Apprentissage automatique 2 (KINX9AB1)

Cours 1 : Introduction

M2 IAFA, IMA

Contributeurs :
Thomas Pellegrini
Philippe Muller
Contacts : prenom.nom@irit.fr

Semestre 9

- Cours, 12 heures
- TD, 8 heures, 2 groupes
- TP, 10h, 3 groupes

Modalités de Contrôle des Connaissances

- CT : 70%, examen écrit de deux heures
- CCTP : 30%, note : devoir et/ou compte-rendus de TP, règle "16" : ABI $\rightarrow$ note 0, ABJ $\rightarrow$ coef 0

Objectifs généraux

- reconnaitre un problème d'apprentissage automatique : supervisé, non supervisé, par renforcement
- savoir traiter des données structurées de type séquence
- appliquer pratiquement les méthodes adéquates pour ces situations
- appliquer des combinaisons de modèles (par ex. en multi-tâches)
- approfondir la compréhension théorique de l'apprentissage automatique

Plan du cours

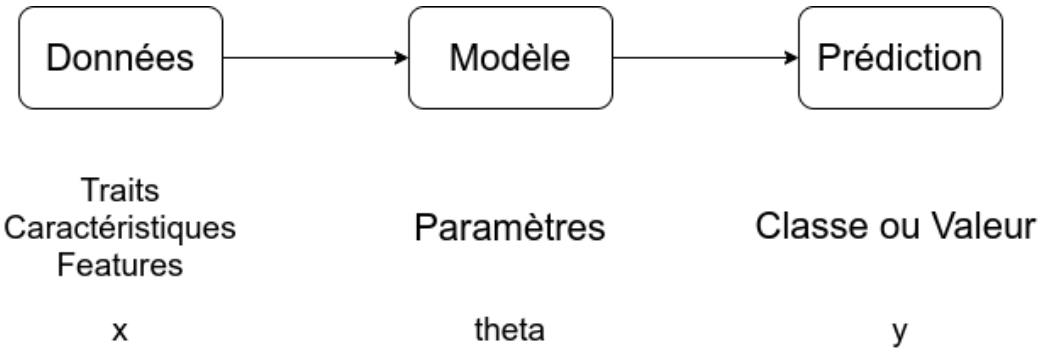
1. Théorie de l'apprentissage
2. Aspects pratiques en pytorch
3. Modèles structurés pour les problèmes séquentiels
4. Apprentissage par renforcement
5. Apprentissage combiné : multi-tâches, joint, ensemble, transfert

Plan du jour

- Rappels sur l'apprentissage artificiel et les différents types
- Contraintes théoriques sur l'apprentissage : PAC learning
- Apprentissage différentiable en pratique avec torch

Apprentissage automatique : rappels et un peu de théorie

- *Modélisation mathématique* : Création en général **manuelle**, d'un modèle à partir de lois physiques ou mathématiques
- $\neq$ *Apprentissage artificiel* : Recherche automatique d'un modèle à partir de données, pouvant être réutilisé dans un nouvel environnement ou une nouvelle situation (ex prédiction) aussi appelé *raisonnement inductif, induction*



Quel est le nombre a qui prolonge la séquence

$1 \quad 2 \quad 3 \quad 5 \quad \ldots a?$

- Solution(s). Quelques réponses valides :
  - $a = 6$. Argument : c'est la suite des entiers sauf 4.
  - $a = 7$. Argument : c'est la suite des nombres premiers.
  - $a = 8$. Argument : c'est la suite de Fibonacci
  - $a =$n'importe quel nombre réel supérieur ou égal à 5. Argument : la séquence présentée est la liste ordonnée des racines du polynôme :

    $$P = x^5 - (11+a)x^4 + (41+11a)x^3 - (61-41a)x^2 + (30+61a)x - 30a$$

    qui est le développement de :

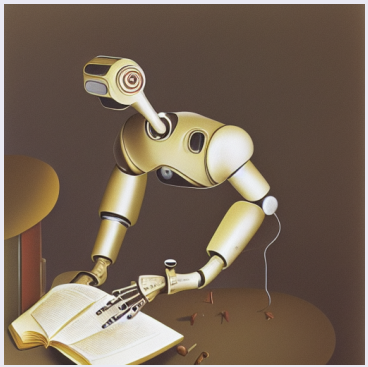
    $$(x-1).(x-2).(x-3).(x-5).(x-a)$$

Généralisation

Il est facile de démontrer ainsi que n'importe quel nombre est une prolongation correcte de n'importe quelle suite de nombre

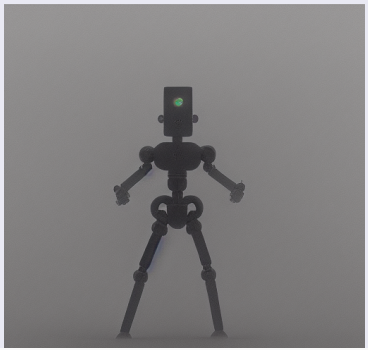
Apprendre suppose un biais sur les modèles possibles

Apprentissage supervisé



Renforcement

Apprentissage non supervisé



- Apprentissage supervisé :
  - À partir de l'échantillon d'apprentissage $S = \{(x_i, u_i)\}_{1,m}$
  - on cherche une loi de dépendance sous-jacente
- Par exemple une fonction h aussi proche possible de f (fonction cible) avec f telle que

$$u_i = f(x_i)$$

- Ou bien une distribution de probabilités $P(u_i/x_i)$
- But : afin de prédire l'avenir

- Si u est une valeur continue
  - Régression
  - Estimation de densité
- Si u est une valeur discrète/nominale
  - Classification
- Si u est une valeur binaire
  - apprentissage de concept

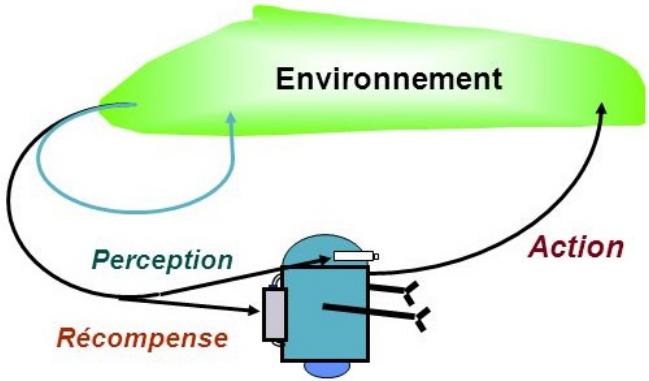
- Apprentissage non supervisé :
  - À partir de l'échantillon d'apprentissage $S = \{x_i \quad | \quad i \in 1...m\}$ "training set"
- on cherche des régularités sous-jacentes :
  - Sous forme de sous-ensembles (Clustering)
  - Sous forme d'une densité (e.g. mixture de gaussiennes)
  - Sous forme d'un modèle complexe (e.g. réseau bayésien)
- Afin de résumer, détecter des régularités, comprendre/explorer la structure des données
  "data mining"

Données

- Une séquence de perceptions, d'actions et de récompenses
- Des renforcement *rt*
- *rt* peut sanctionner ou valider des actions

Le problème

Dans une situation donnée, choisir action afin de maximiser un gain sur le long terme



Quel type de problème ?

