PLDAC

Classification de séries temporelles avec des modèles de réseaux de neurones profonds

PERGAMENT Melissa, REGUIG Ghiles
Encadrant : GALLINARI Patrick

May 26, 2017

Contents

Abstract

Les réseaux de neurones sous l'appellation *Deep Learning* représentent l'état de l'art dans de nombreux domaines de l'intelligence artificielle comme la reconnaissance d'objets dans les images, de scènes sur des vidéos, la traduction automatique, etc.
Les objectifs du projet étaient

1. découvrir un ensemble de méthodes du Deep Learning.
2. les appliquer au problème de la classification de séries temporelles en s'appuyant sur une des plateformes logicielles disponibles.
3. réaliser une implémentation permettant de tester automatiquement un ensemble d'architectures de réseaux de neurones pour sélectionner les plus performantes pour un problème de classification donné.

Nous nous sommes donc intéressés aux différentes architectures (*MLP*, *CNN*, *RNN*) et hyperparamètres (nombre de couches, nombre de neurones par couche, paramètres de régularisation,...) de ces classifieurs pour la classification de séries temporelles. On étudie l'impact, aussi bien en terme de convergence que de précision de chaque variation d'hyperparamètres afin d'en déduire les modèles les plus efficaces dans le traitement des séries temporelles.
Enfin, nous proposons une implémentation logicielle permettant la sélection automatique d'un modèle de réseau de neurones via un *GridSearch* selon les meilleures performances sur les données considérées.

1 Introduction

En raison de leur application dans de nombreux domaines tels que la médecine, l'astronomie, la météorologie ou encore l'économie, les séries temporelles représentent un domaine d'étude à part entière.
Dans ce projet, nous nous intéressons à la classification de telles données en ayant recours à des réseaux de neurones profonds. Nous commencerons, dans un premier temps, par introduire un ensemble de définitions relatives aux séries temporelles ainsi qu'aux réseaux de neurones profonds, ensuite, dans un second temps, nous nous intéresserons aux performances de différents types de réseaux de neurones puis dans un troisième temps, nous proposerons une implémentation logicielle permettant d'automatiser la sélection d'un modèle de réseaux de neurones en fonction des données à traiter.

2 Séries temporelles

2.1 Définition

On appelle série temporelle, ou encore série chronologique, une suite finie de valeurs représentant une évolution d'une ou plusieurs quantités en fonction du temps.
Soit x_t la valeur de la variable x au temps t. On appelle série temporelle de longueur n une suite de valeurs telle que : $(x_1, x_2, \ldots, x_n)$.
L'intervalle de discrétisation (différence temporelle entre le temps $n-1$ et n) peut être, selon les cas, la seconde, la minute, l'heure, le jour,...
Une série temporelle est dite univariée lorsqu'elle n'est décrite que par une seule variable. Elle est dite multivariée lorsque plusieurs variables la décrivent, dans ce cas x_t est un vecteur représentant les valeurs de chaque variable au temps t. Dans la suite, pour simplifier et parce que c'est le choix classiquement effectué dans les benchmarks sur les séries temporelles, nous supposerons que toutes les séries d'un même problème sont de taille fixe. Cela nous permettra également de les considérer comme des vecteurs pour certaines architectures de réseaux de neurones.

2.2 Etude de séries temporelles

A l'instar des autres types de séries, les séries temporelles sont utilisées aussi bien pour la classification que pour la prédiction. La caractéristique fondamentale de ces données est leur caractére séquentiel : la valeur de x au temps t dépend de ses valeurs aux temps précédents.
De nombreux modèles sont utilisés dans l'analyse de séries temporelles pour les tâches de prédiction et de classification.
En prédiction, les modèles classiques se basent en géneral sur des processus *autorégressifs* (AR) et

de moyennes mobiles (*Moving Average*).
En classification, parmi les modèles les plus utilisés, on trouve les modèles markoviens, génératifs dans le cas des *HMM* ou discriminants dans le cas des *CRF*, qui se basent sur une hypothèse markovienne d'ordre n, c'est-à-dire que la valeur x_t dépend de $x_{t-1}, ..., x_{t-(n-1)}$.
D'autres types de modèles sont également utilisés, que ce soit pour la classification ou la prédiction : les *réseaux de neurones* sous la forme de *réseaux convolutifs* (*Convolutional Neural Networks*) ou *récurrents* (*Recurrent Neural Networks*). Dans le cadre de ce projet, nous nous intéressons aux performances de ces modèles en terme de classification.

3 Réseaux de neurones profonds et classification[1]

3.1 Réseau de neurones[2]

Un réseau de neurones est, dans notre cas, un classifieur inspiré du fonctionnement du cerveau humain. Il est composé de neurones, disposés sur plusieurs couches et liés entre eux selon divers schémas d'interconnexion, dénommés architecture du réseau, dont quelques unes seront décrites par la suite.
Dans le cas d'un classifieur, un neurone formel sera une cellule appliquant, à une entrée X une fonction linéaire puis une fonction non linéaire d'activation.

3.1.1 Neurone

Un neurone prend comme paramètre d'entrée un vecteur $X \in \mathbf{R}^d$, puis lui applique en premier lieu une fonction linéaire f paramétrée par un vecteur $W \in \mathbf{R}^{d+1}$ (la dimension supplémentaire correspond à un biais).

$$f : \mathbf{R}^d \to \mathbf{R}$$

$$f(x) = w_0 + \sum_{i=1}^{d} w_i \cdot x_i$$

On applique ensuite une nouvelle fonction g à ce résultat dite fonction d'activation. Cette dernière est généralement non linéaire.

$$g : \mathbf{R} \to \mathbf{R}$$

On obtient comme fonction de transfert de notre neurone une composée des fonctions f et g telle que

$$(g \circ f)(x)$$

Les fonctions d'activation usuelles sont :

- la tangente hyperbolique $g(x) = \frac{2}{1+\exp(-2x)} - 1$
- la sigmoïde $g(x) = \frac{1}{1+\exp(-x)}$
- la rectification linéaire $(ReLU)$ $g(x) = \max(0, x)$

Cette fonction, non linéaire, représente le traitement fait par la plus petite unité d'un réseau de neurones. Il reste toutefois à mettre en relation les différentes cellules au sein d'une architecture adaptée à la classification.

3.1.2 Architecture d'un réseau de neurones

Comme son nom l'indique, un réseau de neurones est un ensemble de neurones comme décrits ci-dessus liés entre eux par des connexions qui permettent à un neurone de recevoir la sortie d'un autre neurone afin de la traiter à son tour.
On définit une couche comme étant l'ensemble des neurones d'un même niveau recevant le même vecteur en paramètre d'entrée.
Une des difficultés du traitement de données avec des réseaux de neurones est de déterminer l'architecture la plus efficace. Dans le cadre de ce projet, nous étudierons trois architectures différentes.

Perceptron Multi Couches

Un perceptron est le nom donné au classifieur binaire représenté par un unique neurone. Un perceptron multi couches (ou *MLP* en anglais pour *Multi Layer Perceptron*) est un classifieur constitué de plusieurs couches ordonnées de perceptrons où tous les neurones d'une certaine couche i sont connectés à tous les neurones de la couche suivante $i + 1$. Ainsi, les sorties de la couche i sont les entrées de la couche $i + 1$. Un exemple de MLP est décrit par la figure 1.
Théoriquement, un perceptron multi couches à une couche cachée tel que dans la figure 1, peut approcher des classes de fonctions comme les fonctions continues sur un ensemble borné si l'on ne limite pas le nombre de perceptrons dans la couche cachée.

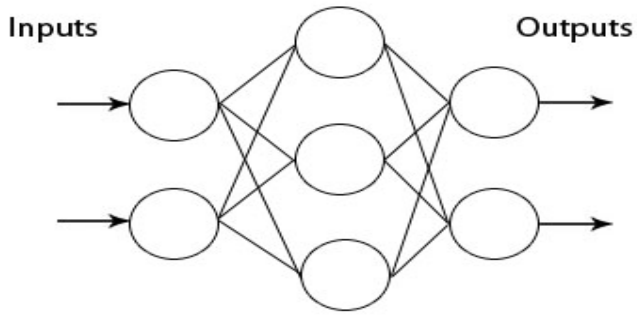

Figure 1: Architecture d'un MLP à une couche cachée.

Réseau neuronal convolutif [3]

Une seconde architecture de réseau de neurones à laquelle nous nous intéressons est celle des réseaux convolutifs. Ces derniers permettent de considérer une représentation "locale" des données. Pour les séries temporelles cette dimension correspond à l'ordre des observations de la série. Pour des images, cette dimension correspond à des caractéristiques spatiales.

Il existe deux types de couches spécifiques aux réseaux convolutifs, qui vont en général par paire : la couche de convolution et la couche de pooling. Dans notre cas, pour les séries temporelles, nous nous plaçons dans le cadre d'une convolution 1D.

