Projet Foot
Cours 5
(Semaine 6)
2I013

Nicolas Baskiotis
nicolas.baskiotis@lip6.fr

Université Pierre et Marie Curie (UPMC)
Laboratoire d'Informatique de Paris 6 (LIP6)

S2 (2014-2015)

Plan

Tournoi 1v1

```
[("FML␣can't␣use␣my␣joueur_plus_proche␣:-(_1", '6␣(2,30,0)␣-␣(35,285)
("FML␣can't␣use␣my␣joueur_plus_proche␣:-(", '13␣(4,27,1)␣-␣(46,274)')
('Carottes', '28␣(9,22,1)␣-␣(103,217)'),
('Poireaux', '29␣(9,21,2)␣-␣(81,239)'),
('team1', '37␣(12,19,1)␣-␣(142,178)'),
('Minute␣Maid␣Tropical', '37␣(11,17,4)␣-␣(143,177)'),
('Minute␣Maid␣Tropical_1', '38␣(11,16,5)␣-␣(148,172)'),
('Cherry', '38␣(11,16,5)␣-␣(150,170)'),
('team1_1', '39␣(12,17,3)␣-␣(150,170)'),
('team1', '44␣(14,16,2)␣-␣(153,167)'),
('Cherry_1', '50␣(16,14,2)␣-␣(160,160)'),
('team1_1', '52␣(17,14,1)␣-␣(164,156)'),
('team_1vs1_Precepteur_1', '63␣(20,9,3)␣-␣(188,132)'),
('DiegoMaradona␣1v1', '66␣(20,6,6)␣-␣(228,52)'),
('team_1vs1_Precepteur', '69␣(23,9,0)␣-␣(197,123)'),
('DiegoMaradona␣1v1_1', '70␣(22,6,4)␣-␣(231,49)'),
('Team␣Solo␣Mid␣1v1', '77␣(24,3,5)␣-␣(239,41)'),
('Team␣Solo␣Mid␣1v1_1', '83␣(26,1,5)␣-␣(242,38)')]
```

Tournoi 2v2

```
[('C.A␣BocaJuniors␣2v2', '19␣(5,27,4)␣-␣(72,248)'),
 ('C.A␣BocaJuniors␣2v2_1', '22␣(5,24,7)␣-␣(82,238)'),
 ('Defensive', '26␣(6,22,8)␣-␣(79,221)'),
 ('team2_1', '36␣(0,0,36)␣-␣(0,0)'),
 ('team2', '36␣(0,0,36)␣-␣(0,0)'),
 ('Cramberrie_1', '36␣(10,20,6)␣-␣(109,211)'),
 ('Cramberrie', '37␣(10,19,7)␣-␣(121,199)'),
 ('team1', '42␣(12,18,6)␣-␣(131,189)'),
 ('team1_1', '42␣(12,18,6)␣-␣(131,189)'),
 ('Minute␣Maid␣Orange_1', '42␣(12,18,6)␣-␣(153,167)'),
 ('Minute␣Maid␣Orange', '43␣(13,19,4)␣-␣(157,163)'),
 ('Fonceurs', '50␣(15,16,5)␣-␣(164,156)'),
 ('Patates', '51␣(14,13,9)␣-␣(160,160)'),
 ('Oignons', '54␣(16,14,6)␣-␣(150,170)'),
 ('Relegation', '57␣(17,13,6)␣-␣(185,135)'),
 ('team_2vs2_Temoin', '62␣(19,12,5)␣-␣(181,139)'),
 ('team_2vs2_Preception', '76␣(24,8,4)␣-␣(223,97)'),
 ('Fnatic␣2v2', '83␣(26,5,5)␣-␣(259,61)'),
 ('Tueur␣de␣fonceur␣2v2', '84␣(25,2,9)␣-␣(240,60)'),
 ('team2', '92␣(29,2,5)␣-␣(263,57)')]
```

