

Apprentissage profond par renforcement (*Deep RL*)


Laëtitia Matignon

5A - Option Ouverture à la recherche SMA

Apprentissage profond par renforcement (*Deep RL*)

- Apprentissage par renforcement : vous vous souvenez ? ...
- Réseaux de neurones : vous vous souvenez ? ...

- Apprentissage automatique : *hot topic* en recherche ... aussi en entreprise ?
- Deep (RL) : *hot topic* en recherche
- Framework d'apprentissage profond : **PyTorch**, Tensorflow, ...



International Joint Conference
on Artificial Intelligence
August 10-16, 2019

Accepted papers, by area

number of papers with at least one keyword in the area:

- Machine learning: 2516 submitted, accepted 438
- Computer vision: 833 submitted, 117 accepted
- Machine learning applications: 785 submitted, 144 accepted
- Multi-agent systems: 518 submitted, 121 accepted
- Natural language processing: 630 submitted, 103 accepted
- Knowledge representation: 349 submitted, 88 accepted
- Humans and AI: 280 submitted, 50 accepted
- Planning and scheduling: 217 submitted, 48 accepted
- Search and game playing: 222 submitted, 43 accepted
- Uncertainty in AI: 185 submitted, 43 accepted
- Constraints and satisfiability: 128 submitted, 30 accepted
- Robotics: 127 submitted, 20 accepted
- Multidisciplinary Topics and Applications: 617 submitted, 119 accepted

CM1 -Rappels réseaux de neurones et Pytorch

Laëtitia Matignon

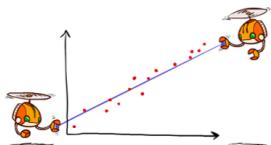
5A - Option Ouverture à la recherche SMA

- 1 Introduction
- 2 Rappels : descente de gradient & régression linéaire (NumPy)
- 3 Régression linéaire en PyTorch
- 4 Modèle/NN avec PyTorch
- 5 Application : classification supervisée

- 1 Introduction
- 2 Rappels : descente de gradient & régression linéaire (NumPy)
- 3 Régression linéaire en PyTorch
- 4 Modèle/NN avec PyTorch
- 5 Application : classification supervisée

Contenu 1er cours

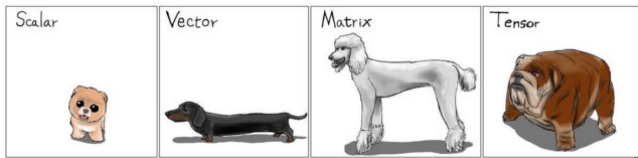
- Rappels sur l'apprentissage supervisé : descente de gradient sur un problème de régression linéaire avec NumPy
- Concepts de base de **PyTorch** illustrés sur un problème de régression linéaire
- Modèle/NN avec PyTorch
- A vous de travailler ! : application des concepts PyTorch sur un problème de **classification supervisée**



 PyTorch

Tenseurs : généralisation des matrices

Tableaux multi-dimensionnels avec éléments tous du même type (vecteur = Tenseur 1D, matrice = Tenseur 2D, ...)



't'
'e'
'n'
's'
'o'
'r'

tensor of dimensions [6]
(vector of dimension 6)

3	1	4	1
5	9	2	6
5	3	5	8
9	7	9	3
2	3	8	4
6	2	6	4

tensor of dimensions [6,4]
(matrix 6 by 4)

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

tensor of dimensions [4,4,2]

Tenseurs : généralisation des matrices

NumPy

- bibliothèque pour opérations d'algèbre linéaire
- manipulation de *nd-array*

```
import numpy as np
# initialisation de la graine du
# generateur
np.random.seed(40)
# 2 x 3 ndarray avec nombres aleatoires
# N(0,1)
x = np.random.randn(2,3)
print(x)
print(type(x), x.shape)
```

```
[[ -0.10091345  1.85968363  1.00008287]
 [ -2.14836824  1.39821122  0.34256522]]
<class 'numpy.ndarray'> (2, 3)
```