Convolution la couche de convolution permet d'appliquer des filtres à l'ensemble des données. Ces derniers se présentent sous la forme de fonctions prenant comme paramètre une taille, qui représente le nombre de valeurs à traiter en même temps et un décalage (stride en anglais) qui indique de combien d'unités de temps on décale les filtres successifs. L'application d'un filtre à une série donne ensuite comme sortie une nouvelle série, dont les valeurs sont remplacées par celles données par l'application de la fonction filtre.
Chaque filtre permet de mettre en évidence une certaine caractéristique au sein des données en transformant la série de manière à accentuer les valeurs les plus représentatives de la caractéristique en question, selon le critère d'apprentissage choisi.
Une couche de convolution correspond à un ensemble de filtres appliqués à une série d'entrée, on obtient en sortie autant de séries images que de filtres utilisés dans la couche.
Les hyperparamètres d'une couche de convolution sont :

- le nombre de filtres.
- la taille de la fenêtre de convolution (*kernel size*), à savoir le nombre de valeurs consécutives prises en compte par chaque filtre.
- le pas de déplacement (*stride)* de la fenêtre de convolution.

Pooling la phase de pooling permet de synthétiser les informations extraites par le filtre de convolution en remplaçant un ensemble de valeurs par une seule et unique mesure basée sur ces dernières.
La fonction de pooling prend comme paramètres :

- la taille t de la fenêtre de pooling (*pool size*) qui correspond au nombre de valeurs que prend la cellule de pooling en entrée.
- le pas de déplacement (*stride*) de la fenêtre de pooling qui définit son glissement sur les valeurs de la série.

La fonction passe par chaque fenêtre de taille t et remplace ses valeurs par la sortie de la fonction de pooling. On considèrera en général du *MaxPooling*, qui remplace la fenêtre par sa valeur maximale, ou du *AveragePooling*, qui remplace la fenêtre par la moyenne de ses valeurs.

Réseau de neurones récurrent

Le dernier type de réseaux de neurones que l'on étudiera sont les réseaux récurrents. La particularité de ces derniers est qu'ils considérent des séries temporelles en supposant que la valeur d'une variable x au temps t dépend de toutes les valeurs de x aux temps antérieurs.
Il existe plusieurs types de réseaux récurrents couramment utilisés aujourd'hui.

- **Simple Recurrent Neural Network**
  Ce premier type de réseau récurrent utilise des cellules cachées dont l'entrée au temps t est la sortie de la même cellule au temps précédent $t-1$ et la valeur de la série au temps t. Quelle que soit l'itération, le vecteur de paramètres W de la cellule reste le même, une fois optimisé. Ce type de réseau permet notamment de palier au problème de variabilité des tailles de séquences.

- **Long Short-Term Memory**
  Les architectures de types *LSTM* sont utilisées afin de modéliser des dépendences à long terme, c'est-à-dire, lorsque le traitement au temps t nécessite une information contenue au temps $t-h$, avec h potentiellement grand.
  Pour ce faire, on introduit des *Portes* (*Gates*) sur la cellule récurrente de base, permettant de modifier le comportement du réseau à certains pas de temps. Les LSTM sont dotées de 3 portes :

  1. **Output Gate** permettant de récupérer la valeur de sortie de la cellule.
  2. **Forget Gate** permettant de pondérer la quantité d'information à utiliser d'un pas à l'autre, en fonction d'une entrée.
  3. **Input Gate** permettant d'ajouter des informations relatives au pas de temps. On peut, par exemple, remplacer les valeurs omises par la *Forget Gate*.

  Chaque porte possède ses propres paramètres (vecteur de poids, fonction(s) d'activation) à optimiser lors de l'apprentissage.

- **Gated Recurrent Unit[4]**
  Les *GRU* permettent, à l'instar des *LSTM*, un traitement dynamique des séquences. Pour ce faire, les *GRU* utilisent 2 portes :

  1. **Reset Gate** permettant de pondérer la quantité d'information à retenir du temps précédent.
  2. **Update Gate** permettant de sélectionner les informations à retenir de l'*Input Gate* et du temps précédent.

  Chaque porte possède ses propres paramétres (vecteur de poids, fonction(s) d'activation) à optimiser lors de l'apprentissage.

3.1.3 Hyperparamètres d'un réseau neuronal

Les réseaux de neurones sont des classifieurs très expressifs et peuvent, théoriquement, définir de larges classes de fonctions de décision. Le revers de la médaille de ce pouvoir d'expression est le grand nombre d'hyperparamètres à optimiser afin de trouver le classifieur le plus adapté au problème abordé.
Dans le cadre de notre projet, nous nous intéresserons à l'influence de la variation des hyperparamètres listés ci-dessous sur les performances de nos classifieurs.

Architecture du réseau Nous avons vu plus haut trois architectures différentes, on s'intéresse aux performances de chacune d'entre elles en faisant varier ses hyperparamètres.

Nombre de neurones/couches
Pour chaque type de réseau de neurones, on peut faire varier le nombre de neurones total ainsi que le nombre de couches.

Taille de la fenêtre
Dans le cas des réseaux convolutifs, on peut faire varier la taille de la fenêtre sur laquelle on applique la fonction de convolution.

Fonction de coût (*Loss*) Afin de pouvoir quantifier l'apprentissage de nos classifieurs, on se dote d'une fonction dite de coût (*Loss*) l qui permet, en apprentissage supervisé, de donner une métrique d'erreur sur les données sur lesquelles le modèle apprend. Le but final est de minimiser la valeur retournée par cette fonction.
Il n'est pas rare qu'un modèle ait une fonction de décision trop complexe vis-à-vis du problème ciblé. Afin de palier à ce problème, on utilise une fonction de régularisation, permettant de controler la complexité de la fonction de décision recherchée.
La forme générale d'une fonction de coût régularisée est :

$$R(f) = \sum_i l(f_w(x_i), y_i) + k\Omega(f)$$

avec :

- x_i un exemple de la base d'apprentissage.
- y_i classe associée à l'exemple x_i.
- $f_w(x_i)$ la classe prédite par le classifieur.
- l la fonction mesurant l'erreur faite sur un exemple entre une prédiction $f(x_i)$ et sa classe y_i.
- Ω le régulariseur.
- k le coefficient de régularisation permettant de pondérer Ω et l.

De cette formule découlent un certain nombre d'hyperparamètres à fixer. Dans le cadre de ce projet, on se focalise sur différents types de régularisations classiques et on utilise un coût entropique.
Nous considérons 4 types de régularisation.

L1 Cette régularisation utilise la norme L1 et a pour principale caractéristique d'éliminer les dimensions qui n'aident pas à la classification.

L2 Cette régularisation utilise la norme L2 et permet de minimiser les poids liés à des dimensions non utiles. Au contraire de la régularisation L1, ces dimensions sont toujours prises en compte et ne sont pas directement éliminées.

L1L2 Cette régularisation utilise les deux normes L1 et L2, telles que décrites ci-dessus.

Dropout[5] Cette régularisation permet d'éliminer avec une probabilité p les activités d'un neurone au sein d'une couche lors de l'apprentissage.
En test, on pondérera par p les paramètres des neurones pris en compte par le Dropout.

Batch Normalisation[6] Cette régularisation permet de normaliser (centrer et réduire) les sorties d'une couche. Elle requière de calculer, à chaque mini-batch d'apprentissage, la moyenne et la variance de l'ensemble et d'ensuite normaliser au niveau de la couche choisie.

Algorithme d'optimisation L'optimisation consiste à trouver les valeurs optimales des poids W liés à chaque couche permettant de minimiser la fonction de coût.

$$\arg\min_w \sum_i l(f_w(x_i), y_i) + k\Omega(f)$$

- *Descente de gradient stochastique*
  L'algorithme de descente de gradient stochastique (*Stochastic Gradient Descent*) effectue la mise à jour des poids par rapport à un exemple de la base d'apprentissage tiré au hasard. La mise à jour des poids est faite selon la formule

$$W_{t+1} = W_t - \eta \nabla R(f(x_i), y_i)$$

  - W_{t+1} les valeurs du vecteurs W à l'itération $t+1$ de l'algorithme d'apprentissage.
  - η *learning rate*, pas avec lequel les poids évoluent.
  - ∇ le gradient (dans notre cas, le gradient de la fonction objectif R).
  - R la fonction objectif.
  - f la fonction de classification du classifieur.

- x_i exemple tiré aléatoirement au sein de la base d'apprentissage.
- y_i classe de l'exemple x_i.

- *Adam*
  Un second algorithme d'optimisation que nous utilisons au sein de ce projet est *Adam*[7] (*Adaptive Moment Estimation*). Ce dernier a l'avantage de faire varier la valeur du *learning rate* au cours des itérations afin d'éviter une éventuelle stagnation et de l'adapter à chaque paramètre du réseau de neurones à chaque intération. Pour ce faire, la moyenne et la variance des gradients sont estimés et le *learning rate* est modifié en fonction de ces estimations. La formule de mise à jour des poids est la suivante :

$$W_{t+1} = W_t - \frac{\eta}{\sqrt[2]{\hat{v}_t} + \epsilon}\hat{m}_t$$

  - W_t valeur des poids a l'itération t.
  - η learning rate.
  - $\hat{v}$ variance des gradients non biaisée estimée.
  - ϵ biais permettant d'éviter une division par zéro (valeur très petite).
  - $\hat{m}$ moyenne des gradients non biaisée estimée.

3.1.4 Evaluation d'un modèle

Quel que soit le classifieur utilisé, il est nécessaire de pouvoir mesurer ses performances afin d'éviter les deux problèmes majeurs de la classification : le sous-apprentissage et le sur-apprentissage.

