

UNIVERSITÉ DE MONTPELLIER 1



MÉMOIRE DE MASTER 2

Les modèles ARMA face aux réseaux de neurones récurrents

Comparaison et applications

Auteur :
Hugues GAILLARD

Directeur de mémoire :
Stéphane MUSSARD

*Mémoire présenté en vue de l'obtention du diplôme
Master 2 Data Mining et Relation Client*

Département Marketing, Vente

4 juin 2019

Déclaration anti-plagiat

Je soussigné, Hugues GAILLARD, certifie que ce mémoire est un travail original.

Je certifie, de surcroît, que je n'ai ni recopié ni utilisé des idées ou des formulations tirées d'un ouvrage, article ou mémoire, en version imprimée ou électronique, sans mentionner précisément leur origine et que les citations intégrales sont signalées entre guillemets.

Conformément à la loi, le non-respect de ces dispositions me rend passible de poursuites devant la commission disciplinaire et les tribunaux de la République Française.

Signature : _____

Date : mardi 4 juin 2019

“

Remember that all models are wrong;
the practical question is how wrong do they
have to be to not be *useful*.

”

George E. P. Box, Norman R. Draper

Empirical Model-Building and Response Surface,
Wiley, 1987.

UNIVERSITÉ DE MONTPELLIER 1

Résumé

Montpellier Management

Master 2 Data Mining et Relation Client

Les modèles ARMA face aux réseaux de neurones récurrents

par Hugues GAILLARD

Ce mémoire présente et compare deux modèles couramment employés pour la modélisation et la prévision de séries temporelles : le modèle général ARMA et les réseaux de neurones récurrents. Il s'agit de modèles fréquemment utilisés pour la prévision du taux d'attrition en marketing, pour comprendre les tendances saisonnières des ventes ou encore pour la prévision de taux de conversion et le ciblage publicitaire. L'enjeu derrière l'étude de ces modèles est donc immense.

L'analyse théorique présentée en première partie permet de dégager les principaux points de différenciation des deux modèles : l'approche auto régressive des modèles ARMA est également modélisée par les neurones à mémoire, cependant la partie moyenne mobile n'est pas directement transposable aux réseaux de neurones récurrents. Ces derniers ont cependant la possibilité d'expliquer une relation non linéaire entre deux pas de temps. Quant à leurs constructions, les modèles ARMA nécessitent des paramètres (notamment p et q) basés sur l'étude des données passées alors que les modèles de réseaux de neurones récurrents ne nécessitent pas cette supposition : il s'agit d'apprentissage automatique, ou *machine learning*. Seuls des hyper-paramètres, relatifs à l'apprentissage des coefficients, doivent être déterminés.

L'exploration empirique proposée en deuxième partie met en exergue les différences de ces deux modèles en terme d'applications et de résultats. Plusieurs cas sont étudiés : des séries synthétisées suivant différentes structures temporelles, l'évolution de la production de bières en Australie et le rendement du Bitcoin.

La fin du mémoire présente un cas d'utilisation des réseaux de neurones autre que la modélisation de séries temporelles. Dans un dernier temps, la question de l'interprétabilité (du point de vue métier) des modèles basés sur les réseaux de neurones sera débattue.

Mots-clés : Data Mining, Série temporelle, ARMA, SARIMA, RNN, LSTM

Remerciements

Je tiens tout d'abord à remercier Monsieur Stéphane Mussard, professeur des Universités, mon directeur de mémoire, pour sa grande disponibilité, ses précieux conseils et l'aide qu'il m'a apportée quant au choix du sujet.

Je souhaite ensuite adresser mes remerciements aux personnes que j'ai rencontrées au sein de CGI Montpellier, notamment Claire Griesel, cheffe de projet, Michael Thebault, Kévin Chicaut et Valentin Palisse, ingénieurs en système d'information, pour leur temps passé à me former.

Je remercie également mon maître de stage, Cyrille Soliman, pour son accueil et la liberté qu'il m'a accordée dans le développement du générateur de jeux de données.

Enfin, je tiens à remercier Alice Lehmann, pour son soutien lors du temps d'écriture et pour sa relecture, ainsi que mes proches et ma famille qui ont contribué par leur bienveillance et leur confiance à l'aboutissement de ce mémoire.

Table des matières

Déclaration anti-plagiat	iii
Résumé	vii
Remerciements	ix
Introduction	1
Point de départ	1
Les modèles de série temporelle à travers le temps	2
Le modèle linéaire général	2
Du modèle linéaire au modèle auto-régressif	3
Réseaux de neurones et deep learning	3
La place des modèles traditionnels en statistiques aujourd'hui	4
Phénomène de sur-apprentissage	5
Difficultés et limites	6
 I Analyse théorique	 7
1 Le modèle ARMA	9
1.1 Brève présentation	9
1.2 Architecture du modèle ARMA	9
1.2.1 Partie AR	9
1.2.2 Partie MA	10
1.2.3 Modèle ARMA	10
1.3 Une hypothèse d'application contraignante	10
1.3.1 Série stationnaire	10
La différenciation	11
Test de stationnarité	11
Hypothèses sur les résidus	12
1.3.2 Validation d'un processus ARMA	12
1.4 ARIMA : une différenciation intégrée au modèle	12
1.5 La prise en compte de la saisonnalité dans SARIMA	13
1.6 Approche de Box & Jenkins pour l'identification du modèle	13
1.7 Optimisation des coefficients du modèle	14
 2 Les réseaux de neurones récurrents	 15
2.1 Brève présentation	15
2.2 Architecture d'un MLP	16
2.3 Modélisation des non-linéarités	16

2.3.1	Fonctions d'activation usuelles	17
2.4	Visualisation du fonctionnement d'un réseau de neurones	19
2.5	Méthode d'optimisation des poids synaptiques	20
2.5.1	Aparté sur l'évolution de la puissance de calcul	20
2.6	Algorithme d'implémentation d'un MLP	22
2.7	Critère d'arrêt de la descente de gradient	24
2.8	L'abandon pour éviter le sur-apprentissage	24
2.9	Apprentissage supervisé	25
2.10	Particularité des RNN : l'ajout de cellules à mémoire	25
2.11	Architectures particulières de réseaux de neurones récurrents	26
2.11.1	Les réseaux LSTM	26
	Réseau stateless ou stateful	27
2.11.2	Les GRU	28
2.12	Conditions d'application	28
2.12.1	Concernant l'hypothèse de stationnarité	28
2.12.2	Échelle des données	29
2.12.3	La couche de sortie	29
3	Confrontation des deux modèles	31
3.1	Points de ressemblances	31
3.2	Les différences	31
3.3	Généralisation des modèles ARMA et RNN aux séries multivariées	32
II	Analyse empirique	33
4	Présentation de la méthode	35
4.1	Pourquoi une seule série est insuffisante pour déterminer la performance d'un modèle	35
4.2	Méthode de benchmark	35
4.2.1	Introduction et subdivision des séries	35
4.2.2	Hyper-paramétrage des modèles	36
	La recherche par grille	36
	Identification d'un modèle SARIMA	37
	Optimisation d'un LSTM par Nesterov Adam	38
4.2.3	Ajout d'un modèle naïf	40
4.2.4	Méthode algorithmique de backtesting des modèles	40
4.2.5	Présentation des jeux de données et leurs utilités distinctes	41
4.2.6	Mesure de l'erreur	41
	Le Mean Squared Error	42
	La médiane des erreurs absolues	42
	Le pourcentage de variance expliquée	42
	Le coefficient de détermination	42
4.3	Configuration matérielle utilisée pour le benchmark	43

5	Analyse sur séries synthétiques	45
5.1	Tendance linéaire	45
5.1.1	Introduction	45
5.1.2	Modèle SARIMA	45
5.1.3	Modèle LSTM	45
5.1.4	Prévisions et performances	46
5.2	Tendance linéaire avec saisonnalité simple	46
5.2.1	Introduction	46
5.2.2	Modèle SARIMA	46
5.2.3	Modèle LSTM	47
5.2.4	Prévisions et performances	47
5.3	Tendance linéaire avec saisonnalité complexe	47
5.3.1	Introduction	47
5.3.2	Modèle SARIMA	47
5.3.3	Modèle LSTM	48
5.3.4	Prévisions et performances	48
5.4	Tendance non polynomiale et saisonnalités imbriquées	48
5.4.1	Introduction	48
5.4.2	Modèle SARIMA	49
5.4.3	Modèle LSTM	49
5.4.4	Prévisions et performances	49
6	Analyse sur données réelles	51
6.1	La production de bières en Australie	51
6.1.1	Introduction	51
6.1.2	Modèle SARIMA	51
6.1.3	Modèle LSTM	51
6.1.4	Prévisions et performances	52
6.2	Les rendements du Bitcoin	52
6.2.1	Introduction	52
6.2.2	Modèle SARIMA	52
6.2.3	Modèle LSTM	53
6.2.4	Prévisions et performances	53
	Conclusion	55
	Applications des RNN hors série temporelle	55
	Génération de musique	55
	Image captioning	55
	Modélisation du langage	56
	La poésie des LSTM	56
	Interprétabilité des résultats selon le modèle	60
	Résumé théorique et empirique	62
A	Informations complémentaires	63
A.1	Récapitulatif des notations employées	63

B	Jeux de données utilisés	65
B.1	Production de bières en Australie	65
B.2	Rendements du Bitcoin sur quatre heures	66
C	Code source	67
C.1	Programme de benchmark SARIMA - LSTM	67
C.2	Programme CharNN pour la génération de Fables	75
D	Résultats détaillés du benchmark	81
D.1	Série 1 - Tendance linéaire	81
D.2	Série 2 - Tendance linéaire avec saisonnalité simple	88
D.3	Série 3 - Tendance linéaire avec saisonnalité multiplicative	95
D.4	Série 4 - Tendance non polynomiale avec saisonnalités imbriquées	102
D.5	Série production de bières en Australie	109
D.6	Série rendement du Bitcoin	116
	Bibliographie	123

Table des figures

1	"Machine Learning" de xkcd	4
2	Exemple de sur-apprentissage/overfitting	5
2.1	Perceptron multi-couche à 3 entrées, 2 neurones en couche cachée et 1 sortie	17
2.2	Exemple de MLP entraîné	19
2.3	Benchmark OpenCL sur CPU et GPU	21
2.4	Architecture d'un RNN simple	26
4.1	Subdivision de la série en set d'entraînement, test et validation	36
6.1	Exemple d'image captioning	56
6.2	Architecture du générateur de Fables, 3 couches cachées	58
6.3	Évolution de la perte du modèle CharNN sur les Fables de La Fontaine . . .	61
D.1	Subdivision de la série 1 en set d'entraînement, test et validation	81
D.2	Diagnostic des résidus du modèle SARIMA pour la série 1	83
D.3	Test de Dickey-Fuller augmenté, série 1	84
D.4	Évolution de la perte du LSTM, série 1	85
D.5	Prévisions one step ahead, série 1	86
D.6	Prévisions dynamiques, série 1	86
D.7	Subdivision de la série 2 en set d'entraînement, test et validation	88
D.8	Diagnostic des résidus du modèle SARIMA pour la série 2	90
D.9	Test de Dickey-Fuller augmenté, série 2	91
D.10	Évolution de la perte du LSTM, série 2	92
D.11	Prévisions one step ahead, série 2	93
D.12	Prévisions dynamiques, série 2	93
D.13	Subdivision de la série 3 en set d'entraînement, test et validation	95
D.14	Diagnostic des résidus du modèle SARIMA pour la série 3	97
D.15	Test de Dickey-Fuller augmenté, série 3	98
D.16	Évolution de la perte du LSTM, série 3	99
D.17	Prévisions one step ahead, série 3	100
D.18	Prévisions dynamiques, série 3	100
D.19	Subdivision de la série 4 en set d'entraînement, test et validation	102
D.20	Diagnostic des résidus du modèle SARIMA pour la série 4	104
D.21	Test de Dickey-Fuller augmenté, série 4	105
D.22	Évolution de la perte du LSTM, série 4	106
D.23	Prévisions one step ahead, série 4	107
D.24	Prévisions dynamiques, série 4	107
D.25	Subdivision de la série production de bière en set d'entraînement, test et validation	109
D.26	Diagnostic des résidus du modèle SARIMA pour la série production de bières	111

D.27	Test de Dickey-Fuller augmenté, série production de bières	112
D.28	Évolution de la perte du LSTM, série production de bières	113
D.29	Prévisions one step ahead, série production de bières	114
D.30	Prévisions dynamiques, série production de bières	114
D.31	Subdivision de la série rendement du Bitcoin en set d'entraînement, test et validation	116
D.32	Diagnostic des résidus du modèle SARIMA pour la série rendement du Bitcoin	118
D.33	Test de Dickey-Fuller augmenté, série rendement du Bitcoin	119
D.34	Évolution de la perte du LSTM, série rendement du Bitcoin	120
D.35	Prévisions one step ahead, série rendement du Bitcoin	121
D.36	Prévisions dynamiques, série rendement du Bitcoin	121

Liste des tableaux

2.1	Fonctions d'activation usuelles	18
4.1	Grille de recherche des hyper-paramètres d'un modèle SARIMA	37
4.2	Grille de recherche des hyper-paramètres d'un modèle LSTM	37
A.1	Table des notations	63
D.1	Liste des modèles SARIMA testés sur la série 1	81
D.2	Liste des modèles LSTM testés sur la série 1	85
D.3	Liste des modèles SARIMA testés sur la série 2	88
D.4	Liste des modèles LSTM testés sur la série 2	92
D.5	Liste des modèles SARIMA testés sur la série 3	95
D.6	Liste des modèles LSTM testés sur la série 3	99
D.7	Liste des modèles SARIMA testés sur la série 4	102
D.8	Liste des modèles LSTM testés sur la série 4	106
D.9	Liste des modèles SARIMA testés sur la série production de bières	109
D.10	Liste des modèles LSTM testés sur la série production de bières	113
D.11	Liste des modèles SARIMA testés sur la série rendement du Bitcoin	116
D.12	Liste des modèles LSTM testés sur la série rendement du Bitcoin	120

Liste des abréviations

ACF	A utoc o rr e lation F unction
AdaGrad	A daptive G radient
ADF	A ugmented D ickey- F uller
AIC	A kaike I nformation C riterion
ARIMA	A utoregressive I ntegrated M oving A verage
ARMA	A utoregressive M oving A verage
CPU	C entral P rocessing U nit
GPU	G raphics P rocessing U nit
GRU	G ated R ecurrent U nit
LSTM	L ong S hort T erm M emory
MedAE	M edian A bsolute E rror
MLG	M odèle L inéaire G énéral
MLP	M ulti L ayer P erceptron
MSE	M ean S quared E rror
NADAM	N esterov-accelerated A daptive M oment
NLP	N atural L anguage P rocessing
PACF	P artial A utoc o rr e lation F unction
ReLU	R ectified L inear U nit
RNN	R ecurrent N eural N etwork
SARIMA	S easonal A utoregressive I ntegrated M oving A verage
SGD	S tochastic G radient D escent
Tanh	T angente h yperbolique

Introduction

Point de départ

Le mémoire et le stage de fin d'étude sont deux éléments obligatoires nécessaires à la validation du diplôme de Master 2. Étant passionné d'informatique et de statistiques, j'ai choisi de m'orienter vers le conseil en technologie de l'information et de la communication (TIC). C'est l'entreprise **CGI** qui a décidé de m'accueillir pour ces 6 mois de stage, en tant que *data engineer*¹.

Le *data engineer* est une personne travaillant en amont des *data analysts*, préparant ainsi les données avant de les analyser. Il s'agit alors principalement d'utiliser un logiciel dit d'*ETL*² et d'automatiser certains traitements sur les données.

Fondé en 1976 au Québec, **CGI** se classe parmi les cinq plus grands groupes mondiaux dans le secteur du conseil en informatique.

L'entreprise compte plusieurs antennes en France, notamment celle de Montpellier dans laquelle je réalise mon stage.

CGI travaille exclusivement avec des clients dits "grands comptes". Ainsi, à Montpellier par exemple, des équipes projets travaillent pour *Airbus*, *Dior* ou encore le groupe *Allianz*.

Le projet auquel j'ai été assigné concerne la banque *Société Générale*. Il s'agit d'un projet de pilotage financier et des risques qui vise notamment à s'assurer de la conformité de la banque aux accords de Bâle³.

Le sujet est donc très sensible puisqu'il nécessite un accès sécurisé à des bases de données comportant d'innombrables informations confidentielles. Ce projet nécessite la manipulation de données relatives aux emprunteurs (données personnelles, financières et légales), aux montants des prêts, aux taux d'intérêts négociés pour chaque contrat, et plus encore. Les différentes technologies employées et méthodes de travail sont également classées comme confidentielles.

En acceptant ce projet, je me suis donc engagé, dans le cadre d'un accord de non-divulgaration, à ne pas communiquer plus que ce qui a déjà été écrit.

Cette contrainte m'a donc amené à opter pour un sujet de mémoire davantage théorique qu'un récit de mon activité chez **CGI**.

Cependant, c'est avec enthousiasme que j'écris ces lignes puisque le sujet conjugue à la fois ma passion pour l'informatique et mon vif intérêt pour la modélisation et la prévision des séries chronologiques.

1. Data Engineer : Ingénieur en système d'information.

2. ETL : Extract Transform Load. Un ETL permet la création de flux de données.

3. Les différentes normes de Bâle constituent une réglementation visant à mieux appréhender les risques bancaires. Il s'agit alors principalement de calculer différents ratios de solvabilité. Cela implique un système d'information extrêmement complexe puisqu'il doit permettre d'agréger toutes les données relatives aux différents crédits et encours, afin de calculer précisément et régulièrement les indicateurs financiers.

Évolution des modèles de série temporelle

Depuis la nuit des temps, l'Homme cherche à anticiper les phénomènes auxquels il doit faire face. De la prévision de l'ensoleillement lors des débuts de l'agriculture à l'anticipation de l'afflux de clients en supermarché, les séries temporelles sont omniprésentes dans notre société. Comprendre et prévoir ces suites de valeurs dans le temps constitue donc un enjeu primordial et un atout stratégique majeur pour toute entreprise.

Aux balbutiements de l'économétrie, domaine pionnier dans l'étude scientifique des séries temporelles, les premiers modèles étaient bâtis sur des relations simples et **linéaires** entre la valeur expliquée et la ou les valeurs explicative(s). Nous retiendrons notamment le modèle linéaire général (abrégé MLG), encore très employé de nos jours grâce à sa simplicité d'utilisation et sa facilité d'interprétation. Le modèle linéaire servant de fondation aux modèles auto régressifs et même aux réseaux de neurones, les quelques lignes qui suivent présenteront les concepts importants à retenir pour la suite.

Le modèle linéaire général

Le modèle linéaire général consiste à déterminer la droite (appelée *droite de régression*) passant "au plus près" des points d'une série : c'est-à-dire minimisant l'erreur entre cette droite et les observations.

Le modèle linéaire univarié en série temporelle le plus simple s'exprime selon cette relation :

$$Y_t = \alpha + \beta X_t + \varepsilon_t$$

La variable à expliquer, Y , évolue en fonction du temps, tout comme la variable explicative, notée X . La droite comporte donc un coefficient directeur, noté β et une ordonnée à l'origine, notée α . Comme la relation linéaire n'est pas respectée en tout point (la série serait alors parfaitement prévisible), on rajoute un terme désignant les aléas, noté ε .

Cette variable synthétise l'ensemble des influences sur Y que seul X ne peut expliquer entièrement. Pour pouvoir utiliser ce modèle, il faut d'abord estimer ces paramètres α et β . La méthode d'optimisation de ces paramètres a pour but de minimiser l'erreur de prédiction, c'est-à-dire la différence entre la valeur estimée et la valeur attendue.

L'erreur pouvant être définie de plusieurs manières, celle retenue pour ce modèle est dite *quadratique*, c'est-à-dire la somme des carrés des résidus⁴.

Le problème d'optimisation du vecteur des paramètres du modèle φ est alors le suivant :

$$\varphi(\hat{\theta}(y)) = \min \sum_{i=1}^n (\hat{e}_i)^2 = \min \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

Ce problème est plus connu sous le nom de **critère des moindres carrés**.

Cette fonction d'évaluation de l'erreur est très importante car elle sera reprise plus tard dans les modèles plus avancés, notamment dans les réseaux de neurones. En effet, les poids synaptiques de ces réseaux peuvent être obtenus par la résolution d'un problème d'optimisation (minimisation) de l'erreur quadratique moyenne. La fonction "carré" a plusieurs avantages, notamment la simplification des calculs de part ses propriétés de dérivabilité et sa progression non linéaire, négligeant les "petites" erreurs mais pénalisant les plus grandes.

4. Le résidu est calculé comme la différence absolue entre la valeur théorique de la série et la valeur estimée du modèle.

Le mémoire présenté se focalisera sur les modèles univariés : c'est-à-dire les modèles prenant en entrée une série temporelle dans le but d'en prédire une autre. De plus, si la valeur future de la série peut être expliquée grâce à un modèle n'ayant connaissance que des valeurs passées de la série, ce modèle est alors qualifié d'auto-régressif.

Les modèles présentés (SARIMA et LSTM) ont cependant l'avantage d'être utilisables en statistique multivariée.

Du modèle linéaire au modèle auto-régressif

La section suivante permet de faire une transition de la régression linéaire aux modèles auto-régressifs. Le modèle complet ARMA et ses hypothèses sont détaillés dans la partie 1 Le modèle ARMA.

Admettons que le modèle linéaire simple décrit plus haut serve maintenant à expliquer une série temporelle, notée X , en utilisant seulement sa dernière valeur passée. Ce modèle, devenu auto-régressif, s'écrit alors :

$$X_t = \beta X_{t-1} + \varepsilon_t$$

Notez la disparition de l'ordonnée à l'origine α : ce paramètre additif est remplacé par les résultats des régressions précédentes (modèles à plusieurs retards). Puisque le modèle n'a qu'un seul retard, la simplification du problème fait disparaître le paramètre α .

Finalement, une auto-régression linéaire simple à p retards s'écrit :

$$X_t = \sum_{i=1}^p \varphi_i X_{t-i} + \varepsilon_t$$

La notation employée est importante : le β cède sa place à φ dans les modèles auto-régressifs.

Les modèles auto-régressifs sont très utilisés pour la prédiction de séries temporelles mais ce ne sont pas les seuls ; des réseaux de neurones adaptés aux séries temporelles ont également fait leur apparition vers la fin des années 1980, notamment avec les travaux de HOPFIELD, 1982.

Réseaux de neurones et deep learning

Ces dernières années, le *deep learning*, ou apprentissage profond, s'impose de plus en plus comme modèle "bon à tout faire" : conduite autonome, marketing prédictif, reconnaissance (d'images, de paroles, d'écriture manuscrite)... Les utilisations sont nombreuses et certaines révolutionnent déjà notre vie quotidienne. Ces modèles sont en effet très performants pour des tâches de régression (prévision d'une valeur réelle, tel qu'un prix par exemple) comme pour des tâches de classification (attribution d'une catégorie, comme la détection de visage sur une photo).

Le *deep learning* désigne en fait des architectures de réseaux de neurones dits "profonds", c'est-à-dire empilant un grand nombre de couches de neurones. Ces modèles sont détaillés plus tard (partie 2 Les réseaux de neurones récurrents).

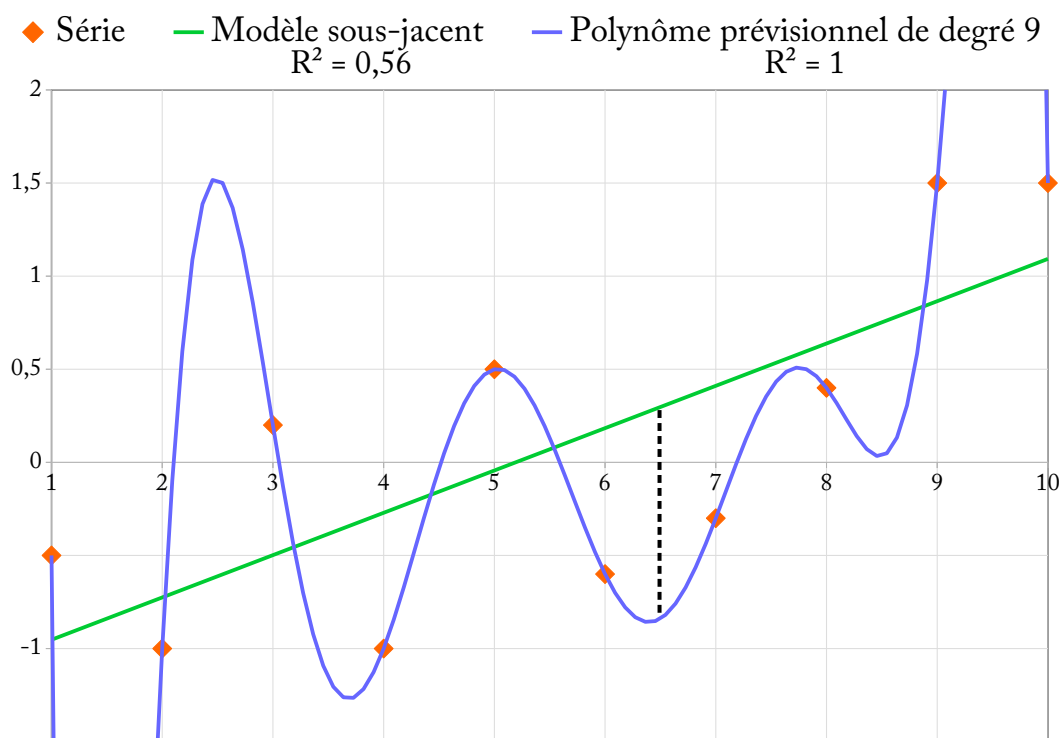
défaut : il devient alors possible de faire tout dire à un réseau de neurones. Les problèmes de sur-apprentissage (ou *overfitting*) sont alors très fréquents.

Phénomène de sur-apprentissage

L'overfitting désigne le phénomène statistique où le modèle se conforme trop étroitement aux données présentées lors de son optimisation (sa phase d'apprentissage). Les indicateurs de précision du modèle sont alors excellents; cependant sa capacité de généralisation à de nouvelles données n'est pas bonne. Le modèle fournit alors des prévisions erronées et devient inutilisable.

L'exemple ci-dessous permet de se rendre compte de ce phénomène. Une série chronologique est générée par une droite et l'ajout d'un bruit (d'où un $R^2 = 0,56$). Un modèle polynomial de degré 9 est alors employé pour estimer les valeurs de la série. Nous constatons que l'erreur théorique du modèle est nulle ($R^2 = 1$). Malgré cela, le modèle n'est pas correct pour faire de la prévision : nous pouvons constater une immense erreur au point 6.5, la valeur estimée est très en dessous de la valeur moyenne réelle du modèle ayant généré la série. Cela est dû au degré élevé du polynôme (égal au nombre de points -1), qui force la courbe estimée à se caler au plus près des points de la série.⁷

FIGURE 2 – Exemple de sur-apprentissage/overfitting



Le très brillant mathématicien John von Neumann disait d'ailleurs à ce sujet :

"With four parameters I can fit an elephant, and with five I can make him wiggle his trunk."

Cette citation expose alors bien le problème du sur-apprentissage et de la difficulté à entraîner des réseaux de neurones contenant, au bas mot, quelques milliers de paramètres.

7. Ce type de polynôme est un polynôme de Lagrange (ou interpolation Lagrangienne).

Pour contourner ce problème, la partie 4 Présentation de la méthode propose deux solutions : la validation croisée et l'utilisation d'une méthode de régularisation (dropout) dédiée aux réseaux de neurones.

Ce mémoire a pour ambition de présenter les modèles adaptés aux séries auto régressives : les modèles basés sur ARMA et les réseaux de neurones récurrents (architecture LSTM) dans une première partie. Une méthode de test de performance est développée en deuxième partie. Celle-ci propose une approche algorithmique objective permettant de comparer la précision de chacun de ces modèles sur des séries chronologiques aux structures variées. En dernier lieu, la pertinence de ces nouveaux modèles, basés sur les réseaux de neurones, sera discutée afin de déterminer s'ils sont aptes à un cadre d'utilisation aussi étendu que la théorie mathématique le laisse entendre.

Difficultés et limites

L'étude des séries temporelles est un vaste sujet et ce mémoire n'effleure que brièvement les modèles auto-régressifs de Box & Jenkins et les réseaux de neurones récurrents. Ainsi, avec le court intervalle de temps ayant permis la rédaction de ce mémoire, quelques difficultés ont été rencontrées. Ce faisant, les variantes plus ou moins exotiques des modèles présentés (NARMA, FARIMA, GRU, ...) n'ont pas été abordées. Le lecteur intéressé pourra alors consulter la partie bibliographique afin de parfaire ses connaissances par la lecture des ouvrages et ressources cités. Le temps passé à la détermination, aux calculs, aux tests et à la prévision des modèles n'a également pas permis l'utilisation optimale des modèles présentés. Les réseaux de neurones nécessitent notamment de longues heures de test et de nombreux paramétrages vers une solution acceptable. Il n'a pas été possible d'utiliser toutes les configurations possibles des modèles SARIMA et LSTM : certaines configurations ont été mises de côté afin d'accélérer les temps des calculs et sont, potentiellement, plus performantes que les modèles exposés.

Les résultats présentés, notamment dans la partie II Analyse empirique, n'ont pas vocation à être exhaustifs, ni d'indiquer la supériorité d'un modèle en toute situation.

Première partie

Analyse théorique

Chapitre 1

Le modèle ARMA

1.1 Brève présentation

Le modèle ARMA, est un modèle dédié à l'étude de séries temporelles. La première application des modèles auto régressifs date des années 1920, avec les travaux de G. U Yule et J. Walker. La méthode fut ensuite décrite par WHITTLE, 1951 puis démocratisée par Box et al., 1970. Ces mêmes auteurs proposent une méthode éponyme, qui permet l'estimation des paramètres du modèle via des outils heuristiques basés sur l'auto-corrélation.

Le modèle ARMA est encore aujourd'hui très utilisé dans plusieurs domaines, notamment en finance, dans des variantes plus complexes. Le modèle GARCH permet par exemple une modélisation de processus hétéroscédastiques, inhérentes aux cours boursiers. Le mémoire s'intéresse spécifiquement à une évolution du modèle standard ARMA : le modèle SARIMA. Ce dernier est plus souple que le modèle ARMA puisqu'il modélise une différenciation¹ et les effets d'une saisonnalité.

Comme présenté en introduction, le modèle ARMA se compose de deux parties : la première, AR (pour *Auto Regressive*) et la deuxième, MA (pour *Moving Average*).

Ces deux parties sont détaillées dans les sections suivantes.

1.2 Architecture du modèle ARMA

1.2.1 Partie AR

La partie AR désigne la partie auto régressive du modèle ARMA. Un AR est un modèle se basant sur la relation de dépendance entre une variable et cette même variable, retardée d'un certain pas de temps.

Ainsi une variable x évoluant au pas de temps t et suivant un modèle auto-régressif simple s'écrit :

$$x_t = \alpha_1 x_{t-1} + \dots + \alpha_p x_{t-p} + w_t = \sum_{i=1}^P \alpha_i x_{t-i} + w_t$$

Avec w_t un processus bruit blanc ($E[w_t] = 0$, $E[w_t^2] = \sigma^2$, $E[w_t w_u] = 0 \forall t \neq u$).

Le paramètre P est l'ordre du modèle. Il s'agit du nombre d'observations retardées à conserver pour expliquer x .

1. La différenciation, en statistiques, est une transformation appliquée à une série chronologique dans le but de la rendre stationnaire.

Une marche aléatoire peut alors être définie comme un processus AR(1) avec $\alpha = 1$:

$$x_t = 1x_{t-1} + w_t$$

Avec un coefficient $\alpha \neq 1$, le processus x_t devient auto corrélé. La fonction d'auto-corrélation devient ensuite décroissante.

1.2.2 Partie MA

La partie MA a un fonctionnement similaire à celle de la partie auto-régressive. En effet, au lieu de régresser la variable par rapport à elle-même, il est question de modéliser la variable par rapport aux termes d'erreurs.

L'ordre du processus moyenne mobile est noté Q . Un processus MA(Q) s'écrit alors ainsi :

$$x_t = w_t + \beta_1 w_{t-1} + \dots + \beta_q w_{t-q} = w_t + \sum_{i=1}^Q \beta_i w_{t-i}$$

Avec w_t un processus bruit blanc.

1.2.3 Modèle ARMA

Finalement, le modèle ARMA(p, q) est la combinaison d'un modèle AR(p) et MA(q).

Un modèle de série temporelle x_t est un modèle ARMA(p, q) si :

$$\begin{aligned} x_t &= \alpha_1 x_{t-1} + \alpha_2 x_{t-2} + \dots + \alpha_p x_{t-p} + w_t + \beta_1 w_{t-1} + \beta_2 w_{t-2} \dots + \beta_q w_{t-q} \\ &= \sum_{i=1}^P \alpha_i x_{t-i} + w_t + \sum_{i=1}^Q \beta_i w_{t-i} \end{aligned}$$

Avec w_t un processus bruit blanc.

1.3 Une hypothèse d'application contraignante

1.3.1 Série stationnaire

L'hypothèse la plus forte pour l'application d'un modèle ARMA sur une série temporelle est que cette dernière doit être **stationnaire**.

Une série stationnaire x_t est une série dont la structure est invariante par translation dans le temps. Ainsi :

1. L'espérance de la série est constante dans le temps.
2. La covariance entre x_t et x_{t+h} , $\forall t \in [0, n]$ est invariante dans le temps.

En pratique, il est très difficile d'obtenir une série brute déjà stationnaire. En effet, la seule présence d'une tendance ou d'une saisonnalité rend la série non stationnaire.

On procède donc à sa *stationnarisation*.

Une première technique permettant de fixer, parfois, la dispersion d'une série est le passage en valeur *log* (transformation logarithmique). Cette technique est cependant peu efficace, notamment sur les séries comportant de la saisonnalité.

Finalement, la technique qui sera retenue pour la suite de ce mémoire est la différenciation. C'est d'ailleurs cette méthode que l'on retrouve directement intégrée au modèle ARIMA.

La différenciation

La différenciation permet de stabiliser la moyenne d'une série temporelle et ainsi d'éliminer la dépendance temporelle.

La différenciation première consiste à retrancher à l'observation t , l'observation $t - 1$. L'opérateur de différenciation est donc défini comme : $\nabla X_t = X_t - X_{t-1}$. Évidemment, le processus inverse permet de récupérer la série d'origine : $x_t = \nabla X_t + X_{t-1}$. Cette opération peut être répétée plusieurs fois, notamment lorsque la tendance n'est pas linéaire. Ainsi, pour supprimer une tendance quadratique, la transformation $\nabla^2 X_t$ peut être appliquée. Le nombre de fois où cet opérateur est appliqué est appelé l'ordre de différenciation.

Cette méthode peut être reproduite pour supprimer non pas l'effet d'une tendance mais d'une saisonnalité. On parle alors d'ordre de désaisonnalisation (utilisé pour le modèle SARIMA). Ainsi, pour des données mensuelles, si l'on souhaite supprimer les effets d'une saisonnalité annuelle, il est intéressant de choisir un ordre de désaisonnalisation de 12.

Il est important de noter que pour un ordre de désaisonnalisation d , les d premières valeurs de la série ne peuvent plus être utilisées.

Test de stationnarité

Pour vérifier qu'une série est bien stationnaire, il existe plusieurs méthodes, notamment :

- **L'approche graphique**, qui consiste à visualiser la série et déterminer si elle présente une composante tendancielle et/ou saisonnière.
- **Faire de la statistique descriptive sur des sous-ensembles de la série**, afin de regarder si, par exemple, la moyenne et la variance d'une sous-série diffère d'une autre.
- **Les tests statistiques**, comme le test de Dickey-Fuller augmenté. Cette approche est plus robuste que les deux citées précédemment et est décrite plus-bas.

Le test de Dickey-Fuller augmenté est un test de racine unitaire et se base sur la corrélation entre la série et une tendance modélisée par un modèle autoregressif.

L'hypothèse nulle d'un test de Dickey-Fuller indique la présence d'une racine unitaire, la série n'est donc pas stationnaire puisqu'elle possède une structure dépendante du temps. L'hypothèse alternative (rejet de l'hypothèse nulle) est que la série ne possède pas de racine unitaire ; la série est donc stationnaire.

Le test de Dickey-Fuller fonctionne en calculant une statistique adf et en la comparant à un seuil dépendant du niveau de risque pris par le test (généralement 5%). Il est également possible de comparer directement la p-value au seuil choisi.

Ainsi, pour un $ADF > \text{seuil}$, l'hypothèse nulle n'est pas rejetée et la série possède une racine unitaire : elle n'est donc pas stationnaire. Au contraire, pour une $ADF \leq \text{seuil}$, l'hypothèse nulle est rejetée et la série est stationnaire.