PyTorch

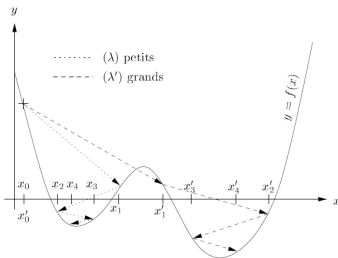
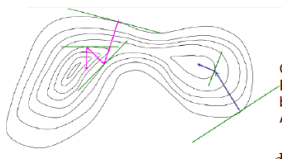
- bibliothèque de calculs tensoriels
- manipulation de *tensor*
- calcul automatique de gradients
- calculs sur CPU ou GPU

```
import torch
# initialisation de la graine du
# generateur
torch.manual_seed(40)
# 2 x 3 Tensor avec nombres aleatoires N
# (0,1)
x = torch.randn(2,3)
print(x)
print(type(x), x.size())
```

```
tensor([[0.8529, 0.0279, 0.2130],
        [0.5421, 0.1813, 0.9069]])
<class 'torch.Tensor'>
torch.Size([2,3])
```


- 1 Introduction
- 2 Rappels : descente de gradient & régression linéaire (NumPy)
- 3 Régression linéaire en PyTorch
- 4 Modèle/NN avec PyTorch
- 5 Application : classification supervisée

Descente de gradient



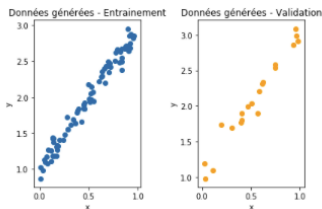
Algorithme itératif pour trouver le minimum local d'une fonction

- A t on part du point \vec{x}_t
- Le gradient de f , noté $\vec{\nabla} f(\vec{x}_t)$, indique la direction de plus grande pente de f
- Le gradient de f est le vecteur des dérivées partielles de f par rapport à ses paramètres :

$$\vec{\nabla} f(x_1, \dots, x_K) = \left(\frac{\partial f(x_1, \dots, x_K)}{\partial x_1}, \frac{\partial f(x_1, \dots, x_K)}{\partial x_2}, \dots \right)^T$$

- On suit le vecteur donné par le gradient de f en \vec{x}_t
- On passe à $\vec{x}_{t+1} = \vec{x}_t - \lambda \vec{\nabla} f(\vec{x}_t)$ $\lambda > 0$ pas d'apprentissage permet de progresser rapidement au début (en sortant si possible de zones d'optima locaux) puis d'affiner le résultat par des pas plus petits.

Problème de régression linéaire 1D

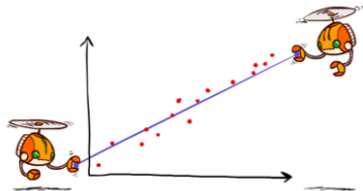


```
# Data Generation
np.random.seed(40)
x = np.random.rand(100, 1)
y = 1 + 2 * x + .1 * np.random.randn(100, 1)
# Shuffles the indices
idx = np.arange(100)
np.random.shuffle(idx)
# train set
train_idx = idx[:80]
x_train, y_train = x[train_idx], y[train_idx]
```

Apprentissage supervisé : données labellisées

- données en entrée (**feature**) labellisées : ensemble de N exemples d'entraînement (x_k, y_k)
- feature* 1D : $y = a + bx$

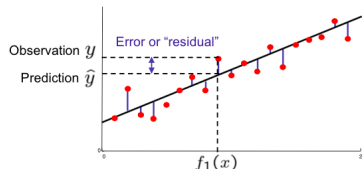
Problème de régression linéaire 1D



Apprentissage supervisé : objectif

- f_{θ} fonction cible (**modèle**) à apprendre ($y = f_{\theta}(x) = a + bx$)
- on cherche le vecteur de **paramètres** $\vec{\theta} = [a, b]$ qui permet de *prédire* y à partir de x .

