Université de Montréal

# Algorithmes d'apprentissage pour la recommandation

par
Valentin Bisson

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Mémoire présenté à la Faculté des études supérieures et postdoctorales
en vue de l'obtention du grade de Maître ès sciences (M.Sc.)
en informatique

Septembre, 2012

Université de Montréal
Faculté des études supérieures et postdoctorales

Ce mémoire intitulé:

**Algorithmes d'apprentissage pour la recommandation**

présenté par:

Valentin Bisson

a été évalué par un jury composé des personnes suivantes:

Pascal Vincent
président-rapporteur

Yoshua Bengio
directeur de recherche

Claude Frasson
membre du jury

## RÉSUMÉ

L'ère numérique dans laquelle nous sommes entrés apporte une quantité importante de nouveaux défis à relever dans une multitude de domaines. Le traitement automatique de l'abondante information à notre disposition est l'un de ces défis, et nous allons ici nous pencher sur des méthodes et techniques adaptées au filtrage et à la recommandation à l'utilisateur d'articles adaptés à ses goûts, dans le contexte particulier et sans précédent notable du jeu vidéo multi-joueurs en ligne. Notre objectif est de prédire l'appréciation des niveaux par les joueurs.

Au moyen d'algorithmes d'apprentissage machine modernes tels que les réseaux de neurones profonds avec pré-entrainement non-supervisé, que nous décrivons après une introduction aux concepts nécessaires à leur bonne compréhension, nous proposons deux architectures aux caractéristiques différentes bien que basées sur ce même concept d'apprentissage profond. La première est un réseau de neurones multi-couches pour lequel nous tentons d'expliquer les performances variables que nous rapportons sur les expériences menées pour diverses variations de profondeur, d'heuristique d'entraînement, et des méthodes de pré-entraînement non-supervisé simple, débruitant et contractant. Pour la seconde architecture, nous nous inspirons des modèles à énergie et proposons de même une explication des résultats obtenus, variables eux aussi. Enfin, nous décrivons une première tentative fructueuse d'amélioration de cette seconde architecture au moyen d'un *fine-tuning* supervisé succédant le pré-entrainement, puis une seconde tentative où ce *fine-tuning* est fait au moyen d'un critère d'entraînement semi-supervisé multi-tâches.

Nos expériences montrent des performances prometteuses, notament avec l'architecture inspirée des modèles à énergie, justifiant du moins l'utilisation d'algorithmes d'apprentissage profonds pour résoudre le problème de la recommandation.



**Mots clés: intelligence artificielle, apprentissage machine, réseaux de neurones, autoencodeurs contractants, jeu vidéo, architectures profondes, modèles à énergie**

## ABSTRACT

The age of information in which we have entered brings with it a whole new set of challenges to take up in many different fields. Making computers process this profuse information is one such challenge, and this thesis focuses on techniques adapted for automatically filtering and recommending to users items that will fit their tastes, in the somehow original context of an online multi-player game. Our objective is to predict players' ratings of the game's levels.

We first introduce machine learning concepts necessary to understand the two architectures we then describe ; both of which taking advantage of deep learning and unsupervised pre-training concepts to solve the recommendation problem. The first architecture is a multilayered neural network for which we try to explain different performances we get for different settings of depth, training heuristics and unsupervised pre-training methods, namely, straight, denoising and contrative auto-encoders. The second architecture we explore takes its roots in energy-based models. We give possible explanations for the various results it yields depending on the configurations we experimented with. Finally, we describe two successful improvements on this second architecture. The former is a supervised fine-tuning taking place after the unsupervised pre-training, and the latter is a tentative improvement of the fine-tuning phase by using a multi-tasking training criterion.

Our experiments show promising results, especially with the architecture inspired from energy-based models, justifying the use of deep learning algorithms to solve the recommendation problem.



**Keywords : artificial intelligence, machine learning, neural networks, contractive autoencoders, video game, deep learning, energy-based models**

## TABLE DES MATIÈRES

## LISTE DES FIGURES

## LISTE DES TABLEAUX

## LISTE DES SIGLES

| | |
|---|---|
| NLL | Negative Log Likelihood (log-vraisemblance négative) |
| MSE | Mean Square Error (Erreur quadratique) |
| MLP | Multi Layer Perceptron (perceptron multi-couches) |
| RBM | Restricted Boltzmann Machine (machine de Boltzmann restreinte) |
| SVD | Singular Value Decomposition (décomposition en valeurs singulières) |
| AE | AutoEncoder (auto-encodeur) |
| DAE | Denoising AutoEncoder (auto-encodeur débruitant) |
| CAE | Contractive AutoEncoder (auto-encodeur contractant) |
| SDN | Standard Deep Network (réseau profond standard) |
| EBM | Energy-Based Model (modèle basé sur l'énergie) |
| EDN | Energy-based Deep Network (réseau profond basé sur l'énergie) |
| UGC | User Generated Content (contenu généré par les utilisateurs) |

# NOTATION

| | |
|---|---|
| $\mathscr{D}$ | ensemble de données, contenant les exemples d'entraînement, et possiblement des cibles |
| $\mathscr{D}_{train}$ | sous-ensemble de $\mathscr{D}$ utilisé pour l'optimisation des paramètres $\theta$ |
| $\mathscr{D}_{valid}$ | sous-ensemble de $\mathscr{D}$ distinct de $\mathscr{D}_{train}$ , utilisé pour l'optimisation des hyper-paramètres |
| $\mathscr{D}_{test}$ | sous-ensemble de $\mathscr{D}$ distinct $\mathscr{D}_{train}$ et de $\mathscr{D}_{valid}$ , utilisé pour estimer l'erreur de généralisation |
| i.i.d. | idépendants et identiquement distribués |
| $\theta$ | paramètres optimisés lors de l'entraînement |
| $\widehat{R}$ | risque empirique |
| L | fonction de perte |
| $lambda$ | taux d'apprentissage |
| $\mathscr{B}$ | lot d'exemples de $\mathscr{D}$ |
| $\frac{\partial f}{\partial x}$ | dérivée partielle de la fonction $f$ par rapport à $x$ |
| $C$ | ensemble des utilisateurs, dans $\mathscr{D}$ |
| $S$ | ensemble des articles, dans $\mathscr{D}$ |
| $U$ | ensemble des notes des utilisateurs pour les articles de $\mathscr{D}$ |
| $n_{epochs}$ | durée d'entraînement d'un modèle, en époques |
| $\tau$ | numéro d'itération, $\in [\tau_0, n_{epochs}]$ |
| $top_k$ | métrique mesurant la performance d'un algorithme de recommandation prédisant $k$ articles à chaque utilisateur |
| $top_k^*$ | valeur maximale de la métrique $top_k$ pour un ensemble de données donné |
| $tanh$ | tangente hyperbolique |

Je tiens avant-tout à remercier Édouard Auvinet, sans qui la rédaction de ce mémoire n'aurait simplement pas eu lieu.

Un grand merci à Yoshua Bengio et Olivier Delalleau pour la patience dont ils ont su faire preuve à mon égard durant notre collaboration, ainsi que pour leurs nombreux conseils. Merci à Frederic Bastien pour son support technique constant, et à Steven Pigeon pour la relecture minutieuse.

Merci aux membres du LISA pour leur dévouement à développer *Theano* et *Pylearn*, les principaux outils ayant permis d'effectuer ce travail dans le temps imparti.

Merci à Ubisoft pour avoir offert un cadre de travail ouvert à la recherche, ainsi que pour avoir autorisé l'accès à l'ensemble de données du jeu Mighty Quest for Epic Loot.

*Last but not least*, un grand merci à mes amis et parents pour m'avoir supporté durant ces deux années tumultueuses.

# INTRODUCTION

## Contexte

La quantité d'information à disposition de chacun est en constante croissance, en majeure partie grâce à Internet. Cette quantité, presque infinie dans certains domaines, a créé plusieurs nouveaux besoins. Le principal d'entre eux a longtemps été de pouvoir accéder à l'information ; c'est le rôle des moteurs de recherches, qui proposent en général une liste de résultats en réponse à la requête d'un utilisateur, sans égard au profil de celui qui l'interroge.

Un autre besoin, étudié depuis au moins 20 ans [30] est celui de pouvoir faire un choix parmi une information pertinente non seulement objectivement mais aussi subjectivement, c'est-à-dire adaptée au profil de la personne effectuant la recherche. C'est le principe de **la recommandation** et des **systèmes de recommandation**. La recommandation est différente de la recherche pure, dans le sens où elle ne nécessite pas toujours une interaction directe de la part de l'utilisateur (la requête). Certaines fois le choix est tellement large que l'utilisateur ne sait pas par où commencer. Il faut alors **selectionner pour lui** des *articles* qui ont le plus de chances de satisfaire ses préférences personnelles, bien que ces dernières ne soient pas forcément connues explicitement. Le terme *article* désignera au long de ce document toute entité faisant l'objet d'une recommandation. Cette notion d'adaptation de la recherche aux goût de l'utilisateur n'est cependant pas toujours ignorée des moteurs de recherche modernes, et l'on voit dans certains cas un genre de fusion s'opérer entre les deux types de systèmes.

## Objectifs

C'est ce problème de selection, de **tri** en vue de ne proposer à l'utilisateur que l'information la plus susceptible de l'intéresser, que les travaux décrits dans ce mémoire tentent de résoudre au moyen de l'apprentissage machine.

Plus précisément, le travail de recherche effectué dans ce mémoire consiste à s'inspirer de la littérature pour implémenter, concevoir et comparer des modèles de recommandation à l'utilisateur, ainsi que de proposer des explications de leurs performances en termes de recommandation.

Nos modèles (ou le meilleur d'entre-eux) trouvent une application dans le domaine du jeu vidéo en recommandant aux joueurs quels niveaux jouer. Ceci est fait au moyen d'un *pipeline* de prédiction que nous avons développé en parallèle à nos recherches et intégré à

un jeu en cours de développement, dans lequel un modèle rafraîchit en permanence et pour chaque joueur une liste de niveaux qui lui sont proposés lorsqu'il se connecte au jeu.

**Plan**

Le chapitre 1 qui suit a pour but de présenter une introduction au domaine de l'apprentissage automatique en expliquant certains des principaux concepts clés qui le composent, en particulier ceux nécessaires à la compréhension de ce mémoire.

Le second chapitre introduit les primitives haut-niveau d'apprentissage automatique utilisées comme composantes des architectures étudiées dans ce mémoire.

Le troisième chapitre présente le domaine d'application dans lequel nous appliquons nos modèles, à savoir la recommendation à l'utilisateur dans le jeu-vidéo multi-joueurs en ligne. Il introduit aussi le lecteur aux différentes tâches couramment rencontrées dans ce domaine, ainsi que les métriques utilisées.

Un quatrième chapitre présente les différents modèles développés dans ce mémoire et le protocole expérimental employé. De plus il fournit une description de l'ensemble de données principal sur lequel les modèles sont évalués.

Le chapitre cinq présente les performances des différents modèles étudiés en fonction de leurs hyper-paramètres, et énonce certaines des raisons possibles des comportements observés.

Enfin, nous concluons ce mémoire en offrant une rétrospective sur la recherche effectuée ainsi que les résultats obtenus, et proposons une perspective constructive sur les prochaines pistes à explorer.

# CHAPITRE 1

# INTRODUCTION À L'APPRENTISSAGE AUTOMATIQUE

## 1.1 Qu'est-ce que l'apprentissage automatique

L'apprentissage automatique (*machine learning*) est un sous domaine de l'intelligence artificielle qui se concentre sur l'élaboration de modèles capables de représenter certaines caractéristiques du monde qui nous entoure, d'apprendre certaines propriétés **statistiques** des distributions des données qu'ils traitent, afin d'accomplir diverses tâches. Le rapport à l'intelligence vient de la capacité de ces modèles à **généraliser**, c'est-à-dire à extraire l'information pertinente de données étudiées au fil d'un processus de mise(s)-à-jour appelé **entraînement**, et de savoir la réutiliser avec efficacité sur de nouvelles données jamais rencontrées auparavant.

## 1.2 Modèles et types d'apprentissage

Un modèle est une **fonction de décision** $f$ qui, dans le cas de **l'apprentissage supervisé**, prend en entrée un exemple $x \in \mathbb{R}^d$ issu d'un ensemble $\mathscr{D}^{n\times d} = \{z_i\}_{i=1}^n = \{(x_i, y_i)\}_{i=1}^n$ où $y_i$ est la valeur cible (ou *étiquette*) associée au $x_i$, et renvoie une prédiction $f(x_i) = \hat{y}_i$. Dans la suite de ce document, on considèrera que les exemples $z_i \in \mathscr{D}$ sont indépendants et identiquement distribués (i.i.d.). Lorsque la cible est discrète, on parle d'une tâche de **classification**. Par exemple $f$ pourrait prédire une nationalité ou une espèce animale. On parle de **régression** lorsque la cible est continue, comme pour la prédiction d'une distance ou d'une position.

En **apprentissage non-supervisé**, il n'existe pas de cibles explicites, c'est-à-dire que $\mathscr{D} = \{z_i = x_i, i = 1, ..., n\}$. C'est le cas de l'estimation de densité qui consiste à évaluer avec quelle probabilité un $x_i$ appartient à la distribution de $\mathscr{D}$ . C'est aussi le cas du partitionnement (*clustering*) qui consiste à regrouper les $x_i$ de façon à obtenir des groupes homogènes.

## 1.3 Risque empirique et fonction de perte

On estime la performance de $f$ à l'aide d'une fonction de perte (ou fonction de coût) $L(f, z)$ et d'un ensemble de données $\mathscr{D}$. C'est le risque empirique $\widehat{R}$ , ou perte espérée sur

$\mathscr{D} \in \mathbb{R}^{n \times d}$ :

$$\widehat{R}(f, \mathscr{D}) = \underset{\mathscr{D}}{E}[L(f, z)], z \in \mathscr{D} = \frac{1}{n} \sum_{i=1}^{N} L(f, z_i) \ .$$

La nature de $L$ dépend beaucoup de la tâche considérée, mais voici quelques exemples :

- En **régression**, on veut mesurer à quel point $f(x) = \hat{y}$ est éloignée de sa cible $y$ dans l'espace des prédictions, on peut donc choisir l'erreur quadratique. Notons que cette erreur, moyennée sur l'ensemble considéré, sera par la suite appelée MSE pour *Mean Square Error* :

$$L(f, z) = (f(x) - y)^2 \ . \tag{1.1}$$

- En **classification**, on veut mesurer la proportion d'exemples auxquels le modèle attribue la bonne étiquette (la bonne classe), on peut donc utiliser un indicateur d'erreur de classification : $L(f, z) = \mathbb{1}_{(f(x) \neq y)}$.
- En **estimation de densité**, il s'agit de mesurer la vraisemblance $p(x)$ que $f$ accorde aux exemples de $\mathscr{D}$ (étant tirés de l'ensemble de données, on est sûr qu'elle devrait être élevée), et pour rester dans le contexte d'un critère d'optimisation à minimiser, la log-vraisemblance négative (*negative log-likelihood*) est un choix standard :

$$L(f, z) = -log(f(x)) \ . \tag{1.2}$$

Du fait que l'ensemble de données utilisé pour s'approcher du risque est fini, le risque empirique est un **estimateur bruité**. Pour estimer le risque réel, il faudrait utiliser un ensemble d'entraînement contenant une infinité de points afin de modéliser parfaitement la distribution de $\mathscr{D}$ , ce qui n'est pas possible concrètement. De plus, en minimisant le risque *empirique*, on biaise le modèle à performer mieux sur le sous-ensemble de points d'entraînement utilisé que sur le reste de l'ensemble. Toute mesure de performance effectuée sur cet ensemble d'entraînement est donc le plus souvent optimiste.

Il est à noter que l'ensemble de données est divisé en 3 sous-ensembles : d'**entraînement** ($\mathscr{D}_{train}$ ), de **validation** ($\mathscr{D}_{valid}$ ) et de **test** ($\mathscr{D}_{test}$ ), les deux derniers étant généralement de tailles inférieures au premier. C'est par cette division que l'on peut s'assurer de la capacité à généraliser des modèles : on entraîne $f$ sur $\mathscr{D}_{train}$ jusqu'à ce que son erreur sur $\mathscr{D}_{valid}$ ne s'améliore plus, et on considère sa performance sur $\mathscr{D}_{test}$ . Sans cette division, un modèle pourrait sembler très performant sur $\mathscr{D}$ en stockant simplement les exemples et leurs cibles, et ainsi obtenir un score parfait tout en étant incapable de conserver une telle performance sur de nouvelles données.

En apprentissage automatique, être performant consiste notamment à ne pas faire d'erreur (ou faire en sorte que l'erreur soit d'un ordre de grandeur acceptable). Pour celà, on cherche à trouver parmi un ensemble de fonctions possibles $\mathscr{F}$ la fonction $\hat{f}^*$ qui **minimise le risque empirique** $\widehat{R}$ (ainsi qu'un terme de régularistaion $\Omega(f)$) sur l'ensemble d'entraînement $\mathscr{D}_{train}$ :

$$\begin{aligned} \hat{f}^* &= \underset{f \in \mathscr{F}}{\operatorname{argmin}} J(f) \\ J(f) &= \widehat{R}(f, \mathscr{D}_{train}) + \Omega(f) \end{aligned} \tag{1.3}$$

## 1.4 Modèles paramétriques et non-paramétriques

Certains modèles, dits **paramétriques**, se basent sur des paramètres $\theta$ (scalaires, vecteurs, matrices) pour caractériser le choix de $f$ dans $\mathscr{F}$. L'apprentissage consiste alors à appliquer un algorithme d'optimisation sur la fonction $f_\theta$ paramétrisée par la configuration initiale $\theta_0$ des paramètres, afin que ces paramètres minimisent $J(f_{\theta_t})$ pour $t < a$. La descente de gradient (1.7) et la divergence contrastive [10, 19] en sont des exemples.

D'autres modèles, dits **non-paramétriques**, ont une capacité qui augmente proportionnellement à la taille de leur ensemble d'entraînement car ils l'utilisent directement pour modéliser la distribution de $\mathscr{D}$ . Pour ce faire, ils peuvent par exemple tirer parti de l'information présente localement à un point de test donné, ou à un ensemble de points déterminé par une heuristique (se basant par exemple sur la similarité entre ces points).

En présupposant *a priori* une régularité locale (des points proches dans l'espace des entrées ont aussi des prédictions proches), la procédure d'inférence de ces modèles pour un point nécessite la présence d'autres points dans son voisinage, ce qui pose problème quand le nombre de dimension augmente, c'est le **fléau de la dimensionalité**. Ce problème vient du fait que lorsque la dimensionalité $d$ des points de $\mathscr{D}$ augmente, le volume de $\mathbb{R}^d$ augmente **exponentiellement**, et le nombre de points nécessaires pour couvrir un certain volume croit aussi exponentiellement. Les modèles se basant seulement sur l'apriori de régularité locale souffrent donc d'une décroissante densité d'exemples à disposition pour faire leur prédictions : chaque point semble plus isolé des autres à mesure que l'on considère un plus grand nombre de dimensions.

L'algorithme des ***k* plus proches voisins** (*k nearest neighbors*) est un exemple de modèle non-paramétrique des plus connus. C'est un algorithme de classification qui calcule la distance du point de test $x_{test}$ à chacun des $x_i$ de $\mathscr{D}_{train}$ et estime que sa classe est celle qui domine parmi les *k* points les plus proches de $x_{test}$. Ce modèle a pour simple entraînement

de stocker $\mathscr{D}_{train}$ . La fonction de distance choisie et $k$ sont des hyper-paramètres.

Les nombre et dimensions des paramètres $\theta$ déterminent la taille de $\mathscr{F}$ que l'on appelle la **capacité** du modèle, ce sont des **hyper-paramètres**, c'est-à-dire des paramètres choisis arbitrairement *avant* l'entraînement et donc non-optimisés lors de celui-ci. On parlera d'hyper-optimisation pour désigner le procédé de selection des hyper-paramètres (davantage de détails sont donnés en 4.4.1). D'autres hyper-paramètres peuvent par exemple déterminer certaines fonctions utilisées au sein de $f$, le critère d'arrêt de l'entraînement ou sa durée maximale, etc. Pour trouver des hyper-paramètres optimaux, on effectue généralement une recherche sur un sous-ensemble des combinaisons d'hyper-paramètres du modèle, c'est l'*hyper-optimisation*, expliquée plus en détail en 4.4.1.

L'**initialisation** des paramètres du modèle correspond à une position de départ $\theta_0$ associée à la fonction initiale $\hat{f}_0 \in \mathscr{F}$, et détermine la suite des futures positions que l'algorithme d'entraînement pourra faire emprunter au modèle. C'est donc une étape importante avant l'entraînement proprement dit, mais tout comme il est généralement impossible de savoir si la configuration des paramètres après entraînement est sur un minimum global de $\widehat{R}$ , il est impossible de savoir exactement quelle est leur meilleure configuration de départ, mais on peut faire des comparaisons empiriques. Pour les modèles dont on sait que les poids doivent avoir des valeurs différentes les unes des autres, une variété d'heuristiques existe. La première consiste à initialiser les paramètres selon une loi **aléatoire** uniforme dans un intervalle souvent petit et possiblement déterminé par la structure interne des paramètres du modèle. La valeur de cet intervalle peut elle-même être déterminée selon différentes heuristiques. On peut aussi utiliser un algorithme d'entraînement non-spécialisé dans une tâche particulière (non-supervisé), mais servant «juste» à s'assurer que les paramètres sont dans une configuration en rapport avec l'ensemble de données considéré, c'est le pré-entraînement, expliqué plus en profondeur en 2.3.2.

