Rapport de stage
Algorithmes de minimisation du regret et integrales de choquet
LABORATOIRE DE RECHERCHE EN INFORMATIQUE

Quentin Lieumont

18 Juin 2019

Sous la direction de : Johanne Cohen
Martine Thomas

Table des matières

1 Introduction

Mon stage c'est déroulé au laboratoire de recherche en informatique, son sujet était le suivant : *Algorithmes de minimisation du regret et integrales de choquet*. Je n'avais jamais fait d'informatique fondamentale (exepté IA) et n'ayant jamais vus la théorie de la mesure, il m'était d'aprhender en totalité la notion d'intégrale de choquet (découlant des intégrale de Lebesgue). C'est pourquoi, durant ce rapport, la partie mathématique théorique ne sera pas vue en detail. Organisation du document

2 Présentation générale du LRI

Le Laboratoire de Recherche en Informatique (LRI) est une unité de recherche de l'Université Paris-Sud et du CNRS. Créé il y a plus de 40 ans, le laboratoire est localisé sur le plateau du Moulon depuis début 2013. Il accueille plus de 250 personnes dont environ un tier de doctorants.

Organisés en neuf équipes, les recherches du laboratoire incluent à la fois des aspects théoriques et appliqués (ex : algorithmique, réseaux et bases de données, graphes, bioinformatique, interaction homme-machine, ect). De part cette diversité, le laboratoire favorise les recherches aux frontières de différents domaines, là où le potentiel d'innovation est le plus grand. En plus de son activité de publication (2000 publications entre début 2008 et mi 2013), le LRI dévelope de nombreux logiciels.

Pour plus d'informations, je vous invite à aller regarder sur leur site ou une présentation détaillée est continuellement tenue à jour [1].

3 Équipe GALaC

Durant mon stage, j'ai travaillé sous la direction de Johanne Cohen qui est la responsable de l'équipe GALaC.
L'équipe GALaC est une équipe du LRI composée d'environ 20 chercheurs issus du cnrs, de l'Université Paris-Sud et de Centrale Supelec qui travaillent sur des thématiques théoriques comme l'algorithmique, la théorie des graphes ou les systèmes en réseaux. Mon travail s'intègre dans cette équipe puisqu'il réside dans l'étude du fonctionnement d'un aglorithme.

4 Contexte et motivations

Le but du stage était de manipuler les intégrales de choquet et les résaux de neurones. La prédiction du prix des maisons à été étudiée grâce à une base de donnée trouvée sur internet [2]. On pourait tout simplement multiplier le prix au metre carré par la superficie de la maison mais on verra par la suite que cette technique est loin d'être optimale.

5 Un petit point théorique...

Afin de mieux comprende les interactions entre les differentes notions, un petit point théorique est necessaire :

— Compréhention du fonctionnement d'un réseau de neurones
— Fonctionnement des intégrales de choquet
— Pourquoi les integrales de choquet associées aux résaux de neurones sont particulierement efficace pour résoudre certains problèmes

5.1 Réseaux de neurones

Les médias parlent souvent d'intéligence artificielle et de réseaux de neurones, ces deux notions sont radicalements différentes mais elles sont souvent mélangées et confondues sur la place publique...

L'intéligence artificelle est un concept informatique, un paradigme de programation, cette notion n'a pas été abordée durant mon stage, il n'en serra pas question ici. Cependant si vous voullez en savoir plus, je vous redirige vers la chaine youtube de Lê Nguyên Hoang. Anciennement chercheur en mathématique et aujourd'hui vulgarisateur sur internet et à l'EPFL ses vidéos sont à regarder sans moderation [3].

Un réseau de neurones est une architecture informatique inventée en 1950 et remis à la mode grâce aux travaux de Yan Le Cun durant les années 1980 permetant de faire des regressions de fonctions, de la généralisation et de l'optimisation. Il est composé, comme son nom l'indique, de neurones mis en réseau grace à des connexions.

5.1.1 Un neurone

Un neurone est une unité de base du réseau, il se decompose en trois phases.
image 1 neurone

L'entrée : Un neurone prend des informations en entrée, de manière générale un nombre réel entre 0 et 1 mais d'autres objets sont envisageables (image, pixel, son...). Il n'y a pas de nombre minimum ou maximums (on pourait imaginer un neurone ne prenant pas d'entrée, ou au contraire en prenant une infinité), ni de contrainte sur les differents objets.