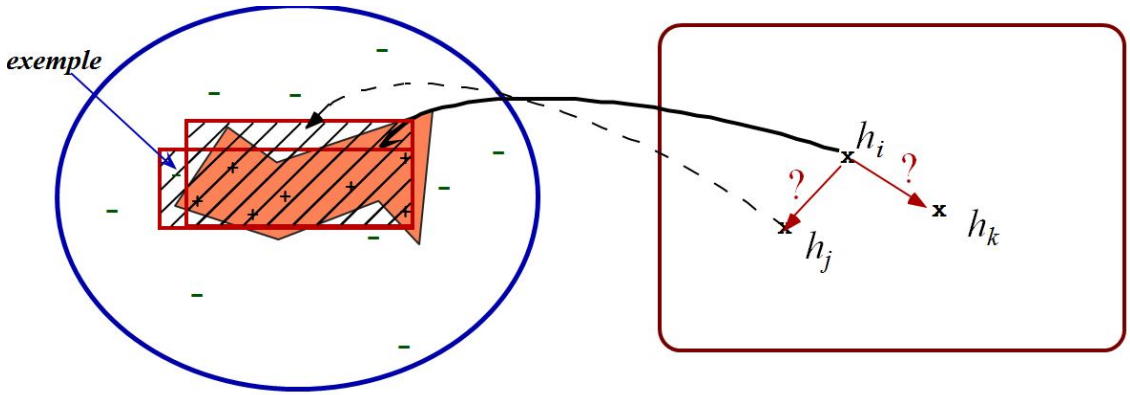
Donnez le type d'apprentissage applicable à ces exemples :

- Génération de catégorie d'élève en fonction de leurs notes
- Prédiction du poids d'une personne en fonction de sa taille et de son âge
- programme apprenant à jouer à Tetris
- Prédiction de catégorie de mauvais payeurs dans une assurance
- Prédiction du taux échappement de CO2 d'une faille en fonction de ses caractéristiques géologiques
- probabilité de la concentration de glucose dans le plasma chez les diabétiques

- Données et connaissances a priori
  - Quelles données sont disponibles ?
  - Que sait-on du problème ?
- Représentation
  - Comment représenter les exemples ?
  - Comment représenter les hypothèses ?
- Méthode et estimation
  - Quel est l'espace des hypothèses ?
  - Comment évaluer une hypothèse en fonction des exemples connus ?
- Évaluation de la performance après apprentissage ?
- Comment reconsidérer l'espace des hypothèses ?

$X = \mathbb{R}^2$ avec deux classes "+" et "-".
L'ensemble des "+" est en orange et est inconnu mais on a quelques observations

Exemple de H : l'ensemble des rectangles parallèles aux axes



Espace des exemples : **X** Espace des hypothèses : **H**

Comment trouver la "meilleure" hypothèse de H ?

Choix d'espace des hypothèses

- Comment explorer l'espace des hypothèses H ? est il fini ? infini ?
- Comment évaluer les hypothèses ? Il faut une fonction d'erreur, aussi appelée risque, $R(h)$ qui est l'erreur de h sur tout l'ensemble X
- Impossible en pratique : on évalue l'erreur/le risque "empirique" R_{emp}, évalué sur le training set, et on cherche l'hypothèse qui minimise ce risque.
- C'est l'apprentissage par minimisation du risque empirique (empirical risk minimization ou ERM).
- Que peut-on dire de $R_{emp}(h)$ par rapport à $R(h)$?

- Que voudrait-on garantir sur $R_{emp}(h)$?

Choix d'espace des hypothèses

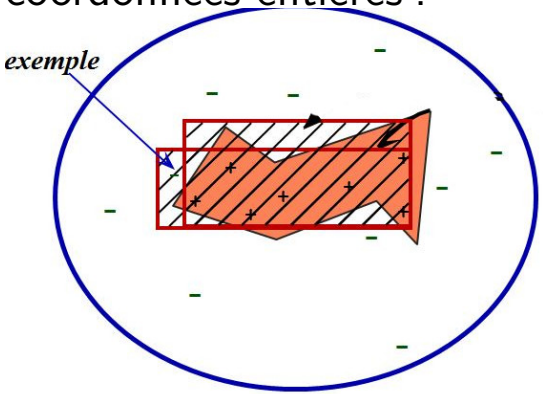
- Comment explorer l'espace des hypothèses H ? est il fini ? infini ?
- Comment évaluer les hypothèses ? Il faut une fonction d'erreur, aussi appelée risque, $R(h)$ qui est l'erreur de h sur tout l'ensemble X
- Impossible en pratique : on évalue l'erreur/le risque "empirique" R_{emp}, évalué sur le training set, et on cherche l'hypothèse qui minimise ce risque.
- C'est l'apprentissage par minimisation du risque empirique (empirical risk minimization ou ERM).
- Que peut-on dire de $R_{emp}(h)$ par rapport à $R(h)$? $R(h) \geq R_{emp}(h)$
- Que voudrait-on garantir sur $R_{emp}(h)$?

Choix d'espace des hypothèses

- Comment explorer l'espace des hypothèses H ? est il fini ? infini ?
- Comment évaluer les hypothèses ? Il faut une fonction d'erreur, aussi appelée risque, $R(h)$ qui est l'erreur de h sur tout l'ensemble X
- Impossible en pratique : on évalue l'erreur/le risque "empirique" R_{emp}, évalué sur le training set, et on cherche l'hypothèse qui minimise ce risque.
- C'est l'apprentissage par minimisation du risque empirique (empirical risk minimization ou ERM).
- Que peut-on dire de $R_{emp}(h)$ par rapport à $R(h)$?
  $R(h) \geq R_{emp}(h)$
- Que voudrait-on garantir sur $R_{emp}(h)$?
  $R(h) \leq R_{emp}(h) + g(N, probleme)$, $g > 0$ et décroit avec N.

