AOS2 - Création d'une Intelligence Artificielle pour le poker

Jean-Baptiste SIX
Yann MAILLET

Mercredi 16 Janvier 2019

1 Remerciement

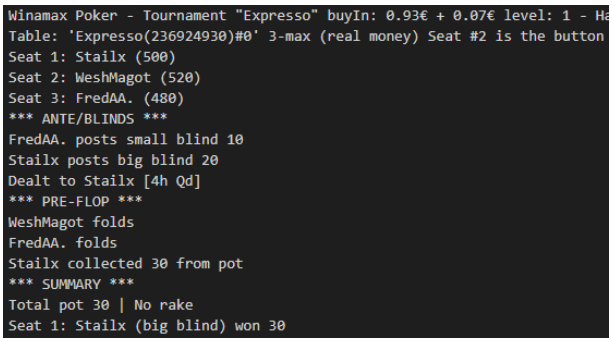
Nous tenons à remercier tout particulièrement Théo qui a accepté nous partager l'historique de ses mains, et qui sans lui rien de tout cela n'aurait été possible.

2 Introduction

Dans le cadre de l'UV AOS 2, nous devions réaliser un projet en rapport avec l'apprentissage profond (*Deep Learning*). Nous avons choisi de faire un bot capable d'imiter les décisions d'un joueur de poker. Pour cela, nous avons fixé un cadre : la variante du poker Hold'em no limit à 3 joueurs. Nous avions pour objectif de faire un classifieur, qui a partir d'un état donné peut prédire une action à réaliser : Se coucher, Suivre, Relancer (*Fold*, *Call*, *Raise*).

3 Récupération des données

Pour les problèmes d'apprentissage profond, une grande quantité de données est nécessaire. Nous avons donc demandé à un bon joueur de poker en ligne de nous partager l'historique de ses mains. Ce n'est pas moins de 3000 parties que nous avons pu récupérer de cette manière.
Cependant les données brutes ne sont pas exploitables : les parties sont stockées en fichier *.txt* et de la forme suivante :



Nous avons donc dû créer un script permettant de parser tous les fichiers. Ce script récupère en entrée les 3000 fichiers contenants toutes les parties du joueur et retourne à la fin 8 tableaux contenant toutes les décisions prises par le joueur à chaque instant. Les tableaux sont séparés par nombre de joueurs (3 ou 2) et par phase de jeu ($Préflop$, $Flop$, $Turn$, $River$). Ils contiennent toutes les informations récupérables au moment de la prise de décision :

- Position du joueur
- Cartes du joueur
- Carte communes ($Board$)
- Nombre de jetons ($Stack$) de chaque joueur
- Valeur des *blinds*
- Mise (Bet) de chaque joueur
- Valeur du pot

Ainsi que la décision du joueur :

- Action à réaliser ($Fold$, $Call$, $Raise$)
- Valeur de la mise relative (x2, x3, ..., $All-in$)

4 Formatage des données

Suite au parsing des données décrit dans la partie précédente, nous obtenons donc 8 formats de données différents (4 phases de jeu, 2 types de situation). Cependant les données ne sont pas bien réparties en fonction des phases de jeu, par exemple voilà la répartition pour le jeu à 3 joueurs :

- total des mains : 55676
- preflop : 30650 — 55 %
- flop : 12165 — 21 %
- turn : 7682 — 13 %
- river : 5179 — 9 %

Faire autant de classifieurs pourrait causer une baisse de précision pour les phases sous-représentées.
Par conséquent, nous décidons de réunir les 4 phases de jeu en 1 seul tableau.
Nous gardons la distinction entre les deux types de situations (2 joueurs / 3 joueurs) pour deux raisons :

- le style de jeu est différent selon la situation
- les variables d'entrée sont différentes

