Comparaison des lois conjointes et marginales par permutation des labels pour la régression et l'estimation de densité conditionnelle

Benjamin Riu

▶ To cite this version:

Benjamin Riu. Comparaison des lois conjointes et marginales par permutation des labels pour la régression et l'estimation de densité conditionnelle. Machine Learning [stat.ML]. Institut Polytechnique de Paris, 2022. Français. NNT : 2022IPPAX056 . tel-04094738

HAL Id: tel-04094738
https://theses.hal.science/tel-04094738v1

Submitted on 11 May 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT POLYTECHNIQUE DE PARIS

NNT : 2022IPPAX056

Thèse de doctorat

Comparaison des lois conjointes et marginales par permutation des labels pour la régression et l'estimation de densité conditionnelle

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à l'École polytechnique

École doctorale n°574 École doctorale de mathématiques Hadamard (EDMH)
Spécialité de doctorat : Mathématiques appliquées (4200001)

Thèse présentée et soutenue à Palaiseau, le 13 septembre 2022, par

BENJAMIN RIU

Composition du Jury :

Arnak Dalayan Enseignant Chercheur, ENSAE (CREST UMR9194)	Président
Madalina Olteanu Enseignante Chercheuse, Université Paris Dauphine (CEREMADE UMR7534)	Rapporteure
Joseph Salmon Enseignant Chercheur, Université de Montpellier (IMAG UMR5149)	Rapporteur
Mohamed Hebiri Maître de conférences, Université Gustave Eiffel (LAMA UMR 8050)	Examinateur
Karim Lounici Enseignant Chercheur, Ecole Polytechnique (UMR7641)	Directeur de thèse
Katia Meziani Maître de conférences, Université Paris Dauphine (CEREMADE UMR7534)	Co-directrice de thèse
Martial Hue Encadrant scientifique, Partenaire Cifre Jellyfish	Invité

Comparaison des lois conjointes et marginales par permutation des labels pour la régression et l'estimation de densité conditionnelle

Benjamin Riu

Cette thèse est dédiée à mon grand père Georges Guez, j’espère un jour honorer ce qu’il représentait et fit perdurer.

To see a World in a Grain of Sand
And a Heaven in a Wild Flower,
Hold Infinity in the palm of your hand
And Eternity in an hour.

William Blake, Auguries of Innocence

Résumé pour le grand public

Les méthodes de statistiques mathématiques et de machine learning permettent, grâce aux données, de faire des liens, d'anticiper, d'optimiser, d'agir plus vite et mieux. C'est dans ce cadre général que s'inscrit cette thèse, centrée en particulier sur la généralisation et l'estimation de densité conditionnelle.

La généralisation est le mécanisme qui permet à un modèle prédictif de capturer à partir d'un jeu de données, le lien qui existe entre une variable que l'on cherche à prédire et des variables explicatives. Pour mesurer la capacité d'un modèle à généraliser, on évalue la précision de ses prédictions pour de nouvelles données qu'il n'a pas vu durant la phase d'apprentissage. On propose ici une nouvelle méthode pour améliorer les performances de généralisation d'un modèle, basée sur l'idée que si, lorsque l'on entraîne un modèle à prédire les données disponibles, on l'empêche d'apprendre par cœur une à une les observations disponibles, le modèle devra chercher à comprendre les mécanismes sous-jacents au problème.

L'estimation de densité conditionnelle est une tâche qui consiste à construire un modèle prédictif qui ne se limite pas à prédire ce qu'il se passera en moyenne ou dans le cas le plus probable, mais va au contraire être capable de donner, pour chaque scénario possible, la probabilité que cet événement puisse survenir. C'est un problème plus compliqué que de simplement prédire la moyenne. On propose ici une méthode pour transformer ce problème en un autre beaucoup plus simple, qui consiste à prédire de façon binaire si oui ou non une observation provient du vrai jeu de données d'origine, ou si elle a été fabriquée artificiellement.

Summary for the general public

Statistics and machine learning solutions can leverage data to make links, anticipate, optimize and act faster and better. This thesis is part of this general framework, focusing in particular on generalization and conditional density estimation.

Generalization is the mechanism that allows a model to capture, from a dataset, the link between a variable that one seeks to predict and explanatory variables. To measure the ability of a model to generalize, we evaluate the accuracy of its predictions for new data that was not used during the learning phase. We propose here a new method to improve the generalization performance of a model, based on the idea that if, when training a model to predict the available data, we prevent it from learning by heart the available observations one by one, the model will have no choice but to try to understand the mechanisms underlying the problem at hand.

Conditional density estimation is a task that consists in building a predictive model that does not just predict what will happen on average or in the most probable case, but will instead be able to give, for each possible scenario, the probability that this event may occur. This is a more complicated problem than simply predicting the average outcome. We propose here a method to transform this problem into a much simpler one, which consists in predicting in a binary way whether or not an observation comes from the real original dataset, or if it has been artificially manufactured.

Remerciements

En premier lieu, je souhaite exprimer mon immense reconnaissance à mon directeur de thèse, Karim Lounici, et à ma directrice de thèse, Katia Meziani. Katia, à la fin de mon master, tu étais bien la seule à croire en mes chances de réussir une thèse, sans ta persévérance, rien de tout cela n'aurait pu avoir lieu. Karim, tu t'es investi bien au delà de ce que j'aurai pu imaginer. Tous les deux avez su me transmettre votre entrain, votre passion pour la recherche, votre savoir, votre exigence, votre persévérance et votre rigueur.

Je tiens également à remercier tout particulièrement les membres de mon jury. Merci à Arnak Dalayan c'est un grand honneur d'avoir pour président de jury quelqu'un qui incarne si bien ce que doit être un grand chercheur. Merci à Madalina Olteanu et Joseph Salmon d'avoir bien voulu rapporter cette thèse avec autant d'attention, tous vos retours ont eu un impact significatif sur la qualité de celle-ci. Merci aussi bien sûr à Mohamed Hebiri et Martial Hue d'avoir participé en tant qu'examinateur et membre invité respectivement, vos questions très pertinentes ouvrent de nouvelles perspectives.

J'aimerai aussi remercier tous les chercheurs et chercheuses, les membres du laboratoire, les enseignants et enseignantes de Dauphine et les collègues chez Jellyfish que j'ai rencontré tout au long de mon parcours. Merci à Sacha Tsybakov, c'est un immense privilège d'avoir pu avoir des échanges aussi riches avec vous. Merci à Vladimir Koltchinskii, votre venue à Dauphine a été l'occasion pour moi d'assister à ma première présentation académique, mettant d'emblée la barre à un niveau stratosphérique. Merci à Pierre Alquier, Victor-Emmanuel Brunel, Jaouad Mourtada et Evgenii Chzhen pour les conseils et connaissances qu'ils m'ont apporté à l'Ensae et à Porquerolles. Merci à Cristina Buttucea, en particulier pour avoir bien voulu m'inviter à Luminy et à l'Ensae. Merci à Emmanuel Gobet, Eric Moulines et Josselin Garnier pour les échanges et les opportunités qu'ils ont su provoquer au sein de l'équipe Simpas, que ce soit à Luminy, au Maroc ou au Living Room. Merci à Nizar Touzi pour sa bienveillance et sa sagesse, pour les évènements de la chaire Natixis et en tant que référent. Merci à Aymeric Dieuleveut, en particulier pour sa capacité à faire du SGM le moment de retrouvailles et d'ouverture que l'on attend chaque mois. Merci à Rémy Flamary pour l'attention et l'intérêt qu'il a porté à mes travaux et ses remarques qui nous ont permis d'orienter et de préciser notre stratégie de publication. Merci Mahdi, c'est une chance de pouvoir échanger avec quelqu'un qui a de telles connaissances et de telles engagements sur des sujets qui me tiennent à coeur.

Bien sûr, cette thèse n'aurait pas été possible sans Dauphine, expérience intense

qui m'a façonné intellectuellement, m'a donné le goût de l'effort, de l'ambition et de l'exigence. Elle a été jonchée de rencontres académiques et professionnelles qui m'ont beaucoup apporté, notamment Pierre Brugière, Vincent Rivoirard, Tristan Cazenave, Guillaume Legendre, Yann Aït Mokhtar et bien sûr Didier Jeannel. J'y ai également fait de belles amitiés qui perdurent encore aujourd'hui.

Merci à la très solidaire famille des doctorants et doctorantes du CMAP. C'est un réconfort et un régal de pouvoir faire parti de ce groupe et de traverser ensemble nos épreuves respectives. Je ne vais pas m'épancher ici, j'ai déjà eu l'occasion de vous dire tout le bien que je pense de vous et de la dynamique saine et enrichissante qui caractérise la face septentrionale du premier étage. Par respect des us, je vais tout de même remercier nommément (liste alphabétiquement ré-ordonnée et non-exhaustive) Armand, Arthur, Baptiste, Clément, Constantin, Corentin, Guillaume, Grégoire, Jessie, Leïla, Louis, Manon, Margaux, Maxime, Naoufal, Pablo, Quentin, Solange, ... Confer les mots sur vos cartes de soutenance de thèse passées et futures.

Plus personnellement, je veux aussi remercier mes amis et celles et ceux qui m'ont accompagné durant ces années de thèse, rencontrés pendant ou bien avant. En particulier Maître Cahn, Nadège, Côme, le docteur Lajaunie, la famille Dam ; pour leur amitié, leur fidélité, leur générosité et leur entrain toujours renouvelés après tant d'années. Et enfin, merci à ma famille, à mon père, à ma mère et à mon petit frère.

Résumé de la thèse en français

Cette thèse introduit de nouvelles techniques qui exploitent des permutations du vecteur des observations de la variable à expliquer pour améliorer les performances de généralisation dans la tâche de régression et transformer l'estimation de la fonction de densité conditionnelle en un problème de classification binaire. Des justifications théoriques et des benchmarks empiriques sur des jeux de données tabulaires sont proposés pour démontrer l'intérêt de ces techniques, en particulier lorsqu'elles sont combinées avec des réseaux de neurones profonds.

La généralisation est un problème central en l'apprentissage machine. La plupart des modèles prédictifs nécessitent une calibration minutieuse des hyper-paramètres sur un échantillon de validation pour obtenir de bonnes performances de généralisation. Une nouvelle approche qui contourne cette difficulté est présentée. Elle est basée sur une nouvelle mesure du risque de généralisation qui quantifie directement la propension d'un modèle à sur-ajuster les données d'entraînement. Le critère associé, appelé MLR (Muddling Labels Regularization) est évalué sur le jeu de données d'entraînement et permet d'estimer la performance sur le jeu de données test. Pour cela, il utilise des permutations du vecteur des observations de la variable à expliquer pour quantifier la propension d'un modèle à mémoriser la part de bruit contenu dans les données. Pour transformer le critère MLR en une fonction de perte pour les réseaux de neurones profonds, l'opérateur Tikhonov est introduit. Il module la capacité de mémorisation d'un réseau de manière adaptative, différentiable et dépendante des données. En combinant la perte MLR et l'opérateur Tikhonov, on obtient la technique d'apprentissage AdaCap (ADAptative CAPacity control) qui optimise la capacité du réseau afin qu'il puisse apprendre les représentation abstraite de haut niveau correspondant au problème posé plutôt que de mémoriser le jeu de données d'entraînement.

Le problème d'estimation de densité conditionnelle est également traité. Il est à la base de la majorité des tâches d'apprentissage machine, y compris l'apprentissage supervisé et non supervisé ainsi que les modèles génératifs. Une nouvelle méthode, MCD (Marginal Contrastive Discrimination) inspirée du noise contrastive learning est introduite. MCD reformule la tâche initiale en un problème d'apprentissage supervisé qui peut être résolu à l'aide d'un classifieur binaire. Des techniques de construction de jeux de données de contraste basées là encore sur des permutations du vecteur de la variable à expliquer sont également proposées. Elles permettent d'obtenir des jeux de données d'entraînement beaucoup plus grands que le jeu de données initial, et de tirer parti d'observations non-étiquetées et d'observations pour lesquelles on dispose de plusieurs réalisations.

English summary of the thesis

This thesis introduces novel techniques which leverage label permutations to improve generalization performances in the regression task and estimate the conditional density function through binary classification. Theoretical justifications and empirical benchmarks on tabular datasets are provided to demonstrate their effectiveness, especially when combined with deep neural networks.

Generalization is a central problem in Machine Learning. Most prediction methods require careful calibration of hyperparameters usually carried out on a hold-out validation dataset to achieve generalization. We introduce a novel approach to achieve generalization without any data splitting, which is based on a new risk measure which directly quantifies a model's tendency to overfit. The associated criterion, called MLR (Muddling Labels Regularization) is an in-sample metric for out-of-sample performance which leverages randomly generated labels to quantify the propensity of a model to memorize. To transform the MLR criterion into a training loss for deep neural networks, we introduce the Tikhonov operator training scheme, which modulates the memorization capacity of a FFNN in an adaptive, differentiable and data-dependent manner. By combining the MLR loss and the Tikhonov operator we obtain the AdaCap training scheme (ADAptative CAPacity control) which optimizes the capacity of FFNN so it can capture the high-level abstract representations underlying the problem at hand without memorizing the training dataset.

Besides generalization, we also consider the problem of conditional density estimation. This task is at the root of the majority of machine learning tasks, including supervised and unsupervised learning or generative modeling. We introduce a new method, MCD (Marginal Contrastive Discrimination) inspired by contrastive learning. MCD reformulates the initial task into a problem of supervised learning which can be solved with any binary classifier. We present construction techniques based on label permutations to produce a contrast dataset with far more observations than in the original dataset and take advantage of unlabeled observations and more than one target value per observation.

Contents

Dédicace 1
Résumé pour le grand public 3
Summary for the general public 4
Remerciements 5
Résumé de la thèse en français 7
English summary of the thesis 8

Contents **9**

1 Introduction générale **13**
1.1 Contexte et Problématique 14
1.1.1 Modèle statistique et données réelles 14
1.1.2 Le problème de Généralisation 17
1.1.3 Calibration des hyper-paramètres. 21
1.1.4 Sur-ajustement et mémorisation 24
1.2 Critère `MLR` : un critère généralisant 26
1.2.1 Intuition du critère `MLR`. 26
1.2.2 Forme close et critère `MLR`. 27
1.2.3 Étude théorique de `MLR`. 28
1.2.4 Nouveaux modèles. 28
1.3 `AdaCap` : Extension aux réseaux de neurones 29
1.3.1 État de l'art non exhaustif. 30
1.3.2 L'opérateur Tikhonov. 31
1.3.3 La méthode AdaCap. 33
1.4 Estimation de densité conditionnelle : la méthode MCD 34
1.4.1 Estimation de densité conditionnelle : une tâche centrale. . . 34
1.4.2 Travaux connexes. 35
1.4.3 La méthode MCD. 37

2 Introduction aux réseaux de neurones **39**
2.1 Introduction 40
2.2 De la régression linéaire aux `DNN` 43
2.2.1 Du modèle simple au neurone 43
2.2.2 Le modèle Perceptron Multicouche 46
2.3 Phase d'apprentissage 49
2.3.1 Fonction de perte et objectif 49
2.3.2 Calcul des dérivées partielles et Rétro-Propagation 52
2.3.3 Descente de gradient 56
2.3.4 Optimizers et learning rate schedulers 59
2.4 Techniques d'apprentissage et de régularisation 72
2.4.1 Spécificités du Deep Learning 73

2.4.2 Contrôle des normes des poids et des activations 75
2.4.3 Régularisation explicite 78
2.5 Conclusion 81

3 MLR: Muddling Labels for Regularization 83
3.1 Introduction 84
3.2 The Muddling Label Regularization Criterion 87
3.3 Novel procedures 91
3.4 Conclusion 101
3.5 Appendix 104
3.5.1 Proof of Theorem 1 104
3.5.2 Proof of Lemma 1 107
3.5.3 Fact 1 108
3.5.4 Proposition 1 109
3.5.5 Lemma 2 112

4 AdaCap: Adaptive Capacity control for Feed-Forward Neural Networks 114
4.1 Introduction 115
4.2 The MLR loss 120
4.3 The AdaCap method to train DNN 121
4.4 Experiments 125
4.4.1 Implementation details 125
4.4.2 Tabular Data benchmark results 128
4.4.3 Ablation, Learning Dynamic, Dependency study 131
4.5 Conclusion 132
4.6 Appendix 134
4.6.1 Additional experiments 134
4.6.2 Training protocol 134
4.6.3 Further discussion of existing works comparing DNN to other models on TD 135
4.6.4 Benchmark description 136
4.6.5 Hardware Details 140

5 MCD: Marginal Contrastive Discrimination 141
5.1 Introduction 142
5.1.1 Related work 143
5.1.2 Our contributions 146
5.2 Marginal Contrastive Discrimination 147
5.2.1 Setting 147
5.2.2 Contrast function 149

5.2.3 Marginal Discrimination Conditions 150
5.3 Contrast datasets construction 151
5.3.1 Classical Dataset (Framework 1) 151
5.3.2 Additional Marginal Data (Framework 2) 152
5.4 Experiments 154
5.4.1 Method implementation 154
5.4.2 Other benchmarked methods and Application Programming Interface (API) 157
5.4.3 Estimation of theoretical models 158
5.4.4 Results on density models 160
5.4.5 Real-world datasets 164
5.5 Ablation 166
5.6 Proofs and dataset constructions 169
5.6.1 Proof Fact 1 169
5.6.2 Proof Proposition 1 170
5.6.3 Proof Theorem 1 and Construction in the *i.i.d.* case 171
5.6.4 Proof Theorem 2 and Construction in the *i.d.* case 172
5.6.5 Proof Theorem 3 and *i.i.d.* Construction in the Additional data setting 173
5.6.6 Proof Theorem 4 and and *i.d.* Construction in the Additional data setting 177
5.6.7 Proof Theorem 5 and Construction in the non-independent case 179
5.7 Appendix 179
5.7.1 Method to rescale estimated densities 179
5.7.2 Exhaustive experimental results 180

Références / References **182**

Chapitre 1
Introduction générale

What is now proved was once, only imagin'd.

William Blake, Proverbs of Hell

1.1 Contexte et Problématique

1.1.1 Modèle statistique et données réelles

Les quinze dernières années ont vu déferler la vague du *Big Data*, et dans son sillage celles de la *data-science* et du *machine learning*. Si ces anglicismes ont infusé dans les discours en entreprise, dans les média et dans les administrations, pour beaucoup encore, ces concepts demeurent flous pour ne pas dire brumeux. La révolution numérique, bien que son impact sur l'économie et la société nous affecte tous, reste le fait de peu. Avec l'émergence de l'I.A. et du *Deep Learning* comme nouvelles figures de proue médiatiques, l'écart semble même se creuser entre ceux qui "baignent dedans", les chercheurs et praticiens qui pensent, développent et utilisent les méthodes de statistiques mathématiques, d'apprentissage machine et de gestion des données ; et tous ceux se sentant submergés par le flot de nouvelles avancées théoriques, de nouveaux logiciels, de nouveaux langages de programmation et de nouveaux algorithmes. Il est pourtant crucial de démocratiser ces techniques qui, grâce aux données, permettent de faire des liens, d'anticiper, d'optimiser, d'agir plus vite et mieux. Si certaines réserves du grand public et des décideurs sur ces outils paraissent légitimes (systèmes boîtes noires, modèles propriétaires, algorithmes biaisés...), elles ne doivent pas occulter tout les efforts de la communauté pour rendre les méthodes de statistiques mathématiques et d'apprentissage machine plus accessibles, plus équitables, plus précises, plus rapides, plus robustes et plus facilement interprétables. C'est dans ce cadre général que s'inscrit cette thèse, centrée en particulier sur la généralisation et l'estimation de densité conditionnelle sur les jeux de données tabulaires.

Données tabulaires

➢ Le besoin de traiter des données tabulaires se fait sentir dans de nombreux domaines tels que les sciences des matériaux ([61]), la médecine ([166]), la publicité en ligne ([225, 78]), la finance ([33]). Travailler avec des données tabulaires présente plusieurs avantages : à l'origine, c'était toujours dans ce format que se présentait les données utilisées en apprentissage machine et en statistiques, ce qui explique à la fois le grand nombre et la richesse des outils disponibles, tant en pratique qu'en théorie. De plus, si les données tabulaires ne sont pas les

plus "parlantes" pour un utilisateur néophyte, elles sont les plus faciles à traiter d'un point de vue informatique. Enfin, il existe une façon simple de formaliser mathématiquement, sous forme de matrices et de vecteurs, les données tabulaires.

Modèle statistique.
➢ Soit (X, Y) un couple de variables aléatoires à valeurs dans $\mathcal{X} \times \mathcal{Y}$, tel que $\mathcal{X} \subset \mathbb{R}^p$ avec $p \in \mathbb{N}_*$ et $\mathcal{Y} \subset \mathbb{R}$. On suppose que les variables X et Y ne sont pas indépendantes et qu'il existe un lien inconnu f, tel que $Y = f(X)$, que nous voulons reconstruire. On dit alors que les composantes $X_1, \cdots, X_p$ de X sont les **variables explicatives**, aussi appelées *"features"*. La variable cible Y est appelée la **variable à expliquer**, aussi appelées *"target variable"*.

Données disponibles.
➢ Afin d'identifier le lien entre la variable cible Y et X, on dispose un **jeu de données d'entraînement**[1], $\mathcal{D}_n^{X,Y} = \{(\mathbf{x}_i, y_i)\}_{i=1,\cdots,n}$, correspondant à la réalisation de $n \in \mathbb{N}_*$ couples de variables aléatoires $\{(X_i, Y_i)\}_{i=1,\cdots,n}$ *i.i.d* de même loi $P_{X,Y}$. Ces données sont structurées de la façon suivante :

- On appelle $\mathbf{X}$ la matrice ***design*** des observations $\{\mathbf{x}_i\}_i$

$$\mathbf{X} = [x_{i,j}]_{\substack{i=1,\cdots,n \\ j=1,\cdots,p}} = \begin{pmatrix} \mathbf{x}_1^\top \\ \vdots \\ \mathbf{x}_n^\top \end{pmatrix} = [\mathbf{X}_1, \cdots, \mathbf{X}_p]$$

  où $\mathbf{x}_i$ est l'**observation** i telle que $\mathbf{x}_i \in \mathcal{X} \subset \mathbb{R}^p$ et X_j est la variable explicative telle que $X_j \in \mathbb{R}^n$.

- On appelle $\mathbf{Y}$ le vecteur des observations $\{y_i\}_i$

$$\mathbf{Y} = (y_1, \cdots, y_n)^\top$$

  où y_i, la variable à expliquer, est telle que $y_i \in \mathcal{Y} \subset \mathbb{R}$.

La problématique des petits jeux de données.
➢ Dans la définition donnée précédemment, n correspond au nombre d'observations, p au nombre de variables explicatives. Si en mathématiques, on peut sans risque choisir, $n \in \mathbb{N}_*$, en pratique, vivons dans un monde fini, où les

[1] Le type peut varier (images, séries temporelles, signaux sonores, etc.). Dans le cadre de cette thèse, on se concentrera sur les données tabulaires, aussi appelées données structurées.

ressources informatiques, tant en mémoire qu'en temps de calcul seront toujours limitées. Dès lors, même à l'ère du *Big Data*, le besoin de traiter les jeux de données de petite taille reste d'actualité. C'est le cas par exemple pour les sciences de l'environnement ([34]) ou en médecine ([166]), pour ne citer que ces deux exemples. En effet, la collecte de données peut être coûteuse, voire impossible de par la nature de la tâche à accomplir (contraintes éthiques, légales, financières...). Par ailleurs, dans bien des contextes (*e.g.* gestion de crise, tests A/B en *web design*), les données sont collectées au fil du temps, et dès lors, traiter des jeux de données de petite taille est nécessaire pour pouvoir agir au plus tôt.

La tâche de régression.
➢ Le lien f entre X et Y peut être de différentes natures, la méthode pour l'identifier également. Une première façon de faire est d'estimer l'espérance conditionnelle de Y en fonction de X, on parle d'apprentissage supervisé. On cherche à estimer la fonction de régression $\mathbb{E}[Y \mid X]$:

$$f: \begin{array}{ccc} \mathcal{X} & \longrightarrow & \mathcal{Y} \\ x & \longmapsto & \mathbb{E}[Y \mid X = x] \end{array}$$

On construit un estimateur $\widehat{f}$ à partir du jeu de données d'entraînement $\mathcal{D}_n^{X,Y}$, en utilisant une **méthode d'apprentissage**. Lorsque l'on parle d'apprentissage, on attend de le modèle, une fois entraîné, soit capable de prédire correctement sur des données jamais vues ("*unseen data*"). On parle de **généralisation**, *i.e.* la capacité du modèle à pouvoir effectuer des prédictions robustes sur de nouvelles données. Un modèle avec des performances de généralisation médiocres serait par exemple un modèle qui "colle" trop aux données d'entraînement, on parle de sur-ajustement (*overfitting*) : le modèle n'est capable que de reproduire et de ressortir des choses déjà vues et il n'est pas capable d'extrapoler les connaissances, c'est à dire de généraliser. La **généralisation** est un problème central dans de nombreuses communautés, étudié tant d'un point de vue théorique que pratique.

La tâche d'estimation de densité conditionnelle
➢ Il est parfois plus intéressant d'estimer la fonction de densité conditionnelle elle-même plutôt que la fonction de régression (section 1.4) :

$$\begin{aligned} \mathcal{X} \times \mathcal{Y} &\longrightarrow \mathbb{R}_+ \\ (x, y) &\longmapsto f_{Y \mid X = x}(y) \end{aligned}$$

1.1.2 Le problème de Généralisation

A. Évaluation empirique de la performance de Généralisation

Soit $\widehat{f}$ un estimateur de la fonction f construit à partir du jeu de données $\mathcal{D}_n^{X,Y}$. Il n'est possible d'évaluer la performance de généralisation de cet estimateur qu'empiriquement. Soit $\widetilde{\mathcal{D}}_{\widetilde{n}}^{X,Y} = \{(\widetilde{\mathbf{x}}_i, \widetilde{y}_i)\}_{i=1,\cdots,\widetilde{n}}$ un **jeu de données test** sur lequel on évaluera la performance de généralisation de $\widehat{f}$ et qui n'aura pas servit à la construction de $\widehat{f}$. Le jeu de données test correspond à la réalisation de $\widetilde{n} \in \mathbb{N}_*$ couples de variables aléatoires $\left\{(\widetilde{X}_i, \widetilde{Y}_i)\right\}_{i=1,\cdots,\widetilde{n}}$ *i.i.d* de même loi $P_{X,Y}$. Ces données sont structurées de la même façon que $\mathcal{D}_n^{X,Y}$, et on note la matrice des nouvelles observations et le vecteur des valeurs à prédire respectivement

$$\widetilde{\mathbf{X}} = \begin{pmatrix} \widetilde{\mathbf{x}}_1^\top \\ \vdots \\ \widetilde{\mathbf{x}}_{\widetilde{n}}^\top \end{pmatrix} \quad \text{et} \quad \widetilde{\mathbf{Y}} = (\widetilde{y}_1, \cdots, \widetilde{y}_{\widetilde{n}})$$

Par abus de notation, on notera pour $\widehat{f}$ un estimateur de la fonction f construit à partir du jeu de données $\mathcal{D}_n^{X,Y}$:

$$\widehat{f}(\widetilde{\mathbf{X}}) = \left(\widehat{f}(\widetilde{\mathbf{x}}_i)\right)_{i=1,\cdots,\widetilde{n}}$$

le vecteur des prédictions de cet estimateur sur le jeu de données test. Pour mesurer la qualité de cette prédiction, on utilise un critère de performance : le R^2-score, aussi appelé coefficient de détermination.

Critère 1. ***[R²-score]***

$$\begin{array}{rcl} \mathbb{R}^{\widetilde{n}} \times \mathbb{R}^{\widetilde{n}} & \longrightarrow & (-\infty, 1] \\ (\boldsymbol{y}, \boldsymbol{y}') & \longmapsto & R^2(\boldsymbol{y}, \boldsymbol{y}') = 1 - \frac{\|\boldsymbol{y}-\boldsymbol{y}'\|_2^2}{\|\boldsymbol{y}-\bar{y}\mathbf{1}_{\widetilde{n}}\|_2^2+\epsilon} \end{array}$$

Ici $\epsilon \in \mathbb{R}_{+*}$ est une constante de stabilité numérique, $\mathbf{1}_{\widetilde{n}}$ est le vecteur unitaire de taille $\widetilde{n}$ et $\bar{y} = \frac{1}{\widetilde{n}} \sum_{i=1}^{\widetilde{n}} y_i$ la moyenne empirique de $\boldsymbol{y}$.

Remarques

- Plus la valeur $\mathrm{R}^2(\widetilde{\mathbf{Y}}, \widehat{f}(\widetilde{\mathbf{X}}))$ du R^2-score de $\widehat{f}$ est élevé meilleure est la prédiction.
- Le R^2-score peut prendre des valeurs négatives.

- Un R^2-score négatif signifie que notre estimateur est moins bon que l'estimateur réduit à l'*intercept*, *i.e.* celui qui prédit $\bar{y}$ quelque soit la valeur de $\mathbf{x}$.

- Notons qu'il existe d'autres critères pour estimer la performance de généralisation d'un modèle. Citons notamment, les critères *SURE* ([194]), *AIC* ([4]) et le C_p de *Mallows* ([131]), ceux-ci sont cependant beaucoup moins utilisés en pratique.

[Performance de Généralisation de f]

$$\mathsf{Gen}\,(f,\ \widetilde{\mathcal{D}}^{X,Y}_{\widetilde{n}}) := \mathrm{R}^2(\widetilde{\mathbf{Y}}, f(\widetilde{\mathbf{X}})) = 1 - \frac{\|\widetilde{\mathbf{y}} - f(\widetilde{\mathbf{X}})\|_2^2}{\|\widetilde{\mathbf{y}} - \bar{\bar{\mathbf{y}}}\mathbf{1}_{\widetilde{n}}\|_2^2 + \epsilon}$$

où $\bar{\bar{\mathbf{y}}} = \frac{1}{\widetilde{n}} \sum_{i=1}^{\widetilde{n}} \widetilde{y}_i$.

La performance de généralisation évaluée sur le jeu de données d'évaluation $\widetilde{\mathcal{D}}^{X,Y}_{\widetilde{n}}$ de $\widehat{f}$, l'estimateur construit à partir du jeu de données d'entraînement $\mathcal{D}^{X,Y}_n$, est $\mathsf{Gen}\,(\widehat{f}, \widetilde{\mathcal{D}}^{X,Y}_{\widetilde{n}})$.

B. Estimation paramétrique

Dans cette thèse, on on se place dans un cadre paramétrique, c.à.d. que l'on considère que "l'*oracle*" f^*, qui maximise la performance de généralisation appartient à une classe paramétrique de fonction. Soit $\mathcal{F}_\omega$ une classe paramétrée par $\omega \in \mathcal{W} \subset \mathbb{R}^D$, $D \in \mathbb{N}^*$ telle que $f^* \in \mathcal{F}_\omega$ et

$$\mathcal{F}_\omega = \{f_\omega\ :\ \omega \in \mathcal{W} \subset \mathbb{R}^D\}.$$

On cherche alors dans cette classe, $f^* := f_{\omega^*}$ qui maximise la performance de généralisation :

$$\omega^* = \underset{\omega \in \mathcal{W}}{\arg\max}\,\mathsf{Gen}\,(f_\omega,\ \widetilde{\mathcal{D}}^{X,Y}_{\widetilde{n}}),$$

où $\widetilde{\mathcal{D}}^{X,Y}_{\widetilde{n}}$ est le jeu de données test, données non utilisées lors de la phase d'entraînement du paramètre ω.

Apprentissage des paramètres

➢ Lors de la phase d'apprentissage, on dispose du jeu de données d'entraînement $\mathcal{D}_n^{X,Y}$. On cherche à construire un estimateur $\widehat{f} = f_{\widehat{\omega}(\mathcal{D}_n^{X,Y})}$ tel que pour tout observation $(\mathbf{x}_i, y_i)$ de $\mathcal{D}_n^{X,Y}$, l'estimation $\widehat{f}(\mathbf{x}_i)$ soit proche de y_i. Pour cela, on optimise un critère. Le critère usuel est le MSE (*Mean Squared Errors*).

Critère 2. ***[MSE]***

$$\begin{array}{rcl} \mathbb{R}^n \times \mathbb{R}^n & \longrightarrow & \mathbb{R}_+ \\ (\boldsymbol{y}, \boldsymbol{y}') & \longmapsto & \mathtt{MSE} := \frac{1}{n}\|\boldsymbol{y} - \boldsymbol{y}'\|_2^2 = \|\boldsymbol{y} - \boldsymbol{y}'\|_n^2 \end{array}$$

Notons que minimiser le MSE sur le jeu de données d'entraînement revient à "coller" ("*fit the data*") au mieux à ces données. Un petit MSE correspond à une bonne performance d'entraînement : le modèle estimé explique (colle/"***fit***") bien les données.

[Performance d'Entraînement de f_ω]

$$\mathsf{Fit}\,(\omega, \mathcal{D}_n^{X,Y}) := \mathtt{MSE}(\mathbf{Y}, f_\omega(\mathbf{X})) = \frac{1}{n}\sum_{i=1}^{n}(y_i - f_\omega(\mathbf{x}_i))^2 = \|\mathbf{y} - f_\omega(\mathbf{X})\|_n^2$$

Une bonne performance d'entraînement n'est pas la garantie d'une bonne performance de généralisation. En effet, l'estimateur $f_{\widehat{\omega}}$ qui fit le mieux les données du jeu d'entraînement, *i.e.*

$$\widehat{\omega}(\mathcal{D}_n^{X,Y}) := \underset{\omega \in \mathcal{W}}{\arg\min}\, \mathsf{Fit}\,(\omega,\ \mathcal{D}_n^{X,Y}),$$

ne donne pas nécessairement la meilleure performance de généralisation qui est évaluée sur le jeu de données test. Il est possible que le paramètre $\widehat{\omega}$ qui minimise Fit soit différent de ω^*. Par exemple, on peut avoir

$$\begin{aligned} & \mathsf{Fit}\,(\omega^*, \mathcal{D}_n^{X,Y}) > \min_{\omega \in \mathcal{W}} \mathsf{Fit}\,(\omega, \mathcal{D}_n^{X,Y}), \\ \text{et} \quad & \mathsf{Gen}\,(f_{\widehat{\omega}(\mathcal{D}_n^{X,Y})}, \widetilde{\mathcal{D}}_{\widetilde{n}}^{X,Y}) < \max_{\omega \in \mathcal{W}} \mathsf{Gen}\,(\omega, \widetilde{\mathcal{D}}_{\widetilde{n}}^{X,Y}). \end{aligned}$$

On appelle ce phénomène le **sur-ajustement** (*overfitting*).

C. Sur-apprentissage et régularisation.

Avoir un MSE nul sur le jeu de données d'entraînement est signe que le modèle *overfit*. Il n'est pas capable de généraliser. Pour parer à ce problème, il existe deux stratégies usuelles simples.

- Pénaliser les grands modèles en ajoutant à la fonction de perte une pénalité P qui dépend de ω. C'est le cas par exemple pour les estimateurs Ridge ([46, 95]) et Lasso ([199]) (pénalités L_2 et L_1 respectivement).

- Restreindre l'espace $\mathcal{W}$ des paramètres considérés à l'aide d'une contrainte C. C'est le cas par exemple des arbres de décision ([29]) et des forêts aléatoires ([94, 30]) (nombre maximal de feuilles, profondeur maximale de l'arbre).

La performance Fit n'est alors pas un objectif approprié et il convient de le modifier. Pour se faire, il existe deux stratégies simples :

Régularisation explicite.
➢ Pour moduler la force de la pénalité et/ou de la contrainte, on introduit l'**hyper-paramètre** $\theta \in \Theta \subset \mathbb{R}^{D'}$, $D' \in \mathbb{N}_*$. On note respectivement $\mathsf{P}(\theta, \omega)$ et $\mathsf{C}(\theta, \omega) = 0$ la pénalité et la contrainte associées au paramètre ω et à l'hyper-paramètre θ. Ainsi, le paramètre $\widehat{\omega}$ obtenu à l'issu de l'apprentissage dépend de θ :

$$\widehat{\omega}(\theta, \mathcal{D}_n^{X,Y}) := \underset{\omega \in \mathcal{W}, \mathsf{C}(\theta,\omega)=0}{\arg\min} \mathsf{Fit}(\omega, \mathcal{D}_n^{X,Y}) + \mathsf{P}(\theta, \omega)$$

L'introduction de θ, le paramètre de **régularisation**, permet de modifier l'estimateur $f_{\widehat{\omega}(\theta, \mathcal{D}_n^{X,Y})}$ pour améliorer la performance de généralisation, *i.e.* d'estimer $\theta^* \in \Theta$, l'hyper-paramètre qui maximise la généralisation (section 1.1.3).

Régularisation implicite et Méthodes ensemblistes.
➢ Il existe d'autre techniques au delà de la régularisation pour améliorer les performances de généralisation. On ne présente pas ici la régularisation dite "implicite" (par opposition à la régularisation "explicite" que nous venons de présenter), elle sera détaillée dans le Chapitre 2. L'autre approche très populaire est l'usage de méthodes ensemblistes. Elle consiste à entraîner un ensemble d'estimateurs (appelés *weak learners*), à l'aide par exemple d'une méthode de *boot-strap*, puis à construire un méta-estimateur qui opère une moyenne pondérée des prédictions de ces *weak learners*. Parmi les méthodes ensemblistes les plus utilisées en pratique, on peut citer les Forêts Aléatoires ([94, 30]), les Splines (MARS) ([66]) et les méthodes de *Boosting* ([65]) telles que XGB ([40]) et **CatBoost** ([160]). Soulignons que les méthodes ensemblistes ont le défaut de nécessiter l'entraînement de plusieurs estimateurs dont le nombre est généralement grand.

1.1.3 Calibration des hyper-paramètres.

Performance sur l'échantillon de validation.
➢ L'introduction de l'hyper-paramètre de régularisation θ doit être calibré de façon à maximiser la généralisation, *i.e.* que l'on cherche

$$\begin{aligned} \widetilde{\theta} &:= \arg\max_{\theta\in\Theta} \mathsf{Gen}\,(f_{\widehat{\omega}(\theta,\mathcal{D}_n^{X,Y})},\ \widetilde{\mathcal{D}}_{\widetilde{n}}^{X,Y}) \\ \text{tel que} \quad \widehat{\omega}(\theta,\mathcal{D}_n^{X,Y}) &= \arg\min_{\omega\in\mathcal{W},\mathsf{C}\,(\theta,\omega)=0} \mathsf{Fit}\,(\omega,\ \mathcal{D}_n^{X,Y}) + \mathsf{P}\,(\theta,\omega) \end{aligned} \tag{1.1}$$

Maximiser la généralisation nécessite un jeu de données n'ayant pas été utilisé pendant la phase d'apprentissage. Le jeu de données test ne devant pas être utilisé, une possibilité est de découper le jeu de données d'entraînement $\mathcal{D}_n^{X,Y}$ en deux :

- Un premier sous échantillon de taille n_{train} pour l'entraînement, *i.e.* la construction de $f_{\widehat{\omega}(\theta,\mathcal{D}_n^{X,Y})}$. On appellera cet échantillon jeu de données *train*.
- Un second sous échantillon de taille $n_{\text{valid}} = n - n_{\text{train}}$ qui jouera le rôle de jeu de données test. On appellera cet échantillon jeu de données de *validation*.

$$\mathcal{D}_n^{X,Y} = \underbrace{\mathcal{D}_{|n_{\text{train}}}^{X,Y}}_{n_{\text{train}}\text{ obs.}} \cup \underbrace{\mathcal{D}_{|n_{\text{valid}}}^{X,Y}}_{n_{\text{valid}}\text{ obs.}}$$

Les observations de $\mathcal{D}_n^{X,Y}$ étant *i.i.d.*, les jeux de données $\mathcal{D}_{|n_{\text{train}}}^{X,Y}$ et $\mathcal{D}_{|n_{\text{valid}}}^{X,Y}$ sont indépendants. On note $\mathbf{X}_{\text{train}}$ et $\mathbf{Y}_{\text{train}}$ respectivement la matrice $\mathbf{X}$ et le vecteur $\mathbf{Y}$ restreints au n_{train} observations. De même, notons $\mathbf{X}_{\text{valid}}$ et $\mathbf{Y}_{\text{valid}}$ respectivement la matrice $\mathbf{X}$ et le vecteur $\mathbf{Y}$ restreints au n_{valid} observations. Cela nous permet de définir la performance de validation $\widehat{\mathsf{Gen}}$ de la façon suivante.

[Performance de validation du paramètre $\theta \in \Theta$]

$$\widehat{\mathsf{Gen}}\,(\theta,\mathcal{D}_{|n_{\text{valid}}}^{X,Y}) = \mathsf{Gen}\,(f_\theta,\ \mathcal{D}_{|n_{\text{valid}}}^{X,Y}) = \mathtt{R}^2(\mathbf{Y}_{\text{valid}}, f_\theta(\mathbf{X}_{\text{valid}}))$$

où $f_\theta = f_{\widehat{\omega}(\theta,\mathcal{D}_{|n_{\text{train}}}^{X,Y})}$ est construit à partir du jeu de données *train* $\mathcal{D}_{|n_{\text{train}}}^{X,Y}$.

Cette solution consiste au final à diviser l'ensemble de données original en trois sous-échantillons : le *train*, la *validation* et le *test* pour évaluer la performance de généralisation du modèle final construit. Les données pouvant être rares et coûteuses, une meilleure solution est la Validation Croisée sur l'échantillon d'entraînement (le *train*) qui permet d'éviter l'échantillon de validation.

Validation Croisée.
➢ Pour $K \in \mathbb{N}_*$, $K \leq n$, cette méthode consiste à partitionner le jeu de données initial d'entraînement, $\mathcal{D}_n^{X,Y}$, en K sous-échantillons de taille n_k, avec $k = 1, \cdots, K$:

$$\mathcal{D}_n^{X,Y} = \bigcup_{k=1}^{K} \mathcal{D}_{|\mathrm{n}_k}^{X,Y}$$

La **performance de Validation Croisée** est alors définie comme la moyenne des K performances de validation calculées sur les échantillons de validations $\mathcal{D}_{|\mathrm{n}_k}^{X,Y}$. Pour chaque découpage $k = 1, \cdots, K$, le jeu de données $\mathcal{D}_{|\mathrm{n}_k}^{X,Y}$ joue le rôle de l'échantillon de *validation* et les $K - 1$ autres jeux de données, $\mathcal{D}_{|n_{\mathrm{train}}}^{X,Y} \setminus \mathcal{D}_{|\mathrm{n}_k}^{X,Y}$, forment le *train*.

[Performance de Validation Croisée de l'hyper-paramètre $\theta \in \Theta$]

$$K\text{-}\mathsf{CV}\,(\theta, \mathcal{D}_n^{X,Y}) = \frac{1}{K}\sum_{k=1}^{K} \widehat{\mathsf{Gen}}\,(\theta, \mathcal{D}_{|\mathrm{n}_k}^{X,Y}) = \frac{1}{K}\sum_{k=1}^{K} \mathsf{Gen}\,(f_\theta,\ \mathcal{D}_{|\mathrm{n}_k}^{X,Y})$$

où $f_\theta := f_{\widehat{\omega}(\theta, \mathcal{D}_{|n_{\mathrm{train}}}^{X,Y} \setminus \mathcal{D}_{|\mathrm{n}_k}^{X,Y})}$ est ici construit à partir des données $\left(\mathcal{D}_{|n_{\mathrm{train}}}^{X,Y} \setminus \mathcal{D}_{|\mathrm{n}_k}^{X,Y}\right)$ qui forment le *train* à l'itération k.

L'hyper-paramètre θ optimal, associé au jeu de données $\mathcal{D}_n^{X,Y}$, est alors obtenu en maximisant la performance de Validation Croisée K-$\mathsf{CV}\,(\theta, \mathcal{D}_n^{X,Y})$:

$$\widehat{\theta}_{\mathsf{CV}}^{K} := \widehat{\theta}_{\mathsf{CV}}^{K}(\mathcal{D}_n^{X,Y}) = \arg\max_{\theta \in \Theta} K\text{-}\mathsf{CV}\,(\theta, \mathcal{D}_n^{X,Y})$$

A l'issu de cette calibration de θ, on effectue un dernier entraînement sur le jeu de données entier $\mathcal{D}_n^{X,Y}$, pour obtenir notre estimateur final $\widehat{f}_{\mathsf{CV}}$:

$$\widehat{f}_{\mathsf{CV}} \quad := \quad f_{\widehat{\omega}(\widehat{\theta}_{\mathsf{CV}}^{K}, \mathcal{D}_n^{X,Y})}. \tag{1.2}$$

L'hyper-paramètre $\widehat{\theta}_{\mathsf{CV}}^{K}(\mathcal{D}_n^{X,Y})$, solution du problème d'optimisation (1.2), dépend de $\{\widehat{\omega}(\theta, \mathcal{D}_{|\mathrm{n}_k}^{X,Y})\}_{k=1,\cdots,K}$, qui sont eux-même solutions de problèmes d'optimisation. On parle d'un problème d'**optimisation Bi-niveau**. En effet, pour chaque candidat $\theta \in \Theta$ dont on veut évaluer la performance de Validation Croisée, il faut effectuer K entraînements, c'est à dire résoudre K problèmes d'optimisations (1.1) pour obtenir les paramètres $\widehat{\omega}(\theta, \mathcal{D}_{|\mathrm{n}_k}^{X,Y})$. On ne dispose pas toujours d'une forme close de ces paramètres, mais des méthodes efficaces existent pour les calibrer : **HPO** (*Hyper-parameter Optimisation*). Ci- dessous, une liste non exhaustive de méthodes d'**HPO** *Black-Box* ([32]).

HPO *Black-Box*.
➢ **Grid-Search.** La méthode la plus élémentaire est *Grid-search*, qui consiste à discrétiser l'espace Θ pour construire une grille de dimension D'. Pour chaque valeur de θ sur cette grille, on résout le problème d'optimisation (1.1) pour obtenir $\widehat{\omega}(\theta, \mathcal{D}^{X,Y}_{|n_{\text{train}}})$. On évalue ensuite $\widehat{\mathsf{Gen}}\,(\theta, \mathcal{D}^{X,Y}_{|n_{\text{valid}}})$. Après avoir parcouru la grille, on choisit la valeur θ qui maximise $\widehat{\mathsf{Gen}}\,(\theta, \mathcal{D}^{X,Y}_{|n_{\text{valid}}})$. L'inconvénient majeur de la méthode *Grid-search* est que si la grille est de largeur $\ell \in \mathbb{N}$ dans chaque direction, avec $\ell \geq 2$, il faut résoudre le problème d'optimisation (1.1), $\ell^{D'}$ fois.

➢**Random Search** ([18]). Une façon plus parcimonieuse de parcourir l'espace Θ est d'utiliser la méthode *Random search*. Elle consiste à considérer θ comme une variable aléatoire suivant une loi de probabilité sur Θ que l'on choisit. On tire alors un échantillon de façon indépendantes $\theta_1, \cdots, \theta_r$ suivant cette loi. Pour tout $i = 1, \cdots, r$, on résout le problème d'optimisation (1.1), puis à l'aide des $\widehat{\omega}(\theta_i, \mathcal{D}^{X,Y}_{|n_{\text{train}}})$ on évalue $\widehat{\mathsf{Gen}}\,(\theta_i, \mathcal{D}^{X,Y}_{|n_{\text{valid}}})$. Suite à ces r évaluations, on choisit la valeur θ_i qui maximise $\widehat{\mathsf{Gen}}\,(\theta_i, \mathcal{D}^{X,Y}_{|n_{\text{valid}}})$. Cette méthode, comme la *Grid-search*, a l'avantage d'être parallélisable. L'inconvénient est que l'on ne peut ni affiner notre recherche pour se concentrer sur les parties de Θ qui semblent les plus prometteuses, ni prioriser les parties de Θ inexplorées.

➢ **Méthodes Bayésiennes et Population-Based.** On procède ici de façon séquentielle. On prend en compte à l'itération i, les évaluations précédentes $(\widehat{\mathsf{Gen}}\,(\theta_1, \mathcal{D}^{X,Y}_{|n_{\text{valid}}}), \cdots, \widehat{\mathsf{Gen}}\,(\theta_{r-1}, \mathcal{D}^{X,Y}_{|n_{\text{valid}}}))$, ce qui permet alors de choisir d' **explorer** les parties de Θ donnant de meilleures performances de Validation. On dit que l'on résout un problème d'arbitrage **exploration/exploitation**. Les méthodes d'optimisation dites "bayésiennes" ([138, 191, 198]) fonctionnent sur ce principe. Elles sont très efficaces lorsque Θ est un espace continu de faible dimension, mais leur coût en temps de calcul croît exponentiellement avec la dimension de l'espace parcouru. Pour les problèmes d'optimisation dans des espaces discontinus de grande dimension, les méthodes dites *population-based*, telles que les algorithmes génétiques sont généralement les plus appropriées ([38, 168, 146, 129, 128]).

Toutes les méthodes *Black-Box* que l'on vient de présenter sont très coûteuses en temps de calcul dès que l'espace des hyper-paramètres Θ est même modérément grand. Cela nécessite de réduire la taille de l'échantillon d'entraînement ($n_{\text{train}} < n$) pour pouvoir estimer la performance de généralisation sur un échantillon de *validation*.

En comparaison, les méthodes d'optimisation *Gradient-Based* utilisées en apprentissage profond que l'on va présenter dans le Chapitre 2 sont très efficaces,

même lorsque l'espace considéré est de très grande dimension et la fonction optimisée fortement non convexe.

Dès lors il semble judicieux d'une part de construire un critère pour estimer la performance de généralisation qui **ne dépende pas d'un échantillon de** *validation*, et d'autre part de proposer des estimateurs paramétriques pour lesquels ce critère généralisant peut être optimisé par une méthode *Gradient-Based*. C'est le propos du Chapitre 3, où l'on présentera un nouveau critère généralisant appelé `MLR`.
Le Chapitre 4, est dédié à `AdaCap`, une nouvelle méthode basée sur `MLR` pour entraîner des *Feed-Forward Neural Networks*. `AdaCap` permet d'améliorer significativement les performances de généralisation.
Dans le Chapitre 5, on s'intéresse à l'estimation de densité conditionnelle. Une nouvelle méthode (**MCD**), basée sur les mêmes intuitions que le critère `MLR`, donne des avancées majeures dans ce domaine.

1.1.4 Sur-ajustement et mémorisation

Afin de définir la notion de "*critère généralisant*", introduisons la notion de **mémorisation** ([7]) utilisée en apprentissage machine, et plus particulièrement en apprentissage profond. La mémorisation est la capacité à prédire fidèlement les observations du jeu de données d'entraînement lorsque la variable à expliquer est indépendante des variables explicatives. À l'instar d'un être humain capable d'apprendre par cœur un numéro de téléphone à l'aide d'une méthode mnémotechnique, mais incapable d'expliquer pourquoi l'opérateur téléphonique a choisi d'attribuer ce numéro à cette personne (et pour cause !), il est incapable de prédire le numéro de téléphone d'une autre personne.

Par exemple, considérons $\mathcal{F}_\omega$, la famille paramétrique des polynômes de degré $n-1$, il est toujours possible de trouver un estimateur capable de prédire parfaitement les valeurs de $\mathbf{Y} = (y_i)_{i=1,\cdots,n}$ uniquement à partir des indices associés (*i.e.* $\mathbf{x}_i = i$), quand bien même les composantes de $\mathbf{Y}$ sont *i.i.d.*. La méthode d'apprentissage utilisée ici (l'interpolation lagrangienne) permet d'obtenir un estimateur qui a **mémorisé** le jeu de données mais qui sera incapable de généraliser.

La mémorisation dépend généralement du niveau de régularisation $\theta \in \Theta$ utilisé durant l'apprentissage. Définissons la performance de Mémorisation.

[Performance de Mémorisation] Pour tout $\theta \in \Theta$ et $\mathcal{D}_n^{X,Y} = (\mathbf{X}, \mathbf{Y})$ tel que $\mathbf{X}$

est **indépendant** de $\mathbf{Y}$, on définit

$$\text{Perf-Mem}\,(\theta, \mathcal{D}_n^{X,Y}) := \texttt{MSE}(\mathbf{Y}, f_{\widehat{\omega}(\theta,\mathcal{D}_n^{X,Y})}(\mathbf{X})) = \|\mathbf{Y} - f_{\widehat{\omega}(\theta,\mathcal{D}_n^{X,Y})}(\mathbf{X})\|_n^2$$

Remarques.

- La pénalité $\mathsf{P}\,(\theta, \widehat{\omega}(\theta, \mathcal{D}_n^{X,Y}))$ n'apparaît pas directement dans l'expression de la performance de mémorisation.
- En revanche, la pénalité intervient dans le calcul des $\widehat{\omega}(\theta, \mathcal{D}_n^{X,Y})$ via le problème d'optimisation (1.1).
- Il est important que pour tout $i = 1, \cdots, n$ les variables aléatoires X_i soient indépendantes des Y_i pour mesurer la performance de mémorisation de θ.
- Il n'est pas nécessaire que les couples (X_i, Y_i) et (X_j, Y_j), tels que $i \neq j$, soient indépendants.
- La performance de mémorisation est définie ici pour le jeu de données entier. Notons qu'il existe une autre définition de la mémorisation qui se concentre sur une observation particulière, appelée *label memorization* ([60, 59]).

Lien entre mémorisation et généralisation.
➢ [221] et [7] ont mis en évidence un lien entre le sur-ajustement et le mécanisme de mémorisation. Ils ont montré que le phénomène de mémorisation peut également se produire lorsque la variable à expliquer et les variables explicatives sont liées. C'est la propension du modèle à mémoriser la part de bruit contenu dans la variable cible qui explique la différence entre les performances d'entraînement et les performances de généralisation. L'intuition suivante, plus clairement énoncée dans [7], est que certaines méthodes de régularisation peuvent réduire la capacité de mémorisation d'un estimateur, permettant ainsi d'améliorer les performances de généralisation.

Cette intuition est le point de départ du critère `MLR`. Il est possible d'**estimer le niveau de régularisation qui maximise la performance de généralisation en comparant la performance d'entraînement et la performance de mémorisation.** Mesurer la performance de Mémorisation est possible dès lors que l'on dispose d'un jeu de données avec des variables explicatives $\mathbf{X}$ indépendantes du vecteur cible $\mathbf{Y}$. Une façon simple d'accéder à un tel jeu de données est de le construire à partir du jeu de données initial $\mathcal{D}_n^{X,Y}$. Il suffit pour cela d'opérer une permutation aléatoire π, sans remise et sans point fixe, des indices du vecteur cible $\mathbf{Y}$, que l'on note $\mathbf{Y}_{\texttt{perm}} = (Y_{\pi(i)})_{i=1,\cdots,n}$ pour obtenir un jeu de données artificiel

$$\mathcal{D}_n^{X,\pi(Y)} := \{(X_i, Y_{\pi(i)}\}_{i=1,\cdots,n}$$

où les variables $Y_{\pi(i)}$ et X_i sont indépendantes. Ainsi, il est possible à partir du jeu de données initial de mesurer à la fois la performance de Mémorisation, $\textsf{Perf-Mem}\,(\theta, \mathcal{D}_n^{X,\pi(Y)})$, et la performance d'Entraînement, $\textsf{Fit}\,(\widehat{\omega}(\theta, \mathcal{D}_n^{X,Y}), \mathcal{D}_n^{X,Y})$.

Mesurer ces deux quantités permet de réaliser un arbitrage entre deux objectifs afin de choisir le modèle le plus approprié. Contrairement au compromis biais-variance traditionnellement utilisé pour déterminer le nombre de feuilles d'un arbre de décisions ou la taille du support d'un modèle linéaire, cet arbitrage reste pertinent pour les modèles sur-paramétrés et notamment les réseaux de neurones.

1.2 Critère `MLR` : un critère généralisant

1.2.1 Intuition du critère `MLR`.

Afin de comprendre la construction du critère `MLR`, notons que dans le cadre classique où la régularisation se justifie :
➢ Pour un niveau de régularisation trop élevé, le modèle estimé ne sera pas assez expressif et il lui sera difficile de capturer le lien entre les variables aléatoires X et Y. Dans le cas le plus extrême, l'estimateur ne dépendra pas des variables explicatives et prédira une valeur constante, *i.e.* il correspond au modèle réduit à l'*intercept* qui est clairement incapable de généraliser. En supposant, sans perte de généralité, $\mathbf{Y}$ centré et réduit, la performance de Mémorisation sera du même ordre que la performance d'Entraînement :

$$\textsf{Perf-Mem}\,(\theta, \mathcal{D}_n^{X,\pi(Y)}) \simeq \textsf{Fit}\,(\widehat{\omega}(\theta, \mathcal{D}_n^{X,Y}), \mathcal{D}_n^{X,Y}) \simeq 1.$$

Dans ce cas, la performance de Mémorisation est très mauvaise ($\simeq 1$) sur l'échantillon permuté $\mathcal{D}_n^{X,\pi(Y)}$ et le modèle, réduit à l'*intercept*, est incapable de généraliser.
➢ Pour un niveau de régularisation trop faible, le modèle estimé mémorisera, *i.e.* il ne fera pas de distinction entre signal et bruit. Dans ce cas, il est possible de construire un modèle qui mémorise $\mathcal{D}_n^{X,\pi(Y)}$, *i.e.* tel que

$$\textsf{Perf-Mem}\,(\theta, \mathcal{D}_n^{X,\pi(Y)}) \simeq \textsf{Fit}\,(\widehat{\omega}(\theta, \mathcal{D}_n^{X,Y}), \mathcal{D}_n^{X,Y}) \ll 1.$$

La performance de Mémorisation est très bonne ($\ll 1$) sur l'échantillon permuté $\mathcal{D}_n^{X,\pi(Y)}$ et le modèle *overfit*, il n'arrive pas à généraliser.
➢ Intuitivement, on se dit qu'il existe un niveau de régularisation intermédiaire optimal θ^* qui correspond au cas où il est facile de généraliser mais difficile de mémoriser l'échantillon permuté $\mathcal{D}_n^{X,\pi(Y)}$. En d'autres termes, *fitter* ($\textsf{Fit}\,(\widehat{\omega}(\theta^*, \mathcal{D}_n^{X,Y}), \mathcal{D}_n^{X,Y})$) l'échantillon $\mathcal{D}_n^{X,Y}$, là où il existe un lien entre les variables X et Y, mais ne pas mémoriser ($\textsf{Perf-Mem}\,(\theta^*, \mathcal{D}_n^{X,\pi(Y)})$) l'échantillon permuté

$\mathcal{D}_n^{X,\pi(Y)}$, là où le lien entre les variables X et Y a été rompu. En comparant ces deux quantités, on obtient le critère MLR, pour *Muddling Labels Regularization* ("régularisation par brouillage des labels").

Critère 3. ***[Muddling Labels Regularization]***

$$\begin{array}{rcl}\Theta & \longrightarrow & \mathbb{R}\\ \theta & \longmapsto & \mathtt{MLR} := \mathtt{MSE}\Big(\mathbf{Y},\, f_{\widehat{\omega}(\theta,\mathcal{D}_n^{X,Y})}(\mathbf{X})\Big) - \mathtt{MSE}\Big(\mathbf{Y}_{\mathtt{perm}},\, f_{\widehat{\omega}(\theta,\mathcal{D}_n^{X,\pi(Y)})}(\mathbf{X})\Big)\end{array}$$

1.2.2 Forme close et critère MLR.

➢ À l'instar de la Validation Croisée, le critère MLR peut être utilisé pour calibrer des hyper-paramètres :

$$\widehat{\theta}_{\mathtt{MLR}}(\mathcal{D}_n^{X,Y}) := \arg\min_{\theta\in\Theta} \mathtt{MLR}(\theta)$$

Contrairement à la Validation Croisée, ce critère ne nécessite pas d'échantillon de *validation* et permet ainsi d'entraîner le modèle sur le jeu de données entier. Le critère MLR calcule simultanément la performance de l'estimateur $f_{\widehat{\omega}(\theta,\mathcal{D}_n^{X,Y})}$ et celle de l'estimateur "*permuté*" $f_{\widehat{\omega}(\theta,\mathcal{D}_n^{X,\pi(Y)})}$. Si l'on dispose d'une forme close de l'estimateur $f_{\widehat{\omega}}$, minimiser le critère MLR utilisé pour apprendre le niveau de régularisation $\widehat{\theta}_{\mathtt{MLR}}$ durant l'entraînement ne nécessite plus de résoudre un problème d'optimisation bi-niveau.

Ainsi, pour la famille de modèles Ridge,

$$\mathcal{F}_\lambda^R = \{f_\lambda(\cdot) = \langle\beta_\lambda, \cdot\rangle,\ \lambda > 0\} \text{ avec } \beta_\lambda = \beta_R(\lambda, \mathbf{X}, \mathbf{Y}) = (\mathbf{X}^\top\mathbf{X} + \lambda\mathbb{I}_p)^{-1}\mathbf{X}^\top\mathbf{Y} \in \mathbb{R}^p.$$

et en supposant, sans perte de généralité, le vecteur $\mathbf{Y}$ et les colonnes de $\mathbf{X}$ centrés et réduits, le critère MLR s'écrit :

$$\beta_R(\lambda, \mathbf{X}, \mathbf{Y}) := \arg\min_{\beta\in\mathbb{R}^p} \|\mathbf{Y} - \mathbf{X}\beta\|_2^2 + \lambda\|\beta\|_2^2 = \left(\mathbf{X}^\top\mathbf{X} + \lambda\,\mathbb{I}_p\right)^{-1}\mathbf{X}^\top\mathbf{Y}$$

Le critère $\mathtt{MLR}(\lambda)$ s'écrit alors :

$$\mathtt{MLR}(\lambda) = \|\mathbf{Y} - \mathbf{X}\left(\mathbf{X}^\top\mathbf{X} + \lambda\,\mathbb{I}_p\right)^{-1}\mathbf{X}^\top\mathbf{Y}\|_2^2 - \|\mathbf{Y}_{\mathtt{perm}} - \mathbf{X}\left(\mathbf{X}^\top\mathbf{X} + \lambda\,\mathbb{I}_p\right)^{-1}\mathbf{X}^\top\mathbf{Y}_{\mathtt{perm}}\|_2^2 \quad (1.3)$$

Le critère $\mathtt{MLR}(\lambda)$ est différentiable en λ, il est donc possible de calibrer λ, l'hyper-paramètre de régularisation du Ridge, à l'aide d'une descente de gradient.

1.2.3 Étude théorique de MLR.

➢Plaçons nous dans le cas simple du modèle de régression linéaire $\mathbf{Y} = \mathbf{X}\boldsymbol{\beta}^* + \xi$, où ξ un bruit sous-gaussien isotrope. On suppose que $\mathbf{X}^\top \mathbf{X}/n$ est une projection orthogonale (notée $P_{\mathbf{X}}$) sur un sous-espace de dimension r de l'espace $\mathbb{R}^p$.

On considère la famille de modèles Ridge $\mathcal{F}_\lambda^R$. Alors, sous l'hypothèse ratio Signal-Bruit (ϵ_n) : $r\sigma^2 \ll \|\mathbf{X}\beta^*\|_2^2 \ll n\sigma^2$, on montre (Théorème 1, Chapitre 3) qu'avec grande probabilité

$$\mathtt{MLR}(\lambda) + \|P_{\mathbf{X}}(\mathbf{Y}_{\mathtt{perm}})\|_2^2 = (1 + o(1))\, R(\lambda), \quad \forall \lambda > \epsilon_n,$$

où $R(\lambda) := \mathbb{E}_\xi[\|\mathbf{X}\beta^* - \mathbf{X}\beta_\lambda\|_2^2]$ est le risque de généralisation et ϵ_n le taux

$$\epsilon_n := \sqrt{\frac{r\sigma^2}{\|\mathbf{X}\beta^*\|_2^2}} + \sqrt{\frac{\|\mathbf{X}\beta^*\|_2^2}{n\sigma^2}}.$$

Remarques :

- Le critère MLR est un bon estimateur du risque de généralisation de l'estimateur Ridge.
- La condition Signal-Bruit (ϵ_n) sous-entend qu'il y a suffisamment de bruit pour qu'il soit nécessaire de régulariser mais pas suffisamment pour que le bruit écrase le signal ; ce qui correspond au cas où il est pertinent d'utiliser de la régularisation.
- Le terme $\|P_{\mathbf{X}}(\mathbf{Y}_{\mathtt{perm}})\|_2^2$ ne dépend pas de λ, ce qui veut dire que dans cet exemple simple, le critère MLR permet d'estimer le niveau de régularisation optimale.
- Les expériences numériques proposées au Chapitre 3 montrent qu'il peut être utiliser pour estimer le paramètre de régularisation optimal de l'estimateur Ridge avec plus de précision qu'une validation croisée.