Hypothèses sur les résidus

Il est important que le modèle, une fois construit, satisfasse plusieurs hypothèses sur ses résidus. Ces hypothèses, ainsi que leurs tests associés sont :

- **Normalité des résidus** : il est possible de réaliser un test de symétrie de la distribution des erreurs du modèle (test de Skewness) et un test d'aplatissement (test de Kurtosis). Ces deux tests précédemment cités peuvent également se combiner en test de Jarque-Bera, permettant de vérifier simultanément les deux hypothèses (symétrie et aplatissement) et ainsi confirmer que la distribution des erreurs obéit bien à une loi normale centrée réduite.
- **Homoscédasticité** : il s'agit ici de tester si la variance des erreurs du modèle ARMA est constante dans le temps, quel que soit le sous-échantillon (de taille suffisante) choisi. Il existe plusieurs tests pour vérifier cette hypothèse. Nous pouvons citer le test de White, le test ARCH d'hétéroscédasticité ou encore le test de Goldfeld et Quandt qui par exemple s'appuie sur une loi de Fisher puis teste sur deux sous-échantillons l'écart entre la somme du carré des erreurs.
- **Non-autocorrélation** : il est important de vérifier que les erreurs ne soient pas autocorrélées. Cela indique souvent que les paramètres du modèle ont mal été choisis ou que la série n'est pas bien déstationarisée. Le test d'autocorrélation des résidus se fait notamment avec le test de Ljung-Box qui permet de tester rapidement l'autocorrélation de l'ordre 1 à n . Le test de Durbin-Watson, valable uniquement pour l'autocorrélation à l'ordre 1, est quant à lui, rarement utilisé.

1.3.2 Validation d'un processus ARMA

Comme nous le verrons par la suite, plusieurs modèles ARMA peuvent convenir à une même série temporelle. Pour les départager, nous privilégierons les modèles dont les coefficients estimés sont statistiquement différents de 0 (via un t -test) et dont l'examen des résidus nous indiquent que ces derniers suivent bien un processus bruit blanc : $e_t \sim w_t(0, \sigma_e^2)$.

1.4 ARIMA : une différenciation intégrée au modèle

Vu l'importance que prend l'hypothèse de stationnarité pour les modèles ARMA, la méthode de différenciation a été intégrée directement dans le modèle ARMA : c'est l'apparition du modèle ARIMA.

Le modèle ARIMA est alors défini par un paramètre supplémentaire, d , qui définit l'ordre de différenciation à appliquer sur la série brute avant d'estimer les paramètres p et q .

Il est important de noter ici que le paramètre d est un entier. Pour des séries comportant une structure de long terme particulière, il est intéressant d'utiliser un paramètre de différenciation non entier permettant de modéliser l'effet d'une dérive de la structure. C'est le rôle des modèles FARIMA qui permettent d'utiliser le paramètre d sous forme fractionnaire².

2. d est alors compris dans l'intervalle $[-0.5, 0.5]$

1.5 La prise en compte de la saisonnalité dans SARIMA

SARIMA signifie *Seasonal AutoRegressive Integrated Moving Average*. Il s'agit d'un modèle basé sur le modèle ARIMA auquel une partie dédiée à la modélisation d'une saisonnalité est rajoutée.

Ce modèle est donc très utilisé dès lors que les données présentent des caractéristiques saisonnières (horaires, mensuelles...). Cette caractéristique est très présente en science humaine où les effets des saisons, des jours de la semaine, des heures, jouent un rôle important dans notre comportement.

La partie saisonnière rajoutée au modèle SARIMA fonctionne de la même manière que la partie classique ARIMA à l'exception que cette structure s'applique sur la série différenciée par l'ordre saisonnier D avec des ordres autoregressifs et moyenne mobile, P et Q , potentiellement distincts des ordres p et q du modèle ARIMA.³

Une série x_t suit un modèle SARIMA(p,d,q)(P,D,Q) si l'égalité suivante⁴ est respectée :

$$\nabla_s^D \times \Phi_s(I - \nabla)^s \times \nabla^d \times \Phi(I - \nabla) \times x_t = \Theta_s(I - \nabla^s) \times \Theta(I - \nabla) \times w_t$$

1.6 Approche de Box & Jenkins pour l'identification du modèle

Une fois que la série a été transformée (*via* Box-Cox, log...) ou différenciée, la méthode de Box & Jenkins permet de déterminer l'ordre autoregressif p et l'ordre de la moyenne mobile q du modèle ARMA.

Cette méthode ne permet pas la détermination du modèle optimal car plusieurs solutions sont possibles et peuvent échapper à cette technique. Elle constitue cependant une base solide et demeure l'une des plus utilisées.

À partir des équations de Yule Walker, il peut être montré qu'un processus AR(p) a une fonction d'autocorrélation simple (ACF) décroissante de manière exponentielle. Ses autocorrélations partielles (PACF) ne sont plus significatives à partir de l'ordre $p + 1$.

De la même manière, l'ordre d'un processus MA(q) peut être choisi en choisissant l'ordre pour lequel l'ACF devient non significative. C'est-à-dire si $\rho(h) = 0$ lorsque $h > q$, h étant le retard. Sa PACF décroît de manière exponentielle.

Les deux ordres p et q peuvent donc être théoriquement déterminés par l'examen des graphiques d'autocorrélation et autocorrélation partielle (corrélogramme). En pratique, l'analyse des fonctions d'autocorrélation ne suffit pas à déterminer complètement les ordres et plusieurs modèles peuvent être conservés. Il conviendra alors de garder le modèle minimisant le critère d'information choisi, souvent le critère d'information d'Akaike, abrégé AIC :

$$AIC(p, q) = \log \hat{\sigma}^2 + \frac{2(p + q)}{N}$$

Avec N , le nombre d'observations et $\hat{\sigma}^2$ l'estimation de la variance du bruit blanc.

3. La stationnarisation s'opère en réalité en éliminant d'abord la périodicité par différence à l'ordre D puis en retirant la tendance, grâce à l'ordre de différence d .

4. Il est possible de substituer dans cette écriture l'opérateur de différenciation ∇ par l'opérateur de retard B , tel que $\nabla = I - B$, avec I , l'application identique.

Il est apparent que lorsque p ou q augmente d'une unité, la variance estimée diminue et le critère d'Akaike pénalise donc cette augmentation : c'est le principe de parcimonie.

Note : la sélection des ordres des séries de la partie II se fera sur une technique similaire à la recherche par grille et non selon l'approche de Box & Jenkins. Les raisons sont développés dans la partie 4.2.2 Identification d'un modèle SARIMA.

1.7 Optimisation des coefficients du modèle

Une fois les ordres du modèle $ARIMA(p,d,q)$ choisis, il faut en estimer les coefficients. Pour cela, il est possible de résoudre les équations de Yule Walker sous l'hypothèse que les autocorrélations empiriques sont un estimateur convergeant des autocorrélations. Une autre méthode consiste à utiliser l'optimisation par moindres carrés conditionnels (estimateur du maximum de vraisemblance)⁵.

L'estimation des paramètres d'un modèle SARIMA est un sujet compliqué qui dépasse l'objectif de ce mémoire et ne sera pas davantage développé ici.

5. C'est la méthode employée par la bibliothèque Python *Statsmodels* lors des simulations dans la partie II.

Chapitre 2

Les réseaux de neurones récurrents

2.1 Brève présentation

Les réseaux de neurones sont une puissante famille d'algorithmes, capables de s'attaquer à des problèmes très divers : des tâches de classification en passant par la régression, rares sont les problèmes qu'ils ne peuvent résoudre. Ils sont par exemple utilisés pour faire de la reconnaissance d'images, de la traduction de texte, de la détection de fraudes et bien entendu de la prévision de séries temporelles.

Le premier véritable réseau de neurones à avoir été inventé s'appelle le perceptron. Créé par ROSENBLATT, 1958, ce modèle n'est finalement qu'un *classifieur* linéaire. Le perceptron est simplement une combinaison linéaire de ses entrées par des coefficients, couramment appelés *poids synaptiques*. Afin de réduire les erreurs de prédiction du réseau, les poids doivent être optimisés : c'est la fameuse étape d'**apprentissage** d'un réseau de neurones.

Cependant, ce modèle ne sera vraiment rendu populaire que bien plus tard, sous une autre forme : le perceptron multi-couches (abrégié **MLP**, pour Multi-Layer Perceptron). Ce modèle plus complexe est capable de classifier des données linéairement non-séparables. Le perceptron multi-couches a été rendu possible par une technique d'apprentissage nommée *rétropropagation*, mise au point par RUMELHART, HINTON, WILLIAMS et al., 1988.

Ce modèle deviendra la base de **toutes** les variantes de réseaux de neurones utilisées aujourd'hui. Pour cette raison, le fonctionnement d'un MLP sera détaillé dans les sections suivantes.

D'autres architectures de réseaux de neurones sont apparues par la suite, dont deux, régulièrement employées pour atteindre les performances de l'état de l'art.

Il s'agit de l'architecture en *convolution*, dédiée au traitement de données en deux dimensions. Cette architecture se prête ainsi parfaitement aux images : classification d'une photographie, reconnaissance d'objet ou même génération d'une image complètement imaginée par le réseau de neurones (on parle alors d'*hallucination*).

La deuxième structure de réseau de neurones est celle qui va nous intéresser : il s'agit des réseaux de neurones récurrents. Pour ces modèles, les neurones qui constituent le réseau sont dotés d'une capacité à "mémoriser" les données qu'ils ont déjà rencontrées.

Ces modèles sont dédiés à la modélisation de séries temporelles. De la même manière que les modèles ARMA, la prévision faite à l'instant t dépend des données à l'instant $t - 1$, $t - 2$, ..., $t - n$.

2.2 Architecture d'un MLP

Un réseau de neurones est constitué de neurones répartis à travers trois couches, ou plus. Chaque couche contient un ou plusieurs neurones. Le nombre de neurones des couches d'entrée et de sortie est fixé par le jeu de données. Le nombre de neurones pour la couche d'entrée est égal au nombre de variables explicatives et le nombre de neurones pour la couche de sortie est égal au nombre de variables à expliquer par le modèle.

On distingue ainsi, dans l'ordre :

- **La couche d'entrée** : cette couche contient un ou plusieurs neurone(s) doté(s) d'une valeur. Cette valeur est la donnée d'entrée du réseau (notée X_n pour un réseau à n entrées). Chacun des neurones de cette couche est relié à **tous** les autres neurones de la couche suivante : la couche cachée. Les neurones de la couche d'entrée ont pour seul but de transmettre leurs valeurs aux neurones de la couche cachée. A l'instar des synapses pour un cerveau biologique, ces valeurs sont transmises *via* une connexion. Les valeurs traversant ces connexions sont pondérées par un poids, appelé poids synaptique.
- **La couche cachée (ou intermédiaire)** : la couche cachée est la couche contenant des neurones dotés d'entrées et de sorties. Ils reçoivent donc en entrée les valeurs reçues par la couche d'entrée (c'est-à-dire les valeurs d'entrée du modèle pondérées par les poids synaptiques). Un neurone de la couche cachée somme ensuite ces valeurs et les retransmet à la couche suivante, qui peut être soit une autre couche cachée (il est possible d'empiler autant de couches cachées que l'on souhaite), soit la couche de sortie. Les valeurs envoyées par les neurones de la couche cachée passent à travers le même système de connexion décrit précédemment : les valeurs sont multipliées par un poids synaptique avant d'atteindre la couche suivante.

La valeur du neurone i de la première couche cachée est donc calculée ainsi :

$$H_i^1 = \sum_{n=1}^N (X_n \times w_{n,1}^1) = \langle X, W_1 \rangle$$

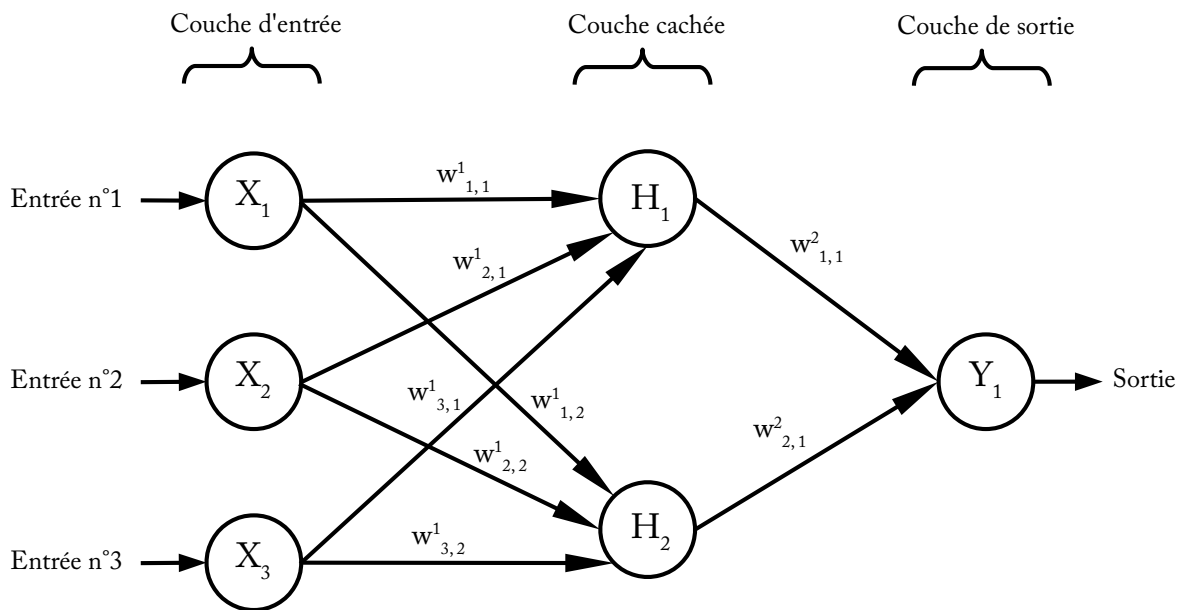
La deuxième écriture repose sur le produit scalaire du vecteur d'entrée X par la matrice de poids W_1 (située entre la couche d'entrée et la première couche cachée), parfois appelée matrice de synapses puisque reliant les neurones entre eux.

- **La couche de sortie** : les neurones de la couche de sortie reçoivent en entrée les valeurs de sortie de la couche cachée (pondérées par les poids synaptiques). Comme pour les autres couches, la valeur d'un neurone de sortie est la somme des valeurs de sortie des neurones de la couche précédente auxquelles il est attaché. La valeur d'un neurone de sortie est directement interprétable : il s'agit de la valeur prédite par le modèle.

2.3 Modélisation des non-linéarités

L'ajout de la couche cachée est ce qui distingue le MLP d'un perceptron simple. Il s'agit là d'une avancée importante puisque cette couche cachée permet la modélisation d'une relation non linéaire entre les variables explicatives et la/les variable(s) endogène(s).

FIGURE 2.1 – Perceptron multi-couche à 3 entrées, 2 neurones en couche cachée et 1 sortie



Cependant, le simple fait de rajouter une couche de combinaisons linéaires ne suffit pas à modéliser une corrélation non linéaire puisque ces nouvelles combinaisons linéaires pourraient être directement intégrées dans la première couche, ramenant le MLP à un perceptron simple.

La solution consiste à ne pas faire transiter **directement** la valeur d'un neurone aux suivants. La valeur devra passer à l'intérieur d'une fonction non linéaire avant d'être transmise au(x) neurone(s) suivant(s). Ces fonctions permettant aux perceptrons multi-couches d'être capables de créer de la corrélation non linéaire entre les entrées et les sorties sont nommées **fonctions d'activation**.

La valeur du neurone i de la première couche cachée équipée d'une fonction d'activation f est maintenant calculée ainsi :

$$H_i^1 = f\left(\sum_{n=1}^N (X_n \times w_{n,1}^1)\right) = f(\langle X, W_1 \rangle)$$

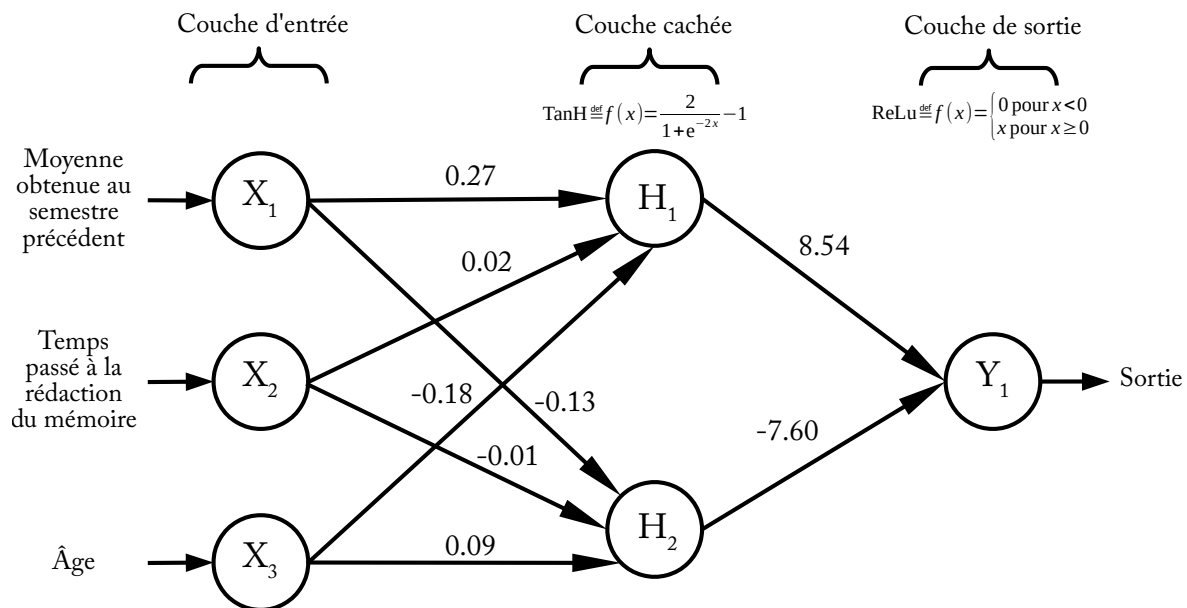
2.3.1 Fonctions d'activation usuelles

Dans la littérature scientifique, seules quelques fonctions d'activation sont réellement utilisées. Le choix dépendra du type de sortie voulue. Le tableau Fonctions d'activation usuelles 2.1 permet de connaître les principales fonctions d'activation retenues communément.

Nom	Expression	Étendue	Utilité
Marche / Fonction d'Heaviside	$f(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}$	$\{0; 1\}$	Première fonction à être utilisée dans le perceptron. On la considèrerait, il y a quelques temps, comme étant la fonction s'approchant le plus du fonctionnement d'un neurone biologique. Aujourd'hui détrônée par les fonctions rampes (ReLU), cette fonction est rarement utilisée en pratique.
Sigmoïde / Logistique	$f(x) = \frac{1}{1 + e^{-x}}$	$[0; 1]$	Très utilisé, elle souffre cependant de problèmes : elle n'est pas centrée en 0 ($= 0.5$) et entraîne une saturation des gradients et donc une convergence lente.
Tangente hyperbolique	$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$	$[-1; 1]$	Cette fonction est généralement préférée à la sigmoïde pour les couches cachées. Elle est centrée et facilite l'optimisation.
ReLU	$f(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}$	$[0; \infty[$	La Rectified Linear Unit est également utilisée pour les couches cachées.
Soft Max	$\sigma(x)_j = \frac{e^{x_j}}{\sum_{k=1}^K e^{x_k}}$		Modélisation de probabilités (utilisée en couche de sortie dans les modèles de classification). Avec K , le nombre de réels à "transformer" en probabilités.

TABLE 2.1 – Fonctions d'activation usuelles

FIGURE 2.2 – Exemple de MLP entraîné



2.4 Visualisation du fonctionnement d'un réseau de neurones

La section suivante présente un cas pratique d'utilisation d'un MLP pour de la régression.

Un étudiant souhaite prévoir la moyenne qu'il obtiendra à son dernier semestre. Pour cela, il regroupe un ensemble d'observations parmi les anciens étudiants de sa formation.

Il obtient ainsi les données suivantes pour chaque étudiant interrogé :

1. La moyenne obtenue au premier semestre (X_1)
2. Le temps (en heures) passé à la rédaction du mémoire (X_2)
3. L'âge (X_3)

Pour cette tâche simple, l'étudiant choisit donc une architecture avec 3 entrées, une couche cachée composée de deux neurones et finalement une seule sortie (la note prévue pour l'année). Il choisit comme fonction d'activation une fonction ReLu pour la couche intermédiaire et une sigmoïde pour la couche de sortie.

Après entraînement du modèle, il obtient le réseau de neurones donné par la figure 2.2 Exemple de MLP entraîné.

Une fois le modèle entraîné, il est possible de l'utiliser à des fins de prévision. Il faut pratiquer alors une **propagation avant** des données d'entrée jusqu'à la sortie. Cette étape est plus communément appelée *feed forward propagation*.

Pour le cas de l'étudiant, il est donné : $X_1 = 14$, $X_2 = 165$, $X_3 = 24$.

La couche cachée est alors déterminée par :

$$H_1 := f\left(\sum_{i=1}^3 X_i \times w_{i,1}^1\right) = f(14 \times 0.27 + 165 \times 0.02 + 24 \times -0.18) = \tanh(2.76) \approx 0.99$$

$$\begin{aligned}
H_2 &:= f\left(\sum_{i=1}^3 X_i \times w_{i,2}^1\right) = f(14 \times -0.13 + 165 \times -0.01 + 24 \times 0.09) \\
&= \tanh(-1.31) \approx -0.86
\end{aligned}$$

La note attendue est calculée avec les deux valeurs précédemment calculées pour les couches cachées :

$$Y_1 := f\left(\sum_{i=1}^2 H_i \times w_{i,1}^2\right) \approx f(8.54 * 0.99 + (-7.60) * -0.86) \approx \text{ReLu}(14.99) \approx 14.99$$

2.5 Méthode d'optimisation des poids synaptiques

La détermination des poids synaptiques w constitue la phase d'entraînement du modèle. Le modèle cherche alors les valeurs optimales des poids minimisant les erreurs commises.

La mesure de l'erreur communément retenue est la distance euclidienne de l'erreur. L'erreur est également appelée perte (ou *loss* en anglais).

Pour un échantillon de données de taille n , l'erreur totale à minimiser est :

$$loss = \sum_{i=1}^n (\hat{y}_i - y_i)^2$$

L'algorithme communément utilisé pour minimiser ces poids s'appelle la **descente de gradient**. Cet algorithme a plusieurs variantes, plus ou moins complexes et spécialisées à un domaine d'application mais le principe général reste le même.

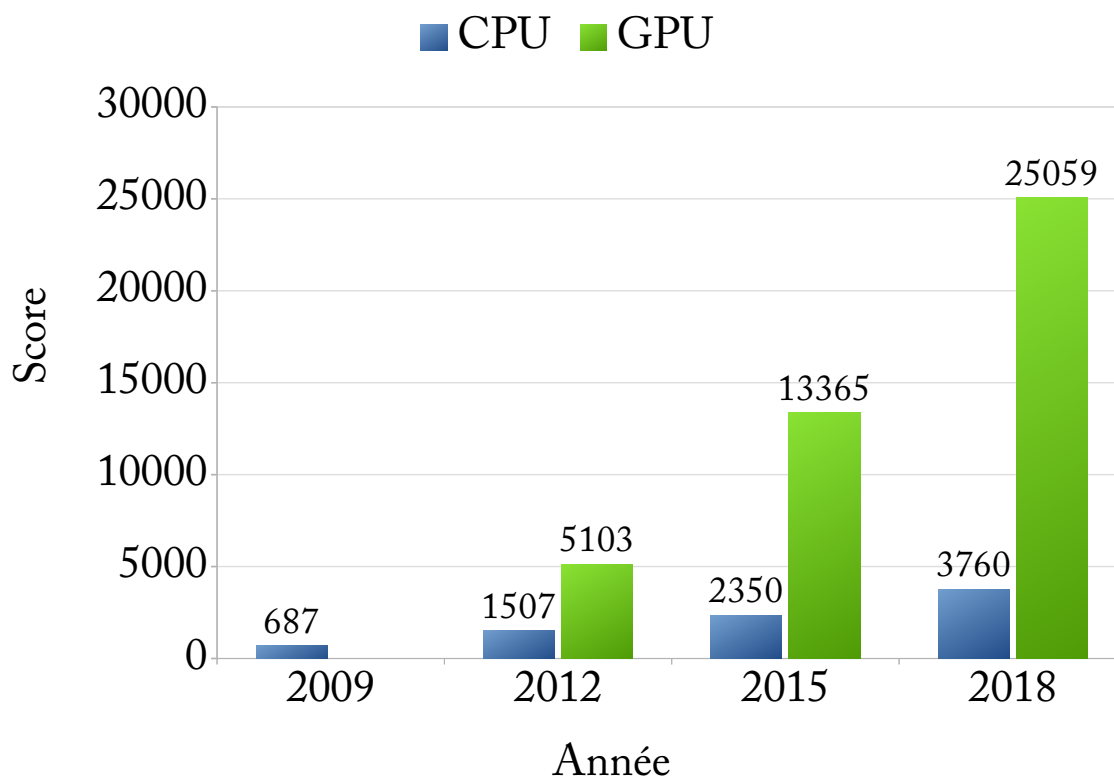
Les poids du réseau à optimiser sont généralement initialisés avec des valeurs aléatoires. Puis, la descente de gradient est utilisée : cette opération est très lourde en termes de calculs et donc en temps de traitement.

2.5.1 Aparté sur l'évolution de la puissance de calcul

Le développement de nouvelles architectures matérielles de calcul a rendu possible l'entraînement de réseaux de neurones profond (*deep learning*). D'abord optimisé par CPU (processeur d'ordinateur commun soit *CPU*, central processing unit), le potentiel de parallélisation de la descente de gradient a permis l'entraînement de ces réseaux via GPU (carte graphique). Ces derniers sont bien plus performants pour les tâches hautement parallélisables.

Le graphique 2.3 Benchmark OpenCL sur CPU et GPU permet de se rendre compte de l'explosion de la puissance de calcul sur ces dix dernières années, à **budget constant** (environ 450€, prix de lancement, par CPU/GPU).

FIGURE 2.3 – Benchmark OpenCL sur CPU et GPU



Ces chiffres sont tirés du test LuxMark¹ permettant de juger la puissance de calcul d'un composant en lui faisant passer un benchmark OpenCL. OpenCL est une plateforme permettant d'écrire du code pouvant être exécuté indépendamment sur CPU ou GPU. Plus le score est élevé, plus le traitement est rapide.

Les composants suivants ont été retenus pour l'étude : Intel Core i7 970 (2009), Intel Core i7 3370 et GTX 680 (2012), Intel Core i7 6700K et GTX 870 (2015), Intel Core i7 9700K et RTX 2060 (2018).

Pour placer les choses en perspective : un réseau de neurones nécessitant 18 heures d'entraînement en 2009 ne nécessite plus que 30 minutes d'entraînement en 2018 pour le même budget.

Remarque : aucun GPU datant de 2009 n'a été benchmarké puisqu'il n'existait pas encore de frameworks permettant d'entraîner des réseaux neuronaux via GPU. TensorFlow, un des premiers frameworks de Deep Learning supportant l'entraînement via GPU, n'a été dévoilé au grand public qu'en 2015 (passage en licence open-source, rendant son adoption fulgurante).

Le principe de la descente de gradient se compose de deux étapes. La première est l'étape de propagation avant, *forward propagation*, comme nous l'avons vu précédemment, qui consiste à faire une prédiction. La deuxième étape consiste à parcourir le chemin inverse : c'est l'étape de propagation arrière, plus communément appelée *backward propagation*. Il

1. LuxMark : La scène LuxBall HDR a été utilisée pour les mesures.

Page de résultats : http://www.luxmark.info/top_results/LuxBall%20HDR/OpenCL/

s'agit d'appliquer aux poids de chaque couche, un delta correspondant au gradient de cette couche, pondéré par un coefficient d'apprentissage.

La mise à jour des poids w peut donc se résumer ainsi, avec Q la fonction de perte à minimiser, dépendante des poids du réseau :

$$w := w - \alpha \nabla Q(w).$$

Le coefficient d'apprentissage (également appelé *learning rate*), noté α , permet de réduire l'impact de la mise à jour des poids : il peut arriver qu'une modification trop importante des poids entraîne non pas une baisse mais une hausse de l'erreur. Cet hyperparamètre du modèle connaît alors un arbitrage. Une faible valeur pour α entraîne une convergence lente vers les poids optimaux (le minimum de la fonction d'erreur n'est pas assurément global, il peut rester local pour plusieurs raisons). Inversement, une valeur élevée du paramètre d'apprentissage augmente les risques de "saut" : le modèle passe alors à un ensemble de poids augmentant l'erreur totale, c'est le problème de l'*exploding gradient*, très courant lors de la phase d'apprentissage. L'algorithme est divergent. Il convient alors de surveiller l'évolution de l'erreur/perte pendant la phase d'apprentissage. Lorsque ce problème survient, le modèle devient alors instable et inutilisable.

Il existe néanmoins plusieurs solutions pour contourner ce problème. Nous pouvons citer les méthodes de régularisations (L1 et L2) et notamment le modèle LSTM, que nous détaillerons plus tard, qui est exempt (sous certaines conditions) de ce phénomène.

2.6 Algorithme d'implémentation d'un MLP

Le pseudo code suivant permet de se familiariser avec la phase d'apprentissage d'un MLP à une couche cachée, contenant une fonction d'activation ReLU.

La descente de gradient vue ici est la version **stochastique** de la descente de gradient. Il existe notamment une alternative, la descente de gradient **AdaGrad** (pour *Adaptive Gradient*). Cette version calcule un deuxième paramètre d'apprentissage qui pondère la mise à jour des poids du réseau. Lors d'une descente rapide (gradient fort), le taux d'apprentissage diminuera l'impact de la mise à jour alors qu'il augmentera la mise à jour des poids pour un gradient faible (apprentissage lent).

Des algorithmes plus avancés, comme RMSProp et Adam, reprennent ce principe de correction du taux d'apprentissage en pondérant le gradient par les moments de second ordre de ce gradient (et ajoute de ce fait une notion de *momentum* / d'élan donné à la descente).

D'autres algorithmes d'optimisation sont basés sur un principe complètement différent de la descente de gradient. L'algorithme L-BFGS-B fait partie des méthodes quasi-Newton et permet, généralement, de trouver une combinaison de poids synaptiques minimisant la fonction d'erreur, en un nombre d'itérations plus faible que la descente de gradient traditionnelle. Cependant, chaque itération nécessite des étapes de calcul supplémentaires. La littérature à ce sujet tend à conseiller l'utilisation des méthodes quasi-Newton sur des jeux de données à faible dimension (Le et al., 2011).

Algorithme 1 : MLP et descente de gradient

N = taille de l'échantillon;
 I = nombre de neurones en couche d'entrée / nombre de variables explicatives;
 H = nombre de neurones en couche cachée;
 O = nombre de neurones en couche de sortie / nombre de variables à expliquer;
 X = matrice de taille (N, I) , contenant les séries explicatives;
 Y = matrice de taille (N, O) , contenant les séries à expliquer;
 $W1$ = matrice de poids reliant la couche d'entrée à la couche cachée de taille (I, H) ;
 $W2$ = matrice de poids reliant la couche cachée à la couche de sortie de taille (H, O) ;
 α = taux d'apprentissage
pour chaque *itération* i **faire**
 $h = \langle X, W1 \rangle$;
 $h_relu = \max(h, 0)$;
 $y_pred = \langle h_relu, W2 \rangle$;
 $loss = \sum (y_pred - Y)^2$;
 $grad_y_pred = 2 \times (y_pred - Y)$;
 $grad_w2 = \langle h_relu, grad_y_pred \rangle$;
 $grad_h_relu = \langle grad_y_pred, W2^T \rangle$;
 $grad_h = \max(grad_h_relu, 0)$;
 $grad_w1 = \langle X^T, grad_h \rangle$;
 $W1 = W1 - (\alpha \times grad_w1)$;
 $W2 = W2 - (\alpha \times grad_w2)$;
fin

2.7 Critère d'arrêt de la descente de gradient

L'algorithme précédemment décrit est itératif et n'a pas de condition d'arrêt explicite. Dans la littérature, il existe plusieurs critères d'arrêt pouvant être utilisés :

1. **Le nombre d'itérations** : l'algorithme s'arrête après un certain nombre d'itérations. A l'arrêt, ce critère n'assure pas d'avoir obtenu une convergence vers l'optimum mais a cependant l'utilité de pouvoir être utilisé pour comparer l'efficacité d'une architecture avec une autre, de benchmarker une plateforme de calcul, etc. Une alternative à cette méthode consiste à laisser l'algorithme itérer pendant x secondes.
2. **La fonction d'erreur atteint un seuil suffisant** : la descente de gradient s'arrête lorsque la fonction d'erreur atteint ϵ , le seuil pour lequel l'erreur est jugée acceptable.
3. **Le gradient de l'erreur est faible** : chaque itération de l'algorithme n'apporte alors qu'une mise à jour très faible des poids. La descente est bloquée : soit l'optimum a été atteint, soit l'erreur est restée sur un point-selle.

Il n'existe pas de règle préétablie : ainsi le critère d'arrêt 1 est régulièrement utilisé pendant les phases de prototypage de l'architecture alors que le critère 3 est plus souvent utilisé lors de la phase de mise en production du modèle.

Ainsi, la partie II Analyse empirique utilise un critère d'arrêt hybride. 500 itérations sont nécessaires mais un arrêt prématuré est possible si la perte sur **l'ensemble de validation** n'a pas connue une baisse sur les 30 dernières itérations.

2.8 L'abandon pour éviter le sur-apprentissage

La descente de gradient permet généralement de converger vers une solution minimisant la fonction d'erreur évaluée sur un jeu de données d'entraînement. Malheureusement, rien n'indique que cette solution peut s'étendre à d'autres jeux de données et les phénomènes de sur-apprentissage sont très communs lors de l'emploi de réseaux de neurones.

Pour pallier ce problème, la technique la plus commune est celle de l'abandon de neurones ou *dropout*. Le *dropout* consiste à choisir un taux p qui déterminera la part de neurones à désactiver dans une couche à chaque itération **pendant la phase d'apprentissage**. Généralement, le taux $p = 0.2$ est utilisé, ce qui signifie qu'à chaque itération de la descente de gradient, la couche visée par le *dropout* ne pourra entraîner que 80% des neurones de cette couche. Les neurones à désactiver sont choisis aléatoirement et leur sortie est évaluée à zéro. Comme une partie des neurones est désactivée, les poids synaptiques auront tendance à compenser ce manque en augmentant en valeur absolue. Ainsi, pendant la phase d'utilisation réelle du modèle, tous les neurones sont réactivés et leur activation est réduite d'un facteur p .

Cette technique de régularisation est très efficace pour lutter contre le sur-apprentissage car elle force le réseau à se focaliser sur les caractéristiques d'entrées les plus robustes. Elle a cependant le désavantage de ralentir la convergence.

Pour la partie II Analyse empirique, les modèles LSTM seront testés avec deux taux de dropout différents : $p = 0$ (pas de dropout) et $p = 0.2$.

2.9 Apprentissage supervisé

Comme nous avons pu le voir précédemment, un réseau de neurones optimisé ne fait que trouver de la corrélation entre sa couche d'entrée et sa couche de sortie. Il y a donc bien deux jeux de données distincts. Pour pouvoir utiliser un réseau de neurones dans le cas d'une régression univariée, nous avons alors besoin de créer une base de données dédiée à l'apprentissage supervisé.

L'apprentissage supervisé consiste à faire apprendre une fonction (un modèle) permettant de relier des observations à un vecteur cible. Pour une tâche de régression, il s'agit donc de construire un jeu de données contenant un ensemble de couples : $(x_n, y_n)_{1 \leq n \leq N}$ avec $x \in \mathbb{R}^I$ et $y \in \mathbb{R}$. Pour rappel, I est la dimension de l'entrée du réseau de neurones.

L'algorithme 2 permet de générer un jeu de données d'apprentissage supervisé à partir d'une série temporelle.

Algorithme 2 : Découpage d'une série pour de l'apprentissage supervisé

Données :

N = taille de la série;

Y = série temporelle;

I = nombre de neurones en couche d'entrée / nombre de variables explicatives;

Résultat :

X = jeu de données d'entrée (initialisé à vide);

y = jeu de données de sortie (initialisé à vide);

pour chaque *itération* $i < N$ **faire**

$index_fin = i + I$;

si $index_fin > N - 1$ **alors**

 arrêt;

fin

$vecteur_x = Y_{i \leq n < index_fin}$;

$cible = Y_{index_fin}$;

 ajouter à X vecteur_ x ;

 ajouter à y cible ;

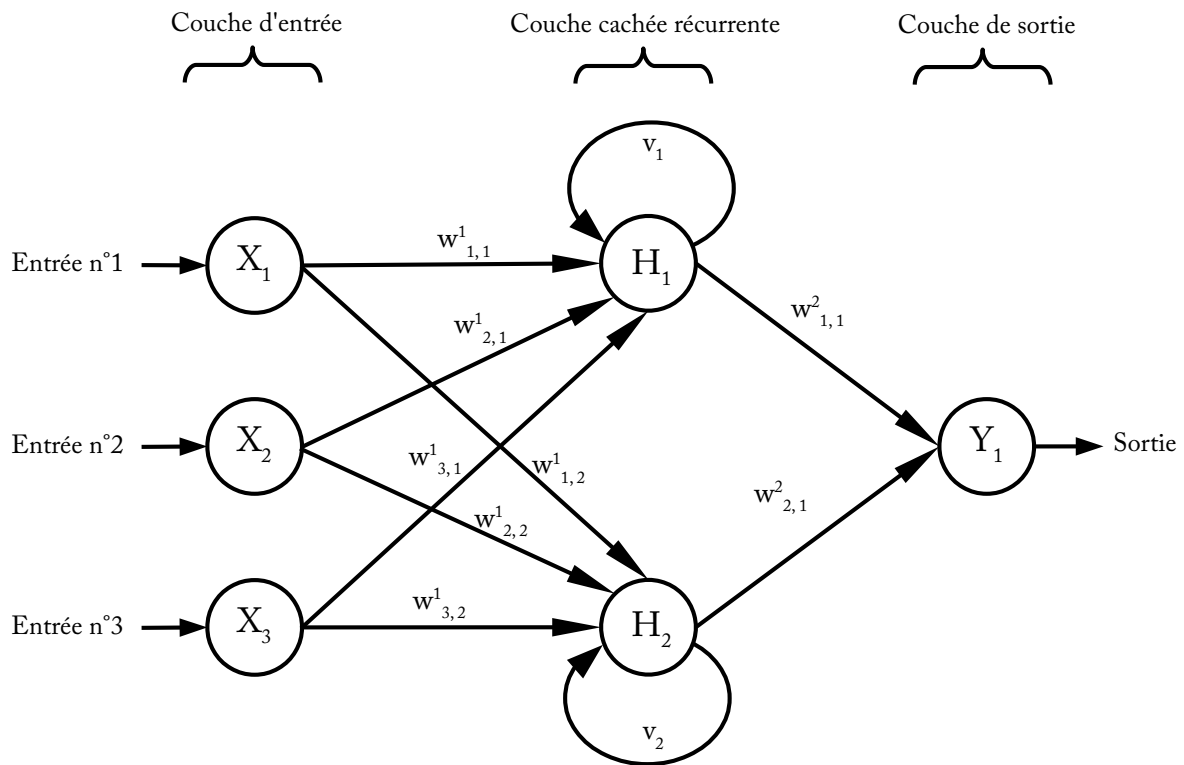
fin

Nous retrouverons cet algorithme plus tard, dans la fonction Python `decouper_serie_apprentissage_supervise(serie, taille)` (fichier *fonctions.py*).

