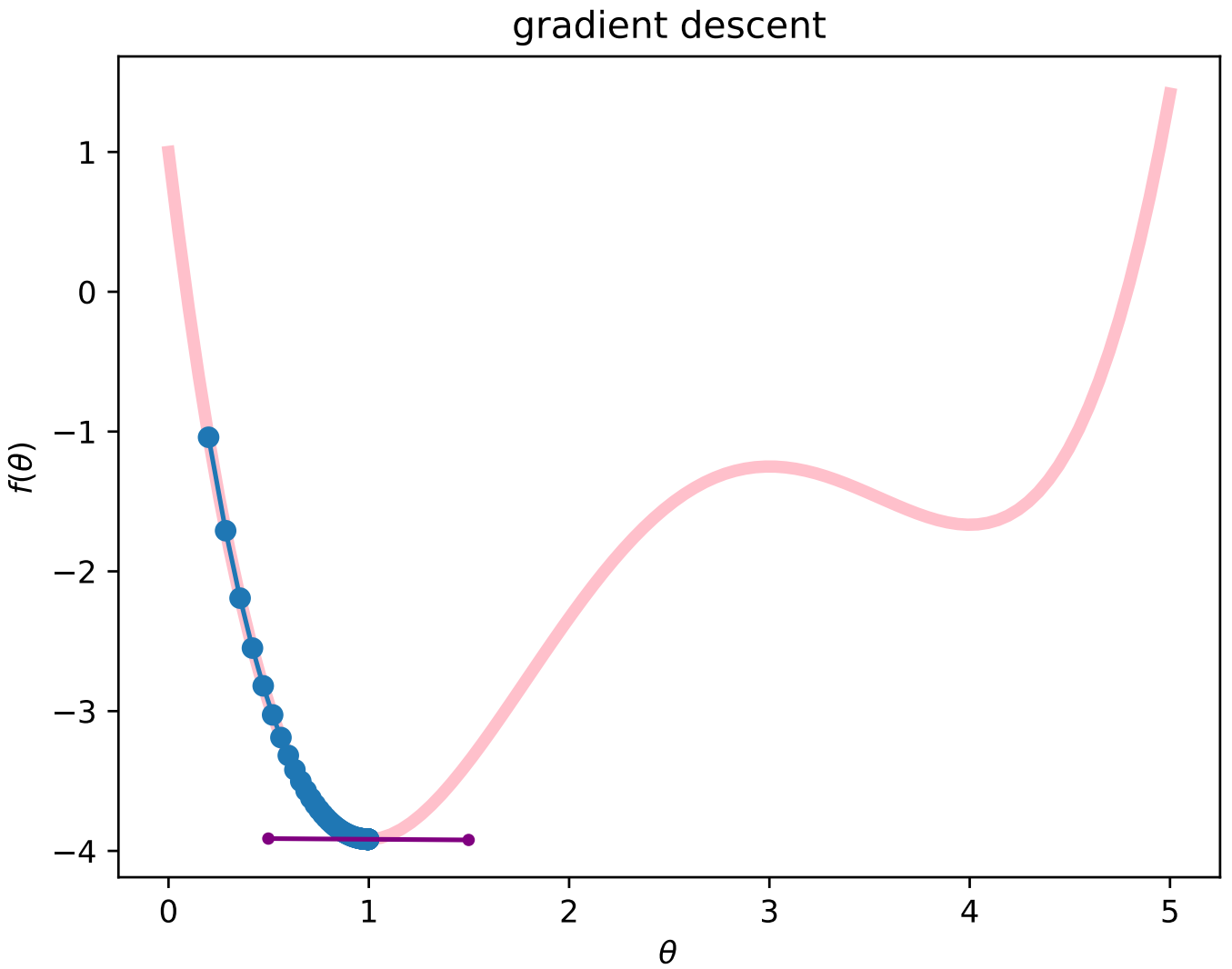
Imagine that we don't know the df fonction we had in the code on slide 13 :

```
def f(x):
    return x*x*x*x/4 -8*x*x*x/3 +19*x*x/2 -12*x +1
def df(x):
    return x*x*x -8*x*x +19*x -12

theta = 0.2
alpha = 0.01
limite= 100; epsilon = 0.01; cpt = 0
dyi = epsilon + 1

while (abs(dyi) > epsilon) and (cpt < limite):
    dyi = df(theta)
    theta = theta - alpha*dyi
    cpt += 1


print('min of f(x) obtained at iteration %d' %cpt)
print(' with x = %0.2f' %theta, end='')
print('and min f(x) = %0.2f'%f(theta))
```

```
min of f(x) obtained at iteration 92

with x = 1.00 and min f(x) = -3.92
```

