

MODULE 1 : STRUCTURE DE DONNÉES PYTORCH

Agro-IODAA-Semestre 1

Vincent Guigue
vincent.guigue@agroparistech.fr

CONFIGURATION GÉNÉRALE



Un environnement performant mais dynamique

Le domaine de l'IA évolue beaucoup \Rightarrow les librairies aussi !!

- L'ajout d'un paquet peut mettre à jour des dépendances et engendrer des incompatibilités
- **Règle 1:** ne pas utiliser la dernière version de python
(rester sur une version d'il y a 6 mois/1 an. – pyth. 3.12 en 2025)
- **Règle 2:** travailler avec les environnements

```
1 >> conda create —name nom_de_votre_environnement python=3.12
2 >> conda activate nom_de_votre_environnement
3 (nom_de_votre_environnement) >>
```

- **Règle 3:** savoir ré-installer sa machine en cas de problème
(et/ou basculer très vite sur colab)

\Rightarrow en cas de doute, demander à chatGPT (ou équivalent)

Sauver / charger un environnement

Il y a plusieurs commandes pour sauvegarder son environnement (et pouvoir le transmettre ou le retrouver).

- Le plus simple est de travailler au niveau des packages avec `pip freeze`
- Il est possible de travailler au niveau des environnements complets avec `conda env export > environment.yml`

Pour les paquets: (1) sauvegarder les paquets (**avec leur version**)

```
1 pip freeze > requirements.txt
```

(2) installer tous les paquets du fichier (**dans les bonnes versions**)

```
1 pip install —r requirements.txt
```

CALCUL MATRICIEL



Matrices & philosophie générale

Pourquoi utiliser des matrices?

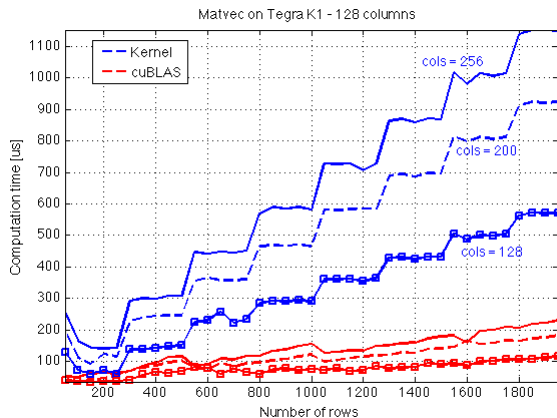
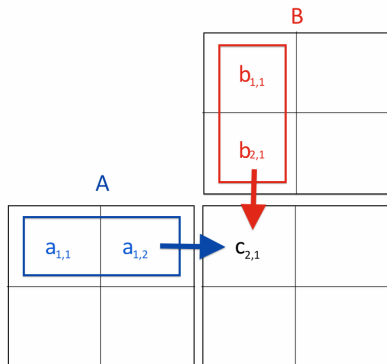
- Matrice = idéale pour stocker un ensemble de données
(e.g. ligne=individu, colonne=descripteur)
- Type des valeurs de base
 - Les matrices sont typées (bon à savoir)
 - `torch.int16`, `torch.int16` \Rightarrow Souvent, on cherche à gagner de la place!
 - Toujours avoir en tête un ordre de grandeur de la mémoire utilisée
- Philosophie = **opérateurs de haut niveau sur les matrices** \neq **boucles**
- Toutes les structures sont torch
 - Passage plus facile sur GPU
 - Calcul de gradient

Réflexions: Combien pèse un réel? Quantification des LLM? Combien de valeurs possibles sur 4 bits?



Parallélisation des calculs matriciels / GPU

- Pas le but premier du cours... Mais un intérêt évident en tant qu'utilisateur
- Attention au cout de transfert entre les mémoires



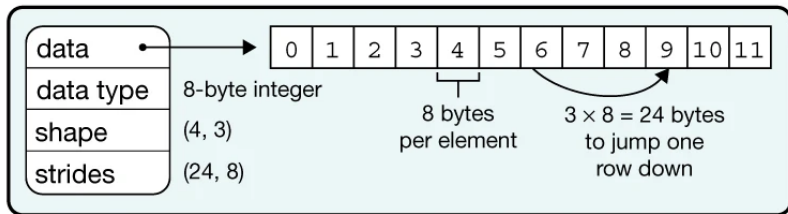


Quelques fonctions pour créer des matrices

```
1  import torch
2  # créer une matrice de 4 lignes & 3 colonnes
3  cst = torch.tensor([[3.1, 2.7, 1.9], [1.6, 0.0, 9.8], \
4                        [1.7, 8.1, 5.3]])
5  zeros = torch.zeros(4, 3)
6  ones = torch.ones(4, 3)
7  x = torch.rand(4, 3)
```

x =

0	1	2
3	4	5
6	7	8
9	10	11



```
1  zeros_like_x = torch.zeros_like(x)
```




Accéder aux valeurs

Penser aux motifs plutôt qu'aux boucles:

```
1 x = torch.tensor([[2, 4, 6], [3, 6, 9]])
2 print(x[0][0], " or ", x[0,0]) # note that each cell is a tensor i
3 print(x[:,0]) # all rows, first column
4 print(x[0,[0,2]]) # first row, column 0 and 2
5
6 # general syntax on rows or column start:end (excluded):step
7 print(x[0,0:2], " or ", x[0,:2]) # upper bound
8 print(x[0,1:3], " or ", x[0,1:]) # lower bound
9 print(x[0,0:3:2], " or ", x[0,::2]) # step = 2
```

