Apprentissage Statistique

Sujet 1

Rubik’s Cube

, **EL BAZ Adrian**

Résumé :

La prédiction du nombre de coups minimum à la résolution d'un Rubik's Cube peut avoir plusieurs applications sur des problèmes d'algèbre combinatoire. L'étude des algorithmes classiques d'apprentissage statistique nous permet de conclure qu'ils ne sont pas très efficaces, notamment à cause du déséquilibre des classes dans le jeu de données, mais aussi à cause des métriques que l'on utilise dans ces algorithmes. Cependant, des méthodes plus prometteuses utilisant le Reinforcement Learning peuvent être utilisées.

Introduction

Le Rubik's Cube est un problème complexe qui a beaucoup été étudié d'un point de vue combinatoire. Certains algorithmes réussissent à déterminer une solution exacte à ce problème de résolution, notamment l'algorithme de Kociemba. Ce type d'algorithme utilise l'algèbre combinatoire pour résoudre ce problème de façon déterministe. Cependant, dans ce projet, on cherche à obtenir une solution satisfaisante sans avoir à « coder » l'ensemble des possibilités.
Dans un premier temps, nous allons explorer les données et définir la nature de notre problème à savoir le type de notre variable d'intérêt.
On implémentera tout d'abord un algorithme de forêts aléatoires puis un algorithme de SVM avant de comparer nos résultats avec la méthode « benchmark » proposé dans la description du projet. Notre objectif est de comparer ces différentes méthodes et de tenter d'expliquer les éventuels mauvais résultats.
Pour finir on explorera une méthode plus adaptée à ce genre de problème : le Reinforcement Learning.

Exploration des données

Le jeu de données est constitué de 1,8 millions de lignes. Il y a 24 variables explicatives qui correspondent aux couleurs des petites facettes du cube. Chaque facette est encodée d'un chiffre allant de 1 à 6 représentant les 6 différentes couleurs. La variable d'intérêt indique le nombre de coup minimum nécessaire à la résolution du Rubik's Cube. Cette variable prend des valeur de 1 à 14 dans notre jeu de données et sont répartis de manière très hétérogènes (c.f **Figure 1**).
On représente dans la figure 2, un résumé des valeurs prises par les variables explicatives et on remarque que 3 de ces variables sont fixes pour toutes les observations. Cela implique qu'un éventuel problème de symétrie dans notre jeu de données est à exclure.
Il est à noter qu'il n'y a aucune données manquantes dans le jeu de données.
Etant donné la taille de notre échantillon, nous allons sous-échantillonner les données en essayant de respecter les mêmes proportions des classes.

```
: distance
  1     0.000002
  2     0.000007
  3     0.000033
  4     0.000145
  5     0.000614
  6     0.002441
  7     0.008997
  8     0.031068
  9     0.098120
  10    0.253279
  11    0.367663
  12    0.212984
  13    0.024572
  14    0.000075
  Name: count, dtype: float64
```

Figure 1. Tableau de fréquence des « distances » à la résolution dans le jeu de données

	ID	pos0	pos1	pos2	pos3	pos4	pos5	pos6	pos7	pos8	...	pos14
count	1.837079e+06	1.837079e+06	1.837079e+06	1.837079e+06	1.837079e+06	1.837079e+06	1.837079e+06	1837079.0	1.837079e+06	1.837079e+06	...	1837079.0
mean	9.185390e+05	3.286525e+00	3.285119e+00	3.285350e+00	3.284730e+00	3.286199e+00	3.284253e+00	6.0	3.286448e+00	3.284806e+00	...	4.0
std	5.303192e+05	1.694338e+00	1.694928e+00	1.695366e+00	1.693529e+00	1.694059e+00	1.693691e+00	0.0	1.695201e+00	1.694212e+00	...	0.0
min	0.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	1.000000e+00	6.0	1.000000e+00	1.000000e+00	...	4.0
25%	4.592695e+05	2.000000e+00	2.000000e+00	2.000000e+00	2.000000e+00	2.000000e+00	2.000000e+00	6.0	2.000000e+00	2.000000e+00	...	4.0
50%	9.185390e+05	3.000000e+00	3.000000e+00	3.000000e+00	3.000000e+00	3.000000e+00	3.000000e+00	6.0	3.000000e+00	3.000000e+00	...	4.0
75%	1.377808e+06	5.000000e+00	5.000000e+00	5.000000e+00	5.000000e+00	5.000000e+00	5.000000e+00	6.0	5.000000e+00	5.000000e+00	...	4.0
max	1.837078e+06	6.000000e+00	6.000000e+00	6.000000e+00	6.000000e+00	6.000000e+00	6.000000e+00	6.0	6.000000e+00	6.000000e+00	...	4.0