Descente de gradient



```
# x_train : ndarray (80,1)
# Computes our model's predicted output
y_hat = a + b * x_train
# Computes the error
error = (y_train - y_hat)
# MSE
loss = (error ** 2).mean()
```

A partir d'un ensemble d'échantillons (x_k, y_k) et des prédictions du modèle $\hat{y}_k = a + bx_k$

1- Calcul de la *loss*

Mean Squared Error

$$MSE = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)^2$$

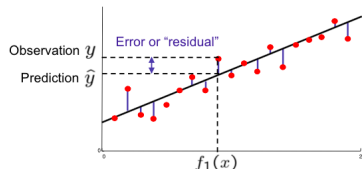
$$MSE = \frac{1}{N} \sum_{i=1}^N (y_i - a - bx_i)^2$$

Batch vs stochastic vs mini-batch gradient descent

Pour le calcul de la *loss* :

- **batch** gradient descent : utilise tous les échantillons d'entraînement (N)
- **stochastic** gradient descent : utilise un seul échantillon
- **mini-batch** gradient descent : utilise $n \in]1; N[$ échantillons

Descente de gradient



```
# Computes the error
error = (y_train - yhat)
# MSE
loss = (error ** 2).mean()
# Computes gradients for both "a" and "b" parameters
a_grad = -2 * error.mean()
b_grad = -2 * (x_train * error).mean()
```

A partir d'un ensemble d'échantillons (x_k, y_k) et des prédictions du modèle $\hat{y}_k = a + bx_k$

1- Calcul de la *loss*

Mean Squared Error

$$MSE = \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)$$

$$MSE = \frac{1}{N} \sum_{i=1}^N (y_i - a - bx_i)$$

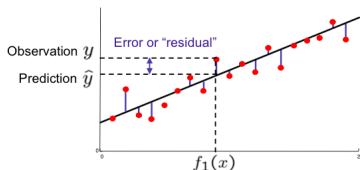
2- Calcul du gradient de la *loss*

Dérivées partielles de la *loss* par rapport aux paramètres a, b :

$$\frac{\partial MSE}{\partial a} = \frac{\partial MSE}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial a} = -2 \frac{1}{N} \sum_{i=1}^N (y_i - \hat{y}_i)$$

$$\frac{\partial MSE}{\partial b} = \frac{\partial MSE}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial b} = -2 \frac{1}{N} \sum_{i=1}^N x_i (y_i - \hat{y}_i)$$

Descente de gradient



```
# Computes the error
error = (y_train - yhat)
# MSE loss
loss = (error ** 2).mean()
# Computes gradients for both "a" and "b" parameters
a_grad = -2 * error.mean()
b_grad = -2 * (x_train * error).mean()
# Updates parameters using gradients and learning
rate
a = a - eta * a_grad
b = b - eta * b_grad
```

A partir d'un ensemble d'échantillons (x_k, y_k) et des prédictions du modèle $\hat{y}_k = a + bx_k$

3- Mise à jour des paramètres (une itération)

$\eta \in [0; 1]$ learning rate :

$$a = a - \eta \frac{\partial MSE}{\partial a}$$

$$b = b - \eta \frac{\partial MSE}{\partial b}$$

Descente de gradient

4- Itération sur plusieurs epochs

1 epoch : tous les échantillons d'entraînement ont été utilisés pour la mise à jour des paramètres :

- **batch** gradient descent : 1 epoch = 1 update
- **stochastic** gradient descent : 1 epoch = N updates
- **mini-batch** gradient descent : 1 epoch = N/n updates

```
# Defines number of epochs
n_epochs = 1000

for epoch in range(n_epochs):
    # Computes our model's predicted output
    yhat = a + b * x_train

    # Computes the error
    error = (y_train - yhat)
    # MSE loss
    loss = (error ** 2).mean()

    # Computes gradients for both "a" and "b"
    # parameters
    a_grad = -2 * error.mean()
    b_grad = -2 * (x_train * error).mean()

    # Updates parameters using gradients and
    # learning rate
    a = a - lr * a_grad
    b = b - lr * b_grad
```