1.2.4 Nouveaux modèles.

➢ Pour $A_\theta(\cdot)$, une application paramétrée et différentiable en θ, on peut introduire une nouvelle famille de modèles linéaires

$$\mathcal{F}_{\lambda,\theta}^R = \{f_{\lambda,\theta}(\cdot) = A_\theta(\cdot)\beta_R(\lambda, A_\theta(\cdot)\mathbf{Y})\}.$$

Le critère MLR défini (1.3) étant également différentiable en $\mathbf{X}$, remplacer $\mathbf{X}$ par $A_\theta(\mathbf{X})$, permet d'optimiser simultanément la fonction de perte MLR en λ et en θ. Les propriétés des modèles $\mathcal{F}^R_{\lambda,\theta}$ dépendent du choix de l'application A_θ. Suivant le choix de A_θ, il est possible de faire, en plus de la régularisation, de la sélection de variables et de l'agrégation d'estimateurs et ceci via une descente de gradient sur la fonction de perte MLR. De telles familles sont présentées au Chapitre 3 pour un choix d'applications linéaires de A_θ, dans le cadre de la régression linéaire, trois familles appelées R-MLR, S-MLR et A-MLR. Elles permettent de calibrer des hyper-paramètres de régularisation par une descente de gradient. R-MLR permet de calibrer la pénalité L_2 de l'estimateur Ridge. S-MLR permet de calibrer simultanément la pénalité L_2 et de faire de la sélection de variable, pour obtenir un modèle parcimonieux. A-MLR permet, toujours grâce à une seule descente de gradient, d'entraîner en parallèle un modèle Ridge et un un modèle parcimonieux et de choisir le modèle le plus approprié en fonction des données.

Dans le Chapitre 4, on s'intéresse à une famille de modèles non linéaires ; *e.g.* A_θ correspond aux couches cachées d'un réseau de neurones, avec θ qui correspond aux poids et biais des couches cachées et l'estimateur $f_{\lambda,\theta}(\cdot) = A_\theta(\cdot)\beta_R(\lambda, A_\theta(\cdot)\mathbf{Y})$ qui est le réseau de neurones associé que l'on peut entraîner avec la fonction de perte MLR. Le Chapitre 4 présente en détail AdaCap, une procédure pour utiliser MLR avec des réseaux de neurones.

1.3 AdaCap : Extension aux réseaux de neurones

Pour les lecteurs non familiers des réseaux de neurones ou de l'apprentissage profond, le Chapitre 2 est une introduction pas-à-pas au domaine en partant du modèle linéaire. On présente également les différentes techniques existantes ainsi que leurs limitations.

La dernière décennie a été témoin des avancées et des performances spectaculaires de l'apprentissage profond dans les domaines tels que la vision par ordinateur ([118]), le traitement du signal sonore ([92]) et le traitement du langage naturel ([53]). Récemment encore, on pensait que l'apprentissage profond n'était pas forcément très pertinent pour traiter les données tabulaires ([164]), les méthodes plus classiques d'apprentissage machine restant en tête d'affiche dans ce cadre. C'est d'autant plus vrai lorsque les jeux de données tabulaires sont de petite taille. En effet, en apprentissage profond, on fait souvent appel à des techniques de *transfer learning* (apprentissage par transfert) pour traiter les jeux de données de petite taille ([147]), ce qui est rarement approprié pour les jeux de données tabulaires ([181]). De plus, les variables explicatives

catégorielles posent des difficultés aux réseaux de neurones ([162, 79, 210, 42, 45]).

Le développement de techniques d'apprentissage profond dédiées aux données tabulaires reste malgré cela un objectif important pour la communauté. Le récent projet `PyTorch Tabular` ([106]) et le nombre important d'articles sur les données tabulaires ces dernières années montrent l'intérêt croissant de la communauté de l'apprentissage profond pour ce sujet (le nombre de références mentionnées dans l'article état-de-l'art [24] en témoigne). Au delà de leurs performances, les réseaux de neurones ont l'avantage de faire par nature du *feature-engineering* (pré-traitement des variables explicatives). Là où habituellement l'étude des caractéristiques des données se fait manuellement, en apprentissage profond ce pré-traitement des variables explicative est intégré à la phase d'apprentissage, et se fait donc de façon automatique durant la descente de gradient.

1.3.1 État de l'art non exhaustif.

➢ Parmi les contributions importantes, citons d'abord les `NODE` ([156]) et `TabNet` ([6]), deux méthodes qui faisaient références jusqu'à récemment. Les `NODE` (*Neural Oblivious Decision Ensembles*) sont des architectures calquées sur les arbres de décision qui peuvent être entraînées de bout en bout par rétropropagation (méthodes dites *end-to-end differentiable*). `TabNet`, pour sa part, utilise des mécanismes d'attention (*attention-based mechanisms* , [12]) pour encoder les variables explicatives durant une phase de pré-entraînement. Ce pré-traitement des variables explicatives reste automatisé, mais l'apprentissage n'est plus *end-to-end differentiable*. La comparaison entre les réseaux de neurones classiques, `NODE`, `TabNet`, les forêts aléatoires et les méthodes de *Gradient Boosting* sur les jeux de données tabulaires a été faite de façon concomitante par [107], [75] et [183]. Leurs benchmarks sont très orientés vers une approche `AutoML` (apprentissage machine automatisé) car ils font tous appel à des procédés d'**HPO** très coûteux en temps de calcul. Il faut alors plusieurs minutes voir plusieurs heures pour entraîner un modèle, même pour des jeux de données de petite taille. Comme l'affirme [183], leurs résultats indiquent que les réseaux de neurones ne sont pas pour l'instant la panacée pour traiter les jeux de données tabulaires. [107] proposent, en plus de leur benchmark, une procédure d'**HPO** clé en main appelée *regularization cocktail* qui, bien qu'elle soit efficace, est très coûteuse à mettre en œuvre.

➢ Pour une vision exhaustive des techniques les plus contemporaines pour traiter les données tabulaires en apprentissage profond, on se référera à l'article état-de-l'art [24] et à la plateforme [212], nous mentionnons ci-dessous que celles utilisées dans les expériences présentées dans les Chapitres 4 et 5. [114] ont

introduit les *Self-Normalizing Networks* (SNN) qui permettent de faire fonctionner des architectures plus profondes, en utilisant la fonction d'activation SeLU. [75] ont proposé de nouvelles architectures : les ResBlock, les *Gated Linear Units*, et les *FeatureTokenizer-Transformers*, qui sont respectivement des adaptations des *ResNet* ([90]), des *Gated Convolutional Networks* ([49]) et des *Transformers* ([207]).

1.3.2 L'opérateur Tikhonov.

➢ Le critère MLR peut être utilisé pour calibrer les hyper-paramètres de régularisation, ce n'est cependant pas la seule façon de l'exploiter dans le cadre de l'apprentissage profond. Contrairement à la Validation Croisée, le critère MLR est differentiable selon les hyper-paramètres. Il permet, pour les modèles dont une forme close est disponible, de calibrer ces derniers directement sur le jeu de données d'entraînement par descente de gradient. En appliquant la forme close Ridge à la dernière couche du réseau, il est possible d'exprimer les poids de la couche de sortie en fonction des couches cachées : c'est l'opérateur Tikhonov imaginé initialement par le mathématicien Andreï Nikolaïevitch Tikhonov. Considérons une architecture de réseau de neurones *Feed-Forward* à L couches. Notons θ les poids et biais de toutes les couches cachées. Soit $N_{L-1} \in \mathbb{N}^*$ la largeur de la dernière couche cachée. Notons $\mathbf{A}^{L-1}(\theta, \cdot) \; : \; \mathbb{R}^{n\times p} \to \mathbb{R}^{n\times N_{L-1}}$ les sorties de la dernière couche cachée en fonction des entrées du réseau et des poids. alors l'opérateur Tikhonov est défini de la façon suivante.

Opérateur Tikhonov
Pour $\lambda \in \mathbb{R}_+^*$, on définit Tikhonov est défini de la façon suivante :

$$\mathbf{H}(\lambda, \theta, \mathbf{X}) := \mathbf{A}^{L-1}\mathbf{P}(\lambda, \theta, \mathbf{X}).$$

où $\mathbf{A}^{L-1} := \mathbf{A}^{L-1}(\theta, \mathbf{X})$ et pour $\mathbb{I}_{N_{L-1}}$ est la matrice identité

$$\mathbf{P}(\lambda, \theta, \mathbf{X}) := \left[\left(\mathbf{A}^{L-1}\right)^\top \mathbf{A}^{L-1} + \lambda \times \mathbb{I}_{N_{L-1}}\right]^{-1} \left(\mathbf{A}^{L-1}\right)^\top .$$

L'introduction de l'opérateur Tikhonov dans le modèle permet d'estimer les paramètres $\widehat{\lambda}$ et $\widehat{\theta}$ simultanément via la perte MSE

$$(\widehat{\lambda}^{\text{Tik.}}, \widehat{\theta}^{\text{Tik.}}) := \underset{\lambda>0,\, \theta\in\Theta}{\arg\min}\, \mathsf{Fit}\,(\lambda, \theta,\, \mathbf{X}, \mathbf{Y}) = \underset{\lambda>0,\, \theta\in\Theta}{\arg\min}\, \mathtt{MSE}\,(\mathbf{Y}, f_{\lambda,\theta,\mathbf{Y}}(\mathbf{X}))\,. \qquad (1.4)$$

Le modèle estimé est

$$\widehat{f}(\cdot) = (\cdot)\mathbf{W}^L(\widehat{\lambda}^{\text{Tik.}}, \widehat{\theta}^{\text{Tik.}}, \cdot, \mathbf{Y}) = \mathbf{H}(\widehat{\lambda}^{\text{Tik.}}, \widehat{\theta}^{\text{Tik.}}, \cdot)\,\mathbf{Y}$$

où $\mathbf{W}^L$ correspond aux poids de la dernière couche. Ceux-ci sont fixés à l'issu de la phase d'apprentissage

$$\mathbf{W}^L(\widehat{\lambda}^{\text{Tik.}}, \widehat{\theta}^{\text{Tik.}}, \mathbf{X}, \mathbf{Y}) = \mathbf{P}(\widehat{\lambda}^{\text{Tik.}}, \widehat{\theta}^{\text{Tik.}}, \mathbf{X})\,\mathbf{Y}$$

Il est important de noter que $\mathbf{Y}$ est fixé, c'est le vecteur cible du jeu de données d'entraînement. On retrouve donc l'architecture d'un réseau de neurones classique, tel que défini dans le Chapitre 2.

Remarques

➢ L'opérateur `Tikhonov` n'est équivalent ni au *Weight-Decay*, ni à une pénalité L_2. Afin de mieux appréhender cette différence, considérons le cas simple d'un réseau de neurones à 1 couche cachée dont les poids de la couche cachée et de la dernière couche sont respectivement $\mathbf{W}^1$ et $\mathbf{W}^2$. Pendant la rétropropagation, le gradient de la fonction de perte usuelle `MSE` est :

$$\nabla_{\mathbf{W}^l}\|\mathbf{Y} - \mathbf{A}^2\|_2^2 = \nabla_{\mathbf{A}^2}\partial\|\mathbf{Y} - \mathbf{A}^2\|_2^2 \times \underbrace{\nabla_{\mathbf{W}^l}\mathbf{A}^2}_{\text{terme usuel}}$$

Si nous ajoutons une régularisation *Weight-Decay* de niveau λ sur la couche de sortie $\mathbf{A}^2$, un autre terme apparaît dans les mises à jour des poids :

$$\lambda\,\nabla_{\mathbf{W}^2}\|\mathbf{W}^2\|_2^2.$$

Avec l'opérateur `Tikhonov`, ce réseau de neurones s'exprime de la façon suivante :

$$\mathbf{A}^1 = \sigma(\mathbf{X}\mathbf{W}^1),$$
$$\mathbf{A}^2 = \mathbf{A}^1\beta_R(\lambda, \mathbf{A}^1, \mathbf{Y}) = \mathbf{A}^1\mathbf{P}(\lambda, \theta, \mathbf{X})\mathbf{Y} = \mathbf{A}^1((\mathbf{A}^1)^\top\mathbf{A}^1 + \lambda\,\mathbb{I}_{N_1})^{-1}(\mathbf{A}^1)^\top\mathbf{Y}.$$

Ici σ correspond à la fonction d'activation sur la couche cachée. Les paramètres appris durant l'entraînement sont $\mathbf{W}^1$ et λ. Le gradient de $\mathbf{A}^2$ par rapport à $\mathbf{W}^1$ s'exprime de la façon suivante :

$$\nabla_{\mathbf{W}^1}\mathbf{A}^2 = \underbrace{\nabla_{\mathbf{W}^1}\mathbf{A}^1}_{\text{terme usuel}} \times \beta_R(\lambda, \mathbf{A}^1, \mathbf{Y}) + \mathbf{A}^1 \times \underbrace{\nabla_{\mathbf{W}^1}\beta_R(\lambda, \mathbf{A}^1, \mathbf{Y})}_{\text{terme supplémentaire}}$$

avec

$$\begin{aligned}
\nabla_{\mathbf{W}^1}\beta_R(\lambda, \mathbf{A}^1, \mathbf{Y}) &= \nabla_{\mathbf{W}^1}\mathbf{P}(\lambda, \theta, \mathbf{X})\mathbf{Y} \\
&= \nabla_{\mathbf{W}^1}\left[((\mathbf{A}^1)^\top\mathbf{A}^1 + \lambda\,\mathbb{I}_{N_1})^{-1}(\mathbf{A}^1)^\top\mathbf{Y}\right] \\
&= \nabla_{\mathbf{W}^1}\left[(\sigma(\mathbf{X}\mathbf{W}^1))^\top\sigma(\mathbf{X}\mathbf{W}^1) + \lambda\,\mathbb{I}_{N_1})^{-1}(\sigma(\mathbf{X}\mathbf{W}^1))^\top\mathbf{Y}\right]
\end{aligned}$$

➢ Ce nouveau terme correspond à l'exploration d'un autre chemin dans le graphe de dérivation de la Rétropropagation. Cela signifie que l'impact de l'opérateur `Tikhonov` sera transmis aux couches cachées du réseau. Ainsi, lorsque l'on applique l'opérateur `Tikhonov` à la sortie de la dernière couche cachée et que nous utilisons ensuite la Rétropropagation pour entraîner le réseau de neurones, c'est toutes les couches qui sont affectées, pas uniquement celle que l'on exprime à l'aide d'une forme close.

➢ Le Chapitre 4 propose des expériences numériques qui montrent la différence radicale en terme de fonctionnement et de performances entre le *Weight-Decay* et l'opérateur `Tikhonov`.

1.3.3 La méthode AdaCap.

➢ Si nous combinons l'opérateur `Tikhonov` au critère `MLR`, on est capable de contrôler la capacité de mémorisation, de façon adaptative durant l'entraînement, de tout le réseau en agissant uniquement sur la dernière couche au travers de l'opérateur `Tikhonov`. C'est de là que vient le nom de la méthode `AdaCap` : *Adaptative Capacity Control.*

Pour utiliser l'opérateur `Tikhonov` avec le critère `MLR`, il suffit d'adapter l'expression 4.5 de $\widehat{\lambda}$ et $\widehat{\theta}$:

$$\begin{aligned}(\widehat{\lambda}_{\texttt{MLR}}, \widehat{\theta}_{\texttt{MLR}}) :=& \underset{\lambda>0,\, \theta\in\Theta}{\arg\min}\ \texttt{MLR}(\lambda, \theta)\\ =& \underset{\lambda>0,\, \theta\in\Theta}{\arg\min}\ \texttt{MSE}\left(\mathbf{Y}, f_{\lambda,\theta,\mathbf{Y}}(\mathbf{X})\right) - \texttt{MSE}\left(\mathbf{Y}_{\texttt{perm}}, f_{\lambda,\theta,\mathbf{Y}_{\texttt{perm}}}(\mathbf{X})\right)\end{aligned}$$

Le quatrième chapitre présente en détail l'extension du champs d'application du critère `MLR` à l'apprentissage profond : `AdaCap`, nouvelle méthode d'apprentissage pour les réseaux de neurones basée sur le critère `MLR` et l'opérateur `Tikhonov`. Ce dernier, technique de régularisation, permet de transformer le critère `MLR` en une fonction de perte différentiable. L'intérêt de la méthode `AdaCap`, dans le cadre de la régression, est validé expérimentalement par un benchmark extensif sur de nombreux jeux de données tabulaires. Nous comparons `AdaCap` avec un nombre important de méthodes tant en apprentissage profond qu'en apprentissage machine. Une implémentation des principales architectures de réseau de neurones pour données tabulaires, avec ou sans l'usage d'`AdaCap` est également proposée. Celle-ci associe l'accessibilité et la praticité de l'interface de la librairie *scikit-learn* avec la puissance et la flexibilité de la librairie *pytorch* .

1.4 Estimation de densité conditionnelle : la méthode MCD

1.4.1 Estimation de densité conditionnelle : une tâche centrale.

➢ Intéressons nous maintenant au problème de l'estimation de densité conditionnelle qui est, tout comme la généralisation, un sujet d'intérêt majeur dans les domaines des statistiques mathématiques et de l'apprentissage machine.

On considère un couple de variables aléatoires (X, Y) à valeurs dans $\mathcal{X} \times \mathcal{Y}$ tel que $\mathcal{X} \subseteq \mathbb{R}^p$ et $\mathcal{Y} \subseteq \mathbb{R}$. On suppose maintenant que ces variables aléatoires admettent une fonction de densité par rapport à une mesure dominante et que ces densités ont pour intégrale 1. On note f_X et f_Y les densités marginales de X et Y par rapport à une mesure dominante.
Notre objectif désormais est d'estimer la fonction de densité conditionnelle :

$$\begin{aligned} \mathcal{X} \times \mathcal{Y} &\longrightarrow \mathbb{R}_+ \\ (x, y) &\longmapsto f_{Y\,|\,X=x}(y) \end{aligned}$$

Le problème d'estimation de densité conditionnelle est à la base de la majorité des tâches d'apprentissage machine, y compris l'apprentissage supervisé et non supervisé et les modèles génératifs. En apprentissage supervisé on cherche à estimer l'espérance conditionnelle, ce qui est un problème équivalent à l'estimation de densité conditionnelle dans le cadre de la classification binaire, puisque $\mathbb{E}[Y\,|\,X = x] = \mathbb{P}[Y = 1\,|\,X = x]$. Dans le cadre de la régression, c'est à dire lorsque Y est une variable continue, la densité conditionnelle est bien plus informative que l'espérance, en particulier lorsque la densité conditionnelle est multimodale, hétéroscédastique ou à queue lourde.

De plus, dans de nombreux domaines tels que l'actuariat, la gestion d'actifs, la climatologie, l'économétrie, la médecine ou l'astronomie, on s'intéresse à des quantités autres que l'espérance, telles que les moments d'ordre supérieur (variance, asymétrie, kurtosis), les intervalles de confiance, la régression par quantile, les valeurs aberrantes, etc.

La plupart des techniques d'apprentissage non supervisé visent à découvrir des relations et des modèles entre des variables aléatoires. Cela correspond à une tâche d'estimation de la fonction de densité conjointe, qui est elle-même une sous-tâche de l'estimation de densité conditionnelle. De même, les modèles génératifs (ex. : les *GAN* [74]), dont le but est de générer des données synthétiques

en exprimant la densité conjointe comme un produit de densité conditionnelles univariées par rapport à des variables latentes, peut également être considéré comme une sous-tâche de l'estimation de densité conditionnelle, car pour qu'un échantillon synthétique (ex. : une image ou un son) soit réaliste il doit correspondre à une valeur modale de cette densité.

1.4.2 Travaux connexes.

➢ **L'estimation non paramétrique** est un outil puissant pour l'estimation de densité conjointe car elle ne nécessite aucune connaissance préalable de la densité sous-jacente. Les premières intuitions concernant les estimateurs à noyau ont été proposées par [171] et plus tard par [149]. Ils ont ensuite été largement étudiés, en particulier le problème de la sélection de la fenêtre (*bandwidth* [72]), l'agrégation non linéaire ([170]), l'optimisation des calculs ([123]) ou encore l'extension aux densités conditionnelles ([20]). Nous renvoyons à l'ouvrage [184] et aux références qui s'y trouvent pour les lecteurs intéressés. Une méthode non paramétrique très populaire et efficace est celle des k plus proches voisins (*KNN* [64]), mais comme d'autres méthodes non paramétriques (estimateurs à noyau, histogrammes [153],...), elle souffre de la malédiction de la grande dimension (*curse of dimensionality*) ([178, 140]). [184] a montré que, dans un cadre d'estimation de densité, le nombre de points n nécessaire pour obtenir des résultats équivalents lorsque l'on estime la densité d'une variable aléatoire à d dimensions croît à une vitesse de $n^{\frac{4}{4+d}}$. Parallèlement, le temps de calcul augmente également de manière exponentielle avec la dimension. À notre connaissance, la méthode la plus performante pour l'estimation des noyaux, qui est basée sur un algorithme de type diviser pour régner ([123]), a une complexité de $\mathrm{T}(n\log(n)^{\max(d-1,1)})$. Ainsi, ces deux problèmes se superposent puisque une dimension plus grande signifie que la taille de l'échantillon nécessaire est exponentiellement plus grande et que le temps de calcul par observation est exponentiellement plus élevé.

➢ **Reformulation de la la tâche d'estimation de densité conditionnelle.** Il existe d'autres façons de reformuler la tâche d'estimation de densité conditionnelle. [195] propose une reformulation de la fonction de densité conditionnelle en un rapport de densités marginales estimées par une méthode des moindres carrés (**LSCond**) et [134] propose de reformuler le problème comme une tâche de régression quantile. Une autre approche populaire consiste à reformuler le problème d'estimation de densité conditionnelle comme une tâche de régression en apprentissage supervisé. **RFCDE** ([159]) et **NNKCDE** ([157]) sont des adaptations respectivement des forêts aléatoires et des méthodes de plus proches voisins à ce problème. D'autres méthodes, comme **Deepcde** ([56]) et **FlexCode** ([102]), vont plus loin et conçoivent

une tâche de régression de substitution qui peut être traitée par une méthode d'apprentissage supervisé, ce qui permet d'utiliser les nombreux outils open-source disponibles pour l'apprentissage supervisé, tels que scikit-learn ([154]), *pytorch* ([151]), *fastai* ([99]) , *keras* ([43]), *tensorflow* ([133]), **CatBoost** ([160]), `XGB`([40]), etc.

➢ **Les méthodes de noise contrastive learning.** Une autre famille importante de techniques d'estimation de densité conjointe est appelée *noise contrastive learning* ([80]). Ces méthodes reformulent la tâche d'estimation de densité en un problème de classification binaire (à une constante près). On introduit une densité de probabilité connue $g(\cdot)$ selon laquelle on va générer un jeu de données synthétique. Ce dernier est concaténé avec le jeu de données d'origine, puis une variable à expliquer Z_i est associée à chaque observation, telle que Z_i est égale à 1 si l'observation provient du jeu de données d'origine et 0 sinon. On introduit ensuite une fonction de contraste $q(\cdot)$, qui permet, sous certaines conditions sur g, d'obtenir une estimation de la fonction de densité :

$$q(x) := \mathbb{P}[Z = 1 \,|\, X = x] = \frac{f_X(x)}{f_X(x) + g(x)} \quad \text{et} \quad f_X(x) = g(x)\frac{q(x)}{1 - q(x)}. \tag{1.5}$$

Il suffit alors d'entraîner un classifieur binaire à prédire la variable à expliquer Z en fonction de l'observation associée X, pour pouvoir, grâce à la fonction de contraste, estimer la fonction de densité conjointe.

Le *noise contrastive learning* a été appliqué avec succès dans le domaine de l'apprentissage auto-supervisé (*self-supervised learning* ([103, 87]), notamment sur des tâches de vision par ordinateur ([11]). De nombreuses extensions et améliorations ont été proposées, notamment en ce qui concerne la fonction de perte ([112]), les techniques de *data-augmentation* ([41]), les architectures de réseaux de neurones ([87]) et les méthodes de calcul ([220]).

➢ La méthode que nous proposons est partiellement basée sur cette technique, avec deux différences majeures. Premièrement, notre technique permet d'estimer une densité conditionnelle et non une densité conjointe. Deuxièmement, l'expression de la fonction de densité g introduite dans notre méthode est inconnue et potentiellement insoluble, ce qui enfreint les restrictions habituelles en *noise contrastive learning*, mais permet de faire un choix particulièrement adapté au problème posé. À notre connaissance, la technique qui se rapproche le plus de notre méthode est conçue pour évaluer dans quelle mesure une densité conjointe est différente du produits des densités marginales, en simulant le cas où les variables $\{X_j\}_{j=1,\cdots,p}$ sont indépendantes. Cette méthode brièvement décrite dans la deuxième édition de [200] (pages 495 à 497), correspond à la reformulation utilisée en *noise contrastive learning* en prenant $g(\cdot) = \Pi_{j=1}^{p} f_{X_j}(\cdot)$. Notons que si, ici, g est inconnue, il est néanmoins possible de générer un jeu de données synthétique, il suffit pour cela

d'appliquer des permutations aléatoires sur les colonnes de la matrice *design*. Cela permet d'estimer la fonction de contraste (et donc de découvrir des interactions entre les variables) mais pas la fonction de densité conjointe f_X. Pour estimer cette dernière, il faut en plus de cela estimer p densités marginales, et même si chaque composante de X est unidimensionnelle par définition, les erreurs commises vont se superposer, et ainsi la dimension de X reste un facteur limitant avec cette technique.

1.4.3 La méthode MCD.

➢ Notre méthode, appelée **MCD** pour *Marginal-Contrastive Discrimination*, combine plusieurs caractéristiques des méthodes présentées précédemment, mais sans leurs limites respectives. Le principe de base de notre méthode est de décomposer la fonction de densité conditionnelle en deux sous-problèmes : l'estimation de la fonction de densité marginale de Y d'une part et l'estimation d'un rapport de fonctions de densité, d'autre part :

$$f_{Y|X=x}(y) = f_Y(y) \times \frac{f_{X,Y}(x,y)}{f_X(x) f_Y(y)}, \quad \forall (x,y) \in \mathcal{X} \times \mathcal{Y} \text{ tel que } f_X(x) \neq 0,\ f_Y(y) \neq 0$$

Avec MDcond, ces deux quantités sont estimées séparément. Dans la plupart des applications pratiques de l'estimation de densité conditionnelle, Y est une variable univariée ou de faible dimension, ce qui n'est pas nécessairement le cas de X, il est donc plus facile d'estimer f_Y que f_X et $f_{X,Y}$. Contrairement aux méthodes qui utilisent directement la formule de Bayes, avec MDcond il n'est pas nécessaire d'estimer f_X et $f_{X,Y}$. En effet seules l'estimation de f_Y et l'estimation d'une fonction de contraste sont requises, l'estimation du rapport de fonctions de densité étant reformulé en estimation de contraste.

➢**Expression sous forme de contraste.** Contrairement aux méthodes de *noise contrastive learning*, nous nous concentrons sur le contraste entre la loi conjointe $f_{X,Y}$ et les lois marginales f_X et f_Y. Nous reformulons le rapport $f_{X,Y}/(f_X f_Y)$ à l'aide d'un nouveau contraste, appelé **fonction de contraste marginal MCF**(r) de ratio $r \in (0,1)$

$$\begin{aligned} \mathcal{X} \times \mathcal{Y} &\longrightarrow [0,1) \\ (x,y) &\longmapsto q(x,y) := \frac{r f_{X,Y}(x,y)}{r f_{X,Y}(x,y) + (1-r) f_X(x) f_Y(y)}. \end{aligned}$$

Le problème d'estimation de densité conditionnelle peut alors être reformuler de la façon suivante :

$$f_{Y|X=x}(y) = f_Y(y) \frac{q(x,y)}{1-q(x,y)} \frac{1-r}{r}$$

On s'est donc restreint à l'estimation du contraste et de la densité marginale f_Y.

➢**Reformulation en problème supervisé.** Au Chapitre 5, nous déterminons des conditions, appellées ***Marginal Discrimination Condition*** **MDcond**(r), sous lesquelles l'estimation de la fonction de contraste peut être transformée en un problème supervisé de classification binaire. Nous montrons qu'il est aisé et toujours possible de construire des échantillons à partir du jeu de données initial qui satisfont les conditions MDcond(r). De plus, la méthode **MCD** permet de prendre en compte des données supplémentaires non labelisées et de les valoriser en les intégrant dans le problème de classification binaire.

Mise en œuvre de la méthode MCD
➢ Pour estimer f_Y, une méthode à noyau classique est suffisante. L'estimation de la fonction de contraste q nécessite un jeu de données où les observations $\mathbf{x}_i$ sont indépendantes des observations $\mathbf{y}_i$. Pour cela, nous pouvons construire un jeu de données $\mathcal{D}_n^{X,\pi(Y)}$ à partir du jeu de données intial $\mathcal{D}_n^{X,Y}$ par permutation (voir section 1.1.4). À partir de ce jeu de données $\mathcal{D}_n^{X,\pi(Y)}$, nous pouvons estimer la fonction de contraste comme on le ferait classiquement en *noise contrastive learning*, en entraînant un classifieur binaire. Les expériences présentées dans le Chapitre 5 montrent qu'un simple modèle Perceptron Multicouche permet d'obtenir de très bons résultats en pratique.

La méthode **MCD** est évaluée empiriquement sur un benchmark de modèles de densités et de jeux de données de régression, face à un grand nombre de méthodes existantes. La méthode **MCD**, lorsqu'elle est combinée avec des réseaux de neurones, obtient les meilleurs résultats dans la plupart des cas. Une implémentation de la méthode en python est proposée. Celle-ci est compatible avec n'importe quelle méthode d'apprentissage supervisé qui utilise l'interface de la librairie *scikit-learn*, et en particulier l'implémentation des réseaux de neurones proposée dans le Chapitre 4.

Chapitre 2
Introduction aux réseaux de neurones

> *If the doors of perception were cleansed everything would appear to man as it is, infinite.*
>
> William Blake, A Memorable Fancy

2.1 Introduction

Ce chapitre se veut être un cours d'introduction aux réseaux de neurones, les lecteurs déjà familiers peuvent passer au Chapitre 4. En partant du modèle de régression le plus simple, le modèle linéaire, nous introduirons les notions de base, pas-à-pas, en utilisant uniquement *numpy*, la librairie de calculs numérique usuelle de *python*. Pour faciliter l'acquisition de ces notions, tous les modèles considérés ici seront présenté dans leur version déterministe.

Les réseaux de neurones profonds ou *Deep Neural Network* (`DNN`) sont des modèles mathématiques complexes basés sur deux concepts issus respectivement des sciences cognitives et de la philosophie, le *connexionnisme* et l'*émergence*.

L'émergence est un concept philosophique motivé notamment par le constat qu'un ensemble d'entités suivant des règles simples peut générer des phénomènes dont la complexité dépasse les limites imposées par ces règles élémentaires. C'est le cas des réseaux de neurones : Un `DNN` est constituée de couches, chaque couche étant composée de neurones, des modèles simples à l'expressivité limitée. Le `DNN` possède une couche d'entrée, une couche de sortie et au moins une couche entre les deux. On comprend aisément que plus le nombre de couches est élevé, plus le réseaux de neurones est dit "profond". C'est notamment cet empilement de couches qui permet d'obtenir des modèles toujours plus expressifs, capables d'appréhender des phénomènes d'une très grande complexité. Sans déborder des champs des statistiques mathématiques et du *Machine Learning*, on propose dans ce chapitre une introduction au réseau de neurones qui met en évidence ce phénomène d'émergence au sein de ces modèles ([105]).

Les connexionnistes se donnent pour objectif de représenter et de comprendre les phénomènes mentaux comme des ensembles d'éléments simples organisés en réseaux ([163, 202]). L'un des principaux outils de l'approche connexionniste est le réseau de neurones artificiel où la transmission d'information se fait au travers

des différentes couches du réseau. Chaque couche a pour rôle de "capter" des motifs (*pattern*) particuliers issus des couches précédentes, qu'elle propage aux couches suivantes. A chaque étape de cette **Propagation**, le niveau de complexité de ces motifs augmente, ce qui permet au réseau de traduire des concepts et des phénomènes de plus en plus abstraits. Ce fonctionnement est analogue à celui du cerveau humain, qui peut identifier des objets pertinents au sein d'une scène et reconnaître des caractéristiques et motifs propres à ces objets. Les questions posées par l'approche connexionniste dépassent les seuls champs des statistiques mathématiques et du *Machine Learning*. Elles ne sont pas l'objet de cette thèse. Pour les lecteurs intéressés, le débat entre Yoshua Bengio et Gary Marcus fournit un bon aperçu de l'impact du courant connexionniste au sein de la communauté du *Deep Learning* ([136]).

Les réseaux de neurones sont une famille de modèles très large, composée de nombreuses sous-familles, appelées **architectures** (*e.g.* ConvNet, les LTSM, les Transformers,etc.). Cette grande diversité au sein de cette famille reflète la grande diversité des tâches considérées (*e.g.* apprentissage supervisé, non supervisé, estimation de densité, apprentissage par renforcement, modèles génératifs, etc.) et le large éventail de type de données traitées (images, vidéos, signaux sonores, texte, séries temporelles, etc.).

Dans ce chapitre, nous allons nous concentrer sur une tâche spécifique, l'apprentissage supervisé sur données tabulaires, cadre dans lequel s'inscrit cette thèse. La sous-famille *Feed-Forward Neural Network* (**FFNN**), est particulièrement bien adaptée à ce cadre. Dans ce chapitre nous présenterons l'un des modèles de cette sous-famille, le **Perceptron Multicouche** (*Multilayer Perceptron*, MLP). Les données tabulaires ont connu récemment un regain d'intérêt dans la communauté *Deep Learning*. Pour une bibliographie *état-de-l'art* sur les dernières avancées dans ce domaine, on renvoie au Chapitre 4.

Modèle statistique. Soit (X, Y) un couple de variables aléatoires à valeurs dans $\mathcal{X} \times \mathcal{Y}$, tel que $\mathcal{X} \subset \mathbb{R}^p$ avec $p \in \mathbb{N}_*$ et $\mathcal{Y} \subset \mathbb{R}$. On suppose que les variables X et Y ne sont pas indépendantes et qu'il existe un lien inconnu f, tel que $Y = f(X)$, que nous voulons reconstruire. Pour cela nous disposons d'un jeu de données d'entraînement, $\mathcal{D}_n^{X,Y} = \{(\mathbf{x}_i, y_i)\}_{i=1,\cdots,n}$, correspondant à la réalisation de $n \in \mathbb{N}_*$ couples de variables aléatoires $\{(X_i, Y_i)\}_{i=1,\cdots,n}$ *i.i.d* de même loi $P_{X,Y}$.

Notations 1.

- *Soit* $\mathbf{X}$ *la matrice design des observations* $\{\boldsymbol{x}_i\}_i$

$$\mathbf{X} = [x_{i,j}]_{\substack{i=1,\cdots,n \\ j=1,\cdots,p}} = \begin{pmatrix} \boldsymbol{x}_1^\top \\ \vdots \\ \boldsymbol{x}_n^\top \end{pmatrix} = [\mathbf{X}_1, \cdots, \mathbf{X}_p]$$

  où $\boldsymbol{x}_i$ *est l'observation i telle que* $\boldsymbol{x}_i \in \mathcal{X} \subset \mathbb{R}^p$ *et* X_j *est la variable explicative (feature) telle que* $X_j \in \mathbb{R}^n$.

- *Soit* $\mathbf{Y}$ *la matrice des observations* $\{y_i\}_i$

$$\mathbf{Y} = (y_1, \cdots, y_n)$$

  où y_i *est appelée la variable à expliquer (target/label) et est telle que* $y_i \in \mathcal{Y} \subset \mathbb{R}$.

Approximateurs universels. On cherche à reconstruire le lien déterministe f à partir du jeu de données d'entraînement, $\mathcal{D}_n^{X,Y} = \{(\mathbf{x}_i, y_i)\}_{i=1,\cdots,n}$. On se place dans un cadre paramétrique où f est connue à un paramètre près,*i.e.* que l'on suppose que f appartient à une classe $\mathcal{F}_\omega$ de fonctions paramétrée par le paramètre $\omega \in \mathcal{W} \subset \mathbb{R}^D$, $D \in \mathbb{N}^*$.

$$\mathcal{F}_\omega = \{f_\omega \ : \ \omega \in \mathcal{W} \subset \mathbb{R}^D\}.$$

Dans ce cadre, la taille du modèle est D. Si l'on considère les réseaux de neurones D est en général beaucoup plus grand que souvent la taille du jeu de données :$D \gg n$. On dit que les réseaux de neurones sont des modèles sur-paramétrés. Pour donner un ordre de grandeur, dans les expériences présentées dans le Chapitre 4, on utilise entre autres des réseaux de neurones de taille $D > 10^7$ sur des jeux de données de taille $n < 10^2$. Les réseaux de neurones de très grande taille donc sur-paramétrés font, par nature, du *feature engineering* et permettent d'obtenir des performances de généralisations exceptionnelles (notamment en *Computer Vision*).

Dans la suite du chapitre, on introduira d'abord le DNNle plus simple, le Perceptron Multicouche (*Multi-Layers Perceptron*, MLP), nous expliquerons comment l'entraîner et enfin nous présenterons les principales techniques d'optimisation et de régularisation. Notons qu'un MLPest un modèle paramétrique où ω correspond aux différents paramètres du réseau que nous détaillerons.

2.2 De la régression linéaire aux DNN

2.2.1 Du modèle simple au neurone

Nous allons tout d'abord présenter 3 modèles paramétriques très simple avant d'expliquer le lien qui existe entre un de ces modèles simples et un neurone.

Modèle de Régression Linéaire. Reprenons les Notations 1 avec $k = 1$ et $\mathcal{Y} = \mathbb{R}$.

Modèle 1. ***[Régression Linéaire]***
Soit $\mathcal{W} = \mathbb{R}^p \times \mathbb{R}$. Considérons la famille de modèles linéaires paramétrée par $\omega = (\boldsymbol{w}, b) \in \mathcal{W}$, le modèle de régression linéaire associé est définit par :

$$\boldsymbol{Y} = f_\omega(\mathbf{X}) = \mathbf{X}\boldsymbol{w} + b\mathbb{1}_p, \tag{2.1}$$

où $\mathbb{1}_p$ désigne le vecteur unitaire de taille p. On appelera $\mathbf{w} = (w_j)_{j=1,\cdots,p}$ le vecteur des poids (weights) et b le biais.

Modèle de Régression Logistique. Reprenons les Notations 1 avec $k = 1$ et $\mathcal{Y} = \{0; 1\}$. Définissons s la fonction logistique :

$$s : \begin{array}{ccc} \mathbb{R} & \longrightarrow & (0,1) \\ x & \longmapsto & \frac{1}{1+e^{-x}} \end{array}$$

Modèle 2. ***[Régression Logistique]***
Soit $\mathcal{W} = \mathbb{R}^p \times \mathbb{R}$. Considérons la famille de modèles linéaires paramétrée par $\omega = (\boldsymbol{w}, b) \in \mathcal{W}$, le modèle de Régression Logistique associé est définit par :

$$\boldsymbol{Y} = f_\omega(\mathbf{X}) = s(\mathbf{X}\boldsymbol{w} + b\mathbb{1}_n). \tag{2.2}$$

Soulignons que dans la littérature Deep Learning, on appelle fonction sigmoïde la fonction notée S qui correspond à l'application terme à terme de s : $\forall p \in \mathbb{N}^*$, $\forall \mathbf{x} = (x_j)_{j=1,\cdots,p} \in \mathbb{R}^p$, $S(\mathbf{x}) = (s(x_j))_{j=1,\cdots,p}$. Le modèle logistique est la composition du modèle de régression linéaire par la fonction sigmoïde.

Modèle Perceptron Un modèle très proche du modèle logistique est le modèle Perceptron ([28]). Ce modèle est à la base des réseaux de neurones actuels ([23]). Avec les Notations 1 en prenant $\mathcal{X} = \{0; 1\}^p$ et $\mathcal{Y} = \{0; 1\}$, définissons le modèle Perceptron.

Modèle 3. ***[Perceptron]***
Soit $\mathcal{W} = \mathbb{R}^p \times \mathbb{R}$. Considérons la famille de modèles linéaires paramétrée par $\omega = (\boldsymbol{w}, b) \in \mathcal{W}$, le modèle de Régression Perceptron associé est définit par :

$$\boldsymbol{Y} = f_\omega(\mathbf{X}) = P(\mathbf{X}) = \left[\mathbb{1}_{\left[\mathbf{X}\boldsymbol{w}+\mathbb{1}_p b\right]_i>0}\right]_{i=1,\cdots,n} \tag{2.3}$$

Le Perceptron permet d'encoder des opérateurs de logique binaire (cf Table 2.1 et apprendre ainsi des règles logiques complexes de façon paramétrique.

la **Négation**	le **Ou Inclusif**	le **Ou Exclusif**	le **Et**
$\neg$	$\vee$	*XOR*	$\wedge$

TABLE 2.1 : Opérateurs de logique binaire.

Exemple 1.
On considère les Notations 1 avec $p = 2$, $k = 1$ telles que $\forall \omega = (w_1, w_2) \in \mathbb{R}^2$, $\forall b \in \mathbb{R}$. Pour toute observation $(\boldsymbol{x}_i, y_i)$ telle que $\boldsymbol{x}_i = (x_{i1}, x_{i2})^\top \in \{0;1\}^2$ et $y_i \in \{0;1\}$, définissons les 3 événements suivants :

$$A = \{x_{i1} = 1\},\ B = \{x_{i2} = 1\} \text{ et } C = \{y_i = 1\}.$$

Le Perceptron appliqué à l'observation i est

$$P(x_{i1}, x_{i2}) = \mathbb{1}_{w_1 x_{i1} + w_2 x_{i2} + b > 0}.$$

Les valeurs des paramètres (w_1, w_2, b) donnés en exemple dans le tableau 2.2 permettent d'encoder des relations.

w_1	w_2	b	$\Rightarrow$	Relation logique
2	0	−1	$\Rightarrow$	$C = A$
−2	0	1	$\Rightarrow$	$C = \neg A$
2	2	−3	$\Rightarrow$	$C = A \wedge B$
2	2	−1	$\Rightarrow$	$C = A \vee B$
2	−2	−1	$\Rightarrow$	$C = A \wedge \neg B$

TABLE 2.2 : Encodage des relations logiques. Les valeurs présentées ici sont une solution particulière, il existe une infinité de triplets solutions de ces problèmes.

Remarque 1. *Soulignons une limite de ce modèle dans l'exemple précèdent. Il n'existe aucun $(w_1, w_2, b) \in \mathbb{R}^3$ capable d'encoder $C = AXORB$ connu sous le nom du problème du XOR détaillé section 2.2.1.*

A	B	$\mathbf{x} = (x_1, x_2)$	$\mathbf{xw} + b$	C
FALSE	FALSE	$(0, 0)$	-1	FALSE
TRUE	FALSE	$(1, 0)$	1	TRUE
FALSE	TRUE	$(0, 1)$	1	TRUE
TRUE	TRUE	$(1, 1)$	3	TRUE

TABLE 2.3 : Table de disjonction de cas pour la relation $\vee$ avec $w_1 = 2$, $w_2 = 2$ et $b = -1$.

Neurone. Nous pouvons maintenant définir le neurone en apprentissage profond. Il s'agit d'un algorithme qui pour $p \in \mathbb{N}^*$

1. Prend en **entrée** un vecteur $\boldsymbol{u} \in \mathbb{R}^p$.
2. Multiplie ce vecteur par un vecteur de poids $\mathbf{w} \in \mathbb{R}^p$ et ajoute un biais $b \in \mathbb{R}$.
3. Applique une fonction d'activation $\sigma(\cdot)$ pour renvoyer en **sortie** le résultat v.

TABLE 2.4 : Neurone

$$\boldsymbol{u} \longmapsto v = s(\boldsymbol{u}^T \mathbf{w} + b)$$

Ainsi les 3 modèles précédents répondent à cette définition : le modèle linéaire avec une fonction d'activation qui n'est autre que l'identité ; le modèle logistique avec la sigmoïde et le modèle Perceptron avec $s \equiv P$. Chaque neurone peut être vu comme un petit modèle simple et un réseaux de neurones est constitué d'un grande nombre de neurones qui "communiquent" entre eux.

Le problème du *XOR* Le développement suivant, bien qu'il soit trivial reste une référence historique au sein de la communauté de l'apprentissage profond, et a donc sa place au sein d'un chapitre à visée didactique. Reprenons l'exemple 1 où il est apparu qu'un modèle simple comme le Perceptron ne permet pas l'encodage du **ou exclusif** (*XOR*). Il est toutefois possible de l'encoder à partir en combinant plusieurs neurones de type Perceptron. En effet, en reprenant les événements définis dans l'exemple 1, on peut réécrire le *XOR* comme une combinaison des relations simples $\neg$, $\wedge$ et $\vee$ (*cf* Table 2.2) :

$$AXORB = (A \wedge \neg B) \vee (\neg A \wedge B) := D_1 \vee D_2, \tag{2.4}$$

où $D_1 := A \wedge \neg B$ et $D_2 := \neg A \wedge B$. Ces événements intermédiaires correspondent à deux modèles de perceptrons

$$D_1 = \{P_1(x_{i1}, x_{i2}) = 1\} = \{\mathbb{1}_{w_{1,1}x_{i1}+w_{1,2}x_{i2}+b_1>0} = 1\} \quad \text{avec} \quad (w_{1,1}, w_{1,2}, b_1) = (2, -2, -1)$$

$$D_2 = \{P_2(x_{i1}, x_{i2}) = 1\} = \{\mathbb{1}_{w_{2,1}x_{i1}+w_{2,2}x_{i2}+b_2>0} = 1\} \quad \text{avec} \quad (w_{2,1}, w_{2,2}, b_2) = (-2, 2, -1)$$

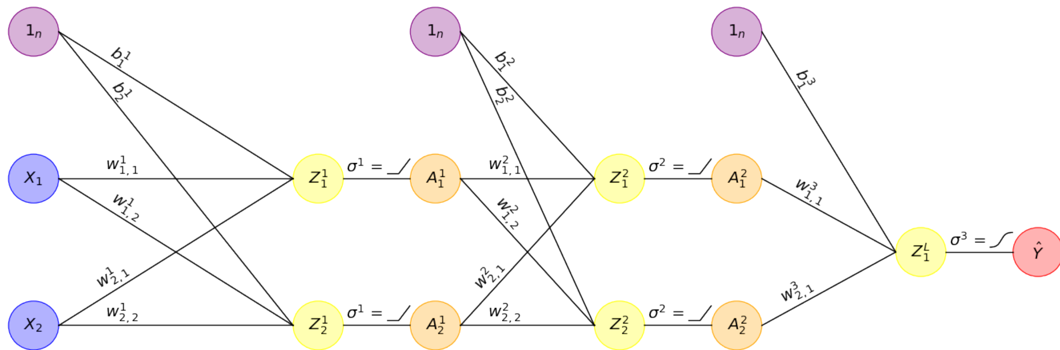
En combinant les sorties de 2 précédents Perceptrons dans un troisième nous obtenons un encodage du XOR : $AXORB = D_1 \vee D_2$

$$AXORB = \{P_3(P_1(x_{i1}, x_{i2}), P_2(x_{i1}, x_{i2})) = 1\} \quad \text{avec} \quad (w_{3,1}, w_{3,1}, b_3) = (2, 2, -1)$$

Ainsi, par composition on a obtenu une règle logique impossible à encoder par un perceptron unique. En composant de façon successive des perceptrons, on peut ainsi encoder toutes les propositions logiques obtenues par composition d'opérations élémentaires $\neg$, $\vee$ et $\wedge$, et ce quel que soit le nombre de variables considérées $X_1, X_2, X_3, \cdots$. Cet exemple va nous permettre d'introduire dans la section 2.2.2, le modèle Perceptron Multicouche 4.

2.2.2 Le modèle Perceptron Multicouche

Avant de définir le modèle Perceptron Multicouche (*Multilayer Perceptron*,MLP), introduisons le vocabulaire élémentaire propre aux réseaux de neurones.



- **Input** : Entrée du réseau, c'est-à-dire une observation $\mathbf{x}_i = (x_{i1}, \cdots, x_{ip})$.
- **Output** : Sortie du réseau, c'est-à-dire la *target* $y_i = (y_{i1}, \cdots, y_{ik})$.

- **Neuron** : .Algorithme définit table 2.2.1
- **Hidden Layer** : Couche cachée composée d'un nombre de neurones.
- **Input Layer** : Couche d'entrée des **Input**.
- **Output Layer** : Dernière couche.
- **Depth :** Profondeur du réseau, *i.e.* le nombre de couches cachées plus la dernière couche (de sortie).
- **Size :** On note D la taille totale du réseau, c'est-à-dire le nombre de paramètres (les différents poids et biais) du réseau.

Remarque 2.

- *On distingue les différentes couches en fonction du nombre d'étapes de composition qui les séparent de la sortie du réseau.*
- *Ainsi, la dernière couche cachée est constituée de l'ensemble des neurones dont la sortie est une des entrées des neurones de la couche de sortie. La couche précédente est l'avant dernière couche cachée, etc.*
- ***Attention**, on ne considère pas la dernière couche comme une couche cachée.*

Définissons le modèle Perceptron Multicouche (Multilayer Percetron, MLP) de **profondeur** $L \in \mathbb{N}^*$:

- Chaque couche l, $l = 0, \cdots, L$ est constituée d'un nombre $J_l \in \mathbb{N}^*$ de neurones, on dit qu'elle est de largeur J_l.
- L'**Input Layer** est constituée des **Input** et est donc de taille $J_0 = p$.
- La dernière couche (**Output Layer**) est de largeur $J_L = k$.

Le modèle MLP est définit de façon récursive ci-dessous.

Modèle 4. ***[Perceptron Multicouche de taille** D**]***

Soit $\omega = \left\{(W^l, \boldsymbol{b}^l)\right\}_{l=1,\cdots,L} \in \mathcal{W} \subseteq \mathbb{R}^D$*, l'ensemble des paramètres du modèle, respectivement les poids et le biais de la couche* l *avec*

$$\forall l = 1, \cdots, L \quad \begin{cases} W^l = [w^l_{j,k}]_{\substack{j=1,\cdots,J_{l-1} \\ k=1,\cdots,J_l}} \in \mathbb{R}^{J_{l-1} \times J_l} \\ \boldsymbol{b}^l = (b^l_k)_{k=1,\cdots,J_l} \in \mathbb{R}^{J_l} \end{cases}$$

Soit un ensemble de fonctions d'activations $\left\{S^l : \mathbb{R}^{n\times J_l} \longmapsto \mathbb{R}^{n\times J_l}\right\}_{l=1,\cdots,L}$, *avec les Notations 1, on définit le MLP de la façon suivante*

$$\forall l = 1,\cdots,L \quad \begin{cases} \mathbf{A}^0 = \mathbf{X} \\ \mathbf{Z}^l = \mathbf{A}^{l-1}W^l + \boldsymbol{b}^l\mathbb{1}^T \in \mathbb{R}^{n\times J_l} \\ \mathbf{A}^l = S_l(\mathbf{Z}^l) \in \mathbb{R}^{n\times J_l} \\ \mathbf{A}^L = \mathbf{Y} \end{cases}$$

*La couche de sortie (**Output layer**) est telle que* $\mathbf{A}^L = \mathbf{Y}$.

Remarque 3. *Posons :*

$$\forall l = 1,\cdots,L \quad \begin{cases} \mathbf{Z}^l = [z^l_{i,k}]_{\substack{i=1,\cdots,n \\ k=1,\cdots,J_l}} = \mathbf{A}^{l-1}W^l + \boldsymbol{b}^l\mathbb{1}^T \in \mathbb{R}^{n\times J_l} \\ \mathbf{A}^l = [a^l_{i,k}]_{\substack{i=1,\cdots,n \\ k=1,\cdots,J_l}} = S_l(\mathbf{Z}^l) \end{cases}$$

Fixons une couche $l = 1,\cdots,L$ *, la fonction notée* S_l *correspond à l'application terme à terme de la fonction d'activation* $s^l : \mathbb{R} \longrightarrow \mathbb{R}$ *à toutes les entrées* $\{z_{i,k}\}_{i,k}$ *:*

$$\forall i = 1,\cdots,n,\ \forall k = 1,\cdots,J_l \quad \begin{cases} z^l_{i,k} = \sum_{j=1}^{J_{l-1}} a^{l-1}_{i,j} w^l_{j,k} + b^l_k \\ \\ a^l_{i,k} = s^l(z^l_{i,k}) \end{cases}$$

Le "Ou exclusif" avec le MLP. Ci-dessous le *XOR* encodé avec un modèle MLP.

On peut aller bien au delà des seules règles de logiques binaires. L'une des forces majeures des réseaux de neurones profonds est en effet leur propriété théorique d'**approximateurs universels** ([97, 47]), capacité des réseaux de neurones à générer de la complexité à partir d'éléments simples. Considérons maintenant que le modèle appartient à une classe inconnue de fonctions. Sous certaines hypothèses peu restrictives sur cette classe, il a été montré qu'il est possible d'approximer la fonction associée au modèle sous-jacent de façon satisfaisante par un réseau de neurones de taille suffisamment grande. Citons les travaux de [97, 47] sur ce sujet qui considèrent des réseaux à une couche et la

fonction d'activation *Sigmoïde*. Par la suite, [177] ont montré qu'avec la fonction d'activation *ReLU*, augmenter la profondeur plutôt que la largeur réduit le nombre de paramètres nécessaires à une bonne approximation.

2.3 Phase d'apprentissage

On va maintenant présenter rapidement les principales procédures pour estimer les paramètres (poids et biais) d'un réseau de neurones. Une des principales forces des réseaux de neurones repose sur le fait que c'est un modèle différentiable, *i.e.* il est possible de calculer le gradient de la fonction à minimiser selon chacun des paramètres du réseau. De plus, même pour des réseaux de très grande taille, l'expression sous forme de composition de fonction d'un réseau de neurones permet aux librairies de *differential-programming* telles que *pytorch* ou *tensorflow* de calculer automatiquement les dérivées associées. Enfin, l'expression de ces opérations sous forme vectorielle, matricielle et tensorielle rend fortement parallélisable les calculs, ce qui permet d'exploiter les accélérateurs de calculs vectoriels (*e.g.* GPU et TPU, voir plus bas).

2.3.1 Fonction de perte et objectif

On s'intéresse maintenant aux procédures pour estimer, $\omega^* \in \Omega$, l'ensemble des paramètres associés au réseau de neurones $f_{\omega^*} \in \mathcal{F}$, correspondant à la meilleure approximation du modèle sous-jacent f. Pour cela, on définit une fonction de perte, $\mathbf{L}$, où avec les Notations 1, pour toute observation $(\mathbf{x}_i, y_i)$

$$\mathbf{L} : \begin{array}{ll} \mathbb{R}^k \times \mathbb{R}^k & \longrightarrow \mathbb{R} \\ (y_i, f_\omega(\mathbf{x}_i)) & \longmapsto \mathbf{L}(y_i, f_\omega(\mathbf{x}_i)) \end{array}$$

telle que $\mathbf{L}(y_i, f_\omega(\mathbf{x}_i))$ désigne l'erreur associée à la prédiction de y_i par $f_\omega(\mathbf{x}_i)$. Le problème d'optimisation se traduit alors par la minimisation du critère $\mathbf{C}$ suivant :

$$\begin{aligned} \omega^* &= \arg\min_{\omega\in\Omega} \mathbf{C}(\mathbf{X}, \mathbf{Y}, \omega) \\ &= \arg\min_{\omega\in\Omega} \sum_{i=1}^{n} \mathbf{L}(y_i, f_\omega(\mathbf{x}_i)). \end{aligned}$$

Notons que l'égalité, vient du fait de l'unicité supposé de cet optimum.
Les fonctions de perte choisies en pratique dépendent de la tâche à résoudre et de la fonction d'activation utilisée en couche de sortie. Pour un problème de régression, on utilise généralement le risque quadratique, associé à la fonction d'activation de sortie identité.

Fonction de Perte 1. ***[Régression : RMSE + Identité]***
Notations 1 pour $k = 1$, $\mathcal{Y} = \mathbb{R}$ et la fonction d'activation $s^L(\cdot) = Id.$ en sortie, on optimise le critère suivant :

$$\boldsymbol{C}(\mathbf{X}, \mathbf{Y}, \omega) = \|\mathbf{Y} - f_\omega(\mathbf{X})\|_2^2 = \sum_{i=1}^{n} (y_i - z_i^L)^2$$

où $z_i^L = f_\omega(\boldsymbol{x}_i)$

Dans un problème de classification binaire, on cherche à la fois à estimer le label $y \in \{0, 1\}$ qui est une valeur discrète et la probabilité $\mathbb{P}[Y = 1 \,|\, X = x] \in [0, 1]$, qui est une valeur continue. En apprentissage profond, l'espace d'arrivée du réseau de neurone dépend de s^L, la fonction d'activation sur la couche de sortie.
En pratique, on utilise généralement la fonction d'activation *Sigmoïde s* lorsque l'on veut prédire la probabilité :

$$f_\omega(\mathbf{x}) = a^L = s(z^L) \in (0, 1).$$

Pour prédire directement la valeur du label y, on remplacera la fonction *Sigmoïde* par la fonction d'activation indicatrice :

$$\widehat{y} = \mathbb{1}_{z^L > 0} = \mathbb{1}_{f_\omega(\mathbf{x}) > 0.5}.$$

Durant l'entraînement, on utilise généralement la fonction de perte *Cross-Entropie* (*CE*), composée à la fonction d'activation *Sigmoïde* de sortie (cf Perte 2).

Fonction de Perte 2. ***[Classification Binaire : CE + Sigmoïde]***
Soit $\mathcal{Y} = \{0; 1\}$ et s la fonction Sigmoïde, on optimise le critère suivant :

$$\begin{aligned}\boldsymbol{C}(\mathbf{X}, \mathbf{Y}, \omega) &= \mathbf{Y}^T \ln(f_\omega(\mathbf{X})) + (\mathbb{1}_n - \mathbf{Y})^T \ln(\mathbb{1}_n - f_\omega(\mathbf{X})) \\ &= \sum_{i=1}^{n} \mathbb{1}_{y_i=1} \ln(s(z_i^L)) + \mathbb{1}_{y_i=0} \ln(1 - s(z_i^L))\end{aligned}$$

Ici, $\mathbb{1}_n$ est le vecteur unitaire de taille n.

Dans un problème de classification multi-classes avec un nombre de classes $k > 2$, on cherche à estimer $y \in \{1, \cdots, k\}$. Au lieu de se limiter à la prédiction du label (*i.e.* $J^L = 1$), il est possible de prédire les probabilités de chaque classe

$(\mathbb{P}[y = k \,|\, X = x])_{k=1,\cdots k} \in [0,1]^k$. Pour cela, on va appliquer la transformation suivante, appelée *One-Hot-Encoding* :

$$\mathrm{OHE}(y) = [\mathbb{1}_{y=k}]_{k=1,\cdots,k}.$$

On note alors la matrice des labels encodés de la façon suivante :

$$\mathbf{Y}_{\mathrm{OHE}} := \begin{bmatrix} \mathrm{OHE}(y_1)^T \\ \vdots \\ \mathrm{OHE}(y_n)^T \end{bmatrix}.$$

Les quantités $\mathrm{OHE}(y)$ et $[\mathbb{P}[\mathrm{OHE}(y)_k = 1 \,|\, X = x]]_{k=1,\cdots k}$ sont toutes deux de dimension k, on choisit alors le nombre de neurones sur la dernière couche égale au nombre de problèmes de classification binaire : $J^L = k$, chaque neurone correspondant à une classe. Il est important de lier ces k prédictions, en effet,

$$\sum_{k=1}^{k} \mathrm{OHE}(y)_k = \sum_{k=1}^{k} \mathbb{P}[\mathrm{OHE}(y)_k = 1 \,|\, X = x] = 1.$$

Il faut donc pouvoir imposer $\forall \mathbf{x} \in \mathcal{X}, \sum_{k=1}^{k} a_k^L = 1$. Pour cela, nous appliquons la fonction d'activation *Softmax* sur la couche de sortie. Cette dernière a pour particularité de ne pas être appliquée terme à terme aux éléments de $\mathbf{z}^L$, mais au vecteur lui-même :

$$\begin{array}{rcl} \mathbb{R}^k & \longmapsto & (0,1)^k \\ \sigma \;:\; \mathbf{z} & \longmapsto & \sigma(\mathbf{z}) = \left(\frac{e^{z_k^L}}{\sum_{k'=1}^{k} e^{z_{k'}^L}} \right) \end{array}$$

On a clairement, $\forall \mathbf{z}^L \in \mathbb{R}^k, \mathbf{a}^L \in (0,1)^k$ et $\sum_{k=1}^{k} a_k^L = 1$. En utilisant la fonction d'activation *Softmax*, on peut donc prédire les probabilités d'appartenance à chacune des classes. Pour prédire directement la classe de y, il suffit de remplacer la fonction *Softmax* par la fonction d'activation arg max :

$$\widehat{y} = \arg\max_{k=1,\cdots,k} z_k^L.$$

Durant l'entraînement pour la multi-classes, on utilise généralement la *CE* composée là encore avec la fonction d'activation de sortie *Softmax*.

Fonction de Perte 3. ***[Classification Multi-Classe : CE + softmax]***
Soit $k > 2$, $\mathcal{Y} = \{0, \cdots, k\}$ *et* σ *la fonction Softmax, on optimise le critère*

suivant :

$$\begin{aligned} C(\mathbf{X}, \mathbf{Y}, \omega) &= Tr\Big(\ln(f_\omega(\mathbf{X}))\mathbf{Y}_{OHE}^T\Big) \\ &= \sum_{i=1}^{n}\sum_{k=1}^{k} \mathbb{1}_{y_i=k} \ln(\sigma(z_i^L)_k) \end{aligned}$$

Ici Tr désigne la trace d'une matrice.

Dans le reste de ce chapitre, on se placera dans le cadre de la régression, en utilisant le critère RMSE et la fonction d'activation identité sur la couche de sortie.

2.3.2 Calcul des dérivées partielles et Rétro-Propagation

Les dérivées partielles de la fonction de perte du modèle Perceptron Multicouche selon les paramètres du modèle (poids et biais) possèdent deux caractéristiques qui rendent très efficace leur calcul numérique :

Mutualisation. Les dérivées partielles des paramètres sur une couche cachée $l = 1, \cdots, L$ peuvent s'exprimer en fonction des dérivées partielles de la couche supérieure $(l+1)$. Ainsi, lorsque l'on calcule le vecteur gradient du biais $(\nabla_{\mathbf{b}^l}\mathbf{L})$ et la matrice gradient des poids $(J_{\mathbf{W}^l}\mathbf{L})$ d'une couche l, on obtient de façon récursive les dérivées partielles de tous les poids et biais des couches supérieures $(> l)$.

Parallélisation. On dispose pour le gradient de la fonction de perte **évaluée en une unique observation** d'une expression sous forme matricielle faisant intervenir uniquement des opérations termes à termes et des produits vectoriels. Cela permet d'exploiter les accélérateurs de calculs vectoriels tels que les GPU (pour *Graphical Processing Units*) et TPU (pour *Tensor Processing Units*), des composants informatiques capables d'effectuer un grand nombre d'opérations vectorielles en parallèle. Sans rentrer dans les détails, on peut donner l'ordre de grandeur suivant : La machine principalement utilisée durant cette thèse possède un CPU disposant de 32 coeurs, en pratique il lui faut environ autant de temps pour entraîner une forêt aléatoire de 32 arbres qu'un seul arbre de décision et mais deux fois plus de temps pour entraîner une forêt aléatoire de 33 arbres. Cette machine possède aussi une carte graphique disposant de 544 *tensor cores*, en pratique il lui faut environ autant de temps pour entraîner un réseau de neurones dont toutes les couches cachées sont de largeur 2 ou 256.