Pour se faire nous créons 4 variables binaires correspondant aux 4 phases de jeu, ainsi la représentation de la phase de jeu $flop$ est $[0, 1, 0, 0]$ par exemple.
Nous procédons de la même manière pour représenter la position (*button*, *smallblind*, *bigblind*), ainsi le fait d'être small blind est exprimé par $[0, 1, 0]$.
Théoriquement le même raisonnement aurait pu être appliqué pour représenter les cartes, cependant nous nous heurtons à un problème : Il y a 52 possibilités pour chacune des deux cartes du joueur, et pour chacune des 5 cartes communes (le *board*). Ce qui fait un total de 364 variables binaires simplement pour la représentation des mains, il resterait également à représenter la force de la main (*paire*, *brelan*, *couleur*...).
Pour palier à ce problème nous décidons de remplacer cela par la probabilité de gagner, cette dernière étant calculée à l'aide d'une simulation de Monte-Carlo.
Le calcul a du être effectué pour l'ensemble du dataset, soit 123 705 fois.
Après l'ensemble des transformations décrites, nous obtenons 2 jeux de données ayant les caractéristiques suivantes :

- dataset 2 joueurs : 68029 samples / 14 variables
- dataset 3 joueurs : 55676 sample / 17 variables

5 Réseaux de neurones

Dans cette partie nous détaillerons seulement le classifieur dans le cas 3 joueurs, le cas 2 joueurs étant similaire.

5.1 Opérations sur le dataset

Pour commencer nous avons séparé le dataset en 2 parties :

- les variables explicatives : 16 variables
- la variable à prédire : 1 variable

Nous obtenons donc les tableaux de données *mains_3J_X* et *mains_3J_Y*.
Nous normalisons ensuite les données à l'aide du StandardScaler de la librairie sklearn. Cependant nous normalisons seulement les données non-binaires.
Enfin, nous divisons de nouveau ces tableaux en 2 parties : un ensemble d'entraînement (90%) et un ensemble de test (10‘%).
Après avoir crée un *dataloader* pour itérer sur les données, ces dernières sont enfin prêtes à être utilisées pour entraîner un réseau de neurones.

5.2 Création du réseau de neurones

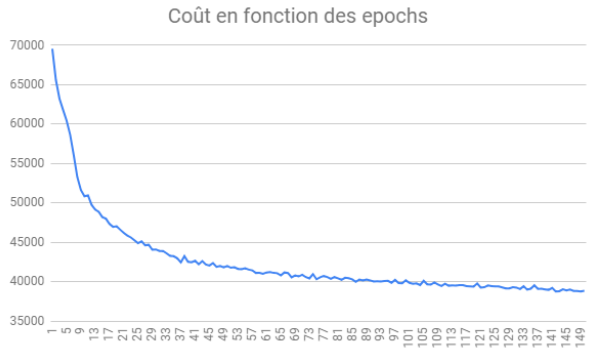
Nous avons effectué plusieurs tests pour déterminer quels étaient les meilleurs paramètres du réseau (fonction de coût, technique d'optimisation, batch size, largeur, profondeur, fonction d'activation...) :

- L'entropie croisée est la fonction de coût la plus efficace
- la descente de gradient stochastique donne les meilleurs performances, mais d'autres méthodes (Adam) donnent quasiment les mêmes performances.
- Changer la taille du batch semble avoir une influence seulement sur la vitesse de calcul et de convergence, pas sur la précision.
- Il semble que le réseau arrive à bien généraliser avec seulement 3 couches de neuronnes : les performances n'augmentent pas en augmentant la profondeur.
- La largeur n'a que peu d'effet, cependant nous avons noté que le réseau était un peu plus performant avec une largeur de l'ordre de 100.
- la fonction d'activation Relu est celle qui donne les meilleurs résultats

Pour le learning rate nous utilisons l'approche suivante : nous parcourons un tableau de valeur de manière décroissante (*[0.1, 0.05, 0.01, 0.005, 0.001]*), et nous effectuons 10 epochs pour chacune de ces valeurs du learning rate. Cela

permet d'avoir une convergence rapide et précise (environ 2 min pour entraîner le réseau).