rows × 25 columns

Figure 2. Description des variables explicatives dans le jeu de données

Comme on peut le voir sur la figure 1, un des problèmes majeurs de cette étude est l'hétérogénéité des classes.
Plusieurs méthodes sont considérés dans la littérature. L'une d'entre elles est le sur-échantillonnage et pour cela nous avons besoin de pouvoir générer de nouvelles observations. Le seul problème est que la « map » qui relie les variables aux facettes du cube est inconnue. Une première approche a été d'identifier les 3 observations de l'échantillon qui se situe à un coup de la résolution. A partir de là, l'idée est d'utiliser le résultat obtenu lors de l'explorations des données, c'est à dire que 3 facettes du cubes sont fixes. Ainsi on peut trouver la map du cube et ainsi générer des nouvelles observations en respectant les règles de mouvement autorisées.

Choix du cadre d'étude : Régression VS Classification Multi-Label

On cherche à justifier le choix de la nature de notre variable d'intérêt, à savoir une variable quantitative ou qualitative.
Le problème se résume à estimer un nombre de coup minimum à la résolution du cube à partir d'un état donné, la variable d'intérêt prend des valeurs discrète entière. Pour considérer notre problème comme une classification multi-label il nous faut montrer que le nombre de valeurs prises par Y est borné. Il s'avère que pour un Rubik's Cube 2x2x2 le nombre de coup maximum à la résolution du cube est de 14 quelque soit l'état de départ du cube. Ce nombre, appelé le « God's Number », est un résultat d'algèbre combinatoire; l'idée étant que le nombre de mouvement de rotation entre les différents états du cube est borné. **[2]**
Cependant étant donné la métrique MAE préconisé dans la description du projet, on va tout de même estimer un modèle de régression pour le comparer aux autres modèles.

Limites de calculs

La taille du jeu de données pose de gros problème au niveau des temps de calcul et notre machine ne peut pas en un temps raisonnable (< 6 heures), effectuer des algorithmes classiques d'apprentissage statistique. On a donc sous-échantillonner le jeu de données. On travaille dans la suite avec un échantillon de 10000 observations pour notre échantillon d'apprentissage et 2000 observations pour notre échantillon de test.

Forêt aléatoire :

La forêt aléatoire est un outil puissant pour établir une fonction de prédiction à l'aide d'arbre de décision. Elle permet, entre autres, de réduire la variance de notre prédiction. Dans sa forme la plus simple, on agrège les résultats de plusieurs arbres de décisions construits à partir d'un certain nombre de variables explicatives sélectionnées aléatoirement.

L'optimisation des paramètres se fait de manière similaire à celle d'une régression Ridge ou Lasso. A la place d'une grille de différentes valeurs possible pour un terme de pénalité, on utilise une grille avec plusieurs arguments :

- La profondeur maximale d'un arbre
- Le nombre d'arbres
- Le nombre maximale de variable explicatives (souvent la racine de ce nombre)
- Une variable booléenne Bootstrap
- Un nombre minimum d'observations pour pouvoir effectuer une coupe
- Un nombre minimum d'observations dans une feuille de l'arbre

L'idée est donc d'estimer une fonction de prédiction par forêt aléatoire pour chaque combinaison de ces variables, et de ne retenir que la meilleure combinaison pour notre modèle.
La fonction *RandomizedSearchCV* dans la librairie Scikit learn sur python est très pratique pour effectuer ces différentes validation croisées. On peut en effet utiliser la validation croisée par la méthode V-fold. Dans la suite on utilise 100 différentes validations croisées avec V=10.

Régression :
Le critère que l'on utilise est celui préconisé dans le projet à savoir la Median Absolute Error (MAE), puis on tronque à l'entier le plus proche.