Conclusion

Résumé : Init + 4 étapes par epoch

Initialisation des paramètres et des hyper-paramètres

- ➊ *forward pass* : calcul des **prédictions** du modèle courant
- ➋ Calcul de la *loss*, en utilisant les **prédictions**, **labels** et la **fonction de perte** appropriée à la tâche
- ➌ Calcul du **gradient** - dérivées partielles pour chaque paramètre
- ➍ Mise à jour des paramètres

```
# Random initialization of parameters
np.random.seed(42)
a = np.random.randn(1)
b = np.random.randn(1)

# Initialization of hyper-parameters
# Sets learning rate
lr = 1e-1
# Defines number of epochs
n_epochs = 1000

for epoch in range(n_epochs):
    # 1- forward pass: Computes our model's
    # predicted output
    yhat = a + b * x_train

    # 2- Computes the error
    error = (y_train - yhat)
    # MSE loss
    loss = (error ** 2).mean()

    # 3- Computes gradients for both "a" and
    # "b" parameters
    a_grad = -2 * error.mean()
    b_grad = -2 * (x_train * error).mean()

    # 4- Updates parameters using gradients
    # and learning rate
    a = a - lr * a_grad
    b = b - lr * b_grad
```

- 1 Introduction
- 2 Rappels : descente de gradient & régression linéaire (NumPy)
- 3 Régression linéaire en PyTorch**
- 4 Modèle/NN avec PyTorch
- 5 Application : classification supervisée

NumPy *ndarray* to/from PyTorch *tensor*

- `from_numpy` : transforme *ndarray* en *tensor*

```
import torch

# Numpy arrays transformed into PyTorch's Tensors
# and cast them into lower precision
x_train_tensor = torch.from_numpy(x_train).float()
y_train_tensor = torch.from_numpy(y_train).float()

print(type(x_train), type(x_train_tensor))
```

```
<class 'numpy.ndarray'> <class 'torch.Tensor'>
```

- `numpy()` : transforme *tensor* en *ndarray*

```
x_ndarray = x_train_tensor.numpy()
print(type(x_ndarray))
```

```
<class 'numpy.ndarray'>
```

Création de *tensor* (pour des données)

```
x = torch.rand(3,2)
print(x)
print("x[0]", x[0])
print("x[0][0]", x[0][0])
```

```
tensor([[0.0324, 0.1923], [0.6838, 0.1085], [0.7307, 0.6133]])
x[0] tensor([0.0324, 0.1923])
x[0][0] tensor(0.0324)
```

```
# Resizing
y = x.view(1,-1)
```

```
tensor([[0.0324, 0.1923, 0.6838, 0.1085, 0.7307, 0.6133]])
```

```
x.resize_(2, 3)
```

```
tensor([[0.0324, 0.1923, 0.6838], [0.1085, 0.7307, 0.6133]])
```

Méthodes *in_place*

- Les méthodes qui se terminent par `_` modifient la variable (`resize_`, `add_`, ...).

Création de *tensors* entraînables (pour des paramètres/poids)

Pour préciser à PyTorch qu'il doit **calculer des gradients** par rapport à des tenseurs :

- utiliser l'argument `requires_grad=True` à la création du tenseur
- modifier l'attribut `requires_grad_(True)` à tout moment

```
torch.manual_seed(42)
# Random initialization of parameters
a = torch.randn(1, requires_grad=True, dtype=torch.float)
b = torch.randn(1, requires_grad=True, dtype=torch.float)

print(a)
print(b)

a.requires_grad_(True)
b.requires_grad_(False)
```

```
tensor([0.3367], requires_grad=True)
```

```
tensor([0.1288], requires_grad=True)
```

Autograd (*automatic differentiation*)

module autograd

- Permet le calcul automatique de gradients par rapport à un tenseur
- Mémoire toutes les opérations réalisées sur un tenseur (**graph de calcul dynamique**)

Calcul du gradient / dérivé partielle de la *loss* selon chaque paramètre ($\frac{\partial MSE}{\partial a}, \frac{\partial MSE}{\partial b}$) :

- mettre `requires_grad=True` à la création des paramètres (tenseurs)
- invoquer `backward()` sur la *loss*
- remettre les gradients à zéro (ils s'accumulent à chaque appel)