Sous-apprentissage On estime qu'un classifieur a sous-appris s'il donne de faibles performances aussi bien sur les données sur lesquelles il a appris que sur celles sur lesquelles il est testé.

Sur-apprentissage On considère qu'un classifieur sur-apprend lorsque ses performances sur les données d'apprentissage sont très bonnes alors que celles des données de test sont mauvaises.

Dans les deux cas ci-dessus, le modèle est incapable de généraliser, soit parce qu'il n'a pas assez appris et qu'il n'a donc pas assez de connaissances à appliquer, soit parce qu'il est incapable d'extrapoler ce qu'il a appris à de nouvelles données.

Ensemble de validation

Afin de selectionner et d'évaluer une architecture de réseaux de neurones, on commence par diviser nos données en 2 sous-ensembles : un ensemble contenant les données de test et celles d'apprentissage et un *ensemble de validation*. Ce dernier contient des données inconnues du modèle et permettra de comparer les performances des différentes architectures afin de selectionner la meilleure.
Afin de s'assurer que les données de validation sont parfaitement dissociées de l'apprentissage, on extrait directement les exemples de cet ensemble que l'on garde de côté. Les exemples de l'ensemble d'apprentissage serviront à apprendre les paramètres W du modèle et l'ensemble de test à mesurer les performances du modèle selectionné sur l'ensemble de validation. Les performances sont évaluées par *validation croisée* en échantillonant plusieurs ensembles apprentissage + test comme décrit ci dessous.

Validation croisée

Nous évaluons également nos modèles par une méthode dite de *validation croisée* (ou *cross-validation*). Cette dernière consiste à diviser nos données X en K sous-ensembles puis à entraîner le classifieur sur $K-1$ sous-ensembles et à évaluer ses performances sur le dernier sous-ensemble. Afin de limiter la variance de l'erreur de notre évaluation, on effectue K fois l'algorithme en faisant varier à chaque itération le sous-ensemble de test.
En calculant la moyenne ainsi que la variance en apprentissage et en test, on obtient une indication sur les performances du classifieur.

Classe	Nombre d'exemples	% du nombre total d'exemples
1	112	10%
2	65	6 %
3	75	7 %
4	48	4 %
5	45	4 %
6	23	2 %
7	49	4 %
8	24	2 %
9	106	9 %
10	594	52 %
Total	1141	100 %

Table 1: Répartitions des exemples du dataset *Medical Images* selon les classes

4 Expérimentations

Afin d'étudier au mieux les performances des réseaux de neurones, nous avons commencé par expérimenter plusieurs architectures en faisant varier à chaque fois les différents hyperparamètres afin de se faire une idée de l'influence de chacun sur l'efficacité du réseau.
Pour ce faire, nous avons utilisé, en langage Python, l'interface Keras[8] basée sur la bibliothèque tensorflow[9] et avons utilisé des données temporelles récupérées sur le site UCR[1] qui constitue une base d'évaluation classique en classification de séries temporelles.

4.1 Le dataset "Medical Images"

Le premier dataset étudié est une série univariée à 10 classes de longueur 99 (pour un problème de classification donné, toutes les séries sur le site UCR ont la même taille). Chacun des 1141 exemples possède donc 99 valeurs qui correspondent aux variations en fonction du temps, soit 99 pas de temps.
Afin de mieux visualiser nos données, on représente 20 séries tirées de quelques-unes des 10 classes sous forme de graphiques dans la figure 2, avec en abscisse le pas de temps et en ordonnée les valeurs contenues dans la série.

Classification

La répartition de nos données en fonction des 10 classes est donnée dans la table 1. La premier constat que l'on fait est l'inégalité de la répartition des exemples avec notamment une classe 10 qui comptabilise à elle-seule plus de 52% des exemples. On s'attend donc à ce que nos modèles aient une précision de prédiction bien supérieure à cette valeur de 52%, autrement on s'expose à une classification majoritaire, c'est-à-dire une prédiction systèmatique de la classe 10.

4.2 MLP

On s'intéresse en premier lieu aux perceptrons multi couches. Afin de se faire une idée de l'influence des hyperparamètres d'un MLP sur ses performances, nous les faisons varier puis mesurons la précision (ou accuracy) de ses prédictions en faisant la moyenne des précisions obtenues pour tous les sous-ensembles de la crossvalidation. On s'assure que chaque sous-ensemble garde la même distribution de classe que le set de données original.

4.2.1 Nombre de neurones et de couches

Dans un premier temps, on s'intéresse aux performances d'un classifieur en fonction de son nombre de couches ainsi que du nombre de neurones par couche. On fixe le nombre d'epochs à 500 pour chaque modèle.
On commence par tracer les courbes de performance en apprentissage (*train*), en test et en validation en fonction du nombre de couches cachées que l'on représente dans la figure 3. On remarque

[1] http://www.cs.ucr.edu/~eamonn/time_series_data/

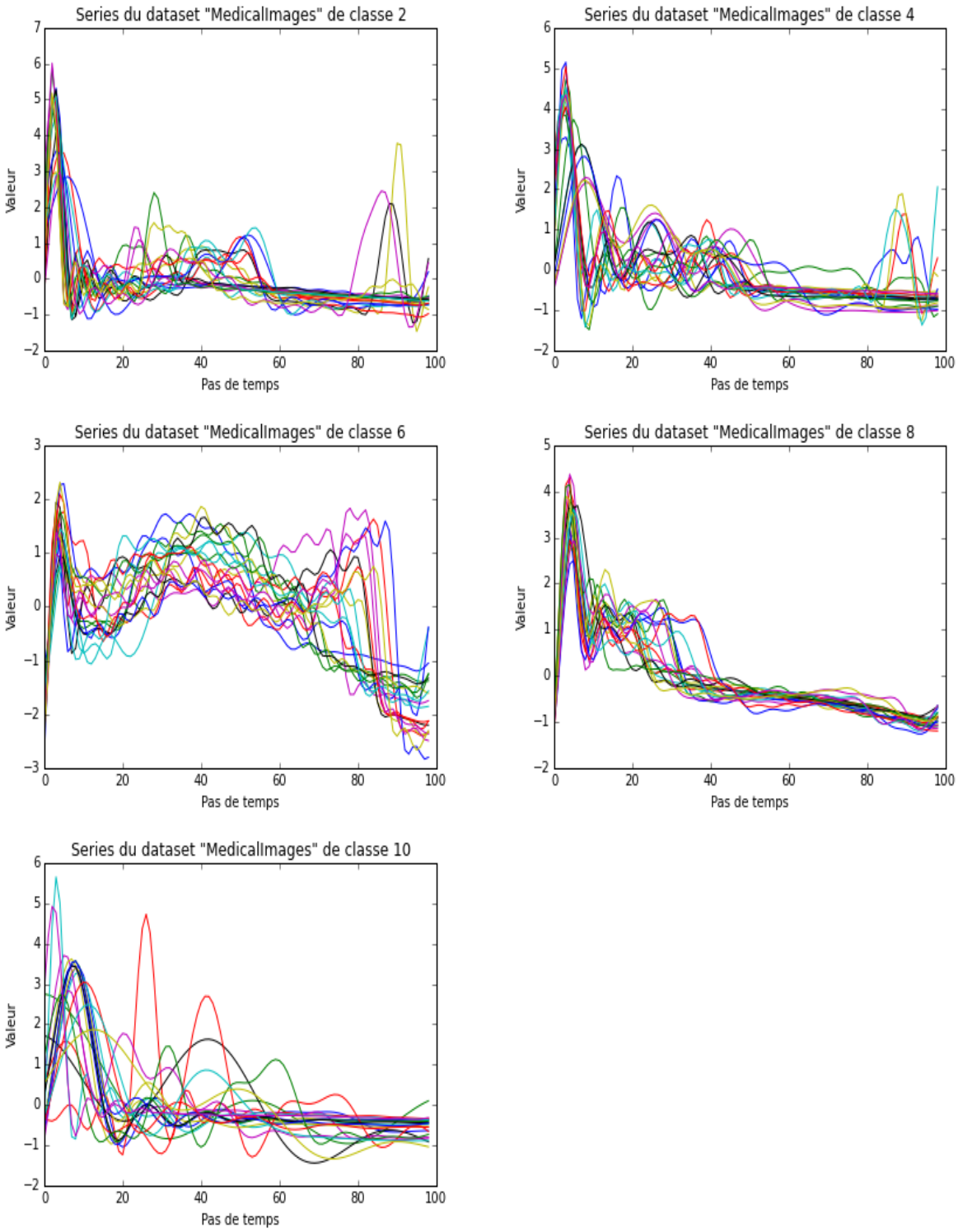

Figure 2: Visualisation d'une vingtaine de séries provenant de quelques classes du dataset *Medical Images*

immédiatement qu'il y a une nette marge entre les performances de nos modèles en apprentissage, qui ne descendent pas en-dessous des 93%, et celles en test/validation qui fluctuent de 65% à 89% pour la plus élevée. Cette différence est tout-à-fait normale compte tenu du fait que les données en apprentissage sont celles qui sont utilisées pour optimiser les paramètres du classifieur.
Si l'on s'intéresse à présent à l'influence du nombre de neurones par couches, on ne note pas de réelles différences de score. Cependant, les modèles ayant beaucoup de neurones dans leurs couches cachées (128 et plus) semblent avoir des scores minimaux légérement plus bas que les autres.
Afin de se faire une meilleure idée de l'influence du nombre de couches sur les performances, on trace au sein de la figure 4, la moyenne des scores en fonction du nombre de couches des données de la figure 3. Encore une fois, on ne peut pas réellement détecter une quelconque influence du nombre de couches sur les scores car ceux-ci restent assez constants. De plus, leurs écart-types étant plutôt faibles, ces valeurs sont donc assez représentatives de l'efficacité des modèles.