Suite de l'exemple

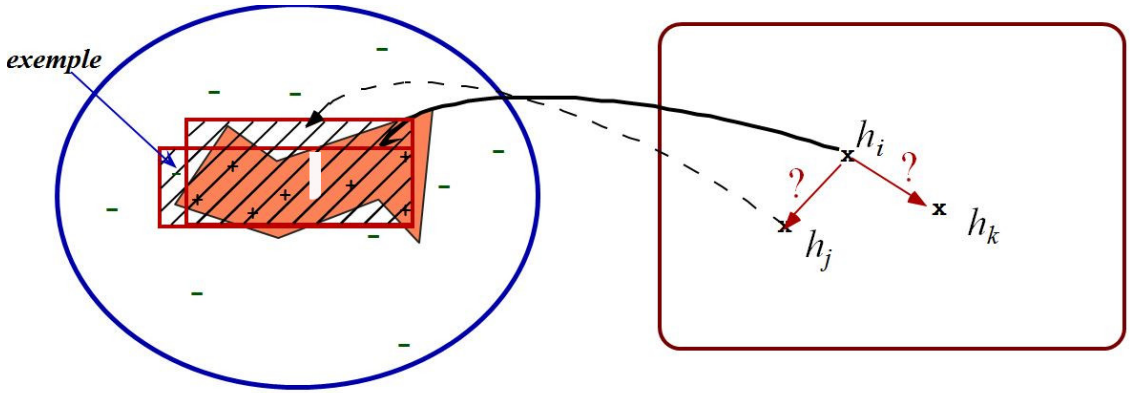
- Une mesure simple d'erreur de prédiction en classification est de donner 0 si la prédiction est bonne, 1 sinon. $R_{emp}(h)$ est donc la somme des erreurs de classification de h sur le training set
- Peut-on trouver un $h \in H$ qui minimise le risque empirique ? Quelle est la valeur du risque empirique ?
- Et si on restreint X à un sous-espace fini (par exemple [0,10]x[0,10]), et H aux rectangles avec des coins de coordonnées entières ?



Espace des exemples : $\mathbf{X}$

Suite de l'exemple

On change maintenant légèrement l'ensemble d'entrainement (observation d'un "-" au milieu de la zone orange)



Espace des exemples : $\mathbf{X}$ Espace des hypothèses : $\mathbf{H}$

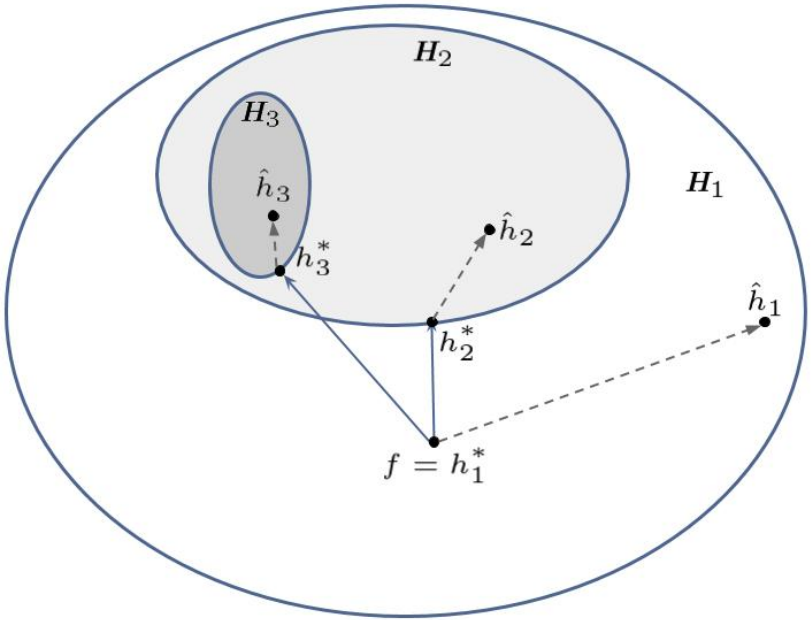
Peut-on trouver un $h \in H$ qui minimise le risque empirique ?
Quelle est alors la valeur du risque empirique ?

Biais

toute connaissance qui restreint le champ des hypothèses que l'apprenant doit considérer à un instant donné.

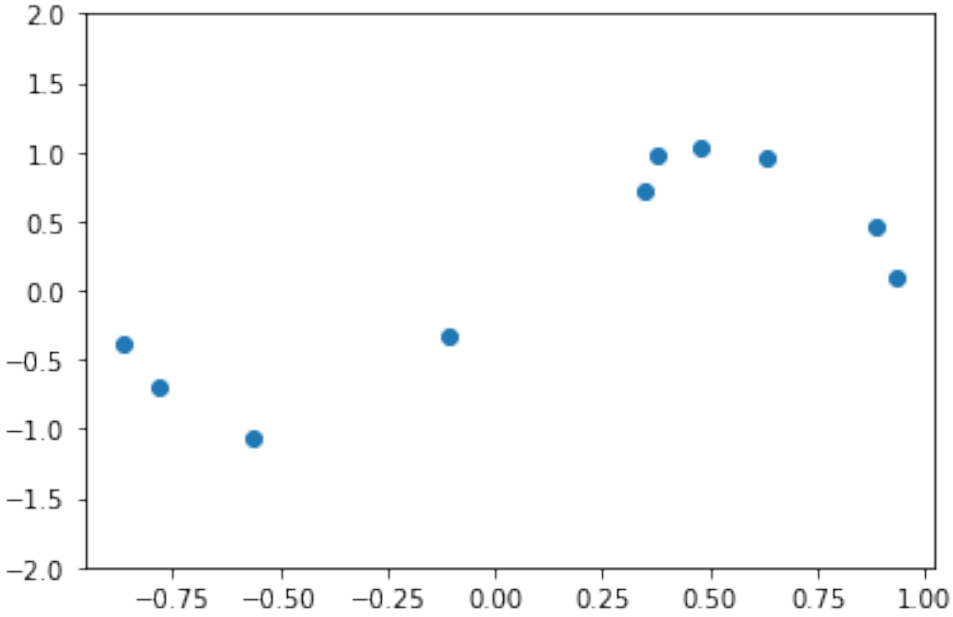
- On ne peut pas apprendre sans biais
- Plus le biais est fort, plus l'apprentissage est "facile"
- différents biais :
  - biais de représentation
  - biais d'hypothèse
  - biais algorithmique

Dans un espace d'hypothèses H, on cherche celle qui minimise le risque, notons là h^*_H.
(en pratique on trouvera une approximation $\hat{h}_H$)
l'erreur de biais est donc $E_{x \in X}[f(x) - h^*_H(x)]$