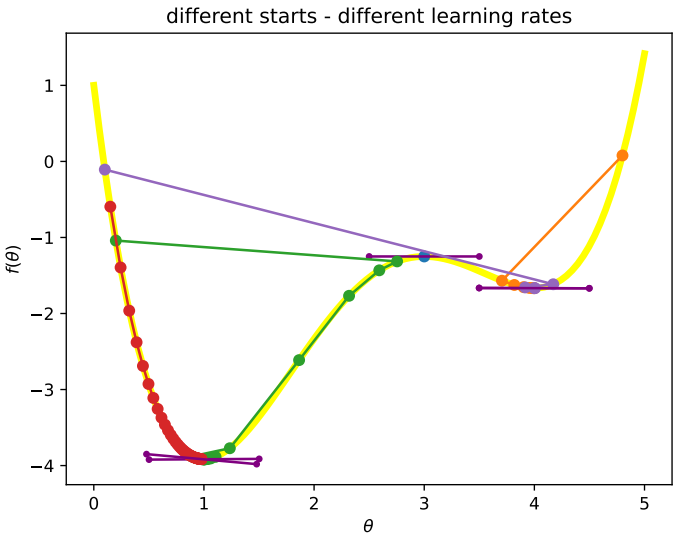
```
def f(x):
    return x*x*x*x/4 -8*x*x*x/3 +19*x*x/2 -12*x +1

theta = torch.tensor(0.2, requires_grad=True)
alpha = 0.01
limite = 100; epsilon = 0.01; cpt = 1

yi = f(theta)
yi.backward()
dyi = theta.grad
with torch.no_grad():
    theta -= alpha*dyi

while (abs(dyi) > 0.01) and (cpt < limite):
    theta.grad.zero_()
    yi = f(theta)
    yi.backward()
    dyi = theta.grad
    with torch.no_grad():
        theta -= alpha*dyi
    cpt += 1

print('min of f(x) obtained at iteration %d' %cpt)
print(' with x = %0.2f' %theta.item(), end='')
print(' and min f(x) = %0.2f'%f(theta).item())
```

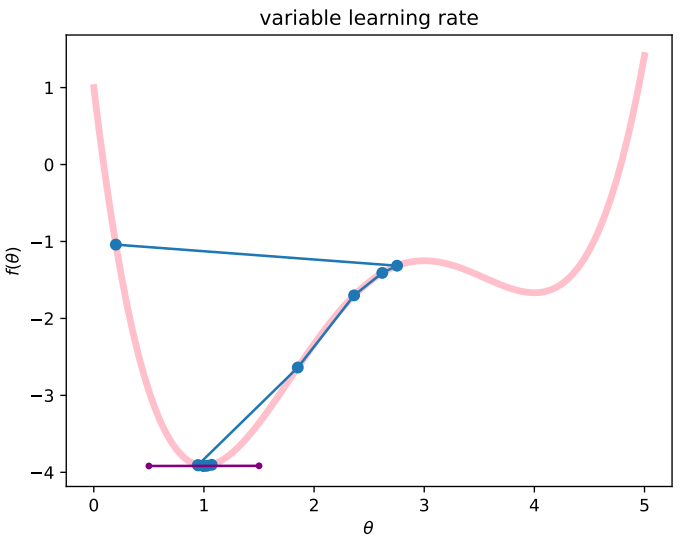
Simple gradient descent (2)