4.2.2 Algorithme d'optimisation

On s'intéresse ensuite à la phase d'apprentissage de nos classifieurs, plus précisément à l'évolution du coût et du score en fonction de l'epoch d'apprentissage. Pour ce faire, nous utilisons un *MLP* à une couche cachée de 100 neurones que l'on entraîne sur 2000 epochs. On utilise 2 algorithmes d'optimisation différents : la *descente de gradient stochastiques (ou SGD)* ainsi qu'*Adam (Adaptive Moment Estimation)*. On trace les courbes des figures 5 et 6.
On constate que la convergence est plus rapide et plus précise avec *Adam* qu'avec la descente de gradient stochastique. Ceci s'explique par le fait qu'*Adam* modifie le pas d'apprentissage en fonction de l'epoch.
On peut également constater que la convergence peut se faire bien avant que le nombre d'epochs spécifié soit atteint. C'est pour cela que l'on utilisera lors de nos apprentissages un *Early Stopping* permettant d'interrompre la phase d'apprentissage en cas de plateau, typiquement une fonction de coût qui ne change pas beaucoup pendant un certain nombre d'epochs.

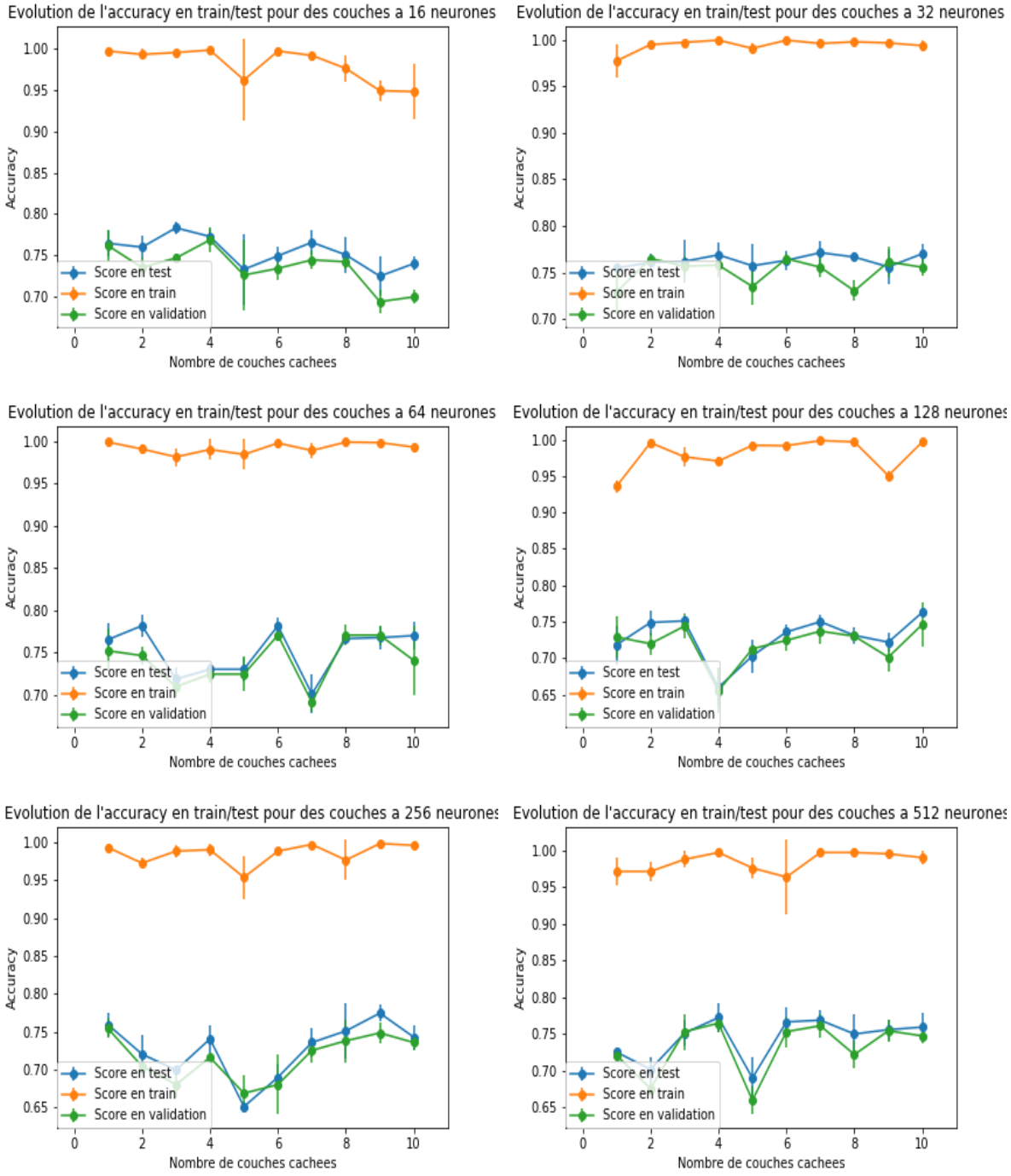

Figure 3: Variation de l'accuracy en fonction du nombre de couches du MLP (avec écart-type)

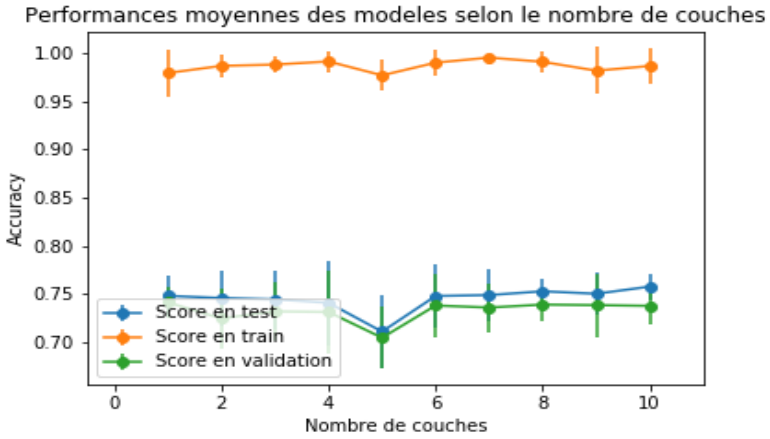

Figure 4: Variation moyenne de l'accuracy en fonction du nombre de couches du MLP (avec écart-type)

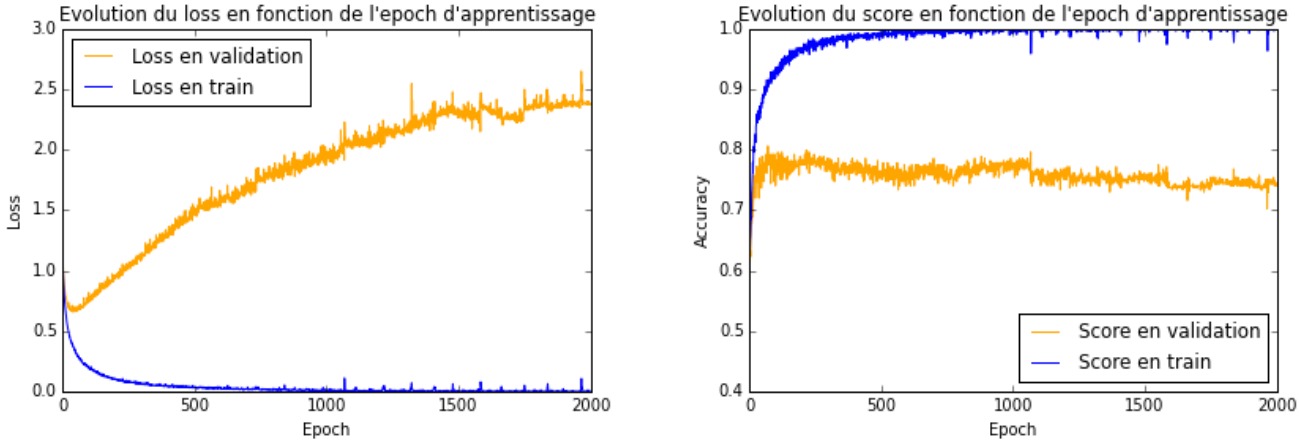

Figure 5: Variation du loss et de l'accuracy en fonction de l'epoch d'apprentissage avec un optimiseur Adam

4.2.3 Régularisation

On s'intéresse à présent aux différents types de régularisation ainsi qu'à leur influence sur les performances de nos classifications. On voudrait réduire l'effet de sur-apprentissage, c'est-à-dire réduire l'écart entre les scores en apprentissage et en test/validation.
Nous testons nos régularisations sur un *MLP* à une couche cachée de 100 neurones avec un nombre d'epochs d'apprentissage fixé à 500.