Exemple(s) :

Un neurone prenant :
- les trois valleur de coulleur d'un pixel (entier entre 0 et 255).
- les trois valleur de coulleur d'un pixel (réel $[0, 1]$).
- une sequence ADN en entrée

On note le vecteur entrée X et chacun de ses element $x_1 \dots x_i \dots x_n$.

La fonction interne : Le réseau vas alors faire un calcul à partir des entrées et de poids, des valleurs définies pour chaque neurones qui peuvent varier durant l'apprentissage.
Le vecteur poid se note W et chacun de ses element $w_1 \dots w_i \dots w_n$.

Exemple(s) :

$f(X) = W.X = \sum_{i=1}^{n} w_i \times x_i$
$f(X) = e^{w_1+x_1} \times w_2$
$f(X) = \max(X)$
$f(X) =$ "nombre de A sur les w_1 premières bases de x_1"
(avec x_1 une sequence ADN)

On note la fonction interne $f(X)$ (avec X le vecteur d'entrée).

La fonction d'activation : Comme vus precedement, ces neurones sont mis en résau, il est donc interessant d'avoir une norme pour l'entré et la sortie afin de pouvoir lier des neurones entre eux sans distinctions.
La norme qui a été choisie est, comme précedement cité, un réel entre 0 et 1.

Or, il peut être remarqué que les fonctions ci dessus ne renvoient pas forcement des nombre entre 0 et 1... On utilise donc la fonction d'activation afin de normaliser les sorties.

Exemple(s) :

Pour le cas precedent sur l'ADN : $f(X)/w_1$
Pour une fonction dans $\mathbb{R}$: fonction sigmoide
Pour une fonction dans $[0, 1]$: fonction identité

On note la fonction d'activation f_{act}.

Pour resumer : Un neurone prend *des entrées*, les passe dans sa *fonction principale*, puis le résultat dans la *fonction d'activation.*
Formellement, on obtient la formule suivante :

$$f_{act}(f(X)) \tag{1}$$

Le tout dépendant bien évidement du vecteur W que l'on peut faire varier afin de modifier la fonction du neurone.

Exemple(s) :

Prenons un neurone avec deux entrées : x_1 et x_2,
de fonction principale $f(X) = X.W$
et de fonction d'activation identité.
Si on a $W = (1, 1)$, ce neurone fait une somme.
Mais si $W = (0.5, 0.5)$ ce neurone fait une moyenne.

5.1.2 Un réseau

Une fois que nous avons de nombreux neurones faisant chacuns des actions bien spécifiques, on voudrais les relier afin de gérer des comportements plus complexes comme faire une somme de moyenne (ou controller un drone, prédire les mouvements boursiers...).

Il sufit alors de relier les neurones entre eux, de maniere générale de manière linéaire de gauche à droite. De nombreuse manière de relier les neurones existent (recurent, convolution...), ici on ne parlera que de la connexion dite "Dense" ou "fully connected" : le réseau se découpe en n couches de neurones, chacunes composée de neurones dont les entrées sont les sorties des neurones le la couche précédente.

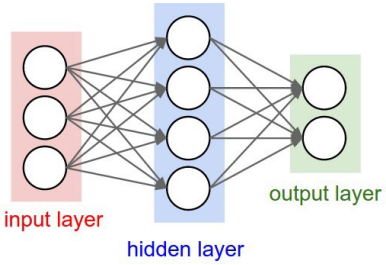

FIGURE 1 – Réseau dense simple

techburst.io/experiment-finding-objects-with-a-neural-network-caa4cec7d2c4

Comme on peut le voir sur la figure, les neurones les plus à gauches sont les neurones d'entrée (capteurs) et ceux les plus à droite ceux de sortie (controle des moteurs, affichage d'une note...). Tout les autres neurones sont "cachés", il n'intéragissent pas directement avec l'environement.

5.1.3 L'apprentissage

Jusqu'ici on a vus comment creer une architecture modulable permetant de générer une fonction à paramètres. Mais une question se pose toujours :
Quel est l'intéret ? Pourquoi ne pas directement écrire la fonction "en dur" ?
La réponse tient en trois mots : *Descente de gradient stochastique* (ou SGD en anglais).
Ce concept est ce qui fait que les réseaux de neurones sont les architectures favorites des data scientists et des chercheurs en IA.