On suppose la fonction de perte $\mathbf{L}$ et toutes les fonctions d'activations s différentiables en tout point. L'ensemble des paramètres du réseau (Modèle 4), noté

$\omega = \left\{(W^l, \mathbf{b}^l)\right\}_{l=1,\cdots,L} \in \mathbb{R}^D$, sont respectivement les poids W^l et le biais $\mathbf{b}^l$ des couches $l = 1, \cdots, L$. Fixons l'observation $(\mathbf{x}_i, y_i)$, $i = 1, \cdots, n$ alors le réseau peut être décrit de façon récursive de la façon suivante :

$$\begin{cases} \mathbf{a}_i^0 = \mathbf{x}_i \quad \text{et} \quad \mathbf{z}_i^l = (\mathbf{a}_i^{l-1})^T W^l + \mathbf{b}^l \in \mathbb{R}^{J_l} \\ \mathbf{a}_i^l = S_l(\mathbf{z}_i^l) \quad \text{et} \quad \mathbf{a}_i^L = f_\omega(\mathbf{x}_i) = y_i \end{cases}$$

Pour une observation i donnée, on considère la fonction de perte comme fonction des paramètres du modèle

$$w \longmapsto \mathbf{L}(y_i, f_\omega(\mathbf{x}_i)) = (y_i - f_\omega(\mathbf{x}_i))^2.$$

Ainsi, pour tout $l = 1, \cdots, L$, les dérivées partielles de la fonction de perte par rapport au biais $\mathbf{b}^l$ et au poids $\mathbf{W}^l$ sont le vecteur et la matrice notés respectivement

$$\nabla_{\mathbf{b}^l}\mathbf{L}(y_i, f_\omega(\mathbf{x}_i)) = \left[\frac{d\mathbf{L}(y_i, f_\omega(\mathbf{x}_i))}{db_j^l}\right]_{j=1,\cdots,J^l} \quad \text{et} \quad J_{\mathbf{W}^l}\mathbf{L}(y_i, f_\omega(\mathbf{x}_i)) = \left[\frac{d\mathbf{L}(y_i, f_\omega(\mathbf{x}_i))}{dw_{j,k}^l}\right]_{\substack{j=1,\cdots,J_{l-1} \\ k=1,\cdots,J_l}}.$$

Il est possible d'obtenir l'expression des dérivées partielles sous la forme d'une récursion descendante. Définissons tout d'abord

$$\delta^l := \nabla_{\mathbf{z}_i^l}\mathbf{L}(y_i, f_\omega(\mathbf{x}_i)) = \left(\frac{d\mathbf{L}(y_i, f_\omega(\mathbf{x}_i))}{dz_{i,j}^l}\right)_{j=1,\cdots,J^l} \qquad \forall l = 1, \cdots, L. \tag{2.5}$$

Comme $\forall k = 1, \cdots, J^l$

$$z_{i,k}^l = \sum_{j=1}^{J^{l-1}} w_{j,k}^l a_{i,j}^{l-1} + b_k^l \quad \forall l = 1, \cdots, L, \tag{2.6}$$

il vient que $\forall j = 1, \cdots, J^l$: $\frac{dz_{i,k}^l}{db_j^l} = \mathbb{1}_{j=k}$ et

$$\frac{d\mathbf{L}(y_i, f_\omega(\mathbf{x}_i))}{db_j^l} = \sum_{k=1}^{J^l} \frac{d\mathbf{L}(y_i, f_\omega(\mathbf{x}_i))}{dz_{i,k}^l}\frac{dz_{i,k}^l}{db_j^l} = \sum_{k=1}^{J^l} \delta_k^l \mathbb{1}_{j=k} = \delta_j^l. \tag{2.7}$$

De même, $\forall j = 1, \cdots, J^{l-1}$, $\forall k = 1, \cdots, J^l$

$$\frac{dz_{i,k'}^l}{dw_{j,k}^l} = a_{i,j}^{l-1}\mathbb{1}_{k=k'}, \quad \forall k' = 1, \cdots, J^l \tag{2.8}$$

Ainsi en utilisant la notation de l'équation (2.5), on en déduit

$$\frac{d\mathbf{L}(y_i, f_\omega(\mathbf{x}_i))}{dw^l_{j,k}} = \sum_{k'=1}^{J^l} \frac{d\mathbf{L}(y_i, f_\omega(\mathbf{x}_i))}{dz^l_{i,k'}} \frac{dz^l_{i,k'}}{dw^l_{j,k}} = \sum_{k'=1}^{J^l} \delta^l_{k'} a^{l-1}_{i,j} \mathbb{1}_{k=k'} = \delta^l_k a^{l-1}_{i,j}. \tag{2.9}$$

Nous avons donc les deux quantités suivantes :

$$\nabla_{\mathbf{b}^l} \mathbf{L}(y_i, f_\omega(\mathbf{x}_i)) = \delta^l \quad \text{et} \quad J_{\mathbf{W}^l} \mathbf{L}(y_i, f_\omega(\mathbf{x}_i)) = \mathbf{a}^{l-1}_i \left(\delta^l\right)^T = \mathbf{a}^{l-1}_i \left(\nabla_{\mathbf{b}^l} \mathbf{L}(y_i, f_\omega(\mathbf{x}_i))\right)^T.$$

De plus, $\forall l = 1, \cdots, L$, on a : $\mathbf{a}^l_i = S_l(\mathbf{z}^l_i)$, c'est-à-dire

$$a^l_{i,k} = s^l(z^l_{i,k}), \quad \forall k = 1, \cdots, J^l \tag{2.10}$$

Par définition de δ^l donnée équation (2.5), il vient

$$\begin{aligned} \delta^{l-1}_j &:= \frac{d\mathbf{L}(y_i, f_\omega(\mathbf{x}_i))}{dz^{l-1}_{i,j}} = \sum_{k=1}^{J^l} \frac{d\mathbf{L}(y_i, f_\omega(\mathbf{x}_i))}{dz^l_{i,k}} \frac{dz^l_{i,k}}{dz^{l-1}_{i,j}} \\ &= \sum_{k=1}^{J^l} \delta^l_k \frac{dz^l_{i,k}}{dz^{l-1}_{i,j}} = \sum_{k=1}^{J^l} \delta^l_k \sum_{j'=1}^{J^{l-1}} \frac{dz^l_{i,k}}{da^{l-1}_{i,j'}} \frac{da^{l-1}_{i,j'}}{dz^{l-1}_{i,j}}. \end{aligned}$$

Or, on a par l'équation (2.10) et l'équation (2.6)

$$\frac{da^{l-1}_{i,j'}}{dz^{l-1}_{i,j}} = \frac{ds^{l-1}(z^{l-1}_{i,j'})}{dz^{l-1}_{i,j'}} \mathbb{1}_{j=j'} \quad \text{et} \quad \frac{dz^l_{i,k}}{da^{l-1}_{i,j'}} = w^l_{j',k}.$$

Ainsi,

$$\delta^{l-1}_j = \sum_{k=1}^{J^l} \delta^l_k \sum_{j'=1}^{J^{l-1}} w^l_{j',k} \frac{ds^{l-1}(z^{l-1}_{i,j'})}{dz^{l-1}_{i,j'}} \mathbb{1}_{j=j'} = \sum_{k=1}^{J^l} \delta^l_k w^l_{j,k} \frac{ds^{l-1}(z^{l-1}_{i,j})}{dz^{l-1}_{i,j}} \tag{2.11}$$

En *Deep Learning*, on appelle cette procédure de calcul en cascade, des dérivées partielles selon chacun des paramètres du réseau, **Rétro-Propagation** (*Back-Propagation*). En pratique, les librairies *pytorch* et *tensorflow* vont garder en mémoire les quantités intermédiaires du graphe de dérivation, qui pourront être "recyclées" lorsque l'on calculera conjointement les gradients des paramètres du réseau.
Afin de traduire sous forme matricielle la procédure de *Back-Propagation*, introduisons certaines notations.

[Produit terme à terme.] Soient **A** et **B** deux matrices réelles de $\mathcal{M}(\mathbb{R}^{r\times q})$:

$$\mathbf{A} = [a_{j,k}]_{\substack{j=1,\cdots,r\\k=1,\cdots,q}} \qquad \mathbf{B} = [b_{j,k}]_{\substack{j=1,\cdots,r\\k=1,\cdots,q}}$$

On note leur produit terme à terme $\odot$ tel que

$$\mathbf{A}\odot\mathbf{B} = [a_{j,k}\times b_{j,k}]_{\substack{j=1,\cdots,r\\k=1,\cdots,q}} .$$

[Produit matricielle orienté.] Soit un entier $L \in \mathbb{N}^*$, $\forall l = 1,\cdots,L$, pour tout entier $J_l \geq 1$ et pour toute matrice réelle $\mathbf{M}_l$ de $\mathcal{M}(\mathbb{R}^{r\times q})$, on définit le produit matricielle orienté, noté $\prod$ comme le produit indicé de gauche à droite, tel que

$$\prod_{l=1}^{L}\mathbf{M}_l = \mathbf{M}_1\times\mathbf{M}_2\times\cdots\times\mathbf{M}_{L-1}\times\mathbf{M}_L.$$

[Fonction d'activation terme à terme.] Fixons une couche $l = 1,\cdots,L$, la fonction notée S_l correspond à l'application terme à terme de la fonction d'activation $s^l : \mathbb{R}\longrightarrow\mathbb{R}$ à toutes les entrées du vecteur $\mathbf{z}_i \in \mathbb{R}^{J^l}$:

$$S_l(\mathbf{z}_i) = \left[s^l(z_{i,j})\right]_{j=1\cdots,J^l} .$$

On notera $S_l^{(1)}$ le gradient des dérivées partielles terme à terme de la fonction S_l, tel que :

$$S_l^{(1)}(\mathbf{z}_i^l) = \left(\frac{ds^l(z_{i,j}^l)}{dz_{i,j}^l}\right)_{j=1\cdots,J^l} .$$

Expression matricielle du gradient. Commençons par réécrire l'équation (2.11) sous forme matricelle : Pour tout $l = 1,\cdots,L$ on a

$$\delta_j^{l-1} = \sum_{k=1}^{J^l}\delta_k^l w_{j,k}^l\frac{ds^{l-1}(z_{i,j}^{l-1})}{dz_{i,j}^{l-1}} \iff \underbrace{\delta^{l-1}}_{J^{l-1}} = \left(\underbrace{\mathbf{W}^l}_{J^{l-1}\times J^l}\odot\underbrace{[S_{l-1}^{(1)}(\mathbf{z}_i^{l-1})\mathbb{1}_{J^l}^T]}_{J^{l-1}\times J^l}\right)\underbrace{\delta^l}_{J^l} .$$

Considérons $l = L$, la fonction de perte RMSE et la fonction d'activation identité, alors

$$\delta^L = \nabla_{\mathbf{b}^L}\mathbf{L}(y_i, f_\omega(\mathbf{x}_i)) = \nabla_{\mathbf{z}_i^L}\mathbf{L}(y_i, f_\omega(\mathbf{x}_i)) = \nabla_{\mathbf{z}_i^L}[(y_i - \mathbf{z}_i^L)^2] = -2\mathbf{z}_i^L \qquad (2.12)$$

En combinant les deux dernières équations, on obtient en appliquant une récursion que $\forall l = 1, \cdots, L-1$

$$\delta^l = \nabla_{\mathbf{b}^l}\mathbf{L}(y_i, f_\omega(\mathbf{x}_i)) = -2\left[\prod_{l'=l+1}^{L}\left(\mathbf{W}^{l'} \odot \left(S^{(1)}_{l'-1}(\mathbf{z}_i^{l'-1})\mathbb{1}^T_{J_{l'}}\right)\right)\right]\mathbf{z}_i^L \tag{2.13}$$

Nous avions déjà réécrit l'équation (2.9) sous forme matricielle : pour tout $l = 1, \cdots, L$ on obtient

$$J_{\mathbf{W}^l}\mathbf{L}(y_i, f_\omega(\mathbf{x}_i)) = \delta^l(\mathbf{a}_i^{l-1})^T.$$

Ainsi, par récursion, $\forall l = 1, \cdots, L-1$

$$J_{\mathbf{W}^l}\mathbf{L}(y_i, f_\omega(\mathbf{x}_i)) = -2\left[\prod_{l'=l+1}^{L}\left(\mathbf{W}^{l'} \odot \left(S^{(1)}_{l'-1}(\mathbf{z}_i^{l'-1})\mathbb{1}^T_{J_{l'}}\right)\right)\right]\mathbf{z}_i^L(\mathbf{a}_i^{l-1})^T. \tag{2.14}$$

Considérons la couche de sortie $l = L$, la fonction de perte RMSE et la fonction d'activation identité, alors on a :

$$\nabla_{w_j^L}\mathbf{L}(y_i, f_\omega(\mathbf{x}_i)) = \left(\nabla_{\mathbf{z}_i^L}\mathbf{L}(y_i, f_\omega(\mathbf{x}_i))\right) \times \left(\nabla_{w_j^L}\mathbf{z}_i^L\right) = \left(-2\mathbf{z}_i^L\right) \times \left(\mathbf{a}_i^{L-1}\right)$$

et

$$J_{\mathbf{W}^L}\mathbf{L}(y_i, f_\omega(\mathbf{x}_i)) = -2\mathbf{z}_i^L\,\mathbf{a}_i^{L-1} = -2y_i\,\mathbf{a}_i^{L-1} \tag{2.15}$$

la dernière égalité découlant du choix de la fonction d'activation (l'identité) de la dernière couche qui implique $\mathbf{z}_i^L = y_i$.

2.3.3 Descente de gradient

Pour déterminer des paramètres appropriés lors de la phase d'apprentissage, on cherche à minimiser un critère donné $\mathbf{C}(\mathbf{X}, \mathbf{Y}, \cdot)$:

$$\begin{aligned}\omega^* &= \arg\min_{\omega\in\Omega}\mathbf{C}(\mathbf{X}, \mathbf{Y}, \omega)\\ &= \arg\min_{\omega\in\Omega}\sum_{i=1}^{n}\mathbf{L}(y_i, f_\omega(\mathbf{x}_i)).\end{aligned}$$

Pour se faire, la méthode d'optimisation la plus utilisée en pratique pour optimiser les paramètres d'un réseau de neurones est la descente de gradient. Il s'agit d'une méthode itérative où les paramètres sont réévalués à chaque itération. Plus précisément, on choisit le nombre d'itérations $I \in \mathbb{N}^*$ à effectuer et on introduit $\omega(e)$ les paramètres du réseau à chaque itération $e \in \{0, \cdots, I\}$, tels que

$$\omega(e) = \left\{\left\{\mathbf{W}^l(e), \mathbf{b}^l(e)\right\}\right\}_{l=1,\cdots,L} \in \Omega.$$

À l'issu de la procédure, on prend $\widehat{\omega} = \omega(I)$.

Initialisation. On initialise tous les paramètres $\omega(0)$ selon une procédure aléatoire ou déterministe, appelée technique d'initialisation. De nombreuses techniques ont été proposées, on présentera ici la méthode [71], pour la fonction d'activation sigmoïde :

Initialisation_ 1. [*Glorot*]

Pour chaque couche $l = 1, \cdots, L$, $\mathbf{b}^l(0)$ est nul et les entrées de la matrice de poids de $\mathbf{W}^l$ sont indépendantes et identiquement distribuées telles que :

$$\begin{aligned} w^l_{1,1}(0) &\sim \mathcal{U}\left(0, \frac{1}{\sqrt{J^l}}\right) \\ \mathbf{b}^l(0) &= \mathbf{0}_{J^l} \end{aligned}$$

On discutera dans la dernière section de ce chapitre les caractéristiques des principales techniques d'initialisation utilisées en pratique.

Entraînement. La descente de gradient est une méthode itérative où pour tout $e = 1, \cdots, I$, la fonction $\omega(e)$ est fonction de $\omega(e-1)$ et des dérivées partielles du critère selon $\omega(e-1)$. Cela peut paraître contre-intuitif d'avoir recours à une méthode basée sur le gradient en *Deep Learning*, puisque dans ce cadre la majorité des conditions usuelles ne sont pas remplies. En effet, la fonction minimisée est fortement non convexe et en pratique, on constate la présence de points selles et de minima locaux. Qui plus est, il n'existe aucune garanti que le minimum global, s'il existe, est unique. Néanmoins, de très nombreuses procédures basées sur la descente de gradient permettent d'obtenir une convergence vers un ensemble de paramètres satisfaisant, malgré ces difficultés. Elles reposent toutes sur trois étapes : Propagation-Avant, Rétro-Propagation et Mise-à-jour.

Propagation-Avant_ 1. [*GD*]

Partons du modèle Percetron Multi-Couches 4 :$\forall l = 1, \cdots, L$,

$$\begin{cases} \mathbf{Z}^l(e) = \mathbf{A}^{l-1}(e)W^l(e-1) + \mathbf{b}^l(e-1)\mathbf{1}_n^T \\ \mathbf{A}^l(e) = S_l(\mathbf{Z}^l(e)) \end{cases}$$

Cette étape consiste alors simplement à calculer $\mathbb{Z}(e) := \left\{\mathbf{Z}^l(e)\right\}_{l=1,\cdots,L}$ et $\mathbb{A}(e) := \left\{\mathbf{A}^l(e)\right\}_{l=1,\cdots,L}$ avec les paramètres $\omega(e-1)$.

Rétro-Propagation_ 1. [*GD*]

On procède à une Rétro-Propagation sur les paramètres du réseau, afin d'obtenir les gradients des poids et biais sur chaque couche, en utilisant les équations (2.12),(2.13), (2.15) et (2.14). Pour alléger les notations, les gradients à l'itération $e = 1, \cdots, I$ seront notés $\mathbb{G}(e) := \left\{\mathbf{G_W}^l(e), \mathbf{g_b}^l(e)\right\}_{l=1,\cdots,L}$ avec $\forall l = 1, \cdots, L$

$$\begin{cases} \mathbf{G_W}^l(e) & := & J_{\mathbf{W}^l(e-1)}\mathbf{C}(\mathbf{X}, \mathbf{Y}, \omega(e-1)) \\ \mathbf{g_b}^l(e) & := & \nabla_{\mathbf{b}^l(e-1)}\mathbf{C}(\mathbf{X}, \mathbf{Y}, \omega(e-1)) \end{cases}$$

Mise-à-jour_ 1. [*GD*]

On modifie les paramètres du réseau en appliquant une correction qui dépend de $\mathbb{G}(e)$, tel que pour tout $l = 1, \cdots, L$

$$\begin{cases} \mathbf{W}^l(e) & = & \mathbf{W}^l(e-1) - \eta \frac{\mathbf{G_W}^l(e)}{\|\mathbf{G_W}^l(e)\|_2 + \delta} \\ \mathbf{b}^l(e) & = & \mathbf{b}^l(e-1) - \eta \frac{\mathbf{g_b}^l(e)}{\|\mathbf{g_b}^l(e)\|_2 + \delta} \end{cases}$$

Ici, $\eta \in \mathbb{R}_+^*$ est le *learning rate* (taux d'apprentissage) et $\delta = 10^{-8}$ est une constante de stabilité numérique.

Apprentissage_ 1. [*GD*]

Pour la descente de gradient usuelle, les étapes d'initialisation, de Propagation-Avant, de Rétro-Propagation et de Mise-à-jour, sont utilisées de la façon suivante :

- **Initialisation_** 1 : On détermine $\omega(0)$.
- **Entraînement** : A chaque itération $e = 1, \cdots, I$:
  - **Propagation-Avant_** 1 : On calcule $\mathbb{Z}(e)$ et $\mathbb{A}(e)$.
  - **Rétro-Propagation_** 1 : On calcule $\mathbb{G}(e)$.
  - **Mise-à-jour_** 1 : On calcule $\omega(e)$.

Précisons que dans la descente de gradient, on optimise le critère $\mathbf{C}(\mathbf{X}, \mathbf{Y}, \omega)$, et ainsi, à chaque itération on calcule les gradients pour toutes les observations du jeu de données simultanément avant d'effectuer une mise à jour.

2.3.4 Optimizers et learning rate schedulers

De très nombreuses procédures d'apprentissage ont été proposées, elles sont appelées en *Deep Learning* des *optimizers* . Le choix de l'*optimizer* a un impact crucial sur la solution obtenue ([188]). Comme indiqué précédemment, l'objectif optimisé est fortement non-convexe, et donc le choix de l'*optimizer* a un impact non seulement sur le temps de calcul, mais également sur la stabilité de la méthode et sur la qualité de la solution obtenue.

Dans le reste de cette section, on va présenter les 7 autres *optimizers* suivants :

- **Stochastic Gradient Descent** (**SGD** [37]),
- **Mini-Batch Gradient Descent** (**Mini-batchGD**) ([111]),
- **Momentum** ([161]),
- **Nesterov** ([143, 196])
- **AdaGrad** ([55]),
- **RMSProp** ([76],
- **Adam** ([113]).

Notons que l'*optimizer* **Adam** est utilisé dans les expériences présentées dans les Chapitres 4 et 5 de cette thèse.

On superpose souvent aux *optimizers* des *learning rate schedulers* (ce qui pourrait se traduire par planificateurs du taux d'apprentissage). Il s'agit de techniques utilisées pour faire varier la taille des pas au cours de l'apprentissage, selon une formule prédéfinie, c'est à dire que le *learning rate* est évolutif mais pas adaptatif. On présentera d'abord 2 *learning rate schedulers*, **LinearLR** et **OneCycleLR** (([187])).

Impact du learning rate
➢ Le *learning rate* joue un rôle très important dans la dynamique d'apprentissage ([186, 189, 188, 219]). Un *learning rate* trop petit augmente le nombre d'époques nécessaires. Un *learning rate* trop grand empêche l'algorithme de converger. Cependant, on constate souvent en pratique que plus le *learning rate* est élevé, plus forte est la généralisation ([186]). Celui-ci joue donc le rôle de paramètre de régularisation, qu'il convient de calibrer. Notons que lorsque les gradients sont renormalisés, l'impact de ce paramètre dépend des normes des matrices de poids et des vecteurs de biais sur chaque couche. On utilise le terme de ***learning rate***

effectif pour désigner la pondération apportée au poids, qui dépend du *learning rate* η et de l'*optimizer* choisi.

Puisqu'un *learning rate* élevé améliore la généralisation mais qu'un *learning rate* faible facilite la convergence, il semble naturel de vouloir faire varier la valeur du *learning rate* au cours de l'apprentissage. Pour se faire, un certain nombre de techniques, appelées *learning rate schedulers* ont été proposées, on présentera ici l'une des plus simples, **LinearLR** et l'une des plus efficaces, **OneCycleLR**. Notons que **OneCycleLR** est notamment utilisée dans les expériences présentées dans les Chapitres 4 et 5.

Linear Scheduler

➢ Le principe de **LinearLR** est élémentaire : la valeur du *learning rate* décroît linéairement au cours de l'apprentissage, en partant d'une valeur qui favorise la généralisation ($\eta_{\text{init}} = 0.1$), pour arriver à une valeur qui facilite la convergence ($\eta_E = 0.05$).

Learning Rate_ 1. [*LinearLR*]

On fixe la valeur du *learning rate* à la première époque et à la dernière époque, notés respectivement $\eta_{\text{init}} \in \mathbb{R}_+^*$ et $\eta_{\text{end}} \in \mathbb{R}_+^*$. On définit ensuite la suite arithmétique des *learning rates* $(\eta_e)_{e=1,\cdots,E}$ de la façon suivante : $\forall e = 1, \cdots, E$,

$$\eta_e = \left(1 - \frac{e-1}{E-1}\right)\eta_{\text{init}} + \frac{e-1}{E-1}\eta_{\text{end}} \in \mathbb{R}_+^*.$$

One Cycle LR Scheduler

➢ La technique très simple présentée précédemment permet d'obtenir généralement de meilleurs résultats qu'avec un *learning rate* constant, il est néanmoins possible de faire bien mieux. Parmi toutes les heuristiques proposées, l'une d'entre elles est particulièrement efficace, il s'agit du **OneCycleLR** ([187]). Le principe est de décomposer l'apprentissage en deux phases de même durée ($E/2$), l'une où le *learning rate* croit rapidement d'une valeur initiale $\eta_{\text{init}} \in \mathbb{R}_+^*$ vers son apogée $\eta_{\max} \in \mathbb{R}_+^*$, l'autre où le *learning rate* décroît progressivement vers une valeur finale $\eta_{\text{end}} \in \mathbb{R}_+^*$ bien plus faible que la valeur initiale, c'est à dire $\eta_{\text{end}} \ll \eta_{\text{init}}$. Cette technique permet notamment de réduire considérablement le nombre d'époques nécessaires pour obtenir la convergence vers une valeur satisfaisante. Notons que cette méthode ne repose sur aucun fondement théoriquement et n'a d'autre justification que ses bons résultats en pratique.

Learning Rate_ 2. [*OneCycleLR*]

On choisit d'abord le nombre total d'époque E, puis la taille de la première phase $E_{\max} \in \mathbb{N}$ avec $1 < E_{\max} < E - 1$, on prend généralement $E_{\max} = \lfloor E/2 \rfloor$, ici $\lfloor \cdot \rfloor$ correspond à l'arrondi à l'entier inférieur. Ensuite, on choisit la valeur du *learning rate* à son apogée $\eta_{\max} \in \mathbb{R}_+^*$, la valeur du *learning rate* initial $\eta_{\text{init}} \in \mathbb{R}_+^*$ et la valeur du *learning rate* final $\eta_{\text{end}} \in \mathbb{R}_+^*$ avec $\eta_{\text{end}} \ll \eta_{\text{init}} \ll \eta_{\max}$. On définit enfin la suite des *learning rates* $(\eta_e)_{e=1,\cdots,E}$, en appliquant la règle suivante :

$$\begin{cases} \eta_e & := & \eta_{\max} + \frac{\eta_{\text{init}} - \eta_{\max}}{2}\left(1 + \cos\left(\pi \frac{e-1}{E_{\max}-1}\right)\right) & \forall e = 1, \cdots, E_{\max}, \\ \eta_e & := & \eta_{\text{end}} + \frac{\eta_{\max} - \eta_{\text{end}}}{2}\left(1 + \cos\left(\pi \frac{e-E_{\max}-1}{E-E_{\max}-1}\right)\right) & \forall e = E_{\max}+1, \cdots, E. \end{cases}$$

Descente de Gradient Stochastique (SGD).

➢ Rappelons que dans la méthode de descente de gradient présentée précédemment, à chaque itération on calcule **simultanément** pour toutes les observations $i = 1, \cdots, n$ du jeu de données les gradients de $\mathbf{C}(\mathbf{Y}, \omega)$ avant d'effectuer une mise à jour des paramètres. En revanche, la descente de gradient stochastique (**SGD**) consiste à calculer **séquentiellement** les gradients de $\mathbf{L}(y_i, f_\omega(\mathbf{x}_i))$ selon chaque observation $i = 1, \cdots, n$ et à effectuer une mise à jour entre chaque calcul.
Dans la terminologie utilisée en *Deep Learning*, on introduit le terme d'époque, qui se distingue du terme itération de la façon suivante :

- On appelle **époque** chaque séquence $e = 1, \cdots, E$ de n itérations, de tel sorte qu'à chaque époque on parcourt l'intégralité du jeu de données et ainsi, quand on dit avoir effectuer E époques, on a fait $n \times E$ mis à jour. De plus, à chaque époque, on parcourt le jeu de données dans un ordre aléatoire et indépendant des autres époques.

- On appelle **itération** $i = 1, \cdots, I$, avec $= n \times E$, où on effectue successivement, pour un unique couple $(\mathbf{x}_i, y_i)$: une Propagation-Avant, une Rétro-Propagation et une Mise-à-jour des paramètres.

À chaque époque $e = 1, \cdots, E$, de n itérations, on a :

$$\left| \begin{array}{l} \omega(e, 0) = \omega(e-1) \\ \omega(e, e') = \left\{ \left\{ \mathbf{W}^l(e, e'), \mathbf{b}^l(e, e') \right\} \right\}_{l=1,\cdots,L} \in \Omega, \quad \forall e' = 1, \cdots, n \\ \omega(e) := \omega(e, n) \end{array} \right.$$

où les sont des paramètres intermédiaires $\omega(e, e') \in \Omega$ correspondent aux paramètres de l'étape $e' = 0, \cdots, n$ de l'époque e.

Propagation-Avant_ 2. [*SGD*]

Le jeu de données étant parcouru dans un ordre aléatoire et indépendant à chaque époque e, on introduit des permutations $\pi_e \in \mathfrak{S}_n$ (l'ensemble des permutations de n points) sur les observations de l'ensemble de données initial. Pour une époque donnée e, à l'étape $e' = 1, \cdots, n$, on applique le modèle Perceptron Multicouche 4 à une unique observation $(\mathbf{x}_{\pi(e')}, y_{\pi(e')})$, avec les paramètres $\omega(e, e'-1)$; *i.e.*

$$\forall l = 1, \cdots, L, \quad \begin{cases} \mathbf{Z}^l(e, e') = \mathbf{A}^{l-1}(e, e')W^l(e, e'-1) + \mathbf{b}^l(e, e'-1)\mathbf{1}_n^T \\ \mathbf{A}^l(e, e') = S_l(\mathbf{Z}^l(e, e')) \end{cases}$$

Rétro-Propagation_ 2. [*SGD*]

On procède à une Rétro-Propagation sur les paramètres du réseau, afin d'obtenir les gradients des poids et biais sur chaque couche, en utilisant les équations (2.12),(2.13), (2.14) et (2.15). Pour alléger les notations, les gradients à l'itération $e' = 1, \cdots, n$ seront notés $\mathbb{G}(e, e') = \{\mathbf{G_W}^l(e, e'), \mathbf{g_b}^l(e, e')\}_{l=1,\cdots,L}$ avec $\forall l = 1, \cdots, L$

$$\begin{cases} \mathbf{G_W}^l(e, e'-1) & := & J_{\mathbf{W}^l(e,e')}\mathbf{L}(y_{\pi(e')}, f_{\omega(e,e'-1)}(\mathbf{x}_{\pi(e')})) \\ \mathbf{g_b}^l(e, e') & := & \nabla_{\mathbf{b}^l(e-1)} J_{\mathbf{W}^l(e,e')}\mathbf{L}(y_{\pi(e')}, f_{\omega(e,e'-1)}(\mathbf{x}_{\pi(e')})) \end{cases}$$

Mise-à-jour_ 2. [*SGD*]

On procède de façon analogue à la descente de gradient usuelle, à ceci près qu'on utilise les gradients $\mathbb{G}(e, e')$ avec $\forall l = 1, \cdots, L$:

$$\begin{cases} \mathbf{W}^l(e, e') = \mathbf{W}^l(e, e'-1) - \eta \frac{\mathbf{G_W}^l(e,e'-1)}{\|\mathbf{G_W}^l(e,e'-1)\|_2 + \delta} \\ \mathbf{b}^l(e, e') = \mathbf{b}^l(e, e'-1) - \eta \frac{\mathbf{g_b}^l(e,e'-1)}{\|\mathbf{g_b}^l(e,e'-1)\|_2 + \delta} \end{cases}$$

Apprentissage_ 2. [*SGD*]

On procède de façon analogue à la descente de gradient usuelle, à ceci près qu'on effectue une boucle sur les observations à chaque époque :

- **Initialisation_** 1 : On détermine $\omega(0)$.
- **Entraînement** : A chaque époque $e = 1, \cdots, E$:
  - On génère aléatoirement π_e.
  - On initialise $\omega(e, 0) = \omega(e-1)$.
  - Pour $e' = 1, \cdots, n$,
    - **Propagation-Avant_** 2 : On calcule $\mathbb{Z}(e, e')$ et $\mathbb{A}(e, e')$.
    - **Rétro-Propagation _** 2 : On calcule $\mathbb{G}(e, e')$.
    - **Mise-à-jour_** 2 : On calcule $\omega(e, e')$.
  - On pose $\omega(e) = \omega(e, n)$.

Il y a plusieurs raisons qui viennent motiver l'emploi de la descente de gradient stochastique. D'abord, elle peut se justifier lorsqu'à chaque itération, on n'a pas accès à l'intégralité du jeu de données mais seulement à un sous-ensemble, ce sous-ensemble variant entre chaque itération. C'est le cas notamment quand le jeu de données est trop grand pour être chargé intégralement en mémoire, on parle alors de *Big Data* ("données massives" en français). Parfois, on fait même face à des volumes tels qu'il serait de toute façon trop long de calculer le gradient pour chaque couple $(\mathbf{x}_i, y_i)$ ne serait-ce qu'une seule fois, (on effectue alors moins d'une époque complète). A l'inverse, c'est aussi le cas quand, pour des raisons matérielles ou légales, il n'est pas possible de stocker le jeu de données, on parle alors d'apprentissage *online* ("en continue" en français) ; lorsque les données sont présentées à l'algorithme sous forme de flux. Chaque observation $(\mathbf{x}_i, y_i)$ est alors traitée une fois puis oubliée.

Parfois, même lorsque les contraintes matérielles ne l'impose pas, il peut être préférable d'utiliser la descente de gradient stochastique plutôt que la descente de gradient usuelle. En effet, le fait de changer entre chaque époque l'ordre dans lequel le jeu de données est parcouru, en appliquant une permutation aléatoire sans remise des indices, modifie la fonction optimisée à chaque itération $\mathbf{L}(y_{\pi(i)}, f_\omega(\mathbf{x}_{\pi(i)}))$. Cela permet d'éviter à la trajectoire des paramètres de repasser au même endroit à chaque époque, et de générer des oscillations dans la **dynamique d'apprentissage**. C'est ce procédé qui explique le caractère stochastique de cet *optimizer* . En d'autres termes, la valeur de la fonction de perte n'évoluera pas de façon monotone au fil des itérations. Cela peut permettre d'éviter de rester bloquer dans un minimum local peu satisfaisant, sans pour autant empêcher la convergence vers un optimum ([27]).

Mini-Batch Gradient Descent (Mini-batchGD).
➢ L'*optimizer* **Mini-batchGD** est un compromis entre la descente de gradient usuelle **GD** et la **SGD**. Le jeu de données est découpé sous-ensembles appelés *mini-batchs*. À l'instar de la **SGD**, à chaque époque, on procède de façon séquentielle, à ceci près que la descente ne parcourt pas le jeu de données en ne prenant qu'un élément à chaque fois mais un *mini-batchs* à la fois. En revanche à chaque itération le gradient est calculé simultanément pour tous les éléments d'un des *mini-batchs* comme dans la **GD** classique.
Pour effectuer ce découpage, on choisit la taille $b_s \in \mathbb{N}$ des *mini-batchs*, tel que $1 < b_s < n$ et on note $n_b = \lceil n/b_s \rceil$ le nombre de *mini-batchs*. Ici, $\lceil \cdot \rceil$ désigne l'arrondi à l'entier supérieur. Si n est divisible par b_s, chacun des *mini-batchs* sera de taille b_s. Dans le cas contraire, les $n_b - 1$ premiers *mini-batchs* seront de taille b_s et le dernier sera de taille $n - (n_b - 1) \times b_s$. Ainsi, on effectue n_b itérations par époque. Donc au total, on procède à $I = E \times n_b$ mises à jour des paramètres. Notons qu'à chaque époque, comme pour la **SGD**, on effectue une permutation aléatoire sans remise des indices avant d'effectuer le découpage.

Propagation-Avant_ 3. [*Mini-batchGD*]

À chaque époque, à l'étape $e' = 1, \cdots, n_b$, on définit le *batch* $\{\mathbf{X}_{e'}, Y_{e'}\}$ avec :

$$\begin{aligned}\mathbf{X}_{e'} &:= \left[x_{\pi(i),j}\right]_{\substack{i = (e'-1)\times b_s + 1, \cdots, \min(n, e'\times b_s) \\ j = 1, \cdots, p}}, \\ \mathbf{Y}_{e'} &:= (y_{\pi(i)},)_{i=(e'-1)\times b_s+1,\cdots,\min(n,e'\times b_s)}.\end{aligned}$$

On applique le modèle Perceptron Multicouche 4 en prenant $\mathbf{A}^0 = \mathbf{X}_{e'}$, avec les paramètres $\omega(e, e'-1)$ et tel que $\forall l = 1, \cdots, L$:

$$\begin{cases} \mathbf{Z}^l(e,e') = \mathbf{A}^{l-1}(e,e')W^l(e,e'-1) + \mathbf{b}^l(e,e'-1)\mathbf{1}_n^T \\ \mathbf{A}^l(e,e') = S_l(\mathbf{Z}^l(e,e')) \end{cases}$$

Rétro-Propagation_ 3. [*Mini-batchGD*]

On procède à une Rétro-Propagation sur les paramètres du réseau, afin d'obtenir les gradients des poids et biais $\mathbb{G}(e,e') = \left\{\mathbf{G_W}^l(e,e'), \mathbf{g_b}^l(e,e')\right\}_{l=1,\cdots,L}$ avec $\{\mathbf{X}_{e'}, Y_{e'}\}$ et tels que pour tout $l = 1, \cdots, L$:

$$\begin{cases} \mathbf{G_W}^l(e,e') &:= J_{\mathbf{W}^l(e,e'-1)}\mathbf{C}(\mathbf{X}_{e'}, Y_{e'}, \omega(e,e'-1)) \\ \mathbf{g_b}^l(e,e') &:= \nabla_{\mathbf{b}^l(e,e'-1)}\mathbf{C}(\mathbf{X}_{e'}, Y_{e'}, \omega(e,e'-1)) \end{cases}$$

Apprentissage_ 3. [*Mini-batchGD*]

On procède de façon analogue à la **SGD** mais en appliquant ensuite la **Mise-à-jour_** 2.

- **Initialisation_** 1 : On détermine $\omega(0)$.
- **Entraînement** : A chaque époque $e = 1, \cdots, E$ on effectue les étapes suivantes :
  - On génère aléatoirement π_e.
  - On initialise $\omega(e, 0) = \omega(e-1)$.
  - Pour $e' = 1, \cdots, n_b$,
    - **Propagation-Avant_** 3 : On calcule $\mathbb{Z}(e, e')$ et $\mathbb{A}(e, e')$.
    - **Rétro-Propagation _** 3 : On calcule $\mathbb{G}(e, e')$.
    - **Mise-à-jour_** 2 : On calcule $\omega(e, e')$.
  - On pose $\omega(e) = \omega(e, n_b)$.

La valeur de b_s a un impact majeur sur la dynamique d'apprentissage et les performances obtenues ([41]). Ce choix peut être contraint par le contexte, en effet, souvent en pratique l'un des facteurs limitant est la taille de la mémoire vive des cartes graphiques (*VRAM*). Il n'existe pas de règle universelle permettant de déterminer la valeur la plus appropriée pour b_s dans tous les cas.

Momentum

➢ Une façon simple de se représenter le fonctionnement des différents optimizers est de considérer la surface formée par la fonction de perte dans l'espace des paramètres Ω. On appelle cette surface le *loss landscape* (voir notamment les visualisations proposées par [126]). Durant la descente de gradient, chaque valeur de $\omega(e)$ correspond à une position dans l'espace Ω à laquelle on associe l'altitude correspondant à la fonction de perte. A chaque étape, on fait un pas dans la direction la plus pentue. Soulignons que cette direction ne dépend pas pas de l'inclinaison de la pente (puisque l'on renormalise le gradient), mais du *learning rate* . En *Deep Learning*, le *loss landscape* n'est pas une cuve lisse au fond de laquelle se trouve l'optimum, comme ce serait le cas pour une fonction convexe, mais plutôt une chaîne de montagne accidentée, pleine de creux, de bosses de cols et de plateaux, qu'il faudrait descendre pas à pas dans le noir, sans avoir d'autre information que la direction de la pente à notre position actuelle ([126, 69]). Il est aisé de comprendre qu'une personne s'arrêtant à chaque pas pour ensuite prendre la direction la plus pentue sous ses pieds s'immobilisera dès qu'elle arriverai sur une pierre

plate, ou repartira en arrière à la moindre bosse. En revanche, si cette personne décide de sauter d'une position à l'autre, en conservant l'élan des sauts précédents, cela lui permet de suivre la pente dominante malgré les variations locales du terrain.

L'*optimizer* **Momentum** (*inertie* en français) applique cette intuition, en choisissant à chaque étape de prendre en compte non seulement la direction de plus forte descente au point actuel mais aussi les directions aux pas précédents.

Inertie_ 1. [*Momentum*]

On introduit le terme d'inertie $\mathbb{V} = \{\mathbb{V}(e)\}_{e=0,\cdots,E} \in \Omega^{E+1}$ et on note $\alpha \in (0,1)$ le paramètre d'inertie. Précisons qu'à partir de maintenant on considérera qu'à chaque époque les gradients sont calculés pour toutes les observations simultanément (pas de *Batch-Learning*). Notons qu'en pratique on choisit généralement un niveau d'inertie élevé, par exemple $\alpha = 0.9$. On définit $\mathbb{V}(e) = \left\{\mathbf{V}_w^l(e), \mathbf{v}_b^l(e)\right\}_{l=1,\cdots,L} \in \Omega$ de façon récursive : $\forall l = 1, \cdots, L$,

$$\begin{cases} \mathbf{V}_w^l(0) & := & \mathbf{0}_{J^{l-1}\times J^l}, \\ \mathbf{V}_w^l(e) & := & \alpha \mathbf{V}_w^l(e-1) + \eta \frac{\mathbf{G_W}^l(e)}{\|\mathbf{G_W}^l(e)\|_2+\delta} & \forall e = 1, \cdots, E, \\ \mathbf{v}_b^l(0) & := & \mathbf{0}_{J^l}, \\ \mathbf{v}_b^l(e) & := & \alpha \mathbf{v}_b^l(e-1) + \eta \frac{\mathbf{g_b}^l(e)}{\|\mathbf{g_b}^l(e)\|_2+\delta} & \forall e = 1, \cdots, E. \end{cases}$$

Mise-à-jour_ 3. [*Momentum*]

Dans le cadre d'un **Momentum** , on procède de façon analogue à la descente de gradient usuelle, à ceci près qu'on utilise la règle de Mise-à-jour suivante, avec $\forall e = 1, \cdots, E, \forall l = 1, \cdots, L$:

$$\begin{cases} \mathbf{W}^l(e) = \mathbf{W}^l(e-1) - \mathbf{V}_w^l(e) \\ \mathbf{b}^l(e) = \mathbf{b}^l(e-1) - \mathbf{v}_b^l(e) \end{cases}$$

Apprentissage_ 4. [*Momentum*]

On commence par procéder de façon analogue à la **GD** , puis l'apprentissage se déroule de la façon suivante :

- **Initialisation_** 1 : On détermine $\omega(0)$.
- **Entraînement** : A chaque époque $e = 1, \cdots, E$,
  - **Propagation-Avant_** 1 (**GD**) : On calcule $\mathbb{Z}(e)$ et $\mathbb{A}(e)$.
  - **Rétro-Propagation_** 1 (**GD**) : On calcule $\mathbb{G}(e)$.
  - **Inertie_** 1 (**Momentum**) : On calcule $\mathbb{V}(e)$.
  - **Mise-à-jour_** 3 (**Momentum**) : On calcule $\omega(e)$.

Nesterov

➢ Reprenons notre comparaison avec la montagne. Le fait d'ajouter de l'inertie permet de dévaler le flanc d'une montagne sans s'arrêter au moindre relief. En revanche, une fois arrivé au creux de la vallée, l'élan accumulé peut nous pousser à continuer notre trajectoire pour remonter sur la montagne suivante, au lieu de changer de direction. Pour palier à ce problème, une solution est d'anticiper la direction de la pente au pas suivant, afin de ralentir pour empêcher cette remontée. L'*optimizer* **Nesterov** répond à ce problème en choisissant à chaque étape e de calculer le gradient non pas pour la position actuelle $\omega(e)$, mais à la position atteinte si l'on ajoute $\mathbb{V}(e)$, l'inertie actuelle.

Propagation-Avant_ 4. [*Nesterov*]

On conserve l'expression de l'inertie $\mathbb{V}(e)$ donnée précédement pour **Momentum** 1 et on définit $\omega_N(e)$, les paramètres anticipés avec *Nesterov* suivants avec $\forall e = 1, \cdots, E$,

$$\omega^N(e) = \left\{ \left\{ \mathbf{W}^l(e) + \alpha \mathbf{V}_w^l(e-1), \mathbf{b}^l(e) + \alpha \mathbf{v}_b^l(e-1) \right\} \right\}_{l=1,\cdots,L} \in \Omega.$$

On calcule $\mathbb{Z}(e) = \left\{ \mathbf{Z}^l \right\}_{l=1,\cdots,L}$ et $\mathbb{A}(e) = \left\{ \mathbf{A}^l \right\}_{l=1,\cdots,L}$ avec les paramètres $\omega_N(e)$, en utilisant l'expression donnée pour le modèle Perceptron Multicouche 4 : $\forall l = 1, \cdots, L$,

$$\begin{cases} \mathbf{Z}^l = \mathbf{A}^{l-1} \left(W^l(e-1) + \alpha \mathbf{V}_w^l(e-1) \right) + \left(\mathbf{b}^l(e-1) + \alpha \mathbf{v}_b^l(e-1) \right) \mathbf{1}_n^T \\ \mathbf{A}^l = S_l(\mathbf{Z}^l) \end{cases}$$

Rétro-Propagation_ 4. [*Nesterov*]

Les gradients $\mathbb{G}(e) = \{\mathbf{G_W}^l(e), \mathbf{g_b}^l(e)\}_{l=1,\cdots,L}$ sont définis par rapport à $\omega(e)$ mais avec la fonction $\mathbf{C}$ évaluée en $\omega_N(e)$ avec $\forall l = 1, \cdots, L$:

$$\begin{cases} \mathbf{G_W}^l(e) & := & J_{\mathbf{W}^l(e-1)} \mathbf{C}(\mathbf{X}, \mathbf{Y}, \omega_N(e-1)) \\ \mathbf{g_b}^l(e) & := & \nabla_{\mathbf{b}^l(e-1)} \mathbf{C}(\mathbf{X}, \mathbf{Y}, \omega_N(e-1)) \end{cases}$$

Apprentissage_ 5. [*Nesterov*]

Pour mettre en oeuvre **Nesterov** , on procède comme pour **Momentum** , à ceci près qu'on utilise l'expression des gradients $\mathbb{G}$ donnée précédemment.

- **Initialisation_** 1 : On détermine $\omega(0)$.
- **Entraînement** : A chaque époque $e = 1, \cdots, E$,
  - **Propagation-Avant_** 4 (**Nesterov**) : On calcule $\mathbb{Z}(e)$ et $\mathbb{A}(e)$.
  - **Rétro-Propagation_** 4 (**Nesterov**) : On calcule $\mathbb{G}(e)$.
  - **Inertie_** 1 (**Momentum**) : On calcule $\mathbb{V}(e)$.
  - **Mise-à-jour_** 3 (**Momentum**) : On calcule $\omega(e)$.

AdaGrad

➢ Les *optimizers* **Momentum** et **Nesterov** corrigent la direction des mises à jour pour tenir compte des trajectoires passées et futures respectivement. L'*optimizer* **AdaGrad** contrairement au deux autres, ne fait pas une normalisation du gradient mais repondère terme à terme les entrées du gradient en prenant en compte là encore la trajectoire passée en plus de la position actuelle. Il permet ainsi une correction adaptative du *learning rate effectif*, en pénalisant plus fortement les directions pour lesquelles la dérivée partielle aux itérations précédentes était élevée en valeur absolue, quelque soit son signe. Cela correspond dans notre analogie à introduire un terme de **friction**, pour tenir compte de l'aspect accidenté du terrain.

Friction_ 1. [*AdaGrad*]

Avec l'*optimizer* **AdaGrad** , on conserve l'expression de $\mathbb{G}(e)$ donnée pour la descente de gradient usuelle et on introduit le terme de friction $\mathbb{r} = \{\mathbb{r}(e)\}_{e=0,\cdots,E} \in \Omega^{E+1}$, où $\mathbb{r}(e) := \left\{\mathbf{R}_w^l(e), \mathbf{r}_b^l(e)\right\}_{l=1,\cdots,L} \in \Omega$ est défini de façon récursive : $\forall l = 1, \cdots, L$,

$$\begin{cases} \mathbf{R}_w^l(0) & := & \mathbf{0}_{J^{l-1} \times J^l}, \\ \mathbf{R}_w^l(e) & := & \mathbf{R}_w^l(e-1) + \mathbf{G_W}^l(e) \odot \mathbf{G_W}^l(e) \quad \forall e = 1, \cdots, E, \\ \mathbf{r}_b^l(0) & := & \mathbf{0}_{J^l}, \\ \mathbf{r}_b^l(e) & := & \mathbf{r}_b^l(e-1) + \mathbf{g_b}^l(e) \odot \mathbf{g_b}^l(e) \qquad \forall e = 1, \cdots, E. \end{cases}$$

La règle de Mise-à-jour nécessite l'introduction des nouvelles notations suivantes.

Notations 2.
Soient d_1 et d_2 deux entiers non nuls. On définit $(\cdot)^{\sqrt{\odot}}$, l'opérateur racine carré terme à terme tel que pour toute matrice $\boldsymbol{A} \in \mathbb{R}_+^{d_1 \times d_2}$ et pour tout vecteur $\boldsymbol{b} \in \mathbb{R}_+^{d_1}$, on a :

$$\begin{aligned} \boldsymbol{A}^{\sqrt{\odot}} &:= [\sqrt{a_{i,j}}]_{i,j}, \\ \boldsymbol{b}^{\sqrt{\odot}} &:= \left(\sqrt{b_i}\right)_i. \end{aligned}$$

Notations 3.
Soient d_1 et d_2 deux entiers non nuls.On définit $\oslash$, la division terme à terme $\oslash$, telle que pour toute matrice $\boldsymbol{A} \in \mathbb{R}^{d_1 \times d_2}$, $\boldsymbol{B} \in \mathbb{R}_{+*}^{d_1 \times d_2}$ et pour tout vecteur $\boldsymbol{c} \in \mathbb{R}^{d_1}$ et $\boldsymbol{d} \in \mathbb{R}_{+*}^{d_1}$, on a :

$$\begin{aligned} \boldsymbol{A} \oslash \boldsymbol{B} &:= \left[\frac{a_{i,j}}{b_{i,j}}\right]_{i,j}, \\ \boldsymbol{c} \oslash \boldsymbol{d} &:= \left(\frac{c_i}{d_i}\right)_i. \end{aligned}$$

Mise-à-jour_ 4. [*AdaGrad*]

La règle de Mise-à-jour pour **AdaGrad** est la suivante : $\forall l = 1, \cdots, L$:

$$\begin{cases} \mathbf{W}^l(e) = \mathbf{W}^l(e-1) - \eta\, \mathbf{G_W}^l(e) \oslash \left(\mathbf{R}_w^l(e)^{\sqrt{\odot}} + \delta \mathbb{1}_{J^{l-1} \times J^l}\right) \\ \mathbf{b}^l(e) = \mathbf{b}^l(e-1) - \eta\, \mathbf{g_b}^l(e) \oslash \left(\mathbf{r}_b^l(e)^{\sqrt{\odot}} + \delta \mathbf{1}_{J^l}\right) \end{cases}$$

Cette règle de Mise-à-jour permet d'appliquer, pour paramètre d'une couche, une normalisation différente qui dépend du cumul des carrés des dérivées partielles précédentes. Ainsi, un paramètre dont les dérivées partielles aux étapes précédentes étaient plus élevées sera divisé par une valeur plus grande qu'un paramètre pour lequel les dérivées partielles précédentes étaient très petites. **AdaGrad** modifie non seulement la norme mais également la direction

Apprentissage_ 6. [*AdaGrad*]

Pour mettre en oeuvre **AdaGrad** , on procède comme pour la descente de gradient usuelle **GD** , à ceci près qu'on utilise la règle de Mise-à-jour donnée précédemment.

- **Initialisation_** 1 : On détermine $\omega(0)$.
- **Entraînement** : A chaque époque $e = 1, \cdots, E$,
  - **Propagation-Avant_** 1 (**GD**) : On calcule $\mathbb{Z}(e)$ et $\mathbb{A}(e)$.
  - **Rétro-Propagation_** 1 (**GD**) : On calcule $\mathbb{G}(e)$.
  - **Friction_** 1 (**AdaGrad**) : On calcule $\mathbb{r}(e)$.
  - **Mise-à-jour_** 4 (**AdaGrad**) : On calcule $\omega(e)$.

RMSProp

➢ Avec **AdaGrad** , chaque élément du terme de friction $\mathbb{r}$ ne décroît jamais durant l'apprentissage. L'inconvénient est donc que l'on a un ralentissement de la trajectoire au cours du temps puisque les gradients sont divisés par des termes de plus en plus grands.

Friction_ 2. [*RMSProp*]

L'*optimizer* **RMSProp** pealie à cet inconvénient en introduisant un facteur d'oubli[1] $\gamma \in (0, 1)$. Le terme de friction $\mathbb{r}$ s'exprime alors de la façon suivante : $\forall l = 1, \cdots, L$,

$$\begin{cases} \mathbf{R}_w^l(0) & := & \mathbf{0}_{J^{l-1} \times J^l}, \\ \mathbf{R}_w^l(e) & := & \gamma\, \mathbf{R}_w^l(e-1) + (1-\gamma)\, \mathbf{G_W}^l(e) \odot \mathbf{G_W}^l(e) \quad \forall e = 1, \cdots, E, \\ \mathbf{r}_b^l(0) & := & \mathbf{0}_{J^l} \\ \mathbf{r}_b^l(e) & := & \gamma\, \mathbf{r}_b^l(e-1) + (1-\gamma)\, \mathbf{g_b}^l(e) \odot \mathbf{g_b}^l(e) \qquad \forall e = 1, \cdots, E. \end{cases}$$

Apprentissage_ 7. [*RMSProp*]

Pour mettre en oeuvre **RMSProp** , on conserve la règle de Mise-à-jour utilisée avec **AdaGrad** 4.

[1] Notons qu'en pratique on choisit généralement un niveau d'oubli assez faible, c'est à dire que $\gamma = 0.9$ est proche de 1.

• **Initialisation_** 1 : On détermine $\omega(0)$.
• **Entraînement** : A chaque époque $e = 1, \cdots, E$,
  • **Propagation-Avant_** 1 (**GD**) : On calcule $\mathbb{Z}(e)$ et $\mathbb{A}(e)$.
  • **Rétro-Propagation_** 1 (**GD**) : On calcule $\mathbb{G}(e)$.
  • **Friction_** 2 (**RMSProp**) : On calcule $\mathbb{r}(e)$.
  • **Mise-à-jour_** 4 (**AdaGrad**) : On calcule $\omega(e)$.

Adam

➢ L'*optimizer* **Adam** combine les effets d'inertie et de normalisation adaptative introduits par **Momentum** et **RMSProp**. En poursuivant l'analogie avec la descente d'un flanc de montagne, cela correspond à combiner inertie et friction, c'est à dire que l'on conserve en partie la même direction qu'aux étapes précédentes, mais que l'on se détourne des directions dans lesquelles on oscille fortement d'avant en arrière. [113] utilise la notation $(\beta_1, \beta_2) \in (0,1)^2$ pour désigner les taux d'oubli pour l'inertie et la friction, ce qui correspond respectivement aux termes α et γ des *optimizers* **Momentum** et **RMSProp**.

Inertie_ 2. [*Adam*]

Avec **Adam**, le terme d'inertie $\mathbb{V}$, auquel on associe le taux d'oubli β_1, est défini de façon récursive : $\forall l = 1, \cdots, L$,

$$\begin{cases} \mathbf{V}^l_w(0) & := & \mathbf{0}_{J^{l-1} \times J^l}, \\ \mathbf{V}^l_w(e) & := & \beta_1 \mathbf{V}^l_w(e-1) + (1-\beta_1)\mathbf{G_W}^l(e) \quad \forall e = 1, \cdots, E, \\ \mathbf{v}^l_b(0) & := & \mathbf{0}_{J^l}, \\ \mathbf{v}^l_b(e) & := & \beta_1 \mathbf{v}^l_b(e-1) + (1-\beta_1)\mathbf{g_b}^l(e) \quad \forall e = 1, \cdots, E. \end{cases}$$

Contrairement à **Momentum**, les les gradients sont multipliés par $(1-\beta_1)$, plutôt que par *learning rate* η, et ne sont par renormalisés. Les gradients sont repondérés lors de la Mise-à-jour décrite ci-dessous, qui utilise la friction **RMSProp** 2 avec le taux d'oubli β_2 au lieu de γ.

Mise-à-jour_ 5. [*Adam*]

Ainsi, **Adam** conserve les avantages de **Momentum** et **RMSProp**. De plus, il corrige un biais vers 0 présent avec **RMSProp** lors des premières itérations. En effet, à la première étape, $\mathbb{V}$ et $\mathbb{r}$ sont initialisés à 0, alors on a respectivement pour **RMSProp** et **Adam** par exemple

$$\begin{cases} \mathbf{V}^l_w(1) = \alpha\, \mathbf{G_W}^l(1) & \text{et} \quad \mathbf{R}^l_w(1) = (1-\gamma)\mathbf{G_W}^l(1) \odot \mathbf{G_W}^l(1), \\ \mathbf{V}^l_w(1) = (1-\beta_1)\mathbf{G_W}^l(1) & \text{et} \quad \mathbf{R}^l_w(1) = (1-\beta_2)\mathbf{G_W}^l(1) \odot \mathbf{G_W}^l(1). \end{cases}$$

On observe un biais vers 0 présent avec **RMSProp** pour les premières itérations lors de la Mise-à-jour. Comme β_1 et β_2 sont généralement proches de 1, (les auteurs recommandent $\beta_1 = 0.9$ et $\beta_2 = 0.99$, valeurs que nous utilisons également dans les Chapitres 4 et 5 de cette thèse), ce biais est corrigé en divisant, lors de la Mise-à-jour, $\mathbb{V}$ et $\mathbb{r}$ par $(1-\beta_1)$ et $(1-\beta_2)$ respectivement :

$$\begin{cases} \mathbf{W}^l(e) = \mathbf{W}^l(e-1) - \frac{\eta}{1-\beta_1}\mathbf{V}^l_w(e) \oslash \left(\frac{1}{1-\beta_2}\mathbf{R}^l_w(e)^{\sqrt{\odot}} + \delta\mathbb{1}_{J^{l-1}\times J^l}\right) \\ \mathbf{b}^l(e) = \mathbf{b}^l(e-1) - \frac{\eta}{1-\beta_1}\mathbf{v}^l_b(e) \oslash \left(\frac{1}{1-\beta_2}\mathbf{r}^l_b(e)^{\sqrt{\odot}} + \delta\mathbf{1}_{J^l}\right) \end{cases}$$

Apprentissage_ 8. [*Adam*]

Pour mettre en oeuvre **Adam**, on utilise l'expression du terme d'inertie donnée pour **Adam** 2, l'expression du terme de friction donnée pour **RMSProp** 2 et la règle de Mise-à-jour donnée pour **Adam** 5.

- **Initialisation_** 1 : On détermine $\omega(0)$.
- **Entraînement** : A chaque époque $e = 1, \cdots, E$,
  - **Propagation-Avant_** 1 (**GD**) : On calcule $\mathbb{Z}(e)$ et $\mathbb{A}(e)$.
  - **Rétro-Propagation_** 1 (**GD**) : On calcule $\mathbb{G}(e)$.
  - **Inertie_** 2 (**Adam**) : On calcule $\mathbb{V}(e)$.
  - **Friction_** 2 (**RMSProp**) : On calcule $\mathbb{r}(e)$.
  - **Mise-à-jour_** 5 (**Adam**) : On calcule $\omega(e)$.

2.4 Techniques d'apprentissage et de régularisation

L'objectif de cette section est de fournir au lecteur des clés de compréhension pour appréhender les spécificités du problème de généralisation dans le contexte du *Deep Learning*, et ainsi saisir l'originalité de l'approche proposée dans le Chapitre 4 pour traiter cette problématique. Pour cela, présentons quelques techniques développées cette dernière décennie pour améliorer, dans le cadre de l'apprentissage supervisé, les performances de généralisation du modèle Perceptron Multicouche 4.

Soulignons d'abord que ce qui suit ne constitue pas un état de l'art exhaustif des dernières avancées en *Deep Learning*. Dans ce domaine extrêmement actif, le terme "SOTA" (pour *State-Of-The-Art*) est souvent utilisé non pas pour désigner l'ensemble des apports historiques dans cette discipline, mais comme adjectif pour qualifier la dernière technique à date permettant d'obtenir un gain quantitatif

sur l'un des benchmarks ("*leaderboards*") reconnus ([165, 116, 197]). Ces benchmarks, initialement conçus pour évaluer empiriquement la capacité d'une méthode à résoudre une tâche générique (classification d'image, estimation de densité, etc.) sont désormais des compétitions féroces. Il serait à la fois vain et redondant de détailler ici ce qui se fait de mieux actuellement dans chaque domaine du *Deep Learning* car d'une part cette section serait obsolète d'ici quelques mois et d'autre part des outils comme Papers[212] existent déjà. Cette plateforme, mise à jour régulièrement, répertorie et classe les apports de la communauté selon leurs performances sur chacun des benchmarks. Notons qu'on trouvera néanmoins dans le Chapitre 4, la bibliographie *état-de-l'art* des avancées les plus contemporaines dans le contexte spécifique abordé : "l'application des réseaux de neurones aux données tabulaires en apprentissage supervisé".

2.4.1 Spécificités du Deep Learning

En *Deep Learning*, une place importante est accordée à la phase d'apprentissage, plus encore qu'à la conception de nouvelles architectures (*i.e.* des nouveaux modèles [192]). Contrairement à d'autres domaines, ou cette phase peut sembler secondaire ou relevant de considérations techniques ou algorithmiques, en *Deep Learning* la façon dont les paramètres du réseau de neurones ont été entraînés conditionne bien plus les performances de l'estimateur obtenu que l'architecture utilisée ([188]).

Régularisation explicite et implicite

➢ Une première explication est que, souvent en *Deep Learning*, atteindre l'objectif global pour la fonction de perte n'est ni possible en pratique ni même souhaitable. À l'instar du *Machine Learning*, où l'objectif à minimiser peut être modifié (*e.g.* pénalité L_2 pour éviter l'$overfitting$), on procède également en *Deep Learning* à des régularisations dites "*explicite*". En apprentissage profond, on distingue cette forme de régularisation "*explicite*", d'une seconde appelée régularisation *implicite* ([224, 144, 167, 190]). Cette dernière consiste à modifier non pas la fonction optimisée pour obtenir un optimum global différent, mais à changer de méthode d'optimisation, pour converger vers un optimum local différent. Le choix de cet optimum peut dépendre de critères extrinsèques tels que la performance sur l'échantillon de validation, ou intrinsèques tels que la forme de la surface autour de cet optimum. On privilégie par exemple les vallées ("*valleys of low loss*") aux puits ("*isolated modes*") ([126, 69]). Il n'est pas nécessaire que le minimum local vers lequel on a convergé soit sélectionné à l'issu de la descente de gradient. Par exemple,

les paramètres finaux sélectionnés peuvent être ceux d'une étape précédente, si ces derniers correspondent à une meilleure performance sur l'échantillon de validation (méthode *Early-Stopping* [127, 218]).

Architecture et loss landscape

➢ Une seconde explication est que le modèle Perceptron Multicouche est un approximateur universel (voir Section 2.1). Ainsi, l'introduction d'une nouvelle architecture ou d'une technique d'optimisation n'est généralement pas motivée par la réduction/augmentation l'expressivité du réseau pour mieux coller au modèle sous-jacent, la classe de fonctions paramétriques associées au réseau contenant déjà une approximation satisfaisante du modèle à estimer. La difficulté est d'identifier cette dernière via une méthode de gradient. Ainsi lorsque l'on choisit l'architecture du réseaux de neurones (*i.e.* l'espace de fonctions paramétriques), le *loss landscape* associé, la surface formée par la fonction de perte dans cet espace des paramètres Ω, est important. En effet, l'objectif est de **placer** cette solution sur cette surface de telle sorte qu'elle soit facile à atteindre de façon rapide et fiable en utilisant l'un des procédés décrits précédemment. On comprend ainsi l'importance du *loss landscape*.

Taille du modèle et sur-ajustement

➢ Ce fonctionnement à front renversé explique un certain nombre de comportements des réseaux de neurones qui semblent contredire en apparence certains principes du *Machine Learning*. Par exemple, on observe bien souvent en pratique qu'augmenter le nombre de paramètres du réseau, c'est à dire sa taille, améliore la performance de généralisation en réduisant non pas le biais mais au contraire la variance ([15]). C'est vrai notamment lorsque l'on augmente la largeur des couches cachées ([10, 188, 107]). De même, les techniques développées pour faire fonctionner des réseaux plus profonds cherchent d'abord à remédier aux problèmes de propagation du gradient ([114, 90]) et non celui de sur-apprentissage.

Le rôle des données

➢ Notons enfin que le *loss landscape* dépend autant de la forme du réseau que de l'échantillon d'entraînement. Dès lors, des techniques de natures très diverses ont été proposées pour enrichir le jeu de données d'entraînement. Par exemple, la *data-augmentation* consiste à générer de nouvelles données en appliquant des transformations (*e.g.* : rotation ou translation d'image) aux observations existantes ([216, 206, 213]). Dans le cadre de la classification, on peut également fabriquer des observations en appliquant une combinaison linéaire de deux observations et de leurs labels ([223]). Cette technique, appelée *Mix-Up*, permet d'obtenir des

règles de décisions qui évoluent progressivement entre deux classes plutôt que par rupture, ce qui rend l'estimateur plus résilient aux erreurs de labélisation. En présence d'un jeu de données de petite taille, on peut dans certains contextes (*e.g.* : reconnaissance d'images) utiliser le *transfer learning* ([147]), *i.e* on procède à une première descente de gradient en utilisant un autre jeu de données beaucoup plus volumineux tel *ImageNet* ([137, 88]), puis on fait une seconde descente de gradient avec le jeu de données d'origine, en ne mettant à jour que les paramètres des deux dernières couches.

2.4.2 Contrôle des normes des poids et des activations

L'un des pré-requis essentiels pour que la descente de gradient fonctionne correctement en *Deep Learning* est le contrôle des normes au sein du réseau. Plus précisément, des normes des matrices $\{\mathbf{W}^l\}_{l=1,\cdots,L}$ ainsi que des quantités $\{\mathbf{A}^l\}_{l=1,\cdots,L}$ et $\{\mathbf{Z}^l\}_{l=1,\cdots,L}$. La raison principale est que les réseaux de neurones sont des fonctions récursives. Si à chaque étape l de la récursion, la norme de l'activation en sortie $\mathbf{A}^l$ est deux fois plus élevée que la norme de l'activation en entrée $\mathbf{A}^{l-1}$, le rapport entre la norme de $\mathbf{A}^L = \widehat{\mathbf{Y}}$ et $\mathbf{A}^0 = \mathbf{X}$ sera de 2^L. Cela peut être la source non seulement d'instabilité numérique lorsque l'on utilise des fonctions d'activations non bornées (*e.g.* : ReLU ([141])), mais également d'un problème appelé *gradient vanishing*. Par exemple, pour la fonction d'activation Sigmoïde, la dérivée tend vers 0 au bord de son domaine de définition et pour des valeurs extrêmes le gradient devient nul, ce qui rend la descente de gradient inopérante. Réciproquement, si à chaque étape la norme est divisée par deux, le rapport sera de 2^{-L}, ce qui peut correspondre à des prédictions dont la norme est si faible que le gradient de la fonction de perte est nul également.