Dropout

La première régularisation que l'on expérimente est le *Dropout* en faisant varier le pourcentage de neurones désactivés de 0.1 à 1.0 par pas de 0.1. On trace les courbes correspondant à l'accuracy et au loss dans la figure 7.
On constate qu'en apprentissage, l'accuracy baisse et le loss augmente, ce qui est logique étant donné que le coût reflète l'erreur de classification. Cependant, en validation, on observe une légère baisse du loss pour des pourcentages de *Dropout* entre 0.7 et 0.9, ce qui correspond au comportement que l'on recherche.
Les courbes décrivent également un comportement étonnant pour des valeurs de 0.9 et 1.0. On note une chute de performance pour 0.9 alors que pour 1.0 le modèle se comporte comme s'il n'y

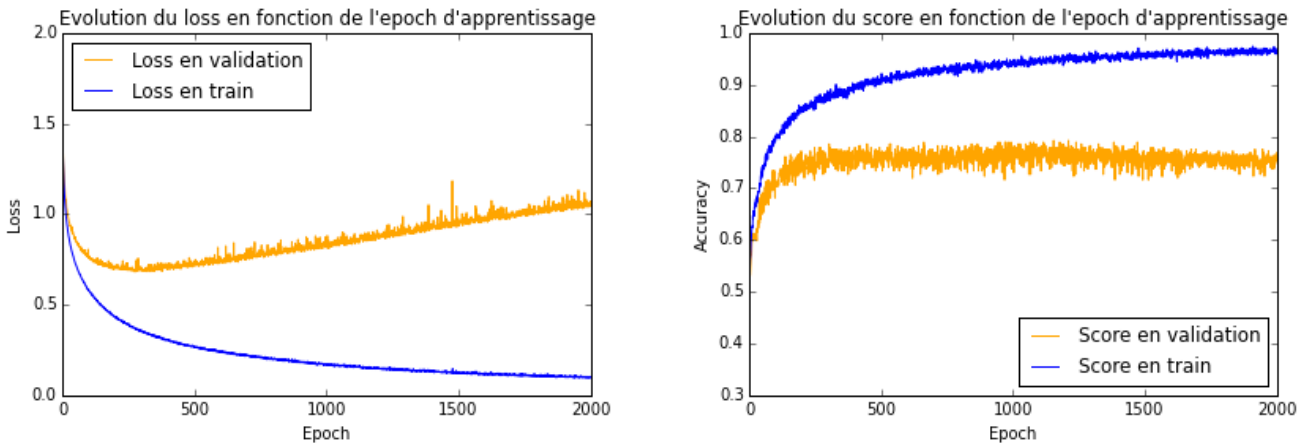

Figure 6: Variation du loss et de l'accuracy en fonction de l'epoch d'apprentissage avec un optimiseur SGD

avait aucune régularisation. On suppose que cela est dûe à l'implémentation logicielle faite qui ne prend en compte le *Dropout* que si l'argument passé est strictement entre 0.0 et 1.0.

L1

Pour tester l'impact de la régularisation *L1*, on fait varier le coefficient de régularisation de 0.001 à 1.0. On mesure ensuite l'accuracy et le loss que l'on reporte dans la figure 8.
On constate que la régularisation fait immédiatement chuter les performances du classifieurs qui passent de 1.0 à 0.52 en train et d'environ 0.75 à 0.53 en test/validation. On rappelle que la classe majoritaire dans le dataset représente 52% des données, on suppose que les scores obtenus correspondent à une classification systèmatique en classe 10. Ce comportement peut être dû aux coefficients de régularisation choisis qui ne correspondent pas du tout aux valeurs optimales. Pour trouver l'optimum, il faudrait procéder un gridSearch plus poussé voire même faire une recherche dichotomique en prenant en compte les performances du modèle. Il peut également être dû au fait que notre modèle n'ait pas besoin de régularisation pour le problème de classification donné, c'est-à-dire qu'il ne sur-apprend pas vraiment.

L2

On s'attaque ensuite à l'étude d'une régularisation de type *L2*. On trace une nouvelle fois le coût et le score en fonction du coefficient de régularisation que l'on fait varier de 0.001 à 1.0. On trace les courbes d'accuracy et de loss dans la figure 9.
On constate, comme pour la régularisation *L1*, une forte chute de performances du modèle que l'on explique également par un mauvais choix de coefficient de régularisation. Cependant, étant donné l'évolution de la courbe, on peut aussi statuer que la régularisation ne permet pas d'améliorer les performances de notre classifieur, son expressivité n'est donc pas trop grande par rapport au problème de classification ciblé.

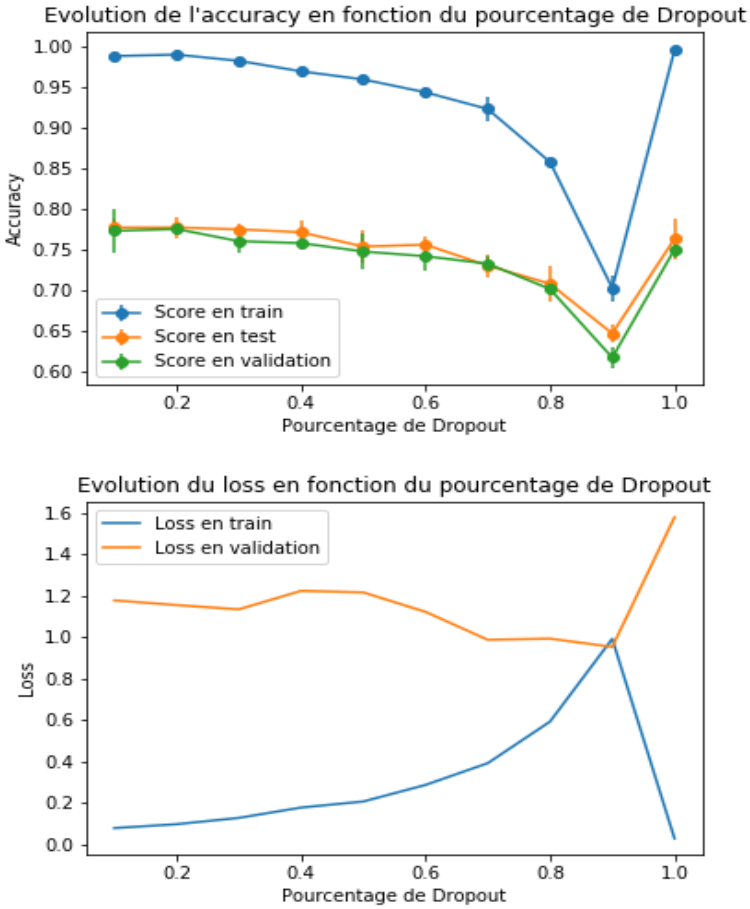

Figure 7: Evolution de l'accuracy et du loss en fonction pourcentage de Dropout

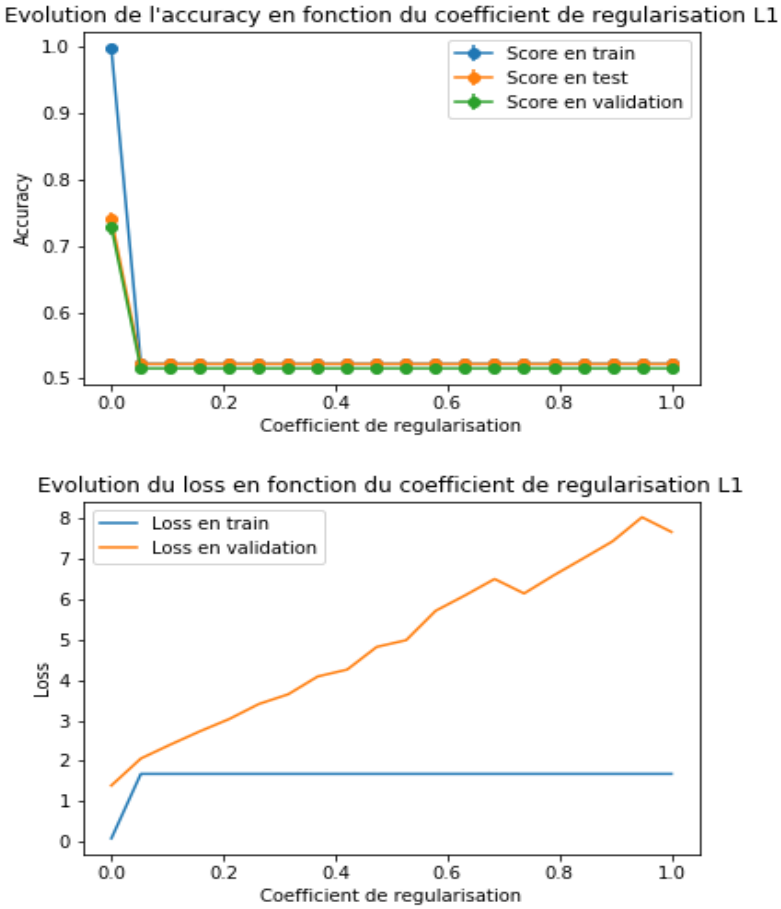

Figure 8: Evolution de l'accuracy et du loss en fonction du coefficient de régularisation L1