H_3 : plus facile de trouver une bonne solution, mais elle sera plus loin de l'optimum

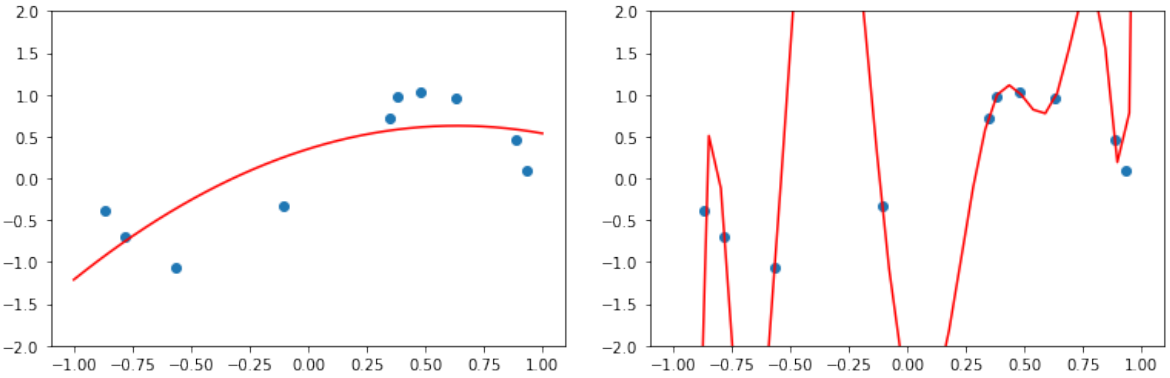
Sur un problème de régression, on a les observations suivantes



Compromis biais-variance

Espace des hypothèses : plusieurs biais possibles

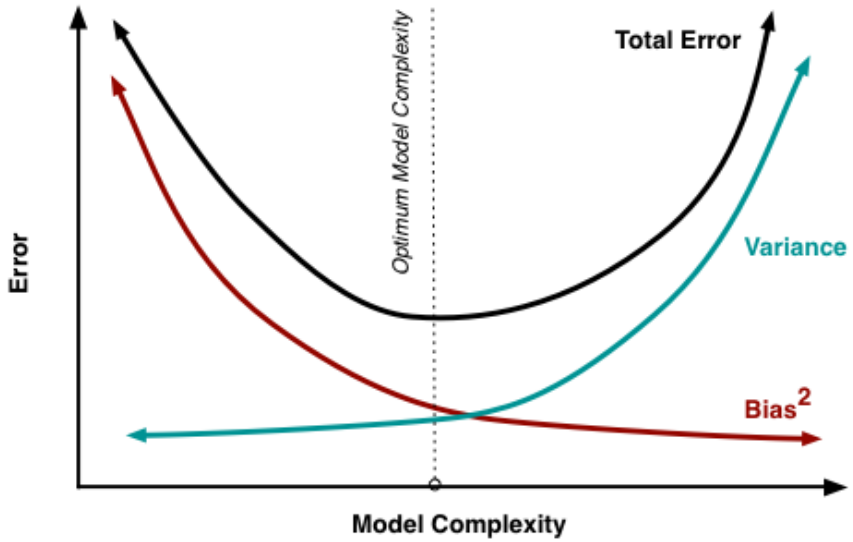
polynôme de degré 2 $\Longleftrightarrow$ polynôme degré 12



Pouvez-vous trouver un meilleur biais ?

Compromis biais-variance

On peut montrer que l'erreur en généralisation revient à :
Erreur = Biais^2 + Variance + erreur irréductible (bruit)



source :
`http://scott.fortmann-roe.com/docs/BiasVariance.html`

- impossible en pratique d'explorer toutes les hypothèses possibles
- Il faut contrôler l'expressivité de l'espace d'hypothèses
- des garanties sur l'erreur réelle ... c'est bien, mais en pratique ?
- quelle complexité algorithmique de trouver un "bon" modèle ?

PAC learning : "Probably Approximately Correct"

- "bon" modèle ? on veut garantir que l'erreur est faible, au besoin en ayant assez d'instances d'entrainement
- on veut être à peu près sur que l'on peut trouver un modèle avec une erreur bornée. Mathématiquement : si h est l'hypothèse trouvée, D l'ensemble sur lequel on l'entraine et on va noter $r_{c,D}$ le risque empirique sur D pour le concept à apprendre c.

  $$P_D(r_{c,D}(h) \leq \epsilon) > 1 - \delta$$

  Avec ϵ, δ des petites constantes fixées.
- on veut maintenant caractériser un problème "apprenable" en pratique : il faut qu'il existe un algorithme A polynomial en fonction de la limite d'erreur et de la chance d'y arriver, donc polynomial en $1/\epsilon$ et $1/\delta$

PAC learning

"Probably Approximately Correct"

- A doit marcher pour tout D et tout concept c à apprendre. Mais : trop général, même dans le cas binaire, les concepts sont toutes les fonctions de $X \to \{0, 1\}$
- on revient à la nécessité du biais : on peut éventuellement espérer avoir un algorithme efficace pour une classe de concepts mais pas tous les concepts à apprendre possible.
- Une "classe" de concepts C est une famille de fonctions $X \to \{0, 1\}$,
  par exemple si X est R^2, la famille des séparateurs linéaires (une fonction qui renvoie 0 ou 1 selon que $x \in X$ soit d'un côté ou de l'autre d'une droite donnée)

PAC learning

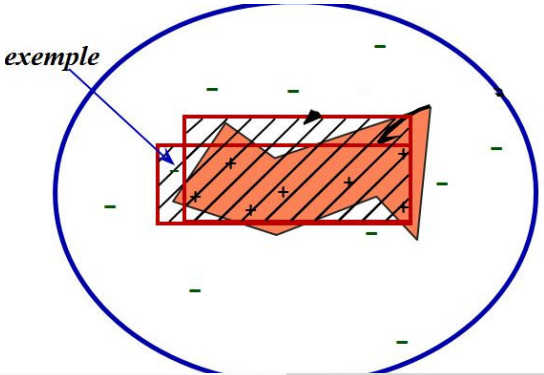
"Probably Approximately Correct"

On dit alors que la classe de concepts C est PAC-apprenable s'il existe un algorithme A en fonction de ϵ, δ (compris entre 0 et $1/2$), polynomial en fonction de $1/\epsilon, 1/\delta$ qui, pour n'importe quelle distribution D sur X et n'importe quel concept $c \in C$, produira une hypothèse h avec une forte probabilité d'avoir une erreur faible, c'est-à-dire

$$P_D(r_{c,D}(h) \leq \epsilon) > 1 - \delta$$

Retour sur l'exemple

- H : l'ensemble des rectangles parallèles aux axes
- h : un rectangle particulier
- X : R^2
- D : les points sur lesquels on a des observations
- C : pourrait être plein de choses : la même chose que H, ou l'ensemble des polygones fermés etc. Dans l'exemple c est un polygone, donc C est au moins un ensemble de polygones.
- c : la zone orange
- exemple de A : prendre le rectangle englobant minimal des exemples + (complexité linéaire en fonction de $|D|$ ... pourquoi ?)