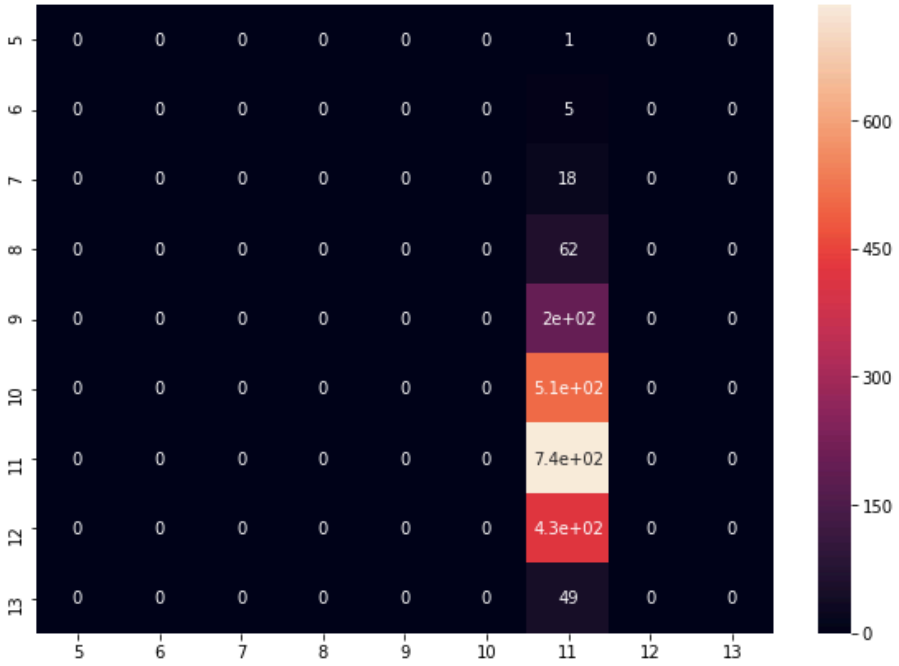


Figure 3. Matrice de confusion pour une classification par forêt aléatoire sur les 9 classes du sous-échantillon.

Classification :

Le critère utilisé est l'indice de Gini qui va capturer l'hétérogénéité au sein des groupes.

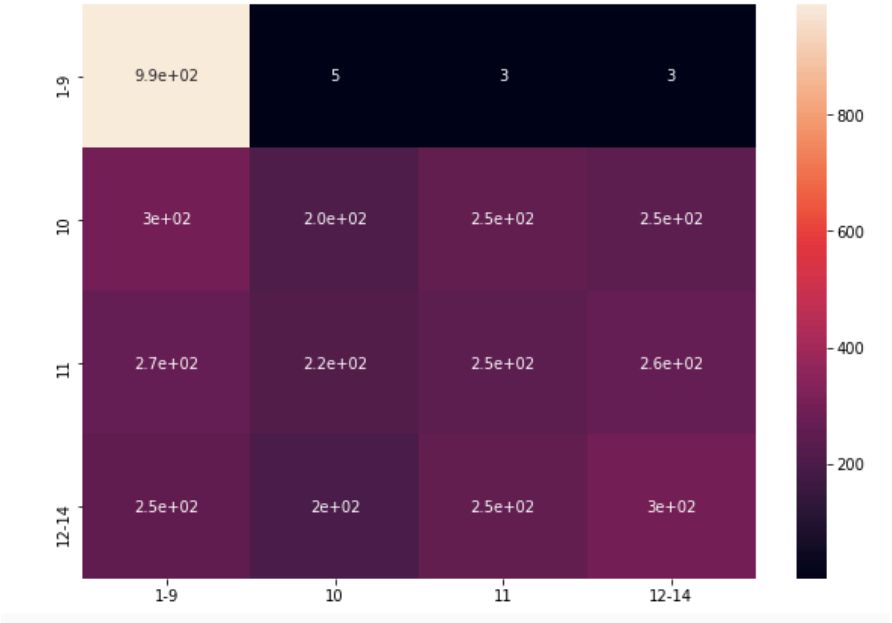


Figure 4. Matrice de confusion pour une classification par forêt aléatoire sur 4 classes : (1-9) , (10) , (11) , (12-14)