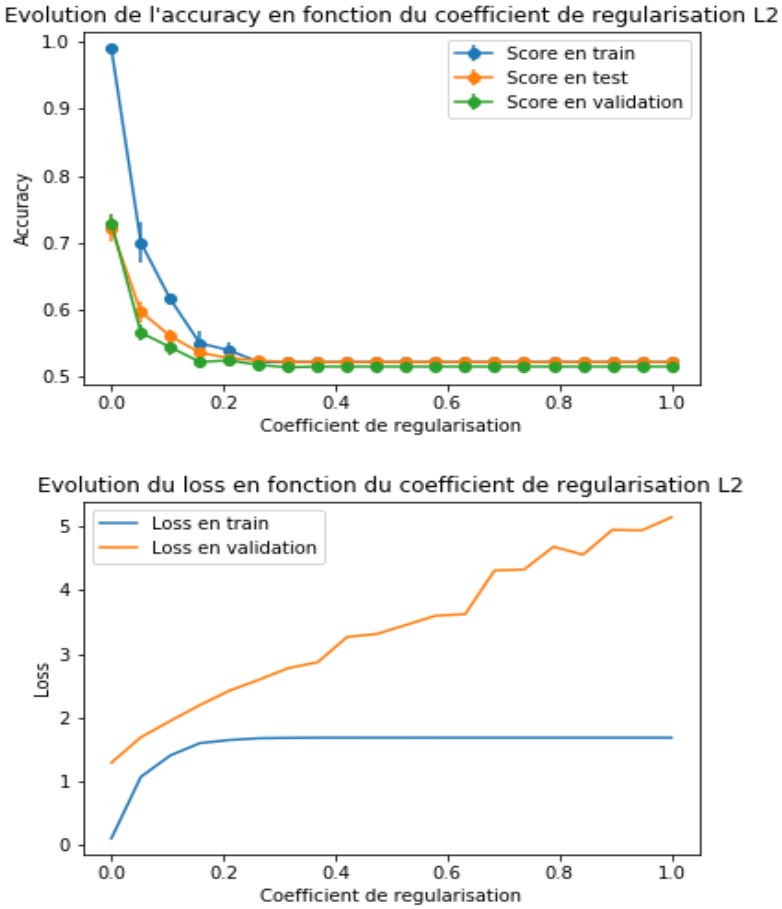

Figure 9: Evolution de l'accuracy et du loss en fonction du coefficient de régularisation L2

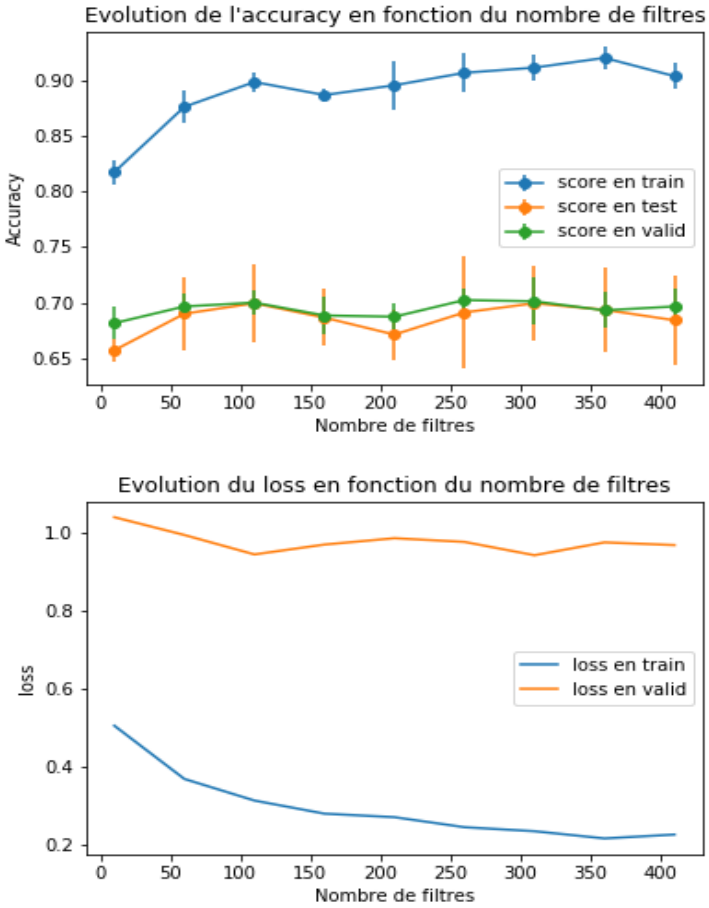

Figure 10: Evolution de l'accuracy et du loss en fonction du nombre de filtres du *CNN*

4.3 CNN

Toujours avec le même dataset, on s'intéresse aux performances d'un réseau de neurones convolutif. Nous procédons de la même manière qu'avec les *MLP*, en faisant varier tour à tour les hyperparamètres du modèle afin d'en mesurer l'impact.

4.3.1 Nombre de filtres et de couches de convolution

Comme dans le cas du *MLP*, nous testons plusieurs architectures de *CNN* en faisant varier le nombre de filtres (l'équivalent du nombre de neurones d'un *MLP*) ainsi que le nombre de couches de convolution.

On se fixe un premier modèle avec une unique couche de convolution dont on fait varier le nombre de filtres. La taille du filtre est de 3 et on ne met aucune couche de pooling. On mesure ensuite le loss ainsi que l'accuracy que l'on reporte sur la figure 10.
On note que le nombre de filtres influe sur la faculté de notre classifieur à apprendre, comme le montre la valeur descendante du loss et l'augmentation de l'accuracy. Cependant, cette amélioration ne se répercute pas sur les performances en test et validation qui changent très peu d'une valeur à l'autre.

Dans un second temps, on fait varier la taille du filtre de convolution de notre réseau de neurones, c'est-à-dire la fenêtre de valeurs à laquelle le filtre est appliqué. On choisit un modèle avec une unique couche de convolution à 100 filtres suivie d'un *MaxPooling* de taille 5. On mesure une nouvelle fois le coût ainsi que l'accuracy, que l'on reporte dans la figure 11. On remarque une légère amélioration du score ainsi qu'une diminution du loss aussi bien en train qu'en test/validation. On en déduit donc que la taille du filtre, lorsqu'elle est bien choisie, permet d'améliorer significativement les performances d'un réseau convolutif.

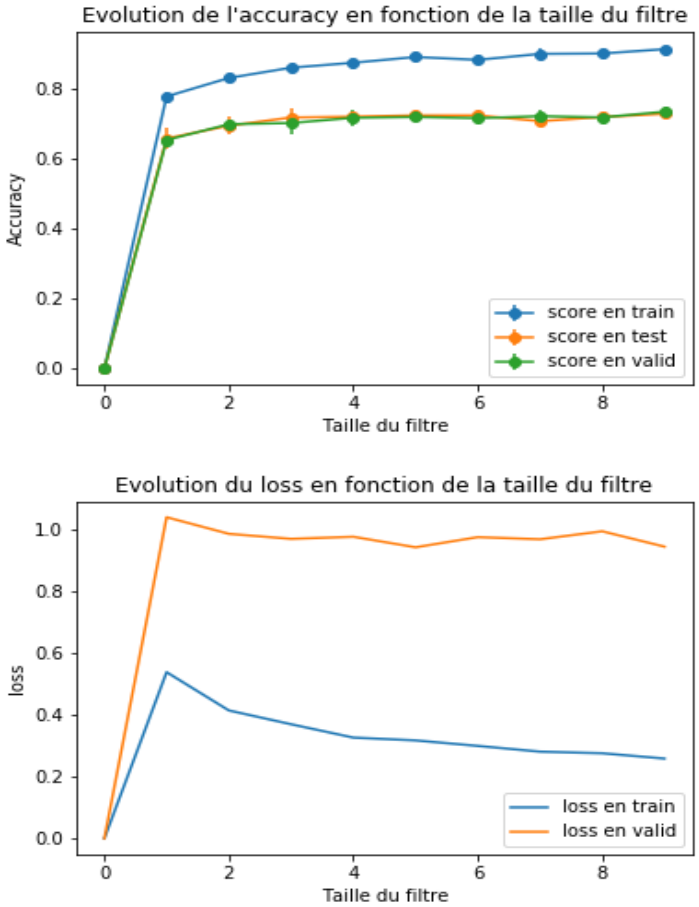

Figure 11: Evolution de l'accuracy et du loss en fonction du nombre de filtres du *CNN*