Résultat théorique

On se posera en fait la question simplifiée où on suppose $H = C$
- si $|H|$ est fini, et est **réalisable** (il existe $h \in H$ d'erreur réelle nulle) on peut montrer que le problème est PAC apprenable et ceci dès qu'on a au moins n exemples, avec

$$n \geq \frac{\log(|H|/\delta)}{\epsilon}$$

- si $|H|$ est fini et le problème est agnostique (non réalisable ou bien on ne sait pas) avec

$$n \geq \frac{\log(2|H|/\delta)}{2\epsilon^2}$$

ok, et si H n'est pas de taille finie ?

On peut avoir des résultats de borne quand même avec $|H|$ quelconque, en utilisant la notion de dimension de Vapnick-Chervonenkis (VC dimension)
Avec m exemples : on peut montrer

$$R_{réel}(h) \leq R_{Emp} + f(G_H/m)$$

- G_H = dimension de Vapnick-Chervonenkis ; f fonction croissante

- Mesure la complexité de l'espace des hypothèse
- Un espace d'hypothèse pulvérise un ensemble si, pour tout étiquetage de cette ensemble, il existe une hypothèse qui ne fait pas d'erreur

VC dim

taille du plus grand ensemble pulvérisé par l'espace des hypothèses

- modèle robuste = erreur réelle du même ordre de grandeur que l'erreur empirique
- apprentissage **consistant** si l'erreur empirique converge vers l'erreur réelle quand le nombre d'exemples augmente

Théorème : **Si VC est finie pour une famille de modèles, l'apprentissage est consistant**

- Calculer la dimension de Vapnick des concepts suivants :
  - Signe de $x - b$ dans R
  - Demi-plan dans R^2
  - Rectangle droit dans R^2
  - Rectangle quelconque dans R^2
  - Polygone dans R^2
  - Hyperplan dans R^d

A ne pas confondre avec le nombre de paramètres libres ...

Dimension VC

- Signe de $x - b$ dans R
  donc H = ensemble des bornes possibles b
  Avec 2 points : toujours une hypothèse d'erreur nulle
  Avec 3 points ?
- Demi-plan dans R^2
  donc H défini par ensemble des droites du plan
  Avec ? points : toujours une hypothèse d'erreur nulle
  Avec plus ?

Dimension VC

- Signe de $x - b$ dans R
  donc H = ensemble des bornes possibles b
  Avec 2 points : toujours une hypothèse d'erreur nulle
  Avec 3 points ? non / exemple
- Demi-plan dans R^2
  donc H défini par ensemble des droites du plan
  Avec ? points : toujours une hypothèse d'erreur nulle
  Avec plus ?

Dimension VC

- Signe de $x - b$ dans R
  donc H = ensemble des bornes possibles b
  Avec 2 points : toujours une hypothèse d'erreur nulle
  Avec 3 points ? non / exemple
- Demi-plan dans R^2
  donc H défini par ensemble des droites du plan
  Avec 3 points : toujours une hypothèse d'erreur nulle
  Avec plus ?

Conséquences pratiques ?

- on peut prouver que certaines classes de concepts sont PAC apprenables avec des bornes garanties sur l'erreur
- le problème : les bornes sont certainement pessimistes
- en pratique les modèles "populaires" semblent avoir des comportements acceptables
- exemple des réseaux de neurones : input de dimension infinie (ou finie mais très grande dimension si on considère les flottants codés en machine) ; peu de garanties théoriques sur l'erreur
- pas clair dans certains cas si la VC dimension est finie ou non

L'induction est une forme d'inférence faillible, il faut donc savoir évaluer sa qualité. Deux aspects :

- Évaluation théorique a priori
  - Dimension de Vapnik-Chervonenkis
  - Critères sur la complexité des modèles : MDL / AIC / BIC
    rasoir d'Ockham :
    « *les hypothèses suffisantes les plus simples sont les plus vraisemblables* »
  - Estimer l'optimisme de la méthode et ajouter ce terme au taux d'erreur
- Évaluation empirique
  - E.g. taux d'erreur : (dans le cas d'un classifieurs binaire avec une fonction de coût liée au nombre d'erreurs)

Compromis complexité / Performances empiriques à trouver
La complexité a un coût : overfitting, mémoire, temps de calcul, interprétation, consommation d'énergie

Critères sur la complexité des modèles

Avec LogLikelihood = la log vraisemblance du modèle appris, n le nombre d'instances d'apprentissage et k le nombre de paramètres du modèle

- AIC : Akaike information criterion

$$AIC = 2 * k/n - 2 * LogLikelihood/n$$

- BIC : Bayesian information criterion

$$BIC = k * \ln(n) - 2 * LogLikelihood$$

- MDL : minimal description length : le nombre minimum de bits nécessaires à représenter le modèle et ses prédictions. si L = nb de bits, h le modèle, D les prédictions sur l'ensemble d'entrainement.

$$MDL = L(h) + L(D|h)$$

peut se ramener au BIC

Bilan de cette partie

Importance des concepts théoriques :

- biais-variance
  - biais sur l'espace d'hypothèses
  - complexité d'un modèle
- pouvoir garantir l'erreur en généralisation, en fonction d'une approximation et de la taille de l'ensemble d'entrainement

Apprentissage artificel en pratique : Implémentations de modèles différentiables avec Pytorch

Modèles différentiables

- on a vu que les réseaux de neurones (RN) reviennent à :
  - un modèle non linéaire différentiable (on peut dériver le résultat par rapport aux paramètres)
  - un optimiseur numérique efficace
- le succès pratique repose sur des librairies qui fournissent les composants pour cette faction de fonctionner, par exemple torch / tensorflow.
- l'approche "différentiable" peut en fait s'appliquer à d'autres cas que les RN (régression logistique, etc)