## 1.5 Capacité

Plus l'ensemble $\mathscr{F}$ est grand, plus la fonction de décision $\hat{f}^*$ a de souplesse dans l'espace de ses paramètres et plus l'algorithme d'optimisation a de choix pour rapprocher $\hat{f}^*$ de la «vraie» meilleure fonction possible $f^*$ : une fonction qui minimiserait le risque espéré réel. En contrepartie, si une très grande capacité est mise à disposition de l'algorithme d'apprentissage, il peut avoir tendance à modéliser la distribution de $\mathscr{D}_{train}$ en particulier plutôt que celle de $\mathscr{D}$ , et renvoyer en conséquence une fonction qui donne une erreur empirique quasi-nulle

sur $\mathscr{D}_{train}$ mais très élevée sur $\mathscr{D}_{valid}$ et $\mathscr{D}_{test}$ . Ce phénomène s'appelle le **sur-apprentissage**. De plus, permettre à $f$ de modéliser une distribution avec davantage de précision la rend plus sensible aux exemples utilisés lors de l'entraînement, et donc rend $\widehat{R}(f, \mathscr{D}_{train})$ plus optimiste. En revanche, si on limite trop fortement la capacité de $f$, bien qu'elle ne soit plus en mesure de sur-apprendre son ensemble d'entraînement, elle ne sera pas non-plus capable de modéliser suffisament bien sa distribution (on peut parler de sous-apprentissage) : une faible capacité implique un fort biais. Il y a donc un compromis à trouver entre la capacité et la généralisation, c'est le **compromis biais-variance**.

### 1.6 Méthodes de régularisation

Une méthode pour limiter artificiellement la capacité de $f$ et ainsi son risque de sur-apprendre $\mathscr{D}_{train}$ est **la régularisation**. Elle consiste à ajouter à la fonction de coût utilisée lors de l'entraînement un terme qui **pénalise des paramètres** du modèle sous certaines conditions.

En pénalisant par exemple des poids de très grande amplitude, on peut limiter le nombre de combinaisons de poids possibles, donc la capacité, sans modifier le nombre de poids directement. Bien qu'elle implique une préconception sur les valeurs des poids, elle est justifiée par le principe du *rasoir d'Ockham* qui dit que parmi deux hypothèses valides, la plus simple est probablement la meilleure. Dans notre contexte, des poids plus faibles en amplitude seront vus comme une solution plus simple, parce que l'ensemble des solutions dans la boule $||\theta|| < a$ est plus petit que celui dans la boule $||w|| < b$. Dans le cas de la régularisation linéaire (dite «$L_1$»), le terme ajouté au coût est proportionnel à la somme des valeurs absolues des poids. Sa linéarité «pousse» les poids près de zéro, favorisant la sparsité (le fait que beaucoup de valeurs soient nulles). La régularisation quadratique, dite «$L_2$» (*weight decay*), pénalise plus fortement des poids forts que des poids faibles, et laisse aussi plus de liberté aux petits paramètres dans cet interval.

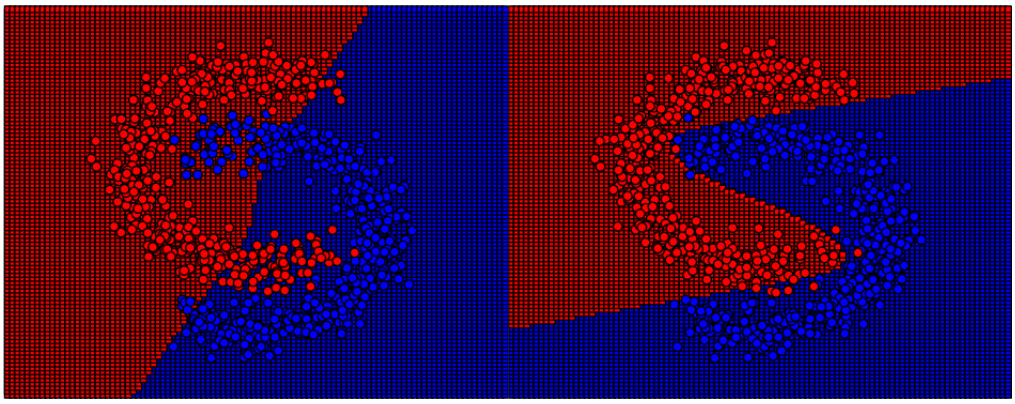
- $L_1(\theta) = \alpha \sum_i |\theta_i|$ ,
- $L_2(\theta) = \alpha \sum_i \theta_i^2$ .

avec $\alpha \in \mathbb{R}^+$ un hyper-paramètre.

Le **nombre d'itérations** de l'algorithme d'entraînement (qui raffine les valeurs des paramètres pas-à-pas), affecte indirectement la capacité du modèle en limitant le nombre de mise à jour des poids effectuées, et donc de la «distance» maximale possible entre la configuration initiale des paramètres $\theta_0$, et $\theta^*$ obtenue en fin de l'entraînement. On déterminera cet

hyper-paramètre au moyen de l'**arrêt prématuré** (*early stopping*), qui consiste à stopper l'entraînement lorsque $\widehat{R}_{valid}$ a cessé de diminuer depuis un nombre donné d'époques (une **époque** correspond à l'utilisation au complet une fois de $\mathscr{D}_{train}$ par l'algorithme d'entraînement). L'effet de la durée d'entraînement est illustré dans la figure 1.1, et les courbes d'erreurs d'entraînement et de test dans la figure 1.2.

Figure 1.1 – Sur-apprentissage et sous-apprentissage



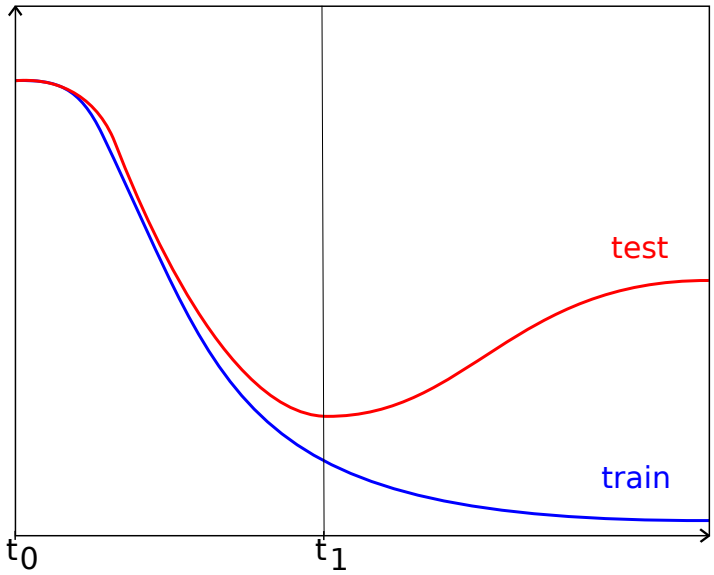
Graphe de la frontière de décision d'un réseau de neurones 2.2.1 sur l'ensemble de données jouet des deux lunes présentant deux classes (rouge et bleue). Le réseau analysé dans les deux graphes a la même architecture. À gauche, l'algorithme d'entraînement a fait 50 époques. À droite, 1000. Les disques représentent les points de l'ensemble d'entraînement. La grille en fond représente une série d'évaluations du modèle. On peut imaginer l'entraînement comme la torsion étape-par-étape de la frontière de décision. L'arrêt prématuré de l'entraînement déterminera alors à quel point on veut laisser le modèle s'adapter aux données.

Une méthode à la justification assez pratique mais communément utilisée pour la régularisation de modèles paramétriques est la réutilisation de paramètres à des fins liées mais différentes. C'est le cas par exemple des **poids liés** (*tied weights*) dans les réseaux de neurones (2.2.1) lors du pré-apprentissage non-supervisé (2.3.2). En forçant le modèle à utiliser les mêmes poids pour deux tâches différentes, on impose que ces poids restent dans un espace de valeurs «compatible» aux *deux* tâches.

## 1.7 Algorithmes d'entraînement

Un algorithme majeur pour l'entraînement de modèles linéaires et non-linéaires est l'**algorithme du gradient**, ou descente de gradient (*gradient descent*), une méthode d'optimisation supervisée, itérative et de premier ordre (c'est-à-dire n'utilisant pas de dérivée du second degré). Il s'agit d'ajuster les paramètres $\theta_i$ du modèle dans le sens opposé au gradient de la fonc-

Figure 1.2 – Courbes d'erreurs typiques




Un modèle est entraîné sur $\mathscr{D}_{train}$ et l'on mesure son erreur sur $\mathscr{D}_{train}$ (en bleu) et $\mathscr{D}_{test}$ (en rouge). Dans $[t_0, t_1]$, l'erreur diminue sur $\mathscr{D}_{train}$ et $\mathscr{D}_{test}$ , on peut donc penser que le modèle apprend des informations utiles en général. Cette phase se termine quand la variation de l'erreur sur $\mathscr{D}_{test}$ change de signe (dans la pratique, ce signe peut osciller pendant quelques époques ; la pente peut aussi passer par des plateaux). Dans $[t_1, t_\infty]$, le modèle continue à apprendre sur $\mathscr{D}_{train}$ , mais cet apprentissage n'est plus bénéfique sur les deux ensembles de données : le modèle apprend des **particularités** de $\mathscr{D}_{train}$ . C'est là qu'on peut parler de sur-apprentissage, et qu'on voudra stopper l'entraînement.

tion de perte par rapport à $\theta$ de façon à en trouver un minimum. Cet algorithme est basé sur l'observation que pour une fonction de perte $L(f_\theta, z)$, on a :

$$L(f_\theta, z) \geqslant L(f_{\theta - \lambda \nabla L(f_\theta, z)}, z)$$

où $\lambda$, appelé **taux d'apprentissage** (*learning rate*), est un hyper-paramètre infinitésimal, et

$$\nabla L(f_\theta, z) = \frac{\partial L(z)}{\partial \theta}$$

Cet algorithme exige que la fonction de perte soit différentiable. Aussi, dans le cas de fonctions de pertes non-différentiables comme celle mentionnée pour la classification (en 1.3), on choisira un critère d'optimisation différent. Précisons que la mise à jour de $\theta$ se fait paramètre par paramètre :

$$\theta_{i_{t+1}} = \theta_{i_t} - \lambda \frac{\partial L(\theta_{it})}{\partial \theta_i}$$

.

Si cette mise à jour est faite pour chaque $x_i$ de $\mathscr{D}_{train}$ , on parlera de **descente de gradient stochastique** : les points d'entraînement étant considérés indépendants et identiquement distribués, entraîner le modèle sur les points successifs de $\mathscr{D}_{train}$ est la même chose qu'une approximation stochastique où l'on tire ces points au hasard. Si l'on effectue la mise à jour des poids selon la moyenne des gradients de $f$ sur tout $\mathscr{D}_{train}$ , on parlera de **descente de gradient par lot** (*batch gradient descent*). C'est un compromis entre ces deux méthodes qui est employé au long des expériences de ce mémoire, puisque les mises à jour des poids sont calculées à partir de «mini-lots» (*mini-batches*) où l'on a :

$$\theta_{t+1} = \theta_t - \lambda \nabla \widehat{R}(f_\theta, \mathscr{B}),$$
$$\nabla \widehat{R}(f_\theta, \mathscr{B}) = \frac{1}{m} \sum_{z_i \in \mathscr{B}} \frac{\partial L(f_\theta, z_i)}{\partial \theta}$$

pour le lot $\mathscr{B}_k = \{z_i | i = j, ..., min(j+m, n)\}$ et $j = k \times m, \forall k \leqslant \left\lfloor \frac{n}{m} \right\rfloor$ , $m \leqslant n$. On peut voir cette méthode comme une médiation des deux précédentes, où la descente de gradient stochastique utilise un $\mathscr{B}$ de taille 1 et la descente par lot un $\mathscr{B}$ de taille $n$. L'avantage d'utiliser des mini-lots par rapport à la descente stochastique, autre que celui computationnel, est que le gradient étant moyenné il est moins bruité qu'avec un exemple unique. L'avantage par rapport à la descente avec un unique lot est la fréquence plus élevée de mises à jour moins biaisées.

On appelle **itération** l'application de l'algorithme sur une unité d'entraînement (un $x_i$ de $\mathscr{D}_{train}$ , ou un $\mathscr{B}_k$), et **époque** son application sur $\mathscr{D}_{train}$ . L'entraînement dure donc un nombre d'époques $n_{epochs}$ que l'on peut borner (hyper-paramètre) afin de régulariser le modèle, et de respecter des contraintes computationnelles : d'une part l'entraînement pouvant converger très longtemps avec des améliorations négligeables, et d'autre part, les mini-lots permettent de profiter de caractéristiques matérielles permettant de paralléliser l'exécution du calcul.

Le taux d'apprentissage $\lambda$ est un hyper-paramètre déterminant dans l'entraînement du modèle. Trop grand, les pas de gradient risquent de passer un éventuel *minimum* intéressant et augmenter l'erreur au lieu de la diminuer. Trop petit, la convergence des poids va prendre très longtemps. On cherchera donc à diminuer $\lambda$ au fur de l'entraînement. Aussi, l'algorithme de descente de gradient ne garantit la convergence vers le minimum global de $L$ que lorsque celle-ci est convexe, une condition insatisfaite dans l'intégralité des modèles étudiés dans ce document. Il existe une variété d'heuristiques pour maximiser les chances que le minimum

local vers lequel l'algorithme va converger soit proche du minimum global de $L$. Deux d'entre elles ont été considérées dans les expériences menées ici :

- l'heuristique de **recuit** (*annealing*), qui correspond à une décroissance de $\lambda$ proportionnelle à l'inverse du nombre d'itérations $\tau_t$, à partir d'une itération $\tau_0$ donnée :

$$\lambda_t = \min(\lambda_0, \frac{\tau_0}{1+\tau_t}).$$

- L'heuristique *ADAGRAD* [15], qui calcule un taux d'apprentissage scalaire $\lambda_{t,i}$ à l'itération $t$ pour le paramètre $\theta_i$ en divisant un taux d'apprentissage de départ par la somme, depuis $t_0$, de la norme 2 du gradient de la perte $L(f_{\theta i}, z_j)$ par $\theta_i$ :

$$\lambda_{t,i} = \frac{\lambda_0}{\sqrt{\alpha_t}},$$
$$\begin{cases} \alpha_{-1} & = 1, \\ \alpha_t & = \alpha_{t-1} + \sum_{j=t_0}^{t} \left( \frac{\partial L(f_\theta, z_j)}{\partial \theta_i} \right)^2 . \end{cases}$$

Dans les deux cas, $\lambda_0 \in \mathbb{R}$ correspond à un taux d'apprentissage de départ. Il s'agit ici d'une version simplifiée de l'algorithme, qui sera utilisée dans ce mémoire. L'idée générale derrière cette heuristique est d'attribuer aux paramètres responsables plus souvent de la perte un taux d'apprentissage plus bas, afin que celui de ceux qui en sont responsables *moins souvent* soit plus grand en comparaison. L'effet escompté est de cibler l'apprentissage sur les features les plus rarement observées, et ainsi faire en sorte que le modèle «remarque» davantage ces erreurs, et que les paramètres qui y sont liés soient optimisés plus rapidement.

Malgré ces méthodes, il n'existe pas de garantie que le minimum global soit atteint. Ça n'est cependant pas nécessairement grave car c'est la généralisation qui nous intéresse avant tout. De plus le critère optimisé n'est souvent qu'un indicateur mandataire (un *proxy*) de l'efficacité du modèle pour la tâche considérée. Par exemple, dans ce mémoire on entraînera des modèles de régression pour une tâche finale de tri, alors que le tri ne prête pas égard aux valeurs en soit, mais juste à leur ordre (leur rang). Dans ces conditions, l'exactitude même de la prédiction pour un $x_i$ importe moins que sa valeur comparative à d'autres $x_j$.

Il est à noter que même si la répartition des points dans les trois ensembles d'entraînement, de validation et de test est différente d'un entraînement à l'autre, ceux-ci sont toujours considérés i.i.d.. Cependant, leur différences relatives les uns aux autres fait que l'ordre dans

lequel ils sont utilisés lors de l'entraînement peut influencer la vitesse de convergence, et donc le minimum local dans lequel il se stabilise. C'est du moins l'hypothèse de l'**apprentissage par curriculum** [4] (*curriculum learning*), qui s'inspire de comparaisons cognitives et sociales dans différents contextes (dressage animal, éducation de l'enfant, etc) pour montrer, sur des tâches d'apprentissage machine, qu'augmenter graduellement $\mathscr{D}_{train}$ avec des exemples d'entraînement de *difficulté* croissante ou porter une emphase croissante au long de l'entraînement, sur les exemples les plus difficiles, améliore significativement la généralisation. Cette notion de difficulté peut prendre plusieurs formes, et donc être estimée de plusieurs façons : entropie des distributions de $\mathscr{D}_{train}$ , distance de $x_i$ à la moyenne, etc.

### 1.8 Dérivation des fonctions composées

La règle de dérivation en chaîne (*chain-rule*) est un outil mathématique utile pour effectuer correctement la rétro-propagation du gradient sur des paramètres de $f$ loins du résultat $L(f(x), y)$ dans le graphe de flot de $L$. On appelle *parent* du nœud $L$ tout nœud $v_j$ faisant partie des termes de l'expression de $L$. La règle explique comment calculer la dérivée partielle de $L$ selon un $v_j$ en fonction des $v_i$ dont $v_j$ est un parent. C'est la somme des produits de toutes les dérivées partielles rencontrées sur les chemins entre $L$ et $v_j$. Plus formellement, pour la sortie du graphe $L$ dont un ou plusieurs nœuds $v_i$ sont parents, eux mêmes calculés à partir d'un nœud parent $u$ :

$$\frac{\partial L}{\partial u} = \sum_i \frac{\partial L}{\partial v_i} \frac{\partial v_i}{\partial u}$$

On applique la règle en commençant par le nœud de sortie $L$ (qui vaut $\frac{\partial L}{\partial L} = 1$), et on «remonte» ainsi le graphe de flot jusqu'au(x) nœud(s) d'entrée. Un exemple d'application sera donné pour le réseau de neurones en 2.2.2.

### 1.9 La normalisation de l'entrée

On considèrera dans ce mémoire que les entrées des différents modèles sont le résultat d'une **normalisation**, c'est-à-dire que les valeurs $x_{i,j}$ originelles sont recalibrées pour avoir certaines propriétés bénéfiques à l'entraînement, comme l'indépendance aux unité et intervalle originaux, avoir les mêmes moyenne et dispersion, etc. L'ensemble normalisé $\mathscr{D}$ ainsi obtenu est une abstraction de $\mathscr{D}_{orig}$ plus facile à comparer avec celle d'un autre ensemble de données que les ensembles de données originaux entre eux. De plus le même pipeline d'en-

traînement d'un modèle peut ainsi être utilisé quel que soit l'ensemble de données considéré.

La **standardisation** consiste, pour chaque colonne de l'ensemble de données original $\mathscr{D}_{orig}$ (matrice de vecteurs-lignes), à centrer puis réduire ses valeurs afin qu'elle contienne des *Z-scores* et corresponde à une loi normale $\mathscr{N}(\mu = 0, \sigma = 1)$ :

$$x_{i,j} = \frac{x_{i,jorig} - \mu_j}{\sigma_j}$$

Avec $\mu_j$ et $\sigma_j$ les moyenne et écart-type pour la colonne $j$ respectivement. La standardisation est la normalisation la plus fréquemment utilisée.

L'**uniformisation** correspond à remplacer $x_{i,j}$ par son rang (quantile) dans $x_j$, c'est-à-dire la valeur de la fonction de répartition pour ce $x_{i,j}$ :

$$x_{i,j} = \int_{-\infty}^{x_{i,jorig}} f(x_i)$$

où $f$ est la fonction de densité de $x_i$, souvent estimée à partir d'un sous-ensemble de $\mathscr{D}_{orig}$ tiré aléatoirement. Cette technique de normalisation est plus adaptée aux cas où les données à disposition suivent une loi dégénérée, par exemple à forte déviation (lorsque la médiane est très éloignée de la moyenne), provocant l'écrasement des valeurs dans le mode de $x_i$. De manière générale, on peut considérer que l'uniformisation est une normalisation plus robuste que la standardisation.

# CHAPITRE 2

# MODÈLES ÉTUDIÉS

Ce chapitre est une introduction à la littérature scientifique concernant la famille d'algorithmes utilisés dans ce mémoire. Ici, nous décrivons dans un premier temps les modèles linéaires, puis les modèles non-linéaires, et détaillons ensuite certains principes régissant l'utilisation des architectures profondes qui sont mises à parti dans nos travaux. Les modèles spécifiquement développés dans le cadre de notre recherche seront abordés au chapitre 4 après avoir décrit le domaine d'application.