2.10 Particularité des RNN : l'ajout de cellules à mémoire

Alors que les MLP vus précédemment sont des méthodes à propagation avant simple, les réseaux de neurones récurrents (ou RNN, *Recurrent Neural Network*) sont des modèles dits acycliques. Il existe donc au moins une connexion récurrente : un neurone reçoit alors en entrée, en plus de l'information présente dans un MLP, une deuxième valeur transmise par ce même neurone. Cette valeur supplémentaire fait office de mémoire. Les différentes

FIGURE 2.4 – Architecture d'un RNN simple



architectures de réseaux de neurones récurrents (GRU, LSTM...) se distinguent notamment par la modélisation de cette mémoire.

La connexion récurrente peut se faire sur n'importe quelle couche, y compris la couche d'entrée. Cette dernière possibilité a un désavantage non négligeable : puisqu'il n'y a qu'une seule connexion récurrente, la couche d'entrée recevra son entrée habituelle (identique au perceptron) + la valeur d'entrée utilisée à l'itération précédente. La "mémoire" d'un réseau ayant ses connexions récurrentes en couche d'entrée n'est donc que d'un seul pas de temps. La possibilité d'ajouter une connexion récurrente dans la couche cachée est alors admirable : le réseau de neurones a la capacité de modéliser un état caché, qui lui permet de donner un sens à des données rencontrées plusieurs pas de temps auparavant.

2.11 Architectures particulières de réseaux de neurones récurrents

2.11.1 Les réseaux LSTM

Initialement proposée par HOCHREITER et SCHMIDHUBER, 1997, l'architecture LSTM est aujourd'hui une des plus populaires pour la modélisation de connexions récurrentes dans les réseaux de neurones. Un réseau LSTM résout partiellement le problème de la disparition et de l'explosion de gradient.

Un réseau LSTM ou *Long Short Term Memory* est un réseau de neurones doté d'une mémoire à court et long terme. Cette mémoire est modélisée grâce à un système de 3 portes.

La cellule (le neurone) prend en entrée l'état caché précédent (ou mémoire) ainsi que l'entrée classique.

La première porte est dédiée à l'entrée de nouvelles données dans la mémoire (*input gate*), cette fonction permet de propager, ou pas, les données d'entrées et donc de modifier l'état caché de la cellule (c'est-à-dire l'information sur les données passées : la mémoire).

La deuxième porte est une porte d'oubli (*forget gate*). Cette porte décide si l'information doit être gardée ou non. La fonction d'oubli utilise une fonction sigmoïde retournant un résultat compris entre zéro et un : une valeur proche de 0 signifie que l'information a peu d'importance et peut être oubliée, une valeur proche de 1, au contraire, conserve l'information. C'est cette porte qui permet notamment aux LSTM d'être exempts du problème de *vanishing/exploding gradient*. En effet, cette porte va limiter les valeurs de la cellule à 1 (le gradient ne peut plus exploser²) et interdire la modification de la mémoire lorsque le gradient est trop petit (le gradient ne disparaît plus).

La troisième porte est la porte de sortie (*output gate*). La porte de sortie calcule le prochain état caché de la cellule.

Le système d'équations suivant permet de modéliser une cellule LSTM :

$$\begin{aligned} i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\ f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\ g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\ o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\ c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\ h_t &= o_t \odot \tanh(c_t) \end{aligned}$$

x_t est le vecteur d'entrée à l'instant t , h_t est l'état caché de la cellule à l'instant t , c_t est l'état de la cellule à l'instant t . i_t , f_t , g_t , o_t sont respectivement les portes d'entrée, d'oubli, l'état de la cellule et la sortie à l'instant t . σ est la fonction sigmoïde et $\odot$ est le produit matriciel de Hadamard.

L'architecture Long Short Term Memory est devenue un des modèles de référence dans la prévision de séries temporelles. Les LSTM ont été utilisés avec une meilleure précision que les modèles ARIMA pour certains domaines. C'est le cas pour de la prévision de long terme sur des sujets de ventes au détail, permettant ainsi d'éviter le gâchi alimentaire (ELMASDOTTER et NYSTRÖMER, 2018).

Réseau stateless ou stateful

Il existe deux implémentations possibles pour un réseau LSTM ; l'une avec conservation de l'état précédent, l'autre sans persistance de l'état. Pour un réseau sans conservation de l'état (plus couramment appelé *stateless*), la mémoire du réseau persiste uniquement pour la durée de la séquence passée en entrée. Il n'y a donc pas de différence entre faire passer les valeurs de la sous-série x_1 puis les valeurs de la sous-série x_2 , ou l'inverse.

A contrario, un réseau à état (*stateful*) comporte une mémoire qui persiste après le passage d'une séquence. L'ordre de passage des sous-séries est donc ici très important puisque

2. En réalité, le modèle LSTM est toujours sujet à l'explosion de gradient mais dans une moindre mesure, voir travaux de GREFF et al., 2016

le réseau utilisera la mémoire acquise lors du passage de la première sous-série pour modéliser la valeur future (et mettre à jour sa mémoire pour la prochaine itération).

Un réseau *stateful* a donc l'avantage de pouvoir accepter des séquences de taille non connue à l'avance et est de ce fait, très utilisé en traitement automatique du langage naturel (NLP). Par exemple, un mot comporte un nombre variable de lettres et un réseau LSTM *stateful* peut accepter en entrée une séquence de taille non-fixe de lettres (un mot) pour prévoir une autre séquence de taille non-fixe (un autre mot). La mémoire doit être réinitialisée explicitement (fonction `reset_states()` avec Keras en Python) lorsque la séquence est terminée (un espace pour délimiter les séquences de mots).

Pour des tâches de régression, il n'existe pas de différences de performance entre un modèle *stateful* et *stateless*. La séquence de sortie est de taille finie (une seule observation pour prévoir l'état en $t + 1$) et la séquence d'entrée est variable mais connue à l'avance (voir section 2.9 Apprentissage supervisé et sous-section 4.2.2 Hyper-paramétrage des modèles). Puisque la mémoire est automatiquement réinitialisée à la fin d'une séquence, le réseau LSTM *stateless* a été choisi du fait de sa simplicité d'utilisation pour la partie II Analyse empirique.

2.11.2 Les GRU

Les modèles GRU, ou *Gated Recurrent Unit*, sont des modèles de réseaux de neurones récurrents apparus après les modèles LSTM. Ils ont été introduit en 2014 par les travaux de CHO et al., 2014. Les GRU sont très similaires au LSTM dans la mesure où ils fonctionnent selon le même système de portes. Ils sont cependant dépourvus de la porte de sortie et possèdent donc, par nature, moins de paramètres que les LSTM. La sortie d'une cellule GRU dépend alors de sa sortie précédente pondérée par sa part d'"oublie" et de sa part de "mémoire" pondérée par l'état actuel de la cellule.

En reprenant les notations précédentes, le nouvel état caché à l'instant t devient :

$$h_t = (1 - f_t) \odot h_{t-1} + f_t \odot (W_h x_t + W_{hi} (\sigma(W_r x_t + W_r h_{t-1} + b_r) \odot h_{t-1}) + b_h)$$

Leurs capacités prédictives n'en sont cependant pas dégradées, du moins pour certaines tâches (CHUNG et al., 2014), et leur entraînement est plus rapide que les réseaux LSTM.

Bien qu'intéressants, ces modèles ne seront pas davantage étudiés par la suite.

2.12 Conditions d'application

Tout comme les modèles Box & Jenkins, les réseaux de neurones nécessitent des conditions vérifiées sur le modèle et les données, cependant celles-ci sont moins contraignantes.

2.12.1 Concernant l'hypothèse de stationnarité

L'architecture des réseaux de neurones offre une très grande flexibilité et une puissante capacité de modélisation non-linéaire et permet notamment de s'affranchir des conditions de stationnarité.

Cependant, pour que la comparaison avec les modèles auto régressifs reste neutre, l'entraînement se fera sur les séries stationnarisées.

De plus, pour des séries en tendance marquée, le réseau peut avoir du mal à prévoir une valeur comprise en dehors de la plage de valeurs rencontrées pendant la phase d'apprentissage, notamment si le réseau utilise des fonctions d'activation bornées sur un ensemble fini.

2.12.2 Échelle des données

Contrairement aux modèles ARMA, dont le problème d'optimisation des paramètres n'est pas sensible à l'ordre de grandeur des observations, les réseaux de neurones simples sont limités à ne recevoir en entrée que des données comprises dans le domaine de définition non saturé des fonctions d'activation.

Par exemple, pour une couche d'entrée équipée d'une fonction d'activation en tangente hyperbolique, la valeur des données pondérées n'appartenant pas à l'intervalle $\approx [-2; 2]$ ne pourra pas être distinguée par les couches de traitement suivantes et l'information contenue dans les valeurs extrêmes est donc perdue.

Deux méthodes sont couramment employées pour la réduction de l'amplitude des données et leurs normalisations :

1. La méthode de centrage-réduction des données :

$$X_{\text{centré réduit}} = \frac{X - \bar{X}}{\sigma_X}$$

2. La méthode "Min-Max" qui consiste à contraindre la valeur dans un intervalle, généralement $[0; 1]$:

$$X_{\text{normalisé}} = \frac{X - X_{\min}}{X_{\max} - X_{\min}}$$

Bien entendu, les statistiques de la série (moyenne, écart-type, min/max) doivent être calculées sur le jeu d'entraînement, sans quoi un biais serait induit (voir 4.2.1 Introduction et subdivision des séries).

La méthode de contrainte MinMax est celle qui sera retenue pour la partie II Analyse empirique.

Note : il est également possible d'ajouter une première couche cachée dont la fonction d'activation permet le "rétrécissement" de l'intervalle des données d'entrée. Ainsi, le modèle "apprendra" par lui-même à redimensionner les entrées pour ne pas perdre d'informations dans les couches suivantes. Par exemple, une fonction logistique modifiée comme $f(x) = \frac{1}{1+e^{0.1 \times x}}$ peut traiter des amplitudes de données plus grandes qu'une fonction $f(x) = \tanh(x)$.

2.12.3 La couche de sortie

Pour des tâches de régression, l'utilisation d'une fonction d'activation linéaire est souvent employée. Celle-ci ne favorise pas la transmission d'une valeur particulière, contrairement à la tangente hyperbolique par exemple, qui dévie la valeur nulle vers les asymptotes.

Chapitre 3

Confrontation des deux modèles

Ce chapitre propose un ensemble de points comparatifs entre les modèles issus de la méthode Box & Jenkins et les réseaux de neurones récurrents.

3.1 Points de ressemblances

Le premier point de similitude évident est qu'un modèle SARIMA ou LSTM, est adapté à la prévision de série temporelle. Les deux modèles prennent en entrée une série temporelle dans le but de prédire une valeur future de cette série.

De la même manière, bien que les équations définissant le modèles soient très distinctes, le principe d'optimisation des paramètres repose sur le même principe : la minimisation d'une mesure d'erreur. Ainsi, les paramètres d'un modèle SARIMA peuvent être optimisés par la méthode du maximum de vraisemblance. Les poids d'un réseau de neurones sont, eux, optimisés par un algorithme itératif qui permet de trouver un potentiel minimum dans la fonction de perte : la descente de gradient.

Les deux modèles possèdent également un énorme point commun : la partie autorégressive. Dans un modèle ARMA, la partie AR tient compte des valeurs passées de la série, au même titre que la mémoire d'une cellule de réseau de neurones récurrents. La partie autorégressive d'un modèle ARMA est cependant très limitée puisque nous ne considérons généralement pas plus de cinq retards. *A contrario*, les LSTM suffisamment larges sont dotés d'une mémoire pouvant gérer plus d'une centaine de pas de temps confortablement.

3.2 Les différences

Bien que les deux modèles proposent de recevoir une série de nombres réels en entrée, les traitements à appliquer sur les entrées sont différents.

Un modèle SARIMA ne nécessite pas vraiment de stationnarisation préalable de la série puisqu'il inclut de base des méthodes de différenciations (la différenciation de tendance et la différenciation de saisonnalité). De la même manière, un réseau LSTM, de par son architecture, n'a pas besoin de recevoir une série stationnaire. Une sous-partie du réseau peut alors "apprendre" à stationnariser la série si les couches suivantes ont réellement besoin de recevoir une série stabilisée. Cependant, en pratique, il est vivement recommandé de traiter la série avant l'utilisation d'un réseau de neurones.

Le modèle LSTM est également bien plus complexe et difficile à entraîner qu'un modèle SARIMA. La convergence des paramètres ARMA est assurée tant que la série est stable et les

ordres corrects. En revanche, rien n'assure la convergence de la perte pour une configuration donnée de réseaux neuronaux. La convergence dépend d'un grand nombre de facteurs, notamment de l'architecture et des paramètres d'apprentissage choisis.

Cependant, la complexité des modèles LSTM cache un énorme avantage : le modèle apprend de lui-même la structure qu'il doit développer afin de s'ajuster au mieux à la série. Les modèles Box & Jenkins nécessitent une étude préalable du jeu de données afin de déterminer les paramètres idéaux.

La sortie de ces modèles est également différente. Par construction, les modèles SARIMA ne peuvent prévoir qu'une valeur à la fois, la valeur suivante de la série. La sortie d'un modèle LSTM est quant à elle bien plus universelle puisqu'elle permet de créer autant de sorties voulues. Les RNN sont donc adaptés à des champs d'application plus larges que les modèles SARIMA classiques.

3.3 Généralisation des modèles ARMA et RNN aux séries multivariées

Il est possible d'étendre les modèles ARMA et RNN en les généralisant aux séries multivariées.

Ainsi, le modèle SARIMA, dont la performance est testée dans la prochaine partie, a une extension adaptée à la statistique multivariée : c'est le modèle SARIMAX. Le modèle prend alors une entrée supplémentaire : un vecteur contenant les variables exogènes.

Les modèles de réseaux de neurones récurrents peuvent, eux, simplement utiliser en couche d'entrée d'une dimension égale aux nombres de variables explicatives. Le principe théorique sous-jacent pour la phase d'entraînement reste alors le même.

Deuxième partie

Analyse empirique

Chapitre 4

Présentation de la méthode

Dans cette partie, nous nous intéresserons à l'analyse empirique des deux approches : le modèle auto-régressif standard et le modèle de réseau de neurones récurrents. Il s'agira de tester les deux méthodes sur des jeux de données synthétiques et réels, comparer leurs performances et robustesse et comprendre leurs limites.

4.1 Pourquoi une seule série est insuffisante pour déterminer la performance d'un modèle

Une série temporelle présente de nombreuses caractéristiques. Un modèle peut très bien capter la tendance d'une série tout en s'abstenant de modéliser la saisonnalité, par exemple (c'est le cas du MLG). Il faut donc tester les modèles sur des séries présentant des caractéristiques distinctes. Les cas classiques sont étudiés dans la partie consacrée à l'analyse des séries synthétiques (tendance linéaire parfaite, ...) alors que la partie sur l'analyse de données réelles se concentrera davantage sur des séries générées par des composantes plus complexes (saisonnalité imparfaite, bruit, ...).

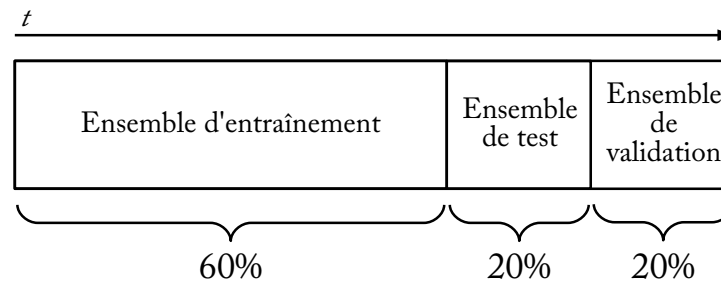
4.2 Méthode de benchmark

4.2.1 Introduction et subdivision des séries

Afin de pouvoir comparer les deux modèles de manière automatisée et en introduisant le moins de biais possible, une méthode algorithmique a été développée pour effectuer les tests de performance (backtest). De plus, l'entraînement des modèles et leurs paramétrages se font selon les méthodes décrites dans la partie relative à l'analyse théorique. Chaque série est ainsi divisée en 3 sous-séries, découpée selon des proportions fixées à l'avance :

1. **Ensemble d'entraînement** : ce jeu représente la plus grande partie de la série (60%). Il sert au calibrage des paramètres du modèle (étape de *fit*). Cette étape correspond à la détermination des paramètres ϕ_i et θ_i pour les modèles auto-régressifs moyenne mobile et l'ensemble des poids synaptiques W_i^j pour les réseaux de neurones.
2. **Ensemble de test** : cet ensemble de données est utilisé pour l'hyper-paramétrage des modèles, c'est-à-dire la détermination de l'architecture idéale à utiliser pour mieux "coller" aux données (on parle alors de *fitting*). Pour un modèle ARMA, il s'agit donc de trouver les paramètres p et q optimaux alors qu'il s'agira de trouver les paramètres α et le nombre de couches cachées pour le réseau de neurones récurrents.

FIGURE 4.1 – Subdivision de la série en set d'entraînement, test et validation



3. **Ensemble de validation** : cette dernière sous-série n'est utilisée que pour mesurer la performance finale du modèle. Il est important de ne pas utiliser ce jeu de données lors des phases d'apprentissages/paramétrages puisque cela conduirait à un biais (notamment de l'overfitting déguisé).

Puisqu'il s'agit de séries temporelles, le processus dépendant du temps générateur de la série doit être conservé : les observations ne peuvent être prises indépendamment. Pour la suite de l'étude, les 60% premières observations de la série serviront à l'entraînement du modèle, les 20% observations suivantes à l'ensemble de test et les 20% dernières observations serviront à la validation du modèle.

Cette méthode s'appelle la validation croisée ou *cross validation*.

4.2.2 Hyper-paramétrage des modèles

Les hyper-paramètres sont des paramètres propres au modèle et qui ne dépendent pas des données. Ils ne peuvent pas être déduits par le modèle. De plus, la valeur de ces paramètres doit être estimée avant la phase d'apprentissage. Le nombre de neurones d'une couche cachée pour un modèle de réseau de neurones est un bon exemple d'hyper-paramètre.

Note : par abus de langage, nous appellerons les paramètres (p, d, q) (P, D, Q)s du modèle SARIMA, des hyper-paramètres, même si ceux-ci peuvent être déterminés par une analyse des fonctions ACF et PACF. Cependant, l'expertise demandée par une telle étude et le nombre de combinaisons possibles pour un modèle SARIMA rendent une analyse classique plus compliquée qu'un modèle ARMA simple. Nous utiliserons donc la même technique d'optimisation que pour un RNN.

La recherche par grille

Afin de déterminer ces hyper-paramètres, il est d'usage d'employer une technique dite de "recherche par grille", qui est une méthode de recherche exhaustive. Il s'agit, dans un premier temps, de dresser la liste des paramètres à optimiser ainsi que leurs différentes valeurs. Ensuite, toutes les combinaisons possibles d'hyperparamètres sont générées et un modèle est créé puis évalué avec chacune de ces combinaisons. Le modèle qui sera retenu optimise un critère de qualité, souvent le MSE pour les tâches de régression (voir 4.2.6).

Les tableaux suivants listent l'ensemble des hyper-paramètres des modèles SARIMA et LSTM ainsi que l'ensemble des valeurs testées pour chacun de ces paramètres.

Paramètre	Valeurs testées
p	Départ : 2, Maximum : 5
d	Déterminé par le test de Kwiatkowski–Phillips–Schmidt–Shin
q	Départ : 2, Maximum : 5
P	Départ : 2, Maximum : 5
D	Déterminé par le test de Osborn-Chui-Smith-Birchenhall
Q	Départ : 2, Maximum : 5
s	Déterminé par la densité de la transformation de Fourier

TABLE 4.1 – Grille de recherche des hyper-paramètres d'un modèle SARIMA

Paramètre	Valeurs testées
Taux d'apprentissage	0.02
Fonction d'activation pour les couches cachées	{ tanh, ReLU, sigmoïde }
Nombre de couches cachées	{ 1, 2, 3 }
Nombre de neurones par couche	{ 10, 50, 200 }
Taux de dropout	{ 0.0, 0.2 }
Taille du vecteur d'entrée	{ 10, 20, 50 }

TABLE 4.2 – Grille de recherche des hyper-paramètres d'un modèle LSTM

Identification d'un modèle SARIMA

Le modèle SARIMA se heurte cependant à un obstacle : le nombre de combinaisons d'hyperparamètres avant l'obtention d'un modèle convenable devient vite élevé avec une méthode de recherche par force brute. Si les paramètres de différenciation varient entre 0 et 2 et les paramètres auto régressifs et moyenne mobiles entre 0 et 5, il conviendrait de calculer 11664 modèles différents, soit plusieurs jours de calcul sur un ordinateur grand public pour une série comportant quelques milliers de points.

Une méthode plus avancée et non exhaustive est alors utilisée : il s'agit d'un algorithme *stepwise* (pas à pas) développé par HYNDMAN, KHANDAKAR et al., 2007.

La méthode consiste à choisir le modèle maximisant un critère d'information¹ donné. Cette méthode nécessite cependant de connaître les valeurs de d , D et s , la période de saisonnalité.

Il convient alors d'utiliser 3 heuristiques pour approximer ces valeurs.

1. La valeur de d est déterminée grâce au test de différenciation de Kwiatkowski – Phillips – Schmidt – Shin².
2. La valeur de différenciation saisonnière D est déterminée d'après le test de Osborn-Chui-Smith-Birchenhall³, abrégé *ocsb* dans le code Python.
3. La période de la saisonnalité, s , est déterminée grâce à une méthode basée sur la période de répétition de la transformée de Fourier de la série stationnarisée. Le détail de cette approche est donné par l'algorithme 3 Identification d'un modèle SARIMA.

1. Le critère d'information d'Akaike sera celui retenu pour la suite des modélisations.
2. Le test de Dickey-Fuller augmenté peut très bien être utilisé également.
3. Le test de Canova-Hansen peut également être utilisé pour déterminer l'ordre de différenciation saisonnier.

Algorithme 3 : Identification de la période de saisonnalité par transformée de Fourier

Données : X = série stationnaire; d = ordre de différenciation utilisé pour rendre X stationnaire; n = nombre d'observations sur X ;

fft = Transformée de Fourier(série stationnaire);

abs_fft = | fft |;

si $\max(\text{abs_fft}) < \overline{\text{abs_fft}} + (4 \times \sigma_{\text{abs_fft}})$ **alors**

| période = 1;

fin**sinon**

| répétition = index(max(abs_fft));

| période = arrondir à l'unité($\frac{n+d}{\text{répétition}}$)**fin**

La méthode de HYNDMAN, KHANDAKAR et al., 2007 est simplifiée et résumée par l'algorithme 4 Identification d'un modèle SARIMA.

Des contraintes supplémentaires peuvent être ajoutées afin d'aider la convergence vers une solution acceptable comme $p + q < 5$.

Optimisation d'un LSTM par Nesterov Adam

Le lecteur attentif aura remarqué que le taux d'apprentissage est fixé à l'avance à 0,02. L'algorithme d'optimisation des poids synaptiques est une version améliorée de la descente de gradient appelée optimisation Nesterov-accelerated Adaptive Moment Estimation et abrégée *Nadam* (DOZAT, 2016). Cette approche combine l'optimisation ADAM RMSProp et les moments de Nesterov.

Nadam est considérée aujourd'hui comme l'une des méthodes d'optimisation les plus robustes et performantes pour l'entraînement de réseaux de neurones, notamment récurrents, comme les LSTM (RUDER, 2016). Cette méthode a l'avantage de fonctionner sans définir un taux d'apprentissage fixé à l'avance puisque celui est calculé automatiquement en étant pondéré par la normalisation L2 des gradients. *Nadam* permet donc de s'affranchir d'un hyperparamètre à optimiser puisque le taux proposé par défaut (0,02) dans le papier d'origine est souvent celui qui est retenu pour l'apprentissage.

Le programme de benchmark devra donc tester pour chaque série, le meilleur modèle parmi une liste de 162 ($3 \times 3 \times 3 \times 2 \times 3$) combinaisons de paramètres possibles pour un modèle LSTM. Il est important de noter que certaines combinaisons de paramètres ne peuvent être utilisées pour le *fit* d'un modèle. Notamment, l'erreur d'un modèle SARIMA ne converge pas pour certaines combinaisons (incompatibilité des paramètres SARIMA avec le critère du maximum de vraisemblance). Ces combinaisons à problème sont, naturellement, éliminées de la recherche.

Algorithme 4 : Sélection automatique d'un modèle SARIMA

Données :

s = période de la saisonnalité, si $s = 1$, il s'agit d'un modèle ARMA, sinon SARIMA;

1. Détermination de quatre modèles de départ :

ARIMA($2, d, 2$) si $s = 1$ et SARIMA($2, d, 2$)($1, D, 1$) si $s > 1$.

ARIMA($0, d, 0$) si $s = 1$ et SARIMA($0, d, 0$)($0, D, 0$) si $s > 1$.

ARIMA($1, d, 0$) si $s = 1$ et SARIMA($1, d, 0$)($1, D, 0$) si $s > 1$.

ARIMA($0, d, 1$) si $s = 1$ et SARIMA($0, d, 1$)($0, D, 1$) si $s > 1$.

Modèle retenu = Modèle ayant le plus faible critère d'information.;

pour chaque *variation du modèle* **< 13** **faire**

Appliquer les transformations suivantes pour obtenir de nouvelles variations du modèle retenu ;

- Variation de ± 1 d'un seul paramètre parmi p, q, P, Q sur le modèle retenu;
- Variation de p et q de ± 1 par rapport au modèle retenu;
- Variation de P et Q de ± 1 par rapport au modèle retenu;

si $AIC_{\text{modèle avec variation}} < AIC_{\text{modèle retenu}}$ **alors**

 Modèle retenu = modèle avec variation ;

fin

fin

4.2.3 Ajout d'un modèle naïf

L'utilisation des deux modèles appliqués à la même série permet d'en faire une comparaison absolue, cependant leurs efficacités dans le cas pratique n'est pas testé. Pour corriger ce problème, nous ajouterons par la suite, un troisième modèle de prédiction : le modèle naïf, également appelé règle de persistance.

Le modèle naïf définit simplement la valeur prévue à l'instant t comme étant égale à la valeur passée à l'instant $t - 1$. La règle de persistance permet donc de juger de l'efficacité et de la pertinence d'un modèle plus complexe. Elle nous permettra également d'évaluer la complexité d'une série : si les trois modèles ont exactement la même performance, la série étudiée n'est sûrement qu'un bruit ou, *a minima*, pas explicable de façon auto-régressive.

D'autres règles simples de prédiction peuvent également être employées :

- Générer des estimations aléatoirement en respectant la distribution des valeurs présentes dans l'ensemble d'entraînement.
- Prévoir systématiquement la valeur moyenne du jeu d'entraînement.

4.2.4 Méthode algorithmique de backtesting des modèles

Les mesures effectuées et présentées par la suite sont issues d'un programme Python écrit pour l'occasion. Le code source complet du programme est disponible en annexe (C.1).

L'algorithme suivant décrit le fonctionnement général du programme :

Algorithme 5 : Méthode générale de backtesting

Génération de données synthétiques;

Chargement des séries issues de données réelles;

pour chaque série s_i **faire**

Stationnarisation de la série s_i ;

pour chaque modèle m_j **faire**

Hyper-paramétrage de m_j sur l'ensemble de train/test;

Fit du modèle m_j dont les paramètres ont été retenus;

Mesure de l'erreur de m_j sur s_i ;

fin

⇒ Faire récapitulatif, graphique... des prévisions de la série s_i

fin

Les modèles sont :

1. Modèle naïf / règle de persistance
2. Modèle auto-régressif et moyenne mobile
3. Réseau de neurones récurrents

Pour écrire le programme, les bibliothèques Python 3.7 suivantes ont été utilisées :

1. **Pandas** : Pour le stockage et la manipulation des séries temporelles, les opérations de décalage (*shift* pour créer le modèle naïf), la désaisonnalisation...
2. **StatsModels** : Pour l'implémentation du modèle SARIMA et le test de Dickey-Fuller augmenté. Cette bibliothèque intègre en fait le modèle SARIMAX mais nous n'utiliserons pas la possibilité d'intégrer des variables exogènes (eXogenous).

3. **Keras** : Pour l'implémentation du modèle LSTM. Keras peut utiliser plusieurs backends : les modèles LSTM proposés dans ce mémoire utilisent le framework TensorFlow, développé par Google.
4. **JobLib** : Sert à faciliter l'implémentation parallèle de la recherche par grille pour le modèle SARIMA. En résulte une réduction du temps de calcul d'environ 40% sur une machine 4 coeurs par rapport à un traitement en série.
5. **Matplotlib** : Utilisé pour le rendu des graphiques, histogrammes et zones de texte.
6. **Numpy** : Bibliothèque de calcul matriciel, sert à stocker les paramètres du réseau LSTM.

4.2.5 Présentation des jeux de données et leurs utilités distinctes

Dans un premier temps, un ensemble de séries générées artificiellement, permettra d'évaluer la performance de chacun des modèles sur des cas théoriques précis, de plus en plus complexes à modéliser. Ainsi, les séries artificielles (et leurs résultats conjecturés) sont :

1. Une série avec tendance linéaire bruitée. Cette série très simple ne devrait poser de problème à aucun des deux modèles et permet de voir si un modèle arrive à éliminer le bruit plus efficacement qu'un autre.
2. Une série avec tendance non-linéaire bruitée. Cette série devrait normalement départager les deux modèles puisque le modèle SARIMA est un modèle linéaire. Cependant, la deuxième différenciation (de saisonnalité) permettrait peut-être de réduire son retard par rapport à un réseau LSTM.
3. Une série avec tendance linéaire bruitée et avec saisonnalité simple. Ce cas est adapté aux deux modèles qui devraient produire de bons résultats.
4. Une série avec tendance non-linéaire bruitée et avec saisonnalité complexe. Il s'agit d'une série comportant des saisonnalités imbriquées. Ce cas plus délicat mettra à rude épreuve le réseau LSTM. La recherche d'une combinaison de paramètres optimaux pour le modèle SARIMA sera également difficile.

Les deux séries suivantes, issues de données réelles, devraient permettre de confronter les résultats précédents à la réalité d'utilisation de ces modèles par l'introduction de nouvelles difficultés (erreur de mesure, nombre d'observations...).

Ces séries sont :

- La production de bières en Australie.
- Les rendements du Bitcoin sur 4 heures.

4.2.6 Mesure de l'erreur

Afin de déterminer quel modèle est le plus performant pour une série donnée, plusieurs indicateurs seront utilisés.

Ces indicateurs doivent pouvoir être employés indépendamment du modèle choisi, il n'est donc pas possible d'utiliser des critères d'information comme celui d'Akaike (AIC) puisqu'ils pénaliseraient énormément les réseaux de neurones selon le principe de parcimonie (nombre important de paramètres).

Le Mean Squared Error

La mesure de performance pour les tâches de régression la plus utilisée est le MSE (*Mean Squared Error*). Cet indicateur calcule la moyenne des erreurs au carré et est donc défini tel que :

$$\text{MSE}(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

C'est ce critère qui sera retenu pour l'entraînement des modèles. Il existe des variantes du MSE permettant de pondérer asymétriquement les erreurs : le critère sanctionnera par exemple les prévisions sous-estimées plus que les prévisions sur-estimées. C'est le cas du MSLE (*Mean Squared Logarithmic Error*), donné par la formule :

$$\text{MSLE}(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^n (\log_e(1 + y_i) - \log_e(1 + \hat{y}_i))^2$$

Ce critère est peu utilisé pour de la prévision de séries temporelles classiques, en raison justement de son asymétrie ⁴.

La médiane des erreurs absolues

La médiane des erreurs absolues (ou MedAE, pour *Median Absolute Error*) est également un indicateur intéressant pour juger la performance d'un modèle.

En effet, le MedAE est insensible aux erreurs aberrantes (*outliers*) grâce aux propriétés de la médiane. Il est défini comme :

$$\text{MedAE}(y, \hat{y}) = \text{médiane}(|y_1 - \hat{y}_1|, \dots, |y_n - \hat{y}_n|)$$

Le pourcentage de variance expliquée

Des indicateurs, comme le pourcentage de variance expliquée ou le coefficient de détermination, se basent sur des critères de variance.

Le pourcentage de variance expliquée permet de se rendre compte rapidement si la série prévue capte la dispersion des valeurs de la série réelle.

$$\text{Pourcentage de variance expliquée} = 1 - \frac{\sigma^2(y - \hat{y})}{\sigma^2(y)}$$

Par définition, le meilleur score possible est 1. En statistique traditionnelle, cet indicateur est peu utilisé et il est couramment délaissé face au coefficient de détermination, R^2 .

Le coefficient de détermination

Le coefficient de détermination, abrégé R^2 , mesure le rapport entre la variance expliquée du modèle (SSE, *Sum Square Explained*) et la variance totale (SST, *Sum Square Total*). Ainsi :

4. Le MSLE a cependant une utilité certaine : celle de tenir compte de la différence **relative** entre la valeur réelle et la valeur prédite. Le MSLE est donc identique par exemple pour un couple valeur réelle - valeur prédite de (100, 60) que (1000, 600).

$$R^2 = \frac{SSE}{SST} = \frac{SSE - SSR}{SST} = \frac{\sum_{i=1}^n (\hat{y}_i - \bar{y})^2}{\sum_{i=1}^n (y_i - \bar{y})^2} = 1 - \frac{\sum_{i=1}^n (y_i - \hat{y}_i)^2}{\sum_{i=1}^n (y_i - \bar{y})^2}$$

Cependant, comme pour le pourcentage de variance expliquée, il souffre d'un grand désavantage : cet indicateur pousse en général à utiliser le maximum de variables explicatives possible afin de faire tendre le score vers 1. Le coefficient de détermination ne teste également pas la colinéarité entre les variables.

Une extension à cet indicateur est le R^2 ajusté qui permet de réduire l'impact de l'amélioration du R^2 par l'augmentation des variables explicatives. Soit p le nombre de variables explicatives dans le modèle, le R^2 ajusté est défini comme :

$$\overline{R^2} = 1 - (1 - R^2) \frac{n - 1}{n - p - 1}$$

4.3 Configuration matérielle utilisée pour le benchmark

Le programme de benchmark a été exécuté sur un serveur de calcul hébergé sur la plateforme *FloydHub*⁵.

La configuration de la machine utilisée est la suivante :

- Intel Xeon ® 8 coeurs
- 32 GB RAM
- 100 GB SSD

Le temps de calcul total est de : 2 jours, 12 heures, 14 minutes et 47 secondes.

5. FloydHub est une plateforme en ligne permettant de louer une machine dédiée au calcul scientifique. La plateforme permet d'envoyer facilement le code Python à exécuter et propose différentes configurations à base de CPU (Intel Xeon) ou GPU (Tesla K80 / Tesla V100).

Chapitre 5

Analyse sur séries synthétiques

Cette partie expose les résultats des modèles SARIMA et LSTM obtenus sur des séries générées de manière artificielle. Pour chaque série, la formule ayant généré la série est donnée. Un bruit blanc est rajouté sur la série brute pour rendre le problème plus réaliste et également éviter les problèmes de convergence pour les modèles SARIMA décomposant parfaitement la série stationarisée (valeur constante pour la tendance linéaire par exemple).

L'ensemble des graphiques et rapports statistiques sont à retrouver en Annexe D Résultats détaillés du benchmark.

5.1 Tendance linéaire

5.1.1 Introduction

Cette série simple de 1000 points est générée par la fonction :

$$f(x) = 0.3x$$

et l'ajout d'un bruit blanc centré et d'écart-type 6.

La représentation graphique de la série et son découpage sont donnés par la figure D.1.

5.1.2 Modèle SARIMA

La saisonnalité détectée par la méthode de la transformée de Fourier est de 1. Il n'y a donc pas de partie saisonnière et le modèle SARIMA se résume alors à un modèle ARIMA.

Le tableau Liste des modèles SARIMA testés sur la série 1 récapitule les modèles testés ainsi que les critères d'informations qui y sont associés.