Improvements of gradient descent

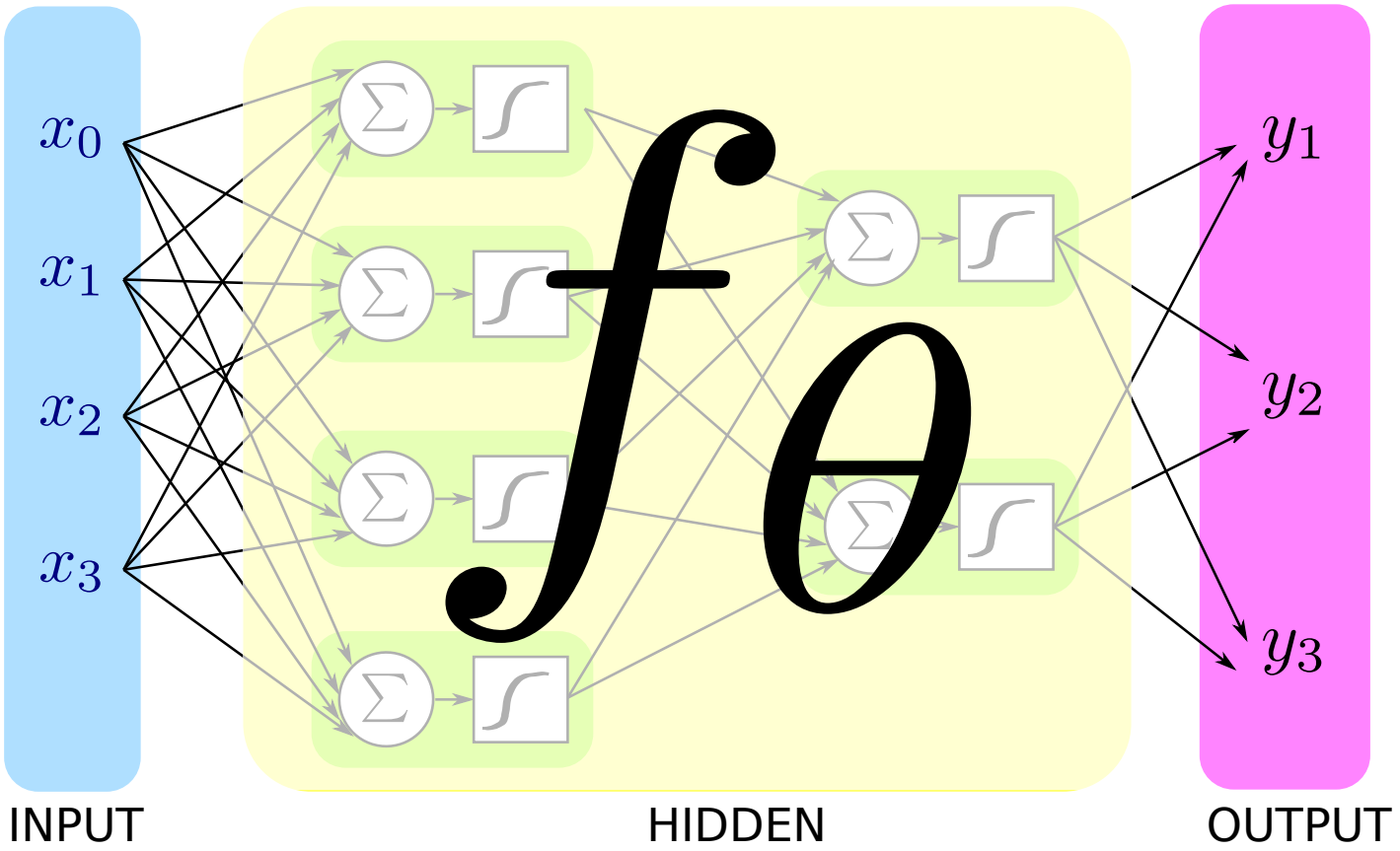
- different starts, different learning rates
- variable learning rate
  - if gradient in the same direction : increase lr
  - if gradient in opposite direction : decrease lr
- momentum : use the acceleration





10 iterations instead of 25 for the green curve

Neural Network as Universal function approximator



- Loss function minimisation
  - related to the problem to be solved
    - regression, classification ...
  - with respect to θ
  - using dataset $(x^j, y^j), j \in [0..m-1]$

TP aujourd'hui

- Découverte de pytorch
  - variables, fonctions ...
- Descente de gradient sur une fonction simple
  - la fonction f(x) des slides
  - une fonction de votre choix avec plusieurs minimums locaux
- Approfondissement de la descente de gradient :
  - différentes initialisations
  - différents taux d'apprentissage fixes
  - taux d'apprentissage variables
  - accélération (la variation de la variable utilise aussi la précédente variation)