Voilà un graphique mettant en évidence la convergence :



Pour un réseau ayant l'architecture suivante (architecture finale) :

```
NN_3J(
  (input): Linear(in_features=16, out_features=100, bias=True)
  (hidden): Linear(in_features=100, out_features=100, bias=True)
  (final): Linear(in_features=100, out_features=3, bias=True)
)
```

Notre modèle ne fait pas de sur-apprentissage car nous obtenons la même précision sur les données de test et d'entraînement. Nous obtenons au final une précision de 80%. Compte tenu du fait qu'il y ai seulement 3 classes, ce n'est pas un résultat très impressionnant. Cependant il faut prendre en compte certains paramètres :

- Les labels ne sont pas une vérité absolue : en effet plusieurs décisions peuvent être considérées comme raisonnables au poker, il n'y a pas de réponse absolue.
- le dataset est constitué de mains jouées par un même joueur mais sur plusieurs mois, son jeu a donc pu évoluer au fur à mesure des mois et donc comprendre quelques incohérences.

Il y a donc très sûrement une limite de précision due aux données. Compte tenu de ces remarques, les 80% de précision atteints nous satisfont et nous passons à l'étape de simulation. A titre de comparaison, un SVM donne environ 67-68% de précision.

6 Test sur simulation

Nous avons trouvé sur Github une librairie en python permettant de simuler des partie de poker Hold'em no limit. Nous l'avons donc exploité afin de tester notre modèle.
Nous avons dans un premier temps créé des adversaires "basiques" pour notre IA, comme par exemple un bot qui ne fait que de suivre ($Call$), un bot qui ne fait que de relancer ($Raise$) quelque soit leur mains, et un adversaire qui joue aléatoirement entre $Fold$, $Call$, et $Raise$. Nous obtenons les résultats suivants : Sur 1000 parties jouées contre :

- 2 $Call$Bot : 72% de parties gagnées
- 2 $Random$Bot : 43% de parties gagnées
- 2 $Raise$Bot : 7% de parties gagnées

Nous précisons bien sur que le taux "normal" de victoire est de 33%, et que les meilleurs joueurs de poker (contre de vrais joueurs) atteignent environ 40% de victoire. Nous constatons que notre Bot perd presque tout le temps contre les $Raise$Bot car il va $Fold$ ses mains. En effet, une personne normale comprendrait que le joueur adversaire $Bluff$ (c'est-à-dire relance sans avoir de bonne main), cependant notre IA n'a pas de mémoire.

Nous avons ensuite réalisé un adversaire "intelligent" qui va pouvoir calculer la force de sa main, et prendre des décisions en fonction. Contre 2 adversaires de ce type, nous obtenons 56% de parties gagnées, toujours sur un total de 1000 parties jouées avec 3 joueurs. Les résultats sont très bons car notre IA a été entraînée contre des joueurs de ce type (pas trop extrêmes). Cet adversaire "intelligent" reste tout de même moins bon qu'un joueur de poker lambda.

7 Améliorations et suite du projet

En le faisant jouer contre des Bot "extrêmes" (Raise tout le temps par exemple), nous nous sommes aperçus que notre IA avait intérêt à adapter son style de jeu (par exemple $Fold$ moins souvent). Pour cela nous analysons le jeu de l'adversaire (on regarde la fréquence à laquelle il $Fold$, $Call$, ou $Raise$) puis nous appliquons une pénalité ou un bonus sur l'*output* final de notre réseau de neurone. Cela lui permet d'être plus flexible, et moins prévisible.
Par exemple en appliquant le correctif *[-4 3 0]* (moins de $Fold$, plus de $Call$) sur l'output nous obtenons contre 2 *RaiseBot* un taux de victoire de 40% (sur 1 000 parties simulées).

Par la suite, nous aimerions continuer ce projet d'IA pour le poker, en utilisant des méthodes de reinforcement learning.