Le modèle retenu est donc un SARIMA(0, 1, 1) dont la description statistique est résumée par la sortie D.1.

Le modèle a un seul paramètre MA, qui est significatif. Le diagnostic des résidus montre bien que les erreurs restantes ne sont pas statistiquement différentes d'un bruit blanc (D.2).

5.1.3 Modèle LSTM

La série stationnarisée en différence première est utilisée pour alimenter le réseau LSTM (identique à SARIMA puisque $d = 1$). Le test de Dickey-Fuller permet de confirmer qu'une différence première suffit pour traiter cette série (D.3).

Finalement, parmi l'ensemble des 162 modèles LSTM testés, le tableau D.2 liste les 3 plus performants.

Le modèle retenu est un LSTM possédant une couche cachée de dix neurones, équipée d'un dropout et d'une fonction d'activation en tangente hyperbolique et entraîné avec une sous-série de 50 points successifs. L'évolution de la perte (D.4) nous permet de dire que le modèle a convergé sans connaître d'overfitting (les deux courbes restent parallèles).

5.1.4 Prévisions et performances

Pour de la prévision à un pas de temps (one-step-ahead), les deux modèles ont le même comportement et lissent le bruit afin de conserver la partie tendancielle (D.5).

Pour les prévisions de long terme, la structure du modèle SARIMA ne permet pas de prendre en compte la tendance. Les prévisions sont constantes, puisque la partie MA n'évolue plus après le premier pas de temps. Le modèle LSTM, quant à lui, a un comportement très proche de la prévision naïve (D.6).

Les mesures de l'erreur donnent un avantage au réseau LSTM mais le modèle SARIMA n'est pas en reste et demeure significativement plus performant que le modèle naïf (D.2).

5.2 Tendance linéaire avec saisonnalité simple

5.2.1 Introduction

Cette série de 1000 points reprend la tendance et ajoute une fonction sinus pour générer une saisonnalité de période 12 :

$$f(x) = \sin\left(x \times \frac{\pi}{6}\right) \times 4 + 0.3x$$

De la même manière, un bruit blanc centré et d'écart-type 2 est ajouté.

La représentation graphique de la série et son découpage sont donnés par la figure D.7.

5.2.2 Modèle SARIMA

La saisonnalité détectée par la méthode de la transformée de Fourier est de 12. Le modèle SARIMA est donc déjà pré-configuré pour s'adapter à cette période alors que le réseau récurrent doit déterminer lui-même la structure de répétition.

Le tableau Liste des modèles SARIMA testés sur la série 2 récapitule les principaux modèles testés ainsi que les critères d'informations qui y sont associés. Pour cette série, 28 modèles auront dû être testés avant d'arriver à celui retenu.

Finalement, le modèle retenu est donc un SARIMA(1, 1, 1)x(1, 0, 1, 12) dont la description statistique est résumée par la sortie D.3.

Les quatre paramètres du modèle sont significatifs, à l'exception du premier ordre auto régressif pour lequel un doute peut être émis ($P > |z| \approx 0.068$). Le diagnostic des résidus montre bien que les erreurs restantes ne sont pas statistiquement différentes d'un bruit blanc (D.8).

5.2.3 Modèle LSTM

La série stationnarisée en différence première est utilisée pour alimenter le réseau LSTM (identique à SARIMA puisque $d = 1$). Le test de Dickey-Fuller permet de confirmer qu'une différence première suffit pour traiter cette série (D.9).

Finalement, parmi l'ensemble des 162 modèles LSTM testés, le tableau D.3 liste les 3 plus performants.

Le modèle retenu est un LSTM possédant une couche cachée de dix neurones, une fonction d'activation en tangente hyperbolique et entraîné avec une sous-série de 50 points successifs. Il s'agit donc du même modèle que précédemment, le dropout en moins. L'évolution de la perte (D.10) nous permet de dire que le modèle a convergé sans connaître d'overfitting (les deux courbes restent parallèles).

5.2.4 Prévisions et performances

Pour de la prévision à un pas de temps (one-step-ahead), les deux modèles ont le même comportement et lissent le bruit afin de conserver la partie tendancielle et saisonnière (D.11).

La structure du modèle SARIMA permet de prendre en compte la tendance et la saisonnalité correctement sur plusieurs pas de temps. Il semble néanmoins que le modèle SARIMA conserve une composante tendancielle sous-évaluée puisque les prévisions deviennent de plus en plus faibles par rapport aux valeurs attendues. Le modèle LSTM quant à lui parvient à se caler parfaitement sur la série, même à long terme (D.12).

Les mesures de l'erreur donnent donc logiquement un avantage au réseau LSTM pour la prévision dynamique, cependant le modèle SARIMA est très sensiblement meilleur pour de la prévision à un pas de temps (D.4).

5.3 Tendance linéaire avec saisonnalité complexe

5.3.1 Introduction

Cette série de 1000 points reprend la fonction précédente et rajoute un coefficient multiplicatif pour générer une saisonnalité multiplicate de période 12 :

$$f(x) = \sin\left(x \times \frac{\pi}{6}\right) \times 4 \times 0.01x + 0.3x$$

De la même manière, un bruit blanc centré et d'écart-type 1.8 est ajouté.

La représentation graphique de la série et son découpage sont donnés par la figure D.13.

5.3.2 Modèle SARIMA

La saisonnalité détectée par la méthode de la transformée de Fourier est correcte puisque de 12.

Le tableau Liste des modèles SARIMA testés sur la série 3 récapitule les principaux modèles testés ainsi que les critères d'informations qui y sont associés. Pour cette série, 28 modèles auront dû être testés avant d'arriver à celui retenu.

Finalement, le modèle retenu est un SARIMA(1, 1, 1)x(2, 0, 2, 12) dont la description statistique est résumée par la sortie D.5.

Sur les 6 paramètres du modèle, seul le paramètre auto régressif saisonnier de premier ordre n'est pas significatif ($P > |z| = 0.912$).

Le diagnostic des résidus montre bien que les erreurs restantes ne sont pas statistiquement différentes d'un bruit blanc (D.14).

5.3.3 Modèle LSTM

La série stationnarisée en différence première est utilisée pour alimenter le réseau LSTM (identique à SARIMA puisque $d = 1$). Le test de Dickey-Fuller permet de confirmer qu'une différence première suffit pour traiter cette série, même si en apparence la série ne semble pas stationnaire en variance (explicable par l'ajout du paramètre multiplicatif) (D.15).

Finalement, parmi l'ensemble des 162 modèles LSTM testés, le tableau D.6 en liste les 3 plus performants.

Le modèle retenu est un LSTM possédant une couche cachée de cinquante neurones équipée d'un taux d'abandon de 20%, d'une fonction d'activation en tangente hyperbolique et entraîné avec une sous-série de 20 points successifs.

Le modèle choisi par l'algorithme n'est vraisemblablement pas stable puisque l'évolution de la perte (D.16), montre des pics sur la fonction de perte du jeu de test alors que l'erreur sur jeu d'entraînement continue de converger vers un *minimum*. Du fait de la nature de la méthode d'optimisation des poids synaptiques, il est possible qu'une solution plus stable soit trouvée lorsque le réseau est initialisé avec des poids différents de ceux utilisés pour cet entraînement.

5.3.4 Prévisions et performances

Pour de la prévision à un pas de temps (one-step-ahead), les deux modèles ont sensiblement la même performance (D.17).

Pour de la prévision de long terme, il est intéressant de noter que l'amplitude de la saisonnalité du modèle SARIMA est fixe alors que la série demande une saisonnalité multiplicative. Pour cette raison, la performance du modèle SARIMA est inférieure à celle du LSTM. Ce dernier arrive à modéliser une amplitude de saisonnalité variable en fonction du temps (D.18).

Les mesures de l'erreur donnent donc un léger avantage au réseau LSTM pour la prévision dynamique, cependant le modèle SARIMA est très sensiblement meilleur pour de la prévision à un pas de temps (D.6).

5.4 Tendances non polynomiales et saisonnalités imbriquées

5.4.1 Introduction

Il s'agit de la série synthétique la plus complexe à modéliser. 1000 points sont générés à partir d'une fonction basée sur la série précédente. Cette nouvelle fonction comporte deux nouvelles saisonnalités (une de court terme et une autre de long terme) et la tendance

devient non polynomiale ($4x^{0.4}$) :

$$f(x) = \sin\left(x \times \frac{\pi}{3.5}\right) \times 8 - \sin\left(x \times \frac{\pi}{6}\right) \times 15 + \sin\left(x \times \frac{\pi}{40}\right) \times 40 + (4x^{0.4}) \times 30$$

Un bruit blanc centré et d'écart-type 2.8 est ajouté à cette fonction.

La représentation graphique de la série et son découpage est donnée par la figure D.19.

5.4.2 Modèle SARIMA

La saisonnalité dominante détectée par la méthode de la transformée de Fourier est de 12.

Le tableau Liste des modèles SARIMA testés sur la série 4 récapitule les principaux modèles testés ainsi que les critères d'informations qui y sont associés. Pour cette série, seulement 15 modèles auront dû être testés avant d'arriver au modèle retenu.

Finalement, le modèle retenu est un SARIMA(2, 1, 2)x(1, 0, 1, 12) dont la description statistique est résumée par la sortie D.7.

Les 6 paramètres du modèle sont significatifs.

Le diagnostic des résidus montre qu'une partie des saisonnalités n'a pas été éliminée, ce qui est tout à fait logique puisque le modèle SARIMA de base ne permet pas de gérer plusieurs saisonnalités (D.20).

5.4.3 Modèle LSTM

La série stationnarisée en différence première est utilisée pour alimenter le réseau LSTM (identique à SARIMA puisque $d = 1$). Le test de Dickey-Fuller permet de confirmer qu'une différence première suffit pour traiter cette série (D.21).

Finalement, parmi l'ensemble des 162 modèles LSTM testés, le tableau D.8 en liste les 3 plus performants.

Le modèle retenu est un LSTM possédant une couche cachée de 200 neurones, d'une fonction d'activation en tangente hyperbolique et entraîné avec une sous-série de 50 points successifs.

De la même manière que pour la série précédente, mais de façon moins prononcée, il semblerait que la solution trouvée ne soit pas complètement stable (D.22).

5.4.4 Prévisions et performances

Pour de la prévision à un pas de temps (one-step-ahead), les deux modèles ont la même performance (D.23).

En revanche, pour de la prévision de long terme, le modèle SARIMA, sans attente, ne parvient qu'à reproduire la saisonnalité de période 12 et une tendance linéaire approximative. Les contraintes d'équation du modèle SARIMA montrent ici leurs limites.

Le modèle LSTM est exempt de ces limites et approxime très bien la série d'origine à long terme.

Le R^2 du modèle SARIMA est inférieur à 0 pour la prévision dynamique, ce qui signifie que le modèle performe moins bien qu'un modèle qui fournirait des prévisions constamment évaluées à la moyenne de la série. Ici, le modèle SARIMA n'explique donc pas du tout

la distribution des points. Le modèle LSTM quant à lui se démarque et explique 86% de la variance de la série. (D.24).

Le modèle LSTM fournit donc en toute logique les meilleurs résultats (D.8).

Chapitre 6

Analyse sur données réelles

Cette partie fait suite au chapitre précédent (5 Analyse sur séries synthétiques) et enchaîne sur les séries issues de données réelles. Nous étudierons deux cas en particulier : une série relativement peu bruitée et présentant une saisonnalité forte ainsi qu'une série très bruitée.

L'ensemble des graphiques et les rapports statistiques sont à retrouver en Annexe D Résultats détaillés du benchmark.

6.1 La production de bières en Australie

6.1.1 Introduction

La première série issue de données réelles permettra de tester les capacités de modélisation d'une saisonnalité et d'une tendance complexe.

La représentation graphique de la série et son découpage est donnée par la figure D.25.

6.1.2 Modèle SARIMA

Étonnamment, la saisonnalité dominante détectée par la méthode de la transformée de Fourier est de 2. Comme nous allons le voir, ce choix n'affectera pourtant pas la performance du modèle.

Le tableau Liste des modèles SARIMA testés sur la série production de bières récapitule les principaux modèles testés ainsi que les critères d'informations qui y sont associés. Pour cette série, seulement 15 modèles auront dû être testés avant d'arriver au modèle retenu.

Finalement, le modèle retenu est un SARIMA(1, 1, 1)x(4, 0, 3, 2) dont la description statistique est résumée par la sortie D.9.

Sur les 9 paramètres du modèles, 3 ne sont pas significatifs. Cependant, les ordres concernés ne sont pas les ordres du modèle ARMA mais les ordres saisonniers. Il s'agit en effet du deuxième ordre saisonnier de l'autorégression et du premier et deuxième ordre moyenne mobile saisonnier.

Le diagnostic des résidus montre que le modèle a tendance à très légèrement sous-estimer ses prévisions (décalage de la distribution des résidus à droite) (D.26).

6.1.3 Modèle LSTM

La série stationnarisée en différence première est utilisée pour alimenter le réseau LSTM (identique à SARIMA puisque $d = 1$). Le test de Dickey-Fuller permet de confirmer qu'une différence première suffit pour traiter cette série (D.27).

Finalement, parmi l'ensemble des 162 modèles LSTM testés, le tableau D.10 en liste les 3 plus performants.

Pour la première fois, le modèle retenu est équipée de deux couches cachées, ne comportant chacune que 10 neurones avec une fonction d'activation en tangente hyperbolique. Le modèle prend 20 valeurs successives de la série avant d'avoir sa mémoire réinitialisée.

L'évolution de la perte nous renseigne quant à la bonne convergence et stabilité de cette configuration de réseau de neurones (D.28).

6.1.4 Prévisions et performances

Pour de la prévision à un pas de temps (one-step-ahead), le modèle SARIMA se montre légèrement supérieur au LSTM choisi (D.29). Il est intéressant de noter que la médiane des erreurs absolues est tout de même plus élevée pour le modèle SARIMA. Le réseau de neurones fait donc, plus régulièrement, de meilleures prévisions.

Pour de la prévision dynamique, le modèle SARIMA fait aussi meilleure figure. Les deux modèles arrivent à conserver la saisonnalité à long terme, sans introduire de décalage. (D.30, D.10).

6.2 Les rendements du Bitcoin

6.2.1 Introduction

La modélisation d'un actif financier est l'un des domaines les plus complexes de l'analyse des séries temporelles. Ces séries sont très bruitées et non stationnaires par essence (hétéroscédastique). Le modèle SARIMA, dans sa version standard, ne permet pas de gérer ces séries hétéroscédastiques. L'invention des modèles GARCH, ENGLE et GRANGER, 1987, a rendu possible l'étude de telles séries financières¹.

Un modèle GARCH semble ici plus adapté. Le peu d'information auto corrélée est donc très difficilement capté et modélisé.

La série ne concerne pas le prix du Bitcoin directement mais ses rendements. Un rendement à l'instant t dépend du prix à l'instant t et $t - 1$:

$$\rho = \frac{p_t}{p_{t-1}}$$

Cette transformation permet d'éliminer les effets de tendance de long terme, très prononcés pour le Bitcoin et stabilise également la variance de la série stationnarisée. Une transformation de type Box-Cox, comme la transformation logarithmique, aurait également permis de corriger l'évolution de la variance liée au niveau de prix.

Nous nous attendons à ce que les trois modèles aient une performance similaire.

La représentation graphique de la série et son découpage est donnée par la figure D.31.

6.2.2 Modèle SARIMA

La saisonnalité détectée par la méthode de la transformée de Fourier est de 3.

1. Ce qui aura valu à leurs auteurs, C. W. J Granger et R. F. Engle, le prix de la Banque de Suède en sciences économiques en mémoire d'Alfred Nobel, en 2003.

Le tableau Liste des modèles SARIMA testés sur la série rendement du Bitcoin récapitule les principaux modèles testés ainsi que les critères d'informations qui y sont associés. Pour cette série, seulement 25 modèles auront dû être testés avant d'arriver au modèle retenu.

Finalement, le modèle retenu est un SARIMA(1, 1, 2)x(2, 0, 0, 3) dont la description statistique est résumée par la sortie D.11.

Tous les paramètres sont significatifs à l'exception du premier ordre autorégressif saisonnier.

Le diagnostic des résidus montre l'absence totale d'autocorrélation (D.32), cependant les résidus ne suivent pas une loi normale. Cela peut s'expliquer par les grands écarts de volatilité entre deux pas de temps, non anticipés par le modèle. La distribution des valeurs extrêmes est plus étalée que la loi normale, ce qui est un cas courant pour les séries financières.

6.2.3 Modèle LSTM

La série stationnarisée en différence première est utilisée pour alimenter le réseau LSTM (identique à SARIMA puisque $d = 1$). Le test de Dickey-Fuller permet de confirmer qu'une différence première suffit pour traiter cette série (D.33).

Finalement, parmi l'ensemble des 162 modèles LSTM testés, le tableau D.12 liste les 3 modèles les plus performants.

Le modèle retenu est un LSTM possédant une couche cachée de 50 neurones, d'une fonction d'activation sigmoïde et entraîné avec une sous-série de 20 points successifs.

L'évolution de la perte converge lentement sur les 60 premières itérations puis continue de diminuer sur le jeu d'entraînement alors que la perte du jeu de test commence à remonter (D.34). C'est l'indication d'un début de sur-apprentissage. Peut-être qu'une configuration basée sur un dropout aurait permis d'éliminer ce phénomène.

6.2.4 Prévisions et performances

Sans surprise, aucun modèle ne parvient à expliquer la variance de cette série de façon significative, que cela soit en prévision à un pas de temps ou en prévisions de long terme (D.35, D.36, D.12).

Le R^2 moyen des modèles présentés est négatif : le modèle le plus performant pour anticiper les rendements du Bitcoin restent donc d'anticiper son rendement moyen (0,0021% pour un $R^2 = 0$).

Conclusion

La conclusion est articulée en deux points. Le premier traite des autres applications possibles des réseaux de neurones récurrents. Le second point débat de la question de la performance et de l'interprétabilité des résultats, avant de résumer l'essence du mémoire.

Applications des RNN hors série temporelle

Cette section traite des avantages que fournit la grande flexibilité d'utilisation des réseaux de neurones récurrents. De par leurs constructions, ces réseaux peuvent prendre une entrée à très grande dimension et fournir non pas une, mais plusieurs sorties, où chacune d'elles peut modéliser une valeur dans le cas d'une régression ou une probabilité pour de la classification.

Cette grande souplesse permet aujourd'hui aux RNN d'être utilisés dans des domaines très variés : de la reconnaissance d'écriture, à la prévision de ventes en passant même par de la description d'images (*Image captioning*).

Génération de musique

Les notes d'une partition peuvent être vues comme une suite de valeurs évoluant dans le temps, à pas de temps (rythme) constant. Dès lors, les réseaux de neurones récurrents sont aptes à générer une musique inspirée d'un ensemble cohérent de partitions par exemple.

Des contraintes supplémentaires peuvent être imposées au modèle, comme la gamme dans laquelle le morceau créé doit être joué, afin d'éviter les notes qui ne sonneraient pas biens. L'ajout de contraintes, comme celle-ci, permet également de réduire la complexité du modèle en ne lui proposant qu'un petit sous-ensemble de sorties possibles. Le modèle peut ainsi être entraîné avec un taux de dropout élevé et ainsi éviter le sur-apprentissage, très courant pour ces cas d'usage. La puissance des réseaux de neurones permet la reproduction de sons polyphoniques très complexes et justifie l'aisance avec laquelle ces modèles permettent de traiter des séries temporelles plus classiques.

Les LSTM se prêtent ainsi bien à ce cas d'utilisation (CHUNG et al., 2014).

Image captioning

Microsoft a créé le site captionbot.ai qui utilise, en partie, un réseau de neurones LSTM afin de générer des légendes automatiquement à partir d'une photo.

Les résultats sont impressionnants, comme le montre la figure 6.1 Exemple d'image captioning. Il s'agit d'une photo de la façade de la faculté d'économie de Montpellier, décrite par l'algorithme comme "un groupe de personne marchant en face d'un bâtiment".

Sur le plan marketing, cette utilisation peut avoir un intérêt pour l'analyse de contenu postée sur les réseaux sociaux. Combiné à l'analyse de sentiment, les réseaux LSTM constituent ainsi un puissant moyen d'automatiser la surveillance de l'image d'une marque.

FIGURE 6.1 – Exemple d'image captioning



Modélisation du langage

Le domaine du traitement naturel du langage naturel, plus couramment abrégé *NLP*², vise à créer des outils de traitement afin de faciliter la compréhension de la langue (comprendre par là, du texte, une voix...) par les machines. Ce domaine est particulièrement adapté à l'utilisation de réseaux de neurones récurrents.

En effet, un enregistrement audio est une série temporelle : l'évolution des fréquences sonores peut alors être étudiée pour reconnaître des mots, des phrases. Il s'agit alors de faire de la reconnaissance automatique de la parole. Ce processus peut être inversé pour permettre de générer une suite de fréquences sonores : c'est la synthèse vocale.

De la même manière, un texte n'est qu'une suite de mots, et les mots une succession de lettres. Entraîner un réseau de neurones récurrents sur un corpus permet alors de générer du texte dont la dépendance temporelle reste similaire au jeu d'entraînement. Cette tâche fait l'objet d'un développement approfondi dans la section suivante.

La poésie des LSTM

Pour justifier de l'utilité des réseaux de neurones récurrents pour du *NLP*, montrons qu'il est possible de générer des phrases cohérentes à l'aide d'une architecture LSTM.

Le jeu d'entraînement est constitué de l'ensemble des *Fables* de La Fontaine. Plus précisément, le jeu de données est issu de la concaténation de toutes ses Fables, du livre I à XII, dans un fichier texte. Son oeuvre est intéressante car elle présente une structure particulière et complexe. Un modèle correctement entraîné devra prédire des mots couramment employés par Jean de La Fontaine (du registre animalier par exemple), faire des phrases de longueur quasi-identique, des titres généralement courts et des rimes en fin de vers. De plus, les Fables sont passées dans le domaine public, ce qui permet de diffuser librement le jeu de données constitué (voir Annexe C.2).

2. NLP : Natural Language Processing

A titre d'exemple, voici les 4 premières lignes du fichier :

```
1 === La Cigale et la Fourmi ===
2
3 La cigale ayant chanté
4 Tout l'été,
```

Pour faire de la génération de texte, il est possible d'agir à plusieurs niveaux :

- Sur les lettres : le réseau prend alors en entrée une suite de lettres dans le but de prédire la lettre suivante.
- Sur les mots : le réseau prend une suite de mots et prédit le mot suivant le plus probable.
- Sur les N-grammes : cette vision probabiliste permet d'obtenir la probabilité conditionnelle d'apparition de la lettre/mot suivant à partir du nombre d'occurrences de cet élément dans le corpus.
- ...

Le modèle créé opérera au niveau des lettres. Ainsi, pour une série de lettres donnée, le modèle devra donner la prochaine lettre qui a la probabilité d'apparition la plus élevée.

Note : Le terme "lettre" est ici utilisé abusivement et fait plutôt référence au terme de "caractère" (*character* en anglais) employé en informatique et typographie. Ce type de modèle est d'ailleurs appelé CharNN.

Comme un réseau de neurones ne peut accepter directement que des valeurs numériques, il convient alors d'encoder les lettres en nombres. En utilisant un encodage naïf de type $A \rightarrow 1, B \rightarrow 2, \dots$ un problème de proximité survient. Ici, les lettres A et B ont une valeur similaire mais ont une signification très différente dans la langue française (une voyelle très employée et une consonne).

Pour contourner ce problème, l'encodage généralement retenu s'appelle encodage *one-hot*. Cet encodage crée pour chaque lettre, un vecteur de valeurs nulles de dimension fixe égale aux nombres de caractères différents dans le corpus et place un 1 à une position unique dans le vecteur. Pour un corpus comportant uniquement les lettres A, B et C, nous obtenons donc l'encodage *one-hot* suivant :

$$A = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}, B = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}, C = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

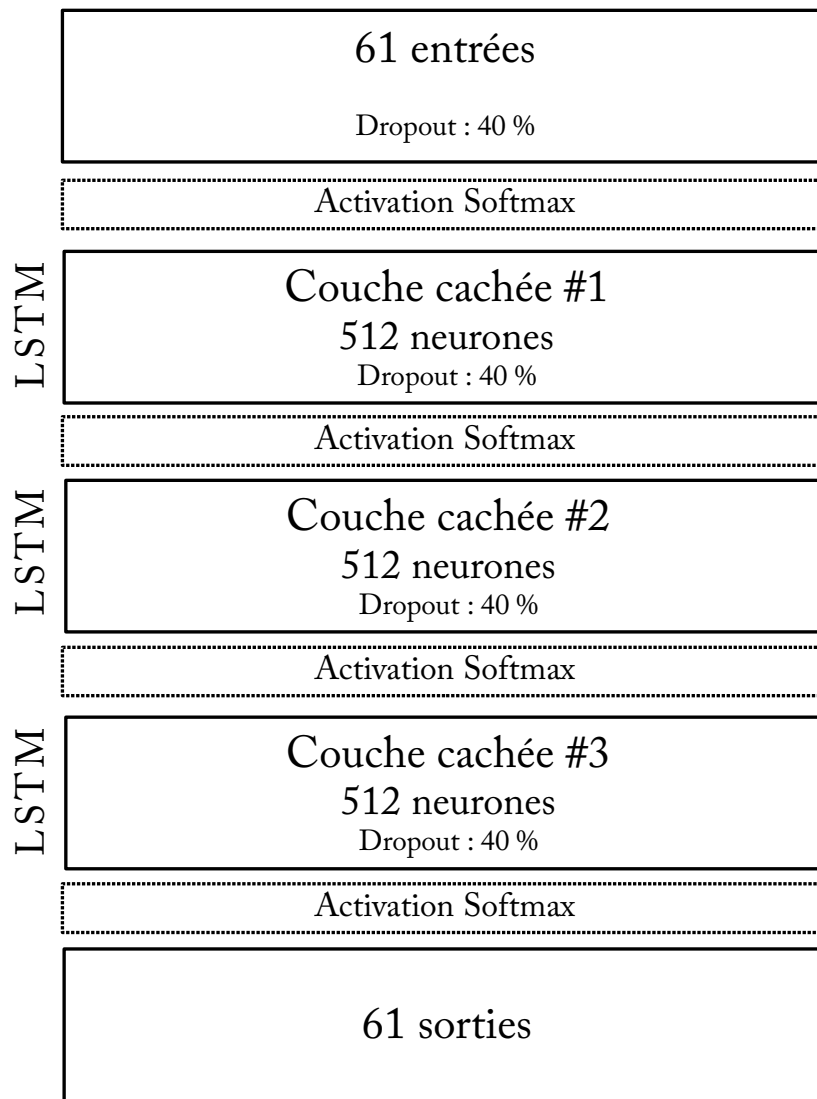
Le réseau de neurones aura donc une couche d'entrée composée d'autant d'entrées que de caractères différents.

Pour les Fables de La Fontaine, le corpus est composé (après suppression des caractères accentués) de 61 caractères différents et comporte un total de 427566 caractères.

L'architecture choisie pour cet exercice est donnée par la figure 6.2. Il s'agit d'une architecture à 3 couches cachées récurrentes équipées d'une fonction d'activation Softmax, adaptée à la modélisation de probabilité, comme vu dans le tableau 2.1 Fonctions d'activation usuelles.

La fonction Softmax est particulièrement intéressante dans ce cas-ci puisqu'un paramètre appelé "température" permet de varier les résultats de sortie de la fonction. Une température élevée aplatira la distribution de probabilités d'apparition des lettres, le réseau sera alors hésitant entre plusieurs lettres et permet de le rendre "créatif" dans le sens où la suite de lettres que nous pouvons obtenir ne vient pas directement du corpus appris (évite

FIGURE 6.2 – Architecture du générateur de Fables, 3 couches cachées



l'overfitting). *A contrario*, une température faible, rendra le réseau plus confiant dans sa prédiction et proposera le caractère qui a le plus chance d'apparaître dans le corpus (pour peu que le réseau soit suffisamment entraîné).

Les valeurs de température testées pendant la phase d'entraînement sont 0.3, 0.6, 1, 1.5 et 2.

La phase d'entraînement se fait sur 100 itérations et nous demandons au modèle de générer une sortie comportant 240 caractères toutes les 5 itérations afin de vérifier l'évolution de l'apprentissage du modèle.

À l'itération 0, le modèle est initialisé avec des poids aléatoires et génère donc une sortie composée de caractères aléatoires :

```

1 ----- Iteration : 0 -----
2
3 Temperature : 0.3
4
5 Fable générée :
6 :juCJJl!Ctuczes?»A
7 !E?PtIoTczPzfXXPU-AvOq-fhrhbg-CHZbti-tJzytgQYa-umDnsgaUDqNloi
8 OADTSV :u:Qgdv.SFVu
9 Pn D!t?tSaM?YBULCYQ,H,yySP
10 -Zperq Nr
11 eXbPirFl»XOnqfnAiaGDa:xgxVccGjrlOfvM =?»g«:SFgZ;Ln1«
12 ?DcObmLDdANNp-eXG.beq:nTt
13 LNd.qndvUXMBxXT'cx
14 aQrta

```

À l'itération 5, le modèle commence à utiliser plus fréquemment l'espace pour séparer les mots, nous pouvons également noter l'utilisation de ponctuations en fin de ligne :

```

1 ----- Iteration : 5 -----
2
3 Temperature : 1.5
4
5 Fable générée :
6 lansque fcis de soigus,
7 «ereut, sarsra postibais maignan,
8 Hieux en je beduit que le manqtefiee,
9 emprope eit et du poisie
10
11
12 Ce dans a des anistabns, et deux Matte.
13 for cruquon, il saurie qu'il un jesass ?

```

À l'itération 10, le réseau de neurones a suffisamment précisé sa mémoire de court terme (quelques caractères) pour former des mots correctement orthographiés :

```

1 ----- Iteration : 10 -----
2
3 Temperature : 1
4

```

```

5 Fable générée :
6 Mais fuissant de fait a la mouton.
7 Ces nous, nous ne payez sous les trois pats.
8
9 Phetre plus courir, les diux ; pour l'hirondelle.
10 Ce qu'il l'etait intermice :
11 Nul fit me celebre et de plus bras.
12 - Monsieur que le voisinage
13
14 Est fort le moe

```

À l'itération 25, le modèle comprend parfaitement la structure générale d'une fable et prédit correctement la séquence composée de ===, un titre puis ===. Il "invente" même un titre, puisque "Le Lion et le Chat" n'est pas une fable écrite par Jean de La Fontaine. Le registre animalier est également présent.

```

1 ----- Iteration : 25 -----
2
3 Temperature : 0.3
4
5 Fable générée :
6 sans cesse en un coin du maitre ;
7 Et cependant j'ai passe les moins de mainte autre ;
8 Et moi, je ne sais qui de ce point.
9
10
11
12 === Le Lion et le Chat ===
13
14 Le Singe avec le Loup en peu de loin ;
15 Car il eut beau la chose : et l'autre a passer :

```

Le modèle converge rapidement et n'évolue plus vraiment après l'itération 50, comme en atteste la figure 6.3.

Une fois l'entraînement terminé (à la 100ème itération), le modèle est stocké de manière persistante en enregistrant ses poids synaptiques dans un fichier h5. Il est alors possible de le réutiliser. Certains programmes, comme *Netron* permettent de charger de fichier dans le but d'en faire une représentation graphique.

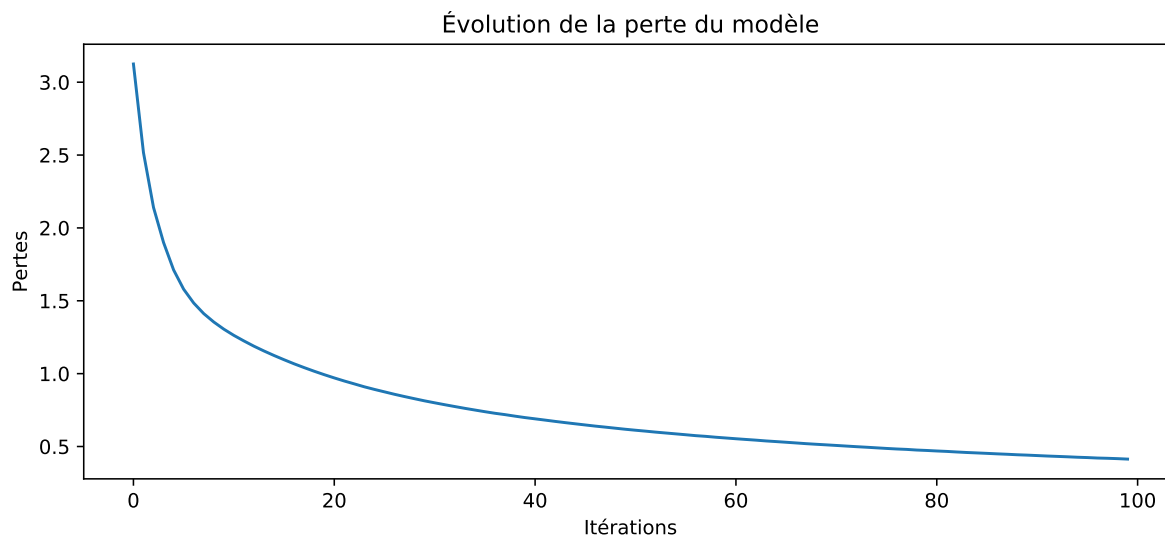
Pour pouvoir générer une "Fable" depuis le début, nous initialisons sa mémoire avec les caractères qui précèdent "===" (début d'un titre), c'est-à-dire 3 retours à la lignes ("\\n"), puis nous le laissons prédire lettre après lettre, le texte. L'arrêt se fait lorsque 3 retours à la ligne sont rencontrés, voir C.2 Programme CharNN pour la génération de Fables.

Interprétabilité des résultats selon le modèle

L'exemple précédent constitue une bonne transition vers la question de l'interprétabilité des modèles basés sur les réseaux de neurones. En effet, le modèle présenté au dessus compte plus de 5 millions de paramètres³. Il est impossible de donner une signification à

3. 5 405 245 de paramètres répartis entre les 3 couches LSTM et la couche de sortie.

FIGURE 6.3 – Évolution de la perte du modèle CharNN sur les Fables de La Fontaine



un paramètre de ce modèle. L'enchevêtrement des connexions récurrentes et entre couches rend l'ingénierie inverse inenvisageable.

Qui plus est, certains domaines d'application nécessitent une parfaite compréhension du modèle employé pour la prévision. La complexité des calculs ne doit pas nuire à l'interprétation des résultats.

C'est le cas du domaine bancaire, où les techniques de scoring doivent pouvoir être expliquées. Un client qui se voit refuser une demande de prêt par un algorithme d'apprentissage automatique (comme un réseau de neurones) serait dans son droit de demander une justification du résultat, ce qui serait alors impossible à produire. Ces modèles en boîte noire, bien que très performants, manquent alors d'une qualité importante : la transparence.

Le Règlement Général sur la Protection des Données (RGPD) impose ce principe par l'article 22 "Décision individuelle automatisée, y compris le profilage" (Chapitre 3)⁴.

Sur un plan plus technique, des méthodes permettent de palier ce manque de transparence mais de manière sommaire et incomplète. Ainsi, OLDEN, 2004, propose une méthode permettant de quantifier l'importance des variables dans un perceptron multi-couches. Le principe est simple. Les valeurs des variables explicatives sans intérêt pour le modèle ne sont que très peu transmises aux couches suivantes : cela se retrouve dans des poids synaptiques proches de zéro.

4. "Article 22 : La personne concernée a le droit de ne pas faire l'objet d'une décision fondée exclusivement sur un traitement automatisé, y compris le profilage, produisant des effets juridiques la concernant ou l'affectant de manière significative de façon similaire."

Résumé théorique et empirique

La partie théorique a permis de mettre en exergue les ressemblances entre les modèles ARMA et LSTM.

Les modèles Box-Jenkins proposent de modéliser la série essentiellement en deux parties. La première, dite modélisation autorégressive, permet d'influencer la valeur future de la série connaissant ses valeurs passées. Ce concept est directement transposable aux réseaux de neurones dits récurrents puisque ce sont ces connexions récurrentes qui jouent le rôle d'AR.

L'état de la mémoire dans un réseau LSTM est cependant modélisé par un système de portes, permettant de conserver dans les cellules à mémoire, une "empreinte" des valeurs déjà rencontrées par le réseau. Cette empreinte reste valide à long terme alors que les modèles ARMA deviennent rapidement limités au bout de quelques pas de temps.

Ce défaut du modèle ARMA est compensé par la rigidité des équations qui le composent : le modèle ARMA est alors moins sensible au bruit que les réseaux de neurones traditionnels.

La partie moyenne mobile d'un modèle ARMA, quant à elle, n'a pas de contrepartie directe sur un LSTM puisque ce dernier est "aveugle" des erreurs qu'il commet, par construction.

Les réseaux de neurones récurrents sont alors en partie une généralisation non-linéaire des modèles SARIMA.

Concernant la performance des modèles de réseaux de neurones, celle-ci n'est pas certaine pour toutes les utilisations. Pour de la prévision de série temporelle, l'écart de performance entre un modèle SARIMA et un réseau LSTM ne justifie pas la complexité introduite. Cet écart est encore moins justifiable pour de la prévision à un pas de temps. Ces résultats concordent donc avec les travaux de SPYROS MAKRIDAKIS, 2018. Il est contrariant qu'un modèle bien plus sophistiqué et bien plus demandeur en terme de calculs ne puisse pas plus s'imposer dans la course à la performance. D'autant plus que les problèmes engendrés par les réseaux de neurones (choix d'une configuration, problème de sur-apprentissage) n'ont pas leurs équivalents avec les modèles de Box & Jenkins. En effet, cette dernière méthode pénalise, par la nature du critère d'information, la complexité des modèles et tend mécaniquement à l'élimination des modèles sujets aux phénomènes d'*overfitting*.