Initialisation

➢ Une composante essentielle pour contrôler ces normes est la technique utilisée pour initialiser les paramètres du réseau. Les écueils les plus connus identifiés par la communauté ([135, 175, 50, 71]) sont les suivants :

- Si un poids est initialisé à 0, la dérivée partielle associée sera nulle, ce qui ôte tout intérêt à la Rétro-Propagation.
- Si tous les poids d'une même couche sont identiques, le gradient pour chaque neurone sera identique, et ainsi les neurones resteront identiques tout au long de l'apprentissage.
- Si la norme des poids sur une couche est très élevée ou très faible, il est très probable que la norme de l'activation en sortie soit respectivement beaucoup

plus élevée ou beaucoup plus faible que la norme de l'activation en entrée. On rencontre alors les problèmes d'instabilité numérique et de *gradient vanishing* décrits précédemment.

Des techniques d'initialisation tenant compte de ces difficultés ont été proposées, telles l'Initialisation 1 Glorot présentée plus haut ([71]).

Batch Norm

➢ La plupart des fonctions d'activation sont efficaces quand les entrées ont des moyennes empiriques proches de 0 et des écart-types empiriques proches de 1. Une façon simple de garantir que cette contrainte soit satisfaite est de centrer et réduire les activations en sortie de chaque couche, cette technique est appelée *Batch-norm* ([101, 100, 8]). Cette technique introduit de nouveaux paramètres $\beta = \{\boldsymbol{\beta}_l\}_{l=1,\cdots,L}$ et $\gamma = \{\boldsymbol{\gamma}_l\}_{l=1,\cdots,L}$, permettant un contrôle des moyennes et variances empiriques sur chaque couche. Ces paramètres sont appris durant la descente de gradient.

Introduisons pour toute époque $e = 0, \cdots, E$, $\beta(e) = \{\boldsymbol{\beta}_l(e)\}_{l=1,\cdots,L}$ et $\gamma(e) = \{\boldsymbol{\gamma}_l(e)\}_{l=1,\cdots,L}$, ces paramètres intermédiaires sont mis à jour à chaque époque : $\forall l = 1, \cdots, L$,

$$\begin{cases} \boldsymbol{\beta}_l(0) & := & \mathbf{0}_{J_l} \\ \boldsymbol{\gamma}_l(0) & := & \mathbf{1}_{J_l} \\ \boldsymbol{\beta}_l(e) & \in & \mathbb{R}^{J_l} \quad \forall e = 1, \cdots, E \\ \boldsymbol{\gamma}_l(e) & \in & \mathbb{R}_+^{J_l} \quad \forall e = 1, \cdots, E \end{cases}$$

Propagation-Avant_ 5. [***Batch-Norm***]

Lors de l'apprentissage, les paramètres $\boldsymbol{\beta}_l(e)$ et $\boldsymbol{\gamma}_l(e)$ interviennent durant la Propagation-Avant comme suit : $\forall e = 1, \cdots, E$, et $\forall l = 1, \cdots, L$,

$$\begin{cases} \mathbf{Z}^l(e) & = & \mathbf{A}^{l-1}(e)W^l(e-1) + \mathbf{b}^l(e-1)\mathbf{1}_n^T \\ \widetilde{\mathbf{A}}^l(e) & = & S_l(\mathbf{Z}^l(e)) \\ \boldsymbol{m}_l(e) & = & \frac{1}{n}\widetilde{\mathbf{A}}^l(e)^T\mathbf{1}_n \in \mathbb{R}^{J_l} \\ \boldsymbol{s}_l^2(e) & = & \frac{1}{n}\mathbf{1}_n^T\Big(\big(\widetilde{\mathbf{A}}^l(e) - \mathbf{1}_n\boldsymbol{m}_l(e)^T\big) \odot \big(\widetilde{\mathbf{A}}^l(e) - \mathbf{1}_n\boldsymbol{m}_l(e)^T\big)\Big) \in \mathbb{R}_+^{J_l} \\ \mathbf{A}^l_{\mathbf{BN}}(e) & = & \big(\widetilde{\mathbf{A}}^l(e) - \mathbf{1}_n\boldsymbol{m}_l(e)^T\big) \oslash \big(\mathbf{1}_n\boldsymbol{s}_l^2(e)^T + \epsilon\mathbb{1}_{n\times J_l}\big) \\ \mathbf{A}^l(e) & = & \big(\mathbf{1}_n\boldsymbol{\gamma}_l(e-1)^T\big) \odot \mathbf{A}^l_{\mathbf{BN}}(e) + \mathbf{1}_n\boldsymbol{\beta}_l(e-1)^T \end{cases}$$

Ici, $\epsilon \in \mathbb{R}_{+*}$ est une constante de stabilité numérique. Les termes $\boldsymbol{m}_l(e)$ et $\boldsymbol{s}_l^2(e)$ correspondent respectivement à l'espérance et à la variance empirique de chaque colonne de la matrice $\widetilde{\mathbf{A}}^l(e)$, de tel sorte que les colonnes de la matrice $\mathbf{A}_{\text{BN}}^l(e)$ sont centrées et réduites.

Durant la Rétro-Propagation et la Mise-à-jour, on calcule $\beta_l(e)$ en fonction de $\beta_l(e-1)$ et $\gamma_l(e)$ en fonction de $\gamma_l(e-1)$ comme on le fait pour $\mathbf{b}_l(e)$ en fonction de $\mathbf{b}_l(e-1)$ A l'issu de l'apprentissage, pour prédire une nouvelle observation, on utilise la formule de **Propagation-Avant_5** avec les paramètres $\widehat{\beta} = \{\widehat{\boldsymbol{\beta}_l}\}_{l=1,\cdots,L}$ et $\widehat{\gamma} = \{\widehat{\boldsymbol{\gamma}_l}\}_{l=1,\cdots,L}$ définis comme suit : $\forall l = 1,\cdots,n$,

$$\begin{cases} \widehat{\boldsymbol{\beta}_l} &= \boldsymbol{\beta}_l(E) - \boldsymbol{m}_l(E), \\ \widehat{\boldsymbol{\gamma}_l} &= \frac{\gamma_l(E)}{\boldsymbol{s}_l^2(E)+\epsilon \mathbf{1}_{J_l}}. \end{cases}$$

Soulignons que le terme **Batch-Norm** est impropre. En effet, d'une part on effectue une opération de centrage et de réduction, et d'autre part, on fait référence aux *mini-batchs*, cette technique étant combinée le plus souvent à un apprentissage **Mini-batchGD** . Les quantités $\boldsymbol{m}_l(e,e')$ et $\boldsymbol{s}_l(e,e')$ correspondent alors respectivement à la moyenne et à l'écart type mobile associés à l'étape $e' = 1,\cdots,n$ de l'époque $e = 1,\cdots,E$.
Cette méthode très efficace pose néanmoins certaines difficultés, la plus importante étant qu'elle fonctionne mal lorsque les *mini-batchs* ne sont pas indépendants et identiquement distribués (cas des données séquentielles pour les réseaux récurrents, de l'apprentissage par renforcement, des réseaux adversariels, etc..) ([96, 215]). Cela pose également des problèmes numériques qui rendent trop imprécis le calcul en demi-flottant (*fp16* au lieu de *fp32*). Enfin, les calculs de l'écart type mobile $\boldsymbol{s}_l(e)$ et de $\frac{\widetilde{\mathbf{A}}^l(e)-\mathbf{1}_n\boldsymbol{m}_l(e)}{\mathbf{1}_n\boldsymbol{s}_l^2(e)+\epsilon\mathbb{1}_{n\times J_l}}$ sont des opérations coûteuses (produits terme à terme) qui augmentent significativement le temps nécessaire et les ressources utilisées pour passer d'une couche à l'autre ([174]).

Self Normalizing Network
➢ Une autre approche pour contrôler les normes des activations, proposée par [114], est d'utiliser un `SNN`(pour "*Self-Normalizing Network*"). Cette architecture correspond au Perceptron Multicouche 4, en prenant sur chaque couche cachée la fonction d'activation `SeLU`(pour "*scaled exponential linear units*"), définie pour les constantes $\lambda \in \mathbb{R}_{+*}$ et $\alpha \in \mathbb{R}_{+*}$ de la façon suivante :

$$\begin{array}{rccl} & \mathbb{R} & \longrightarrow & (-\alpha\lambda, \infty) \\ \texttt{SeLU}_{\lambda,\alpha} : & x & \longmapsto & \begin{cases} \lambda x & \text{si } x > 0, \\ \lambda\left(\alpha e^x - \alpha\right) & \text{sinon.} \end{cases} \end{array}$$

La forme de cette fonction d'activation et les valeurs λ et α ont été choisies de tel sorte que pour chaque couche $l = 1, \cdots, L$, si l'espérance et la variance empirique de $\mathbf{A}^{l-1}$ sont respectivement proches de 0 et 1, alors l'espérance et la variance empirique de $\mathbf{A}^{l}$ seront également respectivement proches de 0 et 1. En pratique, les valeurs $\lambda = 1.6733$ et $\alpha = 1.050$ sont utilisées et ont été calculées en utilisant une formule basée sur le théorème du point fixe ; on renvoie à l'article initial [114] pour plus de détails sur les fondements théoriques de cette méthode. Notons que cette fonction d'activation permet en pratique d'obtenir de très bonnes performances de généralisation, comme on le verra dans les Chapitres 4 et 5.

2.4.3 Régularisation explicite

Drop-Out et Drop-Connect

➢ Une technique de régularisation très populaire consiste durant l'entraînement à ajouter du bruit blanc au sein du réseau. Les méthodes de *data-augmentation* et de *dithering* permettent d'améliorer les performances de généralisations en bruitant les observations. De façon analogue, les méthodes de **Drop-Out** ([193]) et **Drop-Connect** ([209]) bruitent respectivement les activations $\mathbf{A}^l(e)$ et les poids $\mathbf{W}^l(e-1)$ des couches cachées $l = 1, \cdots, L-1$ durant l'apprentissage. Le principe est de remplacer aléatoirement par 0 certains neurones ou certains poids à chaque itération, ce qui évite à la fonction apprise par le réseau de neurones d'être trop dépendante d'un neurone en particulier.

Propagation-Avant_ 6. [*Drop-Out*]

Avant de démarrer l'apprentissage, on associe à chaque couche du réseau une probabilité de *Dropout* (*i.e.* de mettre à 0 un neurone) $\{p_l\}_{l=1,\cdots,L}$ avec $\forall l = 1, \cdots, L-1$, $p_l \in [0,1)$ et $p_L = 0$. Durant l'apprentissage, à chaque époque $e = 1, \cdots, E$ et pour toutes les couches cachées $l = 1, \cdots, L-1$, on tire de façon aléatoire $\mathfrak{W}(e)$ l'ensemble des matrices

$$\mathfrak{W}(e) = \{\mathfrak{w}_l(e)\}_{l=1,\cdots,L-1} \quad \text{où} \quad \mathfrak{w}^l(e) \in \{0,1\}^{n \times J_{l-1}}$$

est une matrice dont les éléments sont indépendants et identiquement distribués selon une loi de probabilité de Bernouilli de paramètre $(1-p_l)$: $\mathfrak{w}^l_{1,1}(e) \sim \mathcal{B}(1-p_l)$. On effectue ensuite la Propagation-Avant de la façon suivante : $\forall e = 1, \cdots, E$ et $\forall l = 1, \cdots, L-1$

$$\begin{cases} \mathbf{Z}^l(e) &= \left(\mathfrak{w}^l(e) \odot \mathbf{A}^{l-1}(e)\right) W^l(e-1) + \mathbf{b}^l(e-1)\mathbf{1}_n^T, \\ \mathbf{A}^l(e) &= S_l(\mathbf{Z}^l(e)). \end{cases}$$

Pour la couche de sortie, on procède comme pour la **Propagation-Avant_** 1 usuelle. A l'issue de l'apprentissage, on fixe $\widehat{\omega}$ de la façon suivante :

$$\widehat{\omega} = \left\{(1 - p_l)\mathbf{W}^l(E), \mathbf{b}^l(E)\right\}_{l=1,\cdots,L}$$

Propagation-Avant_ 7. [***Drop-Connect***]

Comme pour le **Drop-Out**, avant l'apprentissage, on associe à chaque couche du réseau une probabilité de *Dropout* $\{p_l\}_{l=1,\cdots,L}$ avec $\forall l = 1, \cdots, L-1,\ p_l \in [0, 1)$ et $p_L = 0$. Durant l'apprentissage, à chaque époque $e = 1, \cdots, E$ et pour toutes les couches cachées $l = 1, \cdots, L-1$, on tire de façon aléatoire $\mathfrak{V}(e)$ l'ensemble des matrices

$$\mathfrak{V}(e) = \{\mathfrak{v}_l(e)\}_{l=1,\cdots,L-1} \quad \text{où} \quad \mathfrak{v}^l(e) \in \{0, 1\}^{J_{l-1} \times J_l}$$

est une matrice dont les éléments sont indépendants et identiquement distribués selon une loi de probabilité de Bernouilli de paramètre $(1-p_l)$: $\mathfrak{v}^l_{1,1}(e) \sim \mathcal{B}(1 - p_l)$. On effectue ensuite la Propagation-Avant de la façon suivante : $\forall e = 1, \cdots, E$ et $\forall l = 1, \cdots, L-1$

$$\begin{cases} \mathbf{Z}^l(e) &= \mathbf{A}^{l-1}(e)\left(\mathfrak{v}^l(e) \odot W^l(e-1)\right) + \mathbf{b}^l(e-1)\mathbf{1}_n^T, \\ \mathbf{A}^l(e) &= S_l(\mathbf{Z}^l(e)). \end{cases}$$

Pour la couche de sortie, on procède comme pour la **Propagation-Avant_** 1 usuelle. Dans la version de **Drop-Connect** implémentée dans *pytorch*, à l'issue de l'apprentissage, on procède comme pour le **Drop-Out**. Notons que la version initialement proposée par [209] est un petit peu différente, mais ce n'est pas celle utilisée en pratique, on ne la présentera donc pas ici.

Weight Decay

➢ Pour régulariser les poids du réseau, il semble naturel de vouloir s'inspirer des techniques de régression linéaire pénalisée, tel que la pénalité Ridge. En *Deep Learning*, on appelle **Weight-Decay** ([119]) les méthodes de régularisation dont l'objectif, *à l'origine*, était d'ajouter à la fonction de perte un terme de pénalité dépendant de la norme 2 des poids de la matrice, ce qui correspond au critère suivant :

$$\mathbf{C_{WD}}(\mathbf{X}, \mathbf{Y}, \omega) = \sum_{i=1}^{n} \mathbf{L}(y_i, f_\omega(\mathbf{x}_i)) + \lambda \sum_{l=1}^{L} ||\mathbf{W}^l||_2^2$$

Ici, $\lambda \in \mathbb{R}_{+*}$ est un hyperparamètre qu'il convient de calibrer. En pratique, au lieu de modifier l'expression des gradients $\mathbb{G}(e)$ durant l'étape de Rétro-Propagation en remplaçant $\mathbf{C}$ par $\mathbf{C_{WD}}$, on conserve l'expression de $\mathbb{G}(e)$ mais on applique la règle de Mise-à-jour suivante.

Mise-à-jour_ 6. [***Weight-Decay***]

On introduit le paramètre de **Weight-Decay** $\rho \in [0, 1)$ et on modifie les paramètres du réseau en appliquant une correction qui dépend de $\mathbb{G}(e)$, tel que pour tout $l = 1, \cdots, L$

$$\begin{cases} \mathbf{W}^l(e) &= (1-\rho)\mathbf{W}^l(e-1) - \eta \frac{\mathbf{G_W}^l(e)}{\|\mathbf{G_W}^l(e)\|_2 + \delta} \\ \mathbf{b}^l(e) &= \mathbf{b}^l(e-1) - \eta \frac{\mathbf{g_b}^l(e)}{\|\mathbf{g_b}^l(e)\|_2 + \delta} \end{cases}$$

Ici, la quantité $\rho \times \mathbf{W}^l(e-1)$ correspond au gradient de $\lambda\eta \sum_{l=1}^{L} \|\mathbf{W}^l(e-1)\|_2^2$ selon $\mathbf{W}^l(e-1)$, en prenant $\rho = 2\lambda\eta$. A l'origine, les raisons invoquées pour intégrer le **Weight-Decay** à l'étape de Mise-à-jour plutôt qu'à celle de Rétro-Propagation était d'ordre numérique. Cependant, il a été montré par la suite que ces deux façons d'implémenter le **Weight-Decay** ne sont **absolument pas équivalentes** lorsque celui-ci est combiné à la **Batch-Norm** ou à **Adam** . Il nous semble important de développer ce point dans la prochaine section, car il illustre bien le fait que des méthodes de régularisation qui sont strictement équivalentes en régression linéaire peuvent avoir un impact totalement différent en *Deep Learning*. C'est le cas en particulier pour celles basées sur l'estimateur Ridge, telles que la pénalité L_2, le **Weight-Decay** et l'opérateur Tikhonov présenté dans le Chapitre 4.

Interactions entre le Weight Decay, la Batch Norm et Adam

➢ Le **Weight-Decay** est souvent utilisé conjointement avec la **Batch-Norm** , bien que cette dernière neutralise l'effet initial recherché ([222]). En effet, dans la formule de **Propagation-Avant_** 5 de la **Batch-Norm** , la norme de

$$\mathbf{A}^{l-1}(e) = \left(\mathbf{1}_n \boldsymbol{\gamma}_{l-1}(e-1)^T\right) \odot \mathbf{A}_{\mathbf{BN}}^{l-1}(e) + \mathbf{1}_n \boldsymbol{\beta}_{l-1}(e-1)^T$$

dépend entièrement de $\boldsymbol{\beta}^{l-1}(e-1)$ et $\boldsymbol{\gamma}^{l-1}(e-1)$, $\forall l = 2, \cdots, L$. Dans **Mise-à-jour_** 6 on multiplie $\mathbf{W}^l(e-1)$ par un facteur $(1-\rho)$

$$\mathbf{W}^l(e) = (1-\rho)\mathbf{W}^l(e-1) - \eta \frac{\mathbf{G_W}^l(e)}{\|\mathbf{G_W}^l(e)\|_2 + \delta},$$

ce qui revient dans l'expression

$$\mathbf{Z}^l(e) = \mathbf{A}^{l-1}(e)\mathbf{W}^l(e-1) + \mathbf{b}^l(e-1)\mathbf{1}_n^T$$

à multiplier $\mathbf{A}^{l-1}(e)$, donc $\boldsymbol{\beta}^{l-1}(e-1)$ et $\boldsymbol{\gamma}^{l-1}(e-1)$ par ce même facteur. Ainsi la formule dans la **Mise-à-jour_** 6 du **Weight-Decay** donne

$$\frac{1}{1-\rho}\mathbf{W}^l(e) = \mathbf{W}^l(e-1) - \frac{\eta}{1-\rho} \frac{\mathbf{G_W}^l(e)}{\|\mathbf{G_W}^l(e)\|_2 + \delta}$$

ce qui correspond à **Mise-à-jour_** 1 de la descente de gradient usuelle, à ceci près que l'on a remplacé le *learning rate* η par $\frac{\eta}{1-\rho}$. Le **Weight-Decay** modifie le *learning rate effectif*, comme le ferait un *learning rate scheduler*.

Notons que lorsque l'on combine **Adam** et **Weight-Decay** , ajouter le terme de pénalité L_2 $-\rho \times \mathbf{W}^l(e-1)$ dans l'expression de la **Mise-à-jour** des poids n'équivaut pas, là encore, à ajouter une pénalité L_2 à la fonction de perte ([130]). En effet, avec **Adam** , comme avec **AdaGrad** et **RMSProp** , les gradients sont divisés par un terme de friction, qui est différent pour chaque poids d'une même couche, tel que la taille de la mise à jour d'un poids dépend des modifications apportées à ce poids aux itérations précédentes. Lorsque l'on ajoute une pénalité L_2 à la fonction de perte, tous les poids sont modifiés à chaque itération, ce qui atténue l'impact du terme de friction. L'*optimizer* qui intègre le **Weight-Decay** à l'étape de **Mise-à-jour** plutôt que de modifier le critère minimisé est appelé **AdamW**([130]). Sa **Mise-à-jour** est décrite ci-dessous.

Mise-à-jour_ 7. [*AdamW*]

L'*optimizer* **AdamW** est en tout point similaire **Adam** , à ceci près qu'il introduit dans la règle de **Mise-à-jour** un terme de **Weight-Decay** . Notons $\rho \in [0, 1)$ le paramètre associé, les mises à jour sont effectuées de la façon suivante : $\forall e = 1, \cdots, E$ et $\forall l = 1, \cdots, L$,

$$\begin{cases} \mathbf{W}^l(e) = (1-\rho)\mathbf{W}^l(e-1) - \frac{\eta}{1-\beta_1}\mathbf{V}_w^l(e) \oslash \left(\frac{1}{1-\beta_2}\mathbf{R}_w^l(e)^{\sqrt{\odot}} + \delta \mathbb{1}_{J^{l-1}\times J^l}\right), \\ \mathbf{b}^l(e) = \mathbf{b}^l(e-1) - \frac{\eta}{1-\beta_1}\mathbf{v}_b^l(e) \oslash \left(\frac{1}{1-\beta_2}\mathbf{r}_b^l(e)^{\sqrt{\odot}} + \delta \mathbf{1}_{J^l}\right). \end{cases}$$

2.5 Conclusion

Dans ce chapitre, nous n'avons abordé qu'une infime partie du *Deep Learning*, domaine extrêmement riche. On comprend que les réseaux de neurones sont des modèles capables d'apprendre des règles logiques et des fonctions d'une infinie complexité. Si leur performances impressionnent, leur fonctionnement peut, au premier abord, paraître ésotérique : "la fonction de perte est fortement non convexe mais les paramètres sont appris par descente de gradient", "le modèle est sur-paramétré mais augmenter le nombre de paramètres a un effet de régularisation", "injecter du bruit améliore la stabilité", etc. Cependant, tout ceci prend sens, une fois que l'on comprend que pour un modèle de *Deep Learning*, c'est durant l'apprentissage et non à la "naissance" que tout se joue.

Dans le Chapitre 4, on s'intéresse plus en détail au phénomène de mémorisation (capacité à "*fitter*" des labels arbitraires) en *Deep Learning*et on développe

une nouvelle méthode d'entraînement, appelée `AdaCap`, pour s'en prémunir. Au Chapitre 5, nous nous intéressons à un problème plus complexe, l'estimation de densité conditionnelle, une nouvelle méthode appelée `MCD` qui combinée à des réseaux de neurones, donne d'excellents résultats.

Enfin, pour les lecteurs voulant poursuivre leur initiation au *Deep Learning*, on recommande les ressources en ligne suivantes : Coursera ([145]), Fastai ([98]), ainsi que l'ouvrage de référence de [73] et la plateforme de compétition [108].

Chapter 3
MLR: Muddling Labels for Regularization

To Generalize is to be an Idiot

William Blake, Annotations to Sir Joshua Reynolds's Discourses

Chapter Summary

Generalization is a central problem in Machine Learning. Indeed most prediction methods require careful calibration of hyperparameters usually carried out on a hold-out *validation* dataset to achieve generalization. The main goal of this chapter is to introduce a novel approach to achieve generalization without any data splitting, which is based on a new risk measure which directly quantifies a model's tendency to overfit. To fully understand the intuition and advantages of this new approach, we illustrate it in the simple linear regression model ($Y = X\beta + \xi$) where we develop a new criterion. We highlight how this criterion is a good proxy for the true generalization risk. Next, we derive different procedures which tackle several structures simultaneously (correlation, sparsity,...). Noticeably, these procedures **concomitantly** train the model and calibrate the hyperparameters. In addition, these procedures can be implemented via classical gradient descent methods when the criterion is differentiable w.r.t. the hyperparameters. Our numerical experiments reveal that our procedures are computationally feasible and compare favorably to the popular approach (Ridge, LASSO and Elastic-Net combined with grid-search cross-validation) in term of generalization. They also outperform the baseline on two additional tasks: estimation and support recovery of β. Moreover, our procedures do not require any expertise for the calibration of the initial parameters which remain the same for all the datasets we experimented on.

3.1 Introduction

Generalization is a central problem in machine learning. Regularized or constrained Empirical Risk Minimization ($\mathtt{ERM}$) is a popular approach to achieve generalization [121]. Ridge [95], LASSO [199] and Elastic-net [227] belong to this category. The regularization term or the constraint is added in order to achieve generalization and to enforce some specific structures on the constructed model (sparsity, low-rank, coefficient positiveness,...). This usually involves introducing hyperparameters which require calibration. The most common approach is data-splitting. Available data is partitioned into a *training/validation*-set. The *validation*-set is used to evaluate the generalization error of a model built using only the *training*-set.

Several hyperparameter tuning strategies were designed to perform hyperparameter calibration: Grid-search, Random search [18] or more advanced

hyperparameter optimization techniques [19, 17, 176]. For instance, BlackBox optimization [32] is used when the evaluation function is not available [122]. It includes in particular Bayesian hyperparametric optimization such as Thompson sampling [138, 191, 198]. These techniques either scale exponentially with the dimension of the hyperparameter space, or requires a smooth convex optimization space [179]. Highly non-convex optimization problems on a high dimensionnal space can be tackled by Population based methods (Genetic Algorithms [38, 168, 146], Particle Swarm [129, 128]) but at a high computational cost. Another family of advanced methods, called gradient-based techniques, take advantage of gradient optimization techniques [54] like our method. They fall into two categories, Gradient Iteration and Gradient approximation. Gradient Iteration directly computes the gradient w.r.t. hyperparameters on the training/evaluation graph. This means differentiating a potentially lengthy optimization process which is known to be a major bottleneck [155]. Gradient approximation is used to circumvent this difficulty, through implicit differentiation [124, 21]. However, all these advanced methods require data-splitting to evaluate the trained model on a hold-out *validation*-set, unlike our approach.

Another approach is based on unbiased estimation of the generalization error of a model (SURE [194], *AIC* [4], C_p-Mallows [131]) on the *training*-set. Meanwhile, other methods improve generalization during the training phase without using a hold-out *validation*-set. For instance, Stochastic Gradient Descent and the related batch learning techniques [27] achieve generalization by splitting the training data into a large number of subsets and compute the Empirical Risk (ER) on a different subset at each step of the gradient descent. This strategy converges to a good estimation of the generalization risk provided a large number of observations is available. Bear in mind this method and the availability of massive datasets played a crucial role in the success of Deep neural networks. Although batch size has a positive impact on generalization [86], it cannot maximize generalization on its own.

Model aggregation is another popular approach to achieve generalization. It concerns for instance Random Forest [94, 30], MARS [66] and Boosting [65]. This approach aggregates weak learners previously built using bootstrapped subsets of the *training*-set. The training time of these models is considerably lengthened when a large number of weak learners is considered, which is a requirement for improved generalization. Recall XGBOOST [40] combines a version of batch learning and model aggregation to train weak learners.

MARS, Random Forest, XGBOOST and Deep learning have obtained excellent results in Kaggle competitions and other machine learning benchmarks [62, 57].

However these methods still require regularization and/or constraints in order to generalize. This implies the introduction of numerous hyperparameters which require calibration on a hold-out *validation*-set for instance *via* Grid-search. Tuning these hyperparameters requires expensive human expertise and/or computational resources.

We approach generalization from a different point of view. The underlying intuition is the following. We no longer see generalization as the ability of a model to perform well on unseen data, but rather as the ability to avoid finding pattern where none exist. Using this approach, we derive a novel criterion and several procedures which do not require data splitting to achieve generalization.

This chapter is intended to be an introduction to this novel approach. Therefore, for the sake of clarity, we consider here the linear regression setting but our approach can be extended to more general settings like deep learning, which is the subject of Chapter 4.

Let us consider the linear regression model:

$$\mathbf{Y} = \mathbb{X}\beta^* + \xi, \tag{3.1}$$

where $\mathbb{X}^\top = (\mathbf{X}_1, \cdots, \mathbf{X}_n)$ is the $n \times p$ *design matrix* and the n-dimensional vectors $\mathbf{Y} = (Y_i, \cdots, Y_n)^\top$ and $\xi = (\xi_1, \cdots, \xi_n)^\top$ are respectively the response and the noise variables. Throughout this chapter, the noise level $\sigma > 0$ is unknown. Set $\|\mathbf{v}\|_n = (\frac{1}{n}\sum_{i=1}^n v_i^2)^{1/2}$ for any $\mathbf{v} = (v_1, \ldots, v_n)^\top \in \mathbb{R}^n$.

In practice, the correlation between $\mathbf{X}_i$ and Y_i is unknown and may actually be very weak. In this case, $\mathbf{X}_i$ provides very little information about Y_i and we expect from a good procedure to avoid building a spurious connection between $\mathbf{X}_i$ and Y_i. Therefore, by understanding generalization as "*do not fit the data in non-informative cases*", we suggest creating an artificial dataset which preserves the marginal distributions while the link between $\mathbf{X}_i$ and Y_i has been completely removed. A simple way to do so is to construct an artificial set $\widetilde{\mathcal{D}} = (\mathbb{X}, \widetilde{\mathbf{Y}}) = (\mathbb{X}, \pi(\mathbf{Y}))$ by applying permutations $\pi \in \mathfrak{S}_n$ (the set of permutations of n points) on the components of $\mathbf{Y}$ of the initial dataset $\mathcal{D}$ where for any $\mathbf{y} \in \mathbb{R}^n$, we set $\pi(\mathbf{y}) = (y_{\pi(1)}, \ldots, y_{\pi(n)})^\top$.

The rest of the chapter is organized as follows. In Section 3.2 we introduce our novel criterion and highlight its generalization performance. In Section 3.3, this new approach is applied to several specific data structures in order to design **adapted** procedures which are compatible with gradient-based optimization meth-

ods. We also point out several advantageous points about this new framework in an extensive numerical study.

3.2 The Muddling Label Regularization Criterion

In model (3.1), we want to recover β^* from $\mathcal{D} = (\mathbb{X}, \mathbf{Y})$. Most often, the n observations are partitioned into two parts of respective sizes n_{train} and n_{val}, which we denote the *train*-set $(\mathbb{X}_{train}, \mathbf{Y}_{train})$ and the *validation*-set $(\mathbb{X}_{val}, \mathbf{Y}_{val})$ respectively. The *train*-set is used to build a family of estimators $\{\beta(\theta, \mathbb{X}_{train}, \mathbf{Y}_{train})\}_\theta$ which depends on a hyperparameter θ. Next, in order to achieve generalization, we use the *validation*-set to calibrate θ. This is carried out by minimizing the following empirical criterion w.r.t. θ:

$$\|\mathbf{Y}_{val} - \mathbb{X}_{val}\beta(\theta, \mathbb{X}_{train}, \mathbf{Y}_{train})\|_{n_{val}}.$$

In our approach, we use the complete dataset $\mathcal{D}$ to build the family of estimators and to calibrate the hyperparameter θ.

Definition 1. *Fix $T \in \mathbb{N}^*$. Let $\{\pi^t\}_{t=1}^T$ be T permutations in $\mathfrak{S}_n$. Let $\{\beta(\theta, \cdot, \cdot)\}_\theta$ be a family of estimators.*

*The **Muddling Labels Regularization criterion** is defined as:*

$$\mathtt{MLR}_\beta(\theta) = \|\mathbf{Y} - \mathbb{X}\beta(\theta, \mathbb{X}, \mathbf{Y})\|_n - \frac{1}{T}\sum_{t=1}^{T} \|\pi^t(\mathbf{Y}) - \mathbb{X}\beta(\theta, \mathbb{X}, \pi^t(\mathbf{Y}))\|_n. \tag{3.2}$$

The `MLR` criterion performs a trade-off between two antagonistic terms. The first term fits the data while the second term prevents overfitting. Since the `MLR` criterion is evaluated directly on the whole sample without any hold-out *validation*-set, this approach is particularly useful for small sample sizes where data-splitting approaches can produce strongly biased performance estimates [203, 205].

The `MLR` approach uses random labels in an original way. In [221, 7], noise labels are used as a diagnostic tool in numerical experiments. On the theory side, Rademacher Process (RP) is a central tool exploiting random (Rademacher) labels to compute data dependent measures of complexity of function classes used in learning [117]. However, RP are used to derive bounds on the excess risk of already trained models whereas the `MLR` approach can be used directly to select the appropriate hyperparameter value.

THEORETICAL INVESTIGATION.

To understand the core mechanism behind the MLR criterion, we consider the following toy regression model. Let $\mathbf{Y} = \mathbf{x}\boldsymbol{\beta}^* + \boldsymbol{\xi}$ with $\beta^* \in \mathbb{R}^p$ and isotropic sub-Gaussian noise $\boldsymbol{\xi} \in \mathbb{R}^n$ ($\mathrm{Cov}(\boldsymbol{\xi}) = \sigma^2 \mathbb{I}_n$). Here we take $T = 1$ and denote $\mathbf{Y}_{\texttt{perm}} = \pi^1(\mathbf{Y})$ the permuted target vector. We consider the class of Ridge models $\mathcal{F}^R = \{f_\lambda(\cdot) = \langle \beta_\lambda, \cdot \rangle,\ \lambda > 0\}$ with $\beta_\lambda = \beta_\lambda(\mathbf{x}, \mathbf{Y}) = (\mathbf{x}^\top \mathbf{x} + \lambda \mathbb{I}_\lambda)^{-1} \mathbf{x}^\top \mathbf{Y} \in \mathbb{R}^p$. Define the risk $R(\lambda) := \mathbb{E}_{\xi}[\|\mathbf{x}\beta^* - \mathbf{x}\beta_\lambda\|_2^2]$, and the optimal parameter $\lambda^* = \mathrm{argmin}_{\lambda > 0} R(\lambda)$. We assume for simplicity that $\mathbf{x}^\top \mathbf{x}/n$ is an orthogonal projection (denoted $P_{\mathbf{x}}$) onto a r-dimensional subspace of $\mathbb{R}^p$. Define the rate

$$\epsilon_n := \sqrt{\frac{r\sigma^2}{\|\mathbf{x}\beta^*\|_2^2}} + \sqrt{\frac{\|\mathbf{x}\beta^*\|_2^2}{n\sigma^2}}.$$

Theorem 1. *Under the above assumptions. If $r\sigma^2 \ll \|\mathbf{x}\beta^*\|_2^2 \ll n\sigma^2$, then we get w.h.p.*

$$\mathtt{MLR}_\beta(\lambda) + \|P_x(\mathbf{Y}_{perm})\|_2^2 = (1 + o(1))\, R(\lambda), \quad \forall \lambda > \epsilon_n.$$

Proof is provided in Appendix 3.5.1. In our setting, $\|\mathbf{x}\beta^*\|_2^2/(n\sigma^2)$ is the Signal-to-Noise Ratio SNR. The intermediate SNR regime $r\sigma^2 \ll \|\mathbf{x}\beta^*\|_2^2 \ll n\sigma^2$ is the only regime where using Ridge regularization can yield a significant improvement in the prediction. In that regime, the MLR criterion can be used to find optimal hyperparameter λ^*. In the high SNR regime $\|\mathbf{x}\beta^*\|_2^2 \geq n\sigma^2$, no regularization is needed, i.e. $\lambda^* = 0$ is the optimal choice. Conversely in the low SNR regime $\|\mathbf{x}\beta^*\|_2^2 \leq r\sigma^2$, the signal is completely drowned in the noise. Consequently it is better to use the zero estimator, i.e. $\lambda^* = \infty$.

In a nutshell, while the high and low SNR regimes correspond to trivial cases where regularization is not useful, in the intermediate regime where regularization is beneficial, MLR is useful.

We highlight in the following numerical experiment the remarkable generalization performance of the MLR criterion.

NUMERICAL EXPERIMENTS.

We consider two families

$$\mathcal{F}(\theta) = \{\beta(\theta, \mathbb{X}_{train}, \mathbf{Y}_{train})\}_\theta$$

of estimators constructed on a *train*-dataset and indexed by θ: Ridge and LASSO . We are interested in the problem of calibration of the hyperparameter θ on a grid Θ. We compare two criteria $C(\theta)$ to calibrate θ: the `MLR` criterion and cross-validation (implemented as Ridge, Lasso in Scikit-learn [154]). For each criterion $C(\theta)$, the final estimator $\widehat{\beta}_{train} = \beta(\widehat{\theta}, \mathbb{X}_{train}, \mathbf{Y}_{train})$ is *s.t.*

$$\widehat{\theta} = \arg\min_{\theta \in \Theta} C(\theta).$$

$\mathbf{R^2}$-score. For each family, the generalization performance of each criterion is evaluated using the hold-out *test*-dataset $\mathcal{D}_{test} = (\mathbb{X}_{test}, \mathbf{Y}_{test})$ by computing the following $\mathbf{R^2}$-scores:

$$\mathbf{R^2}(\widehat{\beta}_{train}) = 1 - \frac{\|\mathbf{Y}_{test} - \mathbb{X}_{test}\widehat{\beta}_{train}\|_2}{\|\mathbf{Y}_{test} - \overline{\mathbf{Y}}_{test}\mathbb{1}_n\|_2} \ (\leq 1), \tag{3.3}$$

where $\overline{\mathbf{Y}}_{test}$ is the empirical mean of $\mathbf{Y}_{test}$. For the sake of simplicity, we set $\mathbf{R^2}(\widehat{\theta}) = \mathbf{R^2}(\widehat{\beta}_{train})$. The oracle we aim to match, is

$$\theta^* = \arg\min_{\theta \in \Theta} \mathbf{R^2}(\theta).$$

Our first numerical experiments concern synthetic data.

Synthetic data. For $p = 80$, we generate observations $(\mathbf{X}, Y) \in \mathbb{R}^p \times \mathbb{R}$, *s.t.* $Y = \mathbf{X}^\top \beta^* + \epsilon$, with $\epsilon \sim \mathcal{N}(0, \sigma)$, $\sigma = 10$ or 50. We consider three different scenarii. **Scenario A** (correlated features) corresponds to the case where the LASSO is prone to fail and Ridge should perform better. **Scenario B** (sparse setting) corresponds to a case known as favorable to LASSO . **Scenario C** combines sparsity and correlated features. For each scenario we sample a *train*-dataset of size $n_{train} = 100$ and a *test*-dataset of size $n_{test} = 1000$.

For each scenario, we perform $M = 100$ repetitions of the data generation process to produce M pairs of *train*/*test* datasets. Details on the data generation process can be found in the Appendix.

Performances evaluation. For each family $\mathcal{F}(\theta)$ and each criterion $C(\theta)$, we construct on every *train*-dataset, the corresponding model $\widehat{\beta}_{train}$. Next, using the corresponding hold-out *test*-dataset $\mathcal{D}_{test} = (\mathbb{X}_{test}, \mathbf{Y}_{test})$, we compute their $\mathbf{R^2}$-scores.

Impact of T. In Figure 3.2 we study the impact of the number of permutations T on the generalization performance of the criterion measured via the R^2-score in (3.3). The most striking finding is the sharp increase in the generalization performance from the first added permutation in **Scenarios A and C**. Adding more permutations does not impact the generalization but actually improves the running time and stability of the novel procedures which we will introduce in the next section. In a pure sparsity setting (**Scenario B** with LASSO), adding permutations marginally increases the generalization.

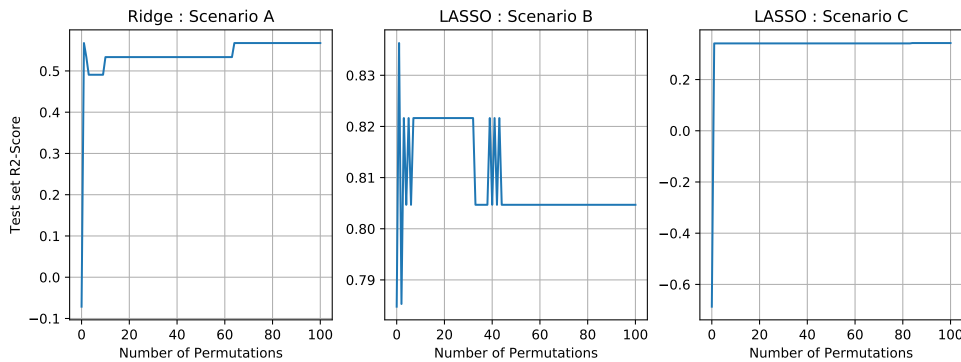


Figure 3.1: Variation of the R^2-score w.r.t. the number of permutation.

Comparison of generalization performance. For **Scenario A** with the Ridge family, we compute the difference $\mathbf{R^2}(\widehat{\theta^{C(\theta)}}) - \mathbf{R^2}(\theta^*)$, for the two criteria $C(\theta)$ (`MLR` and CV). For **Scenarios B and C**, we consider the LASSO family and compute the same difference. Boxplots in Figure 3.2 summarize our finding over 100 repetitions of the synthetic data. The empirical mean is depicted by a green triangle on each boxplot. Moreover, to check for statistically significant margin in $\mathbf{R^2}$-scores between different procedures, we use the Mann-Whitney test (as detailed in [120] and implemented in scipy [208]). The boxplots highlighted in yellow correspond to the best procedures according to the Mann-Whitney (MW) test. As we observed, the `MLR` criterion performs better than CV for the calibration of the Ridge and LASSO hyperparameters in **Scenarios A and C** in the presence of correlation in the design matrix.

Plot of the generalization performance. In order to plot the different criteria together, we use the following rescaling. Let $F : \Theta \to \mathbb{R}$, we set

$$\underline{\theta} = \arg\min_{\theta\in\Theta} F(\theta) \text{ and } \overline{\theta} = \arg\max_{\theta\in\Theta} F(\theta).$$

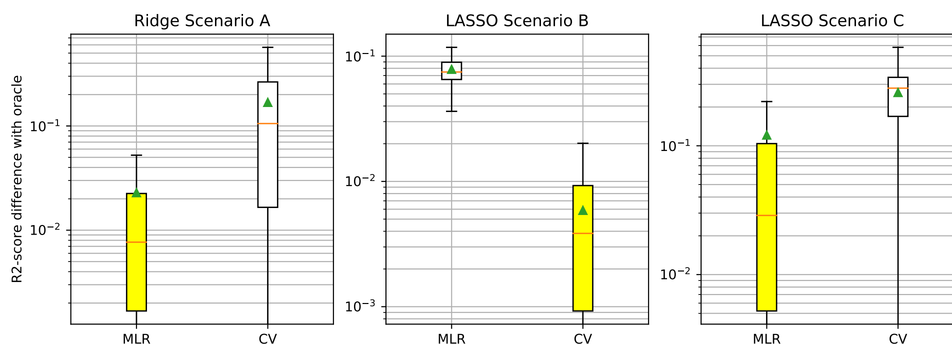


Figure 3.2: Generalization performance for MLRand CV for Ridge and LASSO Hyperparameter calibration

For any $\theta \in \Theta$, we define

$$\Lambda_F(\theta) = \frac{F(\theta) - F(\underline{\theta})}{F(\overline{\theta}) - F(\underline{\theta})}.$$

Figure 3.3 contains the rescaled versions of the $\mathbf{R}^2$-score on the test set and the MLR and *CV* criteria computed on the *train*-dataset. The vertical lines correspond to the selected values of the hyperparameter in the grid Θ by the MLR and CV criteria as well as the optimal hyperparameter θ^* for the $\mathbf{R}^2$-score on the test set. The MLR criterion is a good proxy for the generalization performance (R^2- score on the *test*-dataset) on the whole grid Θ for the Ridge in **Scenario A**. In **Scenario C** with the LASSO, the MLR criterion is a smooth function with steep variations in a neighborhood of its global optimum. This is an ideal configuration for the implementation of gradient descent schemes.

the MLR criterion performs better than CV on **correlated** data for grid-search calibration of the LASSO and Ridge hyperparameters. However CV works better in the pure sparsity scenario. This motivated the introduction of novel procedures based on the MLR principle which can handle the sparse setting better.

3.3 Novel procedures

In Section 3.2, we used (3.2) to perform grid-search calibration of the hyperparameter. However, if $\{\beta(\theta)\}_\theta$ is a family of models differentiable w.r.t. θ, we can minimize (3.2) w.r.t. θ via standard gradient-based methods. This motivated the introduction of new procedures based on the MLR criterion. Chapter 4 presents AdaCap a training scheme for Deep Neural Networks based on MLR. In this section, we remain in the classical linear regression setting and present R-MLR, S-MLR

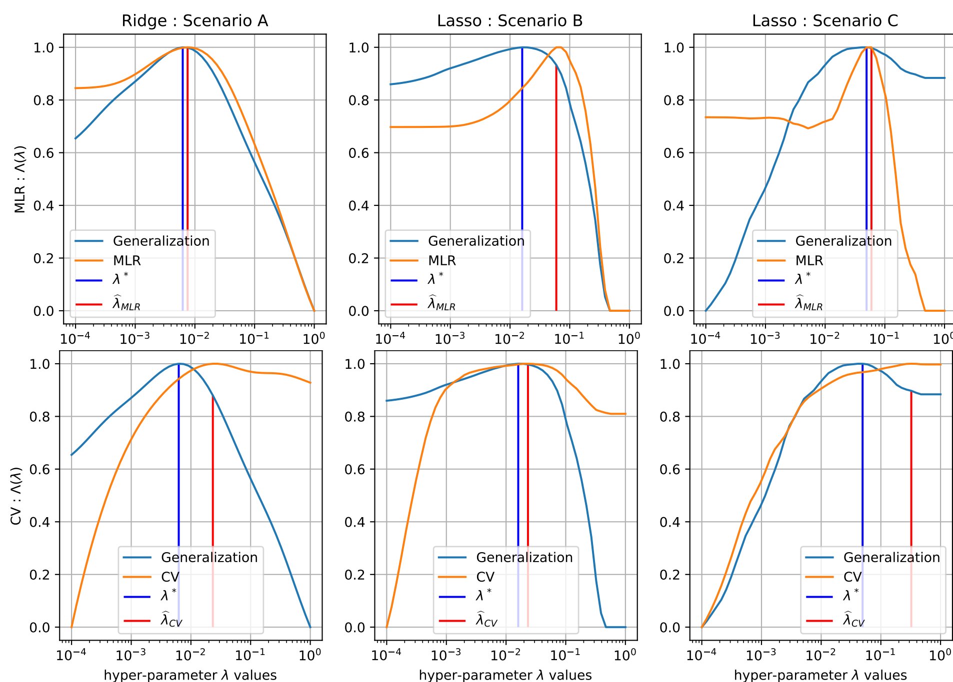


Figure 3.3: Criterion landscape for grid-search calibration with MLR (**Up**) and CV (**down**).

and A-MLR, 3 novel procedures to perform linear regression. AdaCap, R-MLR, S-MLR and A-MLR are all based on a common framework, which we call the **β-MLR procedure**:

Definition 2. *Consider $\{\beta(\theta)\}_\theta$ a family of models differentiable w.r.t. θ. Let $\{\pi^t\}_{t=1}^T$ be T derangements in $\mathfrak{S}_n$. The β-**MLR procedure** is*

$$\widehat{\beta} = \beta(\widehat{\theta}) \quad with \quad \widehat{\theta} = \arg\min_\theta MLR_\beta(\theta). \tag{3.4}$$

where MLR_β is defined in (3.2).

Using our approach, we can also enforce several **additional structures** simultaneously (sparsity, correlation, group sparsity, low-rank,...) by constructing appropriate families of models. In this regard, let us consider the 3 following procedures which do not require a hold-out *validation*-set.

R-MLR procedure for correlated designs. The Ridge family of estimators $\{\beta^R(\lambda, \mathbb{X}, \mathbf{Y})\}_{\lambda>0}$ is defined as follows:

$$\beta^R(\lambda, \mathbb{X}, \mathbf{Y}) = (\mathbb{X}^\top\mathbb{X} + \lambda\mathbb{I}_p)^{-1}\mathbb{X}^\top\mathbf{Y}, \quad \lambda > 0. \tag{3.5}$$

Applying Definition 2 with the Ridge family, we obtain the `R-MLR` procedure $\widehat{\beta}^R = \beta^R(\widehat{\lambda})$ where $\widehat{\lambda} = \arg\min_{\lambda>0} \mathtt{MLR}_{\beta^R}(\lambda)$. This new optimisation problem can be solved by gradient descent, contrarily to the previous section where we performed a grid-search calibration of λ.

`S-MLR` for sparse models. We design in Definition 3 below a novel differentiable family of models to enforce sparsity in the trained model. Applying Definition 2 to this family, we can derive the `S-MLR` procedure: $\widehat{\beta}^S = \beta^S(\widehat{\theta})$ where $\widehat{\theta} = \arg\min_{\theta} \mathtt{MLR}_{\beta^S}(\theta)$.

Definition 3. *Let* $\{\beta^S(\lambda, \kappa, \gamma, \mathbb{X}, \mathbf{Y})\}_{(\lambda,\kappa,\gamma)\in\mathbb{R}_+^*\times\mathbb{R}_+^*\times\mathbb{R}^p}$ *be a family closed-form estimators defined as follows:*

$$\beta^S(\lambda, \kappa, \gamma, \mathbb{X}, \mathbf{Y}) = \mathcal{S}(\kappa, \gamma)\beta^R(\lambda, \mathbb{X}\mathcal{S}(\kappa, \gamma), \mathbf{Y}), \tag{3.6}$$

where β^R *is defined in* (3.5)*, the* ***quasi-sparsifying*** *function* $\mathcal{S} : \mathbb{R}_+^* \times \mathbb{R}^p \to]0, 1[^{p\times p}$ *is s.t.*

$$\mathcal{S}(\kappa, \gamma) = diag\left(\mathcal{S}_1(\kappa, \gamma), \cdots, \mathcal{S}_p(\kappa, \gamma)\right),$$

where for any $j = 1, \cdots, p$,

$$\begin{aligned} \mathcal{S}_j \ : \ & \mathbb{R}_+^* \times \mathbb{R}^p \to]0, 1[\\ & (\kappa, \gamma) \mapsto \mathcal{S}_j(\kappa, \gamma) = \left(1 + e^{-\kappa\times(\sigma_\gamma^2+10^{-2})(\gamma_j-\overline{\gamma})}\right)^{-1}, \end{aligned}$$

with $\overline{\gamma} = \frac{1}{p}\sum_{i=1}^p \gamma_i$ *and* $\sigma_\gamma^2 = \sum_{i=1}^p(\gamma_i - \overline{\gamma})^2$.

The new family (3.6) enforces sparsity on the regression vector but also directly onto the design matrix. Hence it can be seen as a combination of data-preprocessing (performing feature selection) and model training (using the ridge estimator).

Noticeably, the "*quasi-sparsifying*" trick transforms feature selection (a discrete optimization problem) into a continuous optimization problem which is solvable via classical gradient-based methods. The function $\mathcal{S}$ produces diagonal matrices with diagonal coefficients in $]0, 1[$. Although the sigmoid function $\mathcal{S}_j$ cannot take values 0 or 1, for very small or large values of γ_j, the value of the corresponding diagonal coefficient of $S(\kappa, \gamma)$ is extremely close to 0 or 1. In those cases, the resulting model is weakly sparse in our numerical experiments. Thresholding can then be used to perform feature selection.

A-MLR for correlated designs and sparsity. Aggregation is a statistical technique which combines several estimators in order to attain higher generalization performance [201]. We propose in Definition 4 a new aggregation procedure to combine the estimators (3.5) and (3.6). This essentially consists in an interpolation between β^R and β^S models, where the coefficient of interpolation is quantified via the introduction of a new regularization parameter $\mu \in \mathbb{R}$.

Definition 4. *We consider the family of models*

$$\{\beta^{\mathcal{A}}(\lambda, \kappa, \gamma, \mu, \mathbb{X}, \mathbf{Y})\}_{(\lambda,\kappa,\gamma,\mu)\in\mathbb{R}_+^*\times\mathbb{R}_+^*\times\mathbb{R}^p\times\mathbb{R}}$$

with

$$\begin{aligned}\beta^{\mathcal{A}}(\theta, \mathbb{X}, \mathbf{Y}) = \mathbf{S}(\mu) \times \beta^R(\lambda, \mathbb{X}, \mathbf{Y})\\ + (1 - \mathbf{S}(\mu)) \times \beta^S(\lambda, \kappa, \gamma, \mathbb{X}, \mathbf{Y}),\end{aligned} \tag{3.7}$$

where $\beta^R(\lambda, \mathbf{Y})$ *and* $\beta^S(\lambda, \kappa, \gamma, \mathbb{X}, \mathbf{Y})$ *are defined in* (3.5) *and* (3.6) *respectively and* $\mathbf{S}$ *is the sigmoid*[1].

Applying Definition 2 to this family, we can derive the A-MLR procedure: $\widehat{\beta^{\mathcal{A}}} = \beta^{\mathcal{A}}(\widehat{\theta})$ where $\widehat{\theta} = \arg\min_{\theta} \texttt{MLR}_{\beta^{\mathcal{A}}}(\theta)$. This procedure is designed to handle both correlation and sparsity.

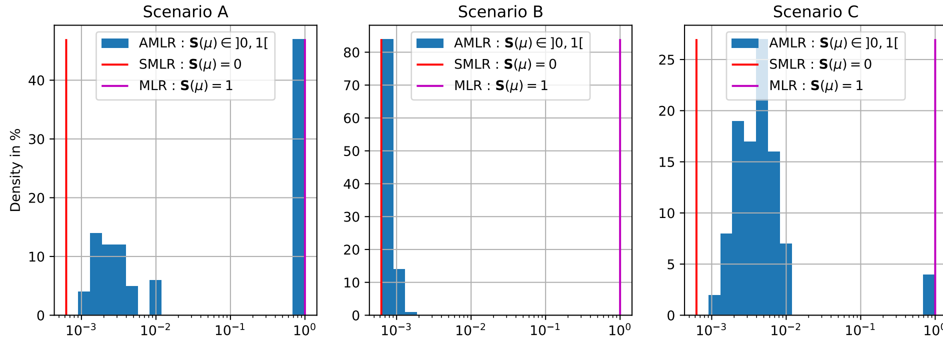


Figure 3.4: Distribution of the value of $\mathbf{S}(\widehat{\mu})$ over 100 repetitions on synthetic data.

Figure 3.4 shows $\widehat{\beta^{\mathcal{A}}}$ behaves almost as a selector which picks the most appropriate family of models between β^R and β^S. Indeed, in all scenarios, $\mathbf{S}(\widehat{\mu})$ only takes values close either to 0 or 1 in order to adapt to the structure of the model. Moreover in **Scenario B** (sparsity), $\widehat{\beta^{\mathcal{A}}}$ always selects the sparse model. Indeed we always have $\mathbf{S}(\widehat{\mu}) \leq 0.002$ over 100 repetitions.

[1] For $\mu \in \mathbb{R}$, $\mathbf{S}(\mu)$ takes values in $(0, 1)$ and is actually observed in practice to be close to 0 or 1.

Algorithmic complexity. Using the `MLR` criterion, we develop fully automatic procedures to tune regularization parameters while simultaneously training the model in a single run of the gradient descent algorithm without a hold-out *validation* set. The computational complexity of our methods is $O(n(p + r)K)$ where n, p, r, K denote respectively the number of observations, features, regularization parameters and iterations of the gradient descent algorithm. The computational complexity of our method grows only arithmetically w.r.t. the number of regularization parameters.

NUMERICAL EXPERIMENTS. We performed numerical experiments on the synthetic data from Section 3.2 and also on real datasets described below.

Real data. We test our methods on several commonly used real datasets (UCI [9] and Svmlib [36] repositories). See Appendix for more details. Each selected UCI dataset is splitted randomly into a 80% *train*-dataset and a 20% *test*-dataset. We repeat this operation $M = 100$ times to produce M pairs of $(train, test)$-datasets.

In order to test our procedures in the setting $n \leq p$, we selected, from Svmlib, the news20 dataset which contains a *train* and a *test* dataset. We fixed the number of features p and we sample six new 20news *train*-datasets of different sizes n from the initial news20 *train*-dataset. For each size n of dataset, we perform $M = 100$ repetitions of the sampling process to produce M *train*-datasets. We kept the initial *test*-set for the evaluation of the generalization performances.

Number of iterations. We choose to solve (3.4) using ADAM but other GD methods could be used. Figure 3.5 contains the boxplots of the number of ADAM iterations for the `MLR` procedures on the synthetic and real datasets over the $M =$ 100 repetitions. Although $\texttt{MLR}_{\beta^{\mathcal{S}}}$ and $\texttt{MLR}_{\beta^{\mathcal{A}}}$ are highly non-convex, the number of iterations required for convergence is always about a few several dozen in our experiments. This was already observed in other non-convex settings [113].

Running time. Our procedures were coded in Pytorch to underline how they can be parallelized on a GPU. A comparison of the running time with the benchmark procedures is not pertinent as they are implemented on cpu by Scikit-learn. The main point of our experiments was rather to show how the `MLR` procedures can be successfully parallelized. This opens promising prospects for the `MLR` approach in deep learning frameworks.

From a computational point of view, the matrix inversion in (3.5) is not expensive in our setting as long as the covariance matrix can fully fit on the

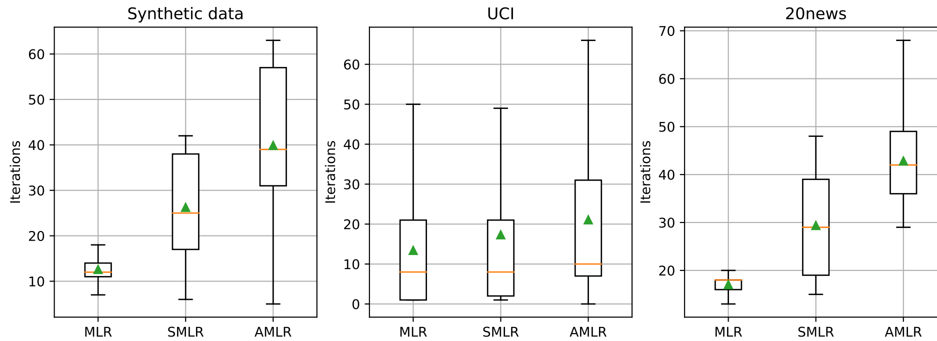


Figure 3.5: **Synthetic, UCI and 20news data**: Number of iterations

GPU[2]. Figures 3.6 and 3.7 confirm the running time is linear in n, p for the `MLR` procedures. This confirms the `MLR` procedures are scalable.

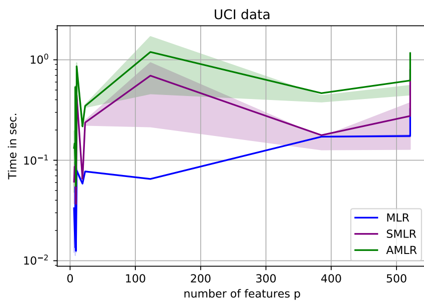


Figure 3.6: **UCI data:** running time as a function of n

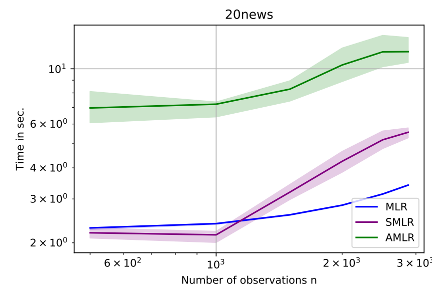


Figure 3.7: **20news data:** running times as a function of p

Consequently, our procedures run in reasonable time as illustrated in Figures 3.8 and 3.9.

Initial parameters. Strinkingly, the initial values of the parameters (see Table 3.1) used to implement our `MLR` procedures could remain the same for all the datasets we considered while still yielding consistently good prediction performances. These initial values were calibrated only once in the standard setting ($n \geq p$) on the Boston dataset [84, 16] which we did not include in our benchmark when we evaluated the performance of our procedures. We emphasize again we used these values without any modification on all the synthetic and real datasets. The synthetic and UCI datasets fall into the standard setting. Meanwhile,

[2] Inversion of a $p \times p$ matrix has a p^2 complexity on CPU, but parallelization schemes provide linear complexity on GPU when some memory constraints are met [139, 142, 44]

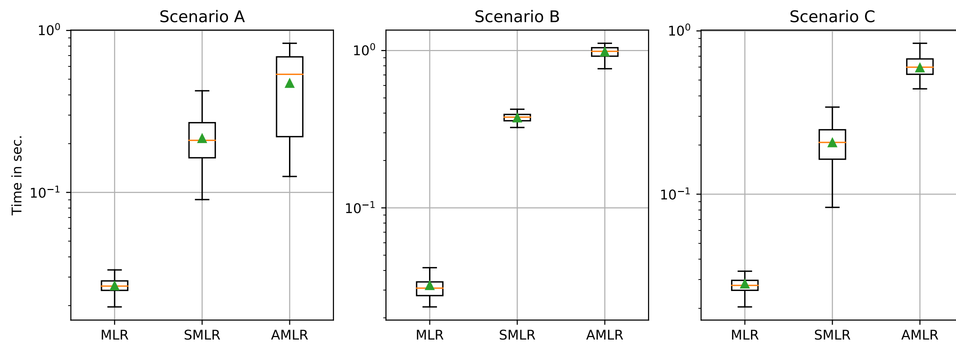


Figure 3.8: **Synthetic data:** running times in seconds.

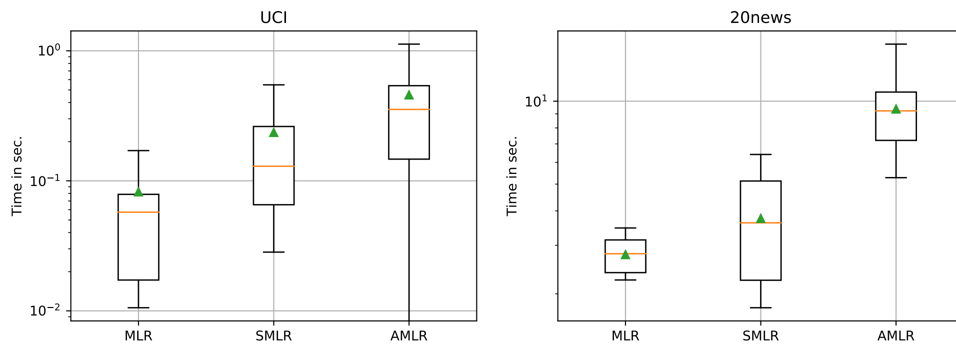


Figure 3.9: **UCI data** (left) and **20news data** (right): running times .

the 20news datasets correspond to the high-dimensional setting ($p \gg n$). As such, it might be possible to improve the generalization performance by using a different set of initial parameters better adapted to the high-dimensional setting. This will be investigated in future work.

However, in this chapter, we did not intend to improve the generalization performance by trying to tune the initial parameters for each specific dataset. This was not the point of this project. We rather wanted to highlight our gradient-based methods compare favorably in terms of generalization with benchmark procedures just by using the default initial values in Table 3.1.

Optimization parameters		**Parameter initialization**	
Tolerance	10^{-4}	T	30
Max. iter.	10^3	λ	10^3
Learning rate	0.5	γ	0_p
Adam β_1	0.5	κ	0.1
Adam β_2	0.9	μ	0

Table 3.1: Parameters for the `MLR` procedures.

We also studied the impact of parameter T on the performances of the `MLR` procedures on the synthetic data. In Figure 3.10, the generalization performance ($\mathbf{R^2}$-score) increases significantly from the first added permutation ($T = 1$) . Starting from $T \approx 10$, the $\mathbf{R^2}$-score has converged to its maximum value. An even more striking phenomenon is the gain observed in the running time when we add T permutations (for T in the range from 1 to approximately 100) when compared with the usual empirical risk ($T = 0$). Larger values of T are neither judicious nor needed in this approach. In addition, the needed number of iterations for ADAM to converge is divided by 3 starting from the first added permutation. Furthermore, this number of iterations remained stable (below 20) starting from $T = 1$. Based on these observations, **the hyperparameter T does not require calibration**. We fixed $T = 30$ in our experiments even though $T = 10$ might have been sufficient.

Performance comparisons. We compare our `MLR` procedures against cross-validated Ridge, LASSO and Elastic-net (implemented as RidgeCV, LassoCV and ElasticnetCV in Scikit-learn [154]) on simulated and real datasets. Our procedures are implemented in PyTorch [151] on the centered and rescaled response $\mathbf{Y}$. Complete details and results can be found in the Appendix. In our approach θ can always be tuned directly on the *train* set whereas for benchmark procedures like

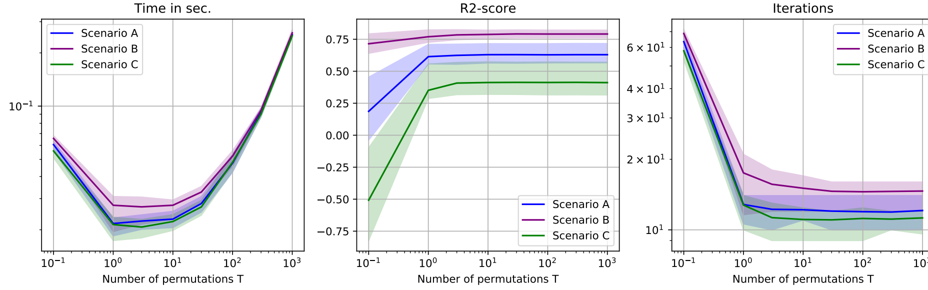


Figure 3.10: **Synthetic data:** impact of T on the `MLR` procedures.