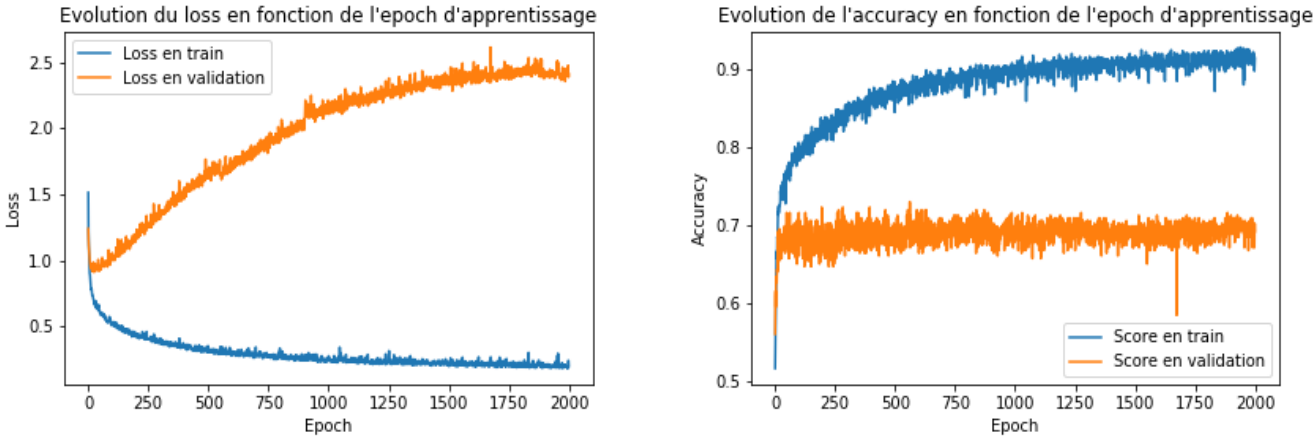

Figure 12: Variation du loss et de l'accuracy en fonction de l'epoch d'apprentissage pour un CNN

4.3.2 Apprentissage

On désire à présent observer l'évolution du loss ainsi que de l'accuracy d'un *CNN* au cours de l'apprentissage avec comme optimiseur *Adam*, ces valeurs sont consignées dans la figure 12.
Les courbes montrent une convergence "classique" faite par l'algorithme *Adam*, à savoir qui se stabilise petit à petit au fil des epochs.
On remarque également que malgré une nette amélioration des performances en train, les scores en validation stagnent et ce malgré l'augmentation du loss.

4.4 LSTM

Enfin, la dernière architecture de réseau de neurones à laquelle on s'intéresse est celle des réseaux récurrents, plus précisément les modèles utilisant comme neurones des unités *LSTM*.
Pour étudier leur fonctionnement, on fait une nouvelle fois varier les hyper-paramètres en observant leur impact sur les performances.

4.4.1 Nombre d'unités LSTM

On s'intéresse en premier lieu à l'influence du nombre d'unités sur les performances. Chacune des unités traite la série en entier avant de retourner la dernière valeur calculée. On se fixe un modèle à une couche cachée où l'on fait varier le nombre de *LSTM*. Les performances sont visibles sur la figure 13. L'apprentissage se fait sur 500 epochs.
Les performances de nos modèles sont très mauvaises et ne dépassent pas les 52% qui correspondent au nombre d'exemples de la classe majoritaire. Nous avons vérifié cela en comptant les classes prédites par le *LSTM* sur le dataset et il en resssort que la classification est systèmatiquement faite en classe 10.
Les *LSTM* étant des unités complexes, il est possible que ces performances soient en fait dûes à une non-convergence de nos modèles (un nombre d'epoch peut être insuffisant). Afin de vérifier cela, nous traçons les courbes relatant l'évolution d'un modèle à une couche cachée contenant 4 *LSTM* en fonction de l'epoch d'apprentissage et on augmente le nombre d'epochs à 2000. Les résultats sont dans la figure 14.
On constate une augmentation des performances qui plafonne à présent autour de 59/60 %. L'optimiseur étant *Adam*, on suppose pouvoir améliorer les performances en réduisant encore le pas d'apprentissage, il faudrait pour cela relancer l'apprentissage avec un nombre d'epochs encore plus élevé, ce qui n'a pas été fait dans le cadre de ce projet.

4.4.2 Classification à chaque pas de temps

Dans les modèles précédents de *LSTM*, nos unités ne retournaient que la dernière valeur calculée, à savoir le résultat après avoir parcouru l'intégralité de la série. On désire à présent s'intéresser au comportement d'un *LSTM* retournant le résultat calculé pour chaque pas de temps, la classification

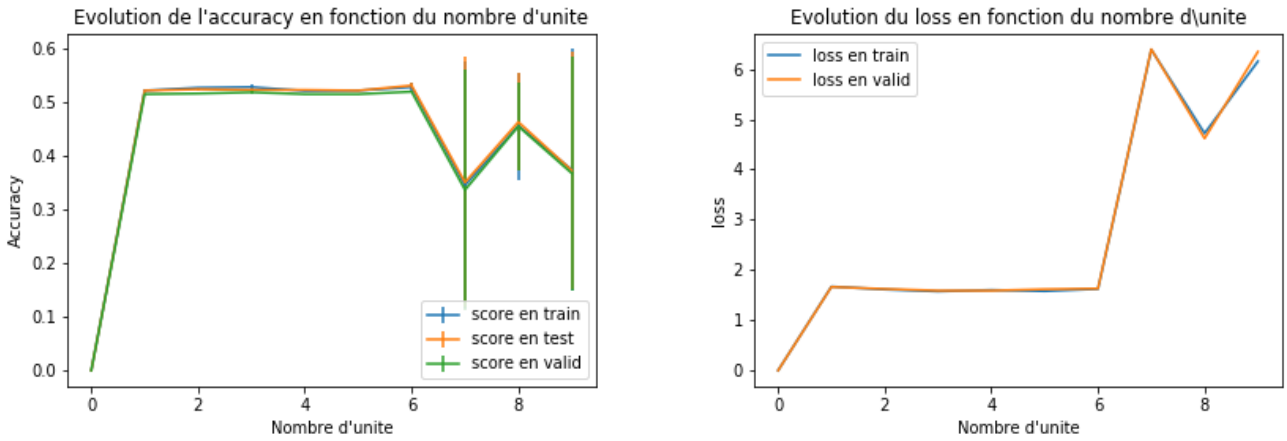

Figure 13: Variation du loss et de l'accuracy en fonction du nombre d'unités *LSTM*.

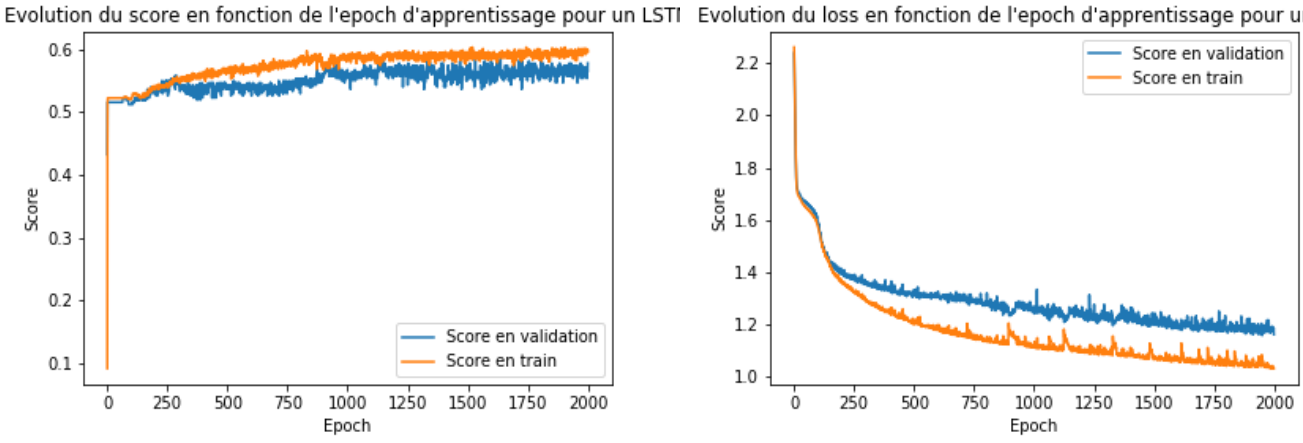

Figure 14: Variation du loss et de l'accuracy en fonction de l'epoch d'apprentissage pour un LSTM

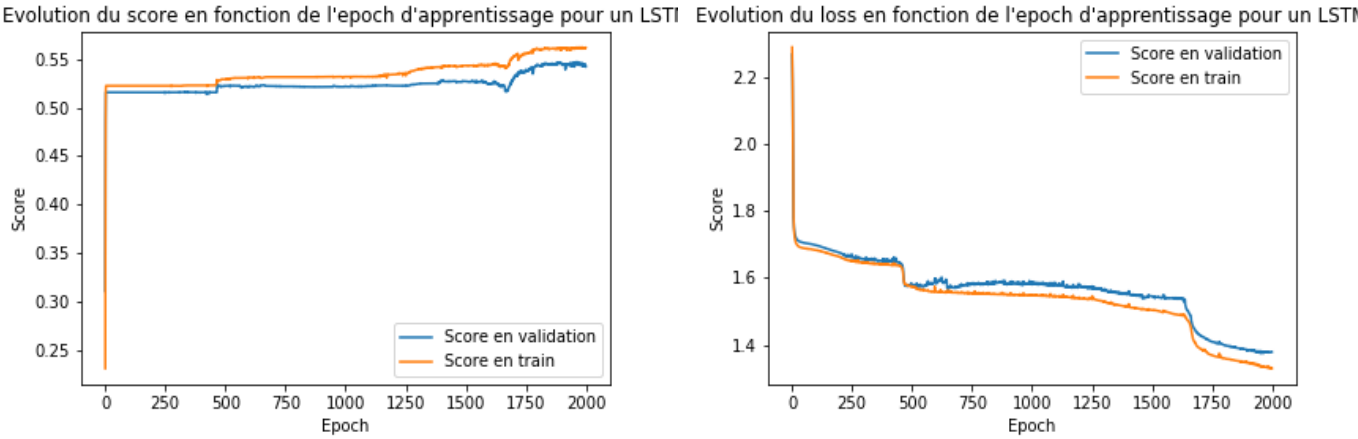

Figure 15: Variation du loss et de l'accuracy en fonction de l'epoch d'apprentissage pour un LSTM en TimeDistributed

est ensuite faite pour chaque pas de temps également.
On se fixe à nouveau le même modèle, à savoir un réseau à une couche cachée composée de 4 unités *LSTM* dont l'apprentissage est fait sur 2000 epochs. Les performances sont reportées sur la figure 15.
On constate une convergence encore moins bonne que pour les *LSTM* ne prenant pas en compte chaque pas de temps. Cependant, on constate une diminution, certes lente, mais notable du loss. De plus, on distingue des paliers dans ces diminutions, probablement dûes à une baisse du learning rate qui aura conduit l'optimiseur à sortir d'un minimum local.