L'idée est assez simple, on recherche une fonction générale dont on ne connais que certaines valeurs discretes. On vas commencer par définir une fonction nomé "loss function" qui décrit la précision du réseau actuel : elle prend en entrée deux valeurs : la valeur théorique et la valeur obtenue. elle retourne un réel positif, plus il est faible, plus le réseau est proche de la fonction théorique.

Exemple(s) :

$$loss(exp, obt) = |exp - obt|$$
$$loss(exp, obt) = (exp - obt)^2$$

Le probleme de regression se résume alors à minimiser la fonction de prete, c'est ici que la descente de gradient stochastique fait son entrée :

Descente de gradient : Pour minimiser loss, on aimerai bien descendre sa pente, c'est à dire dériver suivant les diférentes variables [1], en déduire l'orientation de la pente, et la descendre. Étant donné que de manière générale, il y a plusieurs variables, la dérivée se transforme en gradient. Des applications linéaires sont privilégiées dans les fonctions des neurones, afin de pouvoir calculer facilement les gradients.

Stochastique : On a donc expliqué d'ou vient la descente de gradient, mais que veut dire stochastique ? Ce mot est synonyme de hasard. En effet le temps de calcul du gradient est exponentiel vus que chaque neurones est relié a n neurones, eux même reliés à n autres neurones ect...
Alors lorsque le réseau est grand (un réseau peut facillement atteindre le million de neurones voir même des milliards [4]) il est inimaginable de calculer la totalité du gradient (les calculs peuvent parfois depasser plusieurs fois la durée de l'univers...), on utilise alors le gradient stochastique.

Cette descente de gradient est bien plus rapide, il est donc possible de l'iterer de nombreuses fois pour essayer de minimiser la fonction de perte. L'espaces des solutions étant rarement idéal (rarement une "cuvette" mais constitué de "bosses" et de "trous" chaotiques) il n'est pas obligatoire d'arriver à atteindre le minimum global mais au moins minimum local sera atteind. Avec plusieurs apprentissages on peut alors approximer très efficacement quasiment toutes les fonctions.

1. NB : Ici les variables sur lesquels on intervient sont les w_i (pas les x_i).

5.2 Intégrales de choquet

L'integrale de choquet est une intégrale découlant de la théorie de la mesure [5]. Ici on ne parlera que du modele discret.

Pour l'expliquer simplement, prenons un exemple : Suposons que l'on veuille conseiller des personnes sur l'achat d'ordinateurs. Ces personnes de conaissent absolument rien en informatique. La seul chose qu'ils veullent est une note, plus elle est élevée, plus l'ordinateur est performant. Essayons de faire un algorithme assez simple pour résoudre ce problemme : Chaque composant se voit atribué une note (en fonction de la puissance, la qualité de fabrication...), on appelle cette note *l'utilité* du composant. Une fois toutes les utilitées étudiées, on donne un poid à chaque famille de composants (RAM, processeur, carte mère...). Enfin on fait la somme de utilité fois poid pour tout les composant.
De ce fait, si un ordinateur a de meilleurs composant, plus de puissance, ect, sa note sera superieure.

Ce modèle parait raisonable dans la plupart des cas mais il est extraimement mauvais dans les cas extremes : Suposons qu'un constructeur peu scrupuleux propose un ordinateur assez étrange : Le processeur le moins cher du marché (assez mauvais) mais énormément de RAM, par exemple 128Go. Votre classement le placera forcément en haut de la liste, même si vous en conviendrez, cet ordinateur est assez inutile...

Il faudrait donc trouver un autre système modélisant les interactions entre les composants. Ce modèle est exactement celui sur lequel j'ai travaillé durant mon stage : *les intégrales de choquet*. En plus de donner un poid aux utilité, un poid est donné aux interactions des utilitées par le biais des fonctions min et max. Ces intéractions peuvent être faites deux à deux, trois à trois voir à plus. Mais pour des raison de temps de calcul, ces interactions se sont limités à l'interactions simple (deux à deux) durant mon stage. En effet, la taille du calcul évolue exponentiellement avec les interactions.

Voici donc la fonction que l'on vas vouloir aproximer :

$$C(X) = \sum_{i=1}^{n} w_i \times x_i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} \Big(w_{M\,ij} \times \max(x_i, x_j) + w_{m\,ij} \times \min(x_i, x_j) \Big) \quad (2)$$

Avec X le vecteur des utilitées et W, W_m et W_M les vecteurs des poids.