Cependant, la grande flexibilité de ces réseaux de neurones permet une utilisation plus étendue que les modèles ARMA, comme nous l'avons vu pour de la modélisation du langage, domaine qui était, jusqu'alors, inexploitable par l'approche de Box & Jenkins.

Annexe A

Informations complémentaires

A.1 Récapitulatif des notations employées

Le lecteur non familier avec les notations d'usage en statistiques peut se référer à la table des notations suivantes.

Symbole	Signification
X	Variable explicative (et expliquée si modèle auto régressif)
x_t avec $t = 1, \dots, n$	Série temporelle
y	Valeur réelle à estimer
$\hat{y}$	Valeur estimée par le modèle
$\bar{y}$	Moyenne de y
$\sigma(y)$	Écart-type de y
$\sigma^2(y)$	Variance de y
φ	Paramètre du modèle AR
θ	Paramètre du modèle MA
H_i	i-ème neurone d'une couche cachée
$w_{j,k}^i$	Poids synaptique de la i-ème couche, reliant le j-ème neurone au k-ème neurone de la couche suivante.
A^T	Transposé de la matrice A.
∇f	Gradient de la fonction f
$\odot$	Produit matriciel de Hadamard

TABLE A.1 – Table des notations

Annexe B

Jeux de données utilisés

B.1 Production de bières en Australie

Cette série est souvent utilisée dans la recherche comme référence. Il s'agit de la production mensuelle de bières en Australie de 1956 à 1996. Les unités sont en mégalitres. Les données n'incluent pas les boissons avec un taux d'alcool inférieur à 1,15%.

Le fichier complet étant trop volumineux pour être inséré sur support papier, il est placé à disposition sur le répertoire Github du projet, répertoire "datasets" :

<https://github.com/hugues31/sarimavslstm>

L'extrait suivant permet de se faire une idée de la série :

CODE B.1 – 30 premières lignes du fichier production_biere_australie.csv

Série
93.2
96
95.2
77.1
70.9
64.8
70.1
77.3
79.5
100.6
100.7
107.1
95.9
82.8
83.3
80
80.4
67.5
75.7
71.1
89.3
101.1
105.2
114.1
96.3
84.4
91.2
81.9
80.5

B.2 Rendements du Bitcoin sur quatre heures

Cette série a été obtenue à partir d'un jeu de données contenant toutes les transactions effectuées sur la plateforme d'échange Kraken. Le fichier a été resamplé pour obtenir une granularité sur quatre heures en utilisant comme fonction d'agrégation la moyenne.

Le fichier complet étant trop volumineux pour être inséré sur support papier, il est placé à disposition sur le répertoire Github du projet, répertoire "datasets" :

<https://github.com/hugues31/sarimavslstm>

L'extrait suivant permet de se faire une idée de la série :

CODE B.2 – 30 premières lignes du fichier rendements_bitcoin_4H.csv

Série
−0.02355604293916269
−0.011688176039057807
−0.004295593203887504
0.011422294271822508
0.005574465349909463
0.003824155167288401
−0.028540147393603554
0.00119869616818713
0.01964984215557064
−0.006417411229441838
−0.0037686031529183817
0.0069938976146688425
−0.0005305916427811308
0.006683031768410341
0.005778433778548253
0.01722188402017899
−0.005272675452530962
0.022493249232080625
0.00280496990656931
0.009858979749225893
−0.0031961593567902957
0.006705772158302814
−0.021081786215127463
−0.015283227543934963
−0.010087313007197007
0.011836678337207696
−0.0006666988501383697
−0.013757807961992441
−0.004891758225190168

Annexe C

Code source

C.1 Programme de benchmark SARIMA - LSTM

Le programme de benchmark étant beaucoup trop long pour être placé en annexe, seuls les passages les plus importants ont été conservés. Le code source complet, les logs des phases de calcul et les jeux de données sont disponibles sur le répertoire Github du projet.

Répertoire GitHub : <https://github.com/hugues31/sarimavslstm>.

CODE C.1 – Fichier principal

```

1 import sys
2 from serie import Serie
3
4 def charger_series():
5     # Tendance linéaire simple
6     serie_1 = Serie("Série_1", formule="0.3*x", std=6)
7
8     # Tendance linéaire + saisonnalité simple de période 12
9     serie_2 = Serie("Série_2", formule="sin(x*pi/6)*4+0.3*x", std=2)
10
11     # Tendance linéaire + saisonnalité de période 12 multiplicative
12     serie_3 = Serie("Série_3", formule="sin(x*pi/6)*4*(0.01*x)+0.3*x",
13                     std=1.8)
14
15     # Tendance non polynomiale + saisonnalités imbriquées
16     serie_4 = Serie("Série_4",
17                     formule="(sin(x*pi/3.5)*8 - sin(x*pi/6)*15)
18     +sin(x*pi/40)*40+((4*x)**0.4)*30", std=2.8)
19
20     # Série production de bieres en Australie de 1956 a 1995
21     serie_bieres = Serie("Production_de_bieres",
22                          csv_file="production_biere_australie.csv")
23
24     # Série financiere tres bruitée : le rendement du Bitcoin
25     # sur 4 heures
26     serie_bitcoin = Serie("Rendement_du_Bitcoin",
27                           csv_file="rendements_bitcoin_4H.csv")
28
29     return [serie_1, serie_2, serie_3, serie_4,
30            serie_bieres, serie_bitcoin]
31
32 series = charger_series() # synthétiques + données réelles
33
34 for serie in series:
35     for modele in serie.modeles:

```

```

36         modele.definir_serie(serie)
37         modele.trouver_hyperparametres()
38         modele.fit()
39
40     serie.recapitulatif()

```

CODE C.2 – Modèle naïf

```

1  from modele import Modele
2
3  class Naive(Modele):
4      """
5      Naive est le modele de prévision naïf. La valeur prévue a
6      l'instant t est égale a la valeur a l'instant t-1.
7      Sert de point de comparaison aux autres modeles.
8      """
9
10     def trouver_hyperparametres(self):
11         # aucun hyperparametre pour un modele naïf
12         pass
13
14     def fit(self):
15         # Prévision one step ahead
16         self.serie.data[self.__class__.__name__] =
17             self.serie.data['Série']
18             [self.serie.index_fin_entrainement - 1:].shift(periods=1)
19
20         # Prévision dynamique (t = derniere observation connue)
21         self.serie.data[self.__class__.__name__ + "_dynamique"] =
22             self.serie.data['Série']
23             .iloc[self.serie.index_fin_entrainement]

```

CODE C.3 – Modèle SARIMA

```

1  from math import *
2  from modele import Modele
3  from statsmodels.tsa.statespace.sarimax import SARIMAX
4  import numpy as np
5  import pmdarima as pm
6  from sklearn.metrics import mean_squared_error
7  import math
8  import matplotlib.pyplot as plt
9  from pandas import DataFrame
10 from fonctions import enregistrer_plot
11
12 class SARIMA(Modele):
13     """
14     Cette classe implémente le modele ARMA avec différenciation
15     intégrée et gestion d'une saisonnalité simple (SARIMA).
16     """
17
18     def trouver_hyperparametres(self):
19         # détermination de la fréquence de la saisonnalité
20
21         # difference premiere
22         t = np.diff(self.serie
23                     .data['Série'][0:self.serie.index_fin_entrainement], n=1)

```

```

24
25     w = np.fft.fft(t)    # fft
26
27     # on cherche les périodes de saisonnalité < 30
28     # en évitant la première période
29     tableau = np.absolute(w)[1:-(int(0.1*len(t)))]
30
31     if max(tableau) < np.mean(tableau) + 4 * np.std(tableau):
32         frequence = 1
33
34     else:
35         # on rajoute la première période
36         # (+1 car différenciée une fois)
37         repetition = max(range(len(tableau)),
38                           key=tableau.__getitem__) + 1
39         frequence = round((len(t)+1)/repetition)
40
41     print("Période_détectée_de_la_saisonnalité:_:" + str(frequence))
42
43     # détermination des ordres p, d, q, P, D, Q sur
44     # l'ensemble de train
45     stepwise_fit =
46     pm.auto_arima(self.serie.data['Série'],
47                  [0:self.serie.index_fin_test],
48                  start_p=2, d=None, start_q=2, max_p=5, max_d=4, max_q=5,
49                  start_P=2, D=None, start_Q=1, max_P=5, max_D=4, max_Q=5,
50                  m=frequence, seasonal=True, seasonal_test='ocsb',
51                  trace=True, error_action='ignore',
52                  suppress_warnings=True, stepwise=True,
53                  out_of_sample_size=self.serie.index_fin_test-self.
54                      serie.index_fin_entrainement,
55                  scoring='mse', trend=None, with_intercept=False)
56
57     print(stepwise_fit.summary())
58     print(stepwise_fit.get_params())
59
60     fig = stepwise_fit.plot_diagnostics(figsize=(10, 10), lags=12)
61     fig.axes[0].set_title("Résidus_normalisés")
62     fig.axes[1].set_title("Comparaison_de_la_densité_estimée
63 .....avec_N(0,1)")
64     fig.axes[2].set_title("Diagramme_Quantile-Quantile")
65     fig.axes[2].set_xlabel("Quantiles_théoriques")
66     fig.axes[2].set_ylabel("Quantiles_échantillonnés")
67     fig.axes[3].set_title("Corrélogramme")
68     enregistrer_plot(plt, self.nom_sauvegarde + '_diagnostic.pdf')
69     plt.show()
70
71     params = stepwise_fit.get_params()
72     ordres = params['order']
73     ordres_saisonniers = params['seasonal_order']
74
75     config = (ordres, ordres_saisonniers, None)
76     self.config = config
77
78     def fit_modele(self, config):
79         """
80         Fait un fit d'un modele SARIMA selon les parametres

```

```

81         contenus dans config (p,d,q,P,D,Q,s,t)
82     """
83     key = str(config)
84
85     print("Fit_du_modele_" + key + "_en_cours...")
86
87     # Prédiction sur jeu de test + validation
88     # walk-forward validation
89     longueur_test = len(self.serie.data['Test']).dropna()
90     longueur_test += len(self.serie.data['Validation']).dropna()
91
92     # Prévisions one step ahead
93     serie_predite = []
94     for i in range(longueur_test):
95         print("Prévision_one-step-ahead:_" + str(i) +
96             "_sur_" + str(longueur_test))
97         modele = SARIMAX(self.serie.data['Série']
98             [0:self.serie.index_fin_entrainement+i],
99             order=config[0],
100             seasonal_order=config[1],
101             trend=config[2],
102             simple_differencing=False)
103         modele_fit = modele.fit(dispatch=False)
104         yhat = modele_fit.forecast(steps=1).values[0]
105
106         serie_predite.append(yhat)
107
108     # Prévisions dynamiques
109     modele = SARIMAX(self.serie.data['Série']
110         [0:self.serie.index_fin_entrainement],
111         order=config[0],
112         seasonal_order=config[1],
113         trend=config[2],
114         simple_differencing=False)
115
116     modele_fit = modele.fit(dispatch=False)
117
118     if modele_fit.mle_retvals['converged'] == False:
119         print("Convergence_non_atteinte_pour_" + str(config))
120
121     serie_predite_dynamique = modele_fit.predict(
122         start=self.serie.index_fin_entrainement,
123         end=len(self.serie.data['Série'])-1,
124         dynamic=True)

```

CODE C.4 – Modèle LSTM

```

1 import matplotlib.pyplot as plt
2 from sklearn.preprocessing import MinMaxScaler
3 from keras.models import Sequential
4 from keras.layers import LSTM as kLSTM
5 from keras.layers import Dense
6 from keras.layers import Dropout
7 from keras import optimizers
8 from sklearn.metrics import mean_squared_error
9 from keras.callbacks import EarlyStopping
10 from keras.utils import plot_model

```

```

11
12 import numpy as np
13
14 from modele import Modele
15 from fonctions import decouper_serie_apprentissage_supervise ,
16                       enregistrer_plot
17
18 class LSTM(Modele):
19     """
20     Implémentation d'un réseau LSTM via Keras
21     entraînable sur CPU ou GPU
22     """
23
24     def trouver_hyperparametres(self):
25         config_list = list()
26         for h in range(1, 4):
27             for i in [10, 20, 50]:
28                 for n in [10, 50, 200]:
29                     for f in ['tanh', 'relu', 'sigmoid']:
30                         for d in [0.0, 0.2]:
31                             config = {
32                                 'nbre_couches': h,
33                                 'taille_entree': i,
34                                 'nbre_neurones': n,
35                                 'activation': f,
36                                 'dropout': d
37                             }
38                             config_list.append(config)
39
40         print("Taille_des_combinaisons_LSTM:_:" + str(len(config_list)))
41
42         scores = [self.fit_modele(config) for config in config_list]
43
44         # Enleve les scores vides
45         scores = [r for r in scores if r[1] != None]
46
47         # Trie les configurations par erreur
48         scores.sort(key=lambda tup: tup[1])
49
50         # Affiche les 3 meilleures configurations
51         print("Les_3_meilleurs_configurations_LSTM_sont:_:")
52         print(scores[0:3])
53
54         self.config = eval(scores[0][0])
55
56     def fit_modele(self, config, final=False):
57         """
58         Fait un fit rapide (100 itérations par défaut) d'un LSTM
59         selon les parametres contenus dans config (h, n, f, d)
60         """
61         resultat = None
62         key = str(config)
63
64         # nombre d'itérations max entraînement
65         iter = 500 if final else 100
66
67         nbre_couches = config.get("nbre_couches")

```

```

68     taille = config.get("taille_entree")
69     nbre_neurones = config.get("nbre_neurones")
70     activation = config.get("activation")
71     dropout = config.get("dropout")
72
73     nbre_retards = np.count_nonzero(
74         np.isnan(self.serie.data['Série_stationnarisée']))
75
76     # MinMaxScaler
77     donnees_brutes = self.serie.data['Série_stationnarisée']
78         [0:self.serie.index_fin_entrainement].dropna().values
79     scaler = MinMaxScaler(feature_range=(-1, 1))
80     scaler = scaler.fit(np.array(donnees_brutes).reshape(-1, 1))
81     serie_reduite = scaler.transform(np.array(
82         self.serie.data['Série_stationnarisée'].dropna().values)
83         .reshape(-1, 1))
84
85     a = np.empty((1, nbre_retards))
86     a[:] = np.nan
87     serie_reduite = np.concatenate((a, np.array(serie_reduite))
88         , axis=0)
89
90     self.serie.data['Série_stationnarisée_réduite'] = serie_reduite
91
92     X_train, y_train = decouper_serie_apprentissage_supervise(
93         self.serie.data['Série_stationnarisée_réduite']
94         [0:self.serie.index_fin_entrainement].dropna().values,
95         taille)
96
97     X_test, y_test = decouper_serie_apprentissage_supervise(
98         self.serie.data['Série_stationnarisée_réduite']
99         [self.serie.index_fin_entrainement:
100         self.serie.index_fin_test].dropna().values, taille)
101
102     n_features = 1 # une variable explicative
103     X_train = X_train.reshape(
104         (X_train.shape[0], X_train.shape[1], n_features))
105     X_test = X_test.reshape(
106         (X_test.shape[0], X_test.shape[1], n_features))
107
108     # Création du réseau de neurones
109     model = Sequential()
110
111     # couche d'entrée
112     for i in range(0, nbre_couches):
113         model.add(kLSTM(nbre_neurones, activation=activation,
114             return_sequences=True,
115             input_shape=(taille, n_features)))
116         model.add(Dropout(dropout)) # ajout d'un dropout
117
118     # dernière couche (pas de retour)
119     model.add(kLSTM(nbre_neurones, activation=activation))
120     model.add(Dropout(dropout)) # ajout d'un dropout
121
122     # couche de sortie (1 dimension)
123     model.add(Dense(1))
124
125     methode_optimisation = optimizers.Nadam()

```

```

125
126     model.compile(optimizer=methode_optimisation,
127                  loss='mse')
128
129     # Critere d'arret prématuré
130     # si aucune amélioration sur le jeu de test
131     # pendant plus de 20 itérations
132     critere_stop = EarlyStopping(
133         monitor='val_loss', min_delta=0, patience=20)
134
135     # Fit du modele
136     historique = model.fit(
137         X_train, y_train, validation_data=(X_test, y_test),
138         epochs=iter, verbose=final, callbacks=[critere_stop],
139         shuffle=False) # surtout ne pas "shuffle" les données
140
141     # Prédiction sur jeu de test + validation
142     serie_predite = []
143     serie_predite_temp = [] # stock les prédictions réduites
144     serie_predite_dynamique = []
145
146     # walk-forward validation (one step ahead)
147     for i in range(0, len(self.serie.data['Test']).dropna() +
148                  len(self.serie.data['Validation']).dropna()):
149
150         x_input = self.serie.data['Série_stationnarisée_réduite']
151                 [self.serie.index_fin_entrainement -
152                  taille+i:
153                   self.serie.index_fin_entrainement+i].values
154         x_input = x_input.reshape((1, taille, n_features))
155         yhat = model.predict(x_input, verbose=0)[0][0]
156         serie_predite_temp.append(yhat)
157
158     # inversion de la mise a l'échelle
159     padding = np.zeros(taille-1).reshape(1, taille-1)
160     yhat = np.append(padding, [yhat]).reshape(1, -1)
161     yhat = scaler.inverse_transform(yhat)
162     yhat = yhat[0][-1]
163
164     # déstationnarisation
165     yhat = yhat + \
166         self.serie.data['Série']
167         [self.serie.index_fin_entrainement+i-nbre_retards]
168     serie_predite.append(yhat)
169
170     # prévision dynamique
171     if final:
172         anciennes_predictions = []
173         for i in range(0, len(self.serie.data['Test']).dropna() +
174                        len(self.serie.data['Validation']).dropna()):
175             decoupe = -taille + i
176
177             if decoupe < 0:
178                 x_input_dynamique = np.append(
179                     self.serie.data['Série_stationnarisée_réduite']
180                     [decoupe:].values, anciennes_predictions)
181

```

```

182         else :
183             x_input_dynamique =
184                 np.array(anciennes_predictions)[-taille:]
185
186             x_input_dynamique = x_input_dynamique.reshape(
187                 (1, taille, n_features))
188
189             yhat_dynamique = model.predict(
190                 x_input_dynamique, verbose=0)[0][0]
191
192             anciennes_predictions.append(yhat_dynamique)
193
194             # inversion de la mise a l'échelle
195             padding = np.zeros(taille - 1).reshape(1, taille - 1)
196             yhat_dynamique = np.append(
197                 padding, [yhat_dynamique]).reshape(1, -1)
198             yhat_dynamique = scaler.inverse_transform(yhat_dynamique)
199             yhat_dynamique = yhat_dynamique[0][-1]
200
201             # déstationnarisation
202             yhat_dynamique = yhat_dynamique + \
203                 self.serie.data['Série']
204                 [self.serie.index_fin_entrainement + i
205                  - nbre_retards]
206             serie_predite_dynamique.append(yhat_dynamique)
207
208             # ajout d'un padding avec des nan
209             a = np.empty((1, len(self.serie.data['Entrainement']).dropna()))
210             a[:] = np.nan
211             serie_predite = np.concatenate((a[0], np.array(serie_predite)),
212                 axis=0)
213
214             # calcul du MSE uniquement sur le jeu de test
215             resultat = mean_squared_error(
216                 serie_predite[self.serie.index_fin_entrainement:
217                     self.serie.index_fin_test], self.serie.data['Test']
218                     .dropna())
219
220             if final:
221                 self.serie.data[self.__class__.__name__] = serie_predite
222
223                 # ajout d'un padding avec des nan
224                 a = np.empty((1, len(self.serie.data['Entrainement']
225                     .dropna())))
226                 a[:] = np.nan
227                 serie_predite_dynamique = np.concatenate(
228                     (a[0], np.array(serie_predite_dynamique)), axis=0)
229
230                 self.serie.data[self.__class__.__name__ +
231                     "_dynamique"] = serie_predite_dynamique
232                 self.modele = model
233
234             print("Fit_du_modele_" + key + " : " + str(resultat))
235             return (key, resultat)
236
237     def fit(self):
238         # Création d'un jeu de données d'apprentissage supervisé

```

```

239         print("Configuration_LSTM_retenue : " + str(self.config))
240
241         print("Entraînement du modèle retenu ... ")
242         self.fit_modele(self.config, True)

```

C.2 Programme CharNN pour la génération de Fables

Ci-dessous les deux scripts Python ayant permis d'entraîner et d'utiliser un réseau LSTM sur le jeu de données des Fables de Jean de La Fontaine. Les deux fichiers ont été raccourcis et sont disponibles en version complète sur le répertoire Github, ainsi que le jeu de données contenant les Fables.

Répertoire GitHub : <https://github.com/hugues31/charnn-fables/>.

CODE C.5 – Entraînement du CharNN

```

1  # Ce code est largement inspiré de l'exemple fourni par le framework
2  # Keras, le dépôt Github de Minesh Mathew et celui de Michael Zhang.
3
4  from keras.models import Sequential
5  from keras.layers import Dense, Activation, Dropout
6  from keras.layers import LSTM
7  from keras.layers.wrappers import TimeDistributed
8  from keras.utils.data_utils import get_file
9  import numpy as np
10 import random
11 import pickle
12 import sys
13 from keras.utils import plot_model
14
15 def sample(preds, temperature=1.0):
16     # Retourne l'indice correspondant a la probabilité max des événements
17     # du tableau preds.
18     preds = np.asarray(preds).astype('float64')
19     # Ajout d'une petite constante pour éviter log(0)
20     preds = np.log(preds + 1e-8) / temperature
21     exp_preds = np.exp(preds)
22     preds = exp_preds / np.sum(exp_preds)
23     probas = np.random.multinomial(1, preds, 1)
24     return np.argmax(probas)
25
26
27 def test_model(model, char_to_indices, indices_to_char, seed_string=" ",
28               temperature=1.0, test_length=150):
29     """
30     L'utilisation d'une température plus élevée permet de créer une
31     séquence plus originale que le contenu d'entraînement mais pénalise
32     moins les erreurs
33     """
34     num_chars = len(char_to_indices.keys())
35     for i in range(test_length):
36         test_in = np.zeros((1, len(seed_string), num_chars))
37         for t, char in enumerate(seed_string):
38             test_in[0, t, char_to_indices[char]] = 1
39     entire_prediction = model.predict(test_in, verbose=0)[0]

```

```

40         next_index = sample(entire_prediction[-1], temperature)
41         next_char = indices_to_char[next_index]
42         seed_string = seed_string + next_char
43     return seed_string
44
45 # Parametres généraux
46 origin = "version_1"
47 seed = 2
48
49 # LSTM
50 unit_size = 512
51 num_layers = 3
52 dropout = 0.4
53
54 # Parametres d'optimisation
55 optimizer = 'rmsprop'
56 training_epochs = 100
57
58 maxlen = 200 # Longueur max des phrases
59 step = 13    # chevauchement de 13 caracteres
60
61 test_length = 240 # Longueur de la phrase de test a générer
62
63 # Jeu de données
64 text = open("dataset_fables.txt").read()
65
66 # Suppression des caracteres accentués (83 -> 61 caracteres)
67 # ...
68
69 print('Longueur_du_corpus:', len(text))
70 chars = sorted(list(set(text)))
71 num_chars = len(chars)
72 print('Liste_des_caracteres:', chars)
73 print('Nombre_de_caracteres_différents:', num_chars)
74
75 # Dictionnaires permettant de convertir les caracteres en nombre
76 # et vice-versa
77 char_to_indices = dict((c, i) for i, c in enumerate(chars))
78 indices_to_char = dict((i, c) for i, c in enumerate(chars))
79
80 # Découpe le corpus en phrase semi-redondante de longueur maxlen
81 sentences = []
82 targets = []
83 for i in range(0, len(text) - maxlen - 1, step):
84     sentences.append(text[i: i + maxlen])
85     targets.append(text[i + 1: i + maxlen + 1])
86 print('Nombre_de_séquences:', len(sentences))
87
88 # La vectorisation permet de ne pas avoir a envoyer directement des
89 # nombres (correspondant aux lettres) au réseau de neurones. Il ne
90 # s'agit pas de variables ordinales mais de variables catégorielles.
91 # On encode donc les nombres en tableau de 0 et de 1. (one hot
92 # encoding)
93 X = np.zeros((len(sentences), maxlen, num_chars), dtype=np.bool)
94 y = np.zeros((len(sentences), maxlen, num_chars), dtype=np.bool)
95 for i in range(len(sentences)):
96     sentence = sentences[i]

```

```

97     target = targets[i]
98     for j in range(maxlen):
99         X[i][j][char_to_indices[sentence[j]]] = 1
100        y[i][j][char_to_indices[target[j]]] = 1
101
102    print('Construction du modele ...')
103    model = Sequential()
104
105    model.add(LSTM(unit_size, input_dim=num_chars, return_sequences=True))
106    for i in range(num_layers - 1):
107        if dropout:
108            model.add(Dropout(dropout))
109        model.add(LSTM(unit_size, return_sequences=True))
110    if dropout:
111        model.add(Dropout(dropout))
112    model.add(TimeDistributed(Dense(num_chars)))
113    model.add(Activation('softmax'))
114    model.compile(optimizer=optimizer,
115                  loss='categorical_crossentropy',
116                  )
117    print(model.summary())
118    plot_model(model, to_file='model.png')
119    print('Modele construit.')
120
121    pertes = []
122
123    # ----- Entrainement -----
124    for i in range(training_epochs):
125        print('-' * 10 + ' Iteration : {}'.format(i) + '-' * 10)
126
127        # On teste le réseau toutes les 5 itérations
128        if i % 5 == 0:
129            for temperature in [0.3, 0.6, 1, 1.5, 2]:
130                generated_string = test_model(model,
131                                             char_to_indices=char_to_indices,
132                                             indices_to_char=indices_to_char,
133                                             temperature=temperature,
134                                             test_length=test_length)
135                output = "\n\nTemperature : {}\n\n"
136                "Fable générée : {}".format(temperature, generated_string)
137                print(output)
138                model.save("modeles/{}-{}.h5".format(origin, i))
139
140            history = model.fit(X, y, batch_size=256, epochs=1, verbose=1)
141            perte = history.history['loss'][0]
142            print('La perte est de {}'.format(perte))
143            pertes.append(perte)
144
145    model.save("modeles/{}.h5".format(origin))
146    print("Evolution de la perte : " + str(pertes))

```

CODE C.6 – Test de génération sur un modèle déjà entraîné

```

1 import os
2 from keras.models import load_model
3 # ... meme imports que précédemment
4

```

```

5 def sample(preds, temperature=1.0):
6     # Retourne l'indice correspondant a la probabilité max des événements
7     # du tableau preds.
8     # meme fonction que précédemment
9
10 def test_model(model, char_to_indices, indices_to_char, seed_string=" ",
11               temperature=1.0, test_length=150):
12     # meme fonction que précédemment
13
14 # Parametres
15 origin = "version_1"
16 seed = 2
17
18 maxlen = 200 # Longueur max des phrases
19 step = 13
20
21 # Longueur de la phrase de test a générer
22 test_length = 1000
23
24 # Jeu de données
25 text = open("dataset_fables.txt").read()
26
27 # Suppression des caracteres accentués (83 -> 61 caracteres)
28 # ...
29
30 np.random.seed(seed)
31 random.seed(seed)
32
33 print('Longueur_du_corpus:', len(text))
34 chars = sorted(list(set(text)))
35 num_chars = len(chars)
36 print('Liste_des_caracteres:', chars)
37 print('Nombre_de_caracteres_différents:', num_chars)
38
39 # Dictionnaires permettant de convertir les caracteres
40 # en nombre et vice-versa
41 char_to_indices = dict((c, i) for i, c in enumerate(chars))
42 indices_to_char = dict((i, c) for i, c in enumerate(chars))
43
44 # Découpe le corpus en phrase semi-redondante de longueur maxlen
45 sentences = []
46 targets = []
47 for i in range(0, len(text) - maxlen - 1, step):
48     sentences.append(text[i: i + maxlen])
49     targets.append(text[i + 1: i + maxlen + 1])
50 print('Nombre_de_séquences:', len(sentences))
51
52 print('Vectorisation...')
53 # Vectorisation identique a la phase d'entrainement
54 # ...
55
56 print('Construction_du_modele...')
57 # On charge le modele entraîné complètement (itération 100)
58 modele = load_model("modeles/version_1.h5")
59 print(modele.summary())
60 print('Modele_construit.')
61

```

```
62 # Enregistrement du texte généré en fichier texte .
63 outfile = open("sortie /{}.txt".format(origin), "w")
64
65 pertes = []
66
67 # ----- Phase de test -----
68 for temperature in [0.8]:
69     # Initialisation avec la fin d'un poeme (pour obtenir le suivant)
70     generated_string = test_model(modele ,
71                                   char_to_indices=char_to_indices ,
72                                   indices_to_char=indices_to_char ,
73                                   temperature=temperature ,
74                                   test_length=test_length ,
75                                   seed_string="\n\n\n")
76
77     output = "\n\nTemperature : {} \nFable générée : \n {}".format(
78         temperature , generated_string)
79     print(output)
```


Annexe D

Résultats détaillés du benchmark

D.1 Série 1 - Tendence linéaire

FIGURE D.1 – Subdivision de la série 1 en set d'entraînement, test et validation

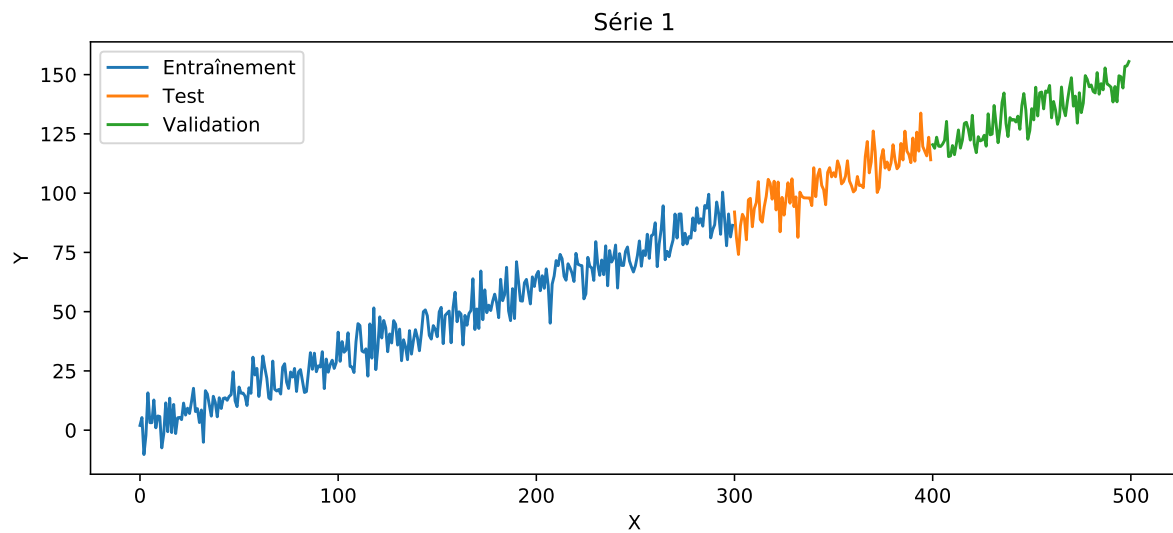


TABLE D.1 – Liste des modèles SARIMA testés sur la série 1

p	d	q	P	D	Q	s	AIC	BIC
2	1	2	0	0	0	1	2667.729	2687.674
0	1	0	0	0	0	1	2879.302	2883.291
1	1	0	0	0	0	1	2756.414	2764.392
0	1	1	0	0	0	1	2663.146	2671.124
1	1	1	0	0	0	1	2663.731	2675.698
0	1	2	0	0	0	1	2663.681	2675.648
1	1	2	0	0	0	1	2665.017	2680.973

CODE D.1 – Description du modèle SARIMA, série 1

Nombre d'observations : 400				AIC : 2663.146	
Modele retenu : SARIMA(0, 1, 1)				BIC : 2671.124	
Log vraisemblance : -1329.573				HQIC : 2666.306	
	Coefficient	Ecart-type	z	P> z	[0.025 0.975]
ma.L1	-0.8048	0.031	-26.274	0.000	-0.865 -0.745
sigma2	45.7885	3.354	13.651	0.000	39.214 52.363
Ljung-Box (Q):		43.67	Jarque-Bera (JB):		0.27
Prob(Q):		0.32	Prob(JB):		0.88
Heteroskedasticity (H):		1.07	Skewness:		-0.02
Prob(H) (two-sided):		0.69	Kurtosis:		2.88

FIGURE D.2 – Diagnostic des résidus du modèle SARIMA pour la série 1

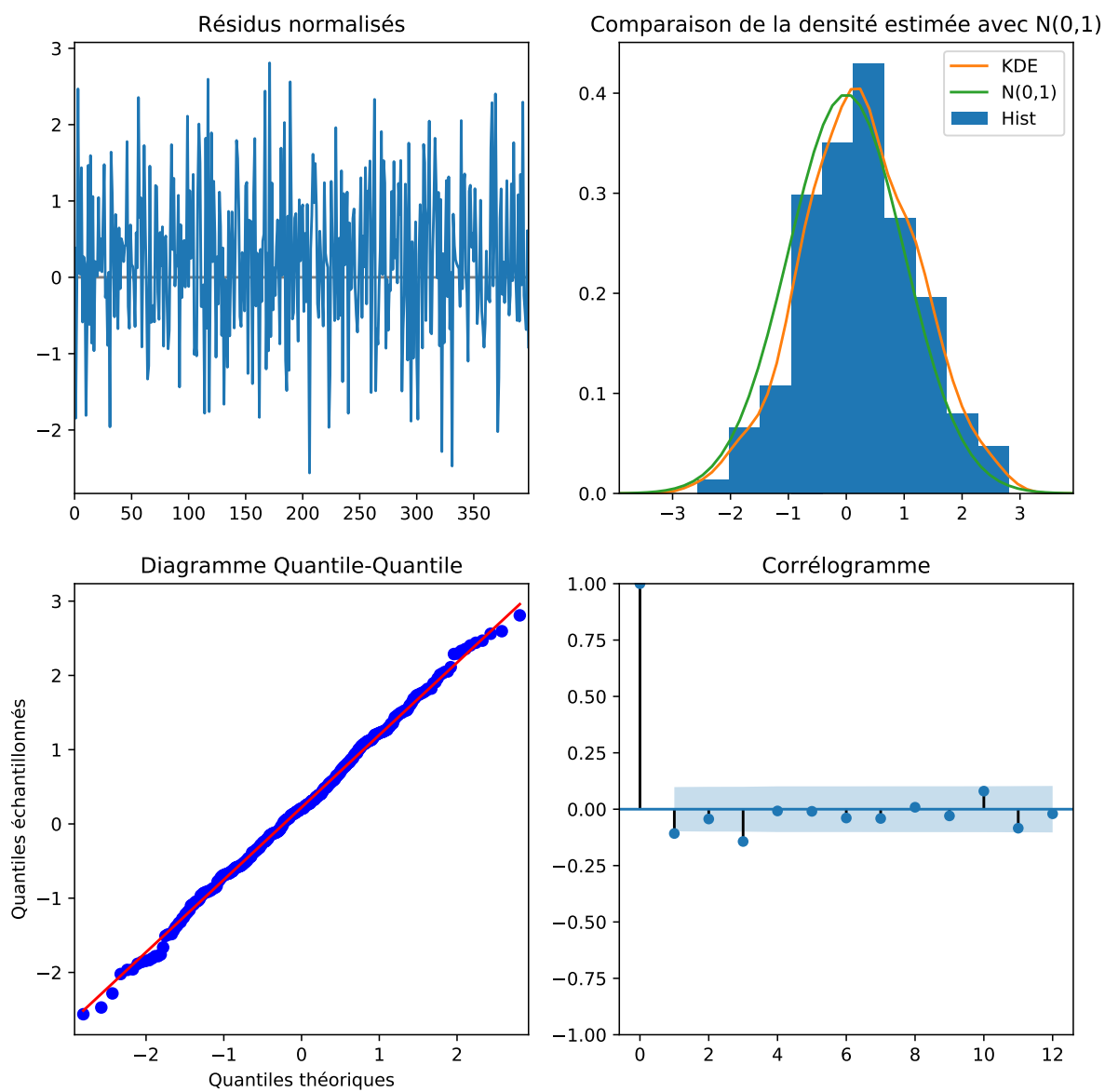
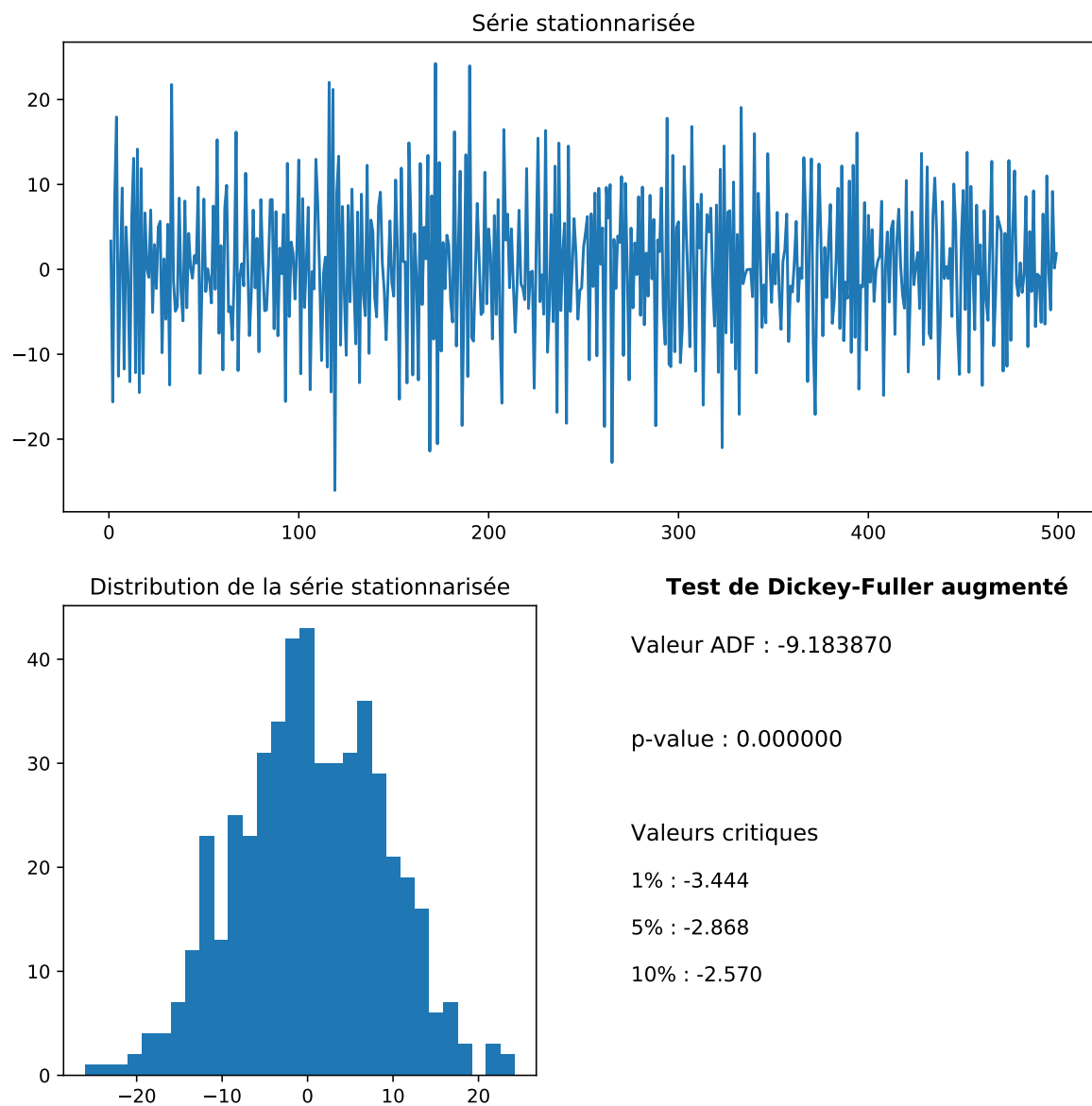