Commentaires sur les performances des *LSTM*

Les *LSTM* étant des unités adaptées au traitement de séries temporelles, on s'attendait à de meilleures performances de ces classfieurs. Cependant, on constate que ces derniers se contentent de classifier les exemples dans la classe majoritaire, ce qui donne une accuracy de 52%.
Une première explication serait que l'optimisation d'un *LSTM* est beaucoup plus longue et nécessite peut être des paramètres différents que ceux donnés durant nos expériences. Une seconde hypothèse serait que les *LSTM* ne sont pas adaptés à des datasets dont la distribution des classes est aussi inégale.

5 Application et automatisation de sélection de modèle

Pour notre implémentation logicielle de sélection automatique de modèle, on utilise, en Python, les librairies Keras (modèles) et ScikitLearn (gridSearch). Afin de pouvoir tirer parti des fonctionnalités des deux bibliothèques, nous avons dû définir et redéfinir plusieurs fonctions/classes.

5.1 Classifieur

Nous désirons utiliser le *GridSearch* de la librairie *sklearn* afin d'entraîner plusieurs réseaux en parallèles, il est donc nécessaire de rendre compatibles les modèles de *Keras* avec les fonctions de *sklearn*.
Pour ce faire, on utilise un *wrapper* prévu à cet effet. De plus, comme notre finalité est d'utiliser plusieurs modèles en parallèle, on se dote de fonctions permettant de retourner un classifieur selon les spécifications des paramètres entrés.
La mesure des performances des modèles est faite selon les valeurs d'un ensemble de validation, que l'on sépare du dataset initial avant même l'apprentissage. Pour s'assurer d'avoir une distribution des classes semblable à celle des données initiales, nous avons recours à l'objet *StratifiedKFold* de la bibliothèque *sklearn*.
Enfin, pour que l'apprentissage se fasse le plus vite possible, on fixe un *EarlyStopping* sur le loss de validation tel que si la variation est inférieure à 0.001 pendant 40 epochs, l'apprentissage s'arrête automatiquement.

Builders

On utilise, pour chaque type de réseau neuronal, une fonction de création de modèle.

MLPBuilder fonction permettant de créer un MLP. Les paramètres sont

input_shape dimensions d'un exemple. Utilisé pour définir les dimensions d'entrée de la toute première couche.

nbClasses Nombre de classes à prédire. Utilisé pour les dimensions de sortie de la dernière couche du réseau.

listDims Liste des dimensions des couches cachées. Utilisé pour fixer le nombre de neurones dans chaque couche et pour inférer le nombre de couches. Par défaut, on utilise une couche cachée à 10 neurones.

activation Fonction d'activation à utliser dans chaque couche (excepté pour la couche cachée pour laquelle on utilise soit une sigmoïde dans le cas binaire, soit un softmax dans le cas multiclasse). Par défaut, toutes les couches utilisent un *relu*.

loss Fonction de coût à optimiser lors de l'apprentissage. Par défaut, on utilise une entropie binaire (dans le cas d'une classification binaire) ou une entropie croisée (dans le cas multi-classes).

opt Algorithme d'optimisation à utiliser en apprentissage. Par défaut, on utilise *adam* pour favoriser une meilleure convergence.

listLayerReg Liste contenant l'indice des couches contenant une régularisation. Par défaut, aucune.

regularisations Liste des différentes régularisations à appliquer. Si la liste ne contient qu'une seule régularisation mais que plusieurs couches doivent être régularisées, la même fonction de régularisation est utilisée pour chaque couche. Par défaut, aucune.

callbacks Liste des *callbacks* à utiliser dans notre modèle. Les *callbacks* sont des fonctions qui interviennent pendant l'apprentissage. Par défaut, on utilisera un *Early Stopping*.

CNN1DBuilder fonction permettant de créer un réseau convolutif en une dimension. Les paramètres sont :

input_shape dimensions d'un exemple. Utilisé pour définir les dimensions d'entrée de la toute première couche.

list_nb_filter Liste du nombre de filtres. Utilisé pour connaitre la dimension de chaque couche.

nbClasses Nombre de classes à prédire. Utilisé pour les dimensions de sortie de la dernière couche du réseau.

list_filter_length Liste des tailles des filtres. Utilisé pour connaitre la taille de chaque filtre, pour chaque couche.

list_pooling Liste des poolings. Utilisé pour connaître le type de pooling et sa valeur, les couches de poolings sont placées après les couches de convolution.

activation Fonction d'activation à utliser dans chaque couche (excepté pour la couche cachée pour laquelle on utilise soit une sigmoïde dans le cas binaire, soit un softmax dans le cas multiclasse). Par défaut, toutes les couches utilisent un *relu*.

loss Fonction de coût à optimiser lors de l'apprentissage. Par défaut, on utilise une entropie binaire (dans le cas d'une classification binaire) ou une entropie croisée (dans le cas multi-classes).

opt Algorithme d'optimisation à utiliser en apprentissage. Par défaut, on utilise *adam* pour favoriser une meilleure convergence.

listLayerReg Liste contenant l'indice des couches contenant une régularisation. Par défaut, aucune.

regularisations Liste des différentes régularisations à appliquer. Si la liste ne contient qu'une seule régularisation mais que plusieurs couches doivent être régularisées, la même fonction de régularisation est utilisée pour chaque couche. Par défaut, aucune.

callbacks Liste des *callbacks* à utiliser dans notre modèle. Les *callbacks* sont des fonctions qui interviennent pendant l'apprentissage. Par défaut, on utilisera un *Early Stopping*.

RNNBuilder fonction permettant de créer un réseau récurrent simple, *LSTM* et *GRU*.

input_shape dimensions d'un exemple. Utilisé pour définir les dimensions d'entrée de la toute première couche.

nbClasses Nombre de classes à prédire. Utilisé pour les dimensions de sortie de la dernière couche du réseau.

nb_variation Nombre de variables de chaque série. Utilisé pour spécifier si les séries sont univariées ou multi-variées.

listDims Liste des dimensions des couches cachées. Utilisé pour fixer le nombre de neurones dans chaque couche et pour inférer le nombre de couches. Par défaut, on utilise une couche cachée à 10 neurones.

activation Fonction d'activation à utliser dans chaque couche (excepté pour la couche cachée pour laquelle on utilise soit une sigmoïde dans le cas binaire, soit un softmax dans le cas multiclasse). Par défaut, toutes les couches utilisent un *relu*.

loss Fonction de coût à optimiser lors de l'apprentissage. Par défaut, on utilise une entropie binaire (dans le cas d'une classification binaire) ou une entropie croisée (dans le cas multi-classes).

opt Algorithme d'optimisation à utiliser en apprentissage. Par défaut, on utilise *adam* pour favoriser une meilleure convergence.

listLayerReg Liste contenant l'indice des couches contenant une régularisation. Par défaut, aucune.

regularisations Liste des différentes régularisations à appliquer. Si la liste ne contient qu'une seule régularisation mais que plusieurs couches doivent être régularisées, la même fonction de régularisation est utilisée pour chaque couche. Par défaut, aucune.

callbacks Liste des *callbacks* à utiliser dans notre modèle. Les *callbacks* sont des fonctions qui interviennent pendant l'apprentissage. Par défaut, on utilisera un *Early Stopping*

5.2 GridSearch

L'objet *GridSearch* fourni par *sklearn* permet d'évaluer un modèle en retournant le score (accuracy) et le loss sur les différents ensembles d'apprentissage et de test. Cependant, on désire également pouvoir utiliser un ensemble de validation afin de sélectionner l'architecture optimale de notre réseau de neurones. Afin de modifier le comportement de cette fonction, nous avons dû copier l'intégralité du code *sklearn* et le modifier en profondeur afin qu'il prenne en compte les sets de validation des modèles *Keras*.
A la fin d'un *GridSearch*, tous les résultats sont consignés dans un dictionnaire de Python. Nous

récupérons toutes ces valeurs et les écrivons dans un fichier *html* sous la forme d'un tableau afin d'avoir plus de lisibilité.
Les principales informations