	0	1	2
0	2	4	6
1	3	6	9



Calcul entre matrices

De nombreux opérateurs disponibles:

$+$, $-$, $*$, $/$, $**$

```
1 # avec des scalaires
2 x = torch.tensor([[2, 4, 6], [3, 6, 9]])
3 y = x+2 # [[4, 6, 8], [5, 8, 11]]
4
5 # entre matrice (terme a terme)
6 z = x * y # [[8, 24, 48], [15, 48, 99]]
7
8 # produit matriciel
9 a = x @ torch.tensor([[1], [2], [3]])
```



Inplace (ou pas)

Selon les situations, pour gagner de la place:

■ Opération classique & stockage dans un nouveau vecteur

```
1  b = torch.sin(a)
2  c = a+b
```

■ Opération *inline* & modification de l'argument

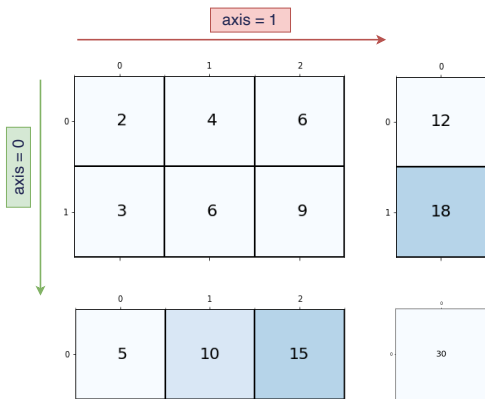
```
1  torch.sin_(a) # => modification de l'argument (a)
2  a.add_(b)     # => modification de l'objet a
```



Calcul agrégatifs dans matrice

```
1 x = torch.tensor([[2, 4, 6], [3, 6, 9]])
2 li = x.sum(1) # ou x.sum(axis=1)
3 co = x.sum(0)
4 tot = x.sum()
5 # somme des colonnes 0 et 2
6 li = x[:, ::2].sum(1)
```

- somme, moyenne, min, max, ...
- ... et on ne s'arrête pas à deux axes !





squeeze / unsqueeze

- Ligne de données = 1d; Image = 2d; corpus d'image = 3d
- Donner une image en entrée du système \Rightarrow unsqueeze
Dimension homogène à un batch
- Récupérer (& analyser) une sortie intermédiaire \Rightarrow squeeze
batch \Rightarrow individu

```
1 a = torch.rand(3, 226, 226)
2 b = a.unsqueeze(0)
3
4 print(a.shape) # torch.Size([3, 226, 226])
5 print(b.shape) # torch.Size([1, 3, 226, 226])
```

GRADIENT



Tensor: données... Et gradient

■ La structure tensor intègre aussi un champ grad

```
1 a = torch.tensor(1.)
2 a.requires_grad = True # activation du champ gradient
3 b = torch.tensor(2., requires_grad=True)
4 z = 2*a + b
5 # Calcul des derivees partielles par rapport a z
6 z.backward()
7 print("Derivee de z/a : ", a.grad.item(), " z/b : ", b.grad.item())
```

■ z : connecté à a et b = graphe de Calcul

■ $z \approx$ fonction de coût \Rightarrow le gradient nous donne la direction pour la minimiser:

Dérivée de z/a : 2.0 z/b : 1.0



Gradient sur un vecteur

- Evidemment, les tensor sont en général des vecteurs, matrices... Mais pas des tenseurs
- Le gradient a la même dimension que le vecteur d'origine:

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots \\ a_{21} & a_{22} & \cdots \end{pmatrix}, \quad L = fct(A)$$

- L doit être un scalaire
- $L.backward() \Rightarrow \nabla_A L$, A : variables sur lesquelles le **gradient est actif**

$$A.grad = \begin{pmatrix} \frac{\partial L}{\partial a_{11}} & \frac{\partial L}{\partial a_{12}} & \cdots \\ \frac{\partial L}{\partial a_{21}} & \frac{\partial L}{\partial a_{22}} & \cdots \end{pmatrix}$$



Syntaxe

Attention à la syntaxe une fois le gradient activé

```
1 b = torch.tensor(2., requires_grad=True)
2 # print(b) # => ERR: ambigu!
3 b.data # ok
4 b.grad.data # ok
```

Plus efficace en coupant le graphe de calcul
e.g pour la mise à jour des paramètres:

```
1 # Logistic regression based on w on b
2 [...]
3 # update parameters:
4 with torch.no_grad():
5     w += -EPS*w.grad
6     b += -EPS*b.grad
```