FIGURE D.3 – Test de Dickey-Fuller augmenté, série 1



Nombre de couches	Taille entrée	Nombre de neurones	Activation	Dropout	MSE
1	50	10	tanh	0.2	43.381
3	10	200	tanh	0.2	43.525
3	50	50	tanh	0.2	43.791

TABLE D.2 – Liste des modèles LSTM testés sur la série 1

FIGURE D.4 – Évolution de la perte du LSTM, série 1

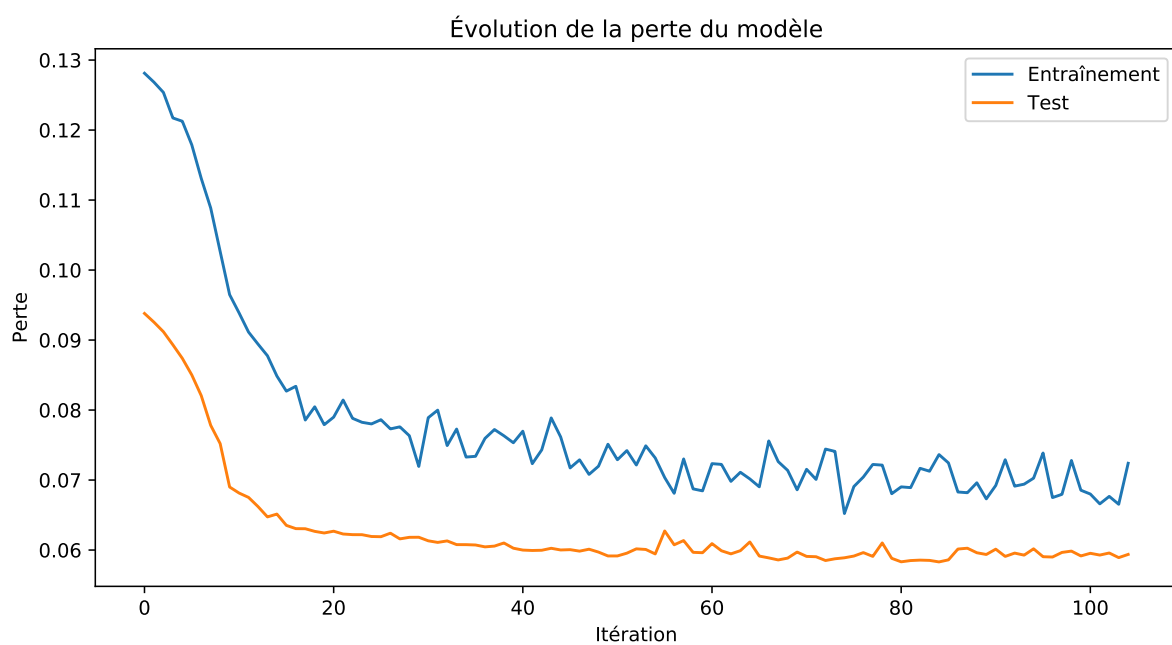


FIGURE D.5 – Prévisions one step ahead, série 1

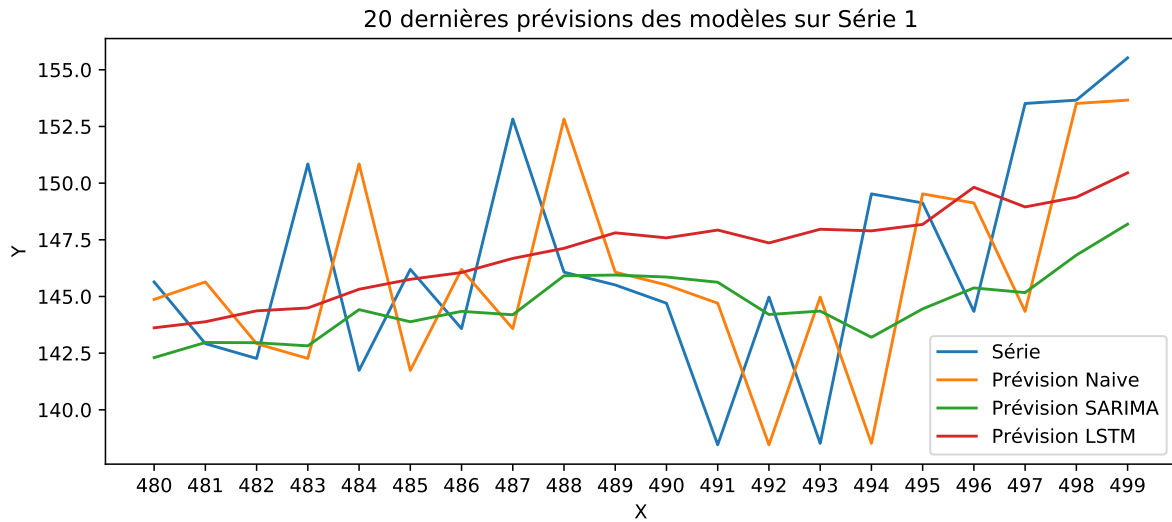
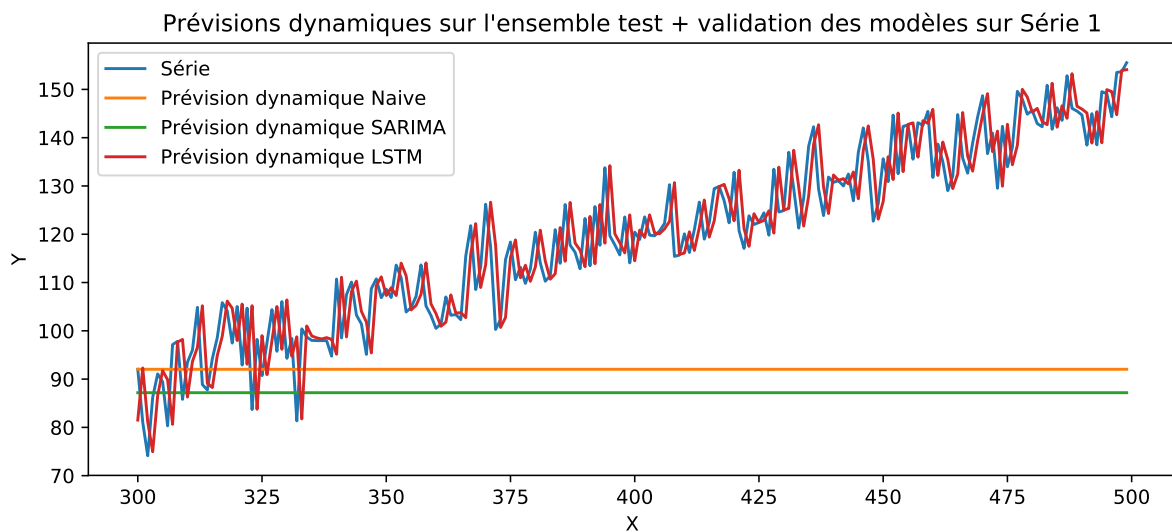


FIGURE D.6 – Prévisions dynamiques, série 1



CODE D.2 – Mesure de l'erreur, série 1

```

===      Série 1      ===
| Observations : 500
|
| — Naive —
|
| — Prévision one step ahead —
| MSE      : 49.571263285893494
| MedAE    : 5.217157264755954
| Var. expl : 0.5403024816587483
| R^2      : 0.5387033996977211
|
| — Prévision dynamique —
| MSE      : 1871.1519513332587
| MedAE    : 41.11986496467945
| Var. expl : -2.220446049250313e-16
| R^2      : -16.412427615994126
|
|
| — SARIMA —
|
| — Prévision one step ahead —
| MSE      : 33.83210319891686
| MedAE    : 3.6584080855263252
| Var. expl : 0.7075885561816518
| R^2      : 0.6851677130613416
|
| — Prévision dynamique —
| MSE      : 2302.219746361397
| MedAE    : 45.97179625717773
| Var. expl : -2.220446049250313e-16
| R^2      : -20.423826462124936
|
|
| — LSTM —
|
| — Prévision one step ahead —
| MSE      : 30.688973704712502
| MedAE    : 4.244950375487733
| Var. expl : 0.7276558623651137
| R^2      : 0.7144168153411072
|
| — Prévision dynamique —
| MSE      : 49.399907772584555
| MedAE    : 5.183487761315078
| Var. expl : 0.5402986933628562
| R^2      : 0.5402979871762887
|

```

D.2 Série 2 - Tendence linéaire avec saisonnalité simple

FIGURE D.7 – Subdivision de la série 2 en set d'entraînement, test et validation

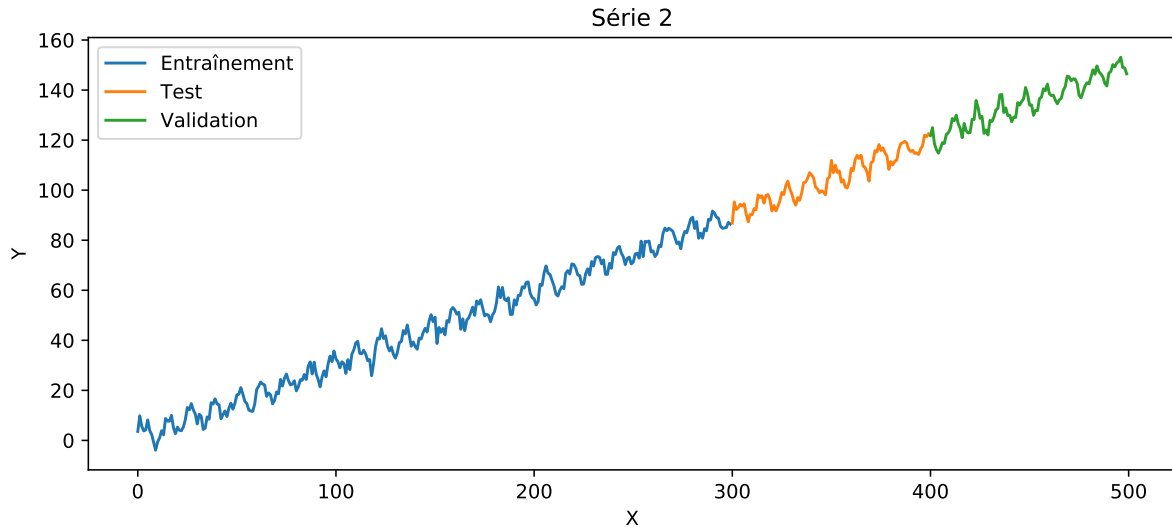


TABLE D.3 – Liste des modèles SARIMA testés sur la série 2

p	d	q	P	D	Q	s	AIC	BIC
2	1	2	2	0	1	12	<i>nan</i>	<i>nan</i>
0	1	0	0	0	0	12	2029.956	2033.945
1	1	1	1	0	1	12	1756.015	1775.960
1	1	0	1	0	1	12	1843.498	1859.453

CODE D.3 – Description du modèle SARIMA, série 2

Nombre d'observations : 400				AIC :	1756.015	
Modele retenu : SARIMA(1, 1, 1)x(1, 0, 1, 12)				BIC :	1775.960	
Log vraisemblance : -873.008				HQIC :	1763.914	
	Coefficient	Ecart-type	z	P> z	[0.025	0.975]
ar.L1	0.0251	0.014	1.826	0.068	-0.002	0.052
ma.L1	-0.9018	0.034	-26.861	0.000	-0.968	-0.836
ar.S.L12	0.9991	0.001	743.322	0.000	0.996	1.002
ma.S.L12	-0.9289	0.050	-18.544	0.000	-1.027	-0.831
sigma2	4.1950	0.327	12.841	0.000	3.555	4.835
Ljung-Box (Q):				37.51	Jarque-Bera (JB):	
Prob(Q):				0.58	3.54	
Heteroskedasticity (H):				0.62	Prob(JB):	
Prob(H) (two-sided):				0.01	0.17	
					Skewness:	
					-0.21	
					Kurtosis:	
					3.18	

FIGURE D.8 – Diagnostic des résidus du modèle SARIMA pour la série 2

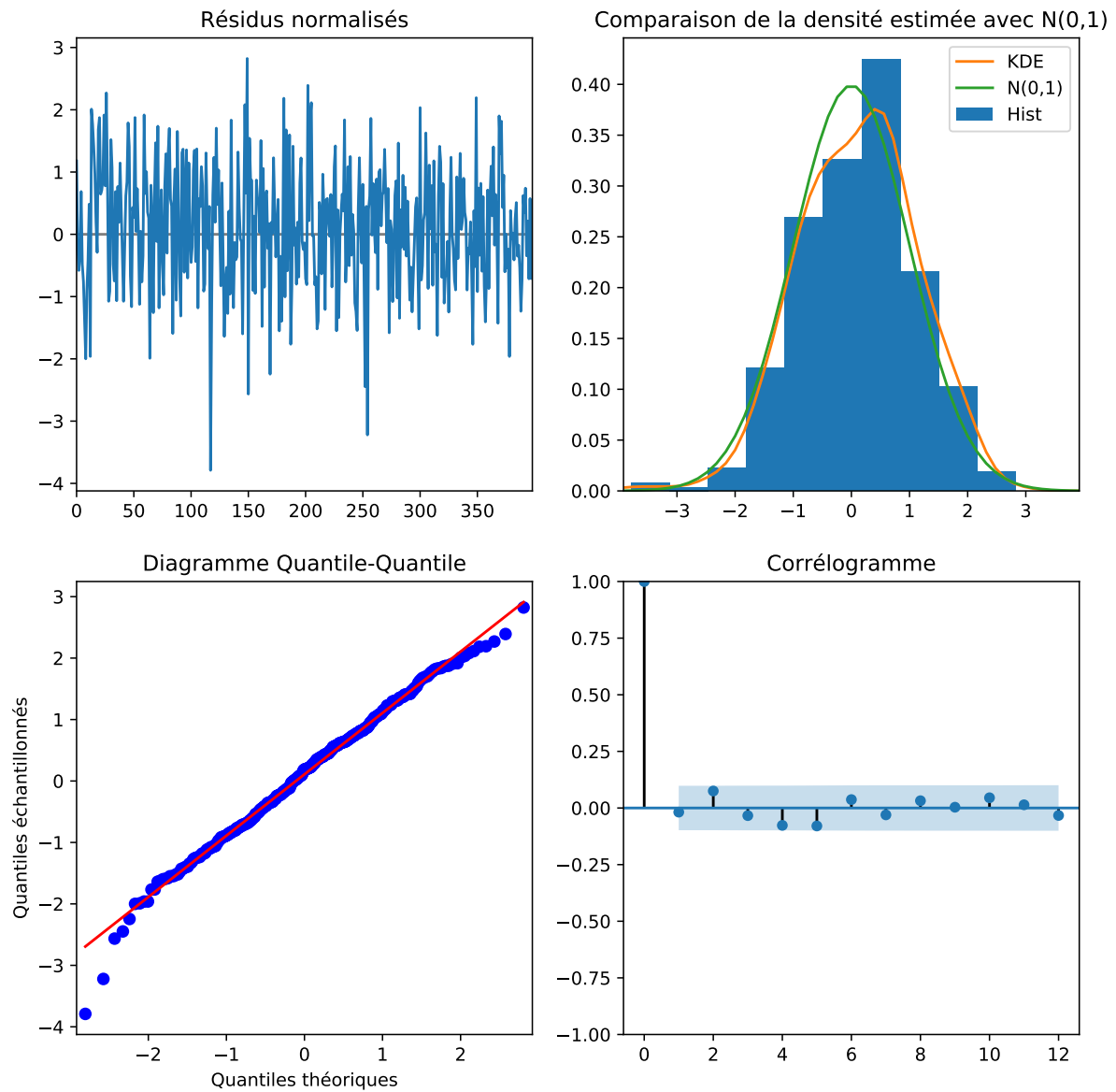
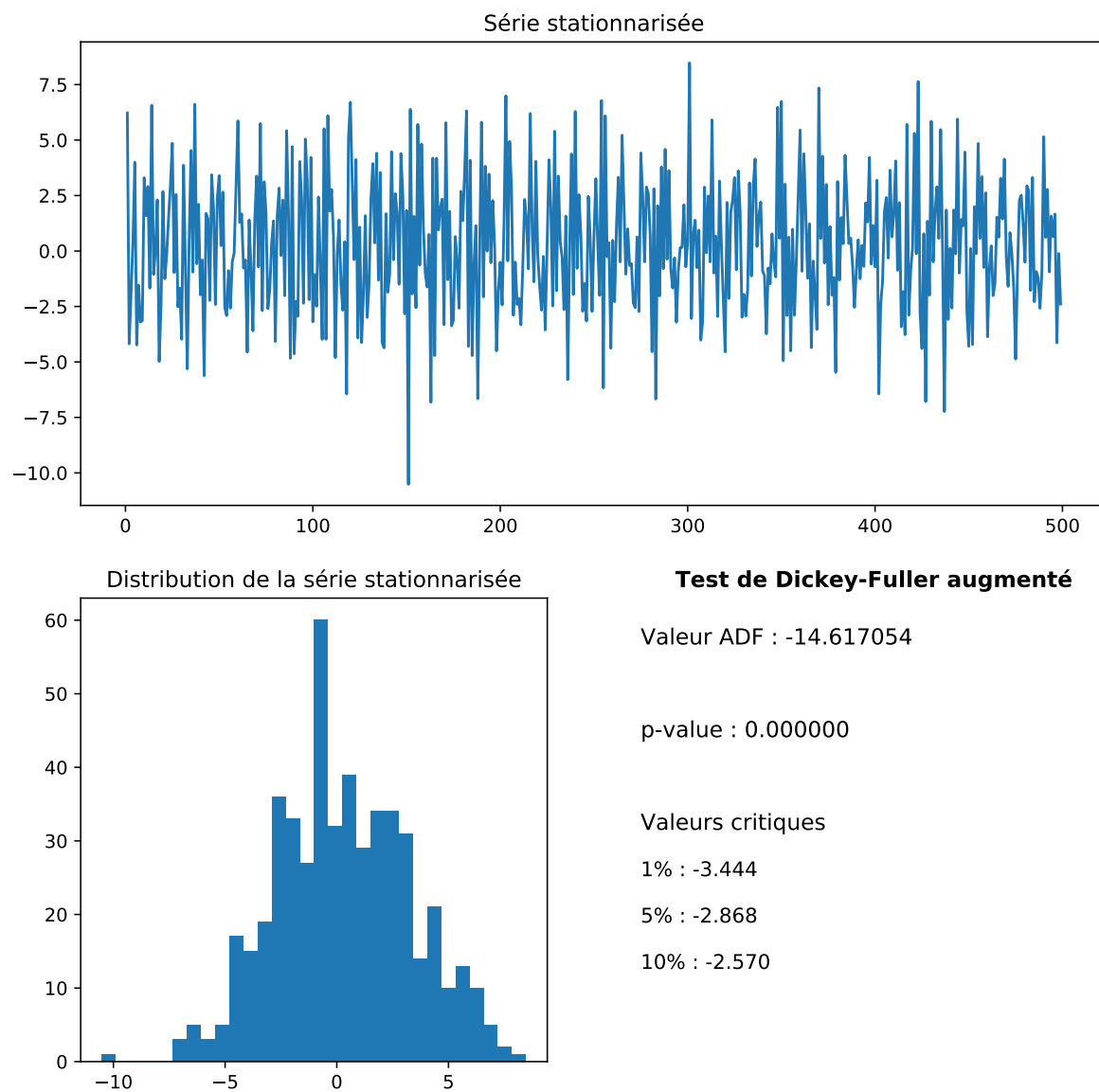


FIGURE D.9 – Test de Dickey-Fuller augmenté, série 2



Nombre de couches	Taille entrée	Nombre de neurones	Activation	Dropout	MSE
1	50	10	tanh	0.0	3.475
1	50	200	tanh	0.0	3.589
1	50	50	tanh	0.2	3.638

TABLE D.4 – Liste des modèles LSTM testés sur la série 2

FIGURE D.10 – Évolution de la perte du LSTM, série 2

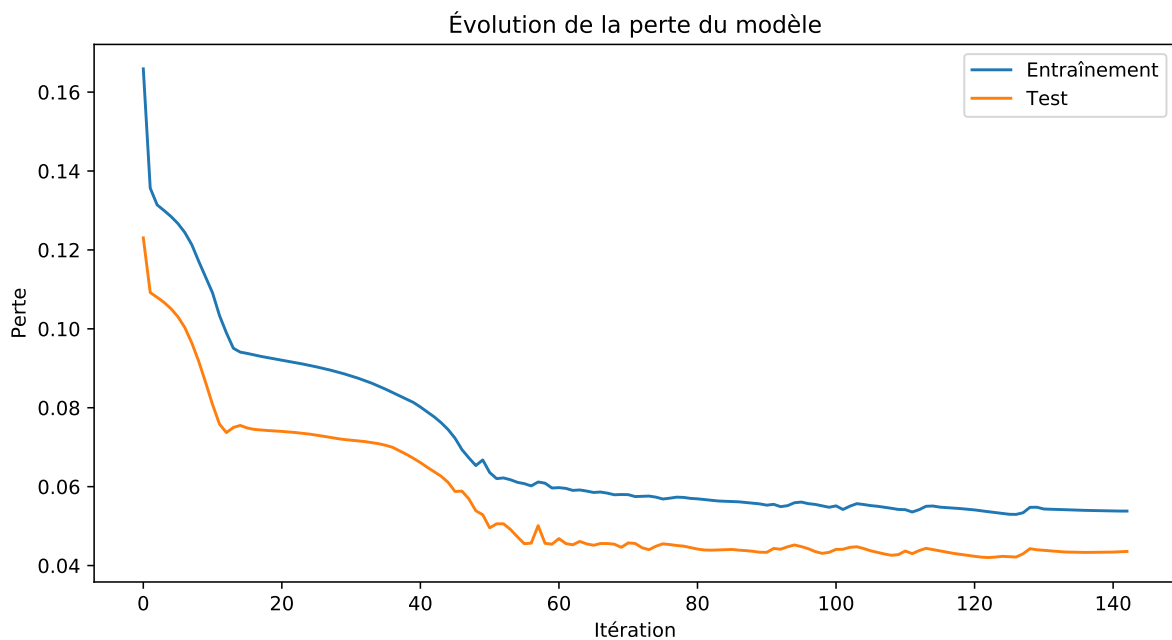


FIGURE D.11 – Prévisions one step ahead, série 2

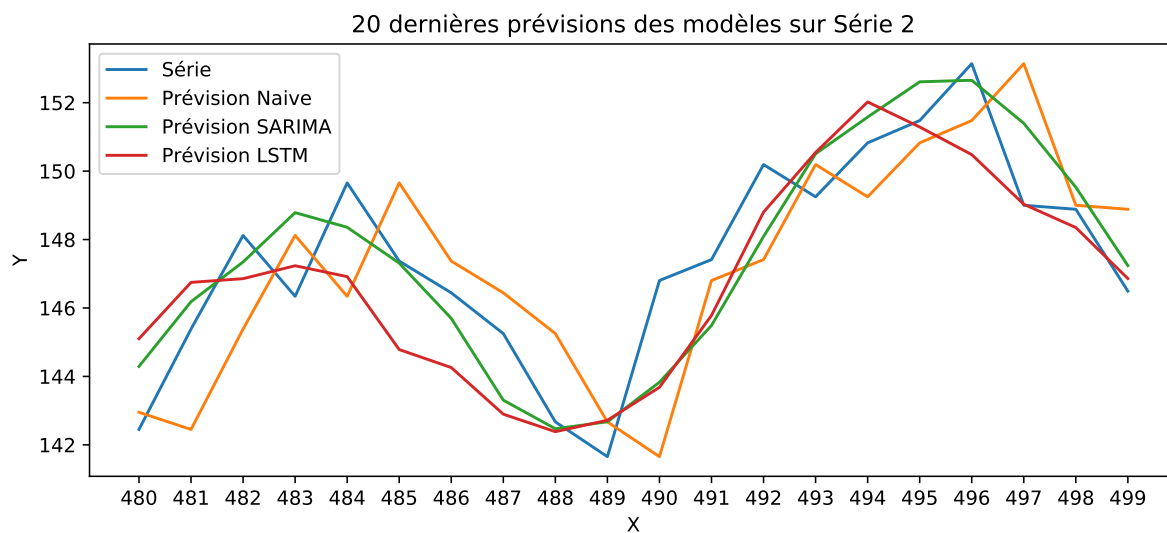
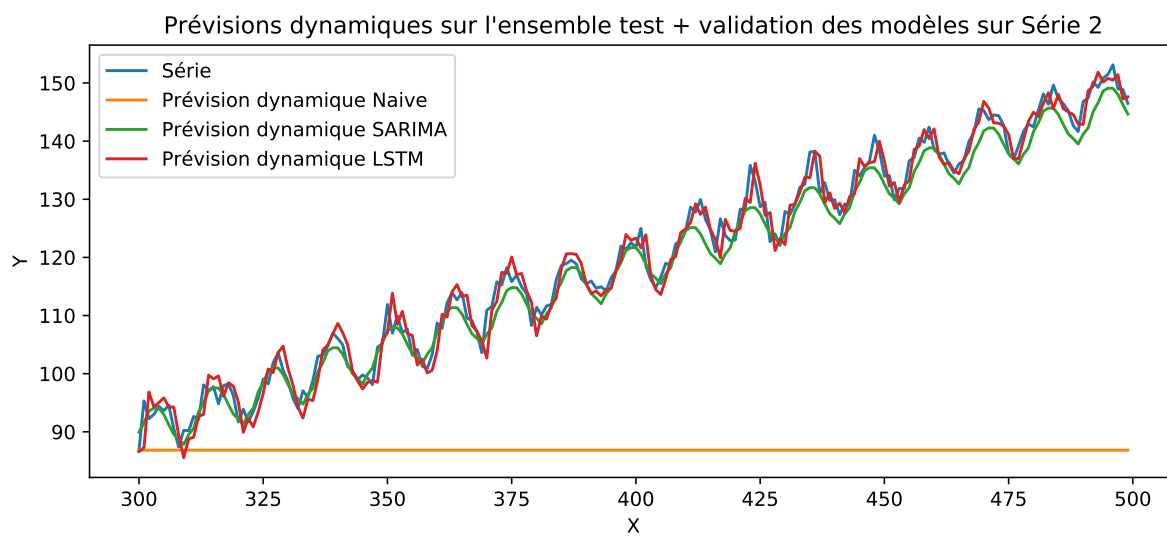


FIGURE D.12 – Prévisions dynamiques, série 2



CODE D.4 – Mesure de l'erreur, série 2

```

===      Série 2      ===
| Observations : 500
|
|   — Naive —
|
|   — Prévvision one step ahead —
|   MSE      : 8.529168054211919
|   MedAE    : 1.8373318941895533
|   Var. expl : 0.9037264621370975
|   R^2      : 0.9030732598411584
|
|   — Prévvision dynamique —
|   MSE      : 2401.1922075938323
|   MedAE    : 48.794222224791895
|   Var. expl : 0.0
|   R^2      : -26.28750702267497
|
|
|   — SARIMA —
|
|   — Prévvision one step ahead —
|   MSE      : 4.28664113555994
|   MedAE    : 1.2838902270045836
|   Var. expl : 0.951356761842714
|   R^2      : 0.951285969644432
|
|   — Prévvision dynamique —
|   MSE      : 9.521765969240453
|   MedAE    : 2.301233797149088
|   Var. expl : 0.9571217074520513
|   R^2      : 0.8917932288251593
|
|
|   — LSTM —
|
|   — Prévvision one step ahead —
|   MSE      : 4.798434473771315
|   MedAE    : 1.362010726239447
|   Var. expl : 0.9479906870237746
|   R^2      : 0.9454698736790883
|
|   — Prévvision dynamique —
|   MSE      : 6.25146286769679
|   MedAE    : 1.6845684486051198
|   Var. expl : 0.929193089225539
|   R^2      : 0.9289574419054074
|

```

D.3 Série 3 - Tendance linéaire avec saisonnalité multiplicative

FIGURE D.13 – Subdivision de la série 3 en set d'entraînement, test et validation

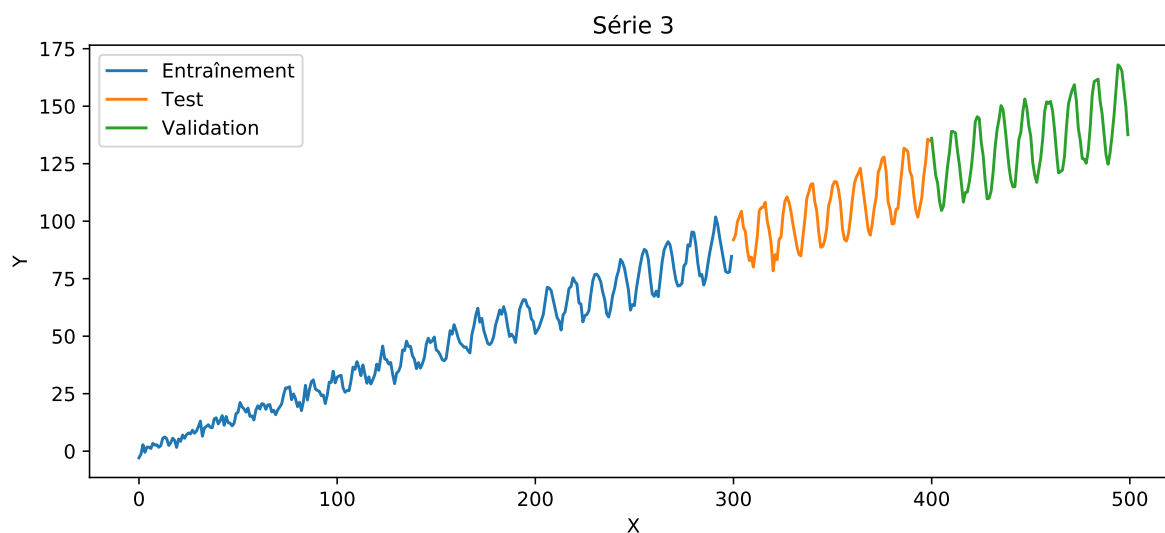


TABLE D.5 – Liste des modèles SARIMA testés sur la série 3

p	d	q	P	D	Q	s	AIC	BIC
2	1	2	2	0	1	12	<i>nan</i>	<i>nan</i>
0	1	0	0	0	0	12	2302.628	2306.617
1	1	1	2	0	2	12	1811.082	1839.005
1	1	0	2	0	2	12	1909.126	1933.060

CODE D.5 – Description du modèle SARIMA, série 3

Nombre d'observations : 400				AIC : 1811.082		
Modele retenu : SARIMA(1, 1, 1)x(2, 0, 2, 12)				BIC : 1839.005		
Log vraisemblance : -898.541				HQIC : 1822.141		
	Coefficient	Ecart-type	z	P> z	[0.025	0.975]
ar.L1	0.1680	0.054	3.084	0.002	0.061	0.275
ma.L1	-0.9500	0.018	-53.012	0.000	-0.985	-0.915
ar.S.L12	-0.0049	0.045	-0.111	0.912	-0.092	0.082
ar.S.L24	0.9887	0.047	21.041	0.000	0.897	1.081
ma.S.L12	0.4275	0.110	3.892	0.000	0.212	0.643
ma.S.L24	-0.5491	0.096	-5.731	0.000	-0.737	-0.361
sigma2	4.7906	0.498	9.626	0.000	3.815	5.766
Ljung-Box (Q):	122.64			Jarque-Bera (JB):	0.43	
Prob(Q):	0.00			Prob(JB):	0.81	
Heteroskedasticity (H):	0.90			Skewness:	-0.06	
Prob(H) (two-sided):	0.56			Kurtosis:	3.11	