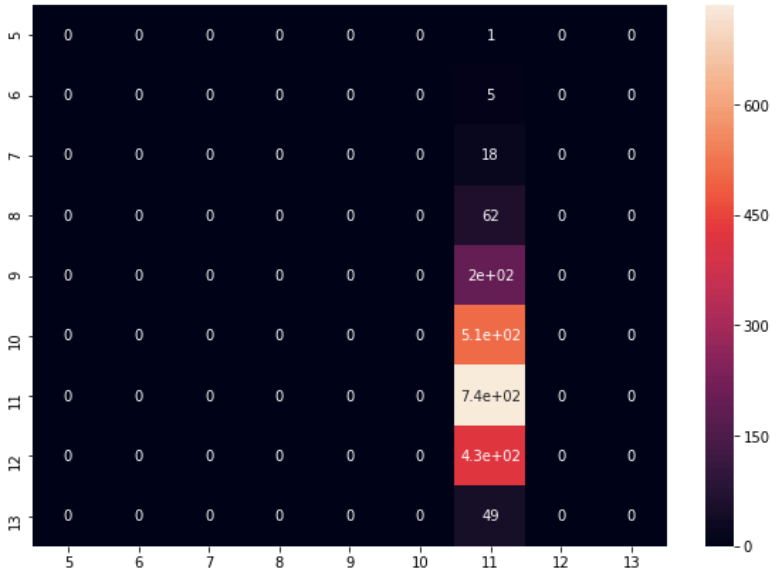


Figure 5. Matrice de confusion pour une classification par forêt aléatoire du sous-échantillon (9 classes)

On va tout d'abord effectuer une forêt aléatoire sur 9 classes (les pourcentages des autres est tellement faible pour 10000 observations qu'on les retire). Ensuite on va essayer de diminuer le nombre de label en réduisant le nombre de classes. On regroupe les classes sous représentées entre elles et les autres sont laissés intactes. Pour cette configuration, on génère un sous-échantillon de 10000 observations où les 4 classes sont représentées de manière équilibrée.

Métriques	Forêt régression (avec troncation)	Forêt classification 9 classes	Forêt classification 4 classes
Accuracy score	0,342	0,367	0,4355
Unbalanced score	0,156	0,111	0,435

On remarque que les résultats sont assez médiocres pour la prédiction; les modèles n'arrivent pas à capter l'information des mouvements de rotations du Rubik's Cube et produisent donc beaucoup d'erreurs. On peut supposer deux raisons principales à ce phénomène. La première étant que les critères qu'on utilise sont inefficaces pour notre problème, dans le sens où un vote majoritaire par exemple, ne suffit pas à séparer de manière optimale les observations. Cela peut être dû à une éventuelle symétrie du problème, en effet les états du cube peuvent être redondant et donc les variables explicatives ne donnent pas beaucoup d'information réelle sur la distance à la résolution. On peut imaginer par exemple 2 configurations différentes qui ont la même distance à la résolution. La seconde raison est le déséquilibre du jeu de données. On voit notamment sur les figures 4 et 5 que les mauvaises classifications sont très nombreuses.
L'idée de réduire le nombre de classe était d'effectuer une première classification sur ce modèle puis si on classifie une observations dans les « groupements » , effectuer des classifications sur ces mêmes groupements entre les classes sous- représentées.

SVM

Un des problèmes fondamentaux lié à ce jeu de données, et la répartition des hétérogènes des différentes classes.
En effet en présence d'un faible nombre d'observation, et surtout en présence d'une majorité de représentant d'une classe, l'algorithme va souvent préféré classifié l'observation à la classe majoritaire étant donné que cela permet d'avoir le plus faible risque.

Nous allons à présent nous intéresser aux SVM ("Support Vector Machine"). Nous cherchons toujours à faire de la classification multi-classes (14 classes). Il existe deux grandes méthodes d'adaptation aux problèmes multi-classes : l'approche OVO (One-VS-One) et l'approche OVR (One-VS-Rest). Cela veut dire que dans l'approche OVO, (14*13)/2= 91
classifieurs sont entrainés. La bonne classe sera alors celle la plus représentée. Cette méthode est très efficace même si elle est très couteuse en calculs.