LASSO , Ridge, Elastic net, θ is typically calibrated on a hold-out *validation*-set using grid-search CV for instance.

Generalisation performance. Figures 3.11 and 3.12 show the `MLR` procedures consistently attain the highest $\mathbf{R^2}$-scores for the synthetic and UCI data according to the Mann-Whitney test over the $M = 100$ repetitions. Regarding, the 20news datasets, the `MLR` procedures are always within 0.05 of the best (E-net).

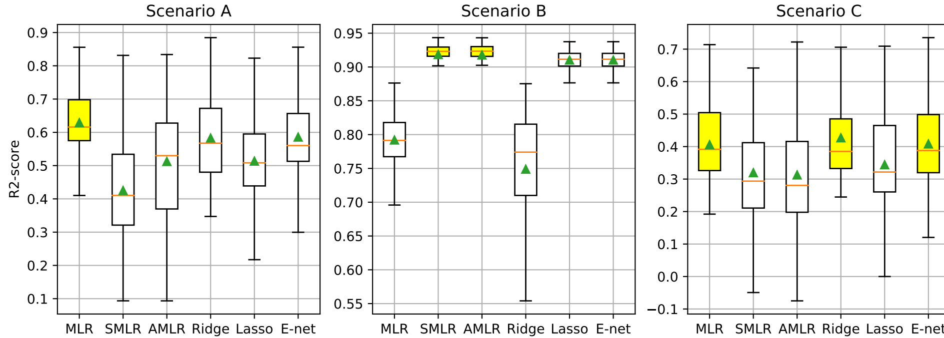


Figure 3.11: **Synthetic data:** R2-score

Estimation of β^* and support recovery accuracy. For the synthetic data, we also consider the estimation of the regression vector β^*. We use the l_2-norm estimation error $\|\widehat{\beta} - \beta^*\|_2$ to compare the procedures. As we can see in Figure 3.13, the `MLR` procedures perform better than the benchmark procedures.

We finally study the support recovery accuracy in the sparse setting (**Scenario B**). We want to recover the support $J(\beta^*) = \left\{ j : \beta^*_j \neq 0 \right\}$. For our procedures, we build the following estimator $\widehat{J(\beta)} = \left\{ j : |\widehat{\beta}_j| > \widehat{\tau} \right\}$ where the threshold $\widehat{\tau}$ corresponds to the first sharp decline of the coefficients $|\widehat{\beta}_j|$. Denote by $\#J$ the

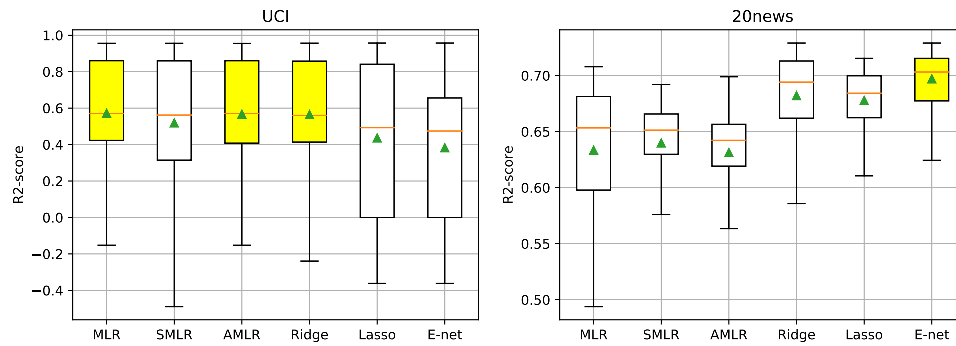


Figure 3.12: **UCI data** (left) and **20news data** (right): R2-score

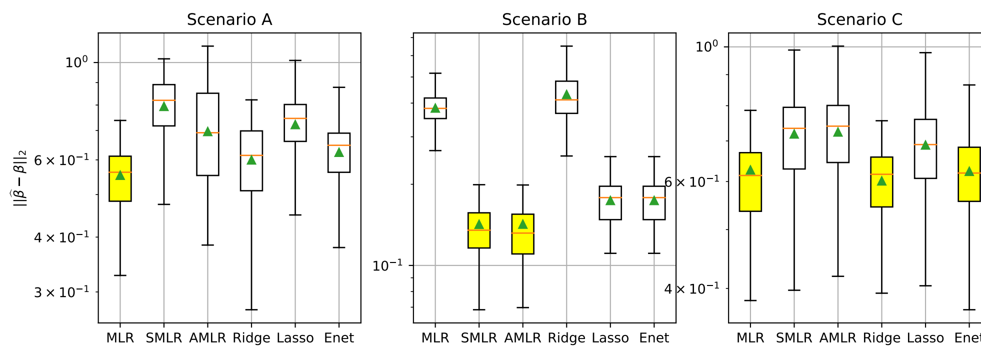


Figure 3.13: β^* estimation (best in yellow according to MW)

cardinality of set J. The support recovery accuracy is measured as follows:

$$Acc(\widehat{\beta}) := \frac{\#\{J(\beta^*) \cap \widehat{J(\widehat{\beta})}\} + \#\{J^c(\beta^*) \cap \widehat{J^c}(\widehat{\beta})\#\}}{p},$$

Our simulations confirm $\widehat{\beta}^S$ is a quasi-sparse vector. Indeed we observe in Figure 3.14 a sharp decline of the coefficients $|\widehat{\beta}^S_j|$. Thus we set the threshold $\widehat{\tau}$ at 10^{-3}.

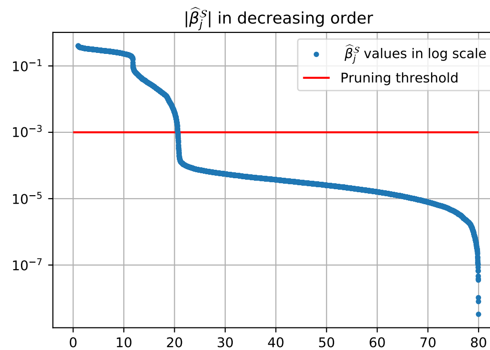


Figure 3.14: Coefficients values (blue) and threshold (red) with $p = 80$.

Overall, $\widehat{\beta}^S$ and $\widehat{\beta}^{\mathcal{A}}$ perform better for support recovery than the benchmark procedures. Moreover in **Scenario B** favorable to LASSO, our procedures perform far better (Figure 3.15).

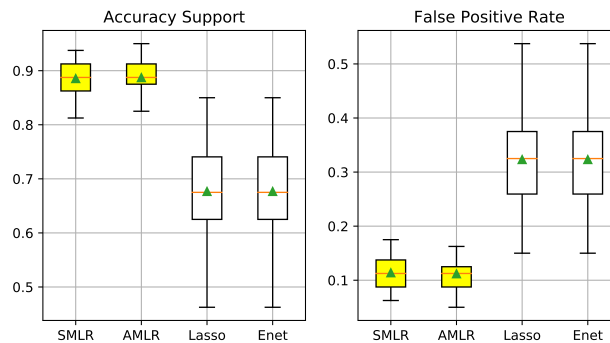


Figure 3.15: Support recovery performance analysis in **Scenario B** (best in yellow according to MW).

3.4 Conclusion

In this chapter, we introduced in the linear regression setting the new `MLR` approach based on a different understanding of generalization. Exploiting this idea, we

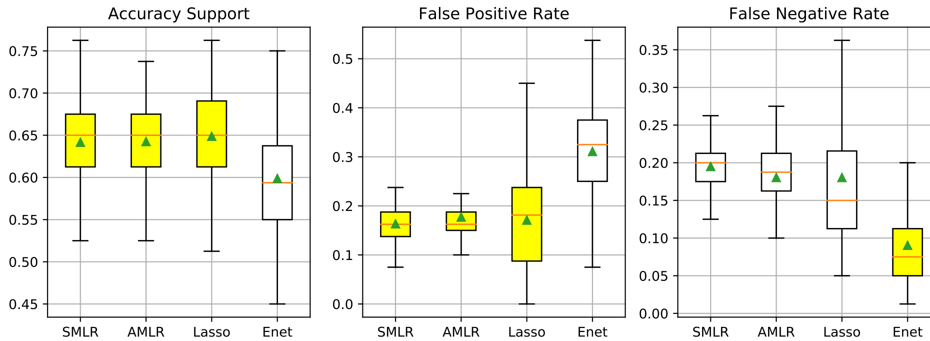


Figure 3.16: Support recovery performance analysis in **Scenario C** (best in yellow according to MW).

derived a novel criterion and new procedures which can be implemented directly on the *train*-set without any hold-out *validation*-set. Within MLR, additional structures can be taken into account without any significant increase in the computational complexity.

We highlighted several additional advantageous properties of the MLR approach in our numerical experiments. The MLR approach is computationally feasible while yielding statistical performances equivalent or better than the cross-validated benchmarks.

We provided numerical evidence of MLR criterion's ability to generalize from the first added permutation. Besides, the strength of our MLR procedures stems from their compatibility with gradient-based optimization methods. As such, these procedures can fully benefit from automatic graph-differentiation libraries (such as pytorch [150] and tensorflow [133]). In our numerical experiments, adding more permutations improves the convergence of the ADAM optimizer while preserving generalisation. As a matter of fact, T does not require any fine-tuning. In that regard, T is not a hyperparameter. Likewise, the other hyperparameters require no tedious initialization in this framework. The same fixed hyperparameters for ADAM and initialization values of the regularization parameters (see Table 3.1) were used for all the considered datasets. Noticeably, these experiments were run using high values for learning rate and convergence threshold. Consequently, only a very small number of iterations were needed, even for non-convex criteria ($\texttt{MLR}_{\beta^S}$ and $\texttt{MLR}_{\beta^{\mathcal{A}}}$).

Beyond the results provided in this chapter, we successfully extended the MLR approach to deep neural networks which is the subject of Chapter 4. Neural networks trained with the MLR criterion can reach state of the art results on benchmarks

usually dominated by Random Forest and Gradient Boosting techniques.

3.5 Appendix

3.5.1 Proof of Theorem 1

Assume $\mathrm{rank}(\mathbf{x}) = r$. Consider the SVD of $\frac{1}{\sqrt{n}}\mathbf{x}$ and denote by $\lambda_1, \ldots, \lambda_r$ the singular values with corresponding left and right eigenvectors $\{\mathbf{u}_j\}_{j=1}^r \in \mathbb{R}^n$ and $\{\mathbf{v}_j\}_{j=1}^r \in \mathbb{R}^p$:

$$\frac{1}{\sqrt{n}}\mathbf{x} = \sum_{j=1}^{r} \sqrt{\lambda_j}\mathbf{u}_j \otimes \mathbf{v}_j. \tag{3.8}$$

Define $\mathbf{H}_\lambda := \mathbf{x}(\mathbf{x}^\top\mathbf{x} + \lambda\mathbb{I}_d)^{-1}\mathbf{x}^\top$. We easily get

$$\mathbf{H}_\lambda = \sum_{j=1}^{r} \frac{\lambda_j}{\lambda_j + \lambda/n}\mathbf{u}_j \otimes \mathbf{u}_j.$$

Compute first the population risk of Ridge model $\boldsymbol{\beta}_\lambda$. Exploiting the previous display, we have

$$\|\mathbf{x}\boldsymbol{\beta}^* - \mathbf{x}\boldsymbol{\beta}_\lambda\|_2^2 = \sum_{j=1}^{r} \frac{(\lambda/n)^2}{(\lambda_j + \lambda/n)^2}\langle \mathbf{x}\boldsymbol{\beta}^*, \mathbf{u}_j\rangle^2 + \sum_{j=1}^{r} \frac{\lambda_j^2}{(\lambda_j + \lambda/n)^2}\langle \mathbf{u}_j, \boldsymbol{\xi}\rangle^2 - 2\langle(\mathbb{I}_n - \mathbf{H}_\lambda)\mathbf{x}\boldsymbol{\beta}^*, \mathbf{H}_\lambda\boldsymbol{\xi}\rangle.$$

Taking the expectation *w.r.t.* $\boldsymbol{\xi}$ and as $\boldsymbol{\xi}$ is centered, we get

$$R(\lambda) = \mathbb{E}_\xi\left[\|\mathbf{x}\boldsymbol{\beta}^* - \mathbf{x}\boldsymbol{\beta}_\lambda\|_2^2\right] = \sum_{j=1}^{r} \frac{(\lambda/n)^2}{(\lambda_j + \lambda/n)^2}\langle \mathbf{x}\boldsymbol{\beta}^*, \mathbf{u}_j\rangle^2 + \sigma^2 \sum_{j=1}^{r} \frac{\lambda_j^2}{(\lambda_j + \lambda/n)^2}. \tag{3.9}$$

Next, compute the representations for the empirical risk for the original data set $(\mathbf{x}, \mathbf{Y})$ and artificial data set $(\mathbf{x}, \mathbf{Y}_{\texttt{perm}})$ obtained by random permutation of $\mathbf{Y}$. We obtain respectively

$$\|\mathbf{Y} - \mathbf{x} - \mathbf{x}\boldsymbol{\beta}_\lambda(\mathbf{x}, \mathbf{Y})\|_2^2 = \|(\mathbb{I}_n - \mathbf{H}_\lambda)\mathbf{Y}\|_2^2 = \sum_{j=1}^{r} \left(\frac{\lambda/n}{\lambda_j + \lambda/n}\right)^2 \langle \mathbf{Y}, u_j\rangle^2, \tag{3.10}$$

$$\|\mathbf{Y}_{\texttt{perm}} - \mathbf{x}\boldsymbol{\beta}_\lambda(\mathbf{x}, \mathbf{Y}_{\texttt{perm}})\|_2^2 = \|(\mathbb{I}_n - \mathbf{H}_\lambda)\mathbf{Y}_{\texttt{perm}}\|_2^2 = \sum_{j=1}^{r} \left(\frac{\lambda/n}{\lambda_j + \lambda/n}\right)^2 \langle \mathbf{Y}_{\texttt{perm}}, u_j\rangle^2. \tag{3.11}$$

As $(\lambda/n)^2 = (\lambda_j + \lambda/n)^2 - 2\lambda_j\lambda/n - \lambda_j^2$, it results the following representation for the MLR criterion:

$$\begin{aligned}\mathtt{MLR}_\beta(\lambda) &= \|\mathbf{Y} - \mathbf{x}\boldsymbol{\beta}_\lambda(\mathbf{x},\mathbf{Y})\|_2^2 - \|\mathbf{Y} - \mathbf{x}\boldsymbol{\beta}_\lambda(\mathbf{x},\mathbf{Y}_{\mathtt{perm}})\|_2^2\\ &= -\|P_{\mathbf{x}}(\mathbf{Y}_{\mathtt{perm}})\|_2^2 + \sum_{j=1}^{r}\left(\frac{\lambda/n}{\lambda_j+\lambda/n}\right)^2\langle\mathbf{Y},\mathbf{u}_j\rangle^2 + \sum_{j=1}^{r}\frac{2\lambda_j\lambda/n+\lambda_j^2}{(\lambda_j+\lambda/n)^2}\langle\mathbf{Y}_{\mathtt{perm}},\mathbf{u}_j\rangle^2.\end{aligned} \tag{3.12}$$

Let's consider now the simple case $\lambda_1 = \cdots = \lambda_r = 1$ corresponding to $\mathbf{x}^\top\mathbf{x}/n$ being an orthogonal projection of rank r. Then, (3.9) and (3.12) become respectively

$$R(\lambda) = \frac{(\lambda/n)^2}{(1+\lambda/n)^2}\sum_{j=1}^{r}\langle\mathbf{x}\boldsymbol{\beta}^*,\mathbf{u}_j\rangle^2 + \sigma^2\sum_{j=1}^{r}\frac{1}{(1+\lambda/n)^2} = \frac{(\lambda/n)^2}{(1+\lambda/n)^2}\|\mathbf{x}\boldsymbol{\beta}^*\|^2 + \frac{1}{(1+\lambda/n)^2}\sigma^2 r. \tag{3.13}$$

$$\mathtt{MLR}_\beta(\lambda) + \|P_{\mathbf{x}}(\mathbf{Y}_{\mathtt{perm}})\|_2^2 = \left(\frac{\lambda/n}{1+\lambda/n}\right)^2\|P_{\mathbf{x}}(\mathbf{Y})\|_2^2 + \frac{2\lambda/n+1}{(1+\lambda/n)^2}\|P_{\mathbf{x}}(\mathbf{Y}_{\mathtt{perm}})\|_2^2. \tag{3.14}$$

Since $\|P_{\mathbf{x}}(\mathbf{Y}_{\mathtt{perm}})\|_2^2$ does not depend on λ, minimizing $\mathtt{MLR}_\beta(\lambda)$ is equivalent to minimizing

$$\left(\frac{\lambda/n}{1+\lambda/n}\right)^2\|P_{\mathbf{x}}(\mathbf{Y})\|_2^2 + \frac{2\lambda/n+1}{(1+\lambda/n)^2}\|P_{\mathbf{x}}(\mathbf{Y}_{\mathtt{perm}})\|_2^2. \tag{3.15}$$

Comparing the previous display with (3.9), we observe that $\mathtt{MLR}_\beta(\lambda) + \|P_{\mathbf{x}}(\mathbf{Y}_{\mathtt{perm}})\|_2^2$ looks like the population risk $\mathbb{E}_\xi\left[\|\mathbf{x}\boldsymbol{\beta}^* - \mathbf{x}\boldsymbol{\beta}_\lambda\|_2^2\right]$.

Next state the following fact proved in section 3.5.3

Fact 1.

$$MLR_\beta(\lambda) + \|P_{\mathbf{x}}(\boldsymbol{Y}_{perm})\|_2^2 = \left(\frac{1+\lambda/n+\mathsf{a}}{1+\lambda/n}\right)^2\widehat{R}(\lambda+n\mathsf{a}) - \frac{\mathsf{a}\|P_{\mathbf{x}}(\boldsymbol{Y}_{perm})\|_2^2}{(1+\lambda/n)^2} \tag{3.16}$$

Before proving our theorem, define the following quantity a

$$\mathsf{a} := \frac{\|P_{\mathbf{x}}(\mathbf{Y}_{\mathtt{perm}})\|_2^2}{\|P_{\mathbf{x}}(\mathbf{Y})\|_2^2}$$

and state an intermediate result (lemma 1- proved in section 3.5.2)

Lemma 1. *Consider the simple case $\lambda_1 = \cdots = \lambda_r = 1$ corresponding to $\boldsymbol{x}^\top \boldsymbol{x}/n$ being an orthogonal projection of rank r and define*

$$\widehat{R}(\lambda) = \frac{(\lambda/n)^2}{(1+\lambda/n)^2}\|P_{\boldsymbol{x}}(\mathbf{Y})\|_2^2 + \frac{1}{(1+\lambda/n)^2}\|P_{\boldsymbol{x}}(\boldsymbol{Y}_{\texttt{perm}})\|_2^2. \quad (3.17)$$

Then, in the intermediate SNR regime $r\sigma^2 \ll \|\boldsymbol{x\beta}^*\|_2^2 \ll n\sigma^2$, it comes

$$\widehat{R}(\lambda + n\mathsf{a}) = \widehat{R}(\lambda)\left(1 + o\left(\frac{a}{(\lambda/n)}\right)\right) \quad w.h.p., \quad (3.18)$$

$$\widehat{R}(\lambda)) = R(\lambda)\left(1 + O\left(\epsilon_n\right)\right) \quad w.h.p. \quad (3.19)$$

where $\epsilon_n = \sqrt{\frac{\|\boldsymbol{x\beta}^*\|_2^2}{n\sigma^2}} + \sqrt{\frac{r\sigma^2}{\|\boldsymbol{x\beta}^*\|_2^2}}$.

Starting from (3.16) and using successively (3.18) and (3.19), we have *w.h.p.*

$$\texttt{MLR}_\beta(\lambda) + \|P_{\mathbf{x}}(\mathbf{Y}_{\texttt{perm}})\|_2^2 = \left(1 + \frac{\mathsf{a}}{1+\lambda/n}\right)^2 \widehat{R}(\lambda + n\mathsf{a}) - \frac{\mathsf{a}\|P_{\mathbf{x}}(\mathbf{Y}_{\texttt{perm}})\|_2^2}{(1+\lambda/n)^2}$$

Combining (3.18), (3.19) and (3.16)

$$\begin{aligned}\texttt{MLR}_\beta(\lambda) + \|P_{\mathbf{x}}(\mathbf{Y}_{\texttt{perm}})\|_2^2 &= \left(1 + \frac{\mathsf{a}}{1+\lambda/n}\right)^2 \widehat{R}(\lambda)\left(1 + o\left(\frac{a}{(\lambda/n)}\right)\right) - \frac{\mathsf{a}\|P_{\mathbf{x}}(\mathbf{Y}_{\texttt{perm}})\|_2^2}{(1+\lambda/n)^2} \\ &= \left(1 + \frac{\mathsf{a}}{1+\lambda/n}\right)^2 R(\lambda)\left(1 + O(\epsilon_n) + o\left(\frac{a}{(\lambda/n)}\right)\right) - \frac{\mathsf{a}\|P_{\mathbf{x}}(\mathbf{Y}_{\texttt{perm}})\|_2^2}{(1+\lambda/n)^2}\end{aligned}$$

Moreover by (3.33) in lemma 2, we have

$$\|P_{\mathbf{x}}(\mathbf{Y}_{\texttt{perm}})\|_2^2 = r\sigma^2\left(1 + O\left(\sqrt{\frac{\|\mathbf{x}\boldsymbol{\beta}^*\|_2^2}{n\sigma^2}}\right)\right).$$

Then, we get

$$\texttt{MLR}_\beta(\lambda) + \|P_{\mathbf{x}}(\mathbf{Y}_{\texttt{perm}})\|_2^2 = \left(1 + \frac{\mathsf{a}}{1+\lambda/n}\right)^2 R(\lambda)\left(1 + O(\epsilon_n)\right) - \frac{\mathsf{a}r\sigma^2}{(1+\lambda/n)^2}\left(1 + O\left(\sqrt{\frac{\|\mathbf{x}\boldsymbol{\beta}^*\|_2^2}{n\sigma^2}}\right)\right).$$

In the intermediate SNR regime $r\sigma^2 \ll \|\mathbf{x}\boldsymbol{\beta}^*\|_2^2 \ll n\sigma^2$, by lemma 2 (section 3.5.5)

$$\mathsf{a} := \frac{\|P_{\mathbf{x}}(\mathbf{Y}_{\texttt{perm}})\|_2^2}{\|P_{\mathbf{x}}(\mathbf{Y})\|_2^2} = \frac{\sigma^2 r}{\|\mathbf{x}\boldsymbol{\beta}^*\|_2^2}(1 + o(1)), \quad w.h.p.. \quad (3.20)$$

Then, $r\sigma^2 \ll \|\mathbf{x}\boldsymbol{\beta}^*\|_2^2 \ll n\sigma^2 \Rightarrow \epsilon_n = o(1)$ and $\forall \lambda/n >> \mathsf{a}$

$$\texttt{MLR}_\beta(\lambda) + \|P_{\mathbf{x}}(\mathbf{Y}_{\texttt{perm}})\|_2^2 = R(\lambda)(1 + o(1)).$$

3.5.2 Proof of Lemma 1

Consider the simple case $\lambda_1 = \cdots = \lambda_r = 1$ corresponding to $\mathbf{x}^\top \mathbf{x}/n$ being an orthogonal projection of rank r. Recall

$$\widehat{R}(\lambda) = \frac{(\lambda/n)^2}{(1+\lambda/n)^2}\|P_{\mathbf{x}}(\mathbf{Y})\|_2^2 + \frac{1}{(1+\lambda/n)^2}\|P_{\mathbf{x}}(\mathbf{Y}_{\texttt{perm}})\|_2^2. \tag{3.21}$$

Proof of equation (3.18)**.** We have

$$\widehat{R}(\lambda + n\mathsf{a}) \;=\; \|P_{\mathbf{x}}(\mathbf{Y})\|_2^2 \frac{(\lambda/n+\mathsf{a})^2}{(1+\lambda/n+\mathsf{a})^2} + \|P_{\mathbf{x}}(\mathbf{Y}_{\texttt{perm}})\|_2^2 \frac{1}{(1+\lambda/n+\mathsf{a})^2}$$

Next in the intermediate SNR regime $r\sigma^2 \ll \|\mathbf{x}\boldsymbol{\beta}^*\|_2^2 \ll n\sigma^2$, by lemma 2 (section 3.5.5)

$$\mathsf{a} := \frac{\|P_{\mathbf{x}}(\mathbf{Y}_{\texttt{perm}})\|_2^2}{\|P_{\mathbf{x}}(\mathbf{Y})\|_2^2} = \frac{\sigma^2 r}{\|\mathbf{x}\boldsymbol{\beta}^*\|_2^2}(1+o(1)), \quad \textit{w.h.p.}.$$

Then, $\forall (\lambda/n) >> \mathsf{a}$, we get

$$\begin{aligned}
\frac{(\lambda/n+\mathsf{a})^2}{(1+\lambda/n+\mathsf{a})^2} &= \frac{(\lambda/n)^2}{(1+(\lambda/n))^2}\left[\frac{1+\frac{2a}{(\lambda/n)}+\left(\frac{2a}{(\lambda/n)}\right)^2}{(1+\frac{a}{1+(\lambda/n)})^2}\right] = \frac{(\lambda/n)^2}{(1+(\lambda/n))^2}(1+o\left(\frac{a}{(\lambda/n)}\right),\\
\frac{1}{(1+\lambda/n+\mathsf{a})^2} &= \frac{1}{(1+(\lambda/n))^2}\left[\frac{1}{(1+\frac{a}{1+(\lambda/n)})^2}\right] = \frac{1}{(1+(\lambda/n))^2}(1+o\left(\frac{a}{(\lambda/n)}\right).
\end{aligned}$$

Therefore, we get the first ingredient to prove our theorem:

$$\widehat{R}(\lambda+n\mathsf{a}) \;=\; \widehat{R}(\lambda)(1+o\left(\frac{a}{(\lambda/n)}\right).$$

Proof of equation (3.19)**.** Compute now

$$\widehat{R}(\lambda) - R(\lambda) \;=\; \frac{(\lambda/n)^2}{(1+\lambda/n)^2}\|\left(\|P_{\mathbf{x}}((\mathbf{Y})\|_2^2 - \mathbf{x}\boldsymbol{\beta}^*\|^2\right) + \frac{1}{(1+\lambda/n)^2}\left(\|P_{\mathbf{x}}(\mathbf{Y}_{\texttt{perm}})\|_2^2 - \sigma^2 r\right).$$

As by lemma 2, we have *w.h.p.*

$$\|P_{\mathbf{x}}(\mathbf{Y})\|_2^2 = \|\mathbf{x}\boldsymbol{\beta}^*\|_2^2\left(1+O\left(\sqrt{\frac{r\sigma^2}{\|\mathbf{x}\boldsymbol{\beta}^*\|_2^2}}\right)\right), \tag{3.22}$$

$$\|P_{\mathbf{x}}(\mathbf{Y}_{\texttt{perm}})\|_2^2 = r\sigma^2\left(1+O\left(\sqrt{\frac{\|\mathbf{x}\boldsymbol{\beta}^*\|_2^2}{n\sigma^2}}\right)\right). \tag{3.23}$$

Then, we get the second ingredient to prove our theorem.

$$\widehat{R}(\lambda) = R(\lambda)\left(1+\left(O\left(\sqrt{\frac{r\sigma^2}{\|\mathbf{x}\boldsymbol{\beta}^*\|_2^2}}\right)+O\left(\sqrt{\frac{\|\mathbf{x}\boldsymbol{\beta}^*\|_2^2}{n\sigma^2}}\right)\right)\right) = R(\lambda)\left(1+O\left(\epsilon_n\right)\right),$$

where

$$\epsilon_n := \sqrt{\frac{r\sigma^2}{\|\mathbf{x}\boldsymbol{\beta}^*\|_2^2}} + \sqrt{\frac{\|\mathbf{x}\boldsymbol{\beta}^*\|_2^2}{n\sigma^2}}.$$

3.5.3 Fact 1

Compute the following quantity

$$\begin{aligned} I &:= \left(\frac{1+\lambda/n+\mathsf{a}}{1+\lambda/n}\right)^2 \widehat{R}(\lambda+n\mathsf{a}) - \frac{\mathsf{a}\|P_{\mathbf{x}}(\mathbf{Y}_{\texttt{perm}})\|_2^2}{(1+\lambda/n)^2} \\ &= \frac{(\lambda/n+\mathsf{a})^2}{(1+\lambda/n)^2}\|P_{\mathbf{x}}(\mathbf{Y})\|_2^2 + \frac{1}{(1+\lambda/n)^2}\|P_{\mathbf{x}}(\mathbf{Y}_{\texttt{perm}})\|_2^2 - \frac{\mathsf{a}\|P_{\mathbf{x}}(\mathbf{Y}_{\texttt{perm}})\|_2^2}{(1+\lambda/n)^2} \\ &= \frac{(\lambda/n)^2}{(1+\lambda/n)^2}\|P_{\mathbf{x}}(\mathbf{Y})\|_2^2 + \frac{\mathsf{a}^2}{(1+\lambda/n)^2}\|P_{\mathbf{x}}(\mathbf{Y})\|_2^2 + \frac{2(\lambda/n)\mathsf{a}}{(1+\lambda/n)^2}\|P_{\mathbf{x}}(\mathbf{Y})\|_2^2 \\ &\quad + \frac{1}{(1+\lambda/n)^2}\|P_{\mathbf{x}}(\mathbf{Y}_{\texttt{perm}})\|_2^2 - \frac{\mathsf{a}\|P_{\mathbf{x}}(\mathbf{Y}_{\texttt{perm}})\|_2^2}{(1+\lambda/n)^2}. \end{aligned}$$

By (3.20) we have $\mathsf{a}\|P_{\mathbf{x}}(\mathbf{Y})\|_2^2 = \|P_{\mathbf{x}}(\mathbf{Y}_{\texttt{perm}})\|_2^2$, then

$$\begin{aligned} I &= \frac{(\lambda/n)^2}{(1+\lambda/n)^2}\|P_{\mathbf{x}}(\mathbf{Y})\|_2^2 + \frac{\mathsf{a}}{(1+\lambda/n)^2}\|P_{\mathbf{x}}(\mathbf{Y}_{\texttt{perm}})\|_2^2 + \frac{2(\lambda/n)}{(1+\lambda/n)^2}\|P_{\mathbf{x}}(\mathbf{Y}_{\texttt{perm}})\|_2^2 \\ &\quad + \frac{1}{(1+\lambda/n)^2}\|P_{\mathbf{x}}(\mathbf{Y}_{\texttt{perm}})\|_2^2 - \frac{\mathsf{a}\|P_{\mathbf{x}}(\mathbf{Y}_{\texttt{perm}})\|_2^2}{(1+\lambda/n)^2} \\ &= \frac{(\lambda/n)^2}{(1+\lambda/n)^2}\|P_{\mathbf{x}}(\mathbf{Y})\|_2^2 + \frac{2(\lambda/n)+1}{(1+\lambda/n)^2}\|P_{\mathbf{x}}(\mathbf{Y}_{\texttt{perm}})\|_2^2 \\ &= \texttt{MLR}_\beta(\lambda) + \|P_{\mathbf{x}}(\mathbf{Y}_{\texttt{perm}})\|_2^2. \end{aligned}$$

Therefore, we get

$$\texttt{MLR}_\beta(\lambda) + \|P_{\mathbf{x}}(\mathbf{Y}_{\texttt{perm}})\|_2^2 = \left(\frac{1+\lambda/n+\mathsf{a}}{1+\lambda/n}\right)^2 \widehat{R}(\lambda+n\mathsf{a}) - \frac{\mathsf{a}\|P_{\mathbf{x}}(\mathbf{Y}_{\texttt{perm}})\|_2^2}{(1+\lambda/n)^2}.$$

Next, we investigate the impact of random permutations on the geometry of correlated design matrices.

3.5.4 Proposition 1

In a nutshell, consider an isotropic subgaussian random vector $Z \in \mathbb{R}^n$. The deviations of quadratic forms $\|Z\|^2$ and $\langle Z, \pi(Z)\rangle$ from their respective expectations are of size $\sqrt{n}$. Thus the dominating term in $\|Z\|^2$ is its expectation $\mathbb{E}[\|Z\|^2] = n$. However, the expectation of $\langle Z, \pi(Z)\rangle$ is equal to 1. The random permutation almost cancelled out the expectation term. Consequently, the correlation of Z with $\pi(Z)$ is of order $1/\sqrt{n}$ w.h.p. The `MLR` criterion exploits this behavior of randomly permuted random vectors to build a good estimator of the true prediction error. We make this observation rigorous below.

Property 1. *Let $(\eta^{(l)})_{l\geq 1}$ be independent, isotropic subgaussian random vectors in $\mathbb{R}^n$ with $\max_{l\geq 1\{}\|\eta^{(l)}\|_{\psi_2}\} \leq K$ for some $K < \infty$. Let $h, h' \in \mathbb{H}$. We consider the random variables in $\mathbb{R}^n$*

$$Z_h = \sum_{k\geq 1} h_k\, \eta^{(k)}, \quad Z_{h'} = \sum_{k\geq 1} h'_k\, \eta^{(k)}.$$

Let π be a permutation in $\mathfrak{S}_n$ drawn uniformly at random. Then we have for any $t > 0$, with probability at least $1 - e^{-t} - e^{-\sqrt{n}}$,

$$\frac{|\langle Z_h, \pi(Z_{h'})\rangle|}{\|Z_h\|\|Z_{h'}\|} \lesssim \frac{1}{\sqrt{n}} \bigvee K^2 \sqrt{\frac{t}{n}} \bigvee K^2 \frac{t}{n}. \tag{3.24}$$

Using the assumption on $\mathbf{x}$ given in the beginning of Section 3.5.1, we have

$$\mathbb{X} = \sum_{l=1}^{r} \sqrt{\lambda_l}\, \eta^{(l)} \otimes \mathbf{v}_l, \tag{3.25}$$

where we set $\eta^{(l)} = (\eta_1^{(l)}, \dots, \eta_n^{(l)})^\top \in \mathbb{R}^n$. Thus we get

$$\mu^* := \mathbb{X}\beta^* = \sum_{l=1}^{r} \sqrt{\lambda_l}\, \langle \beta^*, \mathbf{v}_l\rangle\, \eta^{(l)}.$$

Let π be a random permutation uniformly distributed in $\mathfrak{S}_n$. We define

$$\pi(\mu^*) = \pi(\mathbb{X}\beta^*) = \sum_{l=1}^{r} \sqrt{\lambda_l}\, \langle \beta^*, \mathbf{v}_l\rangle\, \pi(\eta^{(l)}).$$

We want to bound the correlation between $\pi(\mu^*)$ and μ^*:

$$\frac{\langle \pi(\mu^*), \mu^*\rangle}{\|\mu^*\|\|\pi(\mu^*)\|} = \frac{\langle \pi(\mu^*), \mu^*\rangle}{\|\mu^*\|^2},$$

since we obviously have $\|\pi(\mu^*)\| = \|\mu^*\|$.

We are interested in the impact of random permutations on the correlation of $\mu^* = \mathbb{X}\beta^*$ with itself. Applying Proposition 1 with $h = h' = (\sqrt{\lambda_l}\langle\beta^*, \mathbf{v}_l\rangle)_{l\geq 1}$ gives w.h.p.

$$\frac{|\langle\pi(\mu^*), \mu^*\rangle|}{\|\mu^*\|^2} \lesssim \frac{1 \vee K^2}{\sqrt{n}}.$$

Proof of Proposition 1. We define the operator

$$A = (h \otimes h') \otimes \Pi \ = \ (h_k\, h'_l)_{k,l\geq 1} \otimes \Pi, \tag{3.26}$$

where Π is the permutation matrix corresponding to π. Set $\eta = ((\eta^{(1)})^\top, (\eta^{(2)})^\top, \ldots)$. We have

$$\langle Z_h, \pi(Z_{h'})\rangle = \sum_{k,l\geq 1} h_k h'_l \langle \eta^{(k)}, \pi(\eta^{(l)})\rangle = \eta^\top A \eta.$$

Next we note that

$$\|A\|_{HS} \ = \|h\|\,\|h'\|\ \|\Pi\|_{HS} \ = \|h\|\,\|h'\|\ \sqrt{n},$$

and

$$\|A\| = \|(h_k\, h'_l)_{k,l\geq 1}\|\ \|\Pi\| = \|h\|\,\|h'\|.$$

Then, we will use the following concentration result [82, 214, 173], which concerns quadratic forms of subgaussian random variables.

Property 2 ([173])**.** *Let $Z = (Z_1, \ldots, Z_n)^\top \in \mathbb{R}^n$ be a random vector with independent components satisfying $\mathbb{E}[Z_i] = 0$ and $\|Z_i\|_{\psi_2} \leq K$, for all $i = 1, \ldots, n$. Let A be a $n \times n$ matrix. Then, for every $t > 0$, we have with probability at least $1 - e^{-t}$,*

$$\left|Z^\top A Z - \mathbb{E}[Z^\top A Z]\right| \lesssim K^2 \max\left(\|A\|_{HS}\ \sqrt{t}, \|A\|\, t\right), \tag{3.27}$$

where $\|A\|_{HS}$ and $\|A\|$ are the Hilbert-Schmidt and operator norms respectively.

Applying Proposition 2 conditionally on π, we get for any $t > 0$, with probability at least $1 - e^{-t}$,

$$\begin{aligned} \left|\eta^\top A \eta - \mathbb{E}[\eta^\top A \eta|\pi]\right| &\lesssim K^2 \max\{\|A\|_{HS}\ \sqrt{t}, \|A\| t\} \\ &\lesssim K^2 \|h\|\,\|h'\|\ \max\{\sqrt{n\, t}, t\}. \end{aligned} \tag{3.28}$$

We study next the expectation term. Since the $\eta^{(k)}$'s are independent and zero mean, we have that

$$\mathbb{E}[\eta^\top A\eta|\pi] = \sum_{k,l\geq 1} h_k\, h'_l\, \mathbb{E}[\langle \eta^{(k)}, \pi(\eta^{(l)})\rangle|\pi] = \sum_{k\geq 1} h_k\, h'_k\, \mathbb{E}[\langle \eta^{(k)}, \pi(\eta^{(k)})\rangle|\pi].$$

Next, using that the $\eta_i^{(k)}$ are independent, zero mean with variance 1, we get

$$\mathbb{E}[\langle \eta^{(k)}, \pi(\eta^{(k)})\rangle|\pi] = \sum_{i\,:\,\pi(i)=i} \mathbb{E}[(\eta_i^{(k)})^2] = N_\pi,$$

where we denote by N_π the number of fixed points of π.

Combining the last two displays with (3.28), we get

$$\mathbb{P}\left(\left|\eta^\top A\eta\right| \leq \left|\sum_{k\geq 1} h_k\, h'_k\right| N_\pi + cK^2\|h\|\,\|h'\|\,\max\{\sqrt{n}\,t, t\}\ \mid\ \pi\right) \geq 1 - e^{-t}, \qquad (3.29)$$

for some numerical constant $c > 0$.

We study now the tail behavior of N_π. For any integer $k \in [1, n]$, we have

$$\mathbb{P}\left(N_\pi = k\right) = \frac{1}{k!}\sum_{l=0}^{n-k}\frac{(-1)^l}{l!}.$$

Elementary computations then give

$$\mathbb{P}\left(N_\pi > \sqrt{n}\right) \leq \sum_{k\geq\lceil\sqrt{n}\,\rceil}\frac{1}{k!} \leq \sum_{k\geq\lceil\sqrt{n}\,\rceil}\left(\frac{2}{k}\right)^{\frac{k}{2}} = \sum_{k\geq\lceil\sqrt{n}\,\rceil} e^{-\frac{k}{2}\log\left(\frac{k}{2}\right)} \leq Ce^{-c\sqrt{n}\log(n/\sqrt{2})}.$$

Assuming that $n \geq 2$ and up to a rescaling of the constants, we get

$$\mathbb{P}\left(N_\pi > c'\sqrt{n}\right) \leq e^{-\sqrt{n}}, \qquad (3.30)$$

for some numerical constant $c' > 0$.

Set

$$u = \|h\|\,\|h'\|\left(c'\sqrt{n} + cK^2\max\{\sqrt{n}\,t, t\}\right).$$

We deduce from (3.29)-(3.30) that

$$\begin{aligned}\mathbb{P}\left(\left|\eta^\top A\eta\right| > u\right) &= \mathbb{P}\left(\left\{\left|\eta^\top A\eta\right| > u\right\}\cap\left\{N_\pi \leq c'\sqrt{n}\right\}\right) + \mathbb{P}\left(\left\{\left|\eta^\top A\eta\right| > u\right\}\cap\left\{N_\pi > c'\sqrt{n}\right\}\right)\\ &\leq \mathbb{P}\left(\left\{\left|\eta^\top A\eta\right| > u\right\}\ \mid\ N_\pi \leq c'\sqrt{n}\right) + \mathbb{P}\left(N_\pi > c'\sqrt{n}\right)\\ &\leq e^{-t} + e^{-\sqrt{n}}.\end{aligned}$$

We derive now a concentration bound on $\|Z_h\|$. Note that $\mathbb{E}[Z_h^2] = \|h\|^2 n$. We apply a similar argument to that used to derive (3.28). The only difference is that we replace A in (3.26) by $A = (h \otimes h) \otimes I_n$. Thus we get for any $t > 0$ with probability at least $1 - e^{-t}$,

$$\left| \|Z_h\|^2 - \|h\|^2 n \right| \leq CK^2 \|h\|^2 \max\{\sqrt{n\,t}, t\}, \tag{3.31}$$

where $C > 0$ is an absolute constant . Take $t = c_K n$ with $c_K > 0$ is such that $CK^2\|h\|^2 \max\{\sqrt{n\,t}, t\} \leq \|h\|^2 n/2$.

Combining the last two displays, we get up to a rescaling of the constant, with probability at least $1 - e^{-t} - e^{-\sqrt{n}}$,

$$\frac{|\langle Z_h, \pi(Z_{h'}) \rangle|}{\|Z_h\| \|Z_{h'}\|} \lesssim \frac{1}{\sqrt{n}} \vee K^2 \sqrt{\frac{t}{n}} \vee \frac{t}{n}.$$

□

3.5.5 Lemma 2

Lemma 2. *Under the assumption considered in Theorem 1 and in the intermediate SNR regime $r\sigma^2 \ll \|\boldsymbol{x}\boldsymbol{\beta}^*\|_2^2 \ll n\sigma^2$, we have w.h.p.*

$$\|P_{\boldsymbol{x}}(\mathbf{Y})\|_2^2 = \|\boldsymbol{x}\boldsymbol{\beta}^*\|_2^2 \left(1 + O\left(\sqrt{\frac{r\sigma^2}{\|\boldsymbol{x}\boldsymbol{\beta}^*\|_2^2}}\right)\right), \tag{3.32}$$

$$\|P_{\boldsymbol{x}}(\boldsymbol{Y}_{perm})\|_2^2 = r\sigma^2 \left(1 + O\left(\sqrt{\frac{\|\boldsymbol{x}\boldsymbol{\beta}^*\|_2^2}{n\sigma^2}}\right)\right). \tag{3.33}$$

Therefore,

$$\frac{\|P_{\boldsymbol{x}}(\boldsymbol{Y}_{perm})\|_2^2}{\|P_{\boldsymbol{x}}(\mathbf{Y})\|_2^2} = \frac{\sigma^2 r}{\|\boldsymbol{x}\boldsymbol{\beta}^*\|_2^2}(1 + o(1)). \tag{3.34}$$

Proof of 3.32. Since $\boldsymbol{\xi}$ is subGaussian, we have $\|P_{\mathbf{x}}(\boldsymbol{\xi})\|_2^2 = r\sigma^2 + O\left(\sqrt{r\sigma^2}\right)$ *w.h.p.* [125]. Thus, under the SNR condition ($\|\mathbf{x}\boldsymbol{\beta}^*\|_2^2 \gg r\sigma^2$), we get

$$\frac{\|P_{\mathbf{x}}(\boldsymbol{\xi})\|_2^2}{\|\mathbf{x}\boldsymbol{\beta}^*\|_2^2} \lesssim \frac{r\sigma^2}{\|\mathbf{x}\boldsymbol{\beta}^*\|_2^2} \ll 1, \quad w.h.p.$$

Consequently and as $\langle \mathbf{x}\boldsymbol{\beta}^*, \boldsymbol{\xi}\rangle$ is subGaussian, it comes

$$\begin{aligned}
\|P_{\mathbf{x}}(\mathbf{Y})\|_2^2 &= \|\mathbf{x}\boldsymbol{\beta}^*\|_2^2 + 2\langle \mathbf{x}\boldsymbol{\beta}^*, \boldsymbol{\xi}\rangle + \|P_{\mathbf{x}}(\boldsymbol{\xi})\|_2^2 \\
&= \|\mathbf{x}\boldsymbol{\beta}^*\|_2^2 \left(1 + 2\frac{\langle \mathbf{x}\boldsymbol{\beta}^*, \boldsymbol{\xi}\rangle}{\|\mathbf{x}\boldsymbol{\beta}^*\|_2^2} + \frac{\|P_{\mathbf{x}}(\boldsymbol{\xi})\|_2^2}{\|\mathbf{x}\boldsymbol{\beta}^*\|_2^2}\right) \\
&= \|\mathbf{x}\boldsymbol{\beta}^*\|_2^2 \left(1 + O\left(\sqrt{\frac{r\sigma^2}{\|\mathbf{x}\boldsymbol{\beta}^*\|_2^2}}\right)\right) \quad \textit{w.h.p.}. \qquad (3.35)
\end{aligned}$$

Proof of 3.33. Next we recall that $\mathbf{Y}_{\mathtt{perm}} = \pi(\mathbf{Y}) = \pi(\mathbf{x}\boldsymbol{\beta}^*) + \pi(\boldsymbol{\xi})$ for some π drawn uniformly at random in the set of permutation of n elements. Then

$$\|P_{\mathbf{x}}(\mathbf{Y}_{\mathtt{perm}})\|_2^2 = \|P_{\mathbf{x}}(\pi(\mathbf{Y}))\|_2^2 = \|P_{\mathbf{x}}(\pi(\mathbf{x}\boldsymbol{\beta}^*))\|_2^2 + 2\langle P_{\mathbf{x}}(\pi(\mathbf{x}\boldsymbol{\beta}^*)), P_{\mathbf{x}}(\pi(\boldsymbol{\xi}))\rangle + \|P_{\mathbf{x}}(\pi(\boldsymbol{\xi}))\|_2^2.$$

- Exploiting again subGaussianity of $\boldsymbol{\xi}$, we get $\|P_{\mathbf{x}}(\pi(\boldsymbol{\xi}))\|_2^2 = r\sigma^2 + O(\sqrt{r\sigma^2})$ *w.h.p.*.
- We recall that the n-dimensional vector $\mathbf{x}\boldsymbol{\beta}^*$ lives in a r-dimensional subspace of $\mathbb{R}^n$ with $r \ll n$. Using Property 1, we get the following fact: Randomly permuting its components creates a new vector $\pi(\mathbf{x}\boldsymbol{\beta}^*)$ which is almost orthogonal to $\mathbf{x}\boldsymbol{\beta}^*$:

$$\frac{\langle \pi(\mathbf{x}\boldsymbol{\beta}^*), \mathbf{x}\boldsymbol{\beta}^*\rangle}{\|\mathbf{x}\boldsymbol{\beta}^*\|_2^2} \lesssim \frac{r}{n} \ll 1, \quad \text{and} \quad \|P_{\mathbf{x}}(\pi(\mathbf{x}\boldsymbol{\beta}^*))\|_2^2 \lesssim \frac{r}{n}\|\mathbf{x}\boldsymbol{\beta}^*\|_2^2, \quad \textit{w.h.p.}.$$

- In the non trivial SNR regime $r\sigma^2 \ll \|\mathbf{x}\boldsymbol{\beta}^*\|_2^2 \ll n\sigma^2$ where Ridge regularization is useful, the dominating term in the previous display is $\|P_{\mathbf{x}}(\pi(\boldsymbol{\xi}))\|_2^2$.

Proof of 3.34. Consequently, we get

$$\|P_{\mathbf{x}}(\pi(\mathbf{Y}))\|_2^2 = r\sigma^2 \left(1 + O\left(\sqrt{\frac{\|\mathbf{x}\boldsymbol{\beta}^*\|_2^2}{n\sigma^2}}\right)\right) \quad \textit{w.h.p.}. \qquad \square \qquad (3.36)$$

Chapter 4
AdaCap: Adaptive Capacity control for Feed-Forward Neural Networks

Improvement makes straight roads, but the crooked roads without Improvement are roads of Genius.

William Blake, Proverbs of Hell

Chapter Summary

The *capacity* of a ML model refers to the range of functions this model can approximate. It impacts both the complexity of the patterns a model can learn but also *memorization*, the ability of a model to fit arbitrary labels. We propose **Adaptive Capacity** (AdaCap), a training scheme for Feed-Forward Neural Networks (FFNN). AdaCap optimizes the *capacity* of FFNN so it can capture the high-level abstract representations underlying the problem at hand without memorizing the training dataset. AdaCap is the combination of two novel ingredients, the **Muddling labels for Regularization** (MLR) loss and the **Tikhonov operator** training scheme. The MLR loss leverages randomly generated labels to quantify the propensity of a model to memorize. We prove that the MLR loss is an accurate in-sample estimator for out-of-sample generalization performance and that it can be used to perform Hyper-Parameter Optimization provided a Signal-to-Noise Ratio condition is met. The Tikhonov operator training scheme modulates the *capacity* of a FFNN in an adaptive, differentiable and data-dependent manner. We assess the effectiveness of AdaCap in a setting where DNN are typically prone to memorization, small tabular datasets, and benchmark its performance against popular machine learning methods.

4.1 Introduction

Generalization is a central problem in Deep Learning (*Deep Learning*). It is strongly connected to the notion of *capacity* of a model, that is the range of functions a model can approximate. It impacts both the complexity of the patterns a model can learn but also *memorization*, the ability of a model to fit arbitrary labels [73]. Because of their high capacity, overparametrized Deep Neural Networks (DNN) can memorize the entire train set to the detriment of generalization. Common techniques like Dropout (DO) [92, 193], Early Stopping [218], Data Augmentation [182] or Weight Decay [83, 119, 25] used during training can reduce the capacity of a DNN and sometimes delay memorization but cannot prevent it [7].

We propose AdaCap, a new training technique for Feed-Forward Neural Networks (FFNN) that optimizes the *capacity* of FFNN during training so that it can capture the high-level abstract representations underlying the problem at hand and mitigate memorization of the train set. AdaCap relies on two novel ingredients, the

`Tikhonov` **operator** and the **Muddling labels Regularization** (`MLR`) loss.

The `Tikhonov` **operator** provides a differentiable data-dependent quantification of the capacity of a `FFNN` through the application of this operator on the output of the last hidden layer. The `Tikhonov` operator modulates the capacity of the `FFNN` via the additional `Tikhonov` parameter that can be trained concomitantly with the hidden layers weights by Gradient Descent (`GD`). This operator works in a fundamentally different way from other existing training techniques like Weight Decay (See Section 4.3 and Fig. 4.1).

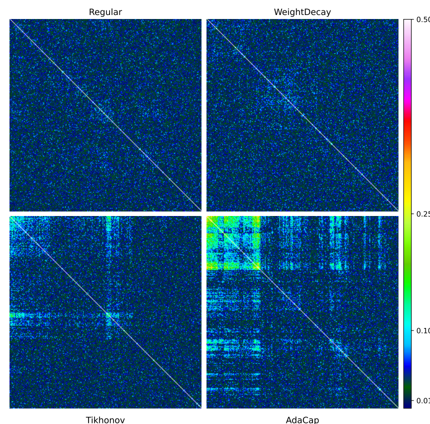


Figure 4.1: We trained a `MLP` on the `Boston` dataset with either usual training (with/without Weight Decay) or `AdaCap`, using identical architectures and parameters in both cases. We plot the clustered correlation matrices of last hidden layer weights (in absolute value). **Upper left:** usual loss and no regularization. **Upper right:** usual loss and Weight Decay only. **Bottom left:** usual loss and `Tikhonov` scheme alone (no Weight Decay). **Bottom right:** `AdaCap` alone. Contrarily to Weight Decay, `AdaCap` produced a model with highly structured hidden layer weights even with a plain `MLP` architecture, indicating its learning behavior is very different from the standard one.

The problem is then the tuning of the `Tikhonov` parameter that modulates capacity as it directly impacts the generalization performance of the trained `FFNN`. This motivated the introduction of the `MLR` loss which performs capacity tuning without using a hold-out validation set. The `MLR` loss is based on a novel way to exploit random labels.

Random labels have been used in [221, 7] as a diagnostic tool to understand how overparametrized `DNN` can generalize surprisingly well despite their capacity to memorise the train set. This benign overfitting phenonemon is attributed in part

to the implicit regularization effect of the optimizer schemes used during training [77, 189].

Understanding that the training of DNN is extremely susceptible to corrupted labels, numerous methods have been proposed to identify the noisy labels or to reduce their impact on generalization. These approaches include loss correction techniques [152], reweighing samples [104], training two networks in parallel [81]. See [39, 85] for an extended survey.

We propose a different approach. We do not attempt to address the noise and corruptions already present in the original labels. Instead, we **purposely generate purely corrupted labels during training** as a tool to reduce the propensity of the DNN to memorize label noise during gradient descent. The underlying intuition is that we no longer see generalization as the ability of a model to perform well on unseen data, but rather as the ability to avoid finding pattern where none exists. Concretely, we propose the **Muddling labels Regularization** loss which uses randomly permuted labels to quantify the overfitting ability of a model on a given dataset. In Section 4.2, we provide a brief presentation of the MLR loss, which is introduced in Chapter 3.

This property motivates using the MLR loss rather than the usual losses during training to perform *adaptive* control of the *capacity* of DNN. This can improve generalization (Fig. 4.2) not only in the presence of label corruption but also in other settings prone to overfitting - *e.g*. Tabular Data [24, 75, 183], Few-Shot Learning (Fig. 4.3), a task introduced in [63, 58]. See [211] for a recent survey.

Our novel training method AdaCap works as follows. Before training: a) generate a new set of completely uninformative labels by *muddling* original labels through random permutations; then, at each GD iteration: b) apply the Tikhonov operator to the output of the last hidden layer; c) quantify the ability of the DNN's output layer to fit true labels rather than permuted labels via the new (MLR) loss; d) back-propagate the MLR objective through the network.

AdaCap is a gradient-based, global, data-dependent method which trains the weights and adjusts the capacity of the FFNN simultaneously during the training phase without using a hold-out *validation* set. AdaCap is designed to work on most FFNN architectures and is compatible with the usual training techniques like Gradient Optimizers [113], Learning Rate Schedulers [189], Dropout [193], Batch-Norm [101], Weight Decay [119, 25], etc.

DNN have not demonstrated yet the same level of success on Tabular Data (TD) as on images [118], audio [92] and text [53], which makes it an interesting

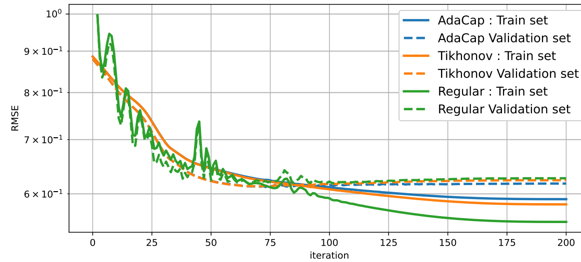


Figure 4.2: RMSE across iterations for train and valid sets when training AdaCap, Tikhonov and regular DNN on Abalone TD. We can see the improvement in terms of generalization and memorization when adding Tikhonov and then MLR: The train RMSE converges to a higher plateau but validation RMSE keeps improving further. Tikhonov smooths the learning dynamic, removes the oscillations. Adding the MLR loss makes no change to the learning dynamic on the first iterations but impact the last iterations and delays memorization even more.

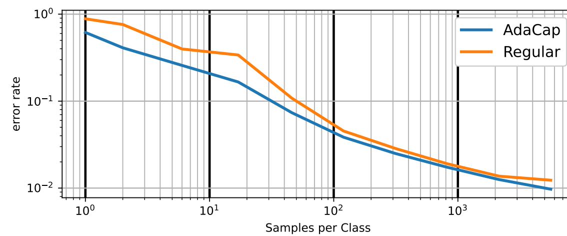


Figure 4.3: Few-shot learning experiment on MNIST [51]. When training a simple ConvNet, the generalization performance of the obtained models *w.r.t.* the number of samples per class is uniformly better over the whole range of samples per class when using AdaCap, especially in the low sample per class regime.

frontier for DNN architectures. TDoften present challenges for traditional deep neural networks, including the following: Training data can be of very low quality, it can contain outliers, missing or incoherent values. Preprocessing data, especially categorical features, can be essential but very complex. The spatial dependencies between features can be absent, undetermined or highly irregular. Conversely, a small change on a single feature can have a massive impact on the expected target value, a phenomenon for which decision trees are well-suited. Due to the popularity of tree-based ensemble methods (CatBoost [160], XGBoost [40], **R.F.** [13, 30]), there has been a strong emphasis on the preprocessing of categorical features which was an historical limitation of *Deep Learning*. Notable contributions include NODE [156] and TabNet [6]. NODE (Neural Oblivious Decision Ensembles) are tree-like architectures which can be trained end-to-end via backpropagation. TabNet leverages Attention Mechanisms [12] to pretrain DNN with feature encoding. The comparison between simple DNN, NODE, TabNet, **R.F.** and GBDT on TD was made concomitantly by [107], [75] and [183]. Their benchmarks are more oriented towards an AutoML approach than ours, as they all use heavy HPO, and report training times in minutes/hours, even for some small and medium size datasets. See Appendix 4.6.3 for a more detailed discussion. As claimed by [183], their results (like ours) indicate that DNN are not (yet?) the alpha and the omega of TD. [107] also introduces an HPO strategy called the regularization cocktail.

Regarding the new techniques for DNN on TD, we mention here a few relevant to our work which we included in our benchmark. See [24] and the references therein for a more exhaustive list. [114] introduced Self-Normalizing Networks (SNN) to train deeper FFNN models, leveraging the SeLU activation function. [75] proposed new architecture schemes: ResBlock, Gated Linear Units GLU, and FeatureTokenizer-Transformers, which are adaptation for TD of ResNet [90], Gated convolutional networks [49] and Transformers [207].

We illustrate the potential of AdaCap on a benchmark of 44 tabular datasets from diverse domains of application, including 26 regression tasks, 18 classification tasks, against a large set of popular methods GBDT [31, 67, 68, 40, 110, 160]; Decision Trees and **R.F.** [29, 13, 30, 70, 115], Kernels [36], MLP [93], GLM [46, 95, 199, 227], MARS [66]).

For *Deep Learning* architectures, we combined and compared AdaCap with MLP, GLU, ResBlock, SNN and CNN. We left out recent methods designed to tackle categorical features (TabNet, NODE, FT-Transformers) as it is not the focus of this benchmark and of our proposed method. Our experimental study reveals that using AdaCap to train FFNN leads to an improvement of the generalization

performance on regression tabular datasets especially those with high Signal-to-Noise Ratio (SNR), the datasets where it is possible but not trivial to obtain a very small test RMSE. AdaCap works best in combination with other schemes and architectures like SNN, GLU or ResBlock. Introducing AdaCap to the list of available DNN schemes allows neural networks to gain ground against the GBDT family. We provide an implementation of AdaCap and the code to replicate all experimental results at https://github.com/benjaminriu/Thesis.

4.2 The MLR loss

The MLR approach is introduced in full detail in Chapter 3. In that chapter, the MLR criterion is studied extensively in a linear regression. We present here the loss MLR, which is the extension of the MLR criterion to a deep learning setting.

Setting. Let $\mathcal{D}_{train} = (\mathbf{X}, \mathbf{Y}) = \{(\mathbf{x}_i, y_i)\}_{i=1}^n$ be the *train*-set with $\mathbf{x}_i \in \mathbb{R}^p$ where p denotes the number of features and $y_i \in \mathcal{Y}$ where $\mathcal{Y} = \mathbb{R}$ for regression and $\mathcal{Y}$ is a finite set for classification. We optimise the objective $\texttt{Loss}(\texttt{act}_{\texttt{out}}(\boldsymbol{f}_\theta(\mathbf{X})), \mathbf{Y})$ where $\boldsymbol{f}_\theta(\mathbf{X})$ is the output of the last hidden layer, Loss is the loss function (MSE for regression and CE for classification) and $\texttt{act}_{\texttt{out}}$ is the activation function ($\mathbb{I}$ for regression, Sigmoid for binary classification and logsoftmax for multiclass).

Random permutations. We build a randomized data set by applying random permutations on the n components of the label vector $\mathbf{Y}$. This randomization scheme presents the advantage of creating an artificial train set $(\mathbf{X}, \mathbf{Y}_{\texttt{perm}})$ with marginal distributions of features and labels identical to those in the initial train set but where the connection between features and labels has been removed[1]. This means that there is no generalizing pattern to learn from the artificial dataset $(\mathbf{X}, \mathbf{Y}_{\texttt{perm}})$. We replace the initial loss Loss by

$$\begin{aligned} \texttt{MLR}(\theta) := &\ \texttt{Loss}\left(\mathbf{Y}, \texttt{act}_{\texttt{out}}(\boldsymbol{f}_\theta(\mathbf{X}))\right) \\ &- \texttt{Loss}\left(\mathbf{Y}_{\texttt{perm}}, \texttt{act}_{\texttt{out}}(\boldsymbol{f}_\theta(\mathbf{X}))\right). \end{aligned} \tag{4.1}$$

The second term on the right-hand side of (4.1) is used to quantify memorization of output layer $\boldsymbol{f}_\theta$. Indeed, since there is no meaningful pattern linking $\mathbf{X}$ to $\mathbf{Y}_{\texttt{perm}}$, any $\boldsymbol{f}_\theta$ which fits $(\mathbf{X}, \mathbf{Y}_{\texttt{perm}})$ well achieves it via memorization only. We want to rule out such models. By minimizing the MLR loss, we hope to retain only the

[1] The expected number of fixed points of a permutation drawn uniformly at random is equal to 1.

generalizing patterns.

4.3 The AdaCap method to train DNN

The Tikhonov operator scheme. Our goal now is to adapt the MLR methodology to a deep learning setting. To do so, we first need to introduce a new tool, the **Tikhonov** operator. Combining this technique with the MLR loss corresponds to AdaCap (for Adaptative Capacity Control), a novel method to train deep neural networks.

Consider a standard feed-forward DNN architecture with L layers where the output layer is a densely connected perceptron of input width J. As such, the output layer activation $\mathbf{A}^L \in \mathbb{R}^n$ can be expressed as a function of the last hidden layer activation $\mathbf{A}^{L-1} \in \mathbb{R}^{n,J}$ and the weights $\mathbf{W}^L \in \mathbb{R}^J$ and biases $b^L \in \mathbb{R}$: $\mathbf{A}^L = \mathtt{act}_{\mathtt{out}}(\mathbf{A}^L\mathbf{W}^L + b^L)$ where $\mathtt{act}_{\mathtt{out}}$ is the output activation function (eg. the identity function in case of univariate regression). For the Tikhonov operator to be applicable, no other conditions on the architecture are needed.

The last hidden layer activation $\mathbf{A}^{L-1}$ is a function (for example a ConvNet of output size J) of $\mathbf{X}$, the input data and θ, the hidden layers weights. By abuse of notation, denote this function $\mathbf{A}^{L-1}(\theta,\cdot) : \mathbb{R}^{n\times p} \to \mathbb{R}^{n\times J}$.