6 Matériel et méthodes

Ce stage s'est déroulé en deux principales phases : Une étape de developement durant laquelle des données et la fonction à apprendre ont été générées. Et une étape plus appliquée durant laquelle une base de donnée réelle à été étudiée.

Afin d'implémenter informatiquement les concepts théoriques vus précédement, la librairie Keras à été utilisée [6]. La partie sur les intégrales de choquet à été entierement recodée en python. L'ensembe du code est bien évidement disponible gratuitement et sous licence libre sur github [7].

7 Keras

La librairie Keras est une interface Python/TensorFlow [8] permetant de travailler avec des réseaux de neurones. Ici, on ne s'attardera que sur les fonctionnalitées principales, a savoir la création d'un réseau simple, la regression par SGD et l'évaluation des performances.

Voici un exemple de réseau de neurone assez simple qui vas essayer de deviner l'application linéaire suivante : $f(X) = 0.2x_1 + 0.8x_2$.

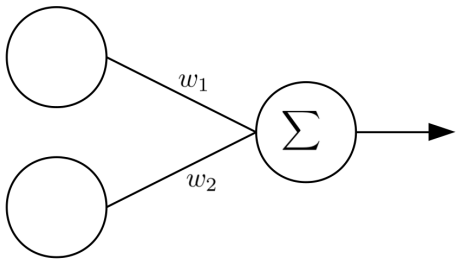

FIGURE 2 – Réseau simple

Pour créer ce réseau et lui faire apprendre la fonction precedement citée, le code suivant est nescessaire :

```
from keras import Sequential, optimizers, layers
import numpy as np
from random import random


def f(X):                       # la fonction que l'on cherche
    W = np.array([0.2, 0.8])
    return W @ X                # Produit scalaire

act = "linear"   # fonction d'activation
n_input = 2      # nombre de neurones
questions = np.array([np.array([random(), random()])
                      for i in range(10000)])
reponses  = np.array([f(q) for q in questions])

sgd = optimizers.SGD(lr=0.01, decay=1e-6,
        momentum=0.9, nesterov=True)
neurones = layers.Dense(1, activation=act,
        input_dim=n_input, use_bias=False)
model = Sequential()            # On cree un reseau
model.add(neurones)             # On lui ajoute des neurones
```

```
22 model.compile(optimizer=sgd,     # On compile le tout
23                 loss="mean_squared_error")
24 model.fit(questions, reponses)  # On essaye de coller aux donn es
25
26 for i, weight in enumerate(model.get_weights()[0]):
27     # on affiche les poids
28     print(f"w{i} : {weight[0]}")
```

code/reseau1.py

En executant ce code, on obtient :

```
1 w0 : 0.19988934695720673
2 w1 : 0.8001111745834351
```

On peut donc bien voir que le réseau de neurones fonctionne et réussit à apprendre des fonctions avec plusieurs paramètres. Il est cependant assez embetant de toujours devoir faire appel a toutes ces fonctions. Une librairie à alors été codée afin de simplifier son utilisation. Elle pourra être appelée avec de nombreux paramtres qui seront abordés dans les parties suivantes.

8 Implementation d'un réseau de choquet

Nous appellerons ici *réseau de choquet* un réseau de neurones ayant une architecture adaptée a la regression d'une intégrale de choquet. Comme vus precedement (5.2), une fonction de choquet a une architecture complexe. Voici l'intégrale de choquet entierement dévelopée pour un vecteur d'entrée taille 3 :

$$\begin{aligned} C \ &= w_1 \times x_1 + w_2 \times x_2 + w_3 \times x_3 \\ &+ w_{m1} \times \min(x_1, x_2) + w_{m2} \times \min(x_1, x_3) + w_{m3} \times \min(x_2, x_3) \\ &+ w_{M1} \times \max(x_1, x_2) + w_{M2} \times \max(x_1, x_3) + w_{M3} \times \max(x_2, x_3) \end{aligned}$$

Voici l'architecture d'un réseau de choquet entierement générée par un réseau de neurones :

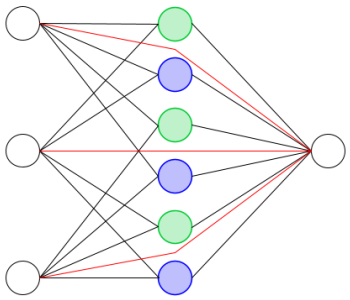

FIGURE 3 – Architecture d'un réseau de choquet

En bleu, des neurone appliquant $\min(X)$*, en vert* $\max(X)$*.*

On peut voir que trois problemes non triviaux se posent :
- Les neurones collorées n'appliquent pas une fonction simple.
- Les neurones ne sont pas reliés de manière simple (*cf.* 5.1).
- Certains neurones passent des informations en sautant une couche de neurones.