Dans l'approche OVR (parfois appelé OVA [one-vs-all]) 14 classifieurs seront entrainés, et chacun sera "spécialisé" dans une classe précise. Pour classer une nouvelle entrée, on regarde à quelle catégorie la nouvelle entrée est le plus probable d'appartenir.
L'inconvénient de cette méthode est que, quand on commence à avoir beaucoup de données, on a beaucoup plus de données dans le "reste" que dans notre classe en question. Quand le déséquilibre entre la quantité de données dans les deux catégories est trop fort, un SVM obtient de moins bons résultats.

Tout au long de cette analyse on préférera la méthode "OneVSOne".
On définit une fonction « erreur » qui retourne l'erreur absolue de nos prédictions. Testons différons kernels et évaluons C par validation croisée.
On choisit arbitrairement C = [0.15,0.3,0.5,0.62,0.7,0.75,0.82,0.9, 1]

Kernel Lineaire:
Paramètres: kernel='linear',gamma='auto',decision_function_shape='ovr')

Kernel Polynomial :
Paramètres: degré = 3,gamma='scale', decision_function_shape='ovo'

Kernel Gaussien :
Paramètres: kernel='rbf',gamma='scale',decision_function_shape='ovo'

Kernel Sigmoid
Paramètres: kernel='sigmoid',gamma='auto',decision_function_shape='ovo')

Récapitulatif de sortie pour le kernel Gaussien, celui qui a eu la meilleure performance avec nos données :

Métriques d'évaluation	11	12	Reste
Accuracy	0,36	0,26	0
Recall	0,89	0,01	0
F1_score	0,52	0,02	0

Pour le Kernel Gaussien, on observe grâce à la validation croisée que la précision décroit avec l'augmentation de la pénalité C. Cela s'explique par le fait que plus la pénalité est grande plus on introduit un biais qui baisse la précision de notre classifieur.
Là encore, on remarque la conséquence du grand déséquilibre des données : le classifieur, pour minimiser son erreur, préfère prédire le label le plus systématiquement.

Clustering

La méthode Benchmark du projet est celle d'un clustering. On implémente un algorithme des K-moyennes sur les variables explicatives, puis on associe une valeur du nombre de coups minimum à la résolution à chaque cluster, en se basant sur le principe du vote majoritaire. Plus précisément, chaque observation d'un cluster a un nombre de coups minimum associé; on effectue un vote majoritaire sur ces « classes » pour assigner la valeur la plus présente au sein d'un cluster, à ce même cluster. On prédira Y pour une nouvelle observation par la classe associée au centroïde le plus proche. On précise que la proximité est au sens de la distance euclidienne. Là encore, une métrique plus adaptée au problème fournirait un résultat certainement plus prometteur.

On peut essayer d'interpréter ces clusters comme des regroupements d'états du cube proches des uns des autres d'une certaine manière. En effet, si on effectue un mouvement qui fait passer l'état du cube de A à B, alors 8 faces restent inchangées et 12 sont différentes. Dans le pire scénario, c'est à dire si les deux états A et B sont totalement opposés (si c'est possible) alors, toutes les face 1 sont les faces 6 dans l'état B et la distance entre les deux états est maximale.

Nombres de clusters	**k=20**	**k=50**	**k=100**
Accuracy	0,347	0,3556	0,349
Balanced score	0,105	0,109	0,110

De même les résultats ne sont pas probants pour le Clustering, un autre type de méthode doit être envisagé.

Pour aller plus loin : Reinforcement learning

Pour conclure, une méthode particulièrement efficace et adaptée à ce type de problème se base sur le Renforcement Learning.
Pour ce faire, nous avons décidé d'utiliser les techniques d'apprentissage par renforcement couplées à une variante de l'algorithme de MCTS (Monte-Carlo Tree Search).

L'apprentissage par renforcement est une classe de problèmes d'apprentissages automatiques. Le but est d'apprendre, par le biais d'expériences, comment atteindre un objectif prédéfini. Dans le cas du Rubik's cube, l'objectif est de réussir à ce que chacune des 6 faces du cube soit d'une seule couleur, à partir d'un cube désordonné. Pour cela, on a le droit à de faire pivoter chacune des faces indépendamment des autres.
Dans ce type de problème, la difficulté réside dans le fait que l'on dispose d'un espace d'états possibles énorme couplé avec un espace d'états « objectif » très petit (1 seul état objectif dans notre cas). En conséquence, un algorithme classique d'apprentissage par renforcement aura du mal à identifier la meilleure action à effectuer dans certaines situations (considérer l'ensemble des états possibles serait ridiculement long). Cette méthode ne parvient donc pas, en un temps raisonnable, à trouver une stratégie efficace à la résolution du problème.