Let $\lambda \in \mathbb{R}_+^*$ be the Tikhonov parameter and define

$$\mathbf{P}(\lambda,\theta,\mathbf{X}) := \left[\left(\mathbf{A}^{L-1}(\theta,\mathbf{X})\right)^\top \mathbf{A}^{L-1}(\theta,\mathbf{X}) + \lambda\mathbb{I}_J\right]^{-1} \left(\mathbf{A}^{L-1}(\theta,\mathbf{X})\right)^\top \tag{4.2}$$

where $\mathbb{I}_J$ is the identity matrix. The Tikhonov operator is

$$\mathbf{H}(\lambda,\theta,\mathbf{X}) := \mathbf{A}^{L-1}(\theta,\mathbf{X})\mathbf{P}(\lambda,\theta,\mathbf{X}). \tag{4.3}$$

During training, the Tikhonov operator scheme outputs the following prediction for target vector[2] $\mathbf{Y}$:

$$\boldsymbol{f}_{\lambda,\theta,\mathbf{Y}}(\mathbf{X}) = \mathbf{H}(\lambda,\theta,\mathbf{X})\mathbf{Y} = \mathbf{A}^{L-1}(\theta,\mathbf{X})\mathbf{P}(\lambda,\theta,\mathbf{X})\mathbf{Y}, \tag{4.4}$$

This corresponds to a DNN of which the last layer is densely connected, meaning $\boldsymbol{f}_{\lambda,\theta,\mathbf{Y}}(\mathbf{X}) = \mathtt{act}_{\mathtt{out}}(\mathbf{A}^L\mathbf{W}^L + b^L)$, where $\mathtt{act}_{\mathtt{out}}$ is the identity function, $b^L = 0$ and $\mathbf{W}^L = \mathbf{P}(\lambda,\theta,\mathbf{X})\mathbf{Y}$. Here the weights on the output layer are not trainable

[2] In multiclass setting, replace $\mathbf{Y}$ by its one-hot encoding.

parameters but are instead completely dependant on λ, θ, $\mathbf{X}$ and $\mathbf{Y}$. By abuse of notation, denote $\mathbf{W}^L(\lambda, \theta, \bullet, \bullet) : \mathbb{R}^{n\times p} \times \mathbb{R}^n \to \mathbb{R}^J$ the corresponding function. Note that $(\lambda, \theta, \mathbf{X}, \mathbf{Y})$ may change at each iteration during training/GD.

To train this DNN, we run a Gradient Descent Optimization scheme over parameters (λ, θ)

$$(\widehat{\lambda}, \widehat{\theta}) = \underset{\lambda>0,\,\theta}{\arg\min}\ \mathtt{Loss}\left(\mathbf{Y}, \mathtt{act}_{\mathtt{out}}\left(\boldsymbol{f}_{\lambda,\theta,\mathbf{Y}}(\mathbf{X})\right)\right). \tag{4.5}$$

Eventually, at test time, we freeze $\widehat{\mathbf{W}^L} = \mathbf{W}^L(\widehat{\lambda}, \widehat{\theta}, \mathbf{X}, \mathbf{Y})$, and obtain our final predictor

$$\boldsymbol{f}_{\widehat{\lambda},\widehat{\theta}}(\bullet) = \mathtt{act}_{\mathtt{out}}\left(\mathbf{A}^{L-1}(\widehat{\theta}, \bullet)\mathbf{P}(\widehat{\lambda}, \widehat{\theta}, \mathbf{X})\mathbf{Y}\right), \tag{4.6}$$

where $\mathtt{act}_{\mathtt{out}}$ is the original activation function applied to the output layer. Here, $\widehat{\mathbf{W}^L} = \mathbf{P}(\widehat{\lambda}, \widehat{\theta}, \mathbf{X})\mathbf{Y}$ are the weights of the output layer set once and for all (using the minibatch $(\mathbf{X}, \mathbf{Y})$ associated with $(\widehat{\lambda}, \widehat{\theta})$ in case of batch-learning). Therefore, we recover the architecture of a standard DNN where the output of the hidden layers $\mathbf{A}^{L-1}(\widehat{\theta}, \bullet)$ is multiplied by the weights of the output layer.

The Tikhonov operator scheme works in a fundamentally different way from Weight Decay. To highlight their differences, let's consider a standard shallow NN and activation function σ. Consider a train sample $\mathcal{D}_{train} = (\mathbf{x}_i, y_i)_{i=1,\cdots,n}$ with $\mathbf{x}_i \in \mathbb{R}^p$ and $y_i \in \mathbb{R}$. We set $\mathbf{A}^0 = \mathbf{X} \in \mathbb{R}^{n\times p}$, then the last hidden layer $\mathbf{A}^1 = \sigma(\mathbf{X}\mathbf{W}^1)$ and output $\mathbf{A}^2 = \mathbf{A}^1\mathbf{W}^2$. Classically, the weights of the NN is $\theta = \{\mathbf{W}^1, \mathbf{W}^2\}$. During backpropagation, we compute the derivative of the train loss for weights $\{\mathbf{W}^l\}_{l=1,2}$:

$$\nabla_{\mathbf{W}^l}\partial\|\mathbf{Y} - \mathbf{A}^2\|_2^2 = \nabla_{\mathbf{A}^2}\partial\|\mathbf{Y} - \mathbf{A}^2\|_2^2 \times \underbrace{\nabla_{\mathbf{W}^l}\mathbf{A}^2}_{\text{common term}}$$

If we add weight decay of level λ on the output layer $\mathbf{A}^2$, another term appears in the weight updates:

$$\lambda\nabla_{\mathbf{W}^2}\|\mathbf{W}^2\|_2^2.$$

Meanwhile, in our approach the standard shallow NN becomes

$$\begin{aligned}
\mathbf{A}^1 &= \sigma(\mathbf{X}\mathbf{W}^1)\\
\mathbf{A}^2 &= \mathbf{A}^1\boldsymbol{\beta}^R_\lambda(\mathbf{Y}, \mathbf{A}^1) = \mathbf{A}^1((\mathbf{A}^1)^\top\mathbf{A}^1 + \lambda\mathbb{I})^{-1}(\mathbf{A}^1)^\top\mathbf{Y}.
\end{aligned}$$

In our approach, $\theta = \{\mathbf{W}^1, \lambda\}$ and the derivative of $\mathbf{A}^2$ *w.r.t.* $\mathbf{W}^1$ becomes

$$\nabla_{\mathbf{w}^1}\mathbf{A}^2 = \underbrace{\nabla_{\mathbf{w}^1}\mathbf{A}^1}_{\text{common term}} \times \boldsymbol{\beta}^R_\lambda(\mathbf{Y},\mathbf{A}^1) + \mathbf{A}^1 \underbrace{\nabla_{\mathbf{w}^1}\boldsymbol{\beta}^R_\lambda(\mathbf{Y},\mathbf{A}^1)}_{\textit{novel term}}$$

with

$$\begin{aligned}\nabla_{\mathbf{w}^1}\boldsymbol{\beta}^R_\lambda(\mathbf{Y},\mathbf{A}^1) &= \nabla_{\mathbf{w}^1}\left[((\mathbf{A}^1)^\top\mathbf{A}^1 + \lambda\mathbb{I})^{-1}(\mathbf{A}^1)^\top\mathbf{Y}\right] \\ &= \nabla_{\mathbf{w}^1}\left[(\sigma(\mathbf{X}\mathbf{W}^1))^\top\sigma(\mathbf{X}\mathbf{W}^1) + \lambda\mathbb{I})^{-1}(\sigma(\mathbf{X}\mathbf{W}^1))^\top\mathbf{Y}\right]\end{aligned}$$

This novel term corresponds to the exploration of another path in the derivation graph of backpropagation. This means that the impact of `Tikhonov` will be passed-through to the other layers. As such, when we apply the `Tikhonov` operator to the output of the last hidden layer and then use backpropagation to train the `DNN`, we are indirectly carrying over its *capacity* control effect to the hidden layers of the `DNN`. Moreover, with `Tikhonov` this is done jointly on all layers, as the gradient for one layer is dependant on the value of other layers weights, contrary to Weight Decay where the regularization applied on one weight only depends on its current value. In other words, we are performing *inter-layers* regularization whereas Weight Decay performs *intra-layer* regularization. To illustrate this, we trained a `DNN` using Weight Decay on the one-hand and Tikhonov operator on the other hand while all the other training choices were the same between the two training schemes (same loss `Loss`, same architecture size, same initialization, same learning rate, etc.). Fig. 4.1 shows that the `Tikhonov` scheme works differently from other L_2 regularization schemes like Weight Decay. Indeed, Fig. 4.2 reveals that the `Tikhonov` scheme completely changes the learning dynamic during `GD`.

Training with `MLR` loss and the `Tikhonov` scheme. We quantify the *capacity* of our model to memorize labels $\mathbf{Y}$ by $\texttt{Loss}\left(\mathbf{Y}, \texttt{act}_{\texttt{out}}\left(\boldsymbol{f}_{\lambda,\theta,\mathbf{Y}}(\mathbf{X})\right)\right)$ w.r.t. to labels $\mathbf{Y}$ where the `Tikhonov` parameter λ modulates the level the *capacity* of this model. However, we are not so much interested in adapting the capacity to the train set $(\mathbf{X},\mathbf{Y})$ but rather to the generalization performance on the test set. This is why we replace `Loss` by `MLR` in (4.5). Since `MLR` is a more accurate in-sample estimate of the generalization error than the usual train loss, we expect `MLR` to provide better tuning of λ and thus some further gain on the generalization performance.
Combining (4.1) and (4.4), we obtain the following train loss of our method.

$$\begin{aligned}\texttt{MLR}(\lambda,\theta) := \texttt{Loss}\Big(&\mathbf{Y}, \texttt{act}_{\texttt{out}}(\mathbf{H}(\lambda,\theta,\mathbf{X})\mathbf{Y})\Big) \\ &- \texttt{Loss}\Big(\mathbf{Y}_{\texttt{perm}}, \texttt{act}_{\texttt{out}}(\mathbf{H}(\lambda,\theta,\mathbf{X})\mathbf{Y}_{\texttt{perm}})\Big)\end{aligned} \tag{4.7}$$

To train this model, we run a Gradient Descent Optimization scheme over parameters (λ, θ):

$$(\widehat{\lambda}, \widehat{\theta}) = \text{argmin}_{\lambda,\theta|\lambda>0} \texttt{MLR}(\lambda, \theta). \tag{4.8}$$

The AdaCap predictor is defined again by (4.4) but with weights obtained in (4.8) and corresponds to the architecture of a standard DNN. Indeed, at test time, we freeze $\mathbf{W}(\widehat{\lambda}, \widehat{\theta}, \mathbf{X})\mathbf{Y}$ which becomes the weights of the output layer. Once the DNN is trained, the corrupted labels $\mathbf{Y}_{\texttt{perm}}$ and the Tikhonov parameter $\widehat{\lambda}$ have no further use and are thus discarded. If using Batch-Learning, we use the minibatch $(\mathbf{X}, \mathbf{Y})$ corresponding to $(\widehat{\theta}, \widehat{\lambda})$. In any case, the entire training set can also be discarded once the output layer is frozen.

Comments.

• Tikhonov is absolutely needed to use MLR on DNN in a differentiable fashion because FFNN have such a high capacity to memorize labels on the hidden layers that the SNR between output layer and target is too high for MLR to be applicable without controlling capacity via the Tikhonov operator. Controlling network capacity via HPO over regularization techniques would produce a standard bi-level optimization problem.

• The random labels are generated before training and are not updated or changed thereafter. Note that in practice, the random seed used to generate the label permutation has virtually no impact as shown in Table 4.7.

• Note that λ is not an hyperparameter in AdaCap . It is trained alongside θ by GD. The initial value $\lambda_{\texttt{init}}$ is chosen with a simple heuristic rule. For initial weights θ, we pick the value which maximizes sensitivity of the MLR loss *w.r.t.* variations of λ (See (4.9) in Appendix 4.6.2).

• Both terms of the MLR loss depend on θ through the quantity $\mathbf{H}(\lambda, \theta, \mathbf{X})$, meaning we compute only one derivation graph *w.r.t.* $\mathbf{H}(\lambda, \theta, \mathbf{X})$.

• When $\texttt{act}_{\texttt{out}} = \mathbb{I}$, the architecture remains the same at train time and test time. Otherwise, minor re-scaling can improve results. For example, in binary classification with balanced classes, the mean of the train set vector of labels $\mathbf{Y}$ is 0.5, meaning we expect predictions to be centered around 0.5 in train mode. Setting $b^L = -0.5$ at test time slides the cutpoint to 0, which corresponds to the probability 0.5 once the Sigmoid is applied. In that case this is equivalent to centering the vector of labels $\mathbf{Y}$ at train time. See code implementation for more details.

• When using the **Tikhonov operator** during training, we replace a matrix multiplication by a matrix inversion. This operation is differentiable and inexpensive as parallelization schemes provide linear complexity on GPU [180][3]. Time computation comparisons are provided in Table 4.2. The overcost depends on the dataset but remains comparable to applying Dropout (DO) and Batch Norm (BN) on each hidden layers for DNN with depth 3+.

• For large datasets, AdaCap can be combined with Batch-Learning. Table 4.6 in appendix reveals that AdaCap works best with large batch-size, but handles very small batches and seeing fewer times each sample much better than regular DNN.

4.4 Experiments

Our goal in this section is to tabulate the impact of AdaCap on simple FFNN architectures, on a tabular data benchmark, an ablation and parameter dependence study, and also a toy few shot learning experiment (Fig. 4.3).

Note that in the main text, we report only the key results. In supplementary, we provide a detailed description of the benchmarked FFNN architectures and corresponding hyperparameters choices; a dependence study of impact of batchsize, DO&BN, and random seed; the exhaustive results for the tabular benchmark; the implementation choices for compared methods; datasets used with sources and characteristics; datasets preprocessing protocol; hardware implementation. Code is provided on a github repository.

4.4.1 Implementation details

Creating a pertinent benchmark for TD is still an ongoing process for ML research community. Because researchers compute budget is limited, arbitrages have to be made between number of datasets, number of methods evaluated, intensity of HPO, dataset size, number of train-test splits. We tried to cover a broad set of usecases [148] where improving DNN performance compared to other existing methods is relevant, leaving out hours-long training processes relying on HPO to get the optimal performance for each benchmarked method. We detail below how this choice affected the way we designed our benchmark.
FFNN Architectures. For binary classification (BinClf), multiclass classification (MultiClf) and regression (Reg), the output activation/training loss are Sigmoid/BCE, logsoftmax/CE and $\mathbb{I}$/RMSE respectively. We also implemented the

[3] This article states that the time complexity of matrix inversion scales as J as long as J^2 threads can be supported by the GPU where J is the size of the matrix.

corresponding MLR losses. In all cases, we used the Adam (Adam) [113] optimizer and the One Cycle Learning Rate Scheduler scheme [185]. Early-Stopping is performed using a validation set of size $\min(n * 0.2, 2048)$. Unless mentioned otherwise, Batch-Learning is performed with batch size $b_s = \min(n * 0.8, 2048)$ and the maximum number of iteration does not depend on the number of epochs and batches per epoch, to cap the training time, in accordance with our benchmark philosophy. We initialized layer weights with Kaiming [89]. Then, for AdaCap, the Tikhonov parameter λ is initialized by maximizing the MLR loss sensitivity *w.r.t.* λ on the first mini-batch (See Appendix 4.6.2). When using AdaCap, we used no other additional regularization tricks. Otherwise we used BN and DO = 0.2 on all hidden layers. Unless mentioned otherwise, we set $\max_{\texttt{Iter}} = 500$ and $\max_{l.r.} = 0.01$, hidden layers width 512 and ReLU activation.

We implemented some architectures detailed in [114, 75]; MLP: MultiLayer Perceptrons of depth 2; ResBlock: Residual Networks with 2 Resblock of depth 2; SNN for MLP with depth 3 and SeLU activation. We define GLU when hidden layers are replaced with Gated Linear Units. Fast denotes a faster version of MLP and SNN with $\max_{\texttt{Iter}} = 200$ and hidden layers width 256. BatchMLP and BatchResBlock denote a slower version where the number of epochs is set at 20 and 50 respectively and the batch size is set at $min(n, 256)$ but the number of iterations is not limited, we enforce a one hour training budget instead. The Batch architectures are outside of the scope of this benchmark and only provided for compute time and performance comparison with iteration bounded versions. In total, we implemented 16 architectures: MLP, FastMLP, BatchMLP, SNN, FastSNN, MLPGLU, ResBlock, BatchResBlock; each time trained with and without AdaCap. These where evaluated individually but to count which methods perform best (Table 4.1) we used a restricted set of methods (#4) for DNN. When the top 1 count is made without AdaCap, we picked MLP, ResBlock, SNN and MLPGLU. When AdaCap is included, we picked MLP, AdaCapResBlock, AdaCapSNN and AdaCapMLPGLU. We do so to avoid biasing results in favor of DNN by increasing the number of contenders from this category. See Table 4.10 in the Appendix.

Other compared methods. We considered CatBoost [160], XGBoost [40], LightGBM [110]), MARS [66] (py-earth implementation) and the scikit learn implementation of **R.F.** and XRF [13, 30], Ridge Kernel and NuSVM[36], MLP [93], Elastic-Net [227], Ridge [95], LASSO [199], Logistic regression (LogReg [46]), CART, XCART [29, 70, 115],Adaboostand XGB [31, 67, 68]. We included a second version of CatBoost denoted FastCatBoost, with hyperparameters chosen to reduce runtime considerably while minimizing performance degradation.

Benchmarked Tabular Data. TD are very diverse. We browsed UCI [9], Kaggle and OpenML [204], choosing datasets containing structured columns, *i.i.d.* samples, one or more specified targets and corresponding to a non trivial learning task, that is the **R.F.** performance is neither perfect nor behind the intercept model. We ended up with 44 datasets (Table 4.8): UCI 34, Kaggle 5 and openml 5, from medical, marketing, finance, human ressources, credit scoring, house pricing, ecology, physics, chemistry, industry and other domains. Sample size ranges from 57 to 36584 and the number of features from 4 to 1628, with a diverse range of continuous/categorical mixtures. The tasks include 26 continuous and ordinal `Reg` and 18 `BinClf` tasks. Data scarcity is a frequent issue in TD [35] and Transfer Learning is almost never applicable. However, the small sample regime was not really considered by previous benchmarks. We included 28 datasets with less than 1000 samples (`Reg` task:15, `BinClf` task:13). We also made a focus on the 8 `Reg` datasets where the smallest `RMSE` achieved by any method is under 0.25, this corresponds to datasets where the SNRis high but the function to approximate is not trivial. For the bagging experiment, we only used the 15 smallest regression datasets to reduce compute time.

Dataset preprocessing. We applied uniformly the following pipeline: • remove columns with id information, time-stamps, categorical features with more than 12 modalities (considering missing values as a modality); • remove rows with missing target value; • replace feature missing values with mean-imputation; • standardize feature columns and regression target column. For some regression datasets, we also applied transformations (*e.g.* $log(\cdot)$ or $\log(1+\cdot)$) on target when relevant/recommended (see Appendix 4.6.4).

Training and Evaluation Protocol. For each dataset, we used 10 different train/test splits (with fixed seed for reproducibility) without stratification as it is more realistic. For each dataset and each split, the test set was only used for evaluation and never accessed before prediction. Methods which require a validation set can split the train set only. We evaluated on both train and test set the $\mathtt{R}^2$-score and `RMSE` for regression and the Accuracy (`Acc.`) or Area Under Curve `AUC` for classification (in a *one-versus-rest* fashion for multiclass). For each dataset and each split we also computed the average performance over the 10 train/test splits for the following global metrics: PMA, P90, P95 and Friedman Rank. We also counted each time a method outperformed all others (`Top1`) on one train/test splits of one dataset.

Meta-Learning and Stacking Since the most popular competitors to `DNN` on TD are ensemble methods, it makes sense to also consider Meta-Learning schemes, as mentionned by [75]. For a subset of `Reg` datasets, we picked the methods from each category which performed best globally and evaluated bagging models, each

comprised of 10 instances of one unique method, trained with a different seed for the method (but always using the same train/test split), averaging the prediction of the 10 weak learners. This scheme multiplies training time by 10, which for most compared methods means a few minutes instead of a few second. Although it has been shown that HPO can drastically increase the performance of some methods on some large datasets, it also most often multiply the compute cost by a factor of 500 (5 Fold CV$*$ 100 iterations in [75]), from several hours to a few days.

Benchmark limitations. This benchmark does not address some interesting but out of scope cases for relevance or compute budget reasons: huge datasets (10M+), specific categorical features handling, HPO, pretraining, Data Augmentation, handling missing values, Fairness, etc., and does not include methods designed for those cases (notably NODE, TabNet, FeatureTokenizer), leaving out the comparison/combination of AdaCap with these.

4.4.2 Tabular Data benchmark results

Main takeaway: AdaCap vs regular DNN.

- Compared with regular DNN, AdaCap is almost irrelevant for classification but almost always improves Reg performance. Its impact compounds with the use of SeLU, GLU and ResBlock.
- Compute time wise, the overcost of the Tikhonov operator matrix inversion is akin to increasing the depth of the network (Table 4.2).

Main takeaway: AdaCap vs other methods.

- There is no SOTA method for TD.
- On regression TD there is a high SNR but the link between observations and target is far from trivial, AdaCap dominates the leaderboard.
- Although AdaCap reduces the impact of the random seed used for initialization (Table 4.7), it still benefits as much from bagging as other non ensemble methods (Table 4.4).

Performance by category. First, we evaluate the global performance of each category of method (GBDT, DNN, **R.F.**, SVM, GLM, MARS, CART) corresponding to a total of 23 contenders (GBDT : 6, DNN : 4, **R.F.** : 2, SVM : 2, GLM : 4, MARS : 1). We consider our benchmark to be a set of 440 contests, (10 train/test splits, 44 datasets). To assess the impact of adding AdaCap to the set of techniques available for DNN,

we run the competition two times: The first time, the DNN category is represented by the 4 best architectures excluding AdaCap (MLP, ResBlock, SNN and GLUMLP). The second time, the methods which leveraged AdaCap where allowed in the DNN team. For fairness, the DNN category is still consisting of 4 methods(MLP, AdaCapResBlock, AdaCapSNN and AdaCapGLUMLP).
Table 4.1 depicts the share of contests won by each category. In terms of achieving Top 1 performance on regression tabular datasets, GBDT comes out on top on only less than 40% of the regression datasets followed by DNN without AdaCap which is at 30 %. By allowing AdaCap to train DNN, the margin of about 9% between GBDT and AdaCap-DNN is divided by 3. The results presented in Columns 4 and 5 are discussed later in this section. Since AdaCap is clearly relevant for regression but ill-suited for classification, the remainder of this section focuses on the results of the regression datasets.

category	RMSE top 1 on 26 TD		RMSE top 1 on the 8 TD with min RMSE < 0.25		BinClf on 18 TD without AdaCap	
	without AdaCap	with AdaCap	without AdaCap	with AdaCap	AUC top 1	Err. rate top 1
GBDT	**39.615** %	**36.538** %	35.0 %	27.160 %	**61.666** %	**73.888**%
DNN	30.0 %	33.461 %	50.0 %	**58.024** %	16.666 %	5.5555%
R.F.	18.461 %	18.076 %	15.0 %	14.814 %	15.0 %	10.0%
SVM	5.7692 %	6.1538 %	0.0 %	0.0 %	0.0 %	0.0%
GLM	4.6153 %	4.6153 %	0.0 %	0.0 %	6.6666 %	7.7777%
MARS	1.5384 %	1.1538 %	0.0 %	0.0 %	N.A.	N.A.
CART	0.000 %	0.0 %	0.0 %	0.0 %	0.0 %	2.7777 %

Table 4.1: Percentage of experiments where the best performing method belongs to the category (higher is better). For each random train/test split of each considered dataset, we evaluate all methods and than consider two competitions: • without AdaCap the DNN category consists of 4 methods: MLP, ResBlock, SNN and MLPGLU; • with AdaCap the DNN category contains the 4 following methods instead: MLP, AdaCapResBlock, AdaCapSNN and AdaCapMLPGLU. In both competitions, all the other categories contains all the method listed in the benchmark description. For classification TD, we did not report results with AdaCap as it under-performs vastly against regular DNN in terms of accuracy and Area Under Curve (AUC), meaning it is not a suitable technique.

Individual performance. Next, we consider methods individually, assessing their performances on the 26 regression datasets. Table 4.2 presents the average RMSE and the P90 R2-score on the 26 regression datasets (averaged over the 10 train/test splits) for the Top 10 methods (by ranking: AdaCapSNN, GLU MLP, AdaCapGLUMLP, AdaCapResBlock, MLP, CatBoost, AdaCapMLP, AdaCapFastSNN , AdaCapBatchResBlock, SNN). In terms of average RMSE, AdaCap should be the default choice (AdaCapSNN: 0.4147, GLU MLP: 0.4201, CatBoost: 0.4221). However, the P90 is higher for CatBoost(66.538) than for AdaCap (AdaCapGLUMLP: 60.384) and regular DNN (GLU MLP: 54.230). Since

the P90 metric corresponds to the percentage of experiments where the best RMSE is not less than 90% of the the RMSE of the considered method, this would indicate that CatBoost is more reliable than AdaCap. However, this metric does not reflect how methods perform when they do not meet this 90% threshold. Again, the results presented in Columns 4 and 5 are discussed later in this section.

method	RMSE avg. all TD	P90 all TD	RMSE avg. TDwith min RMSE < 0.25	P90 TDwith min RMSE < 0.25	avg. runtime avg. (sec.)	max runtime (sec.)
AdaCapSNN	**0.4147**	55.0	0.1486	32.5	**19.798**	169.82
GLU MLP	0.4201	54.230	0.1498	40.0	9.8911	35.699
AdaCapGLUMLP	0.4206	60.384	**0.1455**	**56.25**	22.355	179.88
AdaCapResBlock	0.4214	50.0	0.1532	30.0	17.192	166.13
CatBoost	0.4221	**66.538**	0.1910	45.0	92.518	315.15
MLP	0.4230	50.769	0.1601	26.25	4.0581	22.213
AdaCapMLP	0.4233	46.923	0.1566	17.5	17.208	168.10
AdaCapFastSNN	0.4245	42.307	0.1591	16.25	7.6670	38.286
AdaCapBatchResBlock	0.4257	45.384	0.1580	20.0	194.72	**2654.7**
SNN	0.4260	40.384	0.1526	18.75	**7.1895**	32.581

Table 4.2: Regression task focus: test RMSE (lower is better), P90 (higher is better), and runtime for the 10 methods from all categories which performed best on 26 tabular datasets. RMSE is averaged over 10 train/test splits. The P90 metric measures for each method, the percentage of experiments where the best RMSE is not under 90% of the RMSE of the considered method, meaning it did not underperform too much.

Dataset territories and champions. To assess how methods perform on the datasets where they are not the best, we assign to each category the set of tabular datasets on which it performed best in terms of RMSE, which we call its "dataset territory". We ended up with 6 territories, as CART and MARS nether came first. To avoid unfairly favouring a category that contains more methods, we designate for each category its "champion", *i.e.* the method from that category that obtains the best overall performance. We obtain the following champions: CatBoost, AdaCapSNN , GLU MLP, Elastic-Net, XRF, NuSVM. Table 4.3 depicts the size of each territory (the number of datasets in that set) and the performance of each champion outside of its territory (the added error in percentage w.r.t. to the RMSE of the best performing method). On the 9 regression datasets where CatBoost came first, in average the RMSE performance of AdaCapGLUMLP was only 11.6% higher. Meanwhile, on the 5 datasets where AdaCapGLUMLP came first, the RMSE performance of CatBoost was in average 141.3% worst. This means that AdaCap is actually less likely to underperform catastrophically compared to CatBoost in the cases where the other was best suited.

Datasets with min **RMSE** < 0.25**.** Lastly, we tried to understand the characteristics of the AdaCap territory. The most common denominator between the 5 datasets where AdaCap came first was the absolute value of the best RMSE, which was quite

Dataset Territory			Average variation in RMSE relative to best performance					
Category	Best method	#TD	CatBoost	AdaCapSNN	GLU MLP	Elastic-Net	XRF	NuSVM
GBDT	CatBoost	9	**Best**	11.6%	13.4%	56.8%	12.6%	21.8%
AdaCap	AdaCapSNN	5	141.3%	**Best**	13.6%	176.6%	103.3%	197.9%
DNN	GLU MLP	4	11.5%	5.8%	**Best**	92.1%	31.3%	58.0%
GLM	Elastic-Net	4	0.9%	11.6%	10.3%	**Best**	21.4%	12.6%
R.F.	XRF	3	12.2%	11.2%	13.2%	39.0%	**Best**	40.9%
SVM	NuSVM	1	4.1%	3.1%	4.7%	6.8%	36.8%	**Best**

Table 4.3: Performance of the best methods from each category outside of their dataset "territory" on the regression task: Column 2 shows the overall best model over our regression benchmark for each category of methods, its "champion". Next, we assign each dataset to the champion which obtained top1 performance on it, creating "dataset territories" for each champion. Column 3 corresponds to the number of datasets included in each territory. For example, the territory of CatBoost contains 9 datasets. For each dataset, we compute for each model the ratio of RMSE against smallest obtained RMSE. Columns 4 to 9 display the average variations in % over each territory. Each row corresponds to a dataset territory, each column corresponds to the performance of a champion.

low compared to the others. To verify this intuition, we compared the performances on a benchmark consisting only of the 8 regression TD where the best achievable RMSE is under 0.25. Results are added to Tables 4.1 and 4.2. On this benchmark, DNN with AdaCap included came first 58.024% of the time while GBDT methods only obtained Top 1 performances 27.160% of the time (the comparison is done on each train/test split of each dataset, meaning the ratio is computed over 80 contests). In terms of average RMSE, the best method is AdaCapGLUMLP (RMSE: 0.1455), far above CatBoost (RMSE: 0.1910). Likewise, AdaCapGLUMLP outperforms CatBoost by a wide margin in terms of P90 (56.25 *v.s.* 45). This confirms our claim that AdaCap can delay memorization during training, giving DNN more leeway to capture the most subtle patterns.

Bagging.

Few-shot. We conducted a toy few-shot learning experiment on MNIST [51] to verify that AdaCap is also compatible with CNN architectures in an image multiclass setting. We followed the setting of the pytorch tutorial [151] and we repeated the experiment with AdaCap but without DO nor BN. The results are detailed in Figure 4.3.

4.4.3 Ablation, Learning Dynamic, Dependency study

Figure 4.1 shows the impact of both Tikhonov and MLR on the trained model. AdaCap removes oscillations in learning dynamics Figure 4.2. AdaCap can handle

top 8 best methods	RMSE no bag	RMSE bag10	RMSE % variation %
AdaCapSNN	0.3532	0.3322	−5.933
AdaCapGLUMLP	0.3482	0.3330	−4.362
SNN	0.3615	0.3374	−6.671
GLU MLP	0.3593	0.3402	−5.296
FastMLP	0.3698	0.3492	−5.562
CatBoost	0.3638	0.3610	*-0.782*
FastCat	0.3879	0.3689	−4.884
XRF	0.3813	0.3799	−0.363

Table 4.4: Impact of bagging 10 instances of the same method, on regression TD in terms of average RMSE (lower is better). We took the top methods of each category and for 1 train/test split of 15 regression TD we trained the method 10 times with different random seed and averaged the predictions to evaluate the potential variation of RMSE with simple method ensembling.

small batchsize very well whereas standard MLP fails (Table 4.6). MLP trained with AdaCap performs better in term of RMSE than when trained with BN+DO (Table 4.5). Combining AdaCap with BN or DO does not improve RMSE. The random seed used to generate the label permutation has virtually no impact (Table 4.7).

4.5 Conclusion

We introduced the MLR loss, an in-sample metric for out-of-sample performance, and the Tikhonov operator, a training scheme which modulates the capacity of a FFNN. By combining these we obtain AdaCap, a training scheme which changes greatly the learning dynamic of DNN. AdaCap can be combined advantageously with CNN, GLU, SNN and ResBlock. Its performance are poor on binary classification tabular datasets, but excellent on regression datasets, especially in the high SNR regime were it dominates the leaderboard.

Learning on tabular data has witnessed a regain of interest recently. The topic is difficult given the typical data heterogeneity, data scarcity, the diversity of domains and learning tasks and other possible constraints (compute time or memory constraints). We believe that the list of possible topics is so vast that a single benchmark cannot cover them all. It is probably more reasonable to segment the topics and design adapted benchmarks for each.

In future work, we will investigate development of AdaCap for more recent architectures including attention-mechanism to handle heterogeneity in data. Fi-

nally, we note that the scope of applications for AdaCap is not restricted to tabular data. The few experiments we carried out on MNIST and CNN architectures were promising. We shall also further explore this direction in a future work.

4.6 Appendix

4.6.1 Additional experiments

Table 4.5: Impact of DO and BN on AdaCapMLPFast on ConcreteSlump. DO and BN are applied uniformly on all hidden layers. AdaCapinteracts well with SeLU, GLU, ResBlock but not with DO or BN.

dropout	RMSE without BN	RMSE with BN
0.0	0.1625	0.1872
0.1	0.1850	0.1847
0.2	0.1919	0.1954
0.3	0.2055	0.2067
0.4	0.2176	0.2202
0.5	0.2342	0.2352

Table 4.6: Impact of batch-size on AdaCapMLPFast and MLPFast on the Abalone dataset (n = 3341). Since the number of iterations is bounded ($maxiter$ = 200), the number of time each sample is seen during training diminishes when batch sizes diminishes in our experiments. AdaCap increases the resilience to issues caused by very small batch sizes under fixed number of iterations constraints.

batchsize	MLPFast	AdaCapFast
16	68.646	0.6756
32	7.0196	0.6733
64	2.1226	0.6658
256	0.6644	0.6573
512	0.6626	0.6547
1024	0.6620	0.6522
2048	0.6596	0.6522

4.6.2 Training protocol

Initialization of the Tikhonov parameter. We select λ_{init} which maximizes sensitivity of the MLR objective to variation of λ. In practice, the following heuristic proved successful on a wide variety of data sets. We pick λ_{init} by running a grid-search on the finite difference approximation for the derivative of MLR in (4.9) on the grid $\mathcal{G}_\lambda = \left\{\lambda^{(k)} = 10^{-1} \times 10^{5\times k/11} \ : \ k = 0, \cdots, 11\right\}$:

$$\lambda_{init} = \sqrt{\lambda^{(\hat{k})}\, \lambda^{(\hat{k}+1)}}, \tag{4.9}$$

Table 4.7: Random Seed Impact: 1000 method seeds on 1 train/test split for ConcreteSlump, the only source of randomness for `MLPFast` and `TikhonovMLPFast` is the weight initialization. Strikingly, `Tikhonov`does not just improve performance but also reduces the standard deviation, meaning the impact of initial weights on final results. `AdaCapMLPFast` $T = 1$ uses the `MLR` loss given in (4.7), which introduces randomness when generating label permutations. In all other experiments presented in this chapter, we use $T = 16$ seeds for generating label permutations and average the loss over these 16 label vectors, which is enough to offset the randomness introduced with `MLR`.

DNN	RMSE avg.	RMSE std.
MLPFast	0.1675	0.0148
TikhonovMLPFast	0.0930	0.0071
AdaCapMLPFast $T = 1$	0.0944	0.0091
AdaCapMLPFast $T = 16$	0.0918	0.0084

where

$$\hat{k} = \arg\max \left\{ \left(\texttt{MLR}(\lambda^{(k+1)}, \theta) - \texttt{MLR}(\lambda^{(k)}, \theta) \right), \ \lambda^{(k)} \in \mathcal{G}_{\lambda} \right\}.$$

Our empirical investigations revealed that this heuristic choice is close to the optimal oracle choice of λ_{init} on the test set.

From a computational point of view, the overcost of this step is marginal because we only compute he SVD of $\mathbf{A}^{L-1}$ once and we do not compute the derivation graph of the 11 matrix inversions or of the unique forward pass. We recall indeed that the `Tikhonov` parameter λ is **not an hyperparameter** of our method; it is trained alongside the weights of the `DNN` architecture.

Comments. We use the generic value ($T = 16$) for the number of random permutations in the computation of the `MLR` loss. This choice yields consistently good results overall. This choice of permutations has little impact on the value of the `MLR` loss. In addition, when $T = 16$, GPU parallelization is still preserved.

4.6.3 Further discussion of existing works comparing `DNN` to other models on TD

We complement here our discussion of existing benchmarks in the introduction. *Deep Learning* are often beaten by other types of algorithms on tabular data learning tasks. Very recently, there has been a renewed interest in the subject, with several new methods. See [24] for an extensive review of the state of the art on tabular datasets.

The comparison between simple DNN, NODE, TabNet, **R.F.** and GBDT on TD was made concomitantly by [107], [183] and [75]. Their benchmark are more oriented towards an AutoML approach than ours, as they all use heavy HPO, and report training times in minutes/hours, even for some small and medium size datasets.

The benchmark in [107] compares FFNN to GBDT, NODE, TabNet, ASK-GBDT (Auto-sklearn) also using heavy HPO, on 40 TD (with size ranging from $n = 452$ to $416k+$). They reported that, after 30 minutes of HPO time, regularization cocktails for MLP are statistically significantly better than XGBoost (See Table 3 there). We did not find a similar experiment for CatBoost.

The benchmark in [183] includes 11 datasets from OpenML, Kaggle, Pascal, and MSLR, Million song with n ranging from $7k$ to $1M+$. They compared XGBoost, NODE, TabNet, 1D-CNN, DNF-Net ([109]) and ensemble of these methods.

These 2 benchmarks give mixed signals on how DNN compare to other methods with [107] being more optimistic than [183].

4.6.4 Benchmark description

Our Benchmark philosophy

Current benchmarks usually heavily focus on the AutoML ([183, 228, 217, 91]) usecase ([226]), using costly HPO over a small collection of popular large size datasets, which raised some concerns [116, 52].

Creating a pertinent benchmark for Tabular Data (TD) is still an ongoing process for ML research community. Because researchers compute budget is limited, arbitrages have to be made between number of datasets, number of methods evaluated, intensity of HPO, dataset size, number of train-test splits. We tried to cover a broad set of usecases where improving DNN performance compared to other existing methods is relevant, leaving out hours-long training processes relying on HPO to get the optimal performance for each benchmarked method.

Datasets/preprocessing

Datasets

Our benchmark includes 44 tabular datasets with 26 regression and 18 classification tasks. Table 4.8 contains the exhaustive description of the datasets included in our

benchmark.

Table 4.8: Datasets description

id	name	task	target index	n	p	#cont.	#cat.	bag exp.	*min*RMSE < 0.25
0	Cervical Cancer Behavior Risk	C	−1	57	149	19	14		
1	Concrete Slump Test	R	−1	82	9	9	0	✓	✓
2	Concrete Slump Test	R	−2	82	9	9	0	✓	✓
3	Concrete Slump Test	R	−3	82	9	9	0	✓	✓
4	Breast Cancer Coimbra	C	−1	92	9	9	0		
5	Algerian Forest Fires Dataset Sidi-Bel Abbes	C	−1	96	14	10	1		
6	Algerian Forest Fires Dataset Bejaia	C	−1	97	16	12	1		
7	restaurant-revenue-prediction	R	−1	109	330	37	39		
8	Servo	R	−1	133	21	2	4	✓	✓
9	Computer Hardware	R	−1	167	7	7	0	✓	✓
10	Breast Cancer	C	0	228	42	1	9		
11	Heart failure clinical records	C	−1	239	12	7	5		
12	Yacht Hydrodynamics	R	−1	245	22	4	2	✓	✓
13	Ionosphere	C	−1	280	33	32	1		
14	Congressional Voting Records	C	0	348	48	0	16		
15	Cylinder Bands	C	−1	432	111	1	19		
16	QSAR aquatic toxicity	R	−1	436	34	8	3	✓	
17	Optical Interconnection Network	R	7	512	26	6	5	✓	✓
18	Optical Interconnection Network	R	8	512	26	6	5	✓	✓
19	Optical Interconnection Network	R	5	512	26	6	5	✓	✓
20	Credit Approval	C	−1	552	31	4	8		
21	blood transfusion	C	−1	598	4	4	0		
22	QSAR Bioconcentration classes dataset	R	−1	623	29	9	5	✓	
23	wiki4HE	R	−10	696	284	1	50	✓	
24	wiki4HE	R	−11	704	284	1	50	✓	
25	QSAR fish toxicity	R	−1	726	18	6	2		
26	Tic-Tac-Toe Endgame	C	−1	766	27	0	9		
27	QSAR biodegradation	C	−1	844	123	38	15		
28	mirichoi0218_insurance	R	−1	1070	15	3	4		
29	Communities and Crime	R	−1	1595	106	99	2		
30	Jasmine	C	0	2387	144	8	136		
31	Abalone	R	−1	3341	10	7	1	✓	
32	mercedes-benz-greener-manufacturing	R	1	3367	379	0	359		
33	Sylvine	C	0	4099	20	20	0		
34	christine	C	0	4333	1628	1599	14		
35	arashnic_marketing-seris-customer-lifetime-value	R	2	6479	72	7	15		
36	Seoul Bike Sharing Demand	R	1	7008	15	9	3	✓	
37	Electrical Grid Stability Simulated Data	R	−2	8000	13	12	1		✓
38	snooptosh_bangalore-real-estate-price	R	−1	10656	6	2	1		
39	MAGIC Gamma Telescope	C	−1	15216	10	10	0		
40	Appliances energy prediction	R	2	15788	25	25	0		
41	Nomao	C	−1	27572	272	116	45		
42	Beijing PM2.5 Data	R	5	33405	31	10	3		
43	Physicochemical Properties of Protein Tertiary Structure	R	6	36584	9	9	0		✓

Pre-processing.

To avoid biasing the benchmark towards specific methods and to get a result as general as possible, we only applied as little pre-processing as we could, without using any feature augmentation scheme. The goal is not to get the best possible performance on a given dataset but to compare the methods on equal ground. We first removed uninformative features such as sample index. Categorical features with more than 12 modalities were discarded as learning embeddings is out of the scope of this benchmark. We also removed samples with missing target.

Target treatment. The target is centered and standardized via the function **function-T**$(\cdot)$. We remove the observation when the value is missing.

We applied specific target transforms on some datasets. They are listed below:

id	name	task	target index	source	file name
0	Cervical Cancer Behavior Risk	C	−1	archive.ics.uci.edu	sobar-72.csv
1	Concrete Slump Test	R	−1	archive.ics.uci.edu	slump_test.data
2	Concrete Slump Test	R	−2	archive.ics.uci.edu	slump_test.data
3	Concrete Slump Test	R	−3	archive.ics.uci.edu	slump_test.data
4	Breast Cancer Coimbra	C	−1	archive.ics.uci.edu	dataR2.csv
5	Algerian Forest Fires Dataset Sidi-Bel Abbes	C	−1	archive.ics.uci.edu	Algerian_forest_fires_dataset_UPDATE.csv
6	Algerian Forest Fires Dataset Bejaia	C	−1	archive.ics.uci.edu	Algerian_forest_fires_dataset_UPDATE.csv
7	restaurant-revenue-prediction	R	−1	www.kaggle.com	train.csv.zip
8	Servo	R	−1	archive.ics.uci.edu	servo.data
9	Computer Hardware	R	−1	archive.ics.uci.edu	machine.data
10	Breast Cancer	C	0	archive.ics.uci.edu	breast-cancer.data
11	Heart failure clinical records	C	−1	archive.ics.uci.edu	heart_failure_clinical_records_dataset.csv
12	Yacht Hydrodynamics	R	−1	archive.ics.uci.edu	yacht_hydrodynamics.data
13	Ionosphere	C	−1	archive.ics.uci.edu	ionosphere.data
14	Congressional Voting Records	C	0	archive.ics.uci.edu	house-votes-84.data
15	Cylinder Bands	C	−1	archive.ics.uci.edu	bands.data
16	QSAR aquatic toxicity	R	−1	archive.ics.uci.edu	qsar_aquatic_toxicity.csv
17	Optical Interconnection Network	R	7	archive.ics.uci.edu	optical_interconnection_network.csv
18	Optical Interconnection Network	R	8	archive.ics.uci.edu	optical_interconnection_network.csv
19	Optical Interconnection Network	R	5	archive.ics.uci.edu	optical_interconnection_network.csv
20	Credit Approval	C	−1	archive.ics.uci.edu	crx.data
21	blood transfusion	C	−1	www.openml.org	php0iVrYT
22	QSAR Bioconcentration classes dataset	R	−1	archive.ics.uci.edu	Grisoni_et_al_2016_EnvInt88.csv
23	wiki4HE	R	−10	archive.ics.uci.edu	wiki4HE.csv
24	wiki4HE	R	−11	archive.ics.uci.edu	wiki4HE.csv
25	QSAR fish toxicity	R	−1	archive.ics.uci.edu	qsar_fish_toxicity.csv
26	Tic-Tac-Toe Endgame	C	−1	archive.ics.uci.edu	tic-tac-toe.data
27	QSAR biodegradation	C	−1	archive.ics.uci.edu	biodeg.csv
28	mirichoi0218_insurance	R	−1	www.kaggle.com	insurance.csv
29	Communities and Crime	R	−1	archive.ics.uci.edu	communities.data
30	Jasmine	C	0	www.openml.org	file79b563a1a18.arff
31	Abalone	R	−1	archive.ics.uci.edu	abalone.data
32	mercedes-benz-greener-manufacturing	R	1	www.kaggle.com	train.csv.zip
33	Sylvine	C	0	www.openml.org	file7a97574fa9ae.arff
34	christine	C	0	www.openml.org	file764d5d063390.csv
35	arashnic_marketing-seris-customer-lifetime-value	R	2	www.kaggle.com	squark_automotive_CLV_training_data.csv
36	Seoul Bike Sharing Demand	R	1	archive.ics.uci.edu	SeoulBikeData.csv
37	Electrical Grid Stability Simulated Data	R	−2	archive.ics.uci.edu	Data_for_UCI_named.csv
38	snooptosh_bangalore-real-estate-price	R	−1	www.kaggle.com	blr_real_estate_prices.csv
39	MAGIC Gamma Telescope	C	−1	archive.ics.uci.edu	magic04.data
40	Appliances energy prediction	R	2	archive.ics.uci.edu	energydata_complete.csv
41	Nomao	C	−1	www.openml.org	phpDYCOet
42	Beijing PM2.5 Data	R	5	archive.ics.uci.edu	PRSA_data_2010.1.1-2014.12.31.csv
43	Physicochemical Properties of Protein Tertiary Structure	R	6	archive.ics.uci.edu	CASP.csv

Table 4.9: Dataset description

- $\tilde{y} \leftarrow \log(y)$: `Computer Hardwaremachine`; `Servo`; `Yacht`; `restaurant revenue`; `mirichoi0218 insurance`; `snooptosh bangalore real estate price`; `arashnic marketing seris customer lifetime valuesquark automotive CLV`; `house sales prediction`; `mercedes benz greener manufacturing`; `mcarujo portugal proprieties rent buy and vacation`

- $\tilde{y} \leftarrow \log(y+1)$: `Communities and Crime`.

- $\tilde{y} \leftarrow \log(1-y+y_{\max})$: `tabular playground series jan 2021`; `tabular playground series feb 2021`.

- $\tilde{y} \leftarrow \log(\bar{y}-\bar{y}_{\min}+2)$ with $\bar{y} = -\log(1+y)$: `Seoul Bike Sharing`

Features treatment. The imputation treatment is done during processing. For categorical features, `NAN` Data may be considered as a new class. For numerical features, we replace missing values by the mean. Set $n_j = \#\texttt{set}(X_j)$ the number of distinct values taken by the feature X_j, We proceed as follows :

- When $n_j = 1$, the feature X_j is irrelevant, we remove it.

- When $n_j = 2$ (including potentially `NAN` class), we perform numerical encoding of binary categorical features.

- Numerical features with less than 12 distinct values are also treated as categorical features ($2 < n_j \leq 12$). We apply one-hot-encoding.

- Finally, categorical features with $n_j > 12$ are removed.

Methods

Usual methods. We evaluate `AdaCap` against a large collection of popular methods (`XGB` [31, 67, 68], `XGBoost` [40], `CatBoost` [160], `LightGBM` [110], **R.F.** and `XRF` [30, 13], `SVM` and kernel-based [36], `MLP` [93], `Elastic-Net` [227], `Ridge` [95], LASSO [199], `Logistic regression` [46], `MARS` [66], `CART`, `XCART` [29, 70, 115]).

FastCatBoost hyperparameter set CatBoostis one of the dominant method on TD, but also one of the slowest in terms of training time. We felt it was appropriate to evaluate a second version of CatBoostwith a set of hyper-parameters designed to cut training time without decreasing performance too much. Following the official documentation guidelines, we picked the following hyperparameters for FastCatBoost:

- "iterations":100,
- "subsample":0.1,
- "max_bin":32,
- "bootstrap_type",
- "task_type":"GPU",
- "depth":3.

FFNN architectures See Table 4.10 for the different FFNN architectures included in our benchmark.

Table 4.10: FFNN architectures included in our benchmark. Legend for Block type: L=Linear; Res=Residual,GL=Gated Linear; GR=Gated Residual

Name	Block Type	# Blocks	Block Depth	Activation	Width	Iter/ epochs	Batch Size	max lr
FastMLP	Linear	2	1	ReLU	256	200	$\min(n, 2048)$	$1e-2$
MLP	Linear	2	1	ReLU	512	500	$\min(n, 2048)$	$1e-2$
BatchMLP	Linear	2	1	ReLU	512	20	256	$1e-2$
FastSNN	Linear	2	1	ReLU	256	200	$\min(n, 2048)$	$1e-2$
SNN	Linear	2	1	SeLU	512	500	$\min(n, 2048)$	$1e-2$
ResBlock	ResBlock	2	2	ReLU	512	500	$\min(n, 2048)$	$1e-2$
BatchResBlock	ResBlock	2	2	ReLU	512	50	256	$1e-2$
GLU MLP	GLU	3	1	ReLU/Sigmoid	512	500	$\min(n, 2048)$	$1e-2$

4.6.5 Hardware Details

We ran all benchmark experiments on a cluster node with an Intel Xeon CPU (Gold 6230 20 cores @ 2.1 Ghz) and an Nvidia Tesla V100 GPU. All the other experiments were run using an Nvidia 2080Ti.

Chapter 5
MCD: Marginal Contrastive Discrimination

Without contraries is no progression.

William Blake, The argument

Chapter Summary

We consider the problem of conditional density estimation (**CDE**), which is a major topic of interest in the fields of statistical and machine learning. Our method, called Marginal Contrastive Discrimination, **MCD** , reformulates the conditional density function into two factors, the marginal density function of the target variable and a ratio of density functions which can be estimated through binary classification. Like noise-contrastive methods, **MCD** can leverage *state-of-the-art* supervised learning techniques to perform **CDE** , including neural networks. Our benchmark reveals that our method significantly outperforms in practice existing methods on most density models and regression datasets.

5.1 Introduction

We consider the problem of conditional density estimation (**CDE**), which is a major topic of interest in the fields of statistical and machine learning.

We consider a couple of random variables (X, Y) taking values in $\mathcal{X} \times \mathcal{Y}$ such that $\mathcal{X} \subseteq \mathbb{R}^p$ and $\mathcal{Y} \subseteq \mathbb{R}^k$. We assume in this chapter that all random variables admit a density function with respect to a dominant measure. We also assume all these densities are proper *i.e*. they integrate to 1. We denote f_X and f_Y the marginal densities of X and Y with respect to a dominant measure. Our goal is to estimate the conditional density function:

$$\begin{aligned} \mathcal{X} \times \mathcal{Y} &\longrightarrow \mathbb{R}_+ \\ (x, y) &\longmapsto f_{Y\,|\,X=x}(y) \end{aligned}$$

This problem is at the root of the majority of machine learning tasks, including supervised and unsupervised learning or generative modelling. Supervised learning techniques aims at estimating the conditional mean. Meanwhile, in the binary classification setting, these two tasks are equivalent, since $\mathbb{E}[Y\,|\,X = x] = \mathbb{P}[Y = 1\,|\,X = x]$. In the regression setting where Y is a continuous variable, the conditional density is far more informative than the mean value. This is especially true when the conditional distribution is multi-modal, heteroscedastic or heavy tailed. Moreover, in many fields such as actuarial science, asset management, climatology, econometrics, medicine or astronomy, one is interested

in quantities other than expectation, such as higher order moments (variance, skewness kurtosis), prediction intervals, quantile regression, outlier boundaries, etc.

Meanwhile, most unsupervised learning techniques aim to discover relationships and patterns between random variables. This corresponds to the joint density probability function estimation subtask, itself a subtask of **CDE** . Similarly, the field of generative modeling, whose goal is to generate synthetic data by expressing the joint distribution as a product of univariate conditional distributions with respect to latent variables, can also be considered a **CDE** subtask, as realistic images or sounds correspond to the modes of the distribution.

5.1.1 Related work

Historically, the first attempts were based on the use of Bayes' formula (5.1) which transforms the **CDE** into the estimation of two density functions, thus allowing the use of techniques dedicated to density estimation.

$$f_{Y\mid X=x}(y) = \frac{f_{X,Y}(x,y)}{f_X(x)},\ f_X(x) \neq 0 \tag{5.1}$$

In this chapter, we address the many real-world applications of supervised learning regressions where we have a one-dimensional target but a large number of features. A major flaw of reformulating a conditional density as the ratio of two densities, is that even if Y is low dimensional, we will still incur the curse of dimensionality if X is high-dimensional. Although techniques exist to alleviate the curse to some extent by leveraging sparsity or other properties of X, it would be far easier to reformulate the **CDE** so that only the estimate of the marginal density f_Y is required instead of f_X .

Nonparametric density estimation. Nonparametric estimation is a powerful tool for density estimation as it does not require any prior knowledge of the underlying density. One of the first intuitions of kernel estimators was proposed by [171] and later by [149]. Kernel estimators were then widely studied ranging from bandwidth selection ([72]), non linear aggregation ([170]), computational optimisation ([123]) to the extension to conditional densities ([20]). We refer to [184] and references therein for interested reader. A very popular and effective nonparametric method is the k-Nearest Neighbors ([64]), but like other nonparametric methods (kernel estimators, histogram [153],...), a main limitation is the curse of dimensionality ([178, 140]). [184] showed that, in a density estimation setting, the number of points n, needed to obtain equivalent results when fitting a

d-dimensional random variable, grows at a rate of $n^{\frac{4}{4+d}}$. Meanwhile, the impact of dimensionality on the computational time also scales exponentially. To the best of our knowledge, the best performing method for kernel estimation, based on a divide-and-conquer algorithm ([123]), has a complexity of $\mathrm{T}(n\log(n)^{\max(d-1,1)})$. Both of these relationships are compounded, as higher dimensionality means that exponentially more data is required and the computational time relative to the size of the dataset will also grow exponentially.

Noise Contrastive methods. Another important family of techniques for density estimation is noise-contrastive learning ([80]). These techniques reformulate the estimation of the density into a binary classification problem (up to a constant). It consists in introducing a known probability density $g(\cdot)$ and sampling from it a synthetic dataset. The latter is concatenated with the original dataset, then a target value Z_i is associated such that Z_i is equal to 1 if the observation comes from the original dataset and 0 otherwise. A **contrast** $q(\cdot)$, which can be, under certain conditions on g, directly related to the density of the original observations is introduced to obtain an estimation of the density function :

$$q(x) := \mathbb{P}[Z = 1 \,|\, X = x] = \frac{f_X(x)}{f_X(x) + g(x)} \quad \text{and} \quad f_X(x) = g(x)\frac{q(x)}{1 - q(x)}. \tag{5.2}$$

A binary classifier is then trained to predict this target value conditionally on the associated observation.

Contrast learning has been successfully applied in the area of self-supervision learning ([103, 87]), especially on computer vision tasks ([11]). Many extensions and improvements have been made, including w.r.t. the learning loss function ([112]), data augmentation techniques ([41]), network architectures ([87]) and computational efficiency ([220]). Our proposed method is partially based on this technique, with two major differences. First, our technique addresses **CDE** problem and not density estimation or self-supervised learning. Second, the distribution we choose is unknown, potentially intractable, and/or with a large set of highly dependent components, which violates the usual restrictions for performing noise contrastive density estimation but allows us to taylor the noise distribution precisely to the estimated distribution. This allows us to fit the noise distribution precisely to the estimated distribution. To the best of our knowledge, the technique closest to our method is designed to evaluate the deviation from the independence setting, *i.e.* when all random features $\{X_j\}_{j=1,\cdots,p}$ are independent, in an unsupervised framework. It has been briefly described in the second edition of [200] (pages 495-497) where, based on the noise contrastive reformulation given above with

$g(\cdot) = \Pi_{j=1}^{p} f_{X_j}(\cdot)$. Note that g is unknown and corresponds to what we will call later the noise distribution. Nevertheless, noise samples can be generated by applying a random permutation on the feature columns of the dataset, which is sufficient to discover association rules between the X features but not to estimate the density function f_X. On the other hand, although each component of X is one-dimensional by definition, the errors made when estimating the marginal densities p will compound when estimating g, which means that the dimensionality of X is again a limiting factor.

CDE reformulation. There are other ways to reformulate the **CDE**, for example [195] propose a reformulation into a Least-Squares Density Ratio (**LSCond**) and [134] into a quantile regression task. Still others take advantage of a reformulation of the **CDE** into a supervised learning regression task. Meanwhile, **RFCDE** ([159]) and **NNKCDE** ([157]) are techniques that adapt methods that have proven to be effective to the **CDE** task. Other methods, such as **Deepcde** ([56]) and **FlexCode** ([102]), go further and design a surrogate regression task that can be run by an off-the-shelf supervised learning method taking advantage of many mature and well-supported open-source projects, *e.g.* scikit-learn ([154]), *pytorch* ([151]), *fastai* ([99]) , *keras* ([43]), *tensorflow* ([133]), **CatBoost** ([160]), `XGB`([40]), etc. Our method implementation also has this advantage, since it can take as arguments any class that follows the scikit-learn $init/fit/predict_proba$ API, which includes our off-the-shelf `MLP`implementations, linear grid units (`GLU`,[75]), `ResBlock`([75]) and self-normalizing networks (`SNN`,[114]).

Neural networks and variational inference. Long before the current resurgence of interest in deep learning, there were already neural networks designed specifically to handle **CDE**, *e.g.* [22] adressed the problem of estimating a probability density using Mixture Density Networks (**MDN**). More recently, Kernel Mixing Networks (**KMN**, [5]) are networks trained to estimate a family of kernels to perform the **CDE** task. Another famous variational inference technique is Flow Normalization (**N.Flow**, [169]), designed to solve the problem of finding the appropriate approximation of the posterior distribution. A common challenge with most of these methods is scalability in terms of computation resources, which our experimental benchmark confirmed. A *python* implementation of **MDN**, **KMN** and **N.Flow** is provided by the $freelunchtheorem$ package ([1]), which we included in our benchmark.

5.1.2 Our contributions

Our method, called contrastive marginal discrimination, **MCD**, combines several characteristics of the above methods but without their respective limitations. At a basic level, **MCD** begins by reformulating the conditional density function into two factors, the marginal density function of Y and a ratio of density functions as follows

$$f_{Y|X=x}(y) = f_Y(y)\frac{f_{X,Y}(x,y)}{f_X(x)f_Y(y)}, \quad \forall(x,y) \in \mathcal{X} \times \mathcal{Y} \text{ s.t. } f_X(x) \neq 0,\ f_Y(y) \neq 0$$

We propose to estimate these two quantities separately. In most real applications of **CDE**, Y is univariate or low-dimensional, while X is not. In these cases, it is much easier to estimate f_Y rather than f_X and $f_{X,Y}$. In this chapter, we do not focus on how to choose the estimation method for f_Y or introduce new techniques to estimate f_Y. By contrast, our experimental results show that out-of-shelf kernel estimators with default parameters perform very well on both simulated density models and real datasets, as expected for univariate distributions.

The core of our method is the reformulation of the ratio of the density functions $f_{X,Y}/(f_X f_Y)$ into a contrast. In our method, the introduced noise in equation 5.2 always corresponds to the density function $g(\cdot) = f_X f_Y$. This is akin to the reformulation proposed by [200], except that we only break the relationship between the two elements of the pair (X,Y) but not between each component of X and Y: $f_{X,Y}(\cdot) = f_X(\cdot)f_Y(\cdot)\frac{q(\cdot)}{1-q(\cdot)}$. To estimate the joint density $f_{X,Y}$ it would be necessary to estimate both f_X and f_Y. But when we apply Bayes' formula (5.1), we divide by f_X, which disappears from the expression of $f_{Y|X}$, meaning we only need to estimate f_Y and q.

Like noise-contrastive methods, **MCD** can leverage *state-of-the-art* supervised learning techniques to perform **CDE**, especially neural networks. Our numerical experiments reveal **MCD** performances are far superior when using neural networks compared to other popular classifiers like **CatBoost**, `XGB`or Random Forest. Our benchmark also reveals that our method significantly outperforms in practice **RFCDE**, **NNKCDE**, **MDN**, **KMN**, **N.Flow**, **Deepcde**, **FlexCode** and **LSCond** on most the density models and regression datasets included in our benchmark. Moreover, the **MCD** reformulation enables us to train the binary classifier on a contrast training set much larger than the original dataset. Evermore, **MCD** can easily take advantage of additional data. Unlabeled observations can be directly used to increase the size of the training set, without any drawbacks. Similarly, in the case where each observation is associated with more than one target value, they can all be included in the training dataset.

Our main contributions are as follows:

- We introduce a reformulation of the **CDE** problem into a contrastive learning task which combines a binary classification task and a marginal density estimation task, which are both much easier than **CDE** .
- We prove that given a training set of size n it is always possible to generate a *i.i.d*. training set of size $\left\lfloor \frac{n}{2} \right\rfloor$ corresponding to the contrast learning task. We also prove that it is always possible to generate a training set of size at most n^2 in the non *i.i.d*. case.
- We provide the corresponding construction procedures and the *python* implementation. We also provide construction procedures to leverage additional marginal data and multiple targets per observations, which can improve performances significantly.
- We produce a benchmark of 9 density models and 12 datasets. We combine our method with a large set of classifiers and neural networks architectures, and compare ourselves against a large set of **CDE** methods. Our benchmark reveals that **MCD outperforms all the existing methods on the majority of density models and datasets, sometimes by a very significant margin**.
- We provide a *python* implementation of our method compatible with any *pytorch* module or *scikit-learn* classifier, and the complete code to replicate our experiments.

The rest of this chapter is organised as follows: section 5.2 provides a theoretical background for the reformulation. Section 5.4 provides the implementation details and evaluates our method on density models and regression datasets, comparing results with the methods implemented in the *python* frameworks *CDEtools* [48, 158] and *freelunchtheorem* [3, 172]. Section 5.5 provides an ablation study of our method. Section 5.6 provides the proofs for theoretical results and the algorithms to construct the training dataset.

5.2 Marginal Contrastive Discrimination

5.2.1 Setting

In this chapter, we consider three frameworks corresponding to three different situation in practice.

Framework 1. ***[Independent Identically Distributed Samples]***
In this most classic setting, we consider $\mathcal{D}_n^{X,Y} = \{(X_i, Y_i)\}_{i=1,\cdots,n}$ *a training dataset of size* $n \in \mathbb{N}^*$ *such that* $\forall i = 1, \cdots, n$*, the* (X_i, Y_i) *are i.i.d. of density* $f_{X,Y}$.

In practice, it is often the case that additional observations are available but without the associated target values and vice versa (Framework 2). In this chapter we show that it is possible to take advantage of these additional samples to increase the size of the training dataset without using any additional unsupervised learning techniques.

Framework 2. ***[Additional Marginal Data]***
In this framework, we still consider a i.i.d. training dataset of size $n \in \mathbb{N}^*$ *of density* $f_{X,Y}$ *denoted* $\mathcal{D}_n^{X,Y} = \{(X_i, Y_i)\}_{i=1,\cdots,n}$*. Moreover we assume we have one or two additional datasets.*

- *Let* $\mathcal{D}_{n_x}^{X} = \{\widetilde{X}_i\}_{i=1,\cdots,n_x}$ *be i.i.d. an additional dataset of size* $n_x \in \mathbb{N}$ *of density* f_X.
- *Let* $\mathcal{D}_{n_y}^{Y} = \{\widetilde{Y}_i\}_{i=1,\cdots,n_y}$ *be i.i.d. an additional dataset of size* $n_y \in \mathbb{N}$ *of density* f_Y.

We assume that $\mathcal{D}_n^{X,Y}$, $\mathcal{D}_{n_x}^{X}$ *and* $\mathcal{D}_{n_y}^{Y}$ *are independent.*

Let us introduce a last framework, the one where more than one target value is associated to the same observation. To our knowledge, the article of ([26]) is the only attempt to deal with this case. We show in this chapter that our method can exploit and take advantage of these additional targets, again, without requiring an additional learning scheme. This can be the case for example in mechanics when performing fatigue analysis ([26, 132]).

Framework 3. ***[Multiple Target per Sample]***
Let $(X, \mathbb{Y})$ *be a couple of random variables taking values in* $\mathcal{X} \times \mathcal{Y}^m$ *of density* $f_{X,\mathbb{Y}}$. *Let* $\mathcal{D}_{n,m}^{X,\mathbb{Y}} = \{(X_i, \mathbb{Y}_i)\}_{i=1}^{n}$ *a training dataset of size* $n \in \mathbb{N}^*$ *such that*

- *The* $\{X_i\}_{i=1,\cdots,n}$ *are sampled such that the* X_i *are i.i.d. of density* f_X.

- *The* $\{\mathbb{Y}_i\}_{i=1,\cdots,n}$ *are sampled such that the* $\mathbb{Y}_i \,|\, X_i$ *are i.i.d. of density* $f_{\mathbb{Y}\,|\,X}$.

5.2.2 Contrast function

Our method, **MCD**, is grounded on a trivial approached based on the successive application of the Bayes' formula (5.1).

We reformulate the problem differently from the existing noise contrastive methods, focusing on the contrast between the joint law $f_{X,Y}$ and the marginal laws f_X and f_Y. To do this, we define (Definition 5) a new contrast $q(\cdot,\cdot)$, called the marginal contrast function.