FIGURE D.14 – Diagnostic des résidus du modèle SARIMA pour la série 3

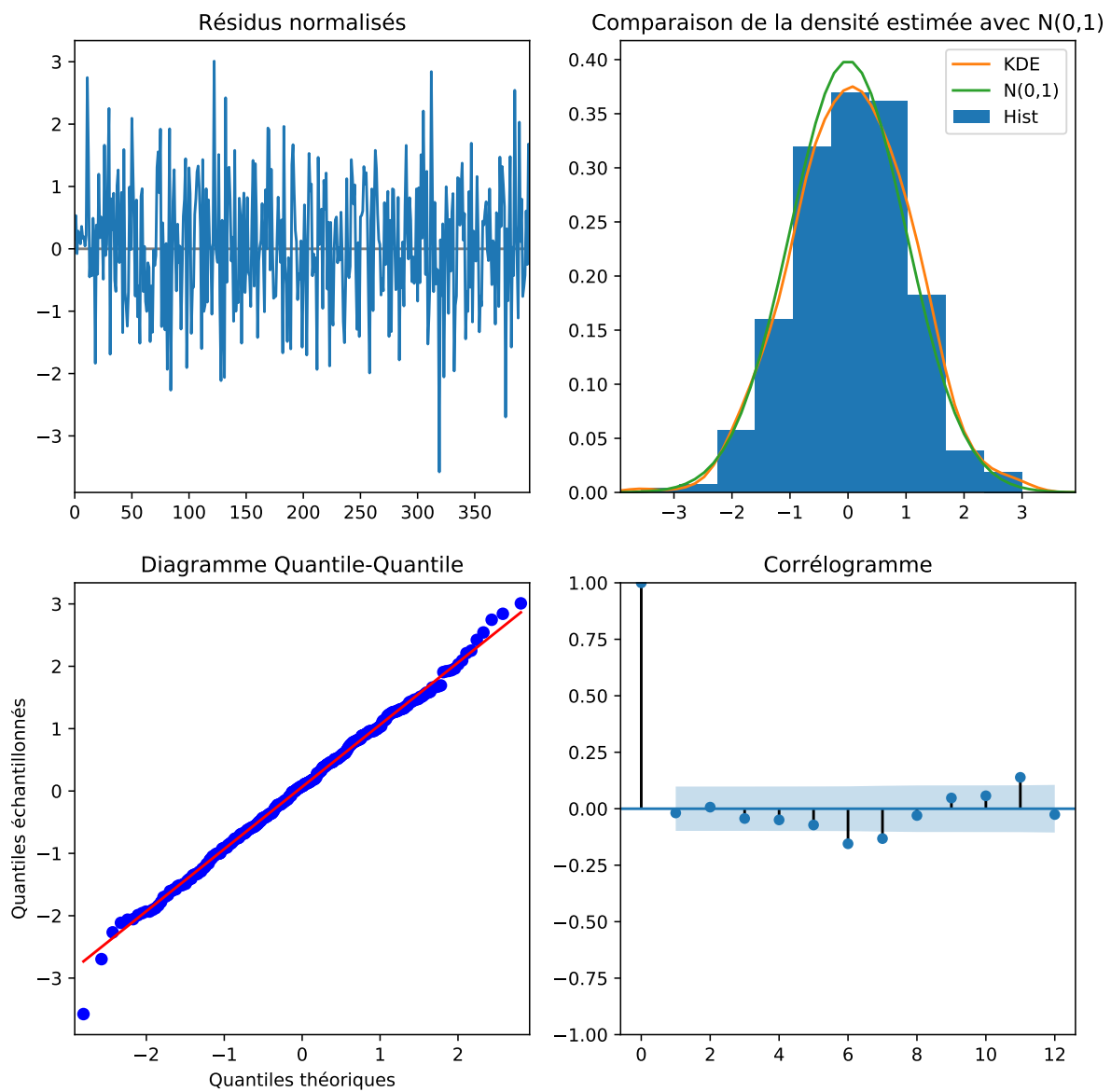
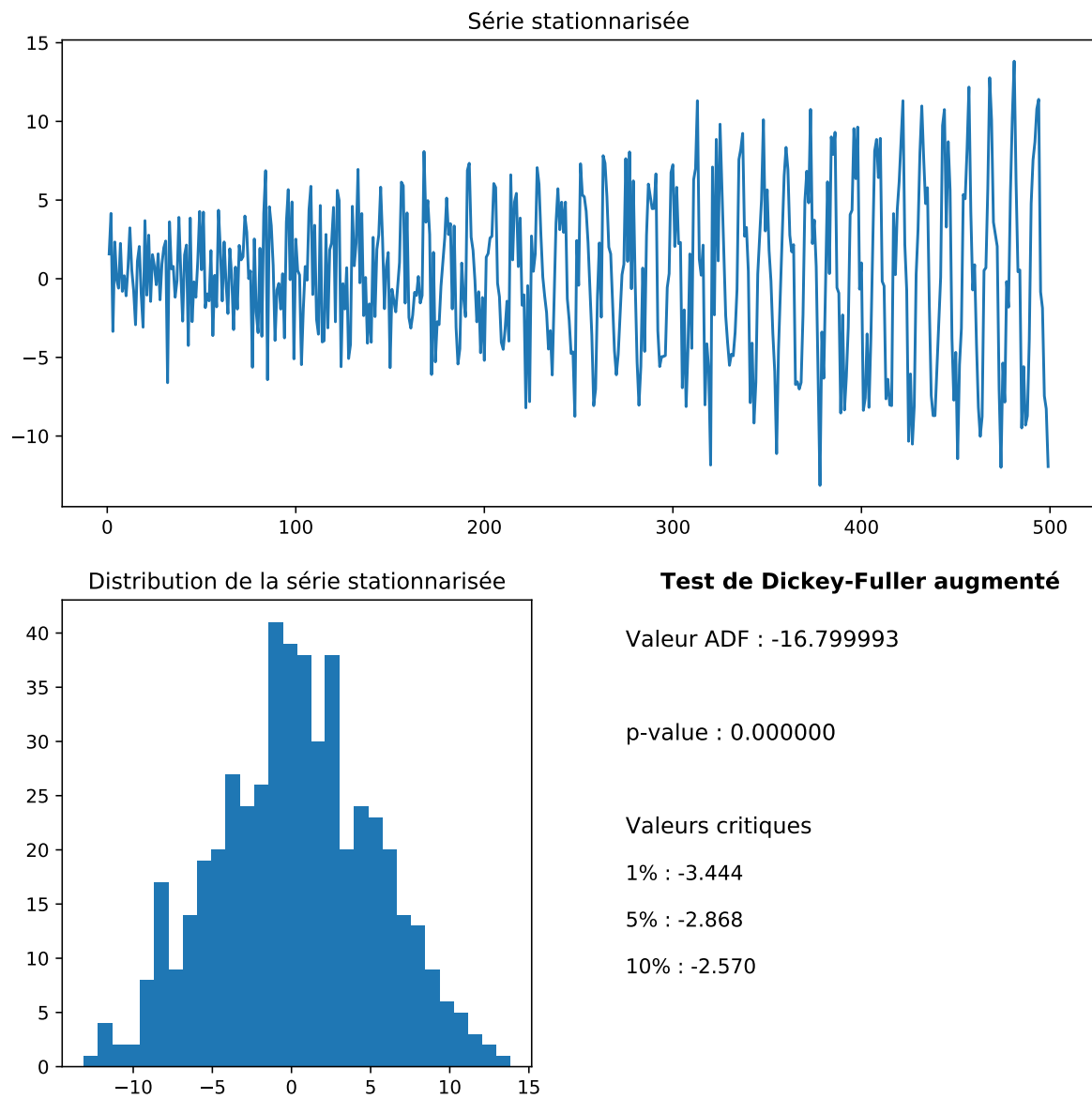


FIGURE D.15 – Test de Dickey-Fuller augmenté, série 3



Nombre de couches	Taille entrée	Nombre de neurones	Activation	Dropout	MSE
1	20	50	tanh	0.2	7.161
1	20	50	sigmoïde	0.0	7.293
1	50	50	tanh	0.2	7.378

TABLE D.6 – Liste des modèles LSTM testés sur la série 3

FIGURE D.16 – Évolution de la perte du LSTM, série 3

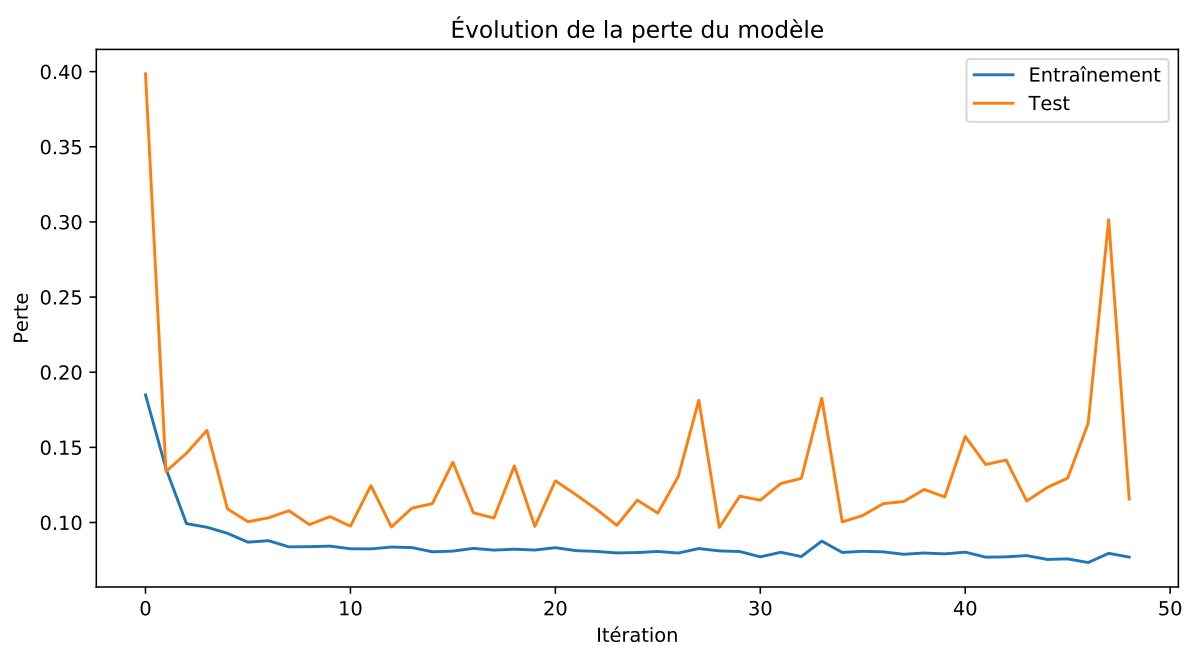


FIGURE D.17 – Prévisions one step ahead, série 3

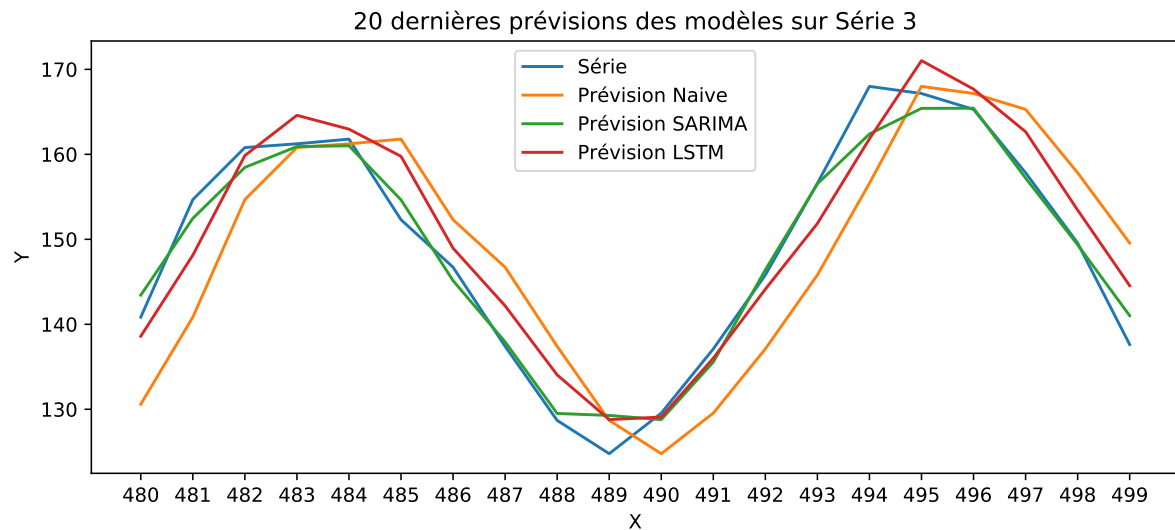
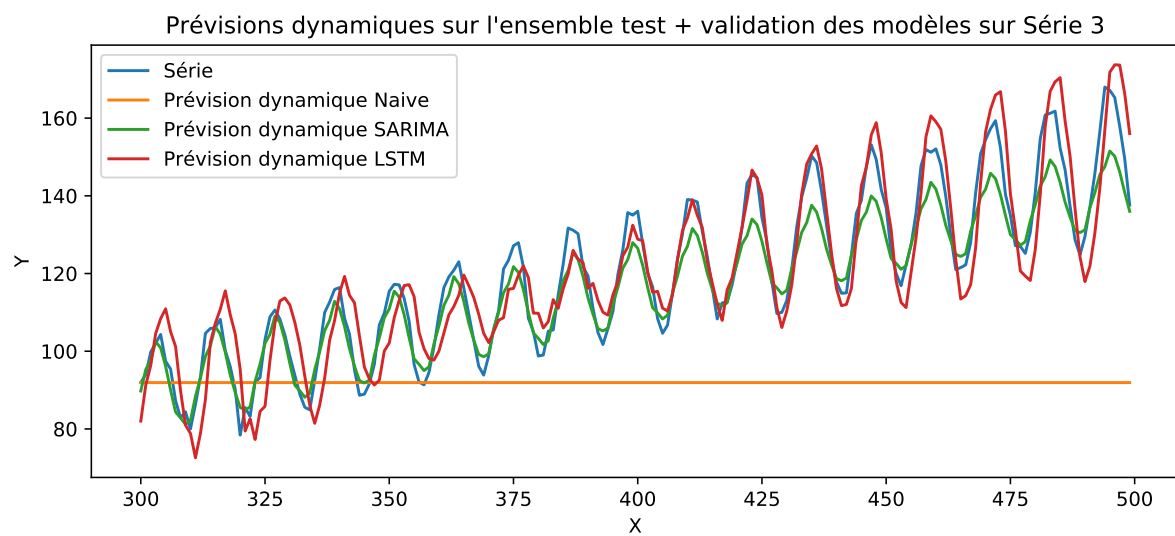


FIGURE D.18 – Prévisions dynamiques, série 3



CODE D.6 – Mesure de l'erreur, série 3

```

===      Série 3      ===
| Observations : 500
|
| — Naive —
|
| — Prévision one step ahead —
| MSE      : 49.76759329378947
| MedAE    : 6.1533312422293704
| Var. expl : 0.7926192231419318
| R^2      : 0.792616425769853
|
| — Prévision dynamique —
| MSE      : 2079.551370955126
| MedAE    : 43.31365774235469
| Var. expl : 0.0
| R^2      : -7.665574675432298
|
|
| — SARIMA —
|
| — Prévision one step ahead —
| MSE      : 4.811177574516698
| MedAE    : 1.8067450218330094
| Var. expl : 0.9799696904799533
| R^2      : 0.979951628446865
|
| — Prévision dynamique —
| MSE      : 63.58471845151192
| MedAE    : 5.484194831350997
| Var. expl : 0.838904021974238
| R^2      : 0.7350399063693992
|
|
| — LSTM —
|
| — Prévision one step ahead —
| MSE      : 12.159002172210105
| MedAE    : 2.493719008118987
| Var. expl : 0.9550960313418769
| R^2      : 0.9493329461471122
|
| — Prévision dynamique —
| MSE      : 59.4553220399806
| MedAE    : 4.730586630946476
| Var. expl : 0.7532811966759538
| R^2      : 0.7522472682400279
|

```

D.4 Série 4 - Tendence non polynomiale avec saisonnalités imbriquées

FIGURE D.19 – Subdivision de la série 4 en set d'entraînement, test et validation

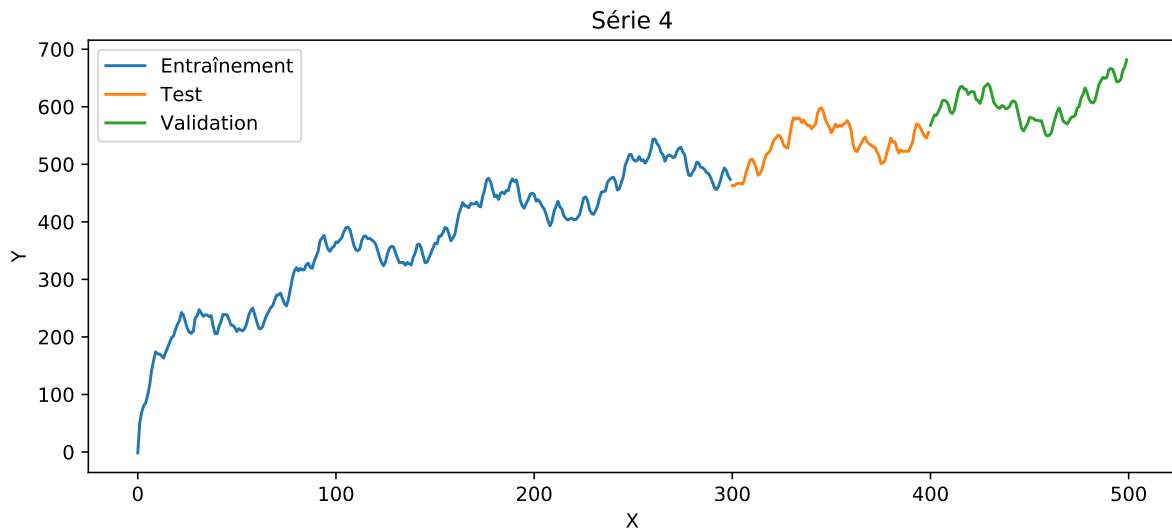


TABLE D.7 – Liste des modèles SARIMA testés sur la série 4

p	d	q	P	D	Q	s	AIC	BIC
2	1	2	2	0	1	12	2533.680	2565.591
0	1	0	0	0	0	12	2896.647	2900.636
2	1	2	1	0	1	12	2529.082	2557.005
2	1	2	1	0	0	12	2563.822	2587.756

CODE D.7 – Description du modèle SARIMA, série 4

Nombre d'observations : 400				AIC :	2529.082
Modele retenu : SARIMA(2, 1, 2)x(1, 0, 1, 12)				BIC :	2557.005
Log vraisemblance : -1257.541				HQIC :	2540.141
	Coefficient	Ecart-type	z	P> z	[0.025 0.975]
ar.L1	1.2451	0.004	286.048	0.000	1.237 1.254
ar.L2	-0.9958	0.001	-679.468	0.000	-0.999 -0.993
ma.L1	-1.1619	0.018	-63.577	0.000	-1.198 -1.126
ma.L2	0.9842	0.028	35.379	0.000	0.930 1.039
ar.S.L12	0.9858	0.009	112.057	0.000	0.969 1.003
ma.S.L12	-0.8545	0.044	-19.260	0.000	-0.941 -0.768
sigma2	30.7231	1.911	16.079	0.000	26.978 34.468
Ljung-Box (Q):		281.43	Jarque-Bera (JB):		229.48
Prob(Q):		0.00	Prob(JB):		0.00
Heteroskedasticity (H):		0.44	Skewness:		0.60
Prob(H) (two-sided):		0.00	Kurtosis:		6.52

FIGURE D.20 – Diagnostic des résidus du modèle SARIMA pour la série 4

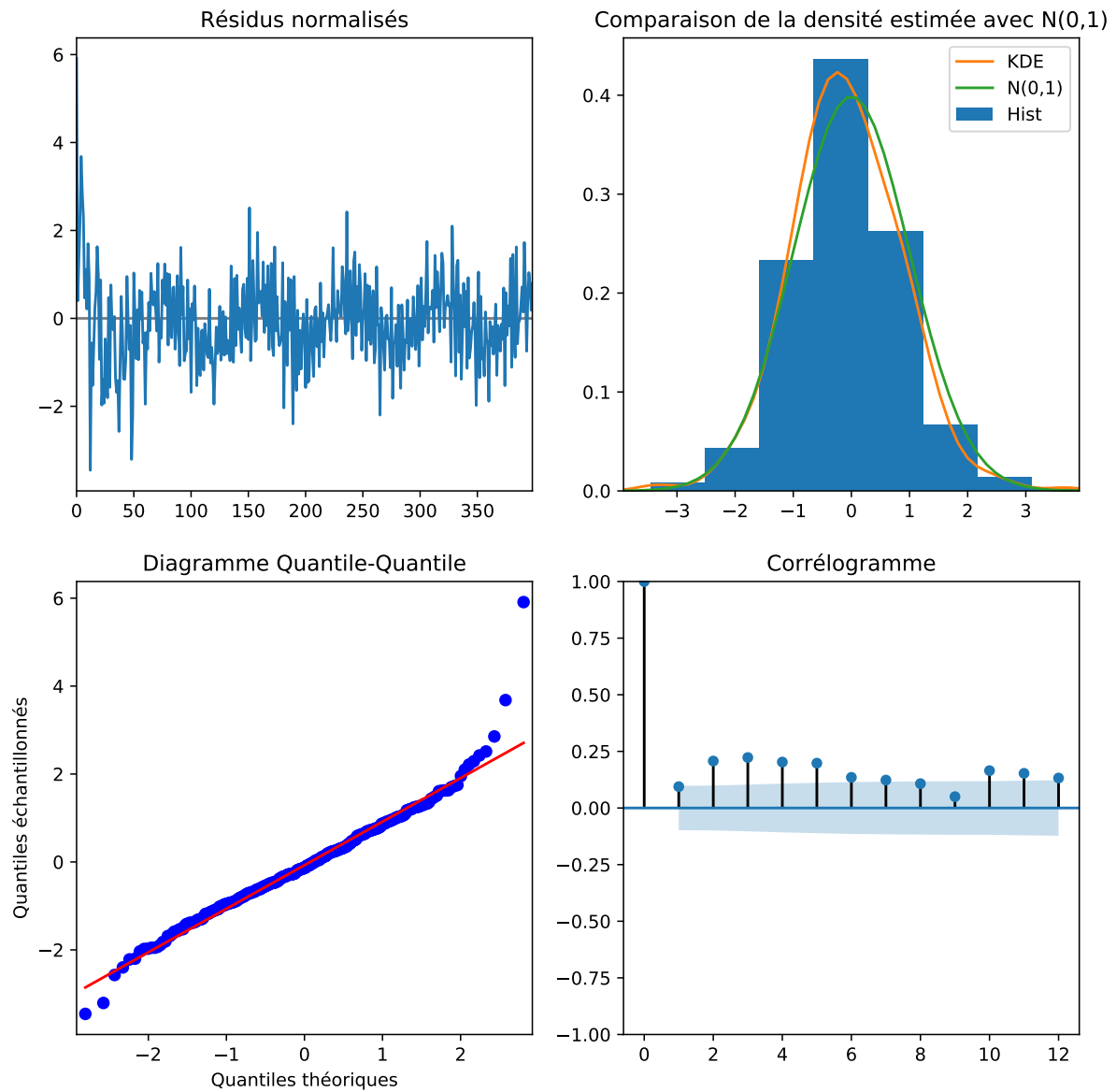
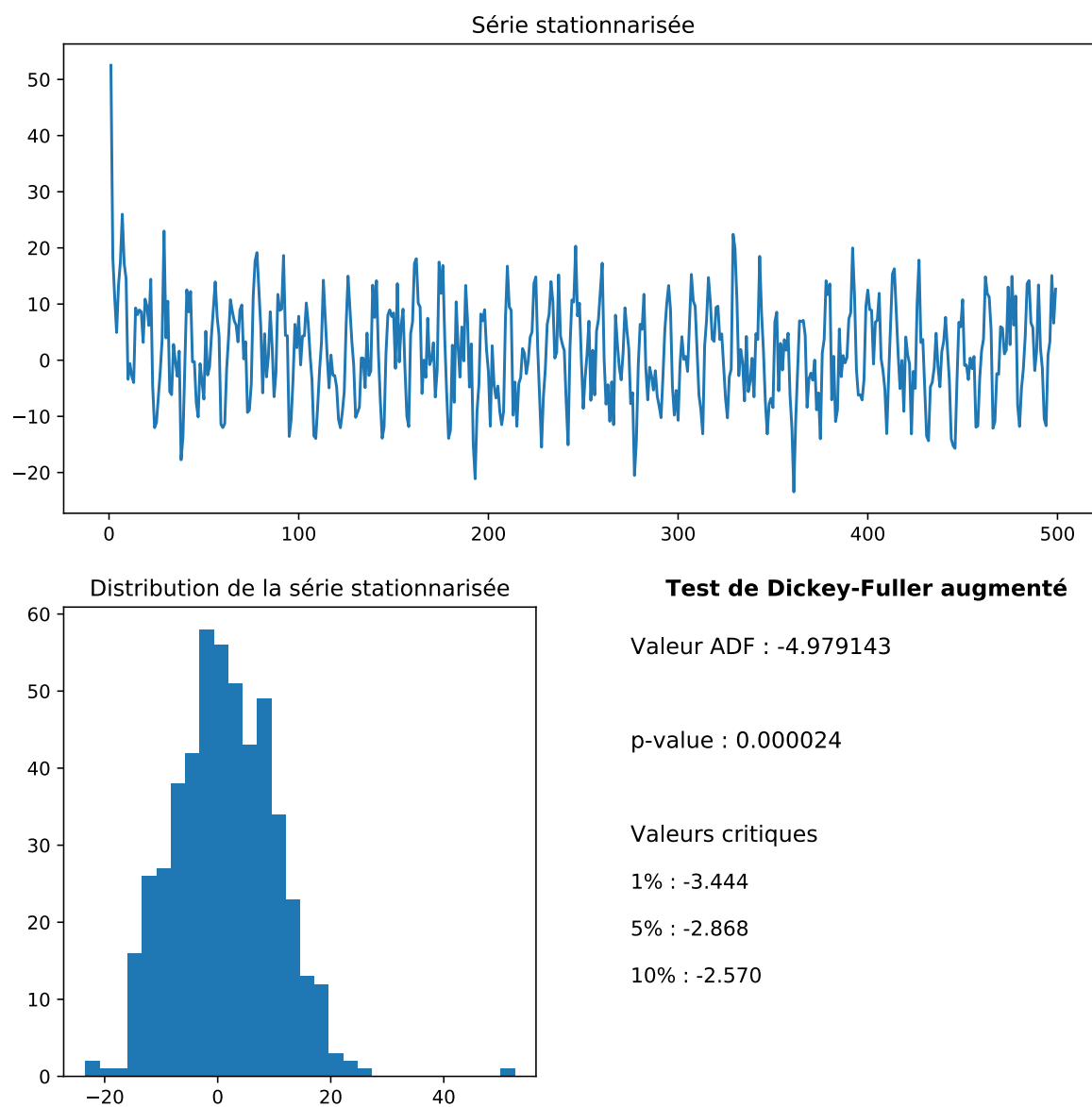


FIGURE D.21 – Test de Dickey-Fuller augmenté, série 4



Nombre de couches	Taille entrée	Nombre de neurones	Activation	Dropout	MSE
1	50	200	tanh	0.0	14.984
1	20	50	tanh	0.2	15.849
1	50	200	tanh	0.2	15.977

TABLE D.8 – Liste des modèles LSTM testés sur la série 4

FIGURE D.22 – Évolution de la perte du LSTM, série 4

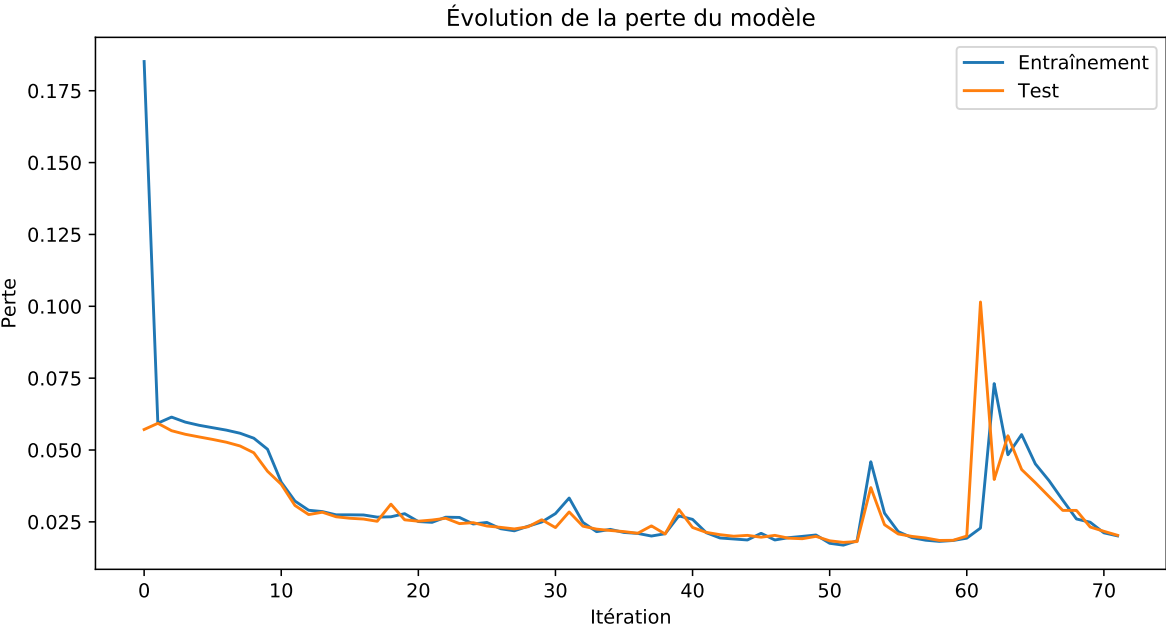


FIGURE D.23 – Prévisions one step ahead, série 4

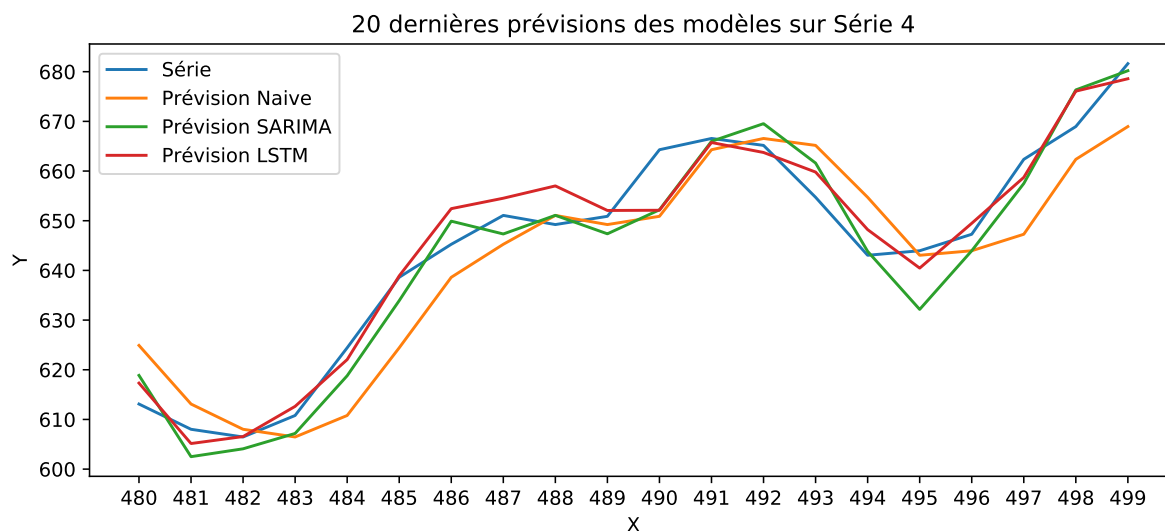
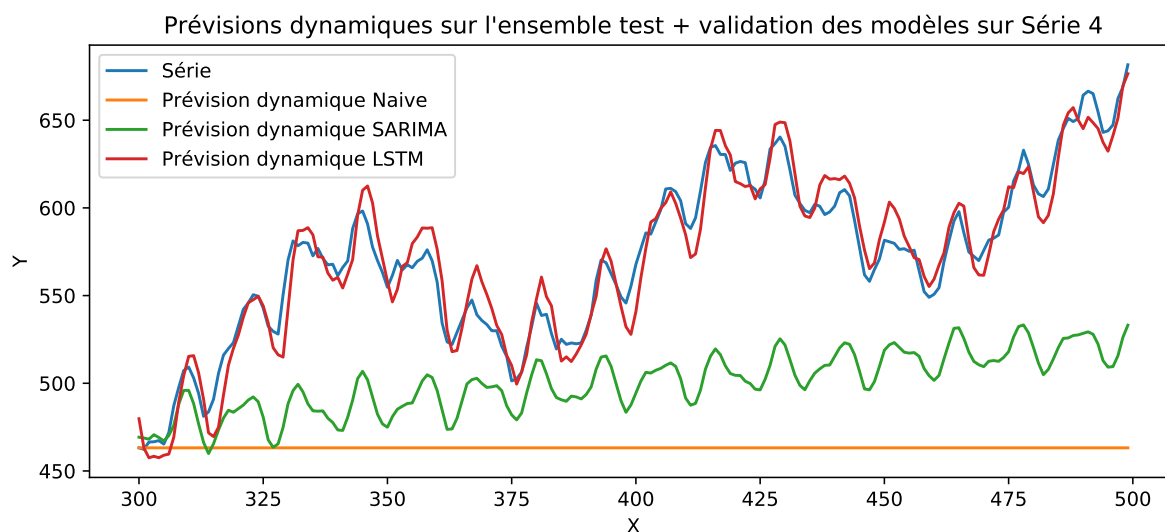


FIGURE D.24 – Prévisions dynamiques, série 4



CODE D.8 – Mesure de l'erreur, série 4

```

===      Série 4      ===
| Observations : 500
|
|   — Naive —
|
|   — Prévvision one step ahead —
|   MSE      : 69.84316296113006
|   MedAE    : 5.354359543003056
|   Var. expl : 0.9241401863030482
|   R^2      : 0.9223678181099569
|
|   — Prévvision dynamique —
|   MSE      : 21361.475115653026
|   MedAE    : 141.70690323216854
|   Var. expl : 0.0
|   R^2      : -22.74374027907245
|
|
|   — SARIMA —
|
|   — Prévvision one step ahead —
|   MSE      : 22.107576846314053
|   MedAE    : 3.174937607879201
|   Var. expl : 0.9755855735291901
|   R^2      : 0.9754269515566425
|
|   — Prévvision dynamique —
|   MSE      : 9468.687075888403
|   MedAE    : 96.2203680519062
|   Var. expl : 0.1873762591295567
|   R^2      : -9.524649889415251
|
|
|   — LSTM —
|
|   — Prévvision one step ahead —
|   MSE      : 23.049041845814337
|   MedAE    : 3.207348710760357
|   Var. expl : 0.9744634998597812
|   R^2      : 0.9743804929057794
|
|   — Prévvision dynamique —
|   MSE      : 119.96984549027033
|   MedAE    : 8.349405467823374
|   Var. expl : 0.8666879141483612
|   R^2      : 0.8666509294316425
|

```

D.5 Série production de bières en Australie

FIGURE D.25 – Subdivision de la série production de bière en set d'entraînement, test et validation

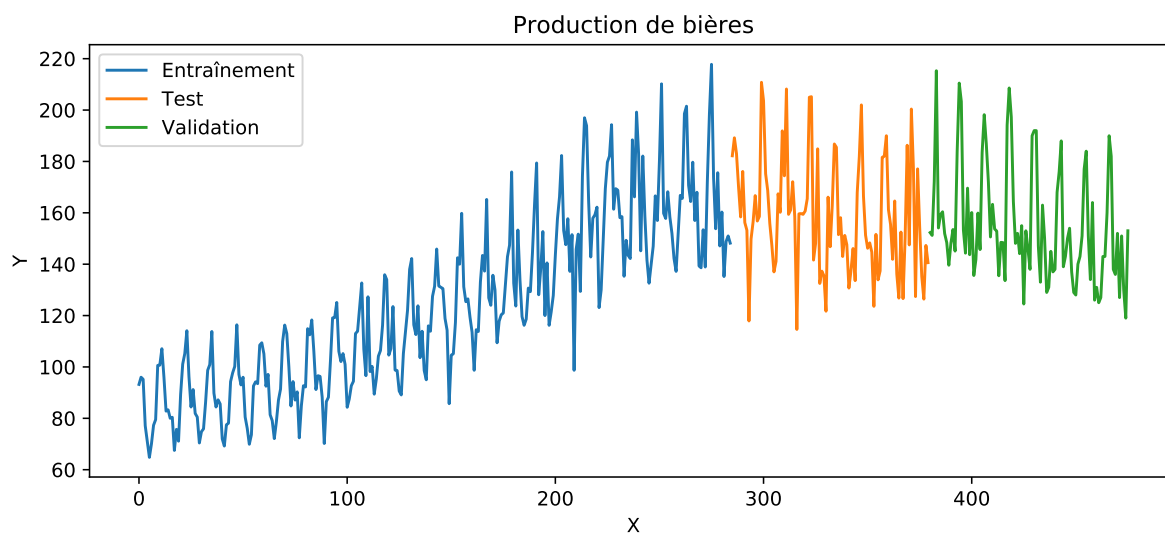


TABLE D.9 – Liste des modèles SARIMA testés sur la série production de bières

p	d	q	P	D	Q	s	AIC	BIC
1	1	1	2	0	1	22	3174.056	3197.682
0	1	0	0	0	2	2	3307.628	3311.565
1	1	1	4	0	3	2	2926.713	2966.089
0	1	1	4	0	3	2	2934.283	2969.721

CODE D.9 – Description du modèle SARIMA, série production de bieres

Nombre d'observations : 380				AIC :	2926.713	
Modele retenu : SARIMA(1, 1, 1)x(4, 0, 3, 2)				BIC :	2966.089	
Log vraisemblance : -1453.357				HQIC :	2942.339	
	Coefficient	Ecart-type	z	P> z	[0.025	0.975]
ar.L1	-0.1223	0.047	-2.586	0.010	-0.215	-0.030
ma.L1	-0.8204	0.032	-25.347	0.000	-0.884	-0.757
ar.S.L2	-0.2033	0.058	-3.513	0.000	-0.317	-0.090
ar.S.L4	-0.0020	0.009	-0.226	0.821	-0.020	0.016
ar.S.L6	-0.9944	0.008	-117.013	0.000	-1.011	-0.978
ar.S.L8	-0.2104	0.053	-3.941	0.000	-0.315	-0.106
ma.S.L2	-0.0684	0.176	-0.389	0.697	-0.412	0.276
ma.S.L4	-0.0028	0.162	-0.017	0.986	-0.321	0.315
ma.S.L6	0.9667	0.313	3.090	0.002	0.354	1.580
sigma2	88.8995	28.215	3.151	0.002	33.599	144.200
Ljung-Box (Q):		311.11		Jarque-Bera (JB):		12.85
Prob(Q):		0.00		Prob(JB):		0.00
Heteroskedasticity (H):		4.21		Skewness:		-0.33
Prob(H) (two-sided):		0.00		Kurtosis:		3.62

FIGURE D.26 – Diagnostic des résidus du modèle SARIMA pour la série production de bières