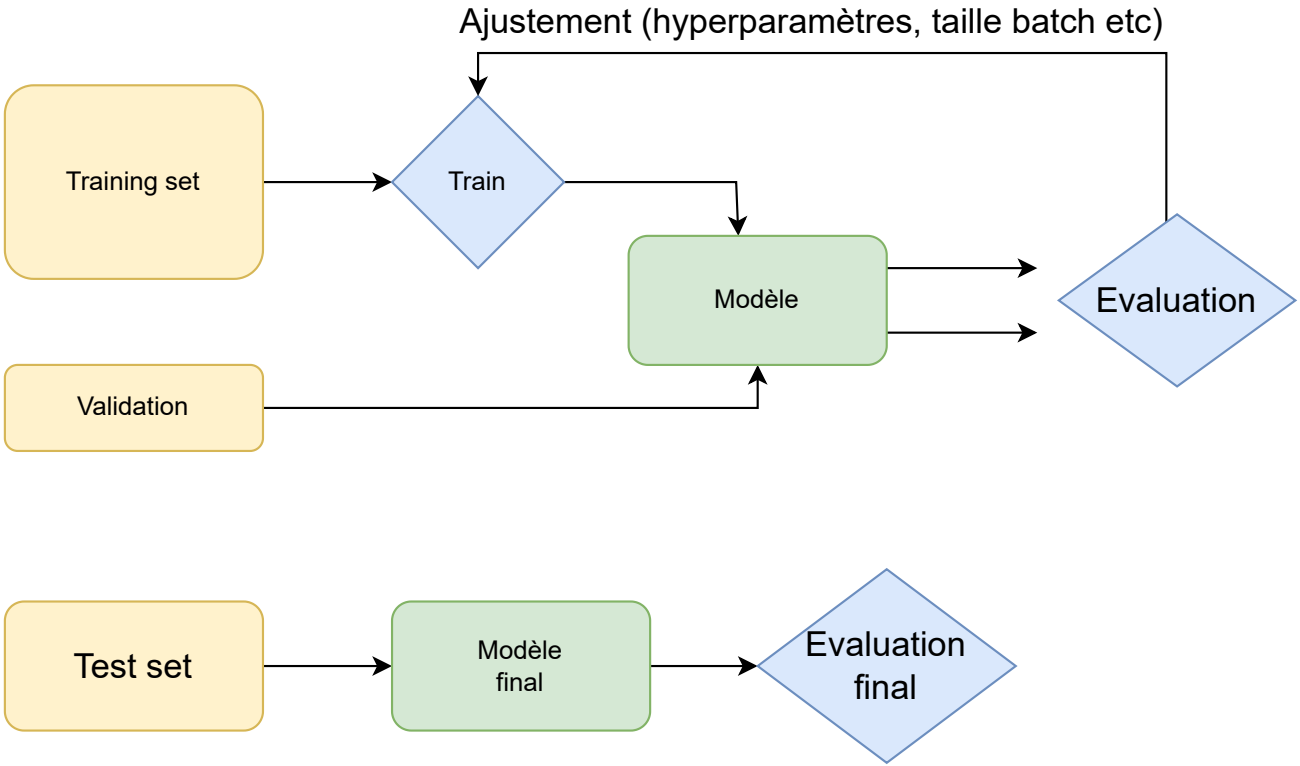
Modèles différentiables

- dans ce cours on utilisera pytorch, l'API python pour torch, pour :
  - processus d'entrainement, de prédiction
  - construction de modèles
  - manipulations de tenseurs
  - gestion des données
  - suivi expérimental

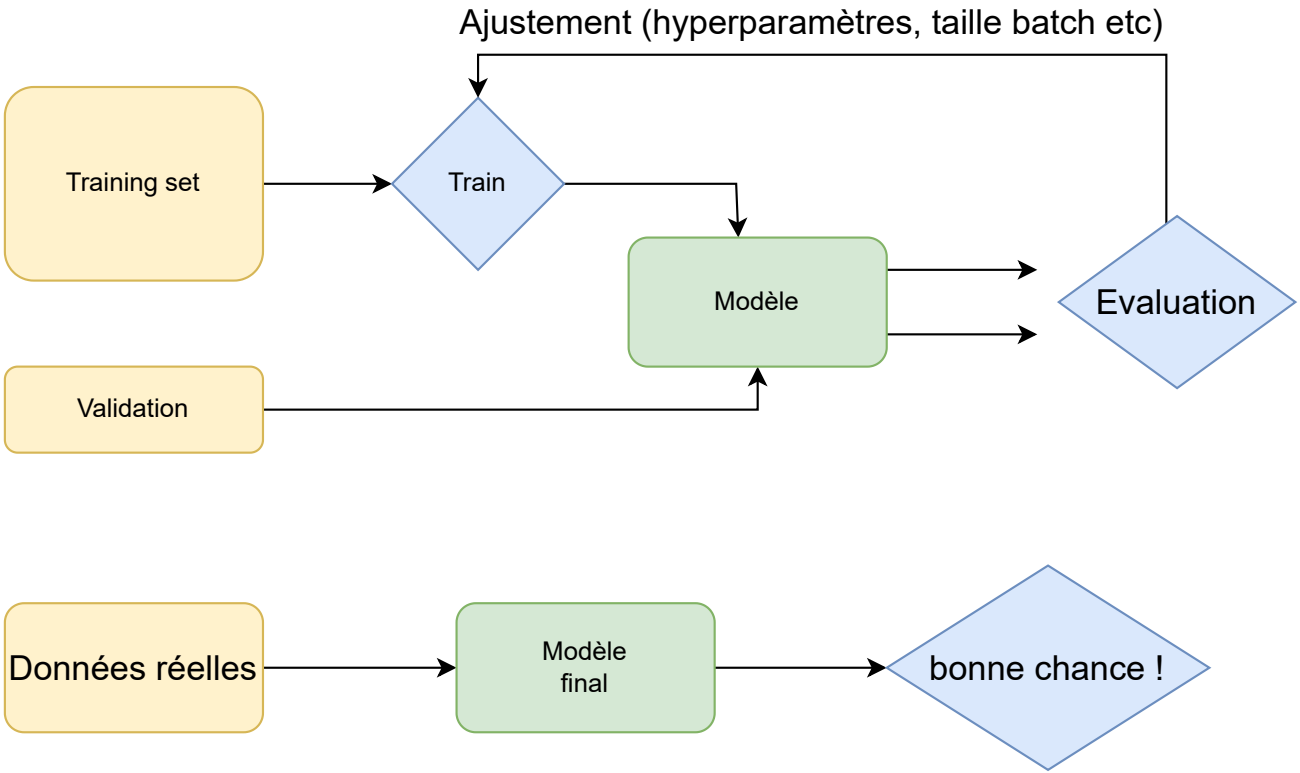
On va maintenant (re)voir les bases de pytorch dans le cadre de l'apprentissage supervisé

$\rightarrow$ servira de base pour les autres : renforcement, non supervisé, ...