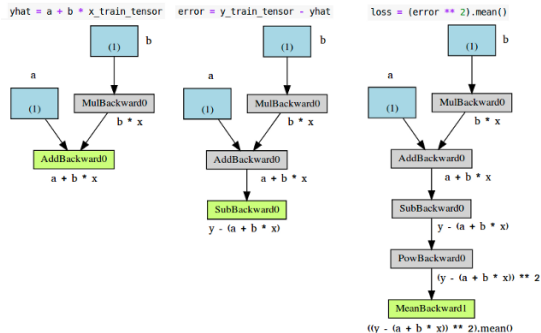
```
# Random initialization of parameters
torch.manual_seed(42)
a = torch.randn(1, requires_grad=True,
                dtype=torch.float)
b = torch.randn(1, requires_grad=True,
                dtype=torch.float)
# Initialization of hyper-parameters
lr = 1e-1
n_epochs = 1000

for epoch in range(n_epochs):
    # 1- forward pass: Computes our model's
    # predicted output
    yhat = a + b * x_train_tensor
    # 2- Computes the error and MSE loss
    error = y_train_tensor - yhat
    loss = (error ** 2).mean()
    # 3- Computes gradients for both "a" and "
    # b" parameters
    loss.backward()

    # —> No more manual computation of
    # gradients!
    # a_grad = -2 * error.mean()
    # b_grad = -2 * (x_tensor * error).mean()

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

Graph de calcul dynamique



```
from torchviz import make_dot

torch.manual_seed(42)
a = torch.randn(1, requires_grad=True, dtype=torch.float)
b = torch.randn(1, requires_grad=True, dtype=torch.float)

yhat = a + b * x_train_tensor
error = y_train_tensor - yhat
make_dot(yhat)
make_dot(error)
loss = (error ** 2).mean()
make_dot(loss)
loss.backward()
```

- rectangle bleu : tenseurs sur lesquels on demande à PyTorch de calculer les gradients
- rectangle gris : opérations PyTorch qui impliquent un calcul de gradient
- rectangle vert : comme ci-dessus mais point de départ pour le calcul de gradient si `backward()` est appelé depuis la variable utilisée pour afficher le graph

Graph de calcul dynamique

```

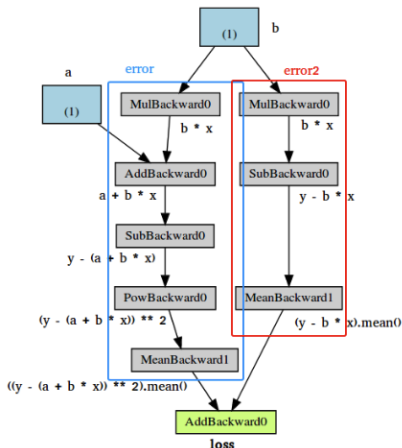
yhat = a + b * x_train_tensor
error = y_train_tensor - yhat

loss = (error ** 2).mean()

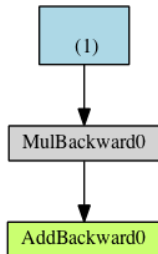
if loss > 0:
    yhat2 = b * x_train_tensor
    error2 = y_train_tensor - yhat2

    loss += error2.mean()

```



Graph de calcul dynamique



```
from torchviz import make_dot

torch.manual_seed(42)
a = torch.randn(1, requires_grad=False,
                 dtype=torch.float)
b = torch.randn(1, requires_grad=True, dtype
                 =torch.float)

yhat = a + b * x_train_tensor
make_dot(yhat)
```

- seuls les tenseurs sur lesquels un calcul de gradient doit être fait sont affichés

Autograd (*automatic differentiation*)

module autograd

- Après l'appel à `backward()`, le gradient par rapport à un tenseur est dans l'attribut `.grad` du tenseur
- `.grad_fn` renvoie l'opération qui a créé le tenseur

```
<AddBackward0 object at 0x120474f70>
<SubBackward0 object at 0x120474f70>
<MeanBackward0 object at
0x120474f70>
dMSE/da tensor([-3.1019])
dMSE/db tensor([-1.8132])
```