Tournoi 4v4

```
[('Lemon_1', '10 (2,26,4) - (46,234)'),
 ('Lemon', '10 (2,26,4) - (50,230)'),
 ('team4_1', '32 (0,0,32) - (0,0)'),
 ('team4', '32 (0,0,32) - (0,0)'),
 ('Argentina 4v4', '32 (9,18,5) - (110,170)'),
 ('team_4vs4_Prescience', '32 (9,18,5) - (134,146)'),
 ('Minute Maid Pomme_1', '35 (10,17,5) - (116,164)'),
 ('Minute Maid Pomme', '37 (11,17,4) - (115,165)'),
 ('team_4vs4_Prescience_1', '37 (10,15,7) - (130,150)'),
 ('Argentina 4v4_1', '38 (11,16,5) - (107,173)'),
 ('4v4 super attaque maggle', '46 (14,14,4) - (160,120)'),
 ('Mauro Chupame La Pija 4v4', '50 (15,12,5) - (151,129)'),
 ('Meet Your Maker_1', '56 (17,10,5) - (177,103)'),
 ('Aubergines', '63 (19,7,6) - (186,93)'),
 ('Meet Your Maker', '67 (20,5,7) - (182,97)'),
 ('Unicorn of Love 4v4', '69 (21,5,6) - (187,93)'),
 ('Unicorn of Love 4v4_1', '70 (21,4,7) - (199,81)'),
 ('Tomates', '73 (22,3,7) - (189,91)')]
```

Plan

Rappel : pour l'instant, sélecteur de stratégie

Plusieurs facons de le coder, quelques exemples

Simple

```
class SimpleSelector(SoccerStrategy):
def __init__(self,list_strat):
    self.name="Selecteur simple"
    self.list_strat=list_strat
def selector(self,state,player,teamid):
    if (...):
      return 0
    if (...):
      return 1
    return -1
def compute_strategy(self,state,player,teamid):
    return self.list_strat[self.selector(state,player,teamid)]\
            .compute_strategy(...)
```

Rappel : pour l'instant, sélecteur de stratégie

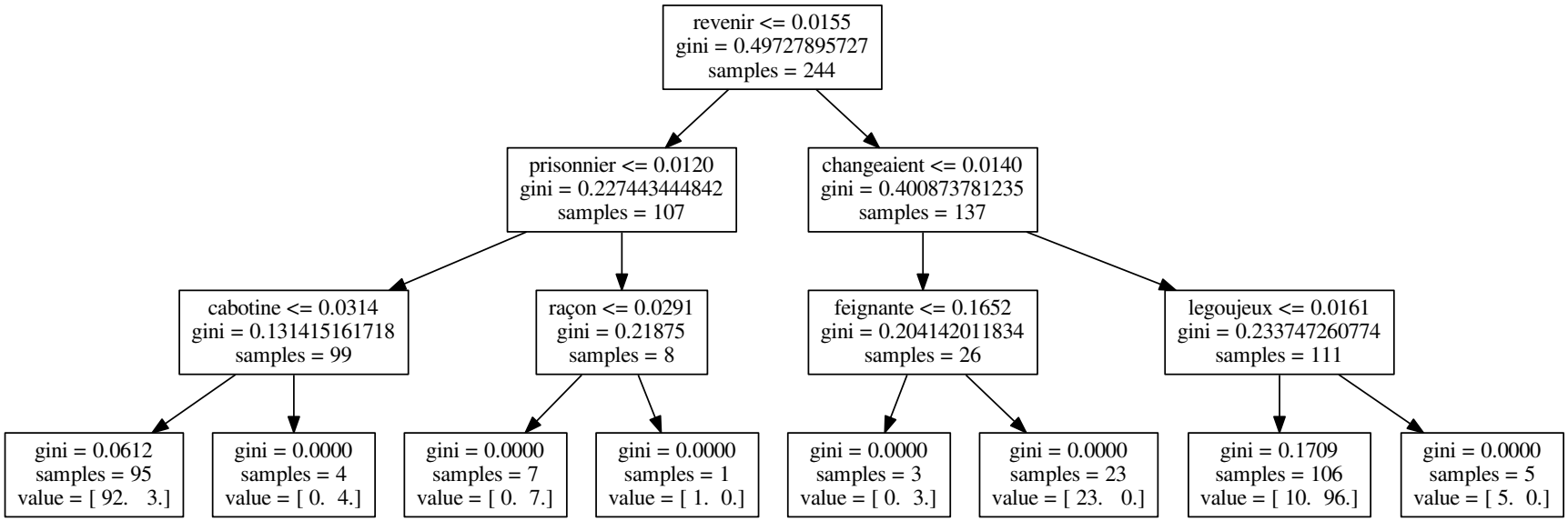
Plusieurs facons de le coder, quelques exemples

Plus élégant

```
#list_cond : liste de fonctions boolennes
 class SelectorStrategy(SoccerStrategy):
    def __init__(self,list_strat,list_cond):
      self.list_strat = list_strat
      self.list_cond = list_cond
      self.name="Selecteur elegant"
    def selector(self,state,player,teamid):
      for strat,cond in zip(self.list_strat,self.list_cond):
         if cond(state,player,teamid):
            return strat(state,player,teamid)
      return list_strat[-1](state,player,teamid)
    def compute_strategy(self,state,player,teamid):
      return self.selector(state,player,teamid)
```