Definition 5. ***[Marginal Contrast function MCF**(r)**]***
Let $r \in (0,1)$ *be a real number. Consider* (X,Y) *a couple of random variables taking values in* $\mathcal{X} \times \mathcal{Y}$. *The Marginal Contrast Function with ratio r of the couple* (X,Y)*, denoted* $q(\cdot,\cdot)$*, is defined as:*

$$\begin{aligned} \mathcal{X} \times \mathcal{Y} &\longrightarrow [0,1) \\ (x,y) &\longmapsto q(x,y) := \frac{r f_{X,Y}(x,y)}{r f_{X,Y}(x,y) + (1-r) f_X(x) f_Y(y)}. \end{aligned}$$

This new contrast is motivated by the Fact 1: **CDE** is equivalent to the marginal density of Y and the marginal contrast function q. The **CDE** task is therefore reduced to the estimation of the constrast and a marginal density.

Fact 1. *Let* $r \in (0,1)$ *be a real number. Consider* (X,Y) *a couple of random variables taking values in* $\mathcal{X} \times \mathcal{Y}$. *For all* $(x,y) \in \mathcal{X} \times \mathcal{Y}$*, we have*

$$f_{Y\,|\,X=x}(y) = f_Y(y) \frac{q(x,y)}{1-q(x,y)} \frac{1-r}{r}$$

where $q(\cdot,\cdot)$ *denotes the* ***MCF***(r).

In the next section, we determine the conditions (Marginal Discrimination Conditions) under which the contrast function estimation can be transformed into an easy supervised learning estimation problem.

5.2.3 Marginal Discrimination Conditions

To transform the problem of estimating q into a problem of supervised learning, we first need to introduce a couple of random variables (W, Z) satisfying the Marginal Discrimination Condition (MDcond) of the couple (X, Y) with ratio $r \in (0, 1)$.

Definition 6. ***[Marginal Discrimination Condition MDcond(r)]***
Let $r \in (0, 1)$ be a real number. Consider (X, Y) two random variables taking values in $\mathcal{X} \times \mathcal{Y}$. A couple of random variables (W, Z) is said to satisfy the Marginal Discrimination Condition (MDcond) of the couple (X, Y) with ratio r if

(Cd 1) *The random variable Z follows a Bernoulli law of parameter r ($Z \sim \mathcal{B}(r)$).*

(Cd 2) *The support of W is $\mathcal{W} = \mathcal{X} \times \mathcal{Y}$.*

(Cd 3) *For all $(x, y) \in \mathcal{X} \times \mathcal{Y}$, we have $f_W(x, y) = r f_{X,Y}(x, y) + (1 - r) f_X(x) f_Y(y)$.*

(Cd 4) *For all $(x, y, z) \in \mathcal{X} \times \mathcal{Y} \times \{0; 1\}$, we have*

$$f_{W \mid Z=z}(x, y) = \mathbb{1}_{z=1} f_{X,Y}(x, y) + \mathbb{1}_{z=0} f_X(x) f_Y(y).$$

Remark that condition (**Cd 3**) is satisfied if conditions (**Cd 1**),(**Cd 2**) and (**Cd 4**) are verified. These conditions are sufficient to characterise both the joint and marginal laws of the couple (W, Z).

Estimation of the contrast function through supervised learning The Proposition 1 specifies how the marginal contrast function q problem can be estimated by a supervised learning task, using either a regressor or a binary classifier, provided that we have access to a sample of identically distributed (*i.d.*) of random variables that satisfies the MDcond(r).

Proposition 1. ***[Constrast Estimation]***
Let $r \in (0, 1)$ be a real number. Consider (X, Y) a couple of random variables taking value in $\mathcal{X} \times \mathcal{Y}$. For all $(x, y) \in \mathcal{X} \times \mathcal{Y}$, the Marginal Contrast Function of couple (X, Y) with ratio r denoted by q satisfies the following property:

$$q(x, y) = \mathbb{E}[Z \mid W = (x, y)] = \mathbb{P}[Z = 1 \mid W = (x, y)].$$

The proof is given in section 5.6.2. It remains to prove that it is possible to construct a training set of identically distributed (*i.d.*) samples of (W, Z) using the elements of the original dataset.

5.3 Contrast datasets construction

5.3.1 Classical Dataset (Framework 1)

Theorem 1 establishes the existence of an *i.i.d.* sample of (W, Z) satisfying the MDcond(r) in Framework 1. In its proof (section 5.6.3), such of construction based on the original data set is derived. From now and for all $\alpha \in \mathbb{R}$, the quantity $\lfloor \alpha \rfloor$ denote the largest integer value smaller or equal to α.

Theorem 1. ***[Construction of an** i.i.d.* ***training Set]***
Let $r \in (0, 1)$ be a real number. Consider the dataset $\mathcal{D}_n^{X,Y}$ defined in Framework 1.

Then, we can construct a dataset $\mathcal{D}_N^{W,Z} = \{(W_i, Z_i)\}_{i=1,\cdots,N}$ of size $N = \lfloor n/2 \rfloor$ of i.i.d. observations satisfying the MDcond(r).

Note that, in practice, having access to a larger data set improves the results considerably. By dispensing with the independence property, it is possible to construct a much larger data set without deteriorating the results (see numerical experiments). This is the purpose of Theorem 2

Theorem 2. ***[Construction of a larger** i.d.* ***training Set]***
Consider the dataset $\mathcal{D}_n^{X,Y}$ defined in Framework 1. Moreover, assume that $\mathcal{D}_n^{X,Y}$ is such that $\forall (i, j) \in \{1, \cdots, n\}^2$ with $i \neq j$

$$X_i \neq X_j \text{ and } Y_i \neq Y_j$$

Then, for any couple of integers (n_J, n_M) such that

$$\begin{cases} 1 \leq n_J \leq n \\ 1 \leq n_M \leq n(n-1) \end{cases}$$

we can construct a dataset $\mathcal{D}_N^{W,Z} = \{(W_i, Z_i)\}_{i=1,\cdots,N}$ *of size* $N = n_J + n_M$ *of i.d. random observations satisfying the MDcond*$\left(\frac{n_J}{N}\right)$.

Note first that we can at most construct a dataset of size $N = n^2$, with $r = \frac{1}{n}$. On the other hand, if we want to have $r = \frac{1}{2}$, we can generate a dataset of size $N = 2n$. Second, the additional conditions on the dataset are introduced to exclude the trivial case where a larger dataset is constructed by simply repeating the existing samples. In practice, we can always avoid this case by removing redundant samples. Note, however, that the repetition of values occurs with probability 0, almost surely, because we consider continuous densities. Finally, the complete construction of such a dataset is described in section 5.6.4

5.3.2 Additional Marginal Data (Framework 2)

Now consider that we have additional features and/or targets for which the target or associated feature is not available. We include this additional data in our training process without using a semi-supervised scheme.

Theorem 3. ***[Construction of an*** *i.i.d.* ***training set]***
Let $r \in (0, 1)$ *be a real number. Consider the datasets defined in Framework 2. Set*

$$N = min\left(n, \left\lfloor \frac{n + n_x + n_y}{2} \right\rfloor\right),$$

then, we can construct $\mathcal{D}_N^{W,Z} = \{(W_i, Z_i)\}_{i=1,\cdots,N}$ *a dataset of size N of i.i.d. observations satisfying the MDcond*(r).

This theorem implies that as soon as we have $n_x + n_y \geq n$, we can generate a training set for the discriminator as large as the original set, i.e. $N = n$. In practice, this can happen in many cases. For example, when data annotation is expensive or difficult, we often have $n_x >> n$. At the same time, to use contrastive marginal discrimination to estimate the conditional density, we need to know or estimate the marginal density f_Y. The proof of this theorem is done in section 5.6.5

Theorem 4. ***[Construction of a larger*** *i.d.* ***training set]***
Consider the dataset defined in Framework 3. Moreover assume

• *The dataset* $\mathcal{D}_n^{X,Y}$ *is such that* $\forall (i,j) \in \{1,\cdots,n\}^2$ *s.t.* $i \neq j$

$$X_i \neq X_j \text{ and } Y_i \neq Y_j.$$

• *The datasets* $\mathcal{D}_{n_x}^{X} n_x$ *and* $\mathcal{D}_{n_y}^{Y} n_y$ *are s.t.* $\forall (i,j) \in \{1,\cdots,n_x\}^2$ *and* $\forall (i',j') \in \{1,\cdots,n_y\}^2$

$$\widetilde{X}_i \neq \widetilde{X}_j \text{ and } \widetilde{Y}_{i'} \neq \widetilde{Y}_{j'}$$

• *Moreover, we assume that* $\mathcal{D}_n^{X,Y}$, $\mathcal{D}_{n_x}^{X}$ *and* $\mathcal{D}_{n_y}^{Y}$ *are such that* $\forall (i,i') \in \{1,\cdots,n\} \times \{1,\cdots,n_x\}$ *and* $\forall (j,j') \in \{1,\cdots,n\} \times \{1,\cdots,n_y\}$

$$X_i \neq \widetilde{X}_{i'} \text{ and } Y_j \neq \widetilde{Y}_{j'}$$

Then, for any couple of integers (n_J, n_M) *such that*

$$\begin{cases} 1 \leq n_J \leq n \\ 1 \leq n_M \leq (n+n_x)(n+n_y) - n \end{cases}$$

we can a dataset $\mathcal{D}_N^{W,Z} = \{(W_i, Z_i)\}_{i=1,\cdots,N}$ *of size* $N = n_J + n_M$ *of i.d. random observations satisfying the MDcond*$(\frac{n_J}{N})$.

Here again, it is possible to build a much larger *i.d.* training dataset under some weak assumption. Indeed, the repetition of values occurs with probability 0, almost surely. The complete construction of such a dataset is described in section 5.6.6

Multiple targets per observations (Framework 3)

In Framework 3, to each of the n observations X_i, there exists a m-associated target $\mathbb{Y}_i = (Y_{i,1}, \cdots, Y_{i,m})$ of *i.i.d.* components such that the $\left(Y_{i,j} \,|\, X_i\right)_{j=1,\cdots,m}$ are *i.i.d.*. In this setting, it is still possible to construct, under some weak assumption, a larger *i.d.* training set satisfying the MDcond. Recall, the repetition of values occurs with probability 0, almost surely. Note that we can construct, at most, a data set of size $n^2 \times m$, with $r = \frac{1}{n}$. On the other hand, if we want to have a ratio of $r = \frac{1}{2}$, the generated dataset will be of size $2n \times m$.

Theorem 5. ***[Construction of a larger*** *i.d.* ***training set]***
Consider the dataset $\mathcal{D}_{n,m}^{X,\mathbb{Y}} = \{(X_i, \mathbb{Y}_i)\}_{i=1}^n$ *a training dataset of size* $n \in \mathbb{N}^*$ *defined in Framework 3. Assume that*

- *For all* $(i, i') \in \{1, \cdots, n\}^2$ *such that* $i \neq i'$

$$X_i \neq X_{i'}.$$

- *For all* $(i, j), (i', j') \in \{1, \cdots, n\} \times \{1, \cdots, m\}$ *such that* $i \neq i'$ *or* $j \neq j'$

$$Y_{i,j} \neq Y_{i',j'}$$

Then, for any couple of integers (n_J, n_M) *such that*

$$\begin{cases} 1 \leq n_J \leq n \times m \\ 1 \leq n_M \leq n(n-1) \times m \end{cases}$$

we can construct a dataset $\mathcal{D}_N^{W,Z} = \{(W_i, Z_i)\}_{i=1,\cdots,N}$ *of size* $N = n_J + n_M$ *of i.d. random observations satisfying the MDcond*$\left(\frac{n_J}{N}\right)$.

The complete construction of such a dataset is described in section 5.6.7

5.4 Experiments

In this section we detail the implementation of **MCD** in section 5.4.1 and provide a benchmark to compare **MCD** to other available methods. All our experiments are done in *python* and the random seed is always set such that the results of our method are fully reproducible.

We compare our method **MCD** with other well-known methods presented in section 5.4.2 on both density models including two new ones, described in section 5.4.3 and real dataset. Our results are displayed in sections 5.4.4 and 5.4.5.

5.4.1 Method implementation

Training. First describe the procedure used for both parametric and nonparametric estimation, to train our estimator **MCD** on dataset corresponding respectively to Framework 1, 2 or 3. Table 5.4.1 details which Construction to use in each Framework.

Training	(Step$_1$). Set r. (Step$_2$). Estimate f_Y using $\{Y_i\}_{i=1,\cdots,n}$ from $\mathcal{D}_n^{X,Y}$. (Step$_3$). Generate $\mathcal{D}_N^{W,Z}$. (Step$_4$). Train either a regressor or a binary classifier on $\mathcal{D}_N^{W,Z}$.

The estimators obtained in steps 2 and 4 are called respectively the **marginal estimator** $\widehat{f_Y}$ and the **discriminator** $\widetilde{q}$. Note that for any $(x, y) \in \mathcal{X} \times \mathcal{Y}$, we have $\widetilde{q}(x, y) \in [0, 1]$ while $q(x, y) \in [0, 1)$. To obtain an appropriate prediction, we introduce a thresholding constant $\epsilon = 10^{-6}$ and set

$$\widehat{q}(x, y) = \min(\widetilde{q}(x, y), 1 - \epsilon) \in [0, 1 - \epsilon] \subset [0, 1).$$

Framework	Available datasets	*i.i.d.* samples	*i.d.* samples
Framework 1	$\mathcal{D}_n^{X,Y}$	Construction 1	Construction 2
Framework 2	$\mathcal{D}_n^{X,Y}, \mathcal{D}_{n_x}^{X}, \mathcal{D}_{n_y}^{Y}$	Construction 3	Construction 4
Framework 3	$\mathcal{D}_{n,m}^{X,\mathbb{Y}}$	Not applicable	Construction 5

Table 5.1: Look-up table to determine the appropriate construction corresponding to each framework.

Next, underline that **MCD** can be used to perform two different tasks:

- Nonparametric estimation of the conditional density $f_{Y|X=x}(\cdot)$ for all $x \in \mathcal{X}$.
- Pointwise estimation of the conditional density $f_{Y|X=x}(y)$ for any $(x, y) \in \mathcal{X} \times \mathcal{Y}$.

Indeed, using Fact 1 we have: $\forall (x, y) \in \mathcal{X} \times \mathcal{Y}$,

$$\widehat{f_{Y|X=x}}(y) = \widehat{f_Y}(y) \frac{\widehat{q}(x, y)}{1 - \widehat{q}(x, y)} \frac{1 - r}{r}. \quad (5.3)$$

This implies that it is sufficient to have estimators of both f_Y and q to have a point estimate of $f_{Y|X=x}(y)$. We may also deduce the literal expression of $\widehat{f_{Y|X=x}}(\cdot)$, the nonparametric estimate of the conditional density, provided we know the literal expressions of $\widehat{q}(x, \cdot)$ and $\widehat{f_Y}(\cdot)$. It is the case for $\widehat{q}$ in Deep Learning, as we can write the literal expression of $\widehat{q}(x, \cdot)$ from the **N.N.** parameters learned in the learning step and a value x (see chapter 2). Under certain assumptions on $\widehat{q}$, $\widehat{f_{Y|X=x}}(\cdot)$ is a true density ([80]).

Prediction. For parametric pointwise estimation, at test time, given any new observation $x \in \mathcal{X}$, for any chosen target value $y \in \mathcal{Y}$, we can estimate $f_{Y|X=x}(y)$ the value of the probability density function evaluated on (x, y):

Prediction
(Step$_1$). Evaluate $\widetilde{q}(x, y)$.
(Step$_2$). Apply thresholding: $\widehat{q}(x, y) = \min(\widetilde{q}(x, y), 1 - \epsilon)$.
(Step$_3$). Evaluate $\widehat{f_Y}(y)$.
(Step$_4$).Plug in $f_{Y\,|\,X=x}(y)$ by applying equation 5.3 given above.

Remark that if the discriminator is a classifier, the predicted value should be the probability of class 1, *i.e.* $\mathbb{P}[Z = 1 \,|\, W = (x, y)]$.

Choice of parameters. There are 4 major choices to make when implementing our method: the marginal density estimator method, the discriminator method, the construction and the contrast ratio r.

• **Marginal density estimator.** Since the point of our method is to provide an estimation of the conditional density in cases where the marginal density is relatively easy to estimate (meaning $\mathcal{Y} \subseteq \mathbb{R}$), we pick a simple yet effective technique, the univariate kernel density estimation *KDEUnivariate* provided in the *statsmodels* package. We always keep the default parameters, *i.e.* gaussian kernels, bandwidth set using the normal reference, and the fast Fourier transform algorithm to fit the kernels.

Neural Networks architectures	Other supervised learning classifiers
MultiLayerPerceptron (MLP)	Random Forests (**R.F.**)
MLPw/o Drop-Out nor Batch-Norm (MLP:**no-D.O.**)	Elastic-net
ResNet (ResBlock) [75]	XGB[40]
Gated Linear Unit (GLU) [75]	**CatBoost** [160]
Self Normalizing Networks (SNN) [114]	**LGBM** [110])

Table 5.2: List of discriminator methods evaluated with **MCD**.

• **Marginal contrast discriminator.** We evaluate the performance of **MCD** combined with Neural Networks, Decision Tree based classifiers and Logistic Elastic-net (see table 5.4.1 for the exhaustive list). Deep Learning architectures used are described in Chapter 4.

• **Dataset construction and ratio:** We compare in our ablation study (Section 5.5) the Constructions 1, 2, 3, 4 and 5 in Frameworks 1, 2 and 3. Following the findings of the ablation study, we use Construction 2 and set $r = 0.05$ in other experiments.

5.4.2 Other benchmarked methods and Application Programming Interface (API)

There is a small number of **CDE** methods for which a readily available open source implementation in *python* exists. One notable difficulty when introducing a new **CDE** package is that there is no gold standard API in *python* on top of which new packages can build upon. Most existing implementations are standalone, with their own unique syntax for common functions. We choose to include in our benchmark the methods provided by two of the most mature *python* projects, the *freelunchtheorem github* repository by [172, 3], and a network of packages created by [48] and [158].

The *freelunchtheorem github* provides an implementation of **KMN**, **N.Flow**, **MDN** and **LSCond**. The*freelunchtheorem github* has a quite consistent API across all provided methods. Notably, [172, 3], also provide implementations for several statistical models (**EconDensity**, **ARMAJump**, **JumpDiffusion**, **LinearGauss.**, **LinearStudentT**, **SkewNormal** and **GaussianMixt**), which we include in our benchmark (see the density model section 5.4.3).

Meanwhile, the project of [48] and [158] is built around the *CDEtools github* repository and consists of several repository of varying maturity and ease of use corresponding to each method they implemented (**RFCDE**, f-**RFCDE**, **FlexCode**, **Deepcde** and **NNKCDE**). Notably, they also provide implementations in Java and R for some of these methods. Their implementation of **Deepcde** allows custom *pytorch* and $tensorflow$ architectures to be plugged-in, which gave us the opportunity to adapt the architectures and training schemes used with **MCD** to **Deepcde**. As such, the comparison between **MCD** and **Deepcde** is done on equal ground.

In total, we include in our benchmark 10 other **CDE** methods: **NNKCDE**, **N.Flow**, **LSCond**, **MDN**, **KMN**, **Deepcde**, **RFCDE**, **f-RFCDE**, **FlexCode** :**N.N.** and **FlexCode** :XGB.

Although it is not the main goal of this work, we provide an overhead over these two projects and our method to facilitate the comparison between them. Each evaluated method is encapsulated in a class which inherits the same unique API from the parent class *ConditionalEstimator*, with same input and output format and global behavior. Similar to *scikit-learn*, the estimator is an instance of the class, with hyper-parameters provided during initialisation (*__init__*), and the observation matrix and target vector provided (as *numpy* arrays) when calling the fit function. At predict time however, the estimator prediction is a function called

pdf_from_X which predicts the probability density function on a grid of target values. This package overhead allows us to compare all methods on equal ground: For density models all methods are trained on the same sampled training set. We also use the same grid of target values and test set of observations to evaluate the probability density function of all compared methods. Likewise, for real-world datasets, we use the same dataset train-test splits for all compared methods.

5.4.3 Estimation of theoretical models

We first evaluate our method on the core task it aims to handle on theoretical models: numerically estimating a conditional density function with respect to a new observation. We choose to evaluate the quality of the prediction empirically: for each predicted and target functions, we evaluate for a grid of target values the **empirical Kullback-Leibler divergence** (**K.L.**).

Density models The *freelunchtheorem* package provides 7 conditional densities implementations for which we have a function to generate a training dataset and a function to evaluate the theoretical density function on a grid of target values given an observation. These 7 models are **EconDensity** , **ARMAJump** , **JumpDiffusion** , **LinearGauss.** , **LinearStudentT** , **SkewNormal** and **GaussianMixt** . We provide here a quick description of each model, refer to the *freelunchtheorem github* documentation [3] and references therein for a more detailed description.

- **EconDensity** is an heteroscedastic model with a quadratic link used in econometrics: $X = |\epsilon|$ with $\epsilon \sim \mathcal{N}(0, 1)$ and $Y = X^2 + |\epsilon'|$ with $\epsilon' \sim \mathcal{N}(0, 1 + X)$.
- **ARMAJump** is an auto-regressive model which does not truly fall into Framework 1, since observations are not *i.i.d.*: $\forall i = 2, \cdots, n,\ x_i = c(1 - \alpha) + \alpha x_{i-1} + (1 - c)j_i$ with $(c, \alpha) \in (0, 1)^2$ and $j_i \in \mathbb{R}$ a jump component.
- **JumpDiffusion** is a market finance model where $p = 3$, X_1, X_2 and X_3 are jump diffusion processes (based on gaussian motions) and Y is a mixture of a jump diffusion process based on X and a compound Poisson process whose parameter is also dependant on X.
- **LinearGauss.** is a linear gaussian heteroscedastic model where $p = 1$, $X \sim \mathcal{U}(-1, 1)$ and $Y \sim \mathcal{N}(0.005 \times X, 0.002 \times X + 0.01)$.
- **LinearStudentT** is a student-t model where where $p = 10$ and Y follows a student-t law parametrized by X: $X \sim \mathcal{N}(\mathbf{0}_p, \mathbb{I}_p)$, $Y = u \times (0.05\overline{|X|}^2 + 0.1) +$

$0.1\overline{X}$ where u follows a student-t law of parameter $2 + 8 \times \sigma(-2\overline{X})$. Here, σ is the sigmoid function, $\overline{X}$ and $\overline{|X|}$ are the means of the components of X and $|X|$ respectively.

- **SkewNormal** is a skewnormal model where Y follows a skewnormal law parametrized by X: $X \sim \mathcal{N}(0, 1)$, $Y = u \times (0.05X^2 + 0.1) + 0.1X$ where u follows a skewnormal law with parameter $4\sigma(X)$ where σ is the sigmoid function.
- **GaussianMixt** is a standard 2-dimensional conditional gaussian mixture model with 5 kernels where the goal is to estimate the density of one component conditionally on the other.

Although these conditional density models cover a diverse set of cases, we do however introduce two other density models to illustrate the specific drawbacks of some benchmarked methods.

Model 1. *[**BasicLinear**]: Let $p = 10$ and fix p coefficients $\beta = \{\beta_j\}_{j=1,\cdots,p}$ drawn independently at random, such that $\forall j = 1, \cdots, p$, β_j is uniformly distributed over $(0, 1)$, i.e. $\beta_j \sim \mathcal{U}(0, 1)$.*

Construct now our first density model

- *Let $X \sim \mathcal{N}(\mathbf{0}_p, \mathbb{I}_p)$ be a gaussian vector.*
- *Let $Y \in \mathbb{R}$ be a random variable such that $Y = X^T\beta + \sigma\epsilon$ where $\epsilon \sim \mathcal{N}(0, 1)$ and ϵ and X independent.*

BasicLinear is a very simple linear model included to check that sophisticated methods which can estimate complex models are not outperformed in simple cases. We also add a second model, **AsymmetricLinear** , which corresponds to **BasicLinear** with a simple modification: we use asymmetric noise ($|\epsilon|$ instead of ϵ).

Model 2. *[**AsymmetricLinear**]: Let β, X and ϵ be as in **BasicLinear** 1. In our second density model, Y is as follows:*

$$Y = X^T\beta + \sigma|\epsilon|.$$

Here, $|\cdot|$ denotes the absolute value.

The major difficulty is that the support of the conditional density of Y with respect to X differs from the support of the marginal density of Y (which is $\mathcal{Y} = \mathbb{R}$) and depends on the observation X.

5.4.4 Results on density models

Evaluation protocol. We use 9 density models (section 5.4.3) as ground truth, on which we evaluate the **MCD** with various discriminators and the methods presented in section 5.4.2.To generate $\mathcal{D}_n^{X,Y}$ for each density model, we sample $n = 100$ observations and for each observation we sample one target value using the conditional law of the density model. We train all benchmarked methods on this same dataset. Next, we sample $n_{test} = 100$ observations from the density model to generate a test set. We also generate a unique grid of 10000 target values, spread uniformly on $\mathcal{Y}$. For each observation of the test set, we evaluate the true conditional density function on the grid of target values. Then, for all benchmarked method, we estimate the conditional density for each observation on that same grid of target values and evaluate the empirical Kullback-Leibler (**K.L.**) divergence defined below.

Empirical Kullback-Leibler divergence.
Set $\delta = 10^{-6}$ a numerical stability constant. Let $x \in \mathbb{R}^p$ be an observation of the test set $\mathcal{D}_{test}$ and $y \in \mathbb{R}$ be a point on $\mathcal{G}$, a grid of target values to be estimated. Then, for the evaluation of the target value $f_x(y) = \max(f_{Y|X=x}(y), \delta)$ and the predicted value $g_x(y) = \max(\widehat{f_{Y|X=x}}(y), \delta)$, we define the empirical $\mathbf{K.L.}_\delta$ divergence as follows:

$$\mathbf{K.L.} := \mathbf{K.L.}_\delta(f \,\|\, g) = \sum_{x \in \mathcal{D}_{test}} \sum_{y \in \mathcal{G}} f_x(y) \times \ln\left(\frac{f_x(y)}{g_x(y)}\right). \tag{5.4}$$

Benchmark results.
➢ We first combine the **MCD** method with a classic Multi Layer Perceptron (`MLP`) as discriminator and a kernel estimator as marginal density estimator, named **MCD** :`MLP`. Table 5.3 depicts the global performance in terms of empirical **K.L.** divergence over 9 models of **MCD** :`MLP`compared to 10 others methods, described in section 5.4.2.

- The main take away is that in 6 out of 9 cases, **MCD** :`MLP`outperforms all others.

- On 3 density models, **BasicLinear** , **LinearGauss.** and **LinearStudentT** , the **K.L.** empirical divergence is less than half of the second best method. Meanwhile, on **EconDensity** , **N.Flow** and **MCD** :`MLP`share the first place.

- When **MCD** :`MLP`is outperformed, which corresponds to **ARMAJump** and **SkewNormal** , the best performing methods are **N.Flow** and **MDN** . Note that the **ARMAJump** density model is auto-regressive, meaning samples are not actually *i.i.d.*, which is a setting outside of the scope of Construction 2. Otherwise, the second best performing method is either **N.Flow** or **NNKCDE** .

Empirical **K.L.**	**MCD** :`MLP`	**NNKCDE**	**N.Flow**	**LSCond**	**MDN**	**KMN**	**Deepcde**	**RFCDE**	**f-RFCDE**	**FlexCode** :**N.N.**	**FlexCode** :`XGB`
ARMAJump	0.573	0.929	**0.196**	0.408	0.29	0.312	1.754	0.529	0.526	1.201	2.226
AsymmetricLinear	**0.158**	0.245	0.498	0.882	0.437	0.338	0.666	0.324	0.328	0.359	0.483
BasicLinear	**0.009**	0.087	0.313	0.473	0.195	0.167	0.317	0.139	0.139	0.174	0.116
EconDensity	**0.006**	0.01	**0.006**	0.021	0.013	0.022	0.068	0.049	0.045	0.034	0.048
GaussianMixt	**0.005**	0.008	0.012	0.023	0.016	0.018	0.127	0.026	0.028	0.048	0.023
JumpDiffusion	**1.352**	1.632	4.481	9.371	4.576	5.568	22.45	10.45	10.45	6.347	4.363
LinearGauss.	**0.189**	1.742	0.868	3.15	2.318	2.892	14.71	15.88	15.88	13.75	3.571
LinearStudentT	**0.141**	0.301	6.238	9.09	3.109	3.136	7.583	2.363	2.363	1.686	0.821
SkewNormal	0.722	0.089	0.019	0.1	**0.014**	0.036	18.94	0.551	0.551	0.636	2.255

Table 5.3: Evaluation of the empirical **K.L.** divergence of different **CDE** methods, for 9 density models with $n = 100$. **Best performance** is in bold print, second best performance is underlined. Lower values are better. Column 2 corresponds to the performance of **MCD** combined with the classic `MLP`. Columns 3 to 12 show the results of the 10 other benchmarked methods.

➢ We also assess the performance of **MCD** combined with other popular supervised learning methods. Table 5.4 depicts the performance of **MCD** with various discriminators on the same benchmark of density models. The classifiers included are listed in Table 5.2 and described in section 5.4.1.

- In density model **ARMAJump** where **MCD** :`MLP`is outperformed by other methods, simply removing the Batch-Normalization and Drop-Out is sufficient to obtain the best performance.

- In density models **BasicLinear** , **EconDensity** , **GaussianMixt** , **JumpDiffusion** , **LinearGauss.** and **LinearStudentT** , the results for **MCD** :`MLP`are very close to other **N.N.** architectures performances. This means that in most cases, **MCD** does not require heavy tuning to perform well.

- The best discriminator besides **N.N.** is always either **CatBoost** or `Elastic-Net`.
- **N.N.** discriminators are outperformed by other classifiers in only one case, the **SkewNormal** density model. This corresponds to the density model included in our benchmark where all versions of **MCD** are outperformed by another method.

Empirical **K.L.**	`MLP`	`SNN`	`GLU`	`ResBlock`	**no-D.O.**	**CatBoost**	`Elastic-Net`	`XGB`	**LGBM**	**R.F.**	`XRF`
ARMAJump	0.573	1.047	0.731	0.224	**0.1**	0.241	1.09	0.598	0.953	2.265	4.469
AsymmetricLinear	0.158	**0.05**	0**0.05**	0.055	0.318	0.2	0.262	0.274	0.28	0.279	0.319
BasicLinear	0.009	**0.008**	0.01	0.009	0.108	0.059	0.083	0.113	0.098	0.094	0.106
EconDensity	0.006	0.007	**0.005**	**0.005**	0.006	0.012	0.016	0.06	0.07	0.305	0.444
GaussianMixt	**0.005**	0.006	0.006	**0.005**	0.007	0.01	0.007	0.057	0.061	0.297	0.422
JumpDiffusion	**1.352**	1.353	**1.352**	**1.352**	**1.352**	1.485	**1.352**	5.694	5.11	10.17	19.99
LinearGauss.	**0.189**	**0.189**	0.19	0.19	0.19	1.274	0.19	9.33	10.99	67.46	94.76
LinearStudentT	**0.141**	**0.141**	**0.141**	**0.141**	**0.141**	0.236	**0.141**	1.58	1.027	0.912	1.617
SkewNormal	0.722	0.796	0.669	0.356	0.155	**0.083**	0.833	0.223	0.12	5.036	7.289

Table 5.4: Evaluation of the empirical **K.L.** divergence of different **MCD** discriminators, for 9 density models with $n = 100$. **Best performance** is in bold print, second best performance is underlined. Lower values are better. Columns 2 to 6 correspond to the performance of **MCD** combined with **N.N.** architectures. Columns 7 to 12 show the results of **MCD** combined with other popular classifiers.

Impact of dimensionality.
➢ Then, we check if in cases where **MCD** outperforms all others, our method maintains its good performances when p, the number of features, changes. Table 5.5 depicts the impact of the dimensionality of X on the performances of each method in **BasicLinear**, the density model where the performance gap in our benchmark between **MCD** and other methods is the largest. Note that with **BasicLinear**, changing p modifies both the dimensionality and the signal-to-noise ratio. Indeed, the norm of the signal term $X^T\beta$ increases when p increases, which is not the case for the noise term $\sigma\epsilon$. Intuitively, increasing the signal-to-noise ratio should simplify the estimation task, as it becomes almost equivalent to estimating the expectation of Y conditionally on X. In that sense, when we increase p we progressively fall back from a low-dimensional density estimation task (*i.e.* a hard task in a simple setting) onto a high-dimensional regression problem (*i.e.* a simpler task in a difficult setting).

- Performances decrease across the board when p increases. Although **BasicLinear** is a setting where a larger p corresponds to a higher signal to noise ratio, this is not enough to counter the curse of dimensionality.
- **MCD** :MLPoutperforms all others at all ranges of p. Note that at $p = 300$, the empirical **K.L.** of **MCD** :MLPis equivalent to that of f_Y, the marginal density of the target value. This means that in this high-dimensional case, where **MCD** :MLPis not able to capture the link between X and Y, it does not overfit the training set, and instead takes a conservative approach (meaning whatever the value of $\mathbf{x}_i$ is, the estimated contrast function predicts $1/r$).
- **NNKCDE** maintains good results across the board. Meanwhile, **MCD** combined with **CatBoost** performs almost as well as **MCD** :MLPwhen $p = 3$, but its relative performances are average at best when $p = 300$.

Empirical **K.L.**	**MCD** MLP	**MCD** **CatBoost**	**NNKCDE**	**FlexCode** XGB	**KMN**	**RFCDE**	**FlexCode** **N.N.**
$p = 3$	**0.008**	0.009	*0.022*	0.093	0.067	0.037	0.094
$p = 10$	**0.036**	0.042	*0.063*	0.076	0.096	0.105	0.106
$p = 30$	**0.115**	0.202	0.154	0.196	*0.181*	0.238	0.22
$p = 100$	**0.162**	*0.224*	0.173	0.264	0.401	0.238	0.234
$p = 300$	**0.244**	0.308	*0.282*	0.302	0.304	0.507	0.253

Table 5.5: Evaluation of the **K.L.** divergence values for various feature sizes p, on the **BasicLinear** density model, with $n = 100$. **Best** performance is in bold print, second best performance is underlined, *third best* performance is between asterisks. Lower is better. Methods depicted achieve top 4 performances for at least one feature size regime.

Execution time and scalability.

➢ Next, we evaluate the scability with respect to the size of the dataset of **MCD** combined with either MLPor **CatBoost**, two classifiers known to be efficient but slow. We include as reference other methods belonging to the same categories, meaning those based on supervised learning **N.N.** and Decision Trees respectively. We also include **NNKCDE** and the methods based on variational inference, as they perform well in our benchmark. Table 5.6 depicts the computation time of **MCD** and other **CDE** methods for $n = 30, 100, 300$ and 1000, on **BasicLinear** , the density model where it is most beneficial to use **MCD** instead of another method. The reported computation times include all steps (initialization, training, prediction). For **MCD** , this includes both the time taken by the discriminator and the time taken by the estimator. Note also that for **MCD** , we use the ratio $r = 0.05$, meaning the actual training set size is 20 times larger.

- The main take away is that **MCD** mostly scales like its discriminator would in a supervised learning setting.
- Regarding methods based on supervised learning with neural networks, we compare **MCD** :`MLP`with **FlexCode** :**N.N.** and **Deepcde** . **FlexCode** :**N.N.** is much faster than **Deepcde** and **MCD** :`MLP`. When using equivalent architectures, **Deepcde** is slower than **MCD** when $n = 1000$ (which corresponds to $N = \frac{n}{r} = 20000$ for **MCD**). This can be partly explained by the fact that **Deepcde** uses a transformation of the target which corresponds to an output layer width of 30, while **MCD** increases the input size of the network by 1, since observations and target values are concatenated.
- Regarding methods based on supervised learning with decision trees, we compare **MCD** :**CatBoost** with **RFCDE** and **FlexCode** :`XGB`. **RFCDE** is the only method which does not leverage *GPU* acceleration, yet, it is faster than **MCD** :**CatBoost** and **FlexCode** :`XGB`.
- Among the best performing methods in our benchmark, **NNKCDE** is by far the fastest. Meanwhile, the 3 variational inference methods included, **MDN** , **KMN** and **N.Flow** , are much slower, which is to be expected.

Category	Decision Tree Based Methods			**N.N.** based Methods			Kernel	Variational inference		
Method	**RFCDE**	**MCD**	**FlexCode**	**FlexCode**	**MCD**	**Deepcde**	**NNKCDE**	**MDN**	**N.Flow**	**KMN**
Based on:	**R.F.**	**CatBoost**	`XGB`	**N.N.**	`MLP`	`MLP`	N.Neighbor	**N.N.**	**N.N.**	**N.N.**
$n = 30$	0.044	0.46	9.574	0.018	3.133	1.943	0.01	35.46	50.74	47.92
$n = 100$	0.149	0.369	9.57	0.017	3.149	2.11	0.022	35.45	50.72	102.7
$n = 300$	0.448	0.446	13.19	0.017	3.132	3.225	0.04	35.65	50.62	150.6
$n = 1000$	0.952	0.689	25.24	0.02	3.296	6.388	0.043	35.73	51.12	143.6

Table 5.6: Training Time in seconds for various training set sizes n, on the **BasicLinear** density model. Row 1 corresponds to the method category, row 2 corresponds to the benchmarked **CDE** method and row 3 corresponds to the underlying supervised learning method used. Column 1 corresponds to the training set size. Columns 2, 3 and 4 correspond to methods which leverage a supervised learning method based on Decision trees. Columns 5, 6 and 7 correspond to methods which leverage a supervised learning method based on **N.N.** . Column 8 corresponds to nearest neighbors kernels. Columns 9, 10 and 11 correspond to three variational inference methods.

5.4.5 Real-world datasets

Dataset origins and methodology.

➢ We include in our benchmark 12 datasets, taken from two sources:

- The *CDEtools* framework provides the dataset "Teddy" (see [14] for a description). We follow the pre-processing used in the given packages.
- The *freelunchtheorem* framework provides 2 toy datasets, 7 datasets from the UCI ([9] repository, and one dataset from the kaggle plateform [2] which includes 2 targets, for a total of 11 datasets. Here again, we follow the pre-processing used in the given package.

Evaluation protocol.
➢ For all datasets, we standardize the observations and target values, then we perform the same train-test split for all methods benchmarked (manually setting the random seed for reproductibility). Because datasets are of varying sizes and as some methods are extremely slow for large datasets, we only take a subset of observations to include in the training set, doing the split as such. Let $n_{\max}$ be the original size of the dataset, the trainset is of size $n = \min(300, \lfloor n_{\max} \times 0.8 \rfloor)$ and the test set is of size $n_{test} = \min(300, n_{\max} - n)$.
One challenge when benchmarking **CDE** methods is that real-world datasets almost never provide the conditional density function associated to an observation, but instead only one realisation. This means that the usual metrics for **CDE** (eg.: **K.L.**) cannot be evaluated on these datasets. We nonetheless evaluate our method on real-world datasets, using the negative log-likelihood metric, denoted NLL , to compare the estimated probability density function against the target value.

Empirical Negative Log-likelihood.
Set $\delta = 10^{-6}$ a numerical stability constant. Let $(x, y) \in \mathbb{R}^p \times \mathbb{R}$ be a sample from the test set $\mathcal{D}_{test}$, and $g_x(y) = \max(\widehat{f_{Y\,|\,X=x}}(y), \delta)$ be the predicted value, we define the NLL metric as follows:

$$\text{NLL} := \text{NLL}_\delta(g) = -\sum_{(x,y)\,\in\,\mathcal{D}_{test}} \ln\left(g_x(y)\right). \tag{5.5}$$

Results.
➢ Table 5.7 depicts the global performance in terms of negative log-likelihood for the 12 datasets.

- The main take away is that this time, **MCD** :`MLP`outperforms existing methods in 7 out of 12 cases, including the popular datasets BostonHousing and Concrete.
- On the WineRed and WineWhite datasets, **MCD** lags far behind **NNKCDE** and **FlexCode** . One possible cause is that for these two datasets,

the target variable Y takes discrete values: $\mathcal{Y} = \{3, 4, 5, 6, 7, 8\}$, which does not correspond to the regression task for which **MCD** was designed.

- Besides **MCD**, the top performing methods are **NNKCDE**, **N.Flow**, **FlexCode** (both versions) and **KMN**.
- Regarding the choice of the discriminator, **MCD**:`MLP`is not outperforming **MCD**:**CatBoost** to the same extent it does on density models. The difference in negative log-likelihood between **MCD**:`MLP`and **MCD**:**CatBoost** is below 0.1 in 7 out of 12 cases.
- Besides, **MCD**:**CatBoost** obtains Top 2 performances in 3 of the 5 cases where **MCD**:`MLP` is outperformed by other methods. Notably, on the Yacht dataset where **MCD**:`MLP` performs poorly, **MCD**:**CatBoost** outperforms all others by a wide margin. This indicates that this time, `MLP` and **CatBoost** are complementary, as together they can obtain at least Top 2 performances in all cases besides the WineRed and WineWhite datasets.

Empirical NLL	**MCD** `MLP`	**MCD CatBoost**	**NNKCDE**	**FlexCode N.N.**	**FlexCode** `XGB`	**KMN**	**N.Flow**
BostonHousing	-0.64	**-0.59**	-0.81	-1.99	-1.84	-1.22	-1.63
Concrete	**-0.86**	-1.02	-1.23	-2.26	-1.25	-2.30	-2.13
NCYTaxiDropoff:lon.	-1.30	**-1.28**	-1.68	-2.05	-2.17	-2.51	-2.85
NCYTaxiDropoff:lat.	**-1.31**	**-1.31**	-1.44	-1.67	-6.18	-2.45	-2.96
Power	**-0.06**	-0.36	-0.73	-1.09	-0.75	-0.35	-0.39
Protein	**-0.09**	-0.42	-0.77	-0.83	-1.38	-0.68	-0.54
WineRed	-0.89	-0.89	**3.486**	1.062	0.965	-0.90	-2.43
WineWhite	-1.18	-1.13	**2.99**	-0.73	-0.63	-1.7	-4.18
Yacht	0.14	**0.822**	-0.46	-1.23	0.144	0.025	0.401
teddy	**-0.47**	-0.51	-0.83	-0.83	-1.34	-0.76	-0.94
toy dataset 1	-0.99	-0.47	-0.63	-0.88	-1.46	**-0.35**	-0.71
toy dataset 2	-1.40	-1.33	**-1.31**	-1.54	-1.43	-1.33	-1.39

Table 5.7: Evaluation of the negative log-likelihood (NLL) for 12 datasets. **Best performance** is in bold print, second best performance is underlined. Higher values are better. Methods included outperform all others besides **MCD** on at least one dataset.

5.5 Ablation

We now present an ablation study of the impact of the construction strategy used to build $\mathcal{D}_N^{W,Z}$ and the chosen value for ratio r. We also assess the impact of additional data on performances in Framework 2 and 3. Our experiments are done on the **AsymmetricLinear** density model, which corresponds to a case where **MCD** is performing well but there is still room for improvement.

- The main take away is that in Framework 1, $i.d.$-Construction 2 is far better than $i.i.d.$-Construction 1.
- The appropriate ratio r for the $i.d.$-Construction 2 should be around 0.05, meaning $N = 20 \times n$.
- In Framework 3, as soon as two target values are associated to each observation, the $i.d.$-Construction 5 can massively improve the performances of **MCD** , which are already very good when using $i.d.$-Construction 2.

Construction strategy and ratio r

➢ Table 5.8 depicts the impact of the construction strategy and ratio r on the performance in terms of empirical **K.L.** divergence in the classical Framework 1 on the **AsymmetricLinear** density model.

- The appropriate ratio r for the $i.i.d.$-Construction 1 is 0.5 which corresponds to a balanced distribution between the two classes.
- It seems clear that the $i.d.$-Construction 2 produces much better results than the $i.i.d.$-Construction 1. For the latter, the performances are worse than those obtained with the concurrent method **NNKCDE** (see Table 5.3: **K.L.** =0.245). On the other hand, the $i.d.$-Construction 2 obtains Top 1 performances on our benchmark, by a wide margin.
- For the $i.d.$-Construction 2, it seems preferable to choose a ratio of 0.15 or 0.05, which corresponds respectively to a 6 or 20 times larger dataset. Indeed, in that case since $N = \frac{n}{r}$, we have to make a trade-off between the size of the training dataset and the imbalance between the classes.
- Following these findings, in our benchmark, we choose to use the $i.d.$-Construction 2 with a ratio of 0.05.

Additional marginal data

➢ Table 5.9 depicts the impact on **K.L.** performance when using Constructions 3 and 4 (corresponding to the $i.i.d.$ and $i.d.$ case respectively) in Framework 2 where additional marginal data is available. We compare the respective benefit of adding only marginal observations ($n_x > 0$, $n_y = 0$), only marginal target values ($n_x = 0$, $n_y > 0$), and both marginal observation and marginal target values ($n_x > 0, n_y > 0$).

Construction	Ratio r	N	**K.L.**
i.i.d.-Construction 1	0.05	50	0.3284
	0.15	50	0.2865
	0.5	50	**0.2771**
	0.85	50	0.4211
i.d.-Construction 2	0.01	10000	0.0563
	0.015	6666	**0.0546**
	0.05	2000	0.0550
	0.15	666	0.0551
	0.5	200	0.0996

Table 5.8: Evaluation of the **K.L.** divergence of the **MCD** :`MLP`method on the **AsymmetricLinear** density model in Framework 1 with various values for ratio r, with *i.i.d.* and *i.d.* constructions. **Best** performance for each construction is in bold print.

- In the *i.i.d.* case, using *i.i.d.*-Construction 3 instead of *i.i.d.*-Construction 1 produces substantial improvements in terms of performances, but not enough to outperform the *i.d.*-Construction 2. When using the *i.d.*-Construction 4 instead of the *i.d.*-Construction 2, the performance gain is smaller, but bear in mind that performances are already very satisfying at that point.

- In the *i.i.d.* case, having access to $n_y = n$ marginal target values also allows the marginal estimator to be trained on a sample size of $n_y + n = 2n$, which may explain why the performance gain is larger with marginal target values $\mathcal{D}^Y_{n_y}$ than with marginal observations $\mathcal{D}^X_{n_x}$ when using the *i.i.d.*-Constructions 3.

- Meanwhile, in the *i.d.*case, the size of the training dataset $\mathcal{D}^{W,Z}_N$ when $\max(n_x, n_y) > 0$ is at most $(n + n_x)(n + n_y) > n^2$, which allows to build an even larger data set ($N > n^2$). However, in that case, the choice of N is constrained by the choice of ratio r: $N = \frac{n}{r}$. Here the appropriate ratio is $r = 0.05$ and $n = 100$, meaning $N = 2000 \leq n^2$. As such, using Construction 4 to increase N beyond n^2 is not useful in that setting.

- The Construction 4 does, however, increase the amount of information present in the dataset $\mathcal{D}^{W,Z}_N$. Here, the performance gain is higher in the presence of marginal observations $\mathcal{D}^X_{n_x}$ than marginal target values $\mathcal{D}^Y_{n_y}$. This may be partly due to the fact that in the **AsymmetricLinear** model, $X \in \mathbb{R}^{10}$ and $Y \in \mathbb{R}$, meaning the observations contain more information than the target values.

Multiple target values per observations

➢ Table 5.10 compares the **K.L.** performances of *i.d.*-Construction 5 in

Setting				Construction			Results
Available datasets	n	n_x	n_y	Strategy	Construction	N	**K.L.**
$\mathcal{D}_n^{X,Y}$	100	0	0		Construction 1	50	0.3456
$\mathcal{D}_n^{X,Y}$, $\mathcal{D}_{n_x}^{X}$	100	100	0	*i.i.d.*	Construction 3	100	**0.1204**
$\mathcal{D}_n^{X,Y}$, $\mathcal{D}_{n_y}^{Y}$	100	0	100	$r = 0.5$	Construction 3	100	**0.1203**
$\mathcal{D}_n^{X,Y}$, $\mathcal{D}_{n_x}^{X}$, $\mathcal{D}_{n_y}^{Y}$	100	25	25		Construction 3	75	0.2058
$\mathcal{D}_n^{X,Y}$	100	0	0		Construction 2	2000	0.0551
$\mathcal{D}_n^{X,Y}$, $\mathcal{D}_{n_x}^{X}$	100	500	0	*i.d.*	Construction 4	2000	**0.0541**
$\mathcal{D}_n^{X,Y}$, $\mathcal{D}_{n_y}^{Y}$	100	0	500	$r = 0.05$	Construction 4	2000	0.0546
$\mathcal{D}_n^{X,Y}$, $\mathcal{D}_{n_x}^{X}$, $\mathcal{D}_{n_y}^{Y}$	100	150	150		Construction 4	2000	0.0639

Table 5.9: Evaluation of the **K.L.** divergence values of the **MCD** :`MLP`method on the **AsymmetricLinear** density model in Framework 1 and 2 with various values for n, n_x and n_y, with *i.i.d.* and *i.d.* constructions. **Best** performance for *i.i.d.* and *i.d.* are in bold print.

Framework 3, when more than one target is associated to each observation in the dataset $\mathcal{D}_{n,m}^{X,\mathbb{Y}}$. Here we denote m the number of observations associated to each target. Remark that when $m = 1$, *i.d.*-Construction 5 is strictly equivalent to *i.d.*-Construction 2.

- In the presence of multiple target values per observation, *i.d.*-Construction 5 allows for unparalleled performances. This can probably be explained by the fact that the goal of the **CDE** task is to determine the relationship between X and Y beyond the conditional expectation, and thus multiple realizations for a single observation better quantify the variance.
- Besides, it seems that when $m > 1$, the appropriate ratio is higher, since the performances for $r = 0.15$ are better than with $r = 0.05$, which is not the case when $m = 1$.

5.6 Proofs and dataset constructions

5.6.1 Proof Fact 1

Let $(x, y) \in \mathcal{X} \times \mathcal{Y}$ be any couple of values for which we need to prove Fact 1. We have by the *Bayes*'s formula:

$$\frac{f_{Y\,|\,X=x}(y)}{f_Y(y)} = \frac{f_{X,Y}(x,y)}{f_X(x) f_Y(y)},\ f_X(x) > 0,\ f_Y(y) > 0 \tag{5.6}$$

Construction	m	Ratio r	N	**K.L.**
i.d.-Construction 2	1	0.5	200	0.0620
		0.15	666	0.0502
		0.05	2000	0.0501
i.d.-Construction 5	2	0.5	400	0.0520
		0.15	1333	0.0438
		0.05	4000	0.0461
i.d.-Construction 5	10	0.5	2000	0.0179
		0.15	6666	**0.0167**
		0.05	20000	0.0191

Table 5.10: Evaluation of the **K.L.** divergence values of the **MCD** :`MLP`method on the **AsymmetricLinear** density model in Framework 1 and 3 with various values for ratio r and m. **Best** performance is in bold print. Best performance ratio for each value of m is underlined. Column 2 corresponds to the number of target values associated to each observation.

Then, for any $r \in (0, 1)$:

$$\frac{f_{X,Y}(x,y)}{f_X(x)f_Y(y)} = \frac{rf_{X,Y}(x,y)}{rf_{X,Y}(x,y) + (1-r)f_X(x)f_Y(y)} \times \frac{rf_{X,Y}(x,y) + (1-r)f_X(x)f_Y(y)}{rf_X(x)f_Y(y)}$$

Let $q(x,y) := \frac{rf_{X,Y}(x,y)}{rf_{X,Y}(x,y)+(1-r)f_X(x)f_Y(y)}$ the marginal constrast function with ratio r defined on Definition 5

$$\begin{aligned} \frac{f_{X,Y}(x,y)}{f_X(x)f_Y(y)} &= q(x,y) \times \left(\frac{f_{X,Y}(x,y)}{f_X(x)f_Y(y)} + \frac{1-r}{r} \right) \\ \Leftrightarrow (1-q(x,y))\frac{f_{X,Y}(x,y)}{f_X(x)f_Y(y)} &= q(x,y)\frac{1-r}{r} \\ \Leftrightarrow \frac{f_{Y|X=x}(y)}{f_Y(y)} &= \frac{1-r}{r}\frac{q(x,y)}{1-q(x,y)} \end{aligned}$$

The last equation is deduced using equation (5.6).

5.6.2 Proof Proposition 1

By condition (**Cd 1**) of Definition 6 we have $Z \sim \mathcal{B}(r)$, then

$$\mathbb{P}[Z = 1] = r \text{ and } f_Z(z) = p^z(1-p)^{1-z}.$$

Moreover, by condition (**Cd 3**) of Definition 6, we have $\forall (x,y) \in \mathcal{X}\times\mathcal{Y}$, $f_W(x,y) = rf_{X,Y}(x,y) + (1-r)f_X(x)f_Y(y)$. By condition (**Cd 4**) of Definition 6 we have

$$f_{W\,|\,Z=1}(x,y) = f_{X,Y}(x,y)$$
$$f_{W\,|\,Z=0}(x,y) = f_X(x)f_Y(y)$$

By Definition 5 we have

$$\begin{aligned}
q(x,y) &= \frac{r\times f_{X,Y}(x,y)}{rf_{X,Y}(x,y)+(1-r)f_X(x)f_Y(y)} = \frac{\mathbb{P}[Z=1]\times f_{W\,|\,Z=1}(x,y)}{rf_{W\,|\,Z=1}(x,y)+(1-r)f_{W\,|\,Z=0}(x,y)} \\
&= \frac{\mathbb{P}[Z=1]\times f_{W\,|\,Z=1}(x,y)}{f_W(x,y)} = \frac{\mathbb{E}_Z[\mathbb{1}_{Z=1}]\times\left[1\times f_{W\,|\,Z=1}(x,y)+0\times f_{W\,|\,Z=0}(x,y)\right]}{f_W(x,y)} \\
&= \frac{\mathbb{E}_Z[\mathbb{1}_{Z=1}]\times\mathbb{E}_Z[f_{W\,|\,Z=z}]}{f_W(x,y)} = \frac{\mathbb{E}_Z[\mathbb{1}_{Z=1}f_{W\,|\,Z=z}(x,y)]}{f_W(x,y)} = \frac{\mathbb{E}_Z[\mathbb{1}_{Z=1}f_{W,Z}(x,y,z)]}{f_W(x,y)f_Z(z)} \\
&= \mathbb{E}_Z\left[\mathbb{1}_{Z=1}\times\frac{f_{Z\,|\,W=(x,y)}(z)}{f_Z(z)}\right] = \mathbb{E}_Z\left[\mathbb{1}_{Z=1}\times\frac{\mathbb{P}\left[Z=z\,|\,W=(x,y)\right]}{\mathbb{P}[Z=z]}\right] \\
&= \mathbb{E}_Z\left[\mathbb{1}_{Z=1}\times\frac{\mathbb{P}\left[Z=1\,|\,W=(x,y)\right]}{\mathbb{P}[Z=1]}\right] = \mathbb{E}_Z[\mathbb{1}_{Z=1}]\frac{\mathbb{P}[Z=1\,|\,W=(x,y)]}{\mathbb{P}[Z=1]} \\
&= \mathbb{P}[Z=1\,|\,W=(x,y)] = \mathbb{E}[Z\,|\,W=(x,y)]
\end{aligned}$$

5.6.3 Proof Theorem 1 and Construction in the *i.i.d.* case

First construct a random vector (W,Z) satisfying the MDcond(r) with $r\in(0,1)$.

- Consider the random vector (X,Y) admitting $f_{X,Y}$ as density of probability.
- Let $\widetilde{Y}$ be a random variable independent of X and Y and of same law f_Y of Y ($\widetilde{Y}\overset{\mathcal{D}}{=}Y$).
- Let r be a real number in $(0,1)$ and $Z\sim\mathcal{B}(r)$.
- Set $W=(X, Y\mathbb{1}_{Z=1}+\widetilde{Y}\mathbb{1}_{Z=0})$.

Then, for all $(x,y,\widetilde{y})\in\mathcal{X}\times\mathcal{Y}\times\mathcal{Y}$ we have

$$\forall z\in\{0;1\}\, f_{W\,|\,Z=z}(x,y,\widetilde{y}) = \begin{cases} f_{X,Y}(x,y) & \text{if } z=1 \\ f_X(x)f_Y(\widetilde{y}) & \text{if } z=0 \end{cases}$$

Therefore, $\forall (x,y,z) \in \mathcal{X}\times\mathcal{Y}\times\{0;1\}$, $f_{W\,|\,Z=z}(x,y) = f_{X,Y}(x,y)\mathbb{1}_{z=1} + f_X(x)f_Y(y)\mathbb{1}_{z=0})$.

Moreover, $\forall (x,y) \in \mathcal{X} \times \mathcal{Y}$

$$\begin{aligned} f_W(x,y) &= \int f_{W\,|\,Z=z}(x,y) f_Z(z)\,dz \\ &= f_{W\,|\,Z=1}(x,y)\mathbb{P}[Z=1] + f_{W\,|\,Z=0}(x,y)\mathbb{P}[Z=0] \\ &= f_{W\,|\,Z=1}(x,y) r + f_{W\,|\,Z=0}(x,y)(1-r) \\ &= r f_{X,Y}(x,y) + (1-r) f_X(x) f_Y(y). \end{aligned}$$

Therefore Z, W satisfies Definition 6. Now, consider the original n-sample $\mathcal{D}_n^{X,Y}$ and set $N = \lfloor n/2 \rfloor$. We can now construct the $\mathcal{D}_N^{W,Z}$ sample.

Construction 1. [$i.i.d.$ $\mathcal{D}_N^{W,Z}$]

Consider the original n-sample $\mathcal{D}_n^{X,Y}$ and set $N = \lfloor n/2 \rfloor$. We can now construct the $\mathcal{D}_N^{W,Z}$ sample:

Step (1): First sample N independent observations $Z_1, \cdots, Z_N$ with respect to $\mathcal{B}(r)$.

Step (2): Next, $\forall i = 1, \cdots, N$, set $W_i = (X_i, Y_i \mathbb{1}_{Z_i=1} + Y_{i+N} \mathbb{1}_{Z_i=0})$.

Since $N > 0$ and the N couples $(W_i, Z_i)_i$ are constructed from the N-$i.i.d.$ quadruplets $(X_i, Y_i, Y_{i+N}, Z_i)_i$, they are $i.i.d.$.

5.6.4 Proof Theorem 2 and Construction in the $i.d.$ case

Let n_J and n_M two integers such that $1 \le n_J \le n$ and $1 \le n_M \le n(n-1)$.

Construction 2. [$i.d.$ $\mathcal{D}_N^{W,Z}$]

Step (1): Construct n^2 observations $\{(\widetilde{W}_i, \widetilde{Z}_i)\}_{i=1,\cdots,n^2}$ such that: $\forall j = 0, \cdots, n-1$, $\forall k = 1, \cdots, n$:

$$\begin{cases} \widetilde{W}_{jn+k} = (X_{j+1}, Y_k) \\ \widetilde{Z}_{jn+k} = \mathbb{1}_{j+1=k} \end{cases} \tag{5.7}$$

Note that the n^2 observations are neither independent nor identically distributed and do not satisfy yet the MDcond.

Step (2) : Split the n^2 couple of observations $\{(\widetilde{W}_i, \widetilde{Z}_i)\}_{i=1,\cdots,n^2}$ into two:

$$\begin{cases} S_J = \{(\widetilde{W}_i, \widetilde{Z}_i)\} : \widetilde{Z}_i = 1\, \forall i \\ S_M = \{(\widetilde{W}_i, \widetilde{Z}_i)\} : \widetilde{Z}_i = 0\, \forall i. \end{cases}$$

Step (3) : Sample at random uniformly without replacement n_J observations $(\widetilde{W}_i, \widetilde{Z}_i)$ from S_J and denote $\widetilde{S}_J$ this n_J sample.

Note that since $\widetilde{Z}_i = 1$ if and only if $j+1 = k$ in equation(5.7), we have

$$\forall (\widetilde{W}_i, \widetilde{Z}_i) \in \widetilde{S}_J, \widetilde{Z}_i = 1 \text{ and } f_{\widetilde{W}_i \,|\, \widetilde{Z}_i = 1} \equiv f_{X,Y} \tag{5.8}$$

This means that $\widetilde{W}_{j(n+1)+1} = (X_{j+1}, Y_{j+1})$.

Step (4) : Sample at random uniformly without replacement n_M observations $(\widetilde{W}_i, \widetilde{Z}_i)$ from S_M and denote $\widetilde{S}_M$ this n_M sample.

Note that since $\widetilde{Z}_i = 0$ if and only if $j+1 \neq k$ in equation(5.7), we have

$$\forall (\widetilde{W}_i, \widetilde{Z}_i) \in \widetilde{S}_M, \widetilde{Z}_i = 0 \text{ and } f_{\widetilde{W}_i \,|\, \widetilde{Z}_i = 0} \equiv f_X f_Y \tag{5.9}$$

This means that $\widetilde{W}_{jn+k} = (X_{j+1}, Y_k)$ (recall $X_l \perp\!\!\!\perp X_k\, \forall l \neq k$).

Step (5) : Concatenate the samples $\widetilde{S}_J$ and $\widetilde{S}_M$ and shuffle uniformly to obtain a N sample $\mathcal{D}_N^{W,Z} = \mathcal{U}\texttt{nif}.\mathcal{S}\texttt{huffle}\Big(\{\widetilde{S}_J; \widetilde{S}_M\}\Big)$.

Prove now that $\forall i = 1, \cdots, N$, the couple $(W_i, Z_i) \in \mathcal{D}_N^{W,Z}$ satisfies MDcond($\frac{n_J}{N}$).

((**Cd 1**)) As we shuffle uniformly the indices, we have $\forall i = 1, \cdots, N$, $Z_i \in \mathcal{D}_N^{W,Z}$, $Z_i \sim \mathcal{B}(\frac{n_J}{N})$.

((**Cd 2**)) By **Step (1)**, it is obvious that all the W_i admit $\mathcal{X} \times \mathcal{Y}$ as support.

((**Cd 4**)) Let (W_i, Z_i) be any element of $\mathcal{D}_N^{W,Z}$, then by equation (5.8)

$$\begin{cases} f_{W_i \,|\, Z_i = 1} \equiv f_{X,Y} & \text{if } Z_i = 1 \\ f_{W_i \,|\, Z_i = 0} \equiv f_X f_Y & \text{if } Z_i = 0 \end{cases}$$

((**Cd 3**)) Moreover, it comes $f_{W_i}(x,y) = r f_{X,Y}(x,y) + (1-r) f_X f_Y$ with $r = \frac{n_J}{N}$.

5.6.5 Proof Theorem 3 and *i.i.d.* Construction in the Additional data setting

To construct $\mathcal{D}_N^{W,Z} = \{(W_i, Z_i)\}_{i=1,\cdots,N}$ a training set of N *i.i.d.* observations, we concatenate 3 datasets denoted $\mathcal{D}_{|N_X}^{W,Z}$, $\mathcal{D}_{|N_Y}^{W,Z}$ and $\mathcal{D}_{|N_{X,Y}}^{W,Z}$ of respective size N_X, N_Y and

$N_{X,Y}$ such that

$$\mathcal{D}_N^{W,Z} = \mathcal{D}_{|N_X}^{W,Z} \cup \mathcal{D}_{|N_Y}^{W,Z} \cup \mathcal{D}_{|N_{X,Y}}^{W,Z}$$

$$\text{and } \begin{cases} N_X = \min(n, n_x), & N_Y = \min(n_y, n - N_X) \\ N_{X,Y} = \left\lfloor \frac{n - N_X - N_Y}{2} \right\rfloor, & N = N_X + N_Y + N_{X,Y} \end{cases}$$

To construct these 3 preliminary datasets, we first split

$$\mathcal{D}_n^{X,Y} = \underbrace{\mathcal{D}_{|2N_{X,Y}}^{X,Y}}_{\text{the first } 2N_{X,Y} \text{ obs.}} \cup \underbrace{\mathcal{D}_{|N_Y}^{X,Y}}_{\text{the next } N_Y \text{ obs.}} \cup \underbrace{\mathcal{D}_{|N_X}^{X,Y}}_{\text{the next } N_X \text{ obs.}} \cup \underbrace{\mathcal{D}}_{\text{the rest.}}$$

We consider $\mathcal{D}_{|N_X}^{X}$ and $\mathcal{D}_{|N_Y}^{Y}$ the observations in datasets $\mathcal{D}_{n_x}^{X}$ and $\mathcal{D}_{n_y}^{Y}$ restricted to the N_X and N_Y first observations respectively.

Construction 3. [*i.i.d.* $\mathcal{D}_N^{W,Z}$ *Additional data*]

Step (1): **Construction of** $\mathcal{D}_{|N_{X,Y}}^{W,Z}$. If $N_{X,Y} = 0$, then $\mathcal{D}_{|N_{X,Y}}^{W,Z} = \emptyset$, otherwise we construct $\mathcal{D}_{|N_{X,Y}}^{W,Z}$ from the initial dataset $\mathcal{D}_{|N_X}^{X}$ as described in Construction 1.

Step (2): **Construction of** $\mathcal{D}_{|N_Y}^{W,Z}$.First note that if $N_Y = 0$, then $\mathcal{D}_{|N_Y}^{W,Z} = \emptyset$. If $N_Y \neq 0$, we proceed as follows:

(i) Sample N_Y independent observations $(Z_i)_{i=1,\cdots,N_Y}$ according to a Bernoulli law of parameter $r \in (0,1)$.