C'est pour cette raison que nous avons décidé de développer une méthode combinant l'apprentissage profond par renforcement et le MCTS. Le fait d'utiliser à la fois l'exploration des états du monde et l'exploitation des états déjà observés permet de pallier partiellement au problème précité. En effet, l'algorithme pourra converger plus rapidement vers une solution satisfaisante.

Description de l'algorithme

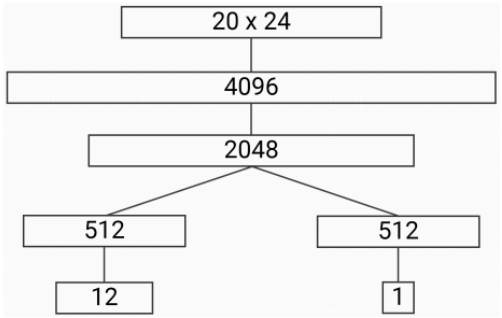
Section 1 : Réseau de neurones

On définit l'espace des actions par A. Les actions possibles sont les mouvements de rotation des différentes faces dans le sens des aiguilles d'une montre et dans le sens inverse (ce dernier est dénoté par le « ‘»). La rotation de la face de droite est désigné par R (Right), celle de gauche par L (Left) etc…

$$a \in A = \{ U, U', F, F', B, B', L, L', R, R', D, D' \}$$

On définit l'espace d'états du monde par S et on note s un de ses représentants.
La récompense associée à l'état du cube résolu est 1, pour les autres états c'est -1.

Créer une architecture de réseau de neurones qui va nous permettre d'évaluer la valeur d'un certain état du cube et la politique associée à cet état.



Input : Etat du cube modélisé en un vecteur 20x24

Couches cachées

Output : Valeur de l'état du cube donné en input

Output : Vecteur de politique associé à chacune des 12 actions possibles

Théorie - Entrainement réseau de neurones :

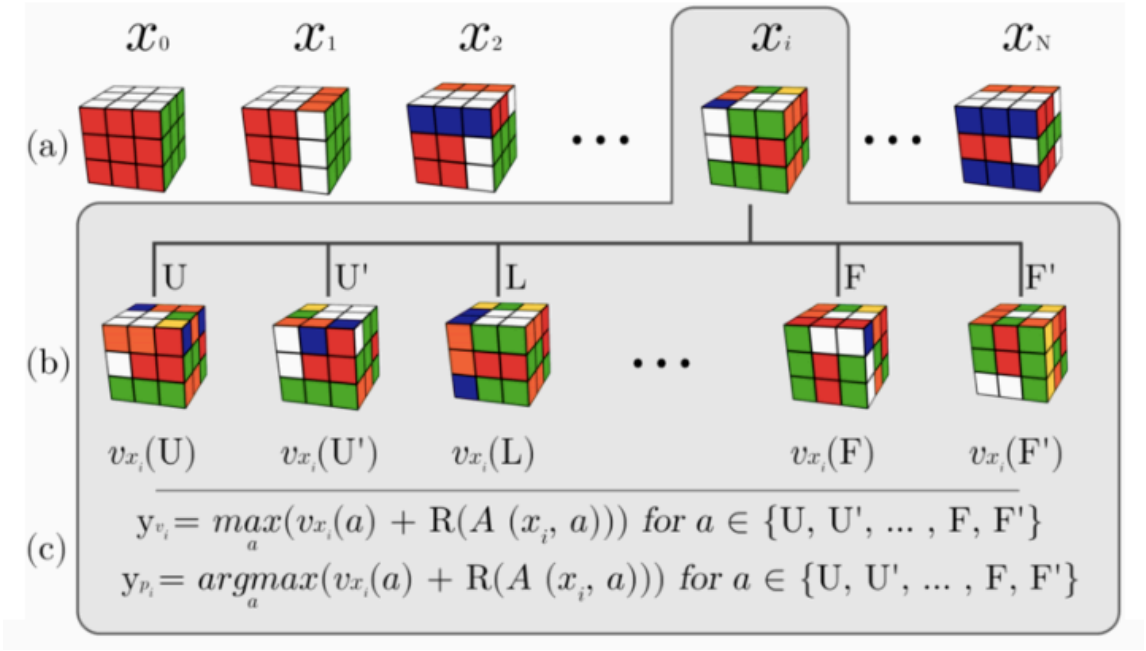
La méthode d'entrainement que l'on va utiliser est appelée « Itérations auto-didactique ». A partir de l'état du cube résolu, on va générer une séquence (de taille prédéfinie N) d'états en appliquant de manière aléatoire des actions (mouvements de rotation du cube). Pour chacun des états de la séquence, on applique la procédure suivante :