Arbres de décision (rappel)

Objectif : Apprentissage pour le choix automatique d'une stratégie

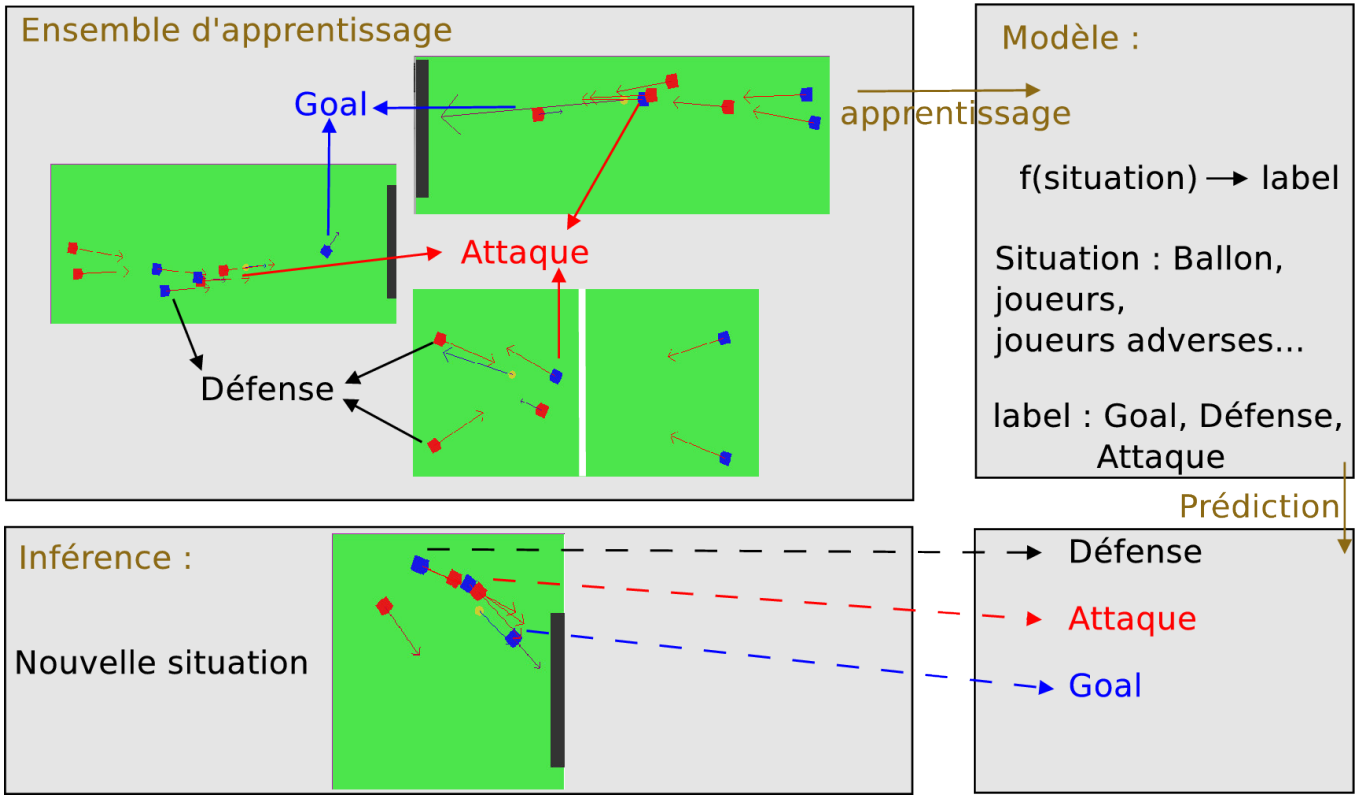


Principe

- Chaque nœud interne : un test sur une des dimensions de $\mathcal{X}$
- Chaque branche : un résultat du test
- Chaque feuille : un label de Y

$\Rightarrow$ classification en parcourant un chemin de la racine à une feuille.

Quels besoins ?