Une autre piste à alors été envisagée : Créer un réseau simple comme dans la figure suivante :

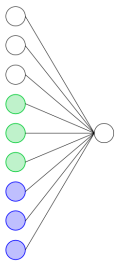

FIGURE 4 – Réseau alternatif

Ici, aucuns neurone n'a de fonction spécifique, ceux de gauche sont les entrées et celui de droite fait le produit scalaire avec un vecteur poid. Ce réseau est bien plus simple à générer : c'est un réseau *Dense* de taille 9 en entrée. Et plus généralement n^2 pour un vecteur de taille n.

Démonstation :

Prenons un vecteur de taille n, Le but est d'énumerer le nombre de neurones utiles a la formule (2). Le resultat est le suivant :

$$n + \binom{n}{2} + \binom{n}{2} = n + 2 \times \frac{n(n-1)}{2} = n^2 \tag{3}$$

Il faut alors créer une base de donnée d'aprentissage à partir de chaqu'unes des fonctions d'utilitées ce qui se fait tout simplement de la manière suivante :

```
1 def two_by_two(vector: iter, func: callable) -> np.array:
2     out = np.array([])
3     length = len(vector)
4     for i in range(length):
5         for j in range(i + 1, length):
6             out = np.append(out, func(vector[[i, j]]))
7     return out
8
9 Xs = np.concatenate((X, two_by_two(X, min), two_by_two(X, max)))
```

Lors de la descente de gradient, le réseau traite les poids indiférement, pour les récupérer, il sufit de prendre les n premiers pour W, les $\frac{n(n-1)}{2}$ suivant pour W_{min} et les $\frac{n(n-1)}{2}$ derniers pour W_{max}.

9 Données réelles

10 Résultats

On peut maintenant faire apprendre au réseau nimporte quelle application linéaire. Par exemple la moyenne :

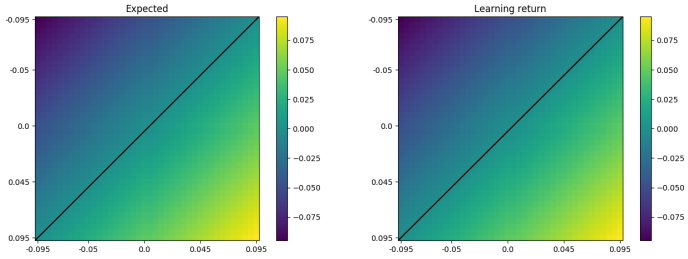

FIGURE 5 – Apprentissage de la moyenne

A gauche, le résutltat atendu, a droite, celui renvoyé.
x_1 en absice, x_2 en ordonées.

On peut voir que la descente de gradients se fait a merveille. Aucunes difference ne sont visibles : en effet les poids associés aux deux noeuds sont les suivants :

$$w_1 = 0.50000010 \qquad w_2 = 0.50000024$$

On peut voir qu'aux aproximations processeur, les poids sont les bons pour faire la moyenne de deux nombres :

$$m = \frac{n_1 + n_2}{2} = 0.5 \times n_1 + 0.5 \times n_2$$

11 Discution et conclusion

Références

[1] "Laboratoire de recherche en informatique." www.lri.fr.

[2] "House sales in king county, USA." kaggle.com/harlfoxem/housesalesprediction.

[3] "Science4all." www.youtube.com/playlist?list=PLtzmb84AoqRTl0m1b82gVLcGU38miqdrC.

[4] "Spectrum ieee." spectrum.ieee.org/tech-talk/computing/software/biggest-neural-network-ever-pushes-ai-deep-learning.

[5] A. Fallah Tehrani, C. Labreuche, and E. Hüllermeier, "Choquistic utilitaristic regression," 11 2014.

[6] "Keras : The python deep learning library." https://keras.io/.

[7] "Github." github.com/QuentinN42/Stage_LRI.

[8] "Tensorflow." tensorflow.org.