- Appliquer toutes les actions possibles à l'état du cube
- Passer en argument les 12 états ainsi obtenus dans le réseau de neurones, pour en déterminer la valeur
- La valeur de l'état passé en argument est calculée ainsi :

$$v_i = max_a(v(s_i, a) + R(s_i, a))$$

- La politique associé à l'état passé en argument est calculée ainsi :

Schéma de la procédure :



NB: L'esprit reste le même pour un Rubik's Cube 2x2x2

Section 2 : Monte Carlo Tree Search

Objectif de cette section : Présenter une méthode qui, couplée au réseau de neurones vu précédemment, va permettre de déterminer une « meilleure » action à appliquer à un état du cube. C'est la partie qui va ajouter la composante d'exploitation d'information.

Théorie
On va s'intéresser à une famille d'algorithme que l'on nomme « Monte Carlo Tree Search ». Plus précisément, à la méthode UCT (Upper Confidence bound1 for Trees). La méthode consiste à prendre en considération l'output de notre réseau tout en pénalisant les actions redondantes mais aussi explorer, dans une certaine situation, les états proches de notre état courant.

On modélise sous forme d'arbre notre problème, où la racine est l'état s que l'on considère, les noeuds correspondent à des états et les arrêtes à des actions. Il y a deux possibilités :

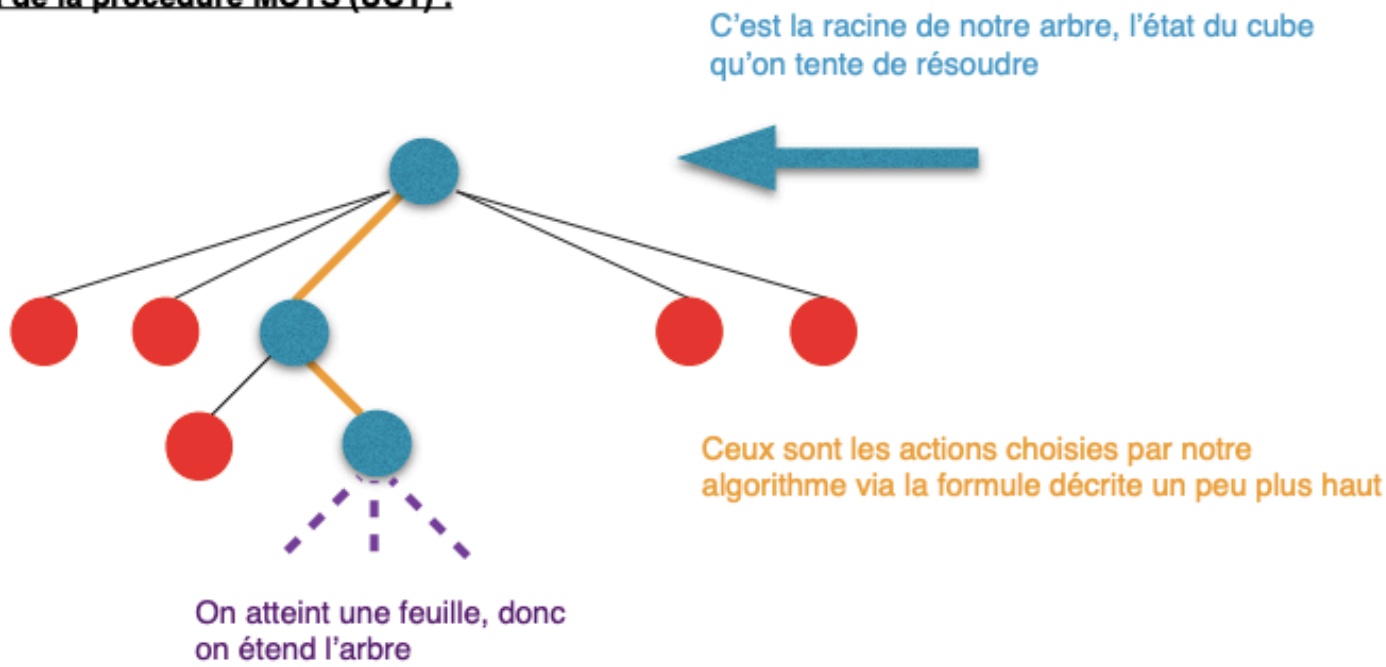
- s est une feuille, i.e. qui n'a pas d'enfants connus (dans le sens où ces états n'ont pas été visité lors de l'entrainement du réseau ou par cet algorithme). Dans ce cas on ne dispose pas d'information concernant l'action à effectuer. On étend donc l'arbre, en appliquant les actions possibles, et on stocke la valeur de s retourné par le réseau.