```
# Random initialization of parameters
torch.manual_seed(42)
a = torch.randn(1, requires_grad=True,
                dtype=torch.float)
b = torch.randn(1, requires_grad=True,
                dtype=torch.float)

# Initialization of hyper-parameters
lr = 1e-1
n_epochs = 1000

for epoch in range(n_epochs):
    # 1- forward pass: Computes our model's
    # predicted output
    yhat = a + b * x_train_tensor
    # 2- Computes the error and MSE loss
    error = y_train_tensor - yhat
    loss = (error ** 2).mean()

    print(yhat.grad_fn)
    print(error.grad_fn)
    print(loss.grad_fn)

    # 3- Computes gradients for both "a" and "
    # b" parameters
    loss.backward()
    # affichage du gradient calcule par
    # rapport au parametre a
    print("dMSE/da", a.grad)
    print(dMSE/db", b.grad)
```

Autograd (*automatic differentiation*)

Mise à jour des paramètres

Attention à ne pas "perdre" le gradient !

- en ré-assignant les paramètres, le gradient du tenseur de départ sont perdus

```
dmSE/da avant tensor([-3.1019])
```

```
dmSE/da apres None
```

```
AttributeError: 'NoneType' object  
has no attribute 'zero_'
```

```
for epoch in range(n_epochs):
    # 1- forward pass: Computes our model's
    # predicted output
    yhat = a + b * x_train_tensor
    # 2- Computes the error and MSE loss
    error = y_train_tensor - yhat
    loss = (error ** 2).mean()

    # 3- Computes gradients for both "a" and "
    # b" parameters
    loss.backward()

    # 4- Update parameters using gradient
    print("dMSE/da_avant_", a.grad)
    a = a - lr * a.grad
    b = b - lr * b.grad
    print("dMSE/da_apres_", a.grad)

    # gradients are accumulated: zero the
    # gradients
    a.grad.zero_()
    b.grad.zero_()
```

Autograd (*automatic differentiation*)

Mise à jour des paramètres

Attention à ne pas "perdre" le gradient !

- en ré-assignant les paramètres
- en utilisant une opération *in-place* sur un paramètre

`RuntimeError: a leaf Variable that requires grad has been used in an in-place operation.`

```
for epoch in range(n_epochs):
    # 1- forward pass: Computes our model's
    #      predicted output
    yhat = a + b * x_train_tensor
    # 2- Computes the error and MSE loss
    error = y_train_tensor - yhat
    loss = (error ** 2).mean()

    # 3- Computes gradients for both "a" and "
    #      b" parameters
    loss.backward()

    # 4- Update parameters using gradient
    a -= lr * a.grad
    b -= lr * b.grad

    # gradients are accumulated: zero the
    #      gradients
    a.grad.zero_()
    b.grad.zero_()
```

Autograd (*automatic differentiation*)

Mise à jour des paramètres

Réaliser des opération sur des tenseurs du graph de calcul en désactivant temporairement le graph pour ne pas calculer de gradient :

- `torch.no_grad()` autour d'un bloc de code

```
tensor([0.6469], requires_grad=True)
dMSE/da apres tensor([-3.1019])
```

```
for epoch in range(n_epochs):
    # 1- forward pass: Computes our model's
    #       predicted output
    yhat = a + b * x_train_tensor
    # 2- Computes the error and MSE loss
    error = y_train_tensor - yhat
    loss = (error ** 2).mean()

    # 3- Computes gradients for both "a" and "
    #       b" parameters
    loss.backward()

    # 4- Update parameters using gradient
    with torch.no_grad():
        a -= lr * a.grad
        b -= lr * b.grad

    print(a)
    print("dMSE/da apres_", a.grad)

    # gradients are accumulated: zero the
    #       gradients
    a.grad.zero_()
    b.grad.zero_()
```

Autograd (*automatic differentiation*)

`detach` détache un tenseur du graph de calcul.

- `b=a.detach()` : renvoie un nouveau tenseur `b` qui est une copie de `a` détaché du graph de calcul (aucun historique d'opérations, `requires_grad` faux, pas de gradient, ...)