## 2.1 Modèles linéaires

### 2.1.1 Perceptron

Un modèle paramétrique de classification élémentaire est la classification linéaire à deux classes : on modélise un hyperplan $H$ (appelé frontière de décision) séparant $\mathscr{D}$ en deux sous-ensembles $C_1$ et $C_2$, et la classification d'un point $x_{test} \in \mathbb{R}^d$ se fait en évaluant sa position par rapport à $H$. L'hyperplan est défini par un vecteur de poids $w \in \mathbb{R}^d$ et un biais $b \in \mathbb{R}$, paramètres du modèle. On appelle **fonction discriminante** la fonction $g(x) = b + w^T \cdot x = b + \sum_i w_i x_i$ qui détermine si $x$ se trouve d'un côté de $H$ ou de l'autre. Enfin, la fonction de décision prédisant la classe de $x$ est :

$$f(x) = \begin{cases} C_1 \text{ si } g(x) < 0 \\ C_2 \text{ si } g(x) > 0 \end{cases}$$

Il est à noter que telle que $f$ est décrite ci-dessus, la fonction de coût n'est pas dérivable par rapport à $\theta$, à cause de la discontinuité de $f$ quand $g(x)$ vaut 0. Aussi, en essayant de déterminer la position d'un hyperplan séparant $\mathscr{D}$ en deux sous-ensembles distincts, ces modèles requièrent que $\mathscr{D}$ soit **linéairement séparable**, ce qui n'est pas forcément le cas.

En modifiant $f$ de façon à ce qu'elle renvoie directement la distance de $x$ à $H$, et en choisissant une fonction de perte proportionnelle à $g(x)$ et nulle lorsque $f$ ne se trompe pas,

on obtient une fonction de perte compatible avec l'algorithme de descente de gradient :

$$\begin{aligned} L(f, z_i) &= -y_i g(x_i) \mathbb{1}_{f(x_i) \neq y_i}, \text{ avec} \\ & y_i \in \{-1, 1\}, \\ & \mathbb{1}_x = \begin{cases} 1 & \text{si x est vrai} \\ 0 & \text{sinon} \end{cases} \end{aligned} \tag{2.1}$$

C'est la règle de mise à jour du **perceptron** [34].

### 2.1.2 Régression logistique

Une variation sur la régression linéaire consiste à appliquer à la fonction discriminante $g$ une fonction logistique en sortie (sigmoïde par exemple). La fonction sigmoïde projette une entrée réelle sur $[0, 1]$ selon $sigmoid(x) = 1/(1 + e^{-x})$ ; il est donc possible d'interpréter sa valeur comme $p(y = 1|x)$, la probabilité que $x$ appartienne à la classe $C_1$, et $f$ peut prédire la classe de $x$ en utilisant un seuil dans $[0, 1]$, typiquement $0.5$ :

$$g(x) = sigmoid(b + w \cdot x)$$

$$f(x) = \begin{cases} C_1 \text{ si } g(x) \leqslant 0.5 \\ C_2 \text{ si } g(x) > 0.5 \end{cases}$$

La fonction de coût habituellement associée à la non-linéarité sigmoïde est l'**entropie-croisée** :

$$L(f, (x, y)) = -y \log(f(x)) - (1 - y) \log(1 - f(x))) \mid y \in \{0, 1\}$$

### 2.1.3 SVM

Dans les cas où la condition de séparabilité est satisfaite, le problème de trouver un hyperplan séparateur accepte plusieurs solutions possibles : il existe en effet une infinité de positions de $H$ pour lesquelles l'erreur sera nulle. Les machines à vecteurs de support (*SVM*) résolvent ce problème en ajoutant la contrainte de maximiser la distance entre $H$ et les points les plus proches de ce plan.

### 2.1.4 Séparation non-linéaire

Il existe diverses methodes pour transformer un $\mathscr{D}$ non-linéairement séparable en $\tilde{\mathscr{D}} = \sigma(\mathscr{D})$ linéairement séparable, afin d'entraîner le modèle sur $\tilde{\mathscr{D}}$. La fonction $\sigma$ pourra être une projection explicite choisie arbitrairement, une projection implicite utilisant un noyau (*kernel*) comme dans les machines à vecteurs de support, ou encore une projection elle-même **apprise**, de la même forme que $g$. C'est sur des variations de ce type de modèle que se concentre ce mémoire.

## 2.2 Modèles non-linéaires

### 2.2.1 Réseaux de neurones

Le réseau de neurones (*artificial neural network*) tient son nom des similarités originelles avec les connaissances contemporaines du cerveau aux niveaux structurel et fonctionnel. En termes assez généraux, tous deux sont composés de couches de neurones traitant l'information issue de la couche précédente, la toute première couche étant le vecteur de données lui-même (ou un lot de vecteurs, le cas échéant), et la dernière la prédiction. Il est important de noter que cette similarité apparente a des limites. Les réseaux de neurones artificiels sont plus des outils mathématiques que les simulations d'une réalité biologique, et la qualité de leurs prédictions compte davantage que leur fidélité biologique, bien que la recherche en neurosciences puisse servir d'inspiration plus ou moins directe à d'éventuels modèles.

Tel que mentionné au paragraphe précédent, un réseau de neurones est composé de plusieurs couches. Une transformation affine possiblement suivie d'une non-linéarité compose la couche dite **de sortie** (*output layer*), prenant en entrée la **représentation** apprise d'une couche intermédiaire dite **cachée** (*hidden layer*), qui elle-même transforme une **couche d'entrée** (*input layer*). Cette couche d'entrée correspond soit à une normalisation des données (1.9), soit à l'identité, c'est à dire que l'on pourra considérer les données comme pré-normalisées. Cette architecture en couches donne au réseau de neurones une capacité plus importante que celle d'un modèle à l'architecture plus plate, à nombre de paramètres égal. On dit que ce modèle est un approximateur universel [21], c'est-à-dire qu'il peut modéliser n'importe quel type de fonction pour peu qu'il dispose d'une capacité suffisante.

Formellement, un réseau de neurones est une fonction $\mathbb{R}^d \mapsto \mathbb{R}^m$ telle que :

$$\hat{y} = f(x) = g(h(x)) = act_{out}(b + W act_{hid}(c + Vx))$$

avec $x \in \mathbb{R}^d$ le vecteur d'entrée, $c \in \mathbb{R}^h$ et $V \in \mathbb{R}^{d \times h}$ les biais et poids de la couche cachée, $b \in \mathbb{R}^m$ et $W \in \mathbb{R}^{h \times m}$ ceux de la couche de sortie de taille $m$, et $act_{hid}$ et $act_{out}$ les fonctions d'activation des couches cachée et de sortie respectivement, qui transforment les activations de leur couche en sorties. Ce sont le plus souvent deux fonctions non-linéaires de type sigmoïde ou tangente hyperbolique. Notons que $f$ décrit ici un réseau de neurones à une couche cachée. Des réseaux plus profonds peuvent être obtenus en ajoutant d'autres couches cachées.

Cet algorithme est souvent appelé perceptron multi-couches (MLP). Il diffère cependant du perceptron par l'utilisation de fonctions d'activations non-linéaires. Les sorties de la couche cachée $H(x)$ sont appelées **unités cachées**. L'entraînement des paramètres du réseau de neurones est composé de deux phases, premièrement de propagation-avant (*feed-forward propagation*) et ensuite de rétro-propagation,

### 2.2.2 Calcul des mise à jours des poids

L'entraînement des réseaux de neurones est un bon exemple d'application de la règle de dérivation en chaîne (1.8). Elle y est utilisée pour rétro-propager le gradient qui servira à optimiser les paramètres utilisés dans le calcul du **critère d'optimisation**.

Prenons un modèle avec pour sortie un vecteur de probabilités $o^s(x) = softmax(o^a)$, pour fonction de perte la log vraisemblance négative (1.2), et pour fonction d'activation de couche cachée la *tanh*, qui attribue à chaque unité de sortie $o^s_k$ une valeur dans $[-1, 1]$ :

$$softmax(x) = \frac{e^x}{\sum_k e^{x_k}},$$

et

$$tanh(x) = \frac{e^x - e^{-x}}{e^x - e^{-x}}.$$

Comme le risque est la moyenne de la perte calculée sur chaque exemple de $\mathscr{D}$ , calculer sa dérivée par rapport à un paramètre revient à calculer la moyenne des dérivées de la perte sur chaque point de $\mathscr{D}$ :

$$\frac{\partial \widehat{R}}{\partial \theta_i} = \frac{1}{n} \sum_{j=1}^{n} \frac{\partial L(f_{\theta_i}, z_j)}{\partial \theta_i}$$

Les mise à jours $\Delta\theta_i = \lambda \frac{\partial \widehat{R}}{\partial \theta_i}$ des paramètres du réseau sont calculées en «descendant» le long du graphe de calcul de $\widehat{R}$ de la façon suivante. On commence par calculer le gradient de la

perte par rapport aux sorties du réseau :

$$\frac{\partial L}{\partial o_k^s} = \frac{\partial - \log\ o_y^s}{\partial o_k^s} = \frac{-1}{o_k^s} \text{ pour k=y, 0 sinon} \tag{2.2}$$

Puis le gradient des activations par rapport aux activations des unités de sortie. Notons que comme seul le gradient du coût par rapport à $o_y^s$ de sortie n'est pas nul, c'est lui qui nous intéresse. De plus en raison de la softmax qui utilise *toutes* les activations de la couche pour le calcul de *chaque* sortie, il existe $m$ chemins entre un $o_k^s$ et un $o_j^a$ :

$$\begin{aligned}
&\frac{\partial o_{k=y}^s}{\partial o_k^a} = \frac{\partial}{\partial o_k^a} \frac{e^{o_y^a}}{\sum_{k'=1}^m e^{o_{k'}^a}} = 2 \text{ cas possibles :} \\
&\begin{cases}
\dfrac{e^{o_y^a} \sum_{k'=1}^m e^{o_{k'}^a} - e^{2o_y^a}}{\left(\sum_{k'=1}^m e^{o_{k'}^a}\right)^2} = o_y^s - 2o_y^s = o_y^s(1 - o_y^s), & \text{pour } k = y \\
\dfrac{0 - e^{o_y^a} e^{o_2^a}}{\left(\sum_{k'=1}^m e^{o_{k'}^a}\right)^2} = -o_y^s o_k^s, & \text{pour } k \neq y
\end{cases}
\end{aligned} \tag{2.3}$$

Enfin, le gradient des activations par les paramètres est donné par :

$$\frac{\partial o_k^a}{\partial W_{j,k}} = \frac{\partial \sum_{k'=1}^m b_{k'} + W_{j,k'} h_j^s}{\partial W_{j,k}} = h_j^s$$

et

$$\frac{\partial o_k^a}{\partial b_k} = \frac{\partial b_k + W_k h^s}{\partial b_k} = 1.$$

On peut alors calculer les valeurs des mise à jour des poids de la couche de sortie :

$$\frac{\partial L}{\partial W_{j,k}} = \frac{\partial L}{\partial o_k^s} \frac{\partial o_k^s}{\partial o_k^a} \frac{\partial o_k^a}{\partial W_{j,k}}$$

et

$$\frac{\partial L}{\partial b_k} = \frac{\partial L}{\partial o_k^s} \frac{\partial o_k^s}{\partial o_k^a} \frac{\partial o_k^a}{\partial b_k}$$

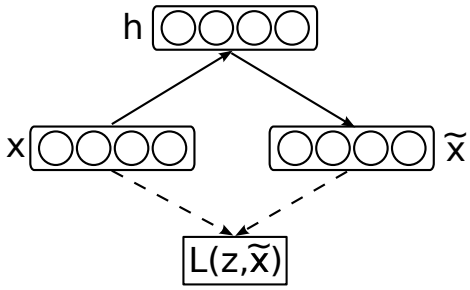
Le calcul des gradients du coût selon les paramètres $V$ et $c$ de la couche cachée se déroule de la même façon mais ne seront pas décrits ici pour des raisons d'espace et de lisibilité.

### 2.2.3 Auto-encodeurs

L'auto-encodeur représente un type particulier de réseau de neurones pour lequel la cible est en fait son entrée, comme l'illustre la figure 2.1. Une fonction paramétrisée d'extraction d'attributs (fonction d'encodage) est d'abord appliquée sur l'entrée, afin de créer un vecteur

d'attributs appelé **représentation cachée** ou encodage. Une seconde fonction, paramétrisée elle aussi, projette le vecteur d'attributs de l'espace de la représentation cachée à l'espace du vecteur d'entrée, et crée ainsi **une reconstruction** ; c'est la fonction de décodage. Le modèle apprend donc à reconstruire son entrée au moyen de deux fonctions composées, et l'entraînement consiste à minimiser, au sein de l'ensemble d'entraînement, la différence entre l'entrée et sa reconstruction.

Figure 2.1 – Architecture d'un réseau de neurones auto-associateur




L'entrée $x$ est projetée en $h$ dans l'espace de la couche cachée au moyen d'une somme des biais et du produit matriciel de $x$ avec les poids $W$ du réseau, puis cette représentation cachée $h$ est reprojetée en $\tilde{x}$ dans l'espace de la couche d'entrée. L'optimisation des paramètres se fait en minimisant l'erreur $L(x, \tilde{x})$.

L'**hypothèse de variété** (*manifold hypothesis* [12]) dit qu'un ensemble de données a un nombre de degrés de variation inférieur ou égal à sa dimensionalité. Une autre formulation est que la majorité des exemples se concentre dans une région précise de l'espace. En reparamétrisant cet espace, il est possible d'en représenter les points dans un espace à moins de dimensions. Apprendre une telle reparamétrisation est le but de l'auto-encodeur, qu'on peut alors utiliser comme algorithme de compression avec perte, ou comme algorithme de réduction de dimensionalité.

Afin d'éviter que le modèle n'apprenne que la fonction d'identité, on pourra soit choisir une taille de couche cachée $h$ inférieure à $m$, de façon à forcer l'**extraction des caractéristiques** de $\mathscr{D}$ , soit utiliser une couche cachée de taille supérieure à $m$, en prenant soin d'ajouter des contraintes (régularisations) sur la nature de cette représentation afin qu'elle soit riche en information. Un exemple de contrainte applicable à une représentation sur-complète est celle de la sparsité : le fait pour un tenseur d'avoir une grande majorité d'indices à 0. On parlera de représentations *sous-complète* et *sur-complète* respectivement, selon que $h < m$ ou que $h > m$.

Lorsque des fonctions d'activations linéaires sont employées, il a été montré que la représen-

tation apprise par le modèle peut correspondre à une rotation des composantes principales de $\mathscr{D}$ [2]. Ceci est donc similaire à l'analyse en composantes principales, qui consiste à projeter un ensemble de points dans un espace de plus faible dimension, dans lequel ils sont linéairement indépendants, de telle façon que les variances des points dans cet espace sont décroissantes au fil des dimensions, c'est-à-dire que les dimensions portent de moins en moins d'information sur l'ensemble de données. En général, et c'est la justification d'un tel traitement des données, on ne conserve que les $n$ premières dimensions. Pour l'auto-encodeur, c'est la taille de la couche cachée qui détermine les composantes conservées.

Avec l'utilisation de fonctions d'activations non-linéaires, on peut s'attendre à ce que les représentations créées soient encore plus riches en information, et puissent être utilisées pour diverses tâches. On peut par exemple imaginer entraîner un algorithme de partitionnement sur les représentations apprises, au lieu de le faire sur les exemples directement.

L'auto-encodeur est un exemple de modèle paramétrique où l'on peut appliquer la régularisation des poids liés. Ainsi, la matrice de poids $W$ servira à l'encodage de $x$, et sa transposée $W'$ sera utilisée pour le décodage. Formellement, l'auto-encodeur sera donc la composition d'une fonction d'encodage $h(x)$ et d'une fonction de décodage $g(x)$ :

$$h_{ae}(x) = act_h(b + Wx)$$
$$\hat{y} = g_{ae}(h_a e(x)) = act_g(c + W'h(x))$$

où $act_h$ est typiquement une sigmoïde ou une tangente hyperbolique, et $act_g$ peut être linéaire, auquel cas on utlisera l'erreur quadratique pour mesurer l'**information mutuelle** entre $x$ et $\hat{y}$, ou non-linéaire et on utilisera l'entropie croisée. En général, on interprétera $\hat{y}$ non-pas comme la reconstruction exacte de $x$ mais plutôt comme les paramètres (par exemple la moyenne) d'une distribution ayant une forte probabilité de générer $x$.

## 2.3 Architectures profondes

Nous entendons par *architecture* l'idée derrière un modèle, le concept qu'il implémente. Une unique architecture pourra donc être implémentée par des modèles différents, qui auront alors une assez forte ressemblance, mais diffèreront par certains hyper-paramètres (dimensions, profondeur, durée d'entraînement, etc).

### 2.3.1 Problématique

La complexité du monde qui nous entoure se retrouve dans les données qui le représentent et nécessite d'imaginer des modèles pouvant modéliser ses très fortes non-linéarités. Un avantage particulier que le réseau de neurones partage avec quelques autres types de modèles est qu'on peut **empiler plusieurs couches cachées** de façon à démultiplier sa capacité.

Bien que le réseau de neurones soit un approximateur universel, modéliser une famille de fonctions très complexe peut demander un nombre de paramètres exponentiellement plus élevé qu'en augmentant sa profondeur [3]. En **composant** les résultats des couches précédentes, chaque nouvelle couche peut tirer avantage des non-linéarités des précédentes pour produire sa représentation, qui peut donc capter des variations plus complexes de la distribution étudiée. Comparons un instant notre modèle au pédalier d'un vélo : ses axes correspondent aux couches cachées, ses unités cachées aux plateaux (les engrenages), et sa capacité est le nombre de vitesses modélisables en reliant ces plateaux (en les combinant). Avec deux axes et trois plateaux par axe, il est possible de modéliser 9 vitesses. Dans le contexte où deux nouveaux plateaux sont disponibles, étudions l'effet de leur placement. En plaçant les deux plateaux sur un même axe, on augmente le nombre de vitesses possibles à 15. En plaçant un plateau sur chaque, on passe à 16. En revanche, si l'on ajoute un nouvel axe sur lequel on place les deux plateaux, il est désormais possible de modéliser 18 vitesses différentes. Augmenter la profondeur d'un modèle a le même effet sur sa capacité.

La composition d'une pile de couches les unes «au dessus» des autres est une contrainte **hiérarchique** qui s'avère refléter assez fidèlement ce que l'on peut observer chez beaucoup de systèmes témoignant d'intelligence. L'étude de ces derniers, naturels ou artificiels, tend en effet à montrer qu'ils observent dans nombre de cas une structure hiérarchique. Prenons par exemple la société : elle est composée à la base d'individus, dont certains s'unissent en familles, qui elle-mêmes s'unissent en communautés, puis en nations, etc, et chacune de ces entités est chargée d'**une fonction différente mais dépendante des autres**, bien qu'elles soient toutes identiques au moins en ce sens qu'elles ne sont que des groupes de personnes liées par des affinités et des buts communs (la survie, le partage et le développement d'une culture, etc).

L'observation la plus motivante vient cependant de l'étude du cerveau animal, qui montre [22] qu'une partie au moins du système visuel traite l'information venant du nerf optique dans des régions successives (V1, V2, V5 et enfin lobe pariétal) et hiérarchisées, c'est-à-dire appliquées à opérer différentes transformations spécifiques au signal qu'elles reçoivent de

régions précédentes. De la même façon, on pourra espérer que chaque couche d'un modèle hiérarchique puisse *capitaliser* sur les représentations successives des couches précédentes et *abstraire* l'information de façon à élargir le champ d'application du modèle.

Créer un réseau de neurones multi-couches d'après les informations énoncées jusqu'à présent, c'est-à-dire en empilant plusieurs couches cachées et procédant à un entraînement supervisé par descente de gradient, est en réalité une tâche ardue, surtout quand il s'agit d'en tirer de bonnes performances. Cela en vient en partie de la technique d'optimisation employée. Les mises à jour des poids semblent moins aptes à passer certains minima locaux de la fonction d'erreur, et n'influencent donc pas assez les poids des couches inférieures pour entraîner le modèle efficacement.

### 2.3.2 Pré-entraînements non-supervisés

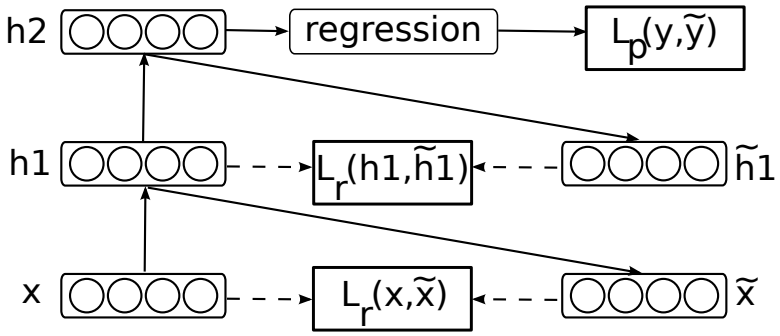
En 2006, Hinton *et al* [20] ont l'idée de précéder l'entraînement supervisé de leur réseau profond (un *deep belief net* détaillé dans le même article) par des entraînements **successifs et non-supervisés** de ses couches intermédiaires comme des machines de Boltzmann restreintes (RBMs). Dans ce contexte d'entraînement en deux phases, on parlera de **pré-entraînement** et de *fine-tuning* des paramètres du modèle.

Une RBM et un auto-encodeur sont différents dans leur entraînement et leurs justifications théoriques, mais ne sont pas si différents dans leur fonctionnement. Une différence essentielle est qu'un auto-encodeur utilise les sorties de sa fonction d'activation cachée comme valeurs de sa couche cachée directement, alors qu'une RBM échantillonne selon une loi de Bernoulli dont les probabilités sont données par ces sorties, et utilise cette représentation binaire comme couche cachée.