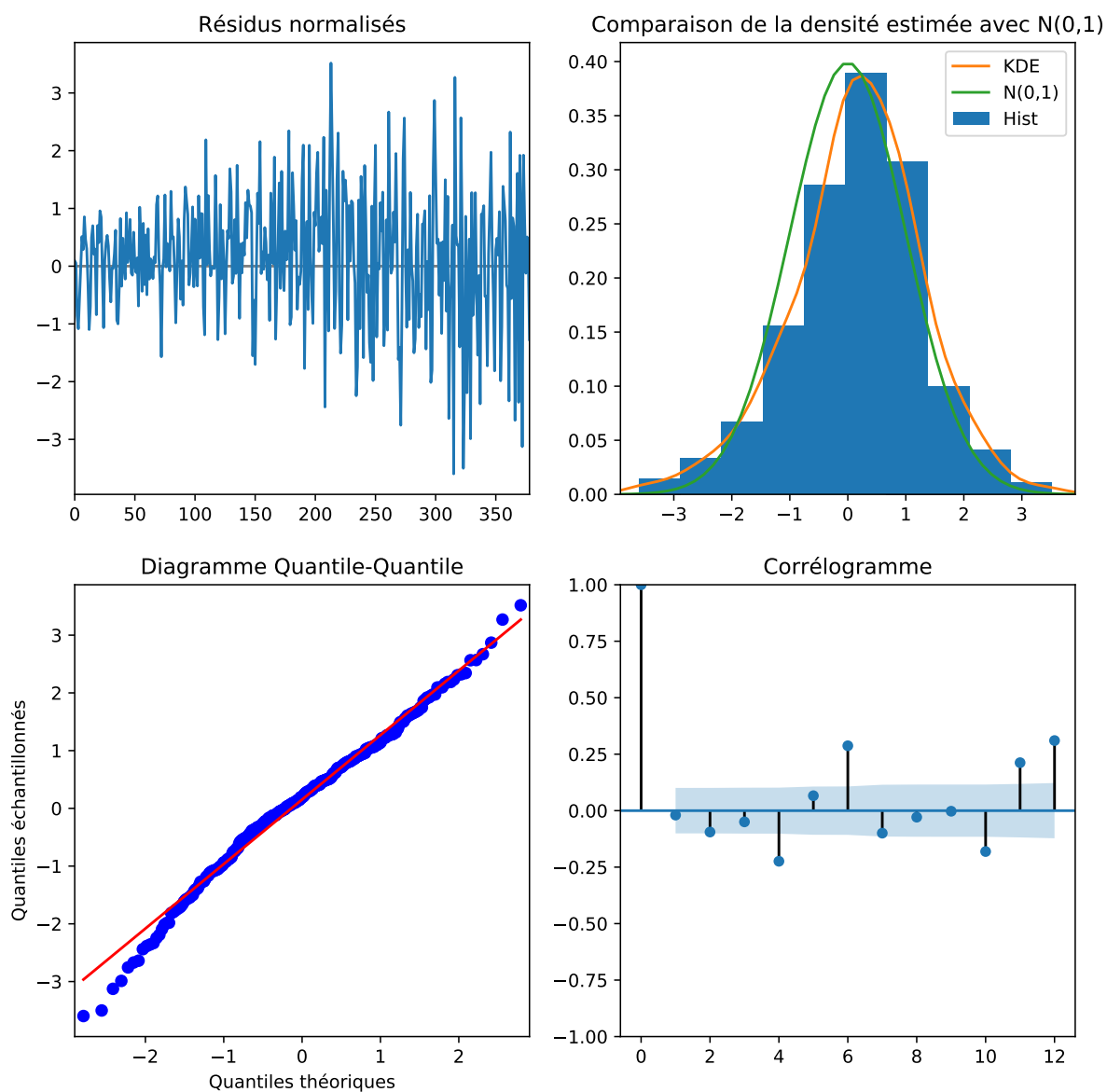
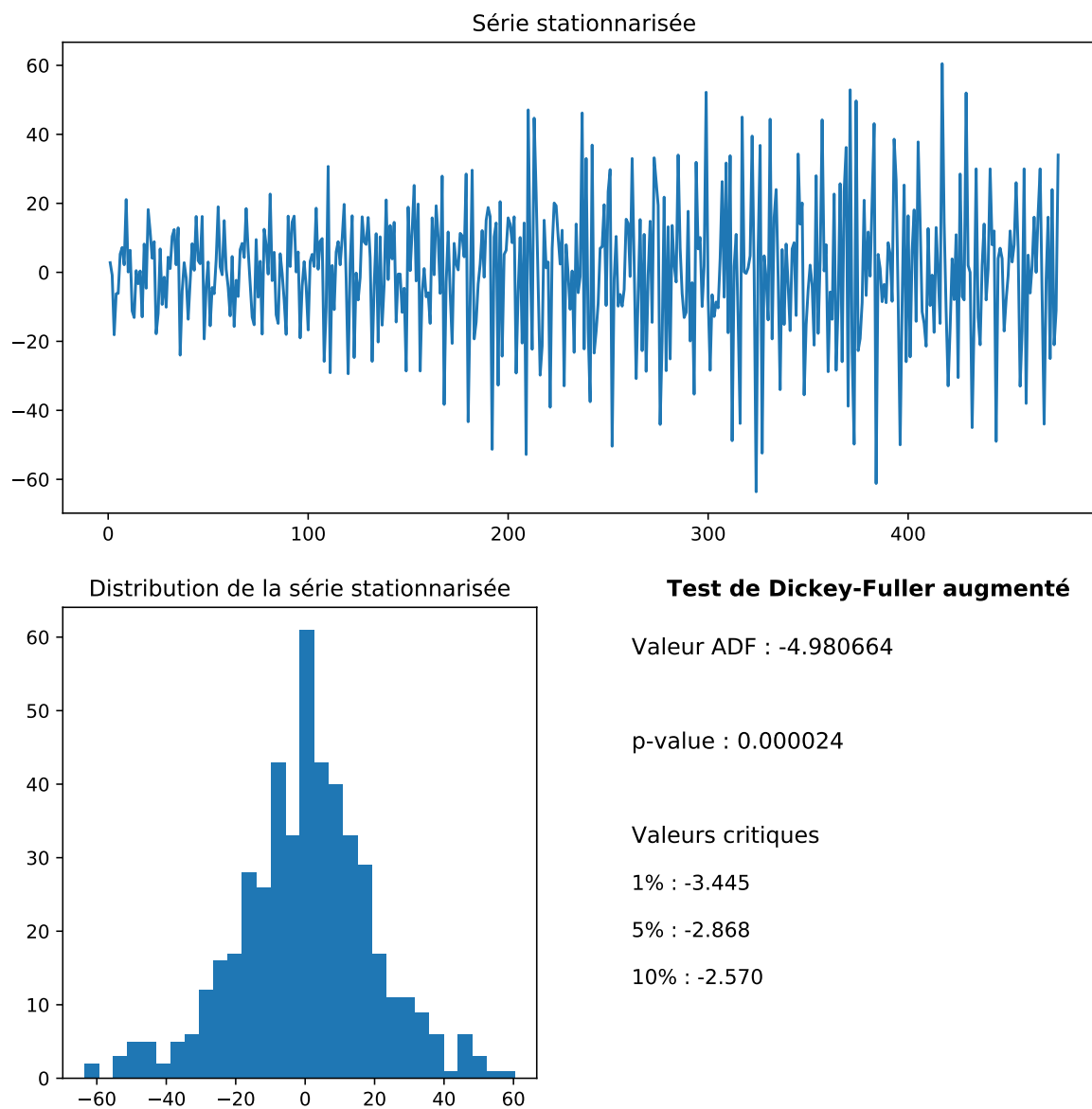


FIGURE D.27 – Test de Dickey-Fuller augmenté, série production de bières



Nombre de couches	Taille entrée	Nombre de neurones	Activation	Dropout	MSE
2	20	10	tanh	0.0	233.003
1	20	10	tanh	0.0	235.093
1	10	50	tanh	0.2	263.435

TABLE D.10 – Liste des modèles LSTM testés sur la série production de bières

FIGURE D.28 – Évolution de la perte du LSTM, série production de bières

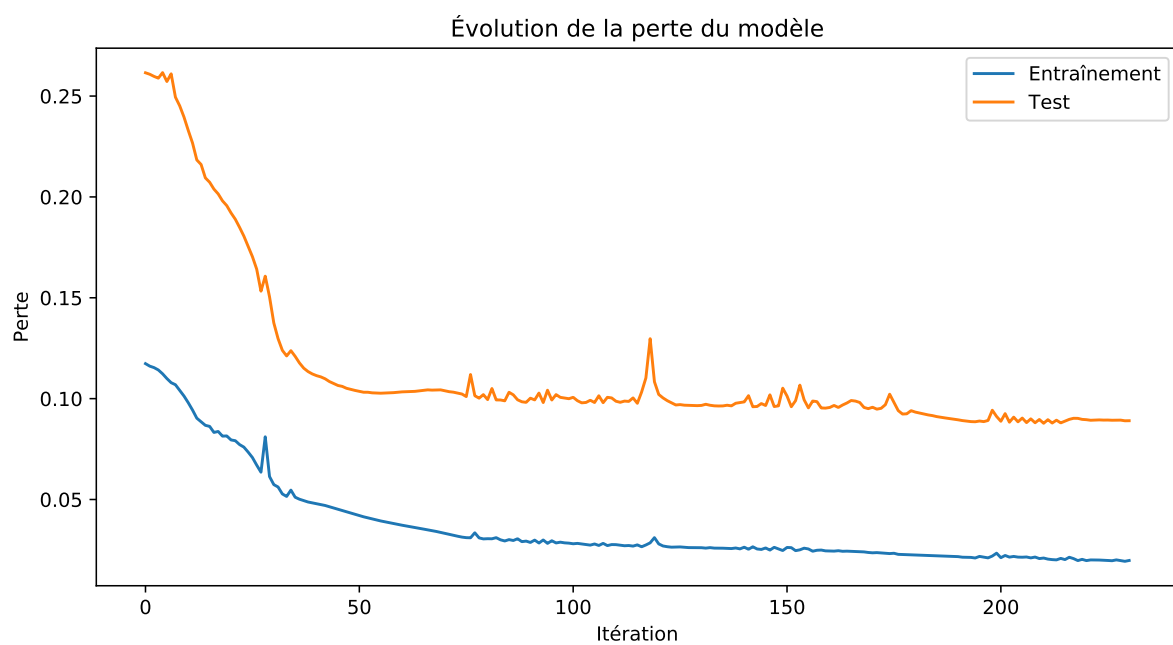


FIGURE D.29 – Prévisions one step ahead, série production de bières

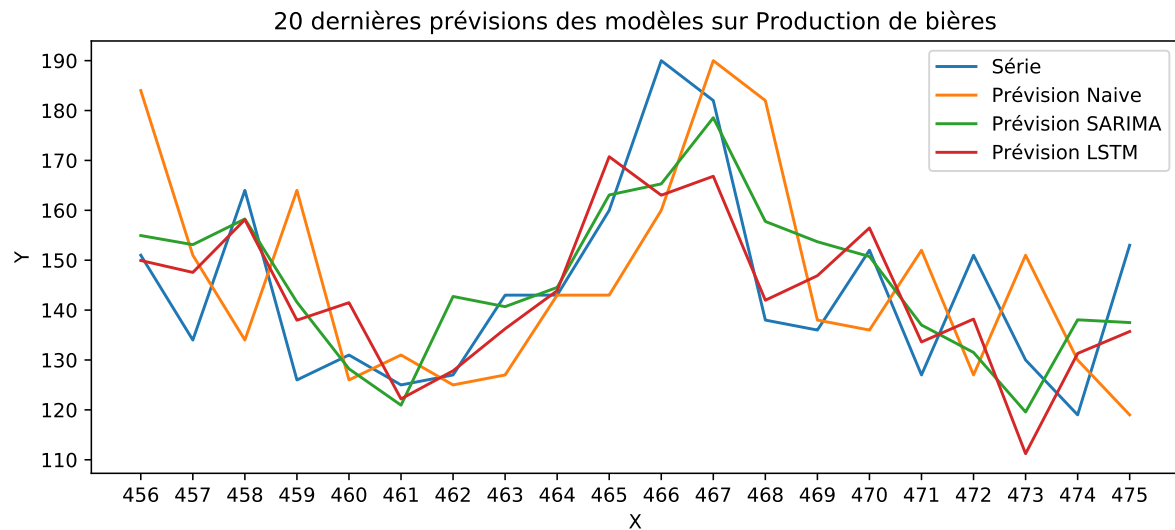
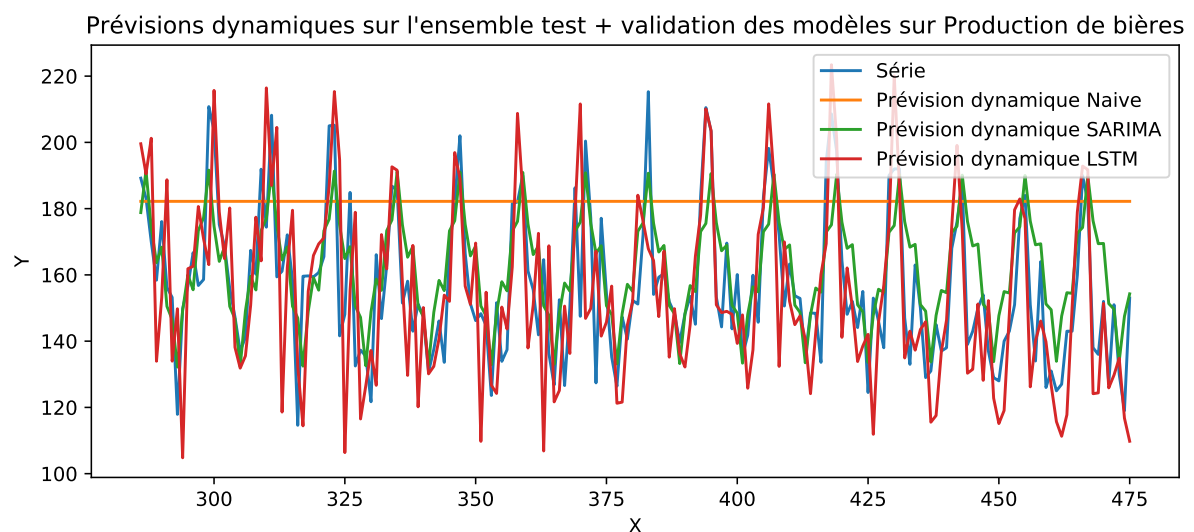


FIGURE D.30 – Prévisions dynamiques, série production de bières



CODE D.10 – Mesure de l'erreur, série production de bières

```

===      Production de bieres      ===
| Observations : 476
|
|      — Naive —
|
|      — Prévision one step ahead —
|      MSE      : 488.8520833333334
|      MedAE     : 13.0
|      Var. expl : -0.020288536404372115
|      R^2       : -0.020323359009909225
|
|      — Prévision dynamique —
|      MSE      : 1212.6782291666661
|      MedAE     : 31.399999999999999
|      Var. expl : 0.0
|      R^2       : -1.5310803950033036
|
|
|      — SARIMA —
|
|      — Prévision one step ahead —
|      MSE      : 169.14896324115844
|      MedAE     : 9.357012022425764
|      Var. expl : 0.6518080971954109
|      R^2       : 0.6469552974583912
|
|      — Prévision dynamique —
|      MSE      : 263.52955575093523
|      MedAE     : 10.16081923542292
|      Var. expl : 0.5470855146397722
|      R^2       : 0.4499658062440155
|
|
|      — LSTM —
|
|      — Prévision one step ahead —
|      MSE      : 186.2130683140181
|      MedAE     : 8.879308819770813
|      Var. expl : 0.6171539865617155
|      R^2       : 0.6113394013621354
|
|      — Prévision dynamique —
|      MSE      : 285.8776892047899
|      MedAE     : 11.751973288506278
|      Var. expl : 0.42995794346251937
|      R^2       : 0.4033211802504908
|

```

D.6 Série rendement du Bitcoin

FIGURE D.31 – Subdivision de la série rendement du Bitcoin en set d'entraînement, test et validation

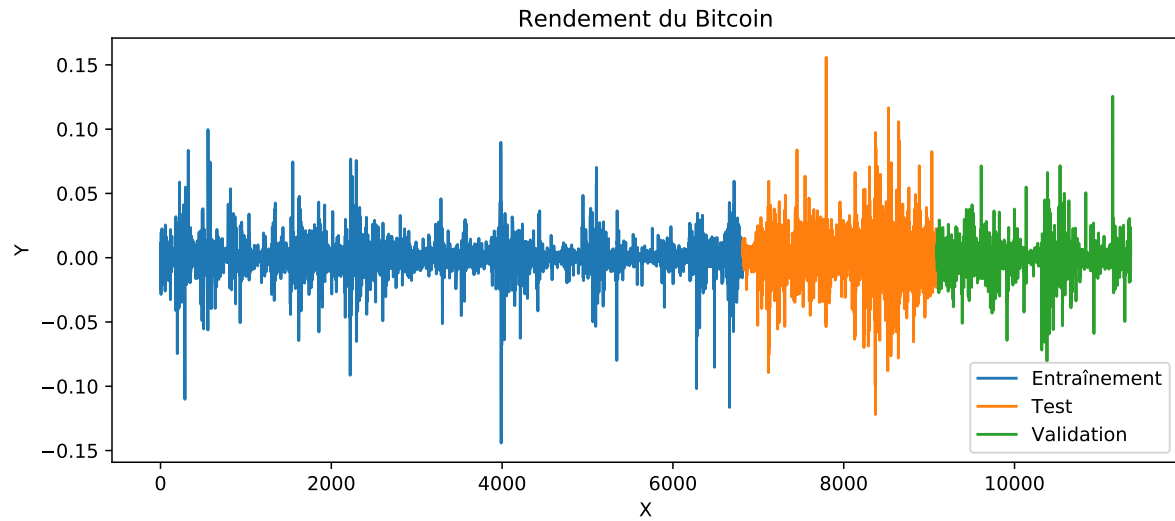


TABLE D.11 – Liste des modèles SARIMA testés sur la série rendement du Bitcoin

p	d	q	P	D	Q	s	AIC	BIC
2	1	2	2	0	1	3	-52418.292	-52361.375
0	1	0	0	0	0	3	-47755.833	-47748.718
1	1	2	2	0	0	3	-52474.891	-52432.203
0	1	2	2	0	0	3	-52468.238	-52432.665

CODE D.11 – Description du modèle SARIMA, série rendement du Bitcoin

Nombre d'observations : 9089			AIC : -52474.891			
Modele retenu : SARIMA(1, 1, 2)x(2, 0, 0, 3)			BIC : -52432.203			
Log vraisemblance : 26243.446			HQIC : -52460.372			
	Coefficient	Ecart-type	z	P> z	[0.025	0.975]
ar.L1	-0.1264	0.023	-5.384	0.000	-0.172	-0.080
ma.L1	-0.6620	0.023	-28.418	0.000	-0.708	-0.616
ma.L2	-0.3378	0.022	-15.174	0.000	-0.381	-0.294
ar.S.L3	0.0022	0.005	0.402	0.687	-0.009	0.013
ar.S.L6	-0.0410	0.006	-7.024	0.000	-0.052	-0.030
sigma2	0.0002	1.75e-06	103.380	0.000	0.000	0.000
Ljung-Box (Q):			169.53	Jarque-Bera (JB):		
Prob(Q):			0.00	Prob(JB):		
Heteroskedasticity (H):			2.14	Skewness:		
Prob(H) (two-sided):			0.00	Kurtosis:		
				64662.27		
				0.00		
				-0.14		
				16.06		

FIGURE D.32 – Diagnostic des résidus du modèle SARIMA pour la série rendement du Bitcoin

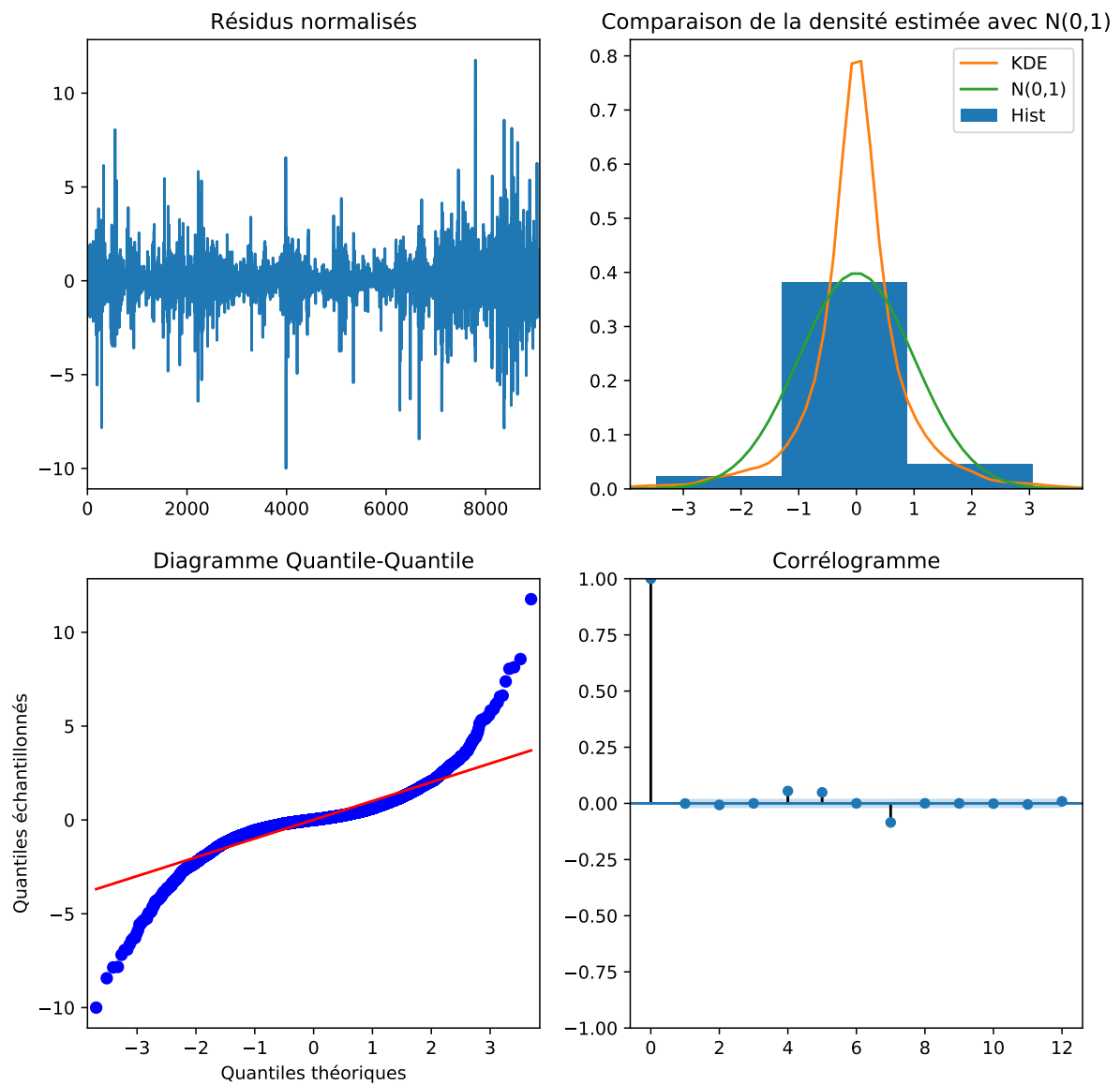
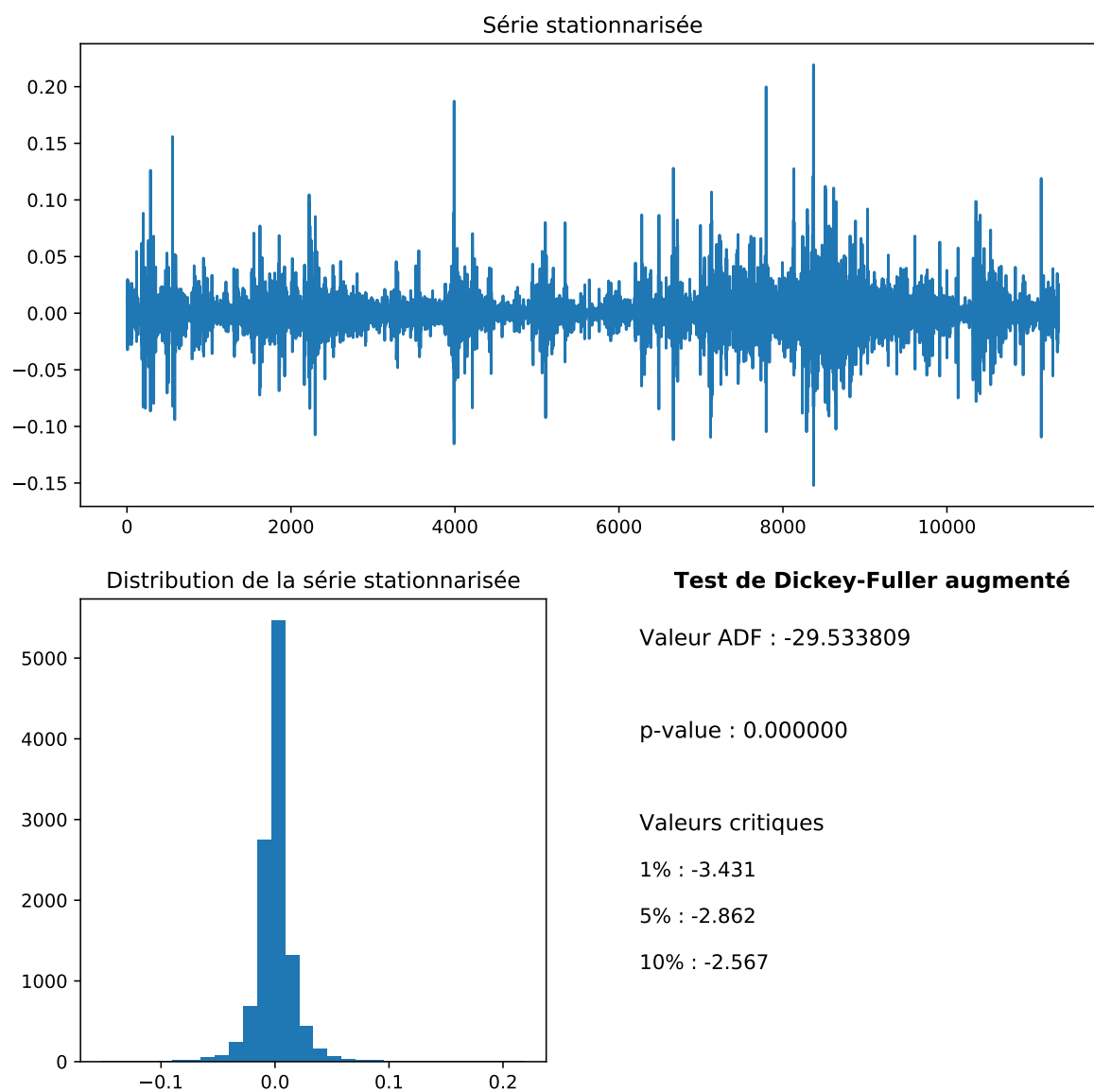


FIGURE D.33 – Test de Dickey-Fuller augmenté, série rendement du Bitcoin



Nombre de couches	Taille entrée	Nombre de neurones	Activation	Dropout	MSE
1	20	50	sigmoïde	0.0	0.000376
1	20	10	tanh	0.0	0.000378
1	10	50	tanh	0.2	0.000379

TABLE D.12 – Liste des modèles LSTM testés sur la série rendement du Bitcoin

FIGURE D.34 – Évolution de la perte du LSTM, série rendement du Bitcoin

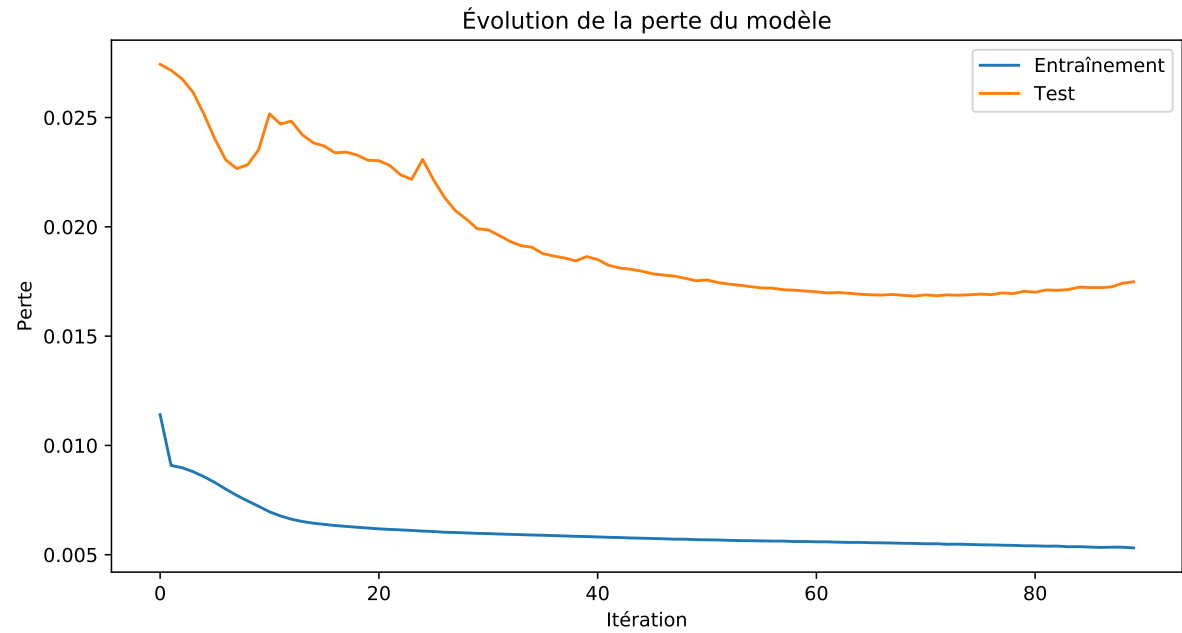


FIGURE D.35 – Prévisions one step ahead, série rendement du Bitcoin

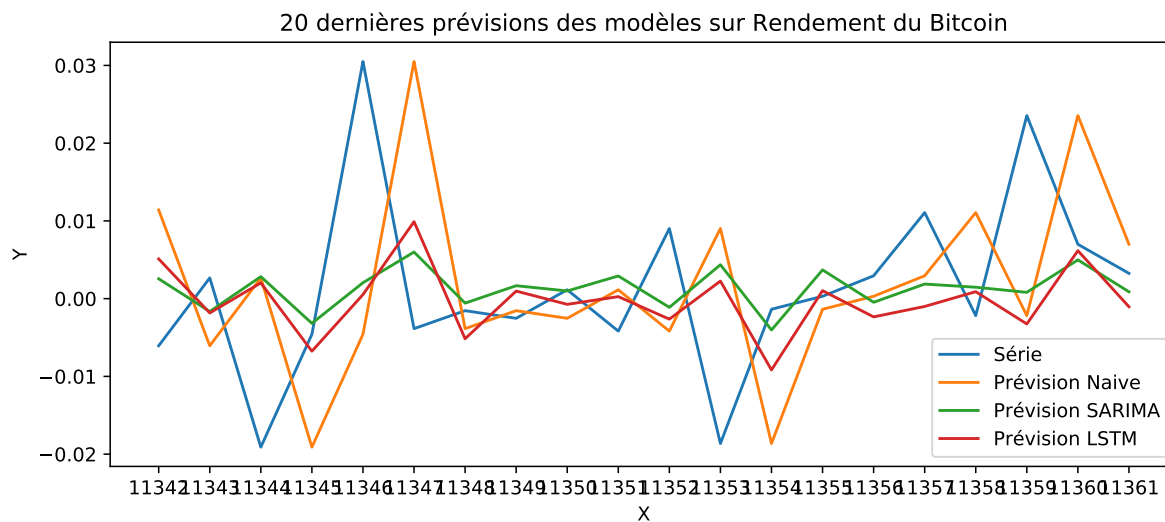
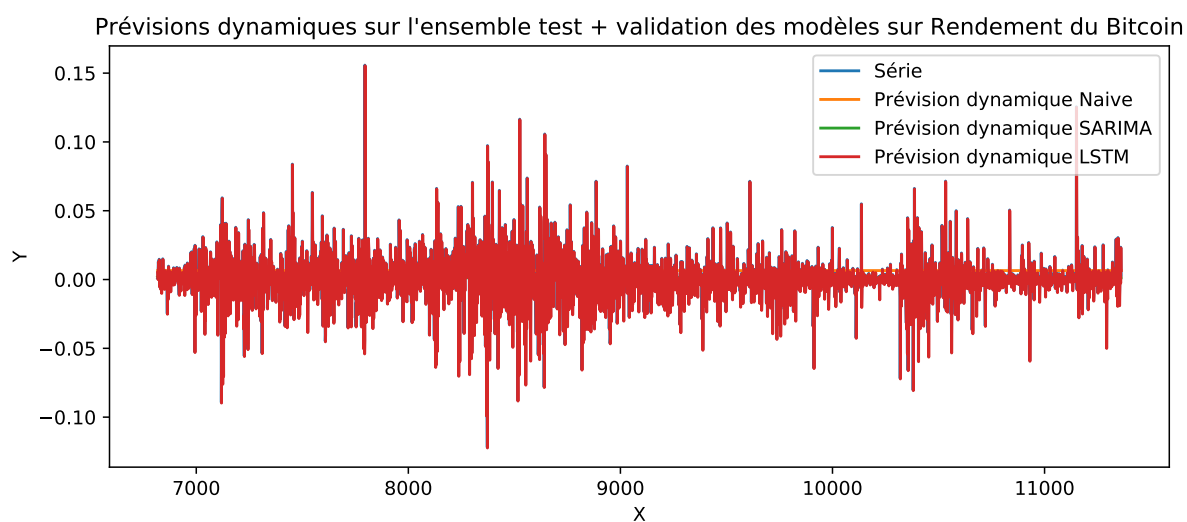


FIGURE D.36 – Prévisions dynamiques, série rendement du Bitcoin



CODE D.12 – Mesure de l'erreur, série rendement du Bitcoin

```

=== Rendement du Bitcoin ===
| Observations : 11362
|
| — Naive —
|
| — Prévvision one step ahead —
| MSE      : 0.00019685270495658293
| MedAE    : 0.005253527758245502
| Var. expl : -0.6044990144656195
| R^2      : -0.6044990314048535
|
| — Prévvision dynamique —
| MSE      : 0.0001660512222366062
| MedAE    : 0.007015889636088457
| Var. expl : 0.0
| R^2      : -0.35344355720684306
|
| — SARIMA —
|
| — Prévvision one step ahead —
| MSE      : 0.00012024285825701486
| MedAE    : 0.0036554372243916685
| Var. expl : 0.019929619950275312
| R^2      : 0.019929395182640586
|
| — Prévvision dynamique —
| MSE      : 0.00012444896958279056
| MedAE    : 0.00357265841300656
| Var. expl : 0.0
| R^2      : -0.014353606159284382
|
| — LSTM —
|
| — Prévvision one step ahead —
| MSE      : 0.00013451861503458495
| MedAE    : 0.004687826494121104
| Var. expl : -0.06006487073953015
| R^2      : -0.09642886327885436
|
| — Prévvision dynamique —
| MSE      : 0.0001970586574260741
| MedAE    : 0.005237598639296704
| Var. expl : -0.6044990139917952
| R^2      : -0.6061777004273945

```

Bibliographie

- BOURBONNAIS, Régis (2003). *Econométrie*. Dunod.
- BOX, George EP et al. (1970). *Time series analysis : forecasting and control*. John Wiley & Sons.
- CHARPENTIER, Arthur (2012). *Modèles de prévision - Séries temporelles*. Freakonometrics.
- CHO, Kyunghyun et al. (2014). « Learning phrase representations using RNN encoder decoder for statistical machine translation ». Dans : *arXiv preprint arXiv :1406.1078*.
- CHUNG, Junyoung et al. (2014). « Empirical evaluation of gated recurrent neural networks on sequence modeling ». Dans : *arXiv preprint arXiv :1412.3555*.
- DOZAT, Timothy (2016). *Incorporating nesterov momentum into adam*.
- ELMASDOTTER, Ajla et Carl NYSTRÖMER (2018). *A comparative study between LSTM and ARIMA for sales forecasting in retail*.
- ENGLE, Robert F et Clive WJ GRANGER (1987). « Co-integration and error correction : representation, estimation, and testing ». Dans : *Econometrica : journal of the Econometric Society*, p. 251–276.
- GREFF, Klaus et al. (2016). « LSTM : A search space odyssey ». Dans : *IEEE transactions on neural networks and learning systems* 28.10, p. 2222–2232.
- HOCHREITER, Sepp et Jürgen SCHMIDHUBER (1997). « Long short-term memory ». Dans : *Neural computation* 9.8, p. 1735–1780.
- HOPFIELD, John J (1982). « Neural networks and physical systems with emergent collective computational abilities ». Dans : *Proceedings of the national academy of sciences* 79.8, p. 2554–2558.
- HYNDMAN, Rob J, Yeasmin KHANDAKAR et al. (2007). *Automatic time series for forecasting : the forecast package for R*. 6/07. Monash University, Department of Econometrics et Business Statistics ...
- LE, Quoc V et al. (2011). « On optimization methods for deep learning ». Dans : *Proceedings of the 28th International Conference on International Conference on Machine Learning*. Omnipress, p. 265–272.
- OLDEN, Julian (2004). « An accurate comparison of methods for quantifying variable importance in artificial neural networks using simulated data ». Dans : *Ecological Modelling* 178. DOI : 10 . 1016/S0304-3800(04)00156-5.
- ROSENBLATT, Frank (1958). « The perceptron : a probabilistic model for information storage and organization in the brain. » Dans : *Psychological review* 65.6, p. 386.
- RUDER, Sebastian (2016). « An overview of gradient descent optimization algorithms ». Dans : *arXiv preprint arXiv :1609.04747*.
- RUMELHART, David E, Geoffrey E HINTON, Ronald J WILLIAMS et al. (1988). « Learning representations by back-propagating errors ». Dans : *Cognitive modeling* 5.3, p. 1.
- SPYROS MAKRIDAKIS Evangelos Spiliotis, Vassilios Assimakopoulos (2018). « Statistical and Machine Learning forecasting methods : Concerns and ways forward ». Dans : *Universidad Veracruzana, MEXICO*.
- WHITTLE, Peter (1951). *Hypothesis testing in time series analysis*. T. 4. Almqvist & Wiksells.