- Un ensemble d'apprentissage : des exemples de situations de jeu avec la bonne stratégie à appliquer
- Un espace de description : des attributs qui décrivent la situation de jeu

Ensemble d'apprentissage

Apprentissage par imitation

- Un prof $\rightarrow$ vous !
- des stratégies $\rightarrow$ les votres (simples ou complexes ?)
- des situations $\rightarrow$ jouer et choisisser les "bonnes" stratégies

Concrétement

- un nouvel interface disponible
- possibilité d'affecter une touche à un couple (stratégie,joueur)
- durant le jeu, à chaque fois qu'une nouvelle stratégie est choisie, enregistrement dans un fichier du couple `(state,stratégie)`

Description d'une situation

Génération d'attributs (*features*)

- Le choix des attributs est crucial pour un bon apprentissage
- Besoin de flexibilité : pouvoir en engendrer d'autres au fur et à mesure de l'avancement du projet

$\rightarrow$ Ne pas reconstruire à chaque fois tous les exemples !

- Solution : une fonction de production d'attributs à partir d'un état
  `gen_feat(state,player,idteam)` $\rightarrow (x_1, x_2, \ldots, x_d)$

Sauver des objets dans un fichier : module `pickle`

- Module magique : permet de (presque) tout sauver (serialisation)
- Utilisation :

```
import pickle
#sauvegarde
with open("nom_fichier","wb") as f:
  pickle.dump(objet,f)
#chargement
with open("nom_fichier","rb") as :
  objet = pickle.load(f)
```

Exemple d'apprentissage d'un arbre

Module mathématique : numpy

```
import numpy as np
#creation de matrice 100x10
mat = np.zeros((100, 10))
#toutes les valeurs de la premiere colonne
mat[:,0]
#toutes les valeurs de la premiere ligne
mat[0,:]
```

Arbres de décision : module sklearn.tree

```
from sklearn.tree import DecisionTreeClassifier
#Creation de l'arbre
arbre = DecisionTreeClassifier()
#Apprentissage de l'arbre sur une matrice x (chaque ligne un exemple)
arbre.fit(x,y)
#prediction d'un exemple
arbre.predict(exemple)
#afficher un arbre
with open("tree.dot","w") as f:
     f = tree.export_graphviz(tr,out_file=f)
os.system("dot -Tpdf tree.dot -o tree.pdf")
```

Exemple d'utilisation

```
class ArbreStrategy(SoccerStrategy):
    #gen_feat  : fonction de generation de feature
    #a partir d'un state, player, teamid
    #tree : arbre de decision
    #dic_strat : dictionnaire de strategie
    #sur lequel l'arbre a ete appris
    def __init__(self, gen_feat,tree,dic_strat):
      self.name="Mon arbre"
      self.tree = tree
      self.gen_feat = gen_feat
      self.dic_strat=dic_strat
    def compute_strategy(self,state,player,teamid):
      strat = self.tree.predict(self.gen_feat(state,player,teamid))
      return dic_strat[strat].compute_strategy(state,player,teamid)
```

Plan

Inspiration biologique

Le cerveau

- Robuste, tolérant aux fautes
- Flexible, sait s'adapter
- Gère les informations incomplètes
- Capable d'apprendre

Composé de neurones !

- 10^{11} neurones dans un cerveau humain
- 10^{4} connexions par neurones
- Potentiel d'action, neuro-transmetteurs, période réfractaire
- Signaux excitateurs / inhibiteurs

Problèmes

- Opacité des raisonnements
- Opacité des résultats

Inspiration biologique

Le cerveau

- Robuste, tolérant aux fautes
- Flexible, sait s'adapter
- Gère les informations incomplètes
- Capable d'apprendre

Composé de neurones !

- 10^{11} neurones dans un cerveau humain
- 10^{4} connexions par neurones
- Potentiel d'action, neuro-transmetteurs, période réfractaire
- Signaux excitateurs / inhibiteurs

Problèmes

- Opacité des raisonnements
- Opacité des résultats

Historique

Prémisses

- Mc Cullch et Pitts (1943) : 1er modèle de neurone formel. Base de l'IA
- Règle de Hebb (1949) : apprentissage par renforcement du couplage synaptique

Premières réalisations

- Adaline (Widrow-Hoff, 1960)
- Perceptron (Rosenblatt, 1958-1962)
- Analyse de Minsky et Papert (1969)

Développement

- Réseau bouclé (Hopfield 1982)
- Réseau multi-couches (1985)