Entrainer un modèle dans un cadre expérimental



Entrainer un modèle pour utilisation en production



Boucle d'entrainement simple en pytorch

```
1  def train(model,training_data):
2      for input, target in training_data:
3          optimizer.zero_grad()
4          # passe forward
5          # -> les scores pour chaque classe
6          output = model(input)
7          # la loss attend des tenseurs
8          target = torch.tensor([target])
9          loss = loss_fn(output, target)
10         # calcul des gradients
11         loss.backward()
12         # mise a jour du modele
13         optimizer.step()
14
```

Prédiction pour validation

```
15 def val_prediction(model,val_data):
16     correct = 0
17     total = len(val_data)
18     errors = []
19 
20     for input, target in val_data:
21         output = model(input)
22         # prediction avec le meilleur score
23         best = output.argmax(1)
24         eval = (best == target).item()
25         if not(eval):
26             errors.append((input,target,
27                            best.item()))
28         correct += eval
29     return correct/total, errors
30 
```

Prédiction en production (simplifié)

```
31 def prediction(model,data):
32     results = []
33     scores = []
34     for input, _ in data:
35         output = model(input)
36         scores, best = output.max(1)
37         results.append(best)
38         scores.append(scores)
39     return results, scores
40
```

Gestion des données ?

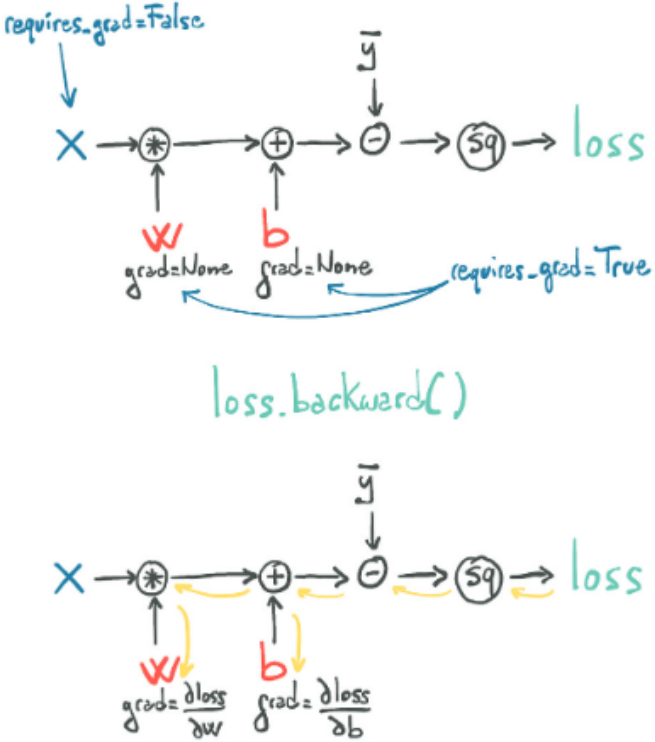
D'où viennent les données ?
- Elles sont stockées dans des "tenseurs" (généralisation des matrices à plus de dimension) similaires à numpy, mais avec une gestion des gradients par rapport aux valeurs des tenseurs.
- Pytorch fournit une abstraction pour lire les données :

```
41 from torchvision import datasets
42
43 training_data = datasets.MNIST(
44     root="~/Ressources/MNIST",
45     train=True,
46     download=True,
47     transform=ToTensor(),
48 )
49 training_data.data.shape
50 #torch.Size([60000, 28, 28])
```

Une énumération sur training_data renvoie une instance et son label

Calculs de gradient

Les tenseurs torch gardent la trace des gradients automatiquement.



- Quand la loss est calculée (passe forward), Pytorch crée un graphe avec le module Autograd
- Quand on appelle ensuite `loss.backward()`, Pytorch (Autograd) parcourt le graphe dans le sens opposé (passe "backward") pour calculer les gradients.

Gestion des données (suite)

L'apprentissage se faisant généralement par "paquets" (batch), torch fournit aussi une abstraction où définir les batch

```
51 from torch.utils.data import DataLoader
52
53 train_dataloader = DataLoader(training_data,
54                                batch_size=64,
55                                shuffle=True)
56
57 train_features, train_labels = next(iter(
       train_dataloader))
58 print(train_features.size())
59 print(train_labels.size())
60 #torch.Size([64, 1, 28, 28])
61 #torch.Size([64])
```

on voit une dimension en plus : chaque valeur issue par le loader regroupe 64 instances d'images dans un seul tenseur

Boucle d'entrainement avec batchs

IL n'y a presque rien à changer, tout étant prévu pour recevoir des tenseurs :

```
62 for input, target in train_dataloader:
63     optimizer.zero_grad()
64     output = model(input)
65     loss = loss_fn(output, target)
66     loss.backward()
67     optimizer.step()
68
```

input et target sont maintenant des batchs d'instances et de labels

Validation avec batch

Ici il faut gérer les résultats renvoyés par batchs

```
69 correct = 0
70 total = len(test_data)
71
72 for input, target in test_dataloader:
73     output = model(input)
74     eval = (output.argmax(1) == target).type(torch.
       float).sum().item() # True or False
75     correct += eval
76 print(correct/total)
77
```

On compare le vrai label à celui qui a le meilleur score selon le modèle

Tenseur : un point

On a donc déjà vu :

- la forme d'un tenseur : t.shape qui renvoie les dimensions comme en numpy (ou t.size())
- quelques méthodes similaires à numpy : max(), argmax(), sum(), qu'on peut projeter sur une dimension particulière.
- item(), qui transforme un tenseur contenant une seule valeur en scalaire.

Créer des tenseurs

Très proche de numpy. Par défaut torch créent de quoi gérer des gradients par rapport aux valeurs des tenseurs

```
78 import torch
79 uninitialized = torch.Tensor(3, 4)
80 rand_initialized = torch.rand(3, 4)
81
82 matrix_zeros = torch.zeros(3, 4)
83
84 tensor_from_list = torch.FloatTensor(python_list)
85 tensor_from_numpy = torch.from_numpy(np.random.rand(2,
86                                                     3))
87 tensor_on_gpu = torch.ones([2, 4], dtype=torch.float64
       , device=cuda0)
88
```

Pour des données qui ne doivent pas être mises à jour à l'entrainement on peut juste déclarer

```
x = torch.tensor([1.], requires_grad=False)
```

Manipulation de tenseurs

l'indexation se fait comme en numpy :

```
89          # training_data.data.shape -> torch.Size
        ([60000, 28, 28])
90          img_batch = training_data.data[0:10]
91          # -> torch.Size([10, 28, 28])
92          img1 = img_batch[0] # 28,28
93          im1[:14,:] # la moitie de l'image
```

multiplication de tenseurs :

```
95        # on reprend notre image img de forme (28x28)
96        # on definit une transformation de l'image
97        transfo = torch.rand(2,28)
98        mult = transfo@img
99        # ou bien: mult = transfo.matmul(img)
100       #mult.shape -> (2,28)
101       # avec un batch ? img_batch de forme (10,28,28)
102       mult = transfo@img_batch # -> (10,2,28) !
```

La multiplication considère la convention que les dimensions de batch sont en premier. La multiplication se fait donc sur les autres dimensions.