Les résultats obtenus avec un tel type d'entraînement ont dépassé l'état de l'art de l'époque et ouvert la voie vers un essor du pré-entraînement non-supervisé dans les modèles profonds en général, y compris dans les réseaux de neurones étudiés dans ce document. La figure 2.2 illustre l'agencement des différentes couches et la façon dont elles sont entraînées.

Le pré-entraînement (*unsupervised pretraining*) des architectures profondes se base sur **l'hypothèse semi-supervisée** [20], [5] qui formalise une notion intuitive : la distribution de probabilité de l'entrée $p(x)$ partage de l'information avec la probabilité conditionnelle des cibles selon les entrées $p(y|x)$. De façon imagée, on conçoit instinctivement qu'il est plus facile de différencier deux sous-champs d'un même domaine lorsque l'on a des connaissances de base dans ce domaine, ou qu'il n'est nul besoin de savoir qu'un chat est un chat et un

Figure 2.2 – Architecture d'un réseau profond




Un modèle à deux couches est présenté. $x$ et $h_1$ sont les entrées des première et seconde couches respectivement, $h_1$ et $h_2$ sont leurs couches cachées, enfin $\tilde{x}$ et $\tilde{h}_1$ sont leurs reconstructions, optimisées successivement lors de la phase de pré-entraînement en minimisant respectivement une fonction de perte $L_r$ (possiblement différente d'une couche à l'autre). La représentation cachée de plus haut niveau $h_2$ est fournie en entrée à un modèle plus simple, produisant les prédictions $\hat{y}$. Enfin, la phase de *fine-tuning* supervisée optimise l'ensemble des poids du modèle en minimisant la fonction de perte $L_p$ de plus haut niveau à l'aide des étiquettes $y$ de l'ensemble de données.

chien un chien pour apprécier des différences et les similarités entre les deux espèces. C'est l'avantage de connaître le *contexte* dans lequel on évolue. La tâche est dite *non-supervisée* car elle n'utilise pas d'étiquettes, mais des cibles sont tout de même utilisées : ce sont les entrées elle-mêmes. On se place donc dans le contexte d'**une tâche de reconstruction**. L'algorithme d'entraînement apprend à la couche $C_i$ à reconstruire la représentation cachée, ou encodage, de la couche $C_{i-1}$. L'information que chacune extrait et fournit à la suivante est donc d'une importance capitale.

On peut considérer le pré-entraînement non-supervisé comme une méthode de régularisation. En effet l'algorithme d'entraînement supervisé minimise la fonction de perte $L$ pour les exemples d'entraînement et se termine (par convergence ou en raison d'un autre critère d'arrêt) aussi bas que possible dans un minimum local de $L$ [1]. Dans l'étude des systèmes dynamiques, on appelle **bassin d'attraction** un sous-ensemble des états possibles de paramètres vers lequel converge les différentes trajectoires (successions d'état, aussi appelées orbites) d'une même règle d'évolution de ces paramêtres, à partir d'états intiaux différents. Il existe généralement une grande quantité de tels bassins dans les fonctions complexes utilisées en apprentissage machine, car il existe beaucoup de minima locaux à la fonction de perte,

1. Différentes techniques [28, 29] permettent, dans certains cas, de ne pas se retrouver piégé dans certains minima locaux peu intéressants, mais ce sont des ajouts à l'algorithme qui ne modifient pas fondamentalement son comportement.

mais l'erreur de généralisation obtenue après avoir convergé dans l'un de ces bassins varie beaucoup. L'initialisation aléatoire place les poids du modèle au hasard sur l'un d'entre eux et l'entraînement assure simplement que l'on descende sur $L$ aussi bas que possible [2]. On remarque généralement que précéder la phase d'entraînement supervisée par une phase de pré-entraînement non-supervisé permet de placer les paramètres $\theta$ près d'un bassin d'attraction pour lequel la généralisation est meilleure.

L'auto-encodeur classique n'impose aucune contrainte à son encodage. Le fait que l'erreur de reconstruction baisse considérablement lors de l'entraînement n'est pas une garantie que la représentation cachée porte une information très représentative en soit, car le modèle peut n'être qu'un très bon décodeur (à l'opposé d'un bon encodeur). Si la représentation de $x$ n'est pas de meilleure qualité que $x$ lui-même, la transformation est inutile et la couche suivante se retrouvera devant une tâche tout aussi ardue. En imposant à la phase d'encodage **une contrainte supplémentaire** garantissant certaines propriétés de la représentation, il est alors possible d'extraire de $x$ des informations de meilleure qualité, qui contribuent à simplifier le problème posé à la couche suivante. Pour des raisons pratiques, on choisira via le critère d'entraînement des contraintes non-supervisées et locales. Lorsque toutes les couches ont ainsi été entraînées, les paramètres du modèle sont dans une meilleure configuration que leur initialisation par défaut car elle est déjà dépendante de l'entrée, donc les représentations qu'ils produisent sont à priori plus **utiles**. On passe alors à la phase d'entraînement supervisé, le *fine-tuning*.

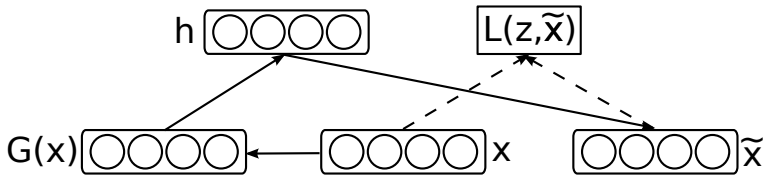
Définir l'utilité d'une représentation (sa qualité) est un problème difficile car on ne sait pas spécifiquement ce qui rend une représentation meilleure qu'une autre. Vincent *et al* utilisent dans [35] les représentations apprises par leurs auto-encodeurs débruitants pour entraîner un classifieur linéaire. Ils usent de pragmatisme en définissant une **bonne représentation** comme *celle qui résulte en un classifieur de meilleure performance*, mais rappellent avec sagesse que tout comme l'avancée de Hinton en 2006 a montré que la performance sur le critère supervisé n'est pas forcément le meilleur critère d'entraînement, on ne devrait pas se fier uniquement à la performance sur le critère supervisé final pour établir l'utilité de la représentation cachée.

2. D'autres technique de régularisation, telle que l'arrêt prématuré, peuvent limiter la taille de la trajectoire de cette descente.

#### 2.3.2.1 Pré-entraînement débruitant

Une première méthode pour favoriser la création de représentations utiles est l'utilisation d'un auto-encodeur débruitant (*denoising autoencoder*). C'est un auto-encodeur classique entraîné à reconstruire son entrée à partir d'une version **corrompue** de celle-ci. Sa tâche est donc d'en extraire les caractéristiques pertinentes de façon à palier aux erreurs qui y sont volontairement introduites. La figure 2.3 illustre son architecture.

Figure 2.3 – Architecture d'un réseau de neurones auto-associateur débruitant




Le processus $G(x)$ renvoie une version corrompue de l'entrée $x$. Cette version est projetée en $h$ dans l'espace de la couche cachée. La reconstruction $\tilde{x}$ est ensuite comparée non-pas à $G(x)$ mais à $x$. Ce procédé force le modèle à capter les caractéristiques essentielles de la distribution afin de pouvoir reconstruire $x$.

La corruption est partielle et déterminée par une fonction de bruit $q(x) = \tilde{x}_i$ dont la nature est déterminée au préalabe (hyper-paramètre) :

- bruit gaussien : tiré d'une normale centrée en x : $\tilde{x} = x + \mathcal{N}(x, \sigma^2)$,
- masque : un sous-ensemble des éléments $x_{i,j}$ d'un vecteur $x_i$, choisi aléatoirement, est forcé à 0.
- bruit poivre et sel : un sous-ensemble des éléments $x_{i,j}$ d'un vecteur $x_i$, choisi aléatoirement, est forcé à 0 ou à 1.

On s'attend à ce que la représentation apprise ainsi soit plus robuste à d'éventuelles corruptions des données utilisées pour son évaluation, et que la tâche de reconstruction tende à faire que le modèle favorise les propriétés de l'entrée les plus utiles à sa reconstruction. Dans le contexte de l'hypothèse de variété, on part du principe que les $x_i$ sont concentrés dans une région particulière de $\mathscr{D}$ . Les transformations bruitées $\tilde{x}_i$ peuvent être vues comme des points situés aux alentours de cette variété et que l'autoencodeur débruitant va apprendre à replacer dans des régions de plus forte densité.

Après avoir pré-entraîné une couche $n$, on pré-entraîne la couche $n+1$ sur une version corrompue de la sortie de la fonction d'encodage de $n$ appliquée **à une entrée non-corrompue**. En d'autres termes, la corruption de l'entrée n'est appliquée que pour la couche en cours de

pré-entraînement, et aucune corruption n'est utilisée lors de la phase de fine-tuning supervisé, ou pour l'évaluation des erreurs sur $\mathscr{D}_{valid}$ ou $\mathscr{D}_{test}$ .

#### 2.3.2.2 Pré-entraînement contractant

L'entraînement d'un auto-encodeur contractant (*contractive autoencoder*) consiste à entraîner un auto-encodeur classique, toujours de façon non-supervisée, mais en ajoutant un terme au critère optimisé : une **pénalité de contraction**. Ce terme pénalise la sensibilité locale du modèle aux variations de son entrée, c'est-à-dire qu'elle tend à minimiser les variations de la représentation cachée pour de faibles variations de l'entrée autour de celle-ci. Cette technique s'inspire aussi de l'hypothèse de variété.

La pénalité correspond à la norme de Frobenius de la dérivée partielle de la représentation cachée $h(x)$ par l'entrée $x$ :

$$L_{cae}(f_\theta(x), y) = L_{ae}(g(h(x)), y) + \lambda \sum_j \left\| \frac{\partial h_j(x)}{\partial x} \right\|^2$$

Le premier terme pousse $f$ à obtenir une reconstruction aussi proche que possible de $x$, alors que le second pousse la représentation cachée à être localement invariante à des variations de $x$, autrement dit à contracter la projection des $x_i$ dans le voisinage de $x$, dans l'espace des représentations. Alors que la pénalité de contraction, employée seule, le pousserait globalement à ignorer ses entrées et ne serait donc pas très utile, la combinaison des deux termes en opposition fait en sorte que le modèle a tendance à n'être sensible qu'aux variations de $x$ ayant un impact manifeste sur sa reconstruction : les directions *dans le sens* de la variété de $\mathscr{D}$ .

# CHAPITRE 3

# INTRODUCTION AU DOMAINE D'APPLICATION

## 3.1 La recommandation

Les *articles* mentionnés au premier chapitre peuvent être de toute nature ; il peut s'agir aussi bien d'articles de presse [13], de films [27, 33], de livres [26], CDs et même, comme ce sera le cas dans ce document, des niveaux d'un jeu vidéo. Tout ce qui peut être l'objet d'une appréciation mesurable peut être l'objet d'une recommandation. La recommandation est faite pour un usager à la fois.

## 3.2 Le jeu vidéo

Dans le contexte du jeu vidéo, le **niveau** est le principal type de sous-espace du jeu dans lequel le joueur fait évoluer un ou des avatars jusqu'à sa complétion. La complétion d'un niveau correspond à la validation d'un critère pouvant prendre des formes diverses (terminer un parcours, vaincre un adversaire spécifique, récolter une certaine quantité de ressources, etc). La complétion d'un niveau est généralement marquée par l'attribution d'un score (points, crédits, temps, etc) au joueur, témoignant de la qualité de son jeu.

Un jeu vidéo peut contenir un unique niveau et sera alors appelé jeu bac-à-sable (*sandbox game*), ou une multitude d'entre eux (et n'aura pas de nom spécifique car c'est la structure tradionnelle). Dans certains cas, les niveaux sont créés par le développeur du jeu, et dans d'autres par les joueurs eux-mêmes (*user generated content games*). Le contenu des jeux *UGC* peut rapidement prendre une taille gigantesque. C'est le cas par exemple de *Little Big Planet* (développé par Media Molecule) qui a récemment fêté son 7 millionième niveau créé par un joueur, presque quatre ans après avoir été publié, et dont les serveurs voient près de 5000 nouveaux niveaux être créés chaque jour [16]. En comparaison, Super Mario Galaxy 2 (développé par Nintendo) contient 242 niveaux. Dans un jeu *UGC*, il est impossible pour un joueur d'essayer tous les niveaux pour savoir lesquels lui plaisent le plus, ou ne serait-ce que lire leur description. C'est dans ces contextes que les systèmes de recommandation prennent toute leur valeur.

### 3.3 Définition du problème

Le but de la recommandation est de founir un résultat **pertinent**, mais ce critère n'est pas toujours facile à estimer. Dans les cas les plus simples, l'utilisateur a la possibilité de donner **une note** (un *rating*) aux articles qu'il consomme. Cette note peut-être appelée **utilité** [1]. Dans d'autres cas, il faut user de mesures mandataires (*proxy*) pour estimer son appréciation. Dans le cadre d'une boutique en ligne, on pourra par exemple compter la visite d'une page dédiée à un produit comme un signe d'intérêt de la part de l'utilisateur pour ce produit (même s'il ne finit pas par l'acheter).

La distinction suivante sera en vigueur tout au long de ce mémoire, et prendra certainement plus de sens au fil de la lecture : on parlera de *note* pour une appréciation réellement donnée par un joueur et donc contenue dans $\mathscr{D}$ , et on parlera de *score* pour la prédiction d'une note, autrement dit un score est une note possible, générée par le modèle.

L'appréciation d'un utilisateur $c \in C$ pour un article $s \in S$ est représentée par une fonction d'utilité $u_{c,s}$. C'est la **note** qu'elle fournit qui permet de trier les articles par ordre de préférence estimée, et c'est donc en étant capable de prédire avec précision les notes des articles que l'utilisateur n'a pas encore notés qu'il va être possible de lui suggérer les articles ayant les scores les plus élevés. Plus formellement, on choisira pour l'utilisateur $c_i$ l'article $s^*$ tel que :

$$s^* = \underset{s \in S}{\operatorname{argmax}}\, u(c_i, s)$$

Aussi, on appellera $U_c = \{u(c,s) | \forall s \in S\}$ l'ensemble des notes attribuées par l'utilisateur $c$, et $U_s = \{u(c,s) | \forall c \in C\}$ l'ensemble des notes attribuées à l'article $s$. De plus, $U = \{u(c,s) | \forall c \in C,\ \forall s \in S\}$ est l'ensemble des notes, et $u(c,s) \in V$ où par exemple $V = \{0,1,2,3,4\}$ est l'ensemble des valeurs possibles d'une note (de une à cinq étoiles).

Afin de générer les scores des articles pour un utilisateur, les algorithmes de la littérature font usage de toute information à leur portée. Cela peut passer par les mesures mandataires mentionnées précédemment, mais aussi par l'usage des notes des autres utilisateurs. Intuitivement, on comprend que si deux joueurs ont des goûts similaires, et attribuent donc des notes similaires aux articles qu'ils ont tous deux consommés, ils ont de bonnes chances de continuer à attribuer des notes similaires aux prochains articles sur lesquels ils passeront. Ainsi, beaucoup d'algorithmes de la littératures se basent essentiellement sur les similarités entre utilisateurs. Ils sont dits **collaboratifs** (*collaborative recommenders*). À l'opposé, un utilisateur ayant aimé un certain article pourrait avoir *a priori* plus de chances d'apprécier

un article similaire qu'un article aléatoire. Les modèles conçus selon cette intuition sont dits **orientés-contenus** (*content-based*). Enfin, un dernier type de **modèle hybride** combine les deux approches.

Souvent, lorsqu'une organisation est en mesure de récolter les notes de ses utilisateurs, elle est aussi en possession d'autres informations sur eux. Ainsi, quasiment tous les sites internet savent par exemple le genre et la tranche d'âge de leurs utilisateurs, sans avoir besoin de leur demander, et peuvent multiplier le volume de ces données en enregistrant les moindres faits et gestes des utilisateurs au moyen de sytèmes de suivi (*tracking systems*). On appellera la combinaison des données personnelles et des notes d'une personne son *profil*. Certains modèles se basent sur le profil des utilisateurs pour produire des suggestions. Cette catégorie d'algorithmes est donc plus exigeante en données, mais permet de mieux grouper les utilisateurs par goût.

## 3.4 Tâches du filtrage collaboratif

Il existe plusieurs contextes dans lesquel un algorithme de recommandation peut être utilisé. De ces contextes dépend la façon dont on évaluera la performance de ce modèle. Cette section explique les différentes tâches possibles du filtrage collaboratif décrites dans [18].

Lorsque les articles sont tous présentables à l'utilisateur, soit parce qu'ils sont suffisament peu nombreux, soit parce qu'il est difficile de les sortir de leur contexte (par exemple les commentaires successifs d'une discussion), on peut vouloir intégrer leurs scores à leur présentation afin d'indiquer à l'utilisateur leur degré de pertinence. L'utilisateur est alors libre de suivre ou non ces indications. Cette tâche est appelée **annotation en contexte**.

Lorsque les articles peuvent être sortis de leur contexte, on cherchera alors à ne présenter que ceux dont le score est supérieur à un seuil, ou on selectionnera les $k$ meilleurs. On appelera cette tâche ***recommender de bons articles***. Elle est la plus fréquente en recommandation ; c'est notamment celle à laquelle les travaux présentés dans ce document s'appliquent. Elle présume l'indépendance des articles entre eux, tout comme on présume que les $x_i$ de $\mathscr{D}$ sont iids (1.2). Souvent les scores ne sont pas présentés directement, car on ne sait pas toujours expliquer pourquoi un modèle note deux articles différemment. C'est en particulier un inconvénient avec les réseaux de neurones.

Une variante de la tâche *recommender de bons articles* est ***recommender tous les bons articles***. Certaines situations exigent en effet de préserver tous les résultats pertinents, comme dans le milieu législatif où des avocats voudront examiner même les cas les plus rares, tant

qu'ils sont pertinents, pour étayer leur argumentation.

La **recommandation de séquence** vise à produire une liste **homogène** d'articles, c'est-à-dire des articles corrélés selon un critère donné. Un site d'écoute par exemple, voudra suggérer des morceaux d'un même style musical. L'appariement (*match making*) relève de cette catégorie. Dans le milieu vidéo-ludique multi-joueurs en ligne, des algorithmes de recommandation sont parfois chargés de la composition des équipes. Un joueur ne s'y voit pas proposer un choix d'adversaires et de coéquipiers, mais ce sont les statistiques prélevées sur son compte (son profil), ainsi que celles de autres joueurs connectés au même moment, qui servent d'entrées à l'algorithme. TrueSkill, conçu par Microsoft [17], est un des plus populaires algorithmes de ce domaine.

## 3.5 Exploitation et exploration

Savoir modéliser le profil et les goûts de ses utilisateurs n'est pas toujours suffisant. Du point de vue de l'usager, il peut y avoir en effet peu à gagner à se voir proposer des articles dont on connait déjà l'existence. L'exemple du supermarché explique bien le problème : on peut concevoir un algorithme de recommandation qui prédit avec une grande précision qu'un client donné achètera des oeufs et du beurre (l'exploitation), mais lui suggérer de tels choix n'a aucun intérêt car il repartira de toute façon avec ces produits. Il faut alors **prendre un risque** et proposer des articles qui sortent de la distribution de l'usager (l'exploration), des articles qu'il n'a pas encore considérés consommer et pour lesquels il pourrait effectivement profiter d'une recommandation. Dans notre cadre, étant donné que le nombre de niveaux est plus ou moins égal au nombre de joueurs, ces *articles récurrents* ont très peu de chances d'exister, et l'état de l'ensemble de données dans la période sur laquelle s'étend la conception de ce mémoire ne permet pas de les détecter. Cet aspect de la recommandation est donc pertinent, mais probablement peu influent dans l'état actuel des choses.

## 3.6 Métriques pour la mesure de performance

On peut mesurer la performance d'un modèle de plusieurs façons différentes. C'est même une bonne idée, puisque le critère utilisé pour l'entraînement du modèle ne reflète pas toujours fidèlement la tâche à laquelle la recommandation s'applique. Dans tous les cas, il sera impossible de mesurer la performance en exploitation sans se placer dans une perspective en ligne afin d'avoir un *feedback* en temps réel des utilisateurs, étant donné la forte sparsité des

ensembles de données, aspect inhérent à la recommandation.

On remarque en effet que dans nombre de cas [37], les distributions des notes respectent une loi exponentielle négative : une faible proportion d'article est notée un très grand nombre de fois, alors qu'une majorité d'articles n'est presque jamais notée. Cette *loi de Zipf*[1] est aussi appelée phénomène des longues queues (*long tails*). Elle représente l'ensemble des niches d'utilisateurs, ayant des goûts particuliers pour des articles tout aussi particuliers. Chacune de ces niches ne représente pas un marché intéressant à elle seule, mais c'est l'ensemble de ces niches qui peut représenter une part importante d'un marché. Un bon système de recommandation devra alors prendre en considération cet aspect du domaine.

