MODULE 1 : STRUCTURE DE DONNÉES PYTORCH

Agro–IODAA–Semestre 1

Vincent Guigue

MIA PARIS-SACLAY EKINOCS

AGROPARISTECH Institut des Sciences et Industries du Vivant et de l'Environnement

Calcul matriciel

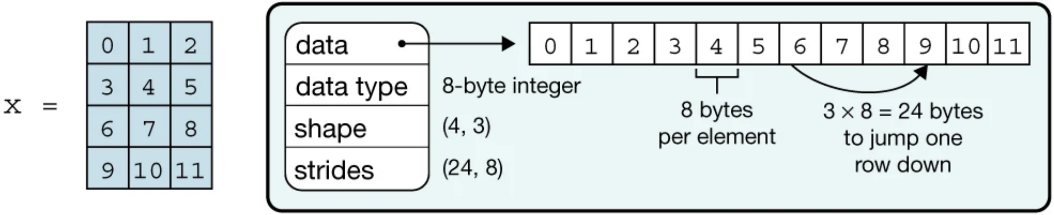
Matrices & philosophie générale

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` $\Rightarrow$ Le plus souvent, on cherche à gagner de la place!
- Philosophie = **opérateurs de haut niveau** sur les matrices $\neq$ boucles
- Toutes les structures sont `torch`
  - Passage plus facile sur GPU
  - Calcul de gradient

Quelques fonctions pour créer des matrices

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



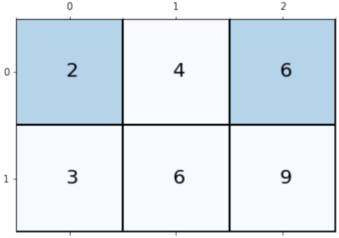

```
zeros_like_x = torch.zeros_like(x)
```

Accéder aux valeurs

Penser aux motifs plutôt qu'aux boucles:

```
x = torch.tensor([[2, 4, 6], [3, 6, 9]])
print(x[0][0], " or ", x[0,0])  # note that each cell is a tensor i
print(x[:,0])                   # all rows, first column
print(x[0,[0,2]])               # first row, column 0 and 2

# general syntax on rows or column  start:end (excluded):step
print(x[0,0:2], " or ", x[0,:2])    # upper bound
print(x[0,1:3], " or ", x[0,1:])    # lower bound
print(x[0,0:3:2], " or ", x[0,::2]) # step = 2
```



Calcul entre matrices

De nombreux opéprateurs disponibles:

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

```python
# avec des scalaires
x = torch.tensor([[2, 4, 6], [3, 6, 9]])
y = x+2 # [[4, 6, 8], [5, 8, 11]]

# entre matrice (terme a terme)
z = x * y # [[8, 24, 48], [15, 48, 99]]

# produit matriciel
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

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

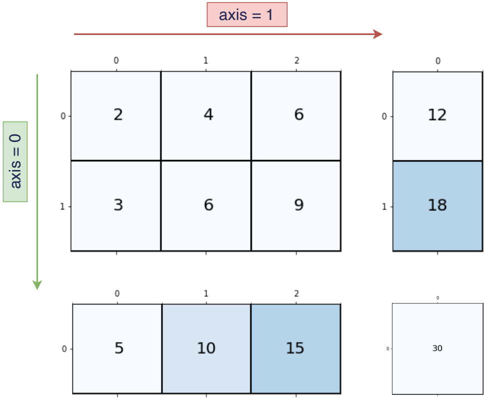
- Opération *inline* & modification de l'argument

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

Calcul agrégatifs dans matrice

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

- somme, moyenne, min, max, ...



squeeze / unsqueeze

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

```python
a = torch.rand(3, 226, 226)
b = a.unsqueeze(0)

print(a.shape) # torch.Size([3, 226, 226])
print(b.shape) # torch.Size([1, 3, 226, 226])
```

Gradient

Tensor: données... Et gradient

- La structure `tensor` intègre aussi un champ `grad`

```python
a = torch.tensor(1.)
a.requires_grad = True # activation du champ gradient
b = torch.tensor(2., requires_grad=True)
z = 2*a + b
# Calcul des derivees partielles par rapport a z
z.backward()
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:

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

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

Syntaxe

Attention à la syntaxe une fois le gradient activé

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

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

```
# Logisitic regression based on w on b
[...]
# update parameters:
with torch.no_grad():
    w += -EPS*w.grad
    b += -EPS*b.grad
```

Accumulation du gradient

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

$\Rightarrow$ Accumulation des gradients

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

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'arret 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

```python
from sklearn.datasets import fetch_california_housing
housing = fetch_california_housing() ## chargement des donnees

# penser a typer les donnees pour eliminer les incertitudes
housing_x = torch.tensor(housing['data'],dtype=torch.float)
housing_y = torch.tensor(housing['target'],dtype=torch.float)
```

Estimateur linéaire

- Déclarer les paramètres en activant le gradient

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

- Construire un estimateur linéaire

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

Autres verrous

- Pas de critère d'arret 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