En fait on peut même avoir des batchs à dimension multiples, et plusieurs dimensions pour l'input, par exemple une image : (JxKx...) x (CxMxN) = batch x (canal,hauteur,largeur)

```
103         super_batch = torch.rand(30,2,3,28,28)
104         super_out = transfo@super_batch
105         super_out.shape
106         #torch.Size([30, 2, 3, 2, 28])
107
```

Methodes utiles : chunk, split, cat

Reprenons notre training_data de dimensions ([60000, 28, 28])
chunk/split créent une liste de tenseurs, soit en précisant leur taille, soit en disant combien de morceaux on veut :

```
108     c = training_data.data.chunk(10) # liste de 10
        tenseurs
109     # c[0].shape  -> 6000,28,28
110     s = training_data.data.split(1000)
111     # liste de tenseurs de 1000 sur la dimension 1
112     # s[0].shape -> (1000,28,28)
113     # len(s) -> 60
114     # on peut aussi choisir l'axe de decoupe:
115     training_data.data.chunk(4,dim=1)
116     # -> liste de tenseur de forme (60000, 7, 28)
117
```

cat fait l'opération inverse de chunk/split :

torch.cat(a.chunk(...)) == a

Methodes utiles : stack/unbind

- "stack" remet une liste de tenseurs en un seul tenseur en ajoutant une dimension
  on reprend c la liste de 10 x torch.Size([6000, 28, 28])
  newt = torch.stack(c) → torch.Size([10, 6000, 28, 28])
- "unbind" fait l'inverse : renvoie une liste des tenseurs selon une dimension (par défaut la 1ere)
  newc = torch.unbind(newt) → liste de 10 x torch.Size([6000, 28, 28])

Methodes utiles : squeeze, unsqueeze

Certaines opérations ont besoin d'avoir une dimension même pour une seule valeur, par exemple un batch b de 1 instance : shape [1,28,28]

- squeeze permet de réduire un tenseur avec des dimensions à valeur unique :
  unique = b.squeeze() : renvoie le tenseur de forme (28,28)
- unsqueeze fait l'inverse : ajoute une dimension artificielle au tenseur
  unique.unsqueeze(1) $\rightarrow$ tenseur de forme (28,1,28) avec les mêmes valeurs que unique

Manipulation de tenseurs : view

reshape en numpy : redéfinir la forme d'un tenseur tout en gardant l'ensemble des valeurs.
Exemple : statistiques de location de vélo par date/heure et avec des infos de météo

(**cf Bike share dataset**)

dteday	season	yr	temp	hum	cnt
2011-01-01	1	0	0.24	0.81	16
2011-01-01	1	0	0.22	0.80	40
2011-01-01	1	0	0.22	0.80	32
2011-01-01	1	0	0.24	0.75	13
2011-01-01	1	0	0.24	0.75	1

Il y a en fait 17 variables dans le jeu de données.
Dans un tenseur :
torch.Size([17376, 17]), soit 17376 instances de 17 valeurs.

Manipulation de tenseurs : view

S'il est plus pratique d'avoir les données regroupées par jour ?
Cela ferait un tenseur de 17376/24=724 x 24 x 17 :

```
par_jour = bikes.view(724,24,17)
```

Plus simplement, on peut laisser une dimension indéfinie qui sera calculée automatiquement :

```
par_jour = bikes.view(-1,24,17)
```

et même encore plus général :

```
par_jour = bikes.view(-1,24,bikes.shape[1])
```

Attention la "view" pointe sur les mêmes données : modifier bikes change aussi la view et réciproquement

Définition du modèle

On va finir en définissant modèle, loss et optimiseur
Un modèle sous-classe torch.nn.Module, et doit fournir un constructeur et une méthode forward() :

```
118 class MyModel(nn.Module):
119     """modele basique lineaire"""
120     def __init__(self):
121         # gere les initialisations communes
122         # a tous les modeles
123         super(MyModel, self).__init__()
124         self.flatten = nn.Flatten()
125         self.linear = nn.Linear(28*28, 10)
126 
127     def forward(self,input):
128         scores = self.linear(self.flatten(input))
129         return scores
130 ########## + loss et optimiseur de base / parametre du
        modele
131 model = MyModel()
132 loss_fn = nn.CrossEntropyLoss()
133 optimizer = torch.optim.SGD(model.parameters(),
134                             lr=1e-3)
```

Un autre modèle

torch.nn fournit l'essentiel des composants utiles pour créer des réseaux de neurones de façons modulaire

```
136 class MyModel(nn.Module):
137     """multi-layer perceptron"""
138     def __init__(self):
139         super(MyModel, self).__init__()
140         self.flatten = nn.Flatten()
141         self.mlp = nn.Sequential(
142             nn.Linear(28*28, 512),
143             nn.ReLU(),
144             nn.Linear(512, 512),
145             nn.ReLU(),
146             nn.Linear(512, 10),
147         )
148 
149     def forward(self,input):
150         scores = self.mlp(self.flatten(input))
151         return scores
152 
153 
154 
```

Contient :

- de quoi mettre en séquence des couches (Sequential, cf ci-dessus)
- les fonctions d'activations ReLU, tanh, ...
- les transformations courantes : Linear, convolution (Conv1D, Conv2D, etc)
- les opérations de pooling, dropout, normalisations diverses
- les fonctions de pertes courantes
- les couches utiles pour des séquences : réseaux récurrents, Transformers, ...

Torch : à venir

- expérimentations en TP bien sûr
- pistes d'implémentations pour les modèles vus en cours
- dans les cours impliquant de l'apprentissage : vision, parole, TAL, ...
- cf aussi les notebooks exemples fournis sur moodle

Bilan du jour

- un retour théorique sur l'apprentissage automatique
- le cadre pratique pour les manipulations expérimentales

Prochain cours : modèles structurés pour les séquences