### 3.6.1 Métriques de précision

Les **métriques de précision** (comme l'erreur quadratique moyenne) sont mal adaptées à la tâche *recommander de bons articles*. Imaginons un modèle faisant des prédictions parfaites pour un joueur, à l'exception qu'elles sont toutes biaisées identiquement. L'erreur sera alors proportionnelle au biais, alors que la selection des $k$ articles de plus haut score correspondra parfaitement aux goûts des utilisateurs. Aussi, on pourrait vouloir accorder moins d'importance à cette précision pour les articles se voyant attribués un mauvais score, puisqu'ils ne seront pas présentés dans les résultats. Ces métriques trouvent mieux leur place en revanche dans la tâche d'annotation en contexte, puisqu'elles permettent à l'utilisateur de juger de l'importance qu'il peut accorder aux articles.

### 3.6.2 Métriques de classification

Partant du principe que nos modèles de recommandation font de la régression, voir le problème comme une tâche de classification revient soit à considérer chaque note possible comme une classe et ainsi arrondir les scores à la classe la plus proche, soit à utiliser un seuil pour faire de la classification binaire. La classification binaire se prête bien à la tâche *recommander de bons articles*. Les métriques de classification, dans le contexte d'une classe par note, sont plus tolérantes aux déviations des scores par rapport aux notes que les métriques de précision, de fait de l'arrondi, mais souffrent en réalité des mêmes défauts. Elles ne cherchent pas à mesurer la précision des scores du modèle, ni ne sont spécifiquement adaptées aux contextes de la recommandation mentionnés ci-haut.

1. En référence au linguiste George Kingsley Zipf ayant observé le phénomène auprès de la fréquence des mots du discours.

### 3.6.3 Métriques de rang

Un troisième type de métrique est la famille des **métriques de rang**. Elle n'a pas été mentionnée dans l'introduction car elle n'est pas un critère d'entraînement de modèle, mais purement une mesure de son efficacité **après** entraînement. Plusieurs métriques existent.

La mesure des $k$ meilleures recommandations consiste à calculer la moyenne des $k$ notes des articles ayant le meilleur score pour chaque utilisateur, puis de prendre la moyenne de ces valeurs :

$$top_k = \frac{1}{|C_{test}|} \sum_{c \in C_{test}} \frac{1}{k} \sum_{k'=1}^{k} u(c, s_{k'}), \text{ avec } s_{k'} \in \underset{s_1,\dots,s_{k'}}{\text{k-max}} f(c, s),$$

où $C_{test}$ est l'ensemble des utilisateurs présents dans $\mathscr{D}_{test}$ , $f(c, s)$ est la fonction de prédiction du score de l'article $s_{k'}$ pour l'utilisateur $c$. Cette mesure ne peut pas être perturbée par un biais des scores car elle ne prend compte que les valeurs relatives des scores entre eux pour les ordonner. Elle est aussi assez peu sensible à des incohérences entre les scores relatifs des articles entre-eux, mis à part autour de $k$. Étant donc relativement stable, elle est préconisée pour la tâche *recommander de bons articles* [18**?** ] et a été la principale métrique pour l'évaluation des modèles étudiés dans ce document.

Les différentes mesures de performance utilisables souffrent toutes du même inconvénient inhérent au fait qu'on n'ait de cibles $u(c, s)$ que pour les articles que les joueurs ont effectivement déjà notés. Ainsi, cette dernière métrique mesure en réalité la capacité du recommandeur à proposer des articles que le joueur a *déjà* bien aimé, plutôt que des articles qu'il aimerait bien. C'est un biais qui masque la performance d'exploration. Il y a donc un grand intérêt à pouvoir tester les modèles en ligne, c'est-à-dire dans un environnement temps réel où l'on peut tester l'appréciation d'un utilisateur pour un article qu'il n'avait jamais vu auparavant, afin de mesurer effectivement la qualité de leur recommandation. Dans notre contexte, bien que le modèle selectionné à l'issue de ces expériences soit bien destiné à être intégré au jeu et à fonctionner dans un environnement *live*, le développement du jeu est tel qu'il est encore trop tôt pour se fier aux résultats qui pourraient être relevés, en raison de l'instabilité de son contenu.

Aussi, il est à noter que cette métrique n'est pas toujours bornée du même maximum. Comparons un instant la métrique $top_k$ à l'erreur de construction quadratique par exemple (MSE). La MSE du modèle est rarement nulle, mais en théorie elle pourrait l'être, **quelles que soit les données considérées**, si le modèle produisait une reconstruction parfaite. Dans

le cas de $top_k$ , rien ne garantit que tous les utilisateurs aient attribué la note maximale à au moins $k$ articles. De ce fait, la valeur maximale de la métrique **change selon l'ensemble de données considéré** (quel que soit l'intervalle de normalisation de ses valeurs). Bien qu'elle tende vers une borne supérieure quand le nombre de notes augmente, ça n'est pas le cas dans notre ensemble de données, c'est pourquoi nous considérerons en réalité le **ratio** $\frac{top_k}{top_k^*}$ pour rapporter la mesure $top_k$ . $top_k^*$ est la valeur de $top_k$ lorsque les notes elles-mêmes sont choisies en guise de scores, c'est-à-dire lorsque l'on fait les meilleures prédictions possibles étant donné l'ensemble de données évalué. Ainsi la valeur rapportée indiquera bien de combien elle est distante d'une même note cible (ici 1 à cause de la normalisation), identique d'une expérience à l'autre et permettant donc leur comparaison.

# CHAPITRE 4

# EXPÉRIMENTATIONS

L'objectif de nos expériences est double. D'une part, nous voulons trouver une architecture, voire un modèle en particulier, capable de prédire de façon satisfaisante les notes que les usagers attribuent aux articles. D'autre part, nous voulons justifier l'utilité des réseaux de neurones —possiblement profonds— pour la tâche de recommandation, et expliquer les variations d'auto-associateurs utilisés pour le pré-entraînement des dits réseaux. Afin de comparer ces modèles, nous mesurerons aussi les performances de quelques modèles moins élaborés.

Ce chapitre présente aussi le cadre expérimental dans lequel les expéricens ont été menées, ainsi qu'une description des données employées pour ces expériences.

Dans ce mémoire, tout comme en apprentissage machine en général, une expérience consiste à entraîner un modèle sur l'ensemble de données après l'avoir configuré selon une combinaison de valeurs des hyper-paramètres issue de l'hyper-optimisation. L'ordre des opérations est le suivant :

- on choisit des hyper-paramètres,
- on génère le modèle correspondant,
- on l'entraîne,
- on sauve les différentes métriques représentant sa performance.

Étant donnée l'optimisation par échantillonage aléatoire, les valeurs d'hyper-paramètres rapportées diffèrent d'un modèle à l'autre, mais comme le nombre d'expériences est suffisament grand on a en quelque sorte une garantie que ces valeurs sont approximativement optimales pour chaque modèle considéré, et justifient en conséquence la comparaison de modèles aux attributs différents.

## 4.1 Modèles comparatifs

### 4.1.1 Choix aléatoire

Le premier modèle comparatif que nous avons implémenté est très simpliste et mérite même à peine la dénomination de modèle, puisqu'il consiste à faire des prédictions **aléatoires**. Les scores rapportés pour ce modèle correspondent à des moyennes sur 100 tirages : 10 par sous-ensemble $\mathscr{D}_{test}$ , parmi 10 choix aléatoires de $\mathscr{D}_{test}$ dans $\mathscr{D}$ .

### 4.1.2 Baseline

Le second modèle comparatif, nommé **baseline** par la suite, est un modèle à 3 facteurs que nous avons implémenté selon la description donnée dans [23], et issu du fort mouvement de recherche qu'a provoqué le grand prix Netflix en 2006. Nombre d'équipes ayant participé à ce concours de conception d'algorithmes de recommandation ont utilisé ce même baseline. Sa composition vient de l'observation de la présence de biais dans les notations : certain usagers notent en moyenne plus favorablement que d'autres, et certains articles sont mieux notés que d'autres en moyenne. Il en découle que la prédiction d'un score $u_{c,s}$ pour l'article $s$ par l'usager $c$ est obtenue par la formule

$$\begin{aligned} u_{c,s} &= \mu + b_c + b_s, \\ b_c &= -\mu + \frac{1}{|U_c|} \sum u_c, \\ b_s &= -\mu + \frac{1}{|U_s|} \sum u_s, \end{aligned}$$

où $\mu$ est la note moyenne dans tout l'ensemble de données, $b_c$ est la déviation de la note moyenne de l'usager $c$ par rapport $\mu$, et $b_s$ est la déviation de la note moyenne de l'article $s$ par rapport $\mu$. Ce modèle est une bonne base pour évaluer les modèles, car il ne tient pas compte de l'**interaction** entre utilisateur et article, c'est-à-dire ce qui fait l'essence même d'un recommandeur de qualité supérieure. Il ignore les raisons pour lesquelles *cet* utilisateur va plus ou moins apprécier *cet* article.

### 4.1.3 Baseline avancé

Le modèle SDV (provenant du terme anglais *Singular Value Decomposition*) est une évolution du baseline utilisant des paramètres (*latents factors*) pour modéliser $b_c$ et $b_s$, ainsi que les interactions de chaque utilisateur avec chaque article. Ce modèle aussi a été implémenté par nos soins. À chaque usager $c$ est donc associé un vecteur $p_c \in \mathbb{R}^k$, et de la même façon à chaque article $s$ un vecteur $q_s \in \mathbb{R}^k$, pour $k \in \mathbb{N}^*$ influant sur la capacité du modèle.

La formule d'inférence est la somme du baseline et du produit scalaire de $q_s$ et $p_c$ :

$$t_{c,s} = \mu + b_c + b_s + q_s^T p_c,$$

et des valeurs optimales des paramètres sont trouvées par descente de gradient sur la somme de l'erreur quadratique et d'un terme de régularisation, pour chaque paramètre.

Par exemple pour $b_c^*$, la valeur optimale du biais usager $b_c$, la formule est la suivante :

$$b_c^* = \underset{b_c}{\operatorname{argmin}} \sum_{c,s} (u_{c,s} - t_{c,s})^2 + \lambda(||p_c||^2 + ||q_s||^2 + b_c^2 + b_s^2)$$

où $\lambda$ est un hyper-paramètre, $u_{c,s}$ est la cible (la note) et $tc, s$ la prédiction (le score).

La référence à la décomposition en valeurs singulières vient de l'utilisation de celle-ci dans le domaine de l'extraction d'information. La SVD est un procédé mathématique permettant de représenter les vecteurs d'une matrice selon une nouvelle base dont on peut trier les dimensions en ordre de variances décroissantes afin d'obtenir de nouvelles représentations de ces vecteurs, plus compactes que les originaux, et dont les dimensions peuvent être décorrélées. On pourra se référer à [14] pour une explication formelle et un exemple d'application de la SVD à l'analyse sémantique latente. Ici, la SVD n'est pas faite explicitement mais *via* l'optimisation des paramètres pour les paires $(c, s)$ présentes dans $\mathscr{D}_{train}$ .

## 4.2 Réseau profond standard, SDN

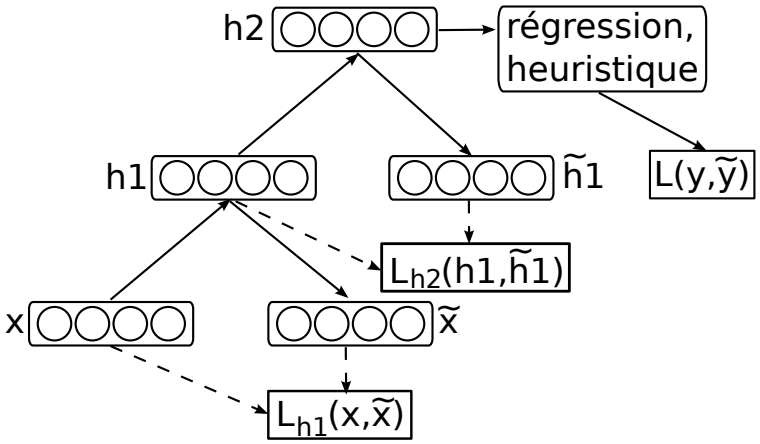
Notre première tentative pour tester la performance d'architectures profondes pour la recommandation a été de concevoir un modèle assez standard, à savoir un réseau de neurones profond à une ou plusieurs couches (*multi layer perceptron*) surmonté d'une couche de régression logistique ayant autant de sorties qu'il y a de scores possibles, chacune représentant la probabilité associée à un score. Nous avons implémenté les différentes variations de cette architecture présentées dans ce mémoire.

La figure 4.1 illustre cette architecture.

On a vu que la nature du pré-entraînement des réseaux profonds place les paramètres dans une meilleure configuration pour la phase de *fine-tuning* (2.3.2). Afin de vérifier ce résultat, on a entraîné ce modèle selon l'algorithme de descente de gradient, sans pré-entraînement et selon les trois méthodes de pré-entraînement mentionnées en 2.3.2. Ce modèle sera nommé SDN$X_Y$, où $X \in \{1, 2, 3\}$ est le nombre de couches cachées, et $Y \in \{\varnothing, AE, DAE, CAE\}$ correspond au type de pré-entraînement effectué pour chaque couche, respectivement pas de pré-entraînement, pré-entraînement comme des auto-encodeurs classiques, débruitants ou contractants. Les résultats d'aucun modèle mixte (composés de couches entraînées de façons différentes) ne sont rapportés dans ce document.

L'entrée du modèle est un vecteur de valeurs réelles composé de la concaténation des attributs uniformisés du joueur et de ceux du niveau dont on cherche à prédire la note.

Figure 4.1 – Architecture d'un Standard Deep Network




Un modèle à 2 couches est ici présenté. La première et la seconde prennent respectivement $x$ et $h_1$ en entrée, produisent respectivement les représentations cachées $h_1$ et $h_2$, et les reconstructions $\widetilde{x}$ et $\widetilde{h_1}$. $L_1(x, \widetilde{x})$ et $L_1(h_1, \widetilde{h_1})$ sont les fonctions de pertes que ces couches sont entraînées à minimiser. La représentation cachée de la couche la plus haute est ensuite fournie en entrée à la couche de régression logistique, puis l'heuristique de calcul de score 4.1 produit une prédiction qui sert au critère d'entraînement au *fine-tuning* $L(y, \widetilde{y})$.

La première couche prend donc un mini-lot de $minibatch\_size$ vecteurs de $\mathscr{D}_{train}$ , passe sa propagation-avant dans une tangente hyperbolique (2.2.2) pour produire une matrice de représentations cachées puis reconstruit le mini-lot. L'erreur quadratique entre cette reconstruction et le mini-lot est mesurée, et chaque paramètre du réseau est mis à jour selon une fraction de la dérivée partielle de l'erreur par rapport à ce paramètre. Les iterations sont répétées jusqu'à satisfaction du critère d'arrêt, puis les représentations cachées de cette couche sont utilisées comme entrées pour entraîner la couche supérieure selon le même algorithme.

La représentation de la couche cachée de plus haut niveau sert à l'entraînement d'une dernière couche de régression logistique dont la sortie subit une normalisation par une softmax (2.2.2) afin de pouvoir être interprétée comme un vecteur des probabilités des notes possibles $p_v(u = v|x)$. La prédiction d'un score $\hat{u}$ dans l'intervalle $[0, 1]$ se fait selon la formule suivante :

$$\hat{u} = \sum_{v \in V} p_v(x) v \tag{4.1}$$

Enfin, cette dernière couche est entraînée par minimisation de la log-vraisemblance de la note

$u^*$, dont la mise à jour du paramètre $\theta_i$ pour l'itération $t+1$ suit la formule

$$\theta_{i,t+1} = \theta_{i+1} - \frac{\partial - \log(p_{u^*})}{\partial \theta_{i,t}}$$

La formule 4.1 convient mieux que de simplement selectionner le score de plus haute probabilité $\hat{u} = \underset{v \in V}{\operatorname{argmax}}\ p_v$ car elle prend toutes les probabilités en compte ; c'est une moyenne pondérée et souffre donc moins de l'impact d'une possible valeur dégénérée.

Bien qu'on ne puisse considérer cette architecture comme originale, nous n'avons pas trouvé dans la littérature beaucoup d'exemples d'utilisation d'*ADAGRAD*pour l'entraînement de tels modèles.

## 4.3 Réseau profond à prédiction basée sur son critère d'erreur, EDN

Une seconde approche a été envisagée, dans laquelle la prédiction est calculée selon une heuristique similaire, c'est-à-dire basée sur un vecteur de probabilités, mais c'est la façon dont ce vecteur est obtenu qui diffère.

Cette approche s'inspire des modèles à énergie (*Energy Based Models*, ou EBM, [25]) qui apprennent à minimiser en $(x, y)$ une fonction d'énergie $f(x, y)$ paramétrisée par $\theta$, où $x$ est un vecteur d'attributs et $y$ la cible correspondante, et à la maximiser ailleurs. L'inférence est faite pour un $x$ donné, en déterminant quel $\hat{y}$ retourne la plus faible énergie :

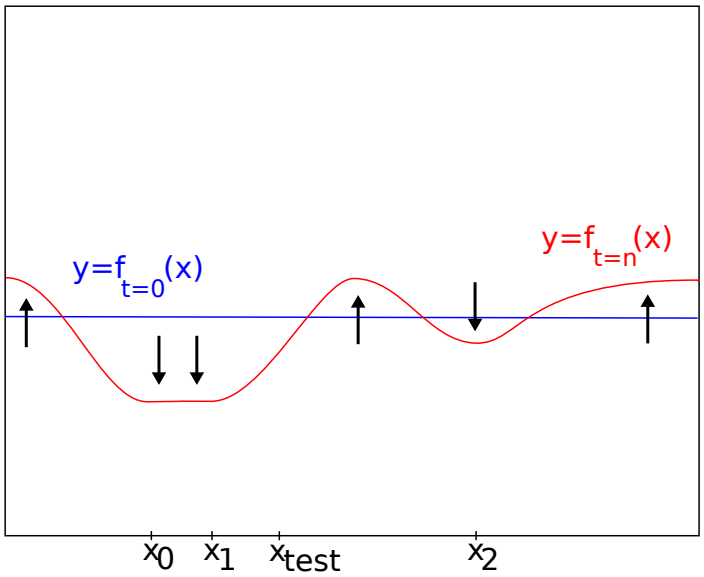
$$\hat{y} = \underset{y \in Y}{\operatorname{argmin}} f(x, y),$$

où $Y$ représente l'ensemble des cibles. Lorsque $|Y|$ est très grand, dans le cas où $y$ est continu par exemple, une heuristique doit être utilisée, dépendamment de la structure interne du modèle.

On peut visualiser très simplement le principe derrière les modèles à énergie en voyant la fonction d'énergie $f(x, y) = z$ comme une surface approximativement plane en trois dimensions $x$, $y$ et $z$. L'entraînement consiste à passer en revue des positions $(x, y)$ et y diminuer la valeur $z$ de $f$ sur chacune d'entre elles, et probablement dans un certain périmètre autour aussi. On voudra aussi augmenter la valeur de $f$ là où l'on n'a pas croisé de $(x, y)$. Enfin, on saura quelle cible $y$ est associée à quel $x$ en parcourant $f$ en $x$, le long de l'axe $y$, et en choisissant le $y$ pour lequel $z$ est minimal. La figure 4.2 illustre ceci.

Revenons à nos réseaux. On sait que les réseaux de neurones auto-associatifs et leurs variantes apprennent à reconstruire leurs entrées en minimisant un critère de similarité entre ces

Figure 4.2 – Principe d'entraînement d'un EBM




Ce diagramme présente une fonction d'énergie en deux dimensions, renvoyant une énergie $y$ associée à une entrée $x$. Avant l'entraînement $f_{t=0}(x)$ est plane (en réalité elle serait davantage bruitée). L'entraînement consiste à pousser $f$ vers le bas dans les zones de forte densité, c'est-à-dire là où se trouvent les $x_i$, afin de modéliser la distribution sous-jacente à $\mathscr{D}_{train}$ . Après $n$ itérations, on obtient $f_{t=n}(x)$, qui permet de répondre à des requêtes sur la probabilité d'un $x_{test} \in \mathscr{D}_{test}$.

entrées et la reprojection de leur représentation cachée. Ce faisant, ces modèles apprennent à minimiser leur erreur pour les exemples les plus probables. Au cours de leur entraînement, ces modèles apprennent à modéliser un ensemble de dépendances propres à ces données. On conçoit qu'en laissant un modèle apprendre à modéliser les exemples contenus dans $\mathscr{D}_{train}$ , il parviendra à avoir une erreur de reconstruction faible pour un point *typique* (facile) de $\mathscr{D}_{train}$ , mais serait moins en mesure de représenter un point *atypique* (mettons aléatoire, pour l'exemple), car ce dernier point serait dans une zone de densité faible dans l'espace d'entrée. L'analogie avec les EBM est très simple. L'erreur de reconstruction sera plus faible pour des entrées probables, on pourra donc la considérer comme notre fonction d'énergie. D'autre part les notes possibles qu'un joueur peut donner à un niveau appartiennent à un ensemble fini de valeurs, donc on peut facilement *parcourir l'axe des y* en testant la qualité de reconstruction de l'entrée pour **toutes** les valeurs de score possible.