```
a tensor([0.6469],
requires_grad=True)
c tensor([0.6469]) False None
```

```
for epoch in range(n_epochs):
    # 1- forward pass: Computes our model's
    #       predicted output
    yhat = a + b * x_train_tensor
    # 2- Computes the error and MSE loss
    error = y_train_tensor - yhat
    loss = (error ** 2).mean()

    # 3- Computes gradients for both "a" and "
    #       b" parameters
    loss.backward()

    # 4- Update parameters using gradient
    #       with torch.no_grad():
    a -= lr * a.grad
    b -= lr * b.grad

    print(a)

    c = a.detach()
    print("c", c, c.requires_grad, c.grad)
```

PyTorch's *optimizer*

Mise à jour des paramètres

Un **optimiseur** (SGD, Adam, ...) met à jour les paramètres :

- ❶ choix de l'optimiseur, des hyper-paramètres et des paramètres à mettre à jour
- ❷ calcul de la mise à jour avec `step()`
- ❸ remise à zéro des gradients avec `zero_grad()`

```
from torch import optim
...
# Defines a SGD optimizer to update the
# parameters
optimizer = optim.SGD([a, b], lr=lr)

for epoch in range(n_epochs):
    # 1- forward pass: Computes our model's
    # predicted output
    yhat = a + b * x_train_tensor
    # 2- Computes the error and MSE loss
    error = y_train_tensor - yhat
    loss = (error ** 2).mean()

    # 3- Computes gradients for both "a" and "
    # b" parameters
    loss.backward()

    # 4- Update parameters using gradient
    # No more manual update!
    # with torch.no_grad():
    #     a -= lr * a.grad
    #     b -= lr * b.grad
    optimizer.step()

    # gradients are accumulated: zero the
    # gradients
    optimizer.zero_grad()
```

Calcul de la *loss*

PyTorch propose de nombreuses fonctions de perte, e.g. `MSELoss`

- ❶ choix de la fonction de perte
- ❷ calcul de la *loss*

```
from torch import optim
from torch import nn
...
# Defines a MSE loss function
loss_fn = nn.MSELoss(reduction='mean')

# Defines a SGD optimizer to update the
# parameters
optimizer = optim.SGD([a, b], lr=lr)

for epoch in range(n_epochs):
    # 1- forward pass: Computes our model's
    # predicted output
    yhat = a + b * x_train_tensor
    # 2- Computes the MSE loss
    # No more manual loss!
    # error = y_tensor - yhat
    # loss = (error ** 2).mean()
    loss = loss_fn(y_train_tensor, yhat)

    # 3- Computes gradients for both "a" and
    # "b" parameters
    loss.backward()

    # 4- Update parameters using gradient
    optimizer.step()

    # gradients are accumulated: zero the
    # gradients
    optimizer.zero_grad()
```


Conclusion

Résumé : Init + 4 étapes par epoch

Initialisation des paramètres et des hyper-paramètres

- ➊ *forward pass* : calcul des **prédictions** du modèle courant
- ➋ Calcul de la *loss*, en utilisant les **prédictions**, **labels** et la **fonction de perte** appropriée à la tâche
- ➌ Calcul du **gradient** - dérivées partielles pour chaque paramètre
- ➍ Mise à jour des paramètres

```
# Random initialization of parameters
torch.manual_seed(42)
a = torch.randn(1, requires_grad=True, dtype=
    torch.float)
b = torch.randn(1, requires_grad=True, dtype=
    torch.float)

# Initialization of hyper-parameters
lr = 1e-1
n_epochs = 1000

# Defines loss function and optimizer
loss_fn = nn.MSELoss(reduction='mean')
optimizer = optim.SGD([a, b], lr=lr)

for epoch in range(n_epochs):
    # 1- forward pass: Computes our model's
    # predicted output
    yhat = a + b * x_train