(ii) For all $i = 1, \cdots, N_Y$; $\forall (X_i, Y_i) \in \mathcal{D}_{|N_Y}^{X,Y}$ and $\forall \widetilde{Y}_i \in \mathcal{D}_{|N_Y}^{Y}$, set

$$W_i = (X_i, Y_i \mathbb{1}_{Z_i=1} + \widetilde{Y}_i \mathbb{1}_{Z_i=0}).$$

Note that by construction, $(W_i, Z_i)_{i=1}^{N_Y}$ are *i.i.d.* and follow $f_{W,Z}$. Moreover, following the same arguments as in proof of Theorem 1, the $\{(W_i, Z_i)\}_{i=1}^{N_Y}$ satisfy the MDcond(r).

Step (3): **Construction of** $\mathcal{D}_{|N_X}^{W,Z}$.

(i) Sample N_X independent observations $(Z_i)_{i=1,\cdots,N_X}$ according to a Bernoulli law of parameter $r \in (0,1)$.

(ii) For all $i = 1, \cdots, N_X$; $\forall (X_i, Y_i) \in \mathcal{D}_{|N_X}^{X,Y}$ and $\forall \widetilde{X}_i \in \mathcal{D}_{|N_X}^{X}$, set

$$W_i = (X_i \mathbb{1}_{z=1} + \widetilde{X}_i \mathbb{1}_{z=0}, Y_i)$$

.

Note that, by construction $\mathcal{D}^{W,Z}_{|N_X} = (W_i, Z_i)_{i=1}^{N_X}$ are *i.i.d.* and follow $f_{W,Z}$. Indeed, the reasoning is similar as in proof of Theorem 1. First construct a random vector (W, Z) satisfying the MDcond(r) with $r \in (0, 1)$:

- Let (X, Y) be a random variable admitting $f_{X,Y}$ as probability density function.
- Let Z be a random variable following a Bernoulli of parameter $r \in (0, 1)$.
- Let $\widetilde{X}$ be a random variable independent of (X, Y) following the law f_X.
- Set $W = (X_i \mathbb{1}_{Z_i=1} + \widetilde{X}_i \mathbb{1}_{Z_i=0}, Y_i)$.

Then, $\forall (x, \widetilde{x}, y) \in \mathcal{X} \times \mathcal{Y} \times \mathcal{Y}$ we have $\forall z \in \{0, 1\}$,

$$g_{W\,|\,Z=z}(x, \widetilde{x}, y) = \begin{cases} f_{X,Y}(x, y) \text{ if } z = 1 \\ f_X(\widetilde{x}) f_Y(y) \text{ if } z = 0 \end{cases}$$

Then $\forall (x, y, z) \in \mathcal{X} \times \mathcal{Y} \times \{0, 1\}$, it comes

$$f_{W\,|\,Z=z}(x, y) = g_{W\,|\,Z=z}(x, x, y) = f_{X,Y}(x, y)\mathbb{1}_{z=1} + f_X(x) f_Y(y)\mathbb{1}_{z=0}.$$

Moreover, $\forall (x, y) \in \mathcal{X} \times \mathcal{Y}$; we have by equation (5.8):

$$f_W(x, y) = r f_{X,Y}(x, y) + (1 - r) f_X(x) f_Y(y).$$

So (W, Z) satisfies the MDcond(r).

Step (4): **Concatenation.** Concatenate the 3 datasets $\mathcal{D}^{W,Z}_{|N_X}$, $\mathcal{D}^{W,Z}_{|N_Y}$ and $\mathcal{D}^{W,Z}_{|N_{X,Y}}$

To conclude our proof, note that :

• Since $\mathcal{D}^{X,Y}_{|2N_{X,Y}}$, $(\mathcal{D}^{X,Y}_{|N_X}, \mathcal{D}^{X}_{|N_X})$ and $(\mathcal{D}^{X,Y}_{|N_Y}, \mathcal{D}^{Y}_{|N_Y})$ are independent, the datasets $\mathcal{D}^{W,Z}_{|N_X}$, $\mathcal{D}^{W,Z}_{|N_Y}$ and $\mathcal{D}^{W,Z}_{|N_{X,Y}}$ are independent by construction.

• Moreover, since $\mathcal{D}^{W,Z}_{|N_X}$, $\mathcal{D}^{W,Z}_{|N_Y}$ and $\mathcal{D}^{W,Z}_{|N_{X,Y}}$ are composed by *i.i.d.* random variables following the law $f_{W,Z}$, the sample of size $N = N_{X,Y} + N_X + N_Y$,

$$\mathcal{D}^{W,Z}_{N} = \mathcal{D}^{W,Z}_{|N_X} \cup \mathcal{D}^{W,Z}_{|N_Y} \cup \mathcal{D}^{W,Z}_{|N_{X,Y}}$$

is composed by *i.i.d.* random variables following the law $f_{W,Z}$.

Construction of $\mathcal{D}^{W,Z}_{|N_{X,Y}}$.

If $N_{X,Y} = 0$, then $\mathcal{D}^{W,Z}_{|N_{X,Y}} = \emptyset$, otherwise we construct $\mathcal{D}^{W,Z}_{|N_{X,Y}}$ from the initial dataset $\mathcal{D}^{X}_{|N_X}$ as described in Construction 1 (see the proof of Theorem 1).

Construction of $\mathcal{D}^{W,Z}_{|N_Y}$.

First note that if $N_Y = 0$, then $\mathcal{D}^{W,Z}_{|N_Y} = \emptyset$. If $N_Y \neq 0$, we proceed as follows:

(i) Sample N_Y independent observations $(Z_i)_{i=1,\cdots,N_Y}$ according to a Bernoulli law of parameter $r \in (0, 1)$.

(ii) For all $i = 1, \cdots, N_Y$; $\forall (X_i, Y_i) \in \mathcal{D}^{X,Y}_{|N_Y}$ and $\forall \widetilde{Y}_i \in \mathcal{D}^{Y}_{|N_Y}$, set

$$W_i = (X_i, Y_i \mathbb{1}_{Z_i=1} + \widetilde{Y}_i \mathbb{1}_{Z_i=0}).$$

Note that by construction, $(W_i, Z_i)_{i=1}^{N_Y}$ are *i.i.d.* and follow $f_{W,Z}$. Moreover, following the same arguments as in proof of Theorem 1, the $\{(W_i, Z_i)\}_{i=1}^{N_Y}$ satisfy the MDcond(r).

Construction of $\mathcal{D}^{W,Z}_{|N_X}$.

First note that if $N_X = 0$, then $\mathcal{D}^{W,Z}_{|N_X} = \emptyset$. If $N_X \neq 0$, we consider the datasets $\mathcal{D}^{X,Y}_{|N_X}$ and $\mathcal{D}^{X}_{|N_X}$, and proceed as follows:

First construct a random vector (W, Z) satisfying the MDcond(r) with $r \in (0, 1)$ following a similar reasoning as in proof of Theorem 1:

- Let (X, Y) be a random variable admitting $f_{X,Y}$ as probability density function.
- Let Z be a random variable following a Bernoulli of parameter $r \in (0, 1)$.
- Let $\widetilde{X}$ be a random variable independent of (X, Y) following the law f_X.
- Set $W = (X_i \mathbb{1}_{Z_i=1} + \widetilde{X}_i \mathbb{1}_{Z_i=0}, Y_i)$.

Then, $\forall (x, \widetilde{x}, y) \in \mathcal{X} \times \mathcal{Y} \times \mathcal{Y}$ we have $\forall z \in \{0, 1\}$,

$$g_{W|Z=z}(x, \widetilde{x}, y) = \begin{cases} f_{X,Y}(x, y) \text{ if } z = 1 \\ f_X(\widetilde{x}) f_Y(y) \text{ if } z = 0 \end{cases}$$

Then $\forall(x, y, z) \in \mathcal{X} \times \mathcal{Y} \times \{0, 1\}$, it comes

$$f_{W\,|\,Z=z}(x, y) = g_{W\,|\,Z=z}(x, x, y) = f_{X,Y}(x, y)\mathbb{1}_{z=1} + f_X(x)f_Y(y)\mathbb{1}_{z=0}.$$

Moreover, $\forall(x, y) \in \mathcal{X} \times \mathcal{Y}$; we have by equation (5.8):

$$f_W(x, y) = rf_{X,Y}(x, y) + (1 - r)f_X(x)f_Y(y).$$

So (W, Z) satisfies the MDcond(r).

(i) Sample N_X independent observations $(Z_i)_{i=1,\cdots,N_X}$ according to a Bernoulli law of parameter $r \in (0, 1)$.

(ii) For all $i = 1, \cdots, N_X$; $\forall(X_i, Y_i) \in \mathcal{D}^{X,Y}_{|N_X}$ and $\forall\widetilde{X}_i \in \mathcal{D}^{X}_{|N_X}$, set

$$W_i = (X_i\mathbb{1}_{z=1} + \widetilde{X}_i\mathbb{1}_{z=0}, Y_i)$$

.

Note that, by construction $\mathcal{D}^{W,Z}_{|N_X} = (W_i, Z_i)_{i=1}^{N_X}$ are *i.i.d.* and follow $f_{W,Z}$.

Concatenation.

Now to conclude our proof, note that $\mathcal{D}^{W,Z}_{|N_X}$, $\mathcal{D}^{W,Z}_{|N_Y}$ and $\mathcal{D}^{W,Z}_{|N_{X,Y}}$ are independent by construction, since $\mathcal{D}^{X,Y}_{|2N_{X,Y}}$, $(\mathcal{D}^{X,Y}_{|N_X}, \mathcal{D}^{X}_{|N_X})$ and $(\mathcal{D}^{X,Y}_{|N_Y}, \mathcal{D}^{Y}_{|N_Y})$ are independent. Moreover, since $\mathcal{D}^{W,Z}_{|N_X}$, $\mathcal{D}^{W,Z}_{|N_Y}$ and $\mathcal{D}^{W,Z}_{|N_{X,Y}}$ samples are *i.i.d.* random variables following the law $f_{W,Z}$ we have $\mathcal{D}^{W,Z}_{N} = \mathcal{D}^{W,Z}_{|N_X} \cup \mathcal{D}^{W,Z}_{|N_Y} \cup \mathcal{D}^{W,Z}_{|N_{X,Y}}$ a training set of $N = N_{X,Y} + N_X + N_Y$ *i.i.d.* samples following the law $f_{W,Z}$.

5.6.6 Proof Theorem 4 and and *i.d.* Construction in the Additional data setting

Let n_J and n_M two integers such that $1 \le n_J \le n$ and $1 \le n_M \le (n + n_x)(n + n_y) - n$.

Construction 4. [*i.d.* $\mathcal{D}^{W,Z}_{N}$ *Additional data*]

The construction is the same as in the proof of Theorem 2, except we replace **Step (1)** in Construction 2 with **Step (1-bis)** detailed below.

Step (1-bis) : We first generate $(n + n_x)(n + n_y)$ observations $\{(W_i, Z_i)\}_{i=1,\cdots,(n+n_x)(n+n_y)}$ by concatenating 4 sets of samples denoted $\mathcal{S}$, $\mathcal{S}^{\widetilde{X}}$, $\mathcal{S}^{\widetilde{Y}}$ and $\mathcal{S}^{\widetilde{X},\widetilde{Y}}$ of size n^2, $n \times n_x$, $n \times n_y$ and $n_x \times n_y$ respectively.

Sub-Step (A) Generate the set of samples $\mathcal{S}$ exactly like in **Step (1)** of the Construction 2.

Sub-Step (B) Construct the set of samples $\mathcal{S}^{\widetilde{X}} = \{(W_i, Z_i)\}_{i=1,\cdots,n\times n_x}$ of size $n \times n_x$ as follows: $\forall j = 0, \cdots, n-1$, $\forall k = 1, \cdots, n_x$,

$$\begin{cases} W_{jn_x+k} = (\widetilde{X}_k, Y_{n^2+j+1}) \\ Z_{jn_x+k} = 0 \end{cases}$$

Since $\mathcal{D}^X_{n_x}$ and $\mathcal{D}^{X,Y}_n$ are independent and their respective elements are *i.i.d.*, we have $\forall i = 1, \cdots, n^2$, $f_{W_i} \equiv f_X f_Y$.

Sub-Step (C) Construct the set of samples $\mathcal{S}^{\widetilde{Y}} = \{(W_i, Z_i)\}_{i=1,\cdots,n\times n_y}$ of size $n \times n_y$ as follows: $\forall j = 0, \cdots, n-1$, $\forall k = 1, \cdots, n_y$,

$$\begin{cases} W_{jn_y+k} = (X_{j+1}, \widetilde{Y}_k) \\ Z_{jn_y+k} = 0 \end{cases}$$

Since $\mathcal{D}^Y_{n_y}$ and $\mathcal{D}^{X,Y}_n$ are independent and their respective elements are *i.i.d.*, we have $\forall i = 1, \cdots, n \times n_y$, $f_{W_i} \equiv f_X f_Y$.

Sub-Step (D) Construct the set of samples $\mathcal{S}^{\widetilde{X},\widetilde{Y}} = \{(W_i, Z_i)\}_{i=1,\cdots,n_x n_y}$ of size $n_x \times n_y$ as follows: $\forall j = 0, \cdots, n_x - 1$, $\forall k = 1, \cdots, n_y$,

$$\begin{cases} W_{jn_y+k} = (\widetilde{X}_{j+1}, \widetilde{Y}_k) \\ Z_{jn_y+k} = 0 \end{cases}$$

Since $\mathcal{D}^X_{n_x}$ and $\mathcal{D}^Y_{n_y}$ are independent and their respective elements are *i.i.d.*, we have $\forall i = 1, \cdots, n_x n_y$, $f_{W_i} \equiv f_X f_Y$.

Sub-Step (E) Finally, concatenate $\mathcal{S}$, $\mathcal{S}^{\widetilde{X}}$, $\mathcal{S}^{\widetilde{Y}}$ and $\mathcal{S}^{\widetilde{X},\widetilde{Y}}$ which ends the **Step (1-bis)**.

(Next-Step) : Next, do **Step (2)**, **Step (3)**, **Step (4)** and **Step (5)** of the Construction 2.

Using the same arguments as in the proof of Theorem 2, $\mathcal{D}^{W,Z}_N$ satisfies the

MDcond(r). Since $\forall i = n^2+1,\cdots,(n_x+n)(n_y+n)$, we have $f_{W_i} \equiv f_X f_Y$ and $Z_i = 0$. Therefore $\forall i = 1,\cdots,(n_x+n)(n_y+n)$, $f_{W_i} \equiv f_{X,Y}\,\mathbb{1}_{Z_i=1} + f_X f_Y\,\mathbb{1}_{Z_i=0}$.

5.6.7 Proof Theorem 5 and Construction in the non-independent case

Let n_J and n_M two integers such that $1 \leq n_J \leq n \times m$ and $1 \leq n_M \leq n(n-1)m$. We construct the final dataset similary to Construction 2, except we replace **Step (1)** with **Step (1-ter)** :

Construction 5. [*i.d.* $\mathcal{D}_N^{W,Z}$ *Framework 3*]

Step (1-ter) : Construct n^2m observations $\{(W_i, Z_i)\}_{i=1,\cdots,n^2m}$ such that:

$$\forall j = 0,\cdots,n-1,\quad \forall k = 1,\cdots,n\,\forall l = 1,\cdots,m,$$
$$\begin{cases} W_{(jn+k)m+l} = (X_{j+1}, Y_l^k) \\ Z_{(jn+k)m+l} = \mathbb{1}_{j+1=k} \end{cases}$$

(Next-Step) : Next, do **Step (2)** , **Step (3)** , **Step (4)** and **Step (5)** of the Construction 2.

Since $\forall j = 0,\cdots,n-1$, $\forall k = 1,\cdots,n$, $\forall l = 1,\cdots,m$,

$$X_{j+1} \perp\!\!\!\perp Y_l^k \;iff\; j+1 \neq k,$$

we can use the same arguments as in the proof of Theorem 2 to show that $\forall i = 1,\cdots,N$, the couple $(W_i, Z_i) \in \mathcal{D}_N^{W,Z}$ satisfies the MDcond$\left(\frac{n_J}{N}\right)$.

5.7 Appendix

5.7.1 Method to rescale estimated densities

If we want our estimation to correspond to a proper probability density function, that is $\int_y \widehat{f}_{Y\,|\,X=x}(y)dy \approx 1$, we can numerically estimate the integral using the trapezoidal rule on a grid of m target values $\{y_i\}_{i=1,\cdots,m}$ such that its values are evenly distributed between the quantiles 0.001 and 0.999 of f_Y (which we can always estimate through the marginal density estimator) and then rescale the predicted value, doing as follows:

MCD.*rescale* (pdf)	• Generate a grid of m target values $\{y_i\}_{i=1,\cdots,m}$. • Estimate $\{f_{Y\mid X=x}(y_i)\}_{i=1,\cdots,m}$. • Use the trapezoidal rule to estimate $\int_{\mathcal{Y}} \widehat{f}_{Y\mid X=x}(y)dy$. • Divide the predicted value by the estimation of $\int_{\mathcal{Y}} \widehat{f}_{Y\mid X=x}(y)dy$.

.

Note that this step must be repeated for each observation x for which we want to estimate a conditional density function, but given a set observation x, we can reuse the computed integral for any number of target values. This technique will also be used for the other benchmarked methods which do not yield proper integrals by default.

5.7.2 Exhaustive experimental results

In this section, we present the exhaustive results corresponding to Tables 5.5, 5.6 and 5.7.

Empirical **K.L.**	**MCD** :MLP	**MCD** :**CatBoost**	**NNKCDE**	**MDN**	**KMN**	**N.Flow**	**LSCond**	**RFCDE**	**FlexCode** :MLP	**FlexCode** :XGB	**Deepcde**
$p = 3$	0.008	0.009	0.022	0.109	0.067	0.189	0.342	0.037	0.094	0.093	0.152
$p = 10$	0.036	0.042	0.063	0.229	0.096	0.228	0.61	0.105	0.106	0.076	0.205
$p = 30$	0.115	0.202	0.154	0.396	0.181	0.432	0.369	0.238	0.22	0.196	0.385
$p = 100$	0.162	0.224	0.173	0.45	0.401	0.411	1.059	0.238	0.234	0.264	0.36
$p = 300$	0.244	0.308	0.282	0.506	0.304	0.67	0.783	0.507	0.253	0.302	0.306

Table 5.11: Evaluation of the **K.L.** divergence values for various feature sizes p, on the **BasicLinear** density model, with $n = 100$.

Time in sec.	**MCD** :MLP	**MCD** :**CatBoost**	**NNKCDE**	**MDN**	**KMN**	**N.Flow**	**LSCond**	**RFCDE**	**FlexCode** :MLP	**FlexCode** :XGB	**Deepcde**
$n = 30$	3.133	0.46	0.01	35.46	47.92	50.74	12.01	0.044	0.018	9.574	1.943
$n = 100$	3.149	0.369	0.022	35.45	102.7	50.72	37.14	0.149	0.017	9.57	2.11
$n = 300$	3.132	0.446	0.04	35.65	150.6	50.62	100.5	0.448	0.017	13.19	3.225
$n = 1000$	3.296	0.689	0.043	35.73	143.6	51.12	167.9	0.952	0.02	25.24	6.388

Table 5.12: Training Time in seconds for various training set sizes n, on the **BasicLinear** density model.

Empirical NLL	**MCD** :MLP	**CatBoost**	**NNKCDE**	**MDN**	**KMN**	**N.Flow**	**LSCond**	**FlexCode** :MLP	**FlexCode** :XGB	**Deepcde**
BostonHousing	-0.64	-0.59	-0.81	-1.17	-1.22	-1.63	-2.19	-1.99	-1.84	-7.09
Concrete	-0.86	-1.02	-1.23	-2.02	-2.30	-2.13	-4.00	-2.26	-1.25	-10.1
NCYTaxiDropoff:lon.	-1.30	-1.28	-1.68	-2.51	-2.51	-2.85	-3.90	-2.05	-2.17	-11.2
NCYTaxiDropoff:lat.	-1.31	-1.31	-1.44	-2.51	-2.45	-2.96	-4.72	-1.67	-6.18	-9.35
Power	-0.06	-0.36	-0.73	-0.55	-0.35	-0.39	-0.70	-1.09	-0.75	-8.97
Protein	-0.09	-0.42	-0.77	-0.54	-0.68	-0.54	-1.09	-0.83	-1.38	-10.5
WineRed	-0.89	-0.89	3.486	-1.27	-0.90	-2.43	-6.25	1.062	0.965	-10.1
WineWhite	-1.18	-1.13	2.99	-2.24	-1.7	-4.18	-5.41	-0.73	-0.63	-13.2
Yacht	0.14	0.822	-0.46	0.083	0.025	0.401	-2.79	-1.23	0.144	-7.52
teddy	-0.47	-0.51	-0.83	-0.87	-0.76	-0.94	-0.91	-0.83	-1.34	-9.59
toy dataset 1	-0.99	-0.47	-0.63	-0.40	-0.35	-0.71	-0.70	-0.88	-1.46	-6.10
toy dataset 2	-1.40	-1.33	-1.31	-1.40	-1.33	-1.39	-1.35	-1.54	-1.43	-3.53

Table 5.13: Evaluation of the negative log-likelihood (NLL) for 12 datasets.

Références / References

Bibliography

[1] Conditional Density Estimation Documentation — Conditional Density Estimation 0.1 documentation.

[2] Kaggle.

[3] Conditional Density Estimation (CDE), March 2022. original-date: 2017-12-18T17:52:01Z.

[4] Hirotugu Akaike. A new look at the statistical model identification. *IEEE transactions on automatic control*, 19(6):716–723, 1974. Publisher: Ieee.

[5] Luca Ambrogioni, Umut Güçlü, Marcel A. J. van Gerven, and Eric Maris. The Kernel Mixture Network: A Nonparametric Method for Conditional Density Estimation of Continuous Random Variables. *arXiv:1705.07111 [stat]*, May 2017. arXiv: 1705.07111.

[6] Sercan O. Arik and Tomas Pfister. TabNet: Attentive Interpretable Tabular Learning, 2020.

[7] Devansh Arpit, Stanis\law Jastrzębski, Nicolas Ballas, David Krueger, Emmanuel Bengio, Maxinder S Kanwal, Tegan Maharaj, Asja Fischer, Aaron Courville, Yoshua Bengio, and others. A closer look at memorization in deep networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 233–242. JMLR. org, 2017.

[8] Devansh Arpit, Yingbo Zhou, Bhargava U Kota, and Venu Govindaraju. Normalization propagation: A parametric technique for removing internal covariate shift in deep networks. *arXiv preprint arXiv:1603.01431*, 2016.

[9] Arthur Asuncion and David Newman. UCI machine learning repository, 2007.

[10] Francis Bach. Breaking the Curse of Dimensionality with Convex Neural Networks. page 53.

[11] Philip Bachman, R. Devon Hjelm, and William Buchwalter. Learning Representations by Maximizing Mutual Information Across Views. *arXiv:1906.00910 [cs, stat]*, July 2019. arXiv: 1906.00910.

[12] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv*, 2014.

[13] Iñigo Barandiaran. The random subspace method for constructing decision forests. *IEEE transactions on pattern analysis and machine intelligence*, 1998.

[14] R. Beck, C.-A. Lin, E. E. O. Ishida, F. Gieseke, R. S. de Souza, M. V. Costa-Duarte, M. W. Hattab, and A. Krone-Martins. On the realistic validation of photometric redshifts, or why Teddy will never be Happy. *Monthly Notices of the Royal Astronomical Society*, 468(4):4323–4339, July 2017. arXiv: 1701.08748.

[15] Mikhail Belkin, Daniel Hsu, Siyuan Ma, and Soumik Mandal. Reconciling modern machine-learning practice and the classical bias–variance trade-off. *Proceedings of the National Academy of Sciences*, 116(32):15849–15854, 2019. Publisher: National Acad Sciences.

[16] David A Belsley, Edwin Kuh, and Roy E Welsch. *Regression diagnostics: Identifying influential data and sources of collinearity*, volume 571. John Wiley & Sons, 2005.

[17] Yoshua Bengio. Gradient-based optimization of hyperparameters. *Neural computation*, 12(8):1889–1900, 2000. Publisher: MIT Press.

[18] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(Feb):281–305, 2012.

[19] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554, 2011.

[20] Karine Bertin, Claire Lacour, and Vincent Rivoirard. Adaptive pointwise estimation of conditional density function. *arXiv:1312.7402 [math, stat]*, December 2014. arXiv: 1312.7402.

[21] Quentin Bertrand, Quentin Klopfenstein, Mathieu Blondel, Samuel Vaiter, Alexandre Gramfort, and Joseph Salmon. Implicit differentiation of Lasso-type models for hyperparameter optimization. *arXiv preprint arXiv:2002.08943*, 2020.

[22] Christopher M. Bishop. Mixture density networks. Technical report, 1994.

[23] H. D. Block. The Perceptron: A Model for Brain Functioning. I. *Reviews of Modern Physics*, 34(1):123–135, January 1962. Publisher: American Physical Society.

[24] Vadim Borisov, Tobias Leemann, Kathrin Seßler, Johannes Haug, Martin Pawelczyk, and Gjergji Kasneci. Deep Neural Networks and Tabular Data: A Survey, 2021. _eprint: 2110.01889.

[25] Siegfried Bos and E Chug. Using weight decay to optimize the generalization ability of a perceptron. In *Proceedings of International Conference on Neural Networks (ICNN'96)*, volume 1, pages 241–246. IEEE, 1996.

[26] Ann-Kathrin Bott and Michael Kohler. Nonparametric estimation of a conditional density. *Annals of the Institute of Statistical Mathematics*, 69(1):189–214, February 2017.

[27] Léon Bottou. Online learning and stochastic approximations. *On-line learning in neural networks*, 17(9):142, 1998.

[28] In The Brain and F. Rosenblatt. The Perceptron: A Probabilistic Model for Information Storage and Organization.

[29] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Wadsworth and Brooks, Monterey, CA, 1984.

[30] Leo Breiman. Bagging Predictors. *Mach. Learn.*, 24(2):123–140, August 1996. Place: USA Publisher: Kluwer Academic Publishers.

[31] Leo Breiman. Arcing the edge. Technical report, 1997.

[32] Eric Brochu, Vlad M Cora, and Nando De Freitas. A tutorial on Bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*, 2010.

[33] Hans Buehler, Lukas Gonon, Josef Teichmann, Ben Wood, Baranidharan Mohan, and Jonathan Kochems. Deep Hedging: Hedging Derivatives Under Generic Market Frictions Using Reinforcement Learning. SSRN Scholarly Paper ID 3355706, Social Science Research Network, Rochester, NY, March 2019.

[34] M. Cassotti, D. Ballabio, R. Todeschini, and V. Consonni. A similarity-based QSAR model for predicting acute toxicity towards the fathead minnow (Pimephales promelas). *SAR and QSAR in Environmental Research*, 26(3):217–243, 2015. Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/1062936X.2015.1018938.

[35] Husanjot Chahal, Helen Toner, and Ilya Rahkovsky. Small Data's Big AI Potential. Technical report, September 2021.

[36] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A Library for Support Vector Machines. 2(3), May 2011. Place: New York, NY, USA Publisher: Association for Computing Machinery.

[37] Pratik Chaudhari and Stefano Soatto. Stochastic gradient descent performs variational inference, converges to limit cycles for deep networks. 2018.

[38] Boyuan Chen, Harvey Wu, Warren Mo, Ishanu Chattopadhyay, and Hod Lipson. Autostacker: A compositional evolutionary learning system. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 402–409, 2018.

[39] Pengfei Chen, Benben Liao, Guangyong Chen, and Shengyu Zhang. Understanding and Utilizing Deep Neural Networks Trained with Noisy Labels. In *ICML*, pages 1062–1070, 2019.

[40] Tianqi Chen and Carlos Guestrin. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA, 2016. Association for Computing Machinery. event-place: San Francisco, California, USA.

[41] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A Simple Framework for Contrastive Learning of Visual Representations. *arXiv:2002.05709 [cs, stat]*, June 2020. arXiv: 2002.05709.

[42] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, Rohan Anil, Zakaria Haque, Lichan Hong, Vihan Jain, Xiaobing Liu, and Hemal Shah. Wide & Deep Learning for Recommender Systems. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*, pages 7–10, Boston MA USA, September 2016. ACM.

[43] François Chollet and The Keras team. Keras: Deep Learning for humans, April 2022. original-date: 2015-03-28T00:35:42Z.

[44] Andrzej Chrzeszczyk and Jakub Chrzeszczyk. *Matrix computations on the GPU, CUBLAS and MAGMA by example*. developer.nvidia.com, January 2013.

[45] Paul Covington, Jay Adams, and Emre Sargin. Deep Neural Networks for YouTube Recommendations. In *Proceedings of the 10th ACM Conference on Recommender Systems*, pages 191–198, Boston Massachusetts USA, September 2016. ACM.

[46] David R Cox. The regression analysis of binary sequences. *Journal of the Royal Statistical Society: Series B (Methodological)*, 20(2):215–232, 1958. Publisher: Wiley Online Library.

[47] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, December 1989.

[48] Niccolò Dalmasso, Taylor Pospisil, Ann B. Lee, Rafael Izbicki, Peter E. Freeman, and Alex I. Malz. Conditional Density Estimation Tools in Python and R with Applications to Photometric Redshifts and Likelihood-Free Cosmological Inference. *Astronomy and Computing*, 30:100362, January 2020. arXiv: 1908.11523.

[49] Yann N Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. In *International conference on machine learning*, pages 933–941. PMLR, 2017.

[50] Aaron Defazio and Léon Bottou. Scaling Laws for the Principled Design, Initialization and Preconditioning of ReLU Networks, 2019. _eprint: 1906.04267.

[51] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012. Publisher: IEEE.

[52] Emily Denton, Alex Hanna, Razvan Amironesei, Andrew Smart, and Hilary Nicole. On the genealogy of machine learning datasets: A critical history of ImageNet. *Big Data & Society*, 8(2):20539517211035955, 2021. Publisher: SAGE Publications Sage UK: London, England.

[53] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[54] Justin Domke. Generic methods for optimization-based modeling. In *Artificial Intelligence and Statistics*, pages 318–326, 2012.

[55] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of machine learning research*, 12(7), 2011.

[56] A. D'Isanto and K. L. Polsterer. Photometric redshift estimation via deep learning - Generalized and pre-classification-less, image based, fully probabilistic redshifts. *Astronomy & Astrophysics*, 609:A111, January 2018. Publisher: EDP Sciences.

[57] Sergio Escalera and Ralf Herbrich. The NeurIPS'18 Competition.

[58] Li Fei-Fei, Rob Fergus, and Pietro Perona. One-shot learning of object categories. *IEEE transactions on pattern analysis and machine intelligence*, 28(4):594–611, 2006. Publisher: IEEE.

[59] Vitaly Feldman. Does learning require memorization? a short tale about a long tail. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*, pages 954–959. Association for Computing Machinery, New York, NY, USA, 2020.

[60] Vitaly Feldman and Chiyuan Zhang. What Neural Networks Memorize and Why: Discovering the Long Tail via Influence Estimation. In *Advances in Neural Information Processing Systems*, volume 33, pages 2881–2891. Curran Associates, Inc., 2020.

[61] Shuo Feng, Huiyu Zhou, and Hongbiao Dong. Using deep neural network with small dataset to predict material defects. *Materials & Design*, 162:300–310, 2019.

[62] Manuel Fernández-Delgado, Eva Cernadas, Senén Barro, and Dinani Amorim. Do we Need Hundreds of Classifiers to Solve Real World Classification Problems? *Journal of Machine Learning Research*, 15(90):3133–3181, 2014.

[63] Michael Fink. Object classification from a single example utilizing class relevance metrics. *Advances in neural information processing systems*, 17:449–456, 2005. Publisher: Curran Associates, Inc. New York.

[64] Evelyn Fix and J. L. Hodges. Discriminatory Analysis. Nonparametric Discrimination: Consistency Properties. *International Statistical Review / Revue Internationale de Statistique*, 57(3):238–247, 1989. Publisher: [Wiley, International Statistical Institute (ISI)].

[65] Yoav Freund and Robert E Schapire. A desicion-theoretic generalization of on-line learning and an application to boosting. In *European conference on computational learning theory*, pages 23–37. Springer, 1995.

[66] Jerome H. Friedman. Multivariate Adaptive Regression Splines. *The Annals of Statistics*, 19(1):1 – 67, 1991. Publisher: Institute of Mathematical Statistics.

[67] Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5):1189 – 1232, 2001. Publisher: Institute of Mathematical Statistics.

[68] Jerome H. Friedman. Stochastic Gradient Boosting. *Comput. Stat. Data Anal.*, 38(4):367–378, February 2002. Place: NLD Publisher: Elsevier Science Publishers B. V.

[69] Timur Garipov, Pavel Izmailov, Dmitrii Podoprikhin, Dmitry Vetrov, and Andrew Gordon Wilson. Loss Surfaces, Mode Connectivity, and Fast Ensembling of DNNs. Technical Report arXiv:1802.10026, arXiv, October 2018. arXiv:1802.10026 [cs, stat] type: article.

[70] Servane Gey and Elodie Nedelec. Model selection for CART regression trees. *IEEE Transactions on Information Theory*, 51(2):658–670, 2005. Publisher: IEEE.

[71] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In Yee Whye Teh and Mike Titterington, editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy, May 2010. PMLR.

[72] Alexander Goldenshluger and Oleg Lepski. BANDWIDTH SELECTION IN KERNEL DENSITY ESTIMATION: ORACLE INEQUALITIES AND ADAPTIVE MINIMAX OPTIMALITY. *Annals of Statistics*, 39(3):1608–1632, 2011. Publisher: Institute of Mathematical Statistics.

[73] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[74] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative Adversarial Networks. *arXiv:1406.2661 [cs, stat]*, June 2014. arXiv: 1406.2661.

[75] Yury Gorishniy, Ivan Rubachev, Valentin Khrulkov, and Artem Babenko. Revisiting Deep Learning Models for Tabular Data. In A. Beygelzimer,

Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, 2021.

[76] Alex Graves. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

[77] Suriya Gunasekar, Jason Lee, Daniel Soudry, and Nathan Srebro. Characterizing Implicit Bias in Terms of Optimization Geometry. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1832–1841. PMLR, July 2018.

[78] Cheng Guo and Felix Berkhahn. Entity Embeddings of Categorical Variables. *arXiv e-prints*, page arXiv:1604.06737, April 2016. _eprint: 1604.06737.

[79] Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He. DeepFM: A Factorization-Machine based Neural Network for CTR Prediction. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*, pages 1725–1731, Melbourne, Australia, August 2017. International Joint Conferences on Artificial Intelligence Organization.

[80] Michael Gutmann and Aapo Hyvärinen. Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 297–304. JMLR Workshop and Conference Proceedings, March 2010. ISSN: 1938-7228.

[81] Bo Han, Quanming Yao, Xingrui Yu, Gang Niu, Miao Xu, Weihua Hu, Ivor Tsang, and Masashi Sugiyama. Co-teaching: Robust training of deep neural networks with extremely noisy labels. In *Advances in neural information processing systems*, pages 8527–8537, 2018.

[82] D. L. Hanson and F. T. Wright. A Bound on Tail Probabilities for Quadratic Forms in Independent Random Variables. *The Annals of Mathematical Statistics*, 42(3):1079 – 1083, 1971. Publisher: Institute of Mathematical Statistics.

[83] Stephen Hanson and Lorien Pratt. Comparing Biases for Minimal Network Construction with Back-Propagation. pages 177–185, January 1988.

[84] David Harrison Jr and Daniel L Rubinfeld. Hedonic housing prices and the demand for clean air. *Journal of environmental economics and management*, 5(1):81–102, 1978. Publisher: Elsevier.

[85] Hrayr Harutyunyan, Kyle Reing, Greg Ver Steeg, and Aram Galstyan. Improving Generalization by Controlling Label-Noise Information in Neural Network Weights. *CoRR*, abs/2002.07933, 2020. arXiv: 2002.07933.

[86] Fengxiang He, Tongliang Liu, and Dacheng Tao. Control Batch Size and Learning Rate to Generalize Well: Theoretical and Empirical Evidence. In *Advances in Neural Information Processing Systems*, pages 1141–1150, 2019.

[87] Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. Momentum Contrast for Unsupervised Visual Representation Learning. *arXiv:1911.05722 [cs]*, March 2020. arXiv: 1911.05722.

[88] Kaiming He, Ross Girshick, and Piotr Dollar. Rethinking ImageNet Pre-Training. pages 4918–4927, 2019.

[89] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *IEEE International Conference on Computer Vision (ICCV 2015)*, 1502, February 2015.

[90] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[91] Xin He, Kaiyong Zhao, and Xiaowen Chu. AutoML: A Survey of the State-of-the-Art. *Knowledge-Based Systems*, 212:106622, 2021. Publisher: Elsevier.

[92] Geoffrey Hinton, Li Deng, Dong Yu, George E. Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N. Sainath, and Brian Kingsbury. Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[93] Geoffrey E. Hinton. Connectionist Learning Procedures, 1989.

[94] Tin Kam Ho. Random decision forests. In *Proceedings of 3rd international conference on document analysis and recognition*, volume 1, pages 278–282. IEEE, 1995.

[95] Arthur E Hoerl and Robert W Kennard. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, 12(1):55–67, 1970. Publisher: Taylor & Francis Group.

[96] Elad Hoffer, Ron Banner, Itay Golan, and Daniel Soudry. Norm matters: efficient and accurate normalization schemes in deep networks. *arXiv preprint arXiv:1803.01814*, 2018.

[97] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, January 1991.

[98] Jeremy Howard. Fastai.

[99] Jeremy Howard and Sylvain Gugger. Fastai: A Layered API for Deep Learning. *Inf.*, 11(2):108, 2020.

[100] Sergey Ioffe. Batch Renormalization: Towards Reducing Minibatch Dependence in Batch-Normalized Models. *arXiv preprint arXiv:1702.03275*, 2017.

[101] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[102] Rafael Izbicki and Ann B. Lee. Converting high-dimensional regression to high-dimensional conditional density estimation. *Electronic Journal of Statistics*, 11(2), January 2017.

[103] Ashish Jaiswal, Ashwin Ramesh Babu, Mohammad Zaki Zadeh, Debapriya Banerjee, and F. Makedon. A Survey on Contrastive Self-supervised Learning. *Technologies*, 2020.

[104] Lu Jiang, Zhengyuan Zhou, Thomas Leung, Li-Jia Li, and Li Fei-Fei. Mentornet: Learning data-driven curriculum for very deep neural networks on corrupted labels. *arXiv preprint arXiv:1712.05055*, 2017.

[105] Mark H. Johnson and Jeff Shrager. Dynamic Plasticity Influences the Emergence of Function in a Simple Cortical Array. *Neural Networks: The Official Journal of the International Neural Network Society*, 9(7):1119–1129, October 1996.

[106] Manu Joseph. PyTorch Tabular: A Framework for Deep Learning with Tabular Data, 2021. _eprint: 2104.13638.

[107] Arlind Kadra, Marius Lindauer, Frank Hutter, and Josif Grabocka. Well-tuned Simple Nets Excel on Tabular Datasets. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, 2021.

[108] Kaggle. Kaggle: Your Machine Learning and Data Science Community.

[109] Liran Katzir, Gal Elidan, and Ran El-Yaniv. Net-{DNF}: Effective Deep Modeling of Tabular Data. In *International Conference on Learning Representations*, 2021.

[110] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[111] Sarit Khirirat, Hamid Reza Feyzmahdavian, and Mikael Johansson. Mini-batch gradient descent: Faster convergence under data sparsity. In *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, pages 2880–2887. IEEE, 2017.

[112] Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. Supervised Contrastive Learning. *arXiv:2004.11362 [cs, stat]*, March 2021. arXiv: 2004.11362.

[113] Diederik P Kingma and Jimmy Ba. Adam (2014), A method for stochastic optimization. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR), arXiv preprint arXiv*, volume 1412, 2014.

[114] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-Normalizing Neural Networks. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

[115] Jason Klusowski. Sparse Learning with CART. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 11612–11622. Curran Associates, Inc., 2020.

[116] Bernard Koch, Emily Denton, Alex Hanna, and Jacob G Foster. Reduced, Reused and Recycled: The Life of a Dataset in Machine Learning Research. *arXiv preprint arXiv:2112.01716*, 2021.

[117] Vladimir Koltchinskii. *Oracle inequalities in empirical risk minimization and sparse recovery problems : Ecole d'été de probabilités de Saint-Flour XXXVIII-2008 / Vladimir Koltchinskii*. Lecture notes in mathematics Ecole

d'été de probabilités de Saint-Flour. Springer, Heidelberg London New York, 2011. Backup Publisher: École d'été de probabilités de Saint-Flour 38 2008 Saint-Flour, Cantal Publication Title: Oracle inequalities in empirical risk minimization and sparse recovery problems : Ecole d'été de probabilités de Saint-Flour XXXVIII-2008.

[118] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks - Book version. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[119] Anders Krogh and John A Hertz. A simple weight decay can improve generalization. In *Advances in neural information processing systems*, pages 950–957, 1992.

[120] William H Kruskal. Historical notes on the Wilcoxon unpaired two-sample test. *Journal of the American Statistical Association*, 52(279):356–360, 1957. Publisher: Taylor & Francis Group.

[121] Jan Kukavcka, Vladimir Golkov, and Daniel Cremers. Regularization for deep learning: A taxonomy. *arXiv preprint arXiv:1710.10686*, 2017.

[122] Alexandre Lacoste, Hugo Larochelle, Mario Marchand, and François Laviolette. Sequential model-based ensemble optimization. In *Proceedings of the Thirtieth Conference on Uncertainty in Artificial Intelligence*, pages 440–448, 2014.

[123] Nicolas Langrené and Xavier Warin. Fast multivariate empirical cumulative distribution function with connection to kernel density estimation. August 2020.

[124] Jan Larsen, Lars Kai Hansen, Claus Svarer, and M Ohlsson. Design and regularization of neural networks: the optimal use of a validation set. In *Neural Networks for Signal Processing VI. Proceedings of the 1996 IEEE Signal Processing Society Workshop*, pages 62–71. IEEE, 1996.

[125] B. Laurent and Pascal Massart. Adaptive estimation of a quadratic functional by model selection. *Annals of Statistics*, 28, October 2000.

[126] Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. Visualizing the Loss Landscape of Neural Nets. In *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[127] Mingchen Li, Mahdi Soltanolkotabi, and Samet Oymak. Gradient Descent with Early Stopping is Provably Robust to Label Noise for Overparameterized Neural Networks. In Silvia Chiappa and Roberto Calandra, editors, *Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics*, volume 108 of *Proceedings of Machine Learning Research*, pages 4313–4324. PMLR, August 2020.

[128] Shih-Wei Lin, Kuo-Ching Ying, Shih-Chieh Chen, and Zne-Jung Lee. Particle swarm optimization for parameter determination and feature selection of support vector machines. *Expert systems with applications*, 35(4):1817–1824, 2008. Publisher: Elsevier.

[129] Pablo Ribalta Lorenzo, Jakub Nalepa, Michal Kawulok, Luciano Sanchez Ramos, and José Ranilla Pastor. Particle swarm optimization for hyperparameter selection in deep neural networks. In *Proceedings of the genetic and evolutionary computation conference*, pages 481–488, 2017.

[130] Ilya Loshchilov and Frank Hutter. Fixing Weight Decay Regularization in Adam. 2018.

[131] Colin L Mallows. Some comments on Cp. *Technometrics*, 42(1):87–94, 2000. Publisher: Taylor & Francis Group.

[132] S. S. Manson. Fatigue: A complex subject—Some simple approximations. *Experimental Mechanics*, 5(4):193–226, July 1965.

[133] Martin Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015.

[134] Nicolai Meinshausen. Quantile Regression Forests. page 17.

[135] Dmytro Mishkin and Jiri Matas. All you need is a good init. *arXiv preprint arXiv:1511.06422*, 2015.

[136] Montreal.AI. AI DEBATE : Yoshua Bengio | Gary Marcus, December 2019.

[137] Mohammad Amin Morid, Alireza Borjali, and Guilherme Del Fiol. A scoping review of transfer learning research on medical image analysis using ImageNet. *Computers in Biology and Medicine*, 128:104115, January 2021.

[138] Jonas Movckus. On Bayesian methods for seeking the extremum. In *Optimization techniques IFIP technical conference*, pages 400–404. Springer, 1975.

[139] Varalakshmi Murugesan, Amit Kesarkar, and Daphne Lopez. Embarrassingly Parallel GPU Based Matrix Inversion Algorithm for Big Climate Data Assimilation. *International Journal of Grid and High Performance Computing*, 10:71–92, January 2018.

[140] Thomas Nagler and Claudia Czado. Evading the curse of dimensionality in nonparametric density estimation with simplified vine copulas. *Journal of Multivariate Analysis*, 151:69–89, October 2016. arXiv: 1503.03305.

[141] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Icml*, 2010.

[142] Rajib Nath, Stanimire Tomov, and Jack Dongarra. Accelerating GPU Kernels for Dense Linear Algebra. In *Proceedings of the 2009 International Meeting on High Performance Computing for Computational Science, VECPAR10*, Berkeley, CA, June 2010. Springer.

[143] Yurii E Nesterov. A method for solving the convex programming problem with convergence rate O (1/k^ 2). In *Dokl. akad. nauk Sssr*, volume 269, pages 543–547, 1983.

[144] Behnam Neyshabur. Implicit Regularization in Deep Learning. Technical Report arXiv:1709.01953, arXiv, September 2017. arXiv:1709.01953 [cs] type: article.

[145] Andrew Y Ng. Deep Learning.

[146] Randal S Olson, Ryan J Urbanowicz, Peter C Andrews, Nicole A Lavender, Jason H Moore, and others. Automating biomedical data science through tree-based pipeline optimization. In *European Conference on the Applications of Evolutionary Computation*, pages 123–137. Springer, 2016.

[147] Maxime Oquab, Leon Bottou, Ivan Laptev, and Josef Sivic. Learning and Transferring Mid-Level Image Representations using Convolutional Neural Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2014.

[148] Andrei Paleyes, Raoul-Gabriel Urma, and Neil D Lawrence. Challenges in deploying machine learning: a survey of case studies. *arXiv preprint arXiv:2011.09926*, 2020.

[149] Emanuel Parzen. On Estimation of a Probability Density Function and Mode. *The Annals of Mathematical Statistics*, 33(3):1065–1076, 1962. Publisher: Institute of Mathematical Statistics.

[150] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[151] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, and others. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[152] Giorgio Patrini, Alessandro Rozza, Aditya Krishna Menon, Richard Nock, and Lizhen Qu. Making deep neural networks robust to label noise: A loss correction approach. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1944–1952, 2017.

[153] Karl Pearson. Contributions to the Mathematical Theory of Evolution. II. Skew Variation in Homogeneous Material. January 1895.

[154] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[155] Fabian Pedregosa. Hyperparameter optimization with approximate gradient. In *International Conference on Machine Learning*, pages 737–746, 2016.

[156] Sergei Popov, Stanislav Morozov, and Artem Babenko. Neural Oblivious Decision Ensembles for Deep Learning on Tabular Data. In *International Conference on Learning Representations*, 2020.

[157] Taylor Pospisil. NNKCDE: Nearest Neighbor Conditional Density Estimation, June 2020. original-date: 2018-02-19T15:10:04Z.

[158] Taylor Pospisil. cdetools: Tools for Conditional Density Estimates, March 2022. original-date: 2018-02-05T19:40:05Z.

[159] Taylor Pospisil and Ann B. Lee. RFCDE: Random Forests for Conditional Density Estimation. *arXiv:1804.05753 [cs, stat]*, May 2018. arXiv: 1804.05753.

[160] Liudmila Prokhorenkova, Gleb Gusev, Aleksandr Vorobev, Anna Veronika Dorogush, and Andrey Gulin. CatBoost: unbiased boosting with categorical features. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[161] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999. Publisher: Elsevier.

[162] Yanru Qu, Han Cai, Kan Ren, Weinan Zhang, Yong Yu, Ying Wen, and Jun Wang. Product-Based Neural Networks for User Response Prediction. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 1149–1154, Barcelona, Spain, December 2016. IEEE.

[163] S. R. Quartz and T. J. Sejnowski. The neural basis of cognitive development: a constructivist manifesto. *The Behavioral and Brain Sciences*, 20(4):537–556; discussion 556–596, December 1997.

[164] Bhiksha Raj. Carnegie Mellon University Deep Learning , Representation Learning, 2018.

[165] Inioluwa Deborah Raji, Emily M. Bender, Amandalynne Paullada, Emily Denton, and Alex Hanna. AI and the Everything in the Whole Wide World Benchmark. Technical Report arXiv:2111.15366, arXiv, November 2021. arXiv:2111.15366 [cs] type: article.

[166] Alvin Rajkomar, E. Oren, K. Chen, Andrew M. Dai, Nissan Hajaj, Michaela Hardt, Peter J. Liu, X. Liu, Jake Marcus, M. Sun, Patrik Sundberg, H. Yee, Kun Zhang, Y. Zhang, Gerardo Flores, Gavin E. Duggan, Jamie Irvine, Quoc V. Le, Kurt Litsch, Alexander Mossin, Justin Tansuwan, D. Wang, James Wexler, J. Wilson, Dana Ludwig, S. Volchenboum, Katherine Chou, Michael Pearson, Srinivasan Madabushi, N. Shah, A. Butte, M. Howell, Claire Cui, Greg Corrado, and Jeffrey Dean. Scalable and accurate deep learning with electronic health records. *NPJ Digital Medicine*, 1, 2018.

[167] Noam Razin and Nadav Cohen. Implicit Regularization in Deep Learning May Not Be Explainable by Norms. Technical Report arXiv:2005.06398, arXiv, October 2020. arXiv:2005.06398 [cs, stat] type: article.

[168] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V Le, and Alexey Kurakin. Large-scale evolution of image classifiers. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 2902–2911. JMLR. org, 2017.

[169] Danilo Jimenez Rezende and Shakir Mohamed. Variational Inference with Normalizing Flows. *arXiv:1505.05770 [cs, stat]*, June 2016. arXiv: 1505.05770.

[170] Philippe Rigollet and Alexandre Tsybakov. Linear and convex aggregation of density estimators. *arXiv:math/0605292*, May 2006. arXiv: math/0605292.

[171] Murray Rosenblatt. Remarks on Some Nonparametric Estimates of a Density Function. *The Annals of Mathematical Statistics*, 27(3):832–837, September 1956. Publisher: Institute of Mathematical Statistics.

[172] Jonas Rothfuss, Fabio Ferreira, Simon Walther, and Maxim Ulrich. Conditional Density Estimation with Neural Networks: Best Practices and Benchmarks. *arXiv:1903.00954 [cs, q-fin, stat]*, April 2019. arXiv: 1903.00954.

[173] Mark Rudelson and Roman Vershynin. Hanson-Wright inequality and subgaussian concentration. In *Electronic Communications in Probability 18*, 2013.

[174] Tim Salimans and Diederik P Kingma. Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks. *arXiv preprint arXiv:1602.07868*, 2016.

[175] Andrew M Saxe, James L McClelland, and Surya Ganguli. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *arXiv preprint arXiv:1312.6120*, 2013.

[176] Jürgen Schmidhuber. *Evolutionary principles in self-referential learning, or on learning how to learn: the meta-meta-... hook.* PhD Thesis, Technische Universität München, 1987.

[177] Johannes Schmidt-Hieber. Nonparametric regression using deep neural networks with ReLU activation function. *The Annals of Statistics*, 48(4), August 2020. arXiv: 1708.06633.

[178] David W. Scott. Feasibility of multivariate density estimates. *Biometrika*, 78(1):197–205, 1991.

[179] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. Taking the human out of the loop: A review of Bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2015. Publisher: IEEE.

[180] Girish Sharma, Abhishek Agarwala, and Baidurya Bhattacharya. A fast parallel Gauss Jordan algorithm for matrix inversion using CUDA. *Computers & Structures*, 128:31–37, November 2013.

[181] Ira Shavitt and Eran Segal. Regularization Learning Networks: Deep Learning for Tabular Datasets. *Neurips*, 2018.

[182] Connor Shorten and Taghi M Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of Big Data*, 6(1):1–48, 2019. Publisher: Springer.

[183] Ravid Shwartz-Ziv and Amitai Armon. Tabular data: Deep learning is not all you need. *Information Fusion*, 81:84–90, 2022.

[184] B. W. Silverman. *Density Estimation for Statistics and Data Analysis*. Routledge, New York, October 2017.

[185] Leslie N. Smith. Cyclical Learning Rates for Training Neural Networks, 2015.

[186] Leslie N. Smith. No More Pesky Learning Rate Guessing Games. *CoRR*, abs/1506.01186, 2015. _eprint: 1506.01186.

[187] Leslie N Smith. Cyclical learning rates for training neural networks. In *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 464–472. IEEE, 2017.

[188] Leslie N Smith. A disciplined approach to neural network hyper-parameters: Part 1–learning rate, batch size, momentum, and weight decay. *arXiv preprint arXiv:1803.09820*, 2018.

[189] Leslie N Smith and Nicholay Topin. Super-convergence: Very fast training of neural networks using large learning rates. In *Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications*, volume 11006, page 1100612. International Society for Optics and Photonics, 2019.

[190] Samuel L. Smith, Benoit Dherin, David Barrett, and Soham De. On the Origin of Implicit Regularization in Stochastic Gradient Descent. In *International Conference on Learning Representations*, 2021.

[191] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959, 2012.

[192] Suvrit Sra, Sebastian Nowozin, Stephen J. Wright, Michael I. Jordan, and Thomas G. Dietterich, editors. *Optimization for Machine Learning*. Neural Information Processing series. MIT Press, Cambridge, MA, USA, September 2011.

[193] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[194] Charles M Stein. Estimation of the mean of a multivariate normal distribution. *The annals of Statistics*, pages 1135–1151, 1981. Publisher: JSTOR.

[195] Masashi Sugiyama, Ichiro Takeuchi, Taiji Suzuki, Takafumi Kanamori, Hirotaka Hachiya, and Daisuke Okanohara. Conditional Density Estimation via Least-Squares Density Ratio Estimation. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 781–788. JMLR Workshop and Conference Proceedings, March 2010. ISSN: 1938-7228.

[196] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147. PMLR, 2013.

[197] Jeyan Thiyagalingam, Mallikarjun Shankar, Geoffrey Fox, and Tony Hey. Scientific machine learning benchmarks. *Nature Reviews Physics*, pages 1–8, April 2022. Publisher: Nature Publishing Group.

[198] William R Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3/4):285–294, 1933. Publisher: JSTOR.

[199] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288, 1996. Publisher: Wiley Online Library.

[200] Sami Tibshirani, Harry Friedman, and Trevor Hastie. *The elements of statistical learning*.

[201] Alexandre Tsybakov. Optimal Rates of Aggregation. *Lect. Notes Artif. Intell.*, 2777:303–313, January 2003. ISBN: 978-3-540-40720-1.

[202] Paul E. Utgoff and David J. Stracuzzi. Many-layered learning. *Neural Computation*, 14(10):2497–2529, October 2002.

[203] Andrius Vabalas, Emma Gowen, Ellen Poliakoff, and Alexander J. Casson. Machine learning algorithm validation with a limited sample size. *PLOS ONE*, 14(11):1–20, November 2019. Publisher: Public Library of Science.

[204] Joaquin Vanschoren, Jan N. van Rijn, Bernd Bischl, and Luis Torgo. OpenML: Networked Science in Machine Learning. *SIGKDD Explorations*, 15(2):49–60, 2013. Place: New York, NY, USA Publisher: ACM.

[205] Gaël Varoquaux. Cross-validation failure: Small sample sizes lead to large error bars. *NeuroImage*, 180:68 – 77, 2018.

[206] Cristina Nader Vasconcelos and Bárbara Nader Vasconcelos. Increasing Deep Learning Melanoma Classification by Classical And Expert Knowledge Based Image Transforms. *ArXiv*, abs/1702.07025, 2017.

[207] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, \Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[208] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, and others. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature methods*, 17(3):261–272, 2020. Publisher: Nature Publishing Group.

[209] Li Wan, Matthew Zeiler, Sixin Zhang, Yann Le Cun, and Rob Fergus. Regularization of neural networks using dropconnect. In *International conference on machine learning*, pages 1058–1066, 2013.

[210] Hao Wang, Naiyan Wang, and Dit-Yan Yeung. Collaborative Deep Learning for Recommender Systems. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1235–1244, Sydney NSW Australia, August 2015. ACM.

[211] Yaqing Wang, Quanming Yao, James T Kwok, and Lionel M Ni. Generalizing from a few examples: A survey on few-shot learning. *ACM Computing Surveys (CSUR)*, 53(3):1–34, 2020. Publisher: ACM New York, NY, USA.

[212] Papers with Code. Papers with Code - Browse the State-of-the-Art in Machine Learning, 2018.

[213] Sebastien C Wong, Adam Gatt, Victor Stamatescu, and Mark D McDonnell. Understanding data augmentation for classification: when to warp? In *2016 international conference on digital image computing: techniques and applications (DICTA)*, pages 1–6. IEEE, 2016.

[214] F. T. Wright. A Bound on Tail Probabilities for Quadratic Forms in Independent Random Variables Whose Distributions are not Necessarily Symmetric. *The Annals of Probability*, 1(6):1068–1070, 1973. Publisher: Institute of Mathematical Statistics.

[215] Sitao Xiang and Hao Li. On the effects of batch and weight normalization in generative adversarial networks. *arXiv preprint arXiv:1704.03971*, 2017.

[216] Yan Xu, Ran Jia, Lili Mou, Ge Li, Yunchuan Chen, Yangyang Lu, and Zhi Jin. Improved relation classification by deep recurrent neural networks with data augmentation. *arXiv preprint arXiv:1601.03651*, 2016.

[217] Quanming Yao, Mengshuo Wang, Yuqiang Chen, Wenyuan Dai, Yu-Feng Li, Wei-Wei Tu, Qiang Yang, and Yang Yu. Taking human out of learning applications: A survey on automated machine learning. *arXiv preprint arXiv:1810.13306*, 2018.

[218] Yuan Yao, Lorenzo Rosasco, and Andrea Caponnetto. On Early Stopping in Gradient Descent Learning. *Constructive Approximation*, 26(2):289–315, August 2007.

[219] R. Yedida and S. Saha. A novel adaptive learning rate scheduler for deep neural networks. *arXiv e-prints*, February 2019. _eprint: 1902.07399.

[220] Chun-Hsiao Yeh, Cheng-Yao Hong, Yen-Chi Hsu, Tyng-Luh Liu, Yubei Chen, and Yann LeCun. Decoupled Contrastive Learning. *arXiv:2110.06848 [cs]*, October 2021. arXiv: 2110.06848.

[221] Zhang, Bengio, Hardt, Recht, and Vinyals. Understanding deep learning requires rethinking generalization. *International Conference on Learning Representations*, 2016.

[222] Guodong Zhang, Chaoqi Wang, Bowen Xu, and Roger Grosse. Three mechanisms of weight decay regularization. *arXiv preprint arXiv:1810.12281*, 2018.

[223] Hongyi Zhang, Moustapha Cisse, Yann N Dauphin, and David Lopez-Paz. mixup: Beyond empirical risk minimization. *arXiv preprint arXiv:1710.09412*, 2017.

[224] Weinan Zhang, Tianming Du, and Jun Wang. Deep Learning over Multi-field Categorical Data. In Nicola Ferro, Fabio Crestani, Marie-Francine Moens, Josiane Mothe, Fabrizio Silvestri, Giorgio Maria Di Nunzio, Claudia Hauff, and Gianmaria Silvello, editors, *Advances in Information Retrieval*, pages 45–57, Cham, 2016. Springer International Publishing.

[225] Guorui Zhou, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, Han Li, and Kun Gai. Deep Interest Network for Click-Through Rate Prediction. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '18, pages 1059–1068, New York, NY, USA, 2018. Association for Computing Machinery. event-place: London, United Kingdom.

[226] Lucas Zimmer, Marius Lindauer, and Frank Hutter. Auto-PyTorch Tabular: Multi-Fidelity MetaLearning for Efficient and Robust AutoDL. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 3079 – 3090, 2021.

[227] Hui Zou and Trevor Hastie. Regularization and variable selection via the Elastic Net. *Journal of the Royal Statistical Society, Series B*, 67:301–320, 2005.

[228] Marc-André Zöller and Marco F Huber. Benchmark and survey of automated machine learning frameworks. *Journal of Artificial Intelligence Research*, 70:409–472, 2021.

ECOLE
DOCTORALE
DE MATHEMATIQUES
HADAMARD

Titre : Comparaison des lois conjointes et marginales par permutation des labels pour la régression et l'estimation de densité conditionnelle

Mots clés : Généralisation, Régression, Estimation de densité conditionnelle, Réseaux de neurones, Apprentissage profond

Résumé : Cette thèse introduit de nouvelles techniques qui exploitent des permutations du vecteur des observations de la variable à expliquer pour améliorer les performances de généralisation dans la tâche de régression et transformer l'estimation de la fonction de densité conditionnelle en un problème de classification binaire. Des justifications théoriques et des benchmarks empiriques sur des jeux de données tabulaires sont proposés pour démontrer l'intérêt de ces techniques, en particulier lorsqu'elles sont combinées avec des réseaux de neurones profonds. La généralisation est un problème central en l'apprentissage machine. La plupart des modèles prédictifs nécessitent une calibration minutieuse des hyper-paramètres sur un échantillon de validation pour obtenir de bonnes performances de généralisation. Une nouvelle approche qui contourne cette difficulté est présentée. Elle est basée sur une nouvelle mesure du risque de généralisation qui quantifie directement la propension d'un modèle à sur-ajuster les données d'entraînement. Le critère associé, appelé MLR (Muddling Labels Regularization) est évalué sur le jeu de données d'entraînement et permet d'estimer la performance sur le jeu de données test. Pour cela, il utilise des permutations du vecteur des observations de la variable à expliquer pour quantifier la propension d'un modèle à mémoriser la part de bruit contenu dans les données. Pour transformer le critère MLR en une fonction de perte pour les réseaux de neurones profonds, l'opérateur Tikhonov est introduit. Il module la capacité de mémorisation d'un réseau de manière adaptative, différentiable et dépendante des données. En combinant la perte MLR et l'opérateur Tikhonov, on obtient la technique d'apprentissage AdaCap (ADAptative CAPacity control) qui optimise la capacité du réseau afin qu'il puisse apprendre les représentation abstraite de haut niveau correspondant au problème posé plutôt que de mémoriser le jeu de données d'entraînement. Le problème d'estimation de densité conditionnelle est également traité. Il est à la base de la majorité des tâches d'apprentissage machine, y compris l'apprentissage supervisé et non supervisé ainsi que les modèles génératifs. Une nouvelle méthode, MCD (Marginal Contrastive Discrimination) inspirée du noise contrastive learning est introduite. MCD reformule la tâche initiale en un problème d'apprentissage supervisé qui peut être résolu à l'aide d'un classifieur binaire. Des techniques de construction de jeux de données de contraste basées là encore sur des permutations du vecteur de la variable à expliquer sont également proposées. Elles permettent d'obtenir des jeux de données d'entraînement beaucoup plus grands que le jeu de données initial, et de tirer parti d'observations non-étiquetées et d'observations pour lesquelles on dispose de plusieurs réalisations.

Title : Comparison of joint and marginal laws by permutation of labels for regression and conditional density estimation

Keywords : Generalization, Regression, conditional density estimation, deep neural networks, deep learning

Abstract : This thesis introduces novel techniques which leverage label permutations to improve generalization performances in the regression task and estimate the conditional density function through binary classification. Theoretical justifications and empirical benchmarks on tabular datasets are provided to demonstrate their effectiveness, especially when combined with deep neural networks. Generalization is a central problem in Machine Learning. Most prediction methods require careful calibration of hyperparameters usually carried out on a hold-out validation dataset to achieve generalization. We introduce a novel approach to achieve generalization without any data splitting, which is based on a new risk measure which directly quantifies a model's tendency to overfit. The associated criterion, called MLR (Muddling Labels Regularization) is an in-sample metric for out-of-sample performance which leverages randomly generated labels to quantify the propensity of a model to memorize. To transform the MLR criterion into a training loss for deep neural networks, we introduce the Tikhonov operator training scheme, which modulates the memorization capacity of a FFNN in an adaptive, differentiable and data-dependent manner. By combining the MLR loss and the Tikhonov operator we obtain the AdaCap training scheme (ADAptative CAPacity control) which optimizes the capacity of FFNN so it can capture the high-level abstract representations underlying the problem at hand without memorizing the training dataset. Besides generalization, we also consider the problem of conditional density estimation. This task is at the root of the majority of machine learning tasks, including supervised and unsupervised learning or generative modeling. We introduce a new method, MCD (Marginal Contrastive Discrimination) inspired by contrastive learning. MCD reformulates the initial task into a problem of supervised learning which can be solved with any binary classifier. We present construction techniques based on label permutations to produce a contrast dataset with far more observations than in the original dataset and take advantage of unlabeled observations and more than one target value per observation.

Institut Polytechnique de Paris
91120 Palaiseau, France