Concrètement, la prédiction d'une note est calculée selon une heuristique sur les erreurs de reconstruction $l_v$ pour chacun des scores possibles $v \in V$ de la manière suivante : le vecteur $l$ contient en $l_v$ l'erreur de reconstruction $L(f, z_v)$, avec $z_v$ la concaténation des at-

tributs du joueur, de ceux du niveau et du score $v$ [1]. Ensuite $l$ est normalisé selon la même normalisation qu'en 2.2.2 pour en faire un vecteur de probabilités $p_v$, alors utilisé comme en 4.1 pour calculer la prédiction $\hat{u}$ :

$$\begin{aligned} l_v &= L(f, z_v), \forall v \in V, \\ p_v &= \frac{e^{-l_v}}{\sum_{v'=0}^{4} e^{-l_{v'}}}, \\ \hat{u} &= \sum_{v \in V} p_v(x) v. \end{aligned}$$

En général, un modèle basé sur une fonction d'énergie apprend non-seulement à minimiser son energie dans les zones de de forte densité dans $\mathscr{D}$ , mais aussi à l'augmenter ailleurs. Ce second aspect de l'entraînement est aussi encouragé dans notre version de modèle à énergie par l'utilisation de la log-vraisemblance négative en tant que critère d'entraînement : minimiser un $p_u$ revient a maximiser $p_v$ pour $v \in V \setminus u$.

Un aspect des modèles à energie habituels qui n'est pas représenté ici est ce que [25] nomme l'erreur la plus dommageable $\check{\mathrm{v}}$ (*most offending incorrect answer*), qui correspond dans notre cadre au score $v$ différent de la note $u$ pour lequel l'énergie est la plus basse. Dans le cas discret, on peut trouver ce score facilement :

$$\check{\mathrm{v}} = \operatorname*{argmin}_{v \in V \setminus u} l_v.$$

Dans le cas continu, on peut considérer que les scores dans un intervalle avec la note sont corrects ; l'erreur la plus dommageable devient alors le score le plus proche de cette marge autour de la note :

$$\check{\mathrm{v}} = \operatorname*{argmin}_{v \in V \setminus u, ||v-u||^2 > \epsilon} l_v.$$

En général, on essaiera d'augmenter l'énergie du modèle pour ce score, de façon à s'assurer que seul le bon score lève une basse énergie et donc que la prédiction soit moins biaisée. Ici, nous comptons sur la minimisation de la log-vraisemblance négative pour faire monter l'énergie des mauvais scores, y compris le plus dommageable, mais il serait intéressant de prévoir une maximisation plus explicite de l'énergie du score le plus dommageable, au moyen d'un critère d'erreur composite par exemple.

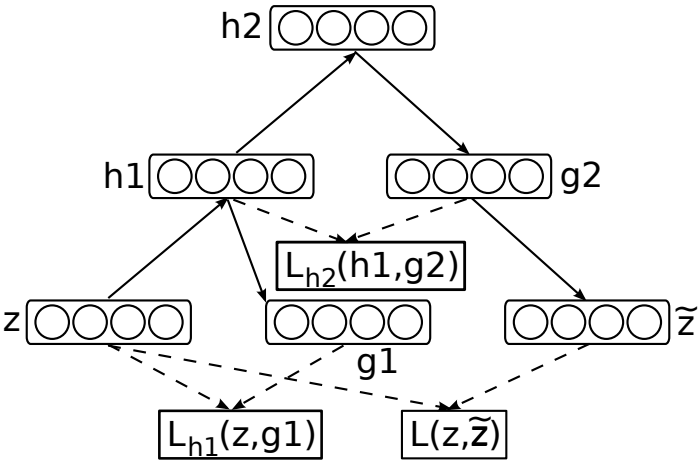
On entraînera successivement des empilements de 1, 2 ou 3 couches, dont les activa-

1. Certains attributs sont des statistiques agrégées sur plusieurs joueurs et/ou plusieurs niveaux. L'architecture n'est pas collaborative au delà de ça.

tions sont des *tanh*, de façon non-supervisée selon les trois styles d'auto-encodeur : simple, débruitant et contractant.

L'erreur de reconstruction utilisée pour les prédictions est celle de la première couche, comme le montre la figure 4.3. C'est-à-dire que le modèle suit d'abord une phase d'encodages successifs jusqu'à la couche de plus haut niveau, puis la reconstruction de cette dernière est utilisée comme entrée pour la phase de décodages successifs jusqu'à la reconstruction produite par la première couche du modèle, représentant l'entrée $x$. Quelques expériences préliminaires où l'erreur des toutes les couches était prise en compte n'avant pas montré de gain significatif, nous n'avons pas d'avantage exploré l'utilisation de ces erreurs. Cette voie reste explorable dans des expériences ultérieures.

Figure 4.3 – Architecture EDN




Un modèle à 2 couches est présenté ici. $z$ et $h_1$ sont les entrées des première et seconde couches respectivement. $h_1$ et $h_2$ sont leurs représentations cachées, $g_1$ et $g_2$ leurs reconstructions. $\widetilde{z}$ est la reconstruction de la première couche en utilisant $g_2$ en guise de représentation cachée $h_1$. Lors du pré-entraînement, la première couche apprend à reconstruire $z$ en minimisant l'erreur $L_{h1}(z, g_1)$. La seconde couche apprend à minimiser $L_{h2}(h_1, g_2)$. La perte $L(z, \widetilde{z})$ est utilisée pour calculer $p_t$ pour $t$ le score inclus dans $z$.

Dans ce modèle, la cible qui fait partie de $z$ est représentée par un vecteur *one-hot*, c'est-à-dire un vecteur binaire nul dont l'indice correspondant à la valeur qu'on veut lui faire représenter est incrémenté [2]. Ce pré-traitement de la note est une décomposition de l'information qu'elle contient. Ainsi au lieu d'exiger du modèle qu'il apprenne à modéliser et reconstruire un scalaire réel et les nuances de ses valeurs (ce scalaire répondrait à la question «à quel point cet article a-t-il été apprécié ?»), on lui fournit plus de scalaires, mais chacun

2. Par exemple, pour $x \in \{0, 1, 2\}$ le vecteur $[0, 0, 1]$, représente l'encodage *one-hot* de $x = 2$. Dans les mêmes conditions, $one\_hot(x = 0) = [1, 0, 0]$ et $one\_hot(x = 2) = [0, 0, 1]$.

répondant à une question plus simple («est-ce que cet article a-t-il été apprécié à *ce* point là», *ce* étant propre à chaque scalaire). Sans modifier le sens de l'entrée, on peut espérer que cette représentation *sparse* aide le modèle dans sa tâche d'apprentissage.

Du point de vue du modèle, aucune distinction n'est faite entre cible et entrée, et l'apprentissage semble non-supervisé. Cependant, la définition de l'apprentissage supervisé (1.2) mentionne l'**utilisation** d'étiquettes, pas la façon dont elles sont utilisées. Comme elles sont ici indispensables à l'entraînement et explicitement présentes dans chaque vecteur d'entrée on est définitivement dans un contexte supervisé, bien que non-discriminant (c'est-à-dire que l'on modélise $P(x, y) = P(z)$ plutôt que $P(y|x)$ comme dans un réseau de neurones standard).

Comme la prédiction ne se base que sur les pré-entraînement successifs des différentes couches, aucune étape de *fine-tuning* n'est faite. On appellera ce modèle utilisant l'erreur de reconstruction pour faire ses prédictions EDN$\{1, 2, 3\}_{\{AE,DAE,CAE\}}$ pour *Error-based Deep Network* ou *Energy-based Deep Network*, selon son nombre de couches et la nature du pré-entraînement utilisé. Cette série d'expériences vise à établir une base de performance pour EDN, afin de permettre une appréciation concrète des modifications envisagées.

Nous avons donc ensuite essayé quelques variations mettant à profit la phase encore non-exploitée de *fine-tuning*. Le but de ces variations est d'améliorer les performances du meilleur EDN. On a donc restreint l'exploration d'hyper-paramètres aux nouveaux hyper-paramètres utilisés dans ces variations, à l'exception de la durée d'entraînement, et utilisé pour les autres hyper-paramètres les valeurs obtenues après la selection de modèle pour EDN. Les deux sous-sections suivantes décrivent ces variations.

### 4.3.1 Avec *fine-tuning* supervisé

Dans un premier temps nous avons procédé à une optimisation globale du modèle par descente de gradient sur la log-vraisemblance négative du score $v$ correspondant à la note $u$, comme pour le *fine-tuning* de SDN (4.2) :

$$L(f_\theta, z) = -\log(p_u)$$

En maximisant $p_u$ et donc $l_u$, diminue $l_v$ pour tout $v \in V \setminus u$.

Nous nommerons cette variante EDN$_{\text{CAE+ft}}$ et EDN3$_{\varnothing\text{+ft}}$ , selon qu'il ait suivi ou non une phase de pré-entraînement non-suprvisée. Ce modèle a été hyper-optimisé selon la même méthodologie que précédemment, à partir de EDN3$_{\text{CAE}}$ :

- exploration des hyper-paramètres. Une unique série de 500 expériences a été menée en raison du faible nombre d'hyper-paramètres,
- sélection du modèle à partir de la meilleur performance $top_{k=1}$ sur $\mathscr{D}_{valid}$ ,
- rapport des aggrégations de ses MSE et $top_{k=1}$ sur 10 expériences finales.

Seuls le taux d'apprentissage de départ et le nombre d'époques de *fine-tuning* optimales demandaient optimisation. L'heuristique *ADAGRAD* a été utilisée dans la série complète d'entraînements pour déterminer le taux d'apprentissage.

### 4.3.2 Avec *fine-tuning* semi-supervisé

Un auto-associateur étant assez proche d'une RBM dans son fonctionnement, nous avons essayé d'appliquer une optimisation d'entraînement décrite dans [24]. Il s'agit lors de la phase de *fine-tuning* d'ajouter au critère **discriminant** supervisé le critère **génératif** non-supervisé utilisé dans la phase pré-entraînement. C'est une méthode de régularisation du *fine-tuning* qui cherche à compenser sa voracité possiblement trop grande. Comme on entraîne le modèle en n'utilisant les cibles des points de $\mathscr{D}_{train}$ que dans *certaines* composantes du critère d'entraînement, on parlera d'entraînement **semi-supervisé**, et le modèle sera appelé $\text{EDN}_{\text{CAE+fth}}$ pour $\text{EDN}_{\text{CAE}}$ à *fine-tuning* hybride.

Cette régularisation est du même type que le pré-entraînement, dans le sens où elle ne limite pas la taille de la zone de $\mathscr{F}$ explorable durant le nombre d'itérations imparti, mais participe plutôt à **la sélection** des fonctions qui en feront partie. Elle pousse notre recherche vers le domaine de l'apprentissage multi-tâches (*multi-tasking*[11, 36]) qui se base lui-aussi sur l'hypothèse semi-supervisée (2.3.2) : certaines caractéristiques d'une distribution peuvent être utiles à l'accomplissement de plusieurs tâches.

Concrètement, la fonction de coût est la somme pondérée de la NLL et de la MSE :

$$\begin{aligned} L(f_\theta, z_t) &= \alpha L_{mse} + (1-\alpha) L_{nll} \\ L_{nll} &= -\log(p_{t=r}) \\ L_{mse} &= \frac{1}{d}\sum_{i=1}^{d}(\widetilde{x_i} - x_i)^2 \end{aligned} \tag{4.2}$$

où $\alpha$ est un coefficient hyper-optimisé dans l'intervalle $[10^{-6}, 1]$ et échantilloné sur une échelle logarithmique (décrite en 4.4.1), $d$ la dimension de $x$, et $\widetilde{x}$ la reconstruction de $x$ par la couche la plus basse après propagation aller-retour dans les diverses couches.

Par ces deux critères, on demande à la fonction d'énergie non-seulement d'être sensi-

ble au score présentement associé aux attributs, mais aussi à bien modéliser les principales caractéristiques de ces attributs.

## 4.4 Cadre expérimental

Cette section présente les conditions dans lesquelles les modèles étudiés dans ce mémoire ont été entraînés, la méthodologie de selection des hyper-paramètres omis jusqu'ici, et la procédure d'évaluation finale des modèles. Ces conditions sont les mêmes pour les différentes variations de modèles considérées, sauf mention contraire.

### 4.4.1 Hyper-optimisation

**L'hyper-optimisation** est la procédure de choix d'hyper-paramètres optimaux pour un modèle. Elle a ici été faite par échantillonage aléatoire (*random sampling*) des hyper-paramètres, échantillonage logarithmique pour certains, dans des intervalles pertinents mais arbitraires. Le tirage aléatoire logarithmique consiste, pour un intervalle $[a, b]$, à choisir la valeur $exp(v)$, pour $v$ tiré selon une loi uniforme dans $[\log a, \log b]$. Cette méthode de tirage a l'avantage de mieux explorer les différents ordres de valeurs dans l'intervalle considéré. Elle est utilisée pour échantiller le taux d'apprentissage ainsi que la pénalité de contraction des CAE (entre $10^{-5}$ et $0.5$). Les autres hyper-paramètres optimisés par échantillonage aléatoire sont :

- les tailles des différentes couches cachées : entre 50 et 500
- l'époque à partir de laquelle le taux d'apprentissage subit un recuit simulé (1.7) : après qu'entre 50% et 95% de l'entraînement se soit déroulé
- le nombre de couches du modèle : 1, 2 ou 3

Chaque échantillon est différent à chaque tirage, ce qui maximise l'exploration de l'intervalle de valeurs considéré, en comparaison avec d'autres stratégies d'hyper-optimisation comme la recherche sur grille par exemple (*grid search*), qui consiste à prendre $n$ valeurs, à intervalle régulier ou-non, dans la plage de recherche pour chacun des $n_{hparams}$ hyper-paramètres, puis à faire des expériences avec un tirage complet ou aléatoire des configurations dans cette grille à $n_{hparams}$ dimensions. Des explications plus détaillées sur l'avantage de l'exploration d'une grille aléatoire peuvent être trouvées dans [8**?** ].

Une première phase d'exploration est effectuée au moyen de 500 expériences dans des intervalles de valeurs plausibles mais assez larges afin de maximiser le volume des configurations couvertes, puis une seconde phase avec 500 nouvelles expériences et des intervalles centrés sur les valeurs ayant abouti aux meilleurs modèles lors de la première phase. Les

modèles sont ordonnés selon leur performance $top_{k=1}$ sur $\mathscr{D}_{valid}$ . À la suite de cette seconde phase, le meilleur ensemble d'hyper-paramètres est utilisé dans 10 dernières expériences identiques sauf au niveau de la génération des nombres aléatoires : les intialisations des poids, et le mélange des exemples de l'ensemble de données sont différents d'une expérience à l'autre. Enfin, de ces 10 expériences aux hyper-paramètres identiques sont aggrégées la moyenne $\mu$ et l'écart-type $\sigma$ sur le critère d'entraînement du modèle ainsi que sur sa performance. Ces deux aggrégations sont calculées sur $\mathscr{D}_{test}$ afin de relayer la performance de généralisation.

On rapportera à la fois ces valeurs pour le critère d'entraînement MSE et pour la métrique $top_{k=1}$ afin de constater leur relation. Comme expliqué précédemment (3.6), le critère optimisé est une mesure mandataire de la performance «réelle», et ne sera pas forcément corrélée à la performance finale du modèle, du fait du domaine d'application. La valeur de $k$ est choisie la plus basse possible afin de compenser la pauvreté des données (le nombre de notes médian dans notre jeu est de 10 par utilisateur).

Dans ce mémoire, la granularité discriminant deux modèles est la suivante : $\text{SDN}_{\text{AE}}$ , $\text{SDN}_{\text{DAE}}$ et $\text{SDN}_{\text{CAE}}$ seront considérés comme trois modèles différents avec pour chacun une exploration d'hyper-paramètres telle que décrite précédemment.

Dans la motié des expériences, le taux d'apprentissage est constant pendant une partie de l'entraînement puis, à partir d'un délai d'entraînement $t_{annealing}$, il diminue selon la technique de recuit mentionnée dans l'introduction (1.7). $t_{annealing} = pm|\mathscr{D}_{train}|$ où $m$ est le nombre d'itérations maximum, et $p \in [0, 1]$ un coefficient hyper-optimisé. Il s'agit donc de commencer le recuit après que le modèle ait vu $t_{annealing}$ exemples d'entraînement. Dans l'autre moité des expériences, le taux d'apprentissage est calculé avec *ADAGRAD* (1.7) et ne nécessite pas d'hyper-paramètre supplémentaire.

### 4.4.2 Entraînement

L'ensemble de données $\mathscr{D}$ est d'abord divisé en trois sous-ensembles $\mathscr{D}_{train}$ , $\mathscr{D}_{valid}$ et $\mathscr{D}_{test}$ disjoints, selon les proportions respectives $[0.6, 0.2, 0.2]$. L'optimisation des paramètres par descente de gradient est faite sur $\mathscr{D}_{train}$ pour un nombre d'époques $n_{epochs}$borné dans $[10, 500]$. Une durée de pré-entraînement minimum de 10 époques est imposée afin de palier au fait que l'amélioration de l'erreur sur $\mathscr{D}_{train}$ ne signifie pas forcément la même amélioration sur $\mathscr{D}_{valid}$ , et comme c'est cette erreur sur $\mathscr{D}_{valid}$ qui détermine l'arrêt de l'expérience il est préférable d'ignorer ses premières valeurs. Après quelques époques, le modèle commence

à capter la distribution des données et on voit l'erreur sur $\mathscr{D}_{valid}$ baisser [3]. La borne supérieure sur la durée d'entraînement n'a qu'une justification computationnelle, et ne sert que dans les configurations d'hyper-paramètres où le taux d'apprentissage est très faible et ralentit considérablement l'entraînement. En plus de cette borne supérieure, on utilise un critère d'arrêt prématuré (1.6) défini comme un seuil sur la moyenne des ratios d'amélioration de l'erreur sur $\mathscr{D}_{valid}$ dans une fenêtre des 15 dernières epoques. Comme on ne cherche pas forcément l'erreur de reconstruction minimale mais juste la garantie que la représentation cachée porte une information pertinente, on économisera quelques époques à faible rendement.

### 4.4.3 Implémentation

L'intégralité des résultats figurant dans ce mémoire provient d'expériences menées par nos soins sur le cluster du LISA/DIRO, au moyen d'une pipeline d'entraînement que nous avons créée nous-même, et qui entraîne nos propres implémentations des modèles présentés.

Les modèles comparatifs aléatoire, baseline et baseline doivent leur conception à des auteurs de la littérature. Nous avons conçu l'architecture SDN nous-même, mais elle est tellement classique qu'on peut très probablement la retrouver dans nombre d'articles. L'architecture EDN en revanche, bien qu'inspirée des modèles à énergie, est nouvelle, dans le sens où l'on n'a su trouver de précédents dans la littérature mentionnant des réseaux profonds avec pré-entraînement non-supervisé utilisés comme des modèle à énergie (sans-même parler d'une application à une tâche de recommendation). Nous l'avons donc implémentée nous-même.

Les libraries Theano [9] et Pylearn2 ont été particulièrement utiles, puisqu'elles automatisent certains aspects d'une implémentation, tels que le calcul de dérivées pour la descente de gradient, ou fournissent des primitives d'apprentissage automatique de plus haut niveau, du type de celles décrites au chapitre 2 (l'auto-encodeur étant une de ces primitives, par exemple). Dans le cas de Pylearn2, nous avons contribué à l'utilisation automatisée d'*ADAGRAD* dans l'optimisation par descente de gradient.

## 4.5 Ensembles de données

Les ensembles de données ne manquent pas dans le domaine de la recommandation [18, 37], mais leurs caractéristiques nuancées ne les rendent pas tous directement utilisables dans le contexte de recherche de ce projet. En effet, les identifiants d'utilisateurs et d'articles, et

3. On pourra se référer à la figure 1.2 page 11 pour visualiser ce phénomène.

leur notes associées sont souvent mis à disposition, mais il est beaucoup plus rare de trouver de larges ensembles de données contenant en plus des informations numériques sur les profils de ces utilisateurs et articles. Au mieux, des étiquettes (*tags*) leur sont associées, mais celles-ci se prettent mal à une numérisation et l'on doit les encoder via des vecteurs binaires.

### 4.5.1 Ensemble Mighty Quest

Cet ensemble de données est issu du jeu *Mighty Quest For Epic Loot* (en phase de développement) dans le cadre de la *chaire industrielle CRSNG-Ubisoft en apprentissage de représentations pour les jeux vidéo immersifs*. Dans ce jeu UGC [4] en ligne, le joueur joue successivement sur des niveaux créés soit par d'autres joueurs, soit par le studio de développement.

Après complétion d'un niveau créé par un autre joueur, il lui est proposé *via* **un sondage** optionnel de rapporter à quel point il a apprécié son expérience, au moyen d'une échelle graduée de un à cinq représentée dans la figure 4.4 sur laquelle il a le choix de cliquer. Cette évaluation optionnelle, dont la valeur est réduite dans l'intervalle $[0, 1]$, fait office de note pour le niveau.

Cette méthodologie n'est pas parfaite pour plusieurs raisons. De façon générale, on sait [32] que les utilisateurs d'un système de recommandation ne sont pas cohérents dans leur notation au fil du temps car leurs références de notation et leurs préférences changent. Plus spécifiquement dans cet ensemble de données, seuls les utilisateurs ayant **terminé** un niveau peuvent le noter. Il en découle un possible biais qui augmente artificiellement la notation. Par exemple un joueur qui perd continuellement et qui, pour une fois, réussit à terminer le niveau pourra soudainement se sentir enclin à donner une bonne note à ce niveau, non pas parce qu'il a été agréable à jouer mais parce qu'il était plus facile que les précédents. Il y a en effet une distinction réelle à faire entre le plaisir à jouer un niveau et sa difficulté. Cette corrélation entre difficulté et plaisir, propre à chaque joueur, est la raison pour laquelle le sondage n'est pas proposé à un joueur échouant un niveau, mais elle biaise les notations dans l'autre sens pour ceux chez qui elle est élevée.