- s n'est pas une feuille et on connait donc ses enfants. Le problème est maintenant de déterminer si on doit suivre ou non le résultat que l'on connait (à savoir la politique donnée par le réseau). On va introduire une fonction qui va nous permettre de prendre cette décision.

$$A_t = argmax_a(U_{s_t}(a) + W_{s_t}(a)) \quad \text{où} \quad U_{s_t}(a) = cP_{s_t}(a)\frac{\sqrt{\sum_{a'} N_{s_t}(a')}}{1 + N_{s_t}(a)}$$

Deux éléments importants dans cette formule. Le premier est que N(s,a) désigne le nombre de fois où l'on a pris la décision « a » quand on était à l'état « s ». On pénalise les chemins trop fréquentés. Le deuxième est que W(s,a) désigne la valeur maximale retournée par le modèle parmi tout les enfants de « s » issus de la branche « a ».
P(s,a) est la valeur de la politique de l'état « s » pour l'action « a ».

Schéma de la procédure MCTS (UCT) :



(**NB** : la racine possède 12 enfants, le noeud bleu de la seconde ligne aussi, et ainsi de suite pour les noeuds qui ont des enfants)

On répète ce processus tant que la solution n'est pas trouvée ou bien le temps imparti est écoulé.
Une fois arrivé à l'état objectif (cube résolu), on applique un algorithme de type BFS (parcours en largeur) pour trouver le plus court chemin à partir de la racine de l'arbre.

Nous n'avons malheureusement pas pu implémenter la méthode, qui aurait probablement donner de bien meilleurs résultats que ceux obtenus avec les méthodes étudiées.

Conclusion

Les méthodes que nous avons utilisée ne s'avèrent pas très concluantes et cela parait naturel étant donné la structure géométrique particulière du Rubik's Cube. En effet les variables ne peuvent pas être traités efficacement en les considérant non-liées. Il y a une relation algébrique entre les observations des états du cube. Un état proche d'un autre en termes de coup minimum à la réalisation n'est pas caractérisé par une proximité de la distance euclidienne entre deux observations, mais probablement par une distance plus sophistiquée qui arriverait à capturer l'information liée aux différents mouvement du cube possible.
Une approche tentant de construire une métrique adaptée à ce problème donnerait des résultats beaucoup plus prometteur. Aussi la méthode de Renforcement Learning semble très adapté au problème et est sans doute la plus adéquate pour résoudre ce problème d'apprentissage statistique.

Références :

[1] The man who found god's number - Mathematical Association of America , B. Boyd
[2] Model evaluation : quantifying the quality of predictions - Scikit Learn Library
[3] Classification of imbalanced data : a review - YANMIN SUN, ANDREW K. C. WONG, MOHAMED S.KAMEL
[4] McAleer, Stephen & Agostinelli, Forest & Shmakov, Alexander & Baldi, Pierre. (2018). Solving the Rubik's Cube Without Human Knowledge.
[5] Reinforcement Learning : an introduction - Richard S. Sutton and Andrew G. Barto 2014-2015