    # 2- Computes the loss
    loss = loss_fn(y_train_tensor, yhat)

    # 3- Computes gradients for both "a" and
    # "b" parameters
    loss.backward()

    # 4- Updates parameters
    optimizer.step()

    optimizer.zero_grad()
```

- 1 Introduction
- 2 Rappels : descente de gradient & régression linéaire (NumPy)
- 3 Régression linéaire en PyTorch
- 4 Modèle/NN avec PyTorch**
- 5 Application : classification supervisée

1 - Modèle avec PyTorch : définition

Représenter la fonction cible à apprendre f_θ .

Héritage de la classe Module

Redéfinition :

- `__init__(self)` : constructeur
- `forward(self, x)` : calcul de la prédiction du modèle pour la feature/entrée x
- utilisation de la classe `Parameter` : récupération d'un itérateur sur les paramètres du modèle (`.parameters()`), des valeurs des paramètres (`state_dict()`), ...

Modèle de régression linéaire 1D $y = f_\theta(x) = a + b * x$

```
from torch import nn

class ManualLinearRegression(nn.Module):
    def __init__(self):
        super().__init__()
        self.a = nn.Parameter(torch.randn(1, requires_grad=True, dtype=torch.float))
        self.b = nn.Parameter(torch.randn(1, requires_grad=True, dtype=torch.float))

    def forward(self, x):
        # Computes the outputs / predictions
        return self.a + self.b * x
```

1- Modèle avec PyTorch : utilisation

La méthode `forward(x)` ne doit pas être appelée, il faut appeler le modèle directement.

train/eval mode

Modèle en mode entraînement ou évaluation car comportements différents (e.g. Dropout, BatchNorm, ...)

```
OrderedDict([('a',
tensor([0.9998])), ('b',
tensor([1.9619]))])
```

```
torch.manual_seed(42)

# Create a model
model = ManualLinearRegression()

lr = 1e-1
n_epochs = 1000

loss_fn = nn.MSELoss(reduction='mean')
# itérateur sur les paramètres
optimizer = optim.SGD(model.parameters(),
                        lr=lr)

for epoch in range(n_epochs):
    model.train()

    # No more manual prediction!
    # yhat = a + b * x_tensor
    yhat = model(x_train_tensor)

    loss = loss_fn(y_train_tensor, yhat)
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()

# Inspect its parameters
print(model.state_dict())
```

2- Modèle avec PyTorch : définition avec Layers de PyTorch

Représenter la fonction cible à apprendre.

Utilisation des couches prédéfinies (Layers) dans PyTorch

- Linear Layers : `nn.Linear`, `nn.Identity`, ...
- Convolutional Layers : `nn.Conv1D`, `nn.Conv2D`, ...
- Recurrent Layers : `nn.RNN`, `nn.LSTM`, ...

Modèle de régression linéaire 1D $y = f_{\theta}(x) = a + b * x$

Transformation linéaire avec `nn.Linear(input_size), output_size`.

```
from torch import nn

class LayerLinearRegression(nn.Module):
    def __init__(self):
        super().__init__()
        # Instead of our custom parameters, we use a Linear layer with single input and
        # single output
        self.linear = nn.Linear(1, 1)

    def forward(self, x):
        # Now it only takes a call to the layer to make predictions
        return self.linear(x)
```

Conclusion

Autres concepts de PyTorch

- ➊ création/chargement de Dataset
 - ➋ DataLoader : itération sur dataset
 - ➌ Sauvegarde/Chargement de modèles
 - ➍ Utilisation du GPU
 - ➎ ...
-
- <https://pytorch.org/tutorials/>
 - <https://pytorch.org/docs/stable/nn.html>

- 1 Introduction
- 2 Rappels : descente de gradient & régression linéaire (NumPy)
- 3 Régression linéaire en PyTorch
- 4 Modèle/NN avec PyTorch
- 5 Application : classification supervisée

TP1 : classification supervisée

- Installation de **conda** pour la gestion d'environnements virtuels Python
- Utilisation de **jupyter notebook**, *cahier électronique* pouvant contenir du texte, des images, des formules mathématiques et du code informatique exécutable. Ils sont manipulables interactivement dans un navigateur web.
- Deux notebooks contenant le code présenté dans ce CM pour tester
- Un notebook à compléter : classification Fashion-MNIST