Chaque note de $\mathscr{D}$ est accompagnée de statistiques sur le joueur et sur le niveau, appelées **attributs**. Le design des attributs à collecter sur les joueurs ainsi que les niveaux ayant été la phase préliminaire de ce mémoire, un bref mot sera placé à leur sujet. Les attributs correspondent globalement à des informations disponibles avant la partie et pouvant influencer la

4. *User Generated Content* pour rappel.

Figure 4.4 – Capture d'écran du jeu *Mighty Quest*


Cette page est affichée après complétion d'un niveau et a pour but de présenter au joueur les gains qu'il a amassés. Le joueur peut attribuer une note au niveau au moyen de la barre graduée d'étoiles située en bas à gauche. Afin de l'inciter à faire de même, l'écran précise qu'un prix surprise lui sera donné s'il note le niveau.

note finale. Du côté du joueur, on cherchera à modéliser les facteurs de complétion du niveau (pour prendre en compte la corrélation plaisir/difficulté), tels que l'équipement de son avatar, ses capacités spéciales, ses sorts, etc. Pour le niveau, les mêmes facteurs se matérialisent par son contenu, c'est-à-dire les nombre, qualité et agencement des créatures et pièges qui le composent. La composition des niveaux du jeu cadre de cette recherche étant assez récursive, beaucoup de statistiques se trouvent présentes dans les attributs sous plusieurs formes d'aggrégation (moyenne, variance, maximum, minimum).

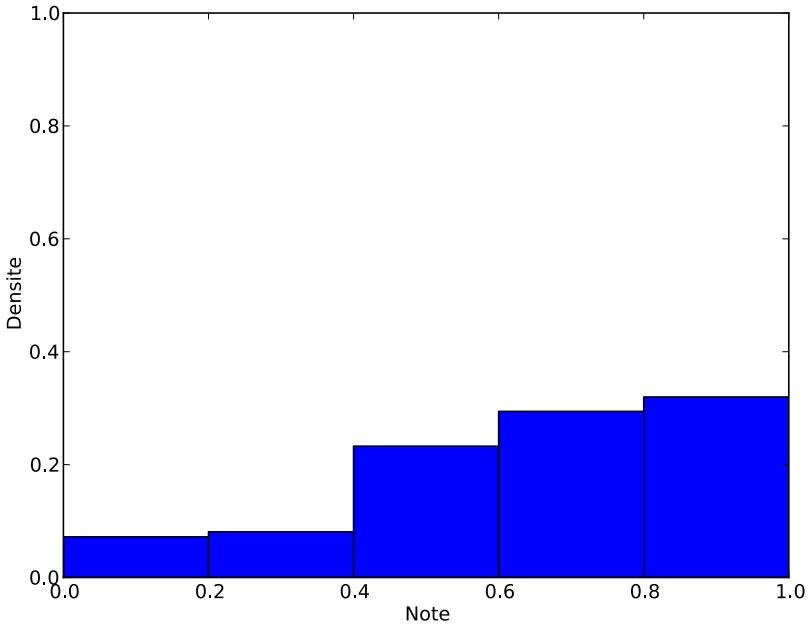
Cet ensemble de données a la caractéristique particulière de contenir des attributs de joueurs et de niveaux différents pour quasiment toutes les notes. Ceci est dû à la capacité donnée aux joueurs de modifier l'équipement de leur avatar entre deux niveaux, ainsi que de modifier le niveau dont ils sont responsable à tout moment. La notion d'identité d'un joueur est ainsi implicite dans les attributs de son avatar au fil de sa notation, car son profil change en permanence.

Au moment où ces lignes sont écrites, cet ensemble de données que l'on nommera *mq* a une taille de l'ordre de $10\,000 \times 187$ [5], ce qui représente assez peu d'exemples à la fois dans

5. Les 187 dimensions de l'entrée sont des moyennes, minima, maxima et variances de divers attributs de

le cadre de l'entraînement de réseaux de neurones, et dans celui de la recommandation en général. Il est important de noter que les notes qu'il contient sont assez fortement biaisées positivement comme le montre la figure 4.5 : la moyenne des notes réduites dans $[0, 1]$ est de 0.68. De plus, la mesure $top^*_{k=1}$ (la moyenne de la meilleure note de chaque joueur) est de 0.763. Une explication plausible d'une telle répartition des notes vient de l'état de développement du jeu d'où l'ensemble de données est extrait. Des joueurs qui s'essayent au jeu ont probablement un *a priori* positif puisqu'ils profitent en avant-première d'un produit non-fini développé par des proches (le jeu était alors en phase de *Friends & Family*). Une partie d'entre eux va *essayer* le jeu peu fréquemment et produira en conséquence un faible nombre de notes, et une autre va y jouer de façon intensive et aura une répartition de note possiblement plus large. On pourra se référer à [37] pour une étude des répartitions habituelles des notes dans les ensembles de données de recommandation.

Figure 4.5 – Distribution des notes de l'ensemble Hyperquest




On remarque aisément à quel point la distribution est biaisée favorablement pour les niveaux.

Toujours est-il que la marge de progression pour l'élaboration d'algorithmes plus performants n'est pas grande. On est en droit de se demander à quel point un joueur peut être sensible à l'amélioration de sa note moyenne, mais tenter d'y répondre rigoureusement pourrait

joueurs et de niveaux

être sujet à une étude à part entière et requerrait probablement divers sondages de populations. Bien que pertinente, cette question restera donc sans réponse.

# CHAPITRE 5

# RÉSULTATS EXPÉRIMENTAUX

Ce chapitre présente les résultats des différentes variantes des modèles décrits au chapître précédent, et propose une analyse des modèles et métriques employés. Les résultats sont décrits dans l'ordre où les modèles ont été présentés.

## 5.1 SDN

Les résultats sont présentés dans la table 5.1. La prédiction aléatoire ayant un score $top_{k=1}$ supérieur à 50% du meilleur score s'explique d'une part par le biais des notes (leur moyenne est supérieure à 0.5), et d'autre part par le fait que le meilleur score $top_k^*$ lui-même n'est pas 1 (il est d'environ 0.83 dans l'ensemble *hq*).

On peut constater que tout comme la littérature l'avertit, le critère d'entraînement et la métrique de rang ne sont pas corrélés. Ceci se voit moins au niveau des modèles étudiés que des modèles de comparaison, où SVD montre une mauvaise généralisation. Il ressort en effet que SVD semble beaucoup sur-apprendre $\mathscr{D}_{train}$ , puisqu'il y obtient une MSE dix fois moins que sa version sans facteurs latents *baseline* sans pour autant s'en démarquer au niveau $top_k$ . SVD performe au mieux sur $\mathscr{D}_{valid}$ avec un coefficient de régularisation $\lambda = 4.2 \times 10^{-06}$, un taux d'apprentissage de $54 \times 10^{-3}$, et $9$ époques d'entraînement, c'est-à-dire un entraînement fort, rapide et très faiblement régularisé. Cette combinaison n'est pas idéale pour la généralisation.

Il est étonnant que des versions de ce modèle avec un plus faible taux d'apprentissage, une plus forte régularisation et de plus grandes valeurs de $k$ n'aient su surclasser la version présentée ici. Nous pensons que ce modèle pourrait bénéficier d'une meilleure régularisation, car les versions juste un petit-peu moins performantes sur $\mathscr{D}_{valid}$ à l'issue de l'hyper-optimisation disposaient d'une plus capacité légèrement plus grande (mais toujours sous la dizaine d'unités), d'un taux d'apprentissage un ordre plus faible ($\sim a \times 10^{-3}$), d'un coefficient de régularisation d'un ordre plus élevé ($\sim b \times 10^{-5}$), et présentaient sur $\mathscr{D}_{test}$ une erreur de généralisation **plus faible** de quelques pourcents sur $top_k$ .

SDN$3_\varnothing$ , le seul modèle sans pré-entraînement, obtient ses résulats grâce à un taux d'apprentissage élevé dans tous les meilleurs ensembles d'hyper-paramètres ($\sim a \times 10^{-2}$), lui permettant en un nombre d'époques assez bas d'obtenir au moyen de trois couches cachées

assez larges une performance avoisinant celle d'un modèle avec une unique couche cachée plus fine mais pré-entraînée. Puisque tous les résultats Les performances des 3 types de pré-entraînement se tiennent dans le même centile, penchons-nous sur les différences entre les hyper-paramètres issus de la selection de modèle.

Tableau 5.1 – Résultats des meilleurs modèles pour l'architecture SDN

| Modèle | capacité | époques | MSE ($\sigma \times 10^{-3}$) | $top_k$ ($\sigma \times 10^{-3}$) | heur. $\lambda$ |
|---|---|---|---|---|---|
| random | - | - | 0.24 (1.6) | 64.54 (5.9) | - |
| baseline | - | - | 0.5 (9.1) | 85.49 (1.2) | - |
| SVD | $k = 2$ | 114 | 0.05 (1.4) | 85.94 (5.5) | - |
| $SDN_{\varnothing}$ | 525, 303, 274 | 0, 0, 0, 9 | 0.4 (3.3) | 88.16 (6.8) | *ADAGRAD* |
| $SDN_{AE}$ | 116 | 15, 36 | 0.39 (4.4) | 88.19 (6.4) | *ADAGRAD* |
| $SDN_{DAE}$ | 202 | 330, 75 | 0.40 (7.8) | 87.86 (1.0) | annealing |
| $SDN_{CAE}$ | 382 | 15, 66 | **0.39 (3.5)** | **88.27 (14)** | *ADAGRAD* |

Performances de l'architecture SDN selon les trois modes de pré-entraînement AE, DAE et CAE, et avec un entraînement supervisé uniquement. La colonne «capacité» correspond pour SVD au nombre $k$ de facteurs latents alloués à chaque utilisateur ou article, et à la taille de chaque couche cachée pour les modèles pré-entraînés. La colonne «époques» rapporte la durée d'entraînement de la première couche à la dernière, le cas échéant, et enfin la durée de la phase de *fine-tuning*. $SDN_{AE}$ n'a pas pris de régularisation. Le taux de corruption de $SDN_{DAE}$ est de $3.7 \times 10^{-3}$. Le coefficient de contraction de $SDN_{CAE}$ est de $1.7 \times 10^{-4}$.

On peut constater l'influence manifeste de l'heuristique *ADAGRAD* pour le calcul du taux d'apprentissage sur la vitesse de convergence. On voit avec $SDN_{AE}$ et $SDN_{CAE}$ , que le nombre de paramètres, aussi bien inférieur ou supérieur respectivement, ne semble pas influencer *ADAGRAD*. En réalité son effet est probablement même plus important dans le cas de $SDN_{CAE}$ . Pour comprendre pourquoi, rappelons qu'*ADAGRAD* calcule un taux d'apprentissage *par* paramètre. En conséquence, si tous les paramètres sont denses, à l'opposé d'épars (*sparses*), leurs scalaires associés par *ADAGRAD*, les $\alpha_i$, tiendront probablement dans un même intervalle, et l'heuristique n'aura pas l'effet escompté de gonfler $\lambda$ pour certains paramètres plus que d'autres. À l'opposé, si les entrées **ou** les paramètres sont épars, alors les $\alpha_{i,t}$ seront plus épars, à chaque itération, et les taux d'apprentissage spécifiques seront ajustés plus judicieusement. L'ensemble de données *hyperquest* n'est pas épars, par contre la pénalité de contraction du CAE pousse le modèle à construire des représentations éparses en saturant ses paramètres afin de minimiser les dérivées $\frac{\partial h}{\partial x}$. Elle semble donc agir comme un catalyseur naturel vis-à-vis d'*ADAGRAD*.

En ce qui concerne $SDN_{DAE}$ , il n'est pas possible qu'*ADAGRAD* se comporte vraiment

comme espéré, puisque la corruption stochastique d'un attribut $x_i$ va faire croître artificiellement les sommes correspondantes, c'est-à-dire gâcher le *potentiel* [1] des paramètres auxquels ces sommes sont dédiées par *ADAGRAD* ; potentiel à améliorer soudainement et fortement la contribution du $x_i$ à la modélisation de la distribution de $\mathscr{D}$ , et ainsi masquer les possibles spécificités de cet attribut dans des zones de plus faibles densité. Pour résumer, les deux méthodes ne sont pas vraiment compatibles car *ADAGRAD* s'attend à ce que chaque poids soit responsable d'une perte décroissante au fil des itérations, alors que l'entraînement débruitant mise sur une entrée bruitée *au hasard* pour apprendre à long terme, et donc implique que la reconstruction va toujours se tromper un minimum.

Comme mentionné dans [6] et [31] en tant qu'argument en faveur du pré-entraînement contractant, l'optimisation opérée par l'entraînement débruitant est *stochastique* plutôt qu'*analytique*, c'est-à-dire que là où la pénalité du CAE encouragerait l'insensibilité du modèle à des variations de $x$ dans *toutes* ses dimensions en une seule itération, le DAE n'apprend à ignorer ces variations qu'une direction à la fois pour chaque dimension corrompue (dans la direction du bruit échantilloné pour cet exemple). Ceci pourrait apporter un complément d'explication de pourquoi le DAE met 20 fois plus d'itérations pour atteindre une configuration de paramètres équivalente au CAE en terme de performance $top_k$ .

Le nombre de couches faisait partie des hyper-paramètres testés dans cette série d'expériences. Des modèles à une, deux et trois couches ont donc bien été optimisés, mais aucun modèle profond n'a surpassé ceux à une couche. Une explication possible vient de la méthodologie de selection de modèle elle-même : peut-être l'exploration d'hyper-paramètres n'a pas été assez poussée. Peut-être que les données sont suffisament simples pour que leurs corrélations soient captées par une seule couche. Mentionnons tout de même que nombre de modèles à deux ou trois couches performaient dans le même centile de $top_{k=1}$ que ceux présentés ici.

## 5.2 EDN

Nous présentons dans cette section les résultats des expériences menées avec l'architecture EDN, et tentons de comprendre les résultats obtenus.

1. Nous appelons *potentiel* le fait, pour un poids $\theta_i$, d'être associé à une faible somme (faible dénominateur), c'est-à-dire avoir un taux d'apprentissage relativement élevé et ainsi conserver un plus fort pouvoir de modification du modèle.

### 5.2.1 Modèles de base

L'architecture EDN est plus sensible que SDN aux variations de la méthode de pré-entraînement des couches qui la composent, et propose en conséquence des performances plus variées. Les résultats sont présentés dans la table 5.2.1.

Lors de l'hyper-optimisation, tous les meilleurs modèles étaient composés d'au moins deux couches, ce qui confirme que l'architecture EDN tire mieux profit de l'entraînement successif non-supervisé, dans notre cadre expérimental tout du moins. Ceci est probablement dû au fait que le calcul du gradient utilise chaque paramètre deux fois, dans la phase d'encodages successifs puis dans la phase de décodages successifs, au lieu d'une seule pour SDN. Comme le gradient dépend directement de la faculté de reconstruction des différentes couches, l'erreur de l'une d'entre elles se répercute «plus longtemps» dans le calcul et influence donc probablement davantage l'erreur de reconstruction finale, rendant l'architecture plus sensible à des incohérences dans l'entrée comme la présence d'un score très différent de la note ; incohérences que l'on veut qu'EDN reconstruise mal.

Tableau 5.2 – Résultats des meilleurs modèles pour l'architecture EDN

| Modèle | capacité | époques | MSE ($\sigma \times 10^{-3}$) | $top_k$ ($\sigma \times 10^{-3}$) | heur. $\lambda$ |
|---|---|---|---|---|---|
| random | - | - | 0.24 (1.6) | 64.54 (5.9) | - |
| baseline | - | - | 0.5 (9.1) | 85.49 (1.2) | - |
| SVD | $k = 2$ | 114 | 0.05 (1.4) | 85.94 (5.5) | - |
| $EDN_{AE}$ | 78, 477, 137 | 30, 138, 15 | 0.12 (1.2) | 79.58 (42) | annealing |
| $EDN_{DAE}$ | 190, 148, 73 | 36, 15, 111 | 0.12 (2.2) | 76.40 (68) | annealing |
| $EDN_{CAE}$ | 56, 200, 261 | 18, 15, 84 | 0.12 (1.4) | **86.97 (4.7)** | *ADAGRAD* |

Performances de l'architecture EDN selon les trois modes de pré-entraînement AE, DAE et CAE. La colonne «époques» rapporte la durée d'entraînement de la première couche à la dernière, le cas échéant. Les performances de *random*, *baseline* et SVD, identiques à la série d'expériences précédente, sont rappelées pour faciliter la comparaison. $EDN_{AE}$ n'a pas pris de régularisation. Le taux de corruption de $EDN_{DAE}$ , identique pour ses 3 couches, est de $2.4 \times 10^{-3}$. Les coéfficients de contraction de $EDN_{CAE}$ pour ses 3 couches sont respectivement de $17 \times 10^{-3}$, $1.1 \times 10^{-5}$ et $82 \times 10^{-3}$.

Encore une fois, on peut observer que l'heuristique *ADAGRAD* se marie mieux avec le pré-entraînement contractant qu'avec les autres, puisqu'$EDN_{CAE}$ présente des durée de pré-entraînement plus courtes que les autres modèles. Seule la troisième couche fait exception ; ceci pourrait être dû au fait que l'extraction de caractéristiques des deux couches précédentes était suffisament de qualité pour qu'il y ait effectivement de nouvelles choses à apprendre et

donc raison à prolonger l'entraînement, et pourrait expliquer l'écart de $top_{k=1}$ avec $\text{EDN}_{\text{AE}}$ et $\text{EDN}_{\text{DAE}}$ .

### 5.2.2 $\text{EDN}_{\text{CAE+ft}}$

La phase de *fine-tuning* optimale a duré $150$ époques pour un taux d'apprentissage de base relativement élevé de l'ordre de $3.5 \times 10^{-2}$, très probablement dû à l'utilisation d'*ADAGRAD*, qui «se réserve» ainsi un plus grand intervalle de valeurs pour ses sommes dédiées aux paramètres. L'optimisation par minimisation de la NLL fait nettement baisser la MSE, tout en augmentant le score $top_k$ . Ce critère sera donc retenu pour sa pertinence dans le contexte des modèles à énergie. Ce résultat semble cohérent. On pouvait en effet s'attendre à ce que la performance soit meilleure puisque cet entraînement est supervisé et pousse donc explicitement le modèle à prédire le bon score pour un niveau, mais la littérature et la pauvre performance de $\text{EDN3}_{\varnothing\text{+ft}}$ montrent que ça n'aurait pas été possible sans le pré-entraînement non-supervisé.

Sans le pré-entraînement non-supervisé, comme les deux phases d'encodages-décodages successifs rendent le modèle deux fois plus profond que son nombre de couches, et cette profondeur empêche la descente de gradient de mettre à jour les poids. Lors du *fine-tuning* les différentes couches apprennent essentiellement à bien reconstruire, mais à aucun moment n'ont-elles été entraînées à construire des représentations utiles. La phase de pré-entraînement non-supervisée est donc indispensable à ce modèle (à cette profondeur tout du moins).

Tableau 5.3 – Résultats des meilleurs modèles pour l'architecture $\text{EDN3}_{\text{CAE+ft}}$

| Modèle | époques | MSE ($\sigma \times 10^{-3}$) | $top_k$ ($\sigma \times 10^{-3}$) |
|---|---|---|---|
| random | - | 0.24 (1.6) | 64.54 (5.9) |
| baseline | - | 0.5 (9.1) | 85.49 (1.2) |
| SVD | 114 | 0.05 (1.4) | 85.94 (5.5) |
| $\text{EDN3}_{\text{CAE}}$ | 18, 15, 84 | 0.12 (1.4) | 86.97 (4.7) |
| $\text{EDN3}_{\varnothing\text{+ft}}$ | 81 | $10\times10^{-2}$ (8.8) | 80.66 (35) |
| $\text{EDN3}_{\text{CAE+ft}}$ | 18, 15, 84, 150 | **7.2**$\times10^{-2}$ **(3.2)** | **89.00 (7.2)** |

Performances de l'architecture $\text{EDN3}_{\text{CAE+ft}}$ après le pré-entraînement successif de ses trois couches pour une durée d'entraînement respectivement 18, 15, et 84 époques. L'entraînement supervisé a duré 150 époques. Après selection du meilleur modèle, moyenne (et écart-type) des MSE et $top_{k=1}$ sont aggrégés sur 10 expériences.