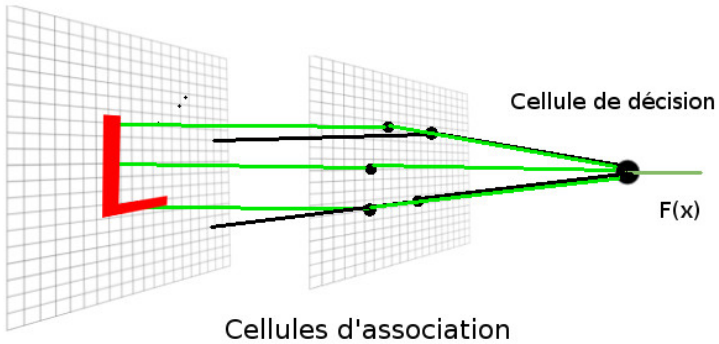
Deuxième renaissance

- Réseaux profonds (2000-)

Le perceptron de Rosenblatt (1960)

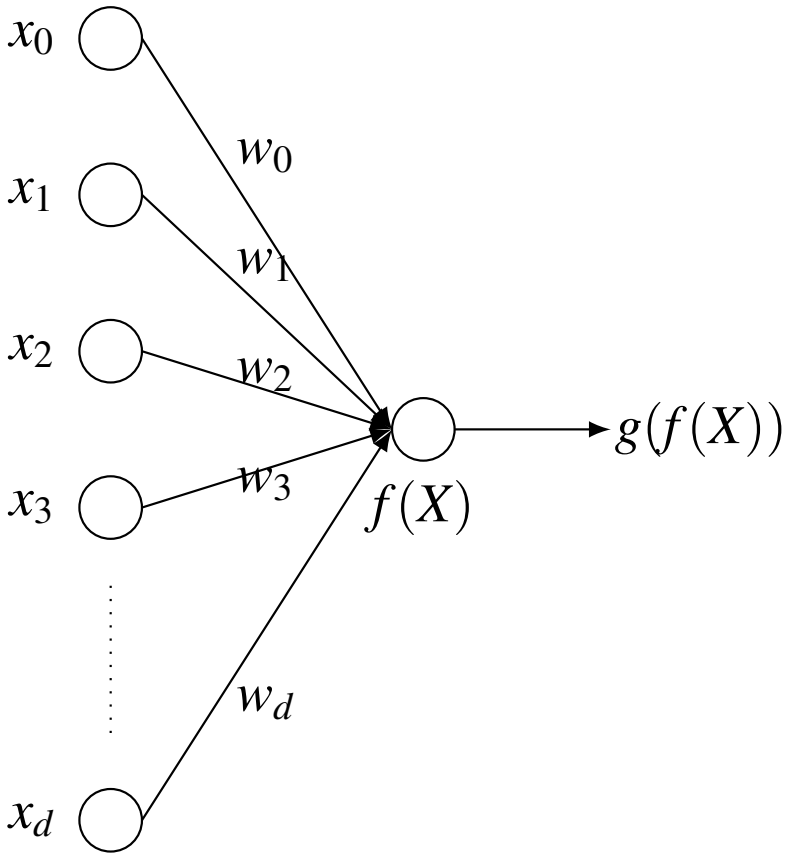
L'idée

- Reconaissance de forme (*pattern*) entre deux classes
- Inspirée cortex visuel



- Chaque cellule d'association produit une sortie $f_i(S)$ en fonction d'un stimulus
- La cellule de décision répond selon une fonction seuil $f_d(\sum w_i f_i(S_i))$

Formalisation



Le perceptron considère

- $f(\mathbf{x}) =< \mathbf{x}, \mathbf{w} >= \sum_{i=1}^{d} x_i w_i$
- Fonction de décision :
  $g(x) = sign(x)$

→ Sortie :
$g(f(\mathbf{x})) = sign(< \mathbf{x}, \mathbf{w} >)$

Considérations géométriques

Soit $y(x)$ la sortie attendue :

- Que représente w par rapport à la séparatrice ?
- Que représente $< w\mathbf{x} >$?
- Que représente $y(x) < w\mathbf{x} >$?
- A quoi correspond la règle de mise à jour :
  - Si $(y(x) < w.\mathbf{x} >) > 0$ ne rien faire
  - Si $(y(x) < w.\mathbf{x} >) < 0$ corriger $w = w + y(x)x$?

Algorithme de résolution

Algorithme du perceptron

- Initialiser au hasard w
- Tant qu'il n'y a pas convergence :
  - pour tous les exemples (x^i, y^i) :
    - si $(y^i < w.\mathbf{x}^i >) < 0$ alors $w = w + \epsilon y^i x^i$
- Décision : $f(x) = sign(< w\mathbf{x} >)$