Accumulation du gradient

- Appels multiples à backward
- A partir d'une fonction de coût ou de plusieurs

⇒ Accumulation des gradients

```
1 a = torch.tensor(1., requires_grad=True)
2 b = torch.tensor(2., requires_grad=True)
3 z = 2*a + b
4 z.backward()
5 z = 2*a + b # il faut redefinir le cout
6 z.backward() # puis relancer le gradient = accumulation
```

Mettre à 0 les gradients à la main:

```
1 a.grad.zero_()
2 b.grad.zero_()
```



Accumulation du gradient

- Appels multiples à backward
- A partir d'une fonction de coût ou de plusieurs

⇒ Accumulation des gradients

```
1 a = torch.tensor(1., requires_grad=True)
2 b = torch.tensor(2., requires_grad=True)
3 z = 2*a + b
4 z.backward()
5 z = 2*a + b # il faut redefinir le cout
6 z.backward() # puis relancer le gradient = accumulation
```

Mettre à 0 les gradients à la main:

```
1 a.grad.zero_()
2 b.grad.zero_()
```



Fonctions (d'activation)

- Seule option pour que les gradients se calculent automatiquement dans les fonctions... Utiliser des fonctions torch !!

```
1 a = torch.tensor(1., requires_grad=True)
2 b = torch.tensor(2., requires_grad=True)
3 # Exemple:
4 z = torch.sin(2*a + b)
5 z.backward()
```

La plupart des fonctions existent dans l'univers pytorch

REGRESSION LINÉAIRE



Quels verrous pour une régression linéaire?

- 1 Lire les données \Rightarrow utilisation de `scikit learn`
- 2 Construire l'estimateur linéaire
- 3 Appliquer la fonction coût
- 4 Trouver un critère d'arrêt pour la descente de gradient
- 5 Evaluer les performances



Accès aux données

- Récupération des données
- Transformation en tensor
 - Vérification des dimensions de la structure créée

```
1 from sklearn.datasets import fetch_california_housing
2 housing = fetch_california_housing() ## chargement des donnees
3
4 # penser a typer les donnees pour eliminer les incertitudes
5 X = torch.tensor(housing['data'], dtype=torch.float)
6 Y = torch.tensor(housing['target'], dtype=torch.float)
```



tous les types sont spécifiques

Structure de données

`torch.tensor ...`

Typage élémentaire des données

`torch.float, torch.int, ...`



Estimateur linéaire

■ Déclarer les paramètres en activant le gradient

```
1 w = torch.randn(1,X.size(1),requires_grad=True)
2 b = torch.randn(1,1,requires_grad=True)
```

■ Construire un estimateur linéaire

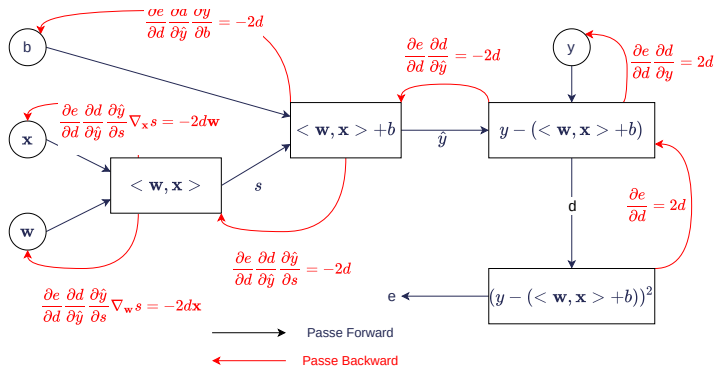
```
1 (x @ w.T)+b
```

■ Construire la fonction de cout

■ ... Et c'est (presque) tout



Autograd et Graphe de calcul



Graphe de calcul

- Graphe orienté, décrit l'enchaînement des opérations de calcul
- *Source* = variable d'entrée + nœud de sortie : le résultat du calcul
- En connaissant les dérivées de chaque opération, le graphe permet de calculer les gradient de la sortie par rapport à chaque variable d'entrée.



Autres verrous

- Pas de critère d'arrêt pour l'instant (5000 itérations?)
- Calculer la performances
- Ne pas oublier de mettre à 0 le gradient pour éviter l'accumulation après la mise à jour

CONCLUSION



Ce que nous verrons dans ce module (Travaux Pratiques)

PyTorch, c'est ...

- Framework de dévelop. + apprentissage de réseaux Deep sur CPU et GPU
- Architecture modulaire de contenants + conteneurs \Rightarrow Architectures flexibles
- Différenciation automatique \Rightarrow **Autograd**
- Couche d'abstraction pour l'optimisation \Rightarrow variété de descentes de gradient
- Gestion simplifiée des données pour la constitution des mini-batches

PyTorch vs TensorFlow

- PyTorch (un peu) moins intégré dans l'industrie
- Déploiement, rapidité et processus industriel en faveur de TensorFlow
- Flexibilité, prototyping, simplicité en faveur de PyTorch

Les deux frameworks ont tendance à se rapprocher en termes de fonctionnalités ces derniers temps.