La supériorité de $\text{EDN3}_{\text{CAE+ft}}$ sur $\text{SDN3}_{\text{CAE}}$ peut s'expliquer par au moins deux raisons venant de l'architecture même d'EDN. Celle-ci est telle que les poids sont utilisés $2 \times |S|$

fois dans la production d'une note [2] : une fois par score possible et par propagation [3]. L'effet de régularisation provoqué par l'utilisation de poids liés (1.6) affecte non-seulement le pré-entraînement de façon locale, mais aussi le *fine-tuning* : les mises à jour des poids dans cette seconde phase contiennent des termes relatifs aux deux propagations (majoritairement ceux de la propagation arrière en raison de la diffusion du gradient 2.3.1). On sait qu'optimiser les premières couches d'un réseau profond est une tâche cruciale et difficile [6] car l'initialisation de ces couches a une importante influence sur l'optimisation qui lui succède (utilité des représentations apprises, intialisation des poids, etc). De plus, chacune des $n$ couches du modèle se trouve à une distance moyenne de $n/2$ couches de la sortie, ce qui protège de l'effet de diffusion du gradient observé dans les réseaux profonds traditionnels, et qui a longtemps bloqué leur optimisation.

la performance de $\text{EDN3}_{\varnothing\text{+ft}}$ au niveau de la métrique $top_k$ met en valeur l'intérêt de l'entraînement non-supêrvisé. On voit que la phase de *fine-tuning* suffit au modèle pour diminuer son erreur quadratique

### 5.2.3 $\text{EDN}_{\text{CAE+fth}}$

Le modèle $\text{EDN3}_{\text{CAE+fth}}$ , du fait de l'entraînement final multi-tâches semi-supervisé, est poussé à la généralisation : en utilisant une composition de critères d'entraînement plus ou moins indépendants dans leurs objectifs et les paramètres qu'ils optimisent, le modèle est forcé à être polyvalent. Il n'est pas capable de vouer toute sa capacité à la même tâche, et doit apprendre à bien généraliser sur les exemples qu'il voit afin de mieux distribuer sa capacité pour accomplir les différentes sous-tâches. De plus, il est possible que l'optimisation d'une composante du critère pousse le modèle à modéliser des dépendances ayant finalement une influence sur sa performance pour d'autres composantes.

L'optimisation du coefficient $\alpha$ pondérant les deux termes du critère d'entraînement (4.2 page 48) semble être très importante, car les différents critères utilisés dans ce contexte multi-tâches fonctionnent dans des ordres de valeur assez différents. La valeur issue de l'hyper-optimisation, $1.02 \times 10^{-2}$, l'illustre parfaitement.

Les résultats nous montrent que le modèle ne profite pas autant de l'entraînement multi-tâches que d'un pur *fine-tuning* supervisé. Peut-être est-ce dû au fait que la capacité n'est pas suffisante pour performer sur les deux tâches à la fois. Une autre explication possible à cette moindre performance est l'utilisation de la MSE comme critère d'optimisation de la

2. Pour rappel, $S$ est l'ensemble des notes possibles.
3. propagation avant lors des encodages successifs, et «propagation arrière» lors des décodages successifs.

Tableau 5.4 – Résultats des meilleurs modèles pour l'architecture $\text{EDN3}_{\text{CAE+fth}}$

| Modèle | époques | MSE ($\sigma \times 10^{-3}$) | $top_k$ ($\sigma \times 10^{-3}$) |
|---|---|---|---|
| random | - | 0.24 (1.6) | 64.54 (5.9) |
| baseline | - | 0.5 (9.1) | 85.49 (12) |
| SVD | 114 | 0.05 (1.4) | 85.94 (5.5) |
| $\text{EDN3}_{\text{CAE+ft}}$ | 150 | **7.2**$\times 10^{-2}$ **(3.2)** | **89.00 (7.2)** |
| $\text{EDN3}_{\text{CAE+fth}}$ | 450 | $1.2 \times 10^{-1}$ (1.4) | 88.05 (7.7) |

Performances de l'architecture $\text{EDN3}_{\text{CAE+fth}}$ après le pré-entraînement successif de ses trois couches pour une durée de pré-entraînement respectivement 18, 15, et 84 époques, et un *fine-tuning* de 450 époques. Après selection du meilleur modèle, moyenne (et écart-type) des MSE et $top_{k=1}$ sont aggrégés sur 10 expériences.

fonction d'énergie. Le score $top_{k=1}$ , meilleur que pour $\text{EDN3}_{\text{CAE}}$ mais moins bon que pour $\text{EDN3}_{\text{CAE+fth}}$ , laisse penser que les effets des deux critères s'annulent en partie.

## 5.3 Synthèse

La table suivante récapitule les performances des différents modèles dont nous avons étudié les caractéristiques dans ce chapitre.

Tableau 5.5 – Résultats des modèles étudiés dans ce document

| Modèle | MSE ($\sigma \times 10^{-3}$) | $top_k$ ($\sigma \times 10^{-3}$) |
|---|---|---|
| random | 0.24 (1.6) | 64.54 (5.9) |
| baseline | 0.5 (9.1) | 85.49 (12) |
| SVD | 0.05 (1.4) | 85.94 (5.5) |
| $\text{SDN}_{\varnothing}$ | 0.4 (3.3) | 88.16 (6.8) |
| $\text{SDN}_{\text{AE}}$ | 0.39 (4.4) | 88.19 (6.4) |
| $\text{SDN}_{\text{DAE}}$ | 0.40 (7.8) | 87.86 (1.0) |
| $\text{SDN}_{\text{CAE}}$ | 0.39 (3.5) | 88.27 (14) |
| $\text{EDN}_{\text{AE}}$ | 0.12 (1.2) | 79.58 (42) |
| $\text{EDN}_{\text{DAE}}$ | 0.12 (2.2) | 76.40 (68) |
| $\text{EDN}_{\text{CAE}}$ | 0.12 (1.4) | 86.97 (4.7) |
| $\text{EDN3}_{\text{CAE+ft}}$ | **7.2**$\times 10^{-2}$ **(3.2)** | **89.00 (7.2)** |
| $\text{EDN3}_{\text{CAE+fth}}$ | $1.2 \times 10^{-1}$ (1.4) | 88.05 (7.7) |

# CHAPITRE 6

# CONCLUSION

Le travail présenté porte sur le problème de la recommandation à l'utilisateur dans un contexte industriel très concret de développement d'un jeu vidéo multi-joueurs en ligne, au moyen de réseaux de neurones profonds. Il compare une approche traditionnelle dans son utilisation des réseaux à une approche inspirée des modèles à énergie plus souple et tirant mieux partie des caractéristiques spécifiques de ce type de modèle. La quantité d'information nous submergeant chaque jour et la façon dont l'informatique est utilisée pour la filtrer dans une multitudes de domaines connexes justifiait cette initiative, tout comme les résultats auxquels nous sommes parvenus.

Dans ce dernier chapitre, nous présentons un retour sur les expériences effectuées et les principales explications y aboutissant. Enfin, nous proposerons quelques voies d'exploration possibles pour compléter et étendre la recherche sur le problème de la recommandation et les modèles proposés pour le résoudre, tant au niveau de la méthodologie que des modèles-mêmes.

## 6.1 Retour sur les expériences

Une première phase, d'exploitation, visait à assouvir la curiosité naturelle d'estimer la valeur d'une architecture standard face à la tâche d'évaluation du plaisir d'un joueur vis-à-vis d'un niveau. Pour celà, un modèle-patron a permis l'évaluation de trois méthodes de pré-entraînement pour trois niveaux de profondeur, selon deux heuristiques de choix du taux d'apprenstissage : un recuit simulé ou *ADAGRAD*. Nous avons observé que les trois méthodes de pré-entraînement — simple, débruitant, contractant — réagissent différemment avec les autres caractéristiques du modèle, comme sa capacité, le nombre d'itérations nécessaires à l'optimisation, ou encore l'heuristique de choix du taux d'apprentissage. Il apparait que ces facteurs intéragissent activement au cours de l'apprentissage, notamment lors de l'utilisation de l'heuristique déterminant l'évolution du taux d'apprentissage, *ADAGRAD*, et s'il était hors de propos de faire état de toutes leurs interactions étant donné le nombre élevé des configurations possibles, nous avons tenté d'expliquer celles que nos expériences révélaient le plus. Bien que nous n'ayons su mettre à profit les avantages théoriques que la superposition de couches devrait apporter avec les variantes profondes, la performance de ce modèle prélim-

inaire sur une métrique reflétant les caractéristiques de la tâche de recommandation montre qu'une architecture standard est capable de relever ce défi, puisque $\text{SDN3}_{\text{CAE}}$ bat significativement le modèle SVD, couramment rencontré dans la littérature. L'architecture SDN justifie donc l'utilisation d'auto-encodeurs dans un *framework* de recommandation. La tâche était d'autant plus difficile pour SDN que le critère d'entraînement utilisé, l'erreur quadratique, n'est possiblement pas le plus adapté pour cette tâche ; il est juste fréquemment employé avec ce type de modèle.

Une seconde phase, plus explorative, nous a permis d'établir une comparaison entre le meilleur modèle de la première phase et différentes variations d'une architecture nouvelle, empruntée des modèles basés sur une fonction d'énergie. En suivant le même protocole expérimental, les trois types de pré-entraînement ont été testés sur une première version de ce modèle sans *fine-tuning*. L'étape de *fine-tuning* n'avait pas semblé indispensable dans un premier temps, car la prédiction est fonction de la capacité du modèle à *reconnaître* son entrée. Dès lors, nous avons pensé qu'un entraînement non-supervisé visant simplement à apprendre la distribution de $\mathscr{D}$ pouvait montrer des performances valides. Ce fut le cas pour un seul type de pré-entraînement, contrastant nettement avec les deux autres, bien que tous trois aient validé le gain qu'apporte l'ajout de profondeur à cette architecture stratifiée ; gain dont nous avons tenté d'énoncer certaines causes. Comme le modèle $\text{EDN3}_{\text{CAE}}$ montrait des performances intéressantes mais en deçà de ce qu'on relevait pour SDN, nous avons essayé une première variation tirant parti de la phase de *fine-tuning*, dans laquelle une maximisation de la log-vraisemblance de la note a permis au modèle de se spécialiser dans sa tâche discriminante au point de légèrement surpasser le modèle SDN. La log-vraisemblance négative en tant que critère d'entraînement s'est montrée particulièrement efficace dans l'optimisation du modèle, en ayant le double effet sur sa fonction d'énergie de favoriser le bon score et de défavoriser les autres. Afin de nous assurer que cette spécialisation ne se faisait pas au prix d'une moindre capacité de généralisation, nous avons appliqué un résultat de la littérature, venant de l'entraînement semi-supervisé multi-tâches des RBM, visant à empêcher une surspécialisation du modèle en composant la NLL avec le critère de pré-entraînement. Au vu du score $top_{k=1}$ comparable à celui de $\text{SDN3}_{\text{CAE}}$ , il semble que cette stratégie n'est pas adéquate telle qu'elle a été appliqué, mais pourrait le devenir s'il l'on trouvait un meilleur critère pour compléter la NLL.

## 6.2 Futurs travaux et améliorations

Bien que les résultats auxquels nous parvenoins justifient l'utilisation des techniques présentées, l'exploration des modèles d'apprentissage-machine profonds pour répondre au problème de la recommandation à l'usager est évidemment loin d'être complète.

### 6.2.1 Méthodologie

Nous avons pu observer que la MSE ne reflète pas toutes les facettes de la métrique $top_k$ . De la même façon, cette métrique ne reflète pas forcément toutes les facettes de la tâche *recommender de bons articles* sur laquelle nous nous sommes concentrés. En particulier, elle n'offre aucune garantie sur les capacités des modèles à explorer le vivier d'articles à disposition. Compléter l'**éventail de métriques** utilisées ici avec d'autres, pertinentes et **complémentaires** nous équiperait mieux pour observer les comportements des différentes variations de modèles étudiées.

La technique d'exploration des hyper-paramètres effectuée est celle qui présentait à nos yeux le meilleur ratio qualité/temps d'implémentation. Ça n'est cependant pas la meilleur stratégie dans l'absolue. Comme cité précédemment, diverses méthodes existent. La profondeur, la nature des sous-modèles utilisés dans les architectures décrites précédemment, et en particulier leur aspect **combinatoire** émergeant du développement des **patrons** cités précédemment, nous pousse à valoriser une méthode plus efficace que la recherche aléatoire. En particulier, des techniques d'optimisation bayesienne [7] visant une hyper-optimisation un peu plus intelligente nous semblent un *next step* assez important.

L'ensemble de données un peu particulier avec lequel nous avons travaillé implique que nos modèles nécessitent des attributs représentant chaque utilisateur et chaque article. Rares sont les ensembles de données avec de telles caractéristiques, même s'il serait intéressant d'observer si les modèles présentés sauraient modéliser de façon prédictive des notes associées à des *embeddings*. De tels *embeddings* pourraient être construits par analyse en composante principale des vecteurs de notes d'usagers $U_c$ et d'articles $U_s$, ou être les représentations cachées d'auto-encodeurs entraînés à reconstruire ces mêmes vecteurs. Sans aller chercher dans des ensembles de données de natures différentes (et donc comprenant possiblement des distributions tout aussi différentes), il aurait pu être bénéfique, de procéder à une validation croisée des résultats des modèles. La faible variance des résultats pour différentes sections de l'ensemble de données ne semblent cependant pas montrer l'absolue nécessité d'un tel procédé.

### 6.2.2 Pistes de recherche

Au niveaux des modèles, plusieurs pistes de recherche nous sont apparues au fil de ce travail, et les contraintes de temps sont les principales responsables de leur absence dans ce mémoire. Les prochains paragraphes listent, sans garantie d'efficacité, les voies inexplorées qui nous semblent mériter une attention.

Le pré-entraînement selon les trois types d'auto-associateur couramment utilisés au LISA nous a aidé à comprendre certaines de leurs caractéristiques, positives ou négatives, ainsi que celles des deux architectures SDN et EDN. L'idée sous-jacente derrière notre stratégie serait de pousser le modèle à prendre davantage en compte les variations du vecteur *one-hot* (représentant la note) durant l'entraînement, afin que sa valeur influe davantage la procédure d'inférence. Après la propagation avant dans la couche de premier niveau, la note se trouve en partie dans chaque unité cachée, mais il est difficile de savoir à quel point elle est diffuse, de savoir si cette représentation cachée porte toujours cette information de façon *utile* pour la couche suivante. Une analyse plus poussée des transformations opérée par chaque couche aurait pu aider à mieux comprendre ce processus, et peut-être à le superviser. Le modèle le plus performant au terme des expériences nous a particulièrement interessé, en particulier au niveau de sa technique de pré-entraînement, et pourrait probablement user de quelques optimisations supplémentaires. Par exemple la pénalité de contraction, qui pousse à l'insensibilité de la représentation cachée pour de faibles variation de l'entrée, fait en réalité l'exact opposé de ce qu'on voudrait **au niveau du vecteur one-hot** encodant la note. Intuitivement, si — au moins — la première couche était très sensible à la note, elle pourrait «déformer» sa représentation des attributs en fonction de cette note ; la tâche des couches supérieures pourrait alors être simplifiée.

Tout comme les métriques utilisées étaient suffisantes mais certainement pas exhaustives pour l'évaluation des modèles, la fonction de coût utilisée dans la phase de fine-tuning a permis l'élaboration de modèles efficaces, mais davantage de variations dessus auraient pu être étudiées. En particulier, [25] liste un certain nombre de fonctions aux propriétés intéressantes, dont la composition dans le contexte multi-tâche pourrait apporter davantage d'information sur le comportement de l'architecture EDN, en fonction des méthodes de pré-entraînement citées auparavant.

## 6.3 En conclusion

Nous avons justifié l'utilisation d'architectures traditionnelles et nouvelles de l'apprentissage machine pour accomplir une tâche de recommandation dans le contexte atypique du jeu multi-joueurs en ligne. Bien que nos différents essais n'aient pas tous atteint les mêmes performances sur le jeu de données utilisé, nous obtenons des résultats intéressants, battant notamment des modèles plus simples mais reconnus dans la littérature. Les premières optimisations sur un modèle en particulier laissent penser qu'une recherche plus poussée pourrait tirer mieux parti de ses caractéristiques afin d'encore mieux répondre au problème.

## BIBLIOGRAPHIE


[1] Gediminas Adomavicius and Alexander Tuzhilin. Toward the next generation of recommender systems : A survey of the state-of-the-art and possible extensions. *IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING*, 17(6) :734–749, 2005.

[2] Pierre Baldi and Kurt Hornik. Learning in linear neural networks : a survey. *IEEE Transactions on neural networks*, 6 :837–858, 1995.

[3] Eric B. Baum and David Haussler. What size net gives valid generalization ? In *NIPS*, pages 81–90, 1988.

[4] Y. Bengio, J. Louradour, R. Collobert, and J. Weston. Curriculum learning. In *International Conference on Machine Learning, ICML*, 2009.

[5] Yoshua Bengio. Learning deep architectures for AI. *Foundations and Trends in Machine Learning*, 2(1) :1–127, 2009. Also published as a book. Now Publishers, 2009.

[6] Yoshua Bengio, Aaron C. Courville, and Pascal Vincent. Unsupervised feature learning and deep learning : A review and new perspectives. *CoRR*, abs/1206.5538, 2012.

[7] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *NIPS*, pages 2546–2554, 2011.

[8] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research*, 13 :281–305, 2012.

[9] James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano : a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral Presentation.

[10] Miguel A. Carreira-Perpinan and Geoffrey E. Hinton. On contrastive divergence learning.

[11] Rich Caruana. Multitask learning. In *Machine Learning*, pages 41–75, 1997.

[12] O. Chapelle, B. Schölkopf, and A. Zien, editors. *Semi-Supervised Learning*. MIT Press, Cambridge, MA, 2006.

[13] Abhinandan S. Das, Mayur Datar, Ashutosh Garg, and Shyam Rajaram. Google news personalization : scalable online collaborative filtering. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pages 271–280, New York, NY, USA, 2007. ACM.

[14] Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. Indexing by latent semantic analysis. *JOURNAL OF THE AMERICAN SOCIETY FOR INFORMATION SCIENCE*, 41(6) :391–407, 1990.

[15] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research (JMLR 2011)*.

[16] Spaff from Media Molecule. "LittleBigPlanet : The Road to 7 Million Levels",, 2012.

[17] Ralf Herbrich, Tom Minka, and Thore Graepel. TrueSkill(TM) : A bayesian skill rating system. 20, 2007.

[18] Jonathan L. Herlocker, Joseph A. Konstan, Loren G. Terveen, and John T. Riedl. Evaluating collaborative filtering recommender systems. *ACM Trans. Inf. Syst.*, 22(1) :5–53, jan 2004.

[19] Geoffrey E. Hinton and Simon Osindero. A fast learning algorithm for deep belief nets. *Neural Computation*, 18 :2006, 2006.

[20] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Comput.*, 18(7) :1527–1554, July 2006.

[21] Kurt Hornik, Maxwell B. Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5) :359–366, 1989.

[22] David H. Hubel and Torsten N. Wiesel. Receptive fields of single neurons in the cat's striate cortex. *Journal of Physiology*, 148 :574–591, 1959.

[23] Yehuda Koren. Factorization meets the neighborhood : a multifaceted collaborative filtering model. In *In Proc. of the 14th ACM SIGKDD conference*, pages 426–434, 2008.

[24] Hugo Larochelle and Yoshua Bengio. Classification using discriminative restricted boltzmann machines. In *Proceedings of the 25th international conference on Machine learning*, ICML '08, pages 536–543, New York, NY, USA, 2008. ACM.

[25] Yann LeCun, Sumit Chopra, Raia Hadsell, Marc'Aurelio Ranzato, and Fu-Jie Huang. A tutorial on energy-based learning. In G. Bakir, T. Hofman, B. Schölkopf, A. Smola, and B. Taskar, editors, *Predicting Structured Data*. MIT Press, 2006.

[26] G. Linden, B. Smith, and J. York. Amazon.com recommendations : item-to-item collaborative filtering. *Internet Computing, IEEE*, 7(1) :76–80, 2003.

[27] Greg Linden, Brent Smith, and Jeremy York. Amazon.com recommendations : Item-to-item collaborative filtering. *IEEE Internet Computing*, 7(1) :76–80, 2003.

[28] Barak Pearlmutter. Gradient descent : Second-order momentum and saturating error. In *In (Moody et al*, pages 887–894. Morgan Kaufmann, 1992.

[29] Ning Qian. On the momentum term in gradient descent learning algorithms, 1999.

[30] P. Resnick, N. Iacovou, M. Suchak, P. Bergstorm, and J. Riedl. Grouplens : An open architecture for collaborative filtering of netnews. In *Proceedings of ACM 1994 Conference on Computer Supported Cooperative Work*, pages 175–186. ACM, 1994.

[31] Salah Rifai, Pascal Vincent, Xavier Muller, Xavier Glorot, and Yoshua Bengio. Contractive auto-encoders : Explicit invariance during feature extraction. In *Proceedings of the Twenty-eight International Conference on Machine Learning (ICML'11)*, jun 2011.

[32] Chris Volinsky Robert M. Bell, Yehuda Koren. The BellKor solution to the netflix prize. 2008.

[33] Chris Volinsky Robert M. Bell, Yehuda Koren. The bellkor solution to the netflix prize, 2009.

[34] F. Rosenblatt. The perceptron : A perceiving and recognizing automaton (project para). Technical Report 85-460-1, Cornell Aeronautical Laboratory, 1957.

[35] Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, and Pierre-Antoine Manzagol. Stacked denoising autoencoders : Learning useful representations in a deep network with a local denoising criterion. 11 :3371–3408, dec 2010.

[36] Bengio Y. Deep learning of representations for unsupervised and transfer learning. *JMLR W CP : Proc. Unsupervised and Transfer Learning*, 2011.

[37] Zied Zaier, Robert Godin, and Luc Faucher. Evaluating recommender systems. In *AXMEDIS '08 : Proceedings of the 2008 International Conference on Automated solutions for Cross Media Content and Multi-channel Distribution*, pages 211–217, Washington, DC, USA, nov 2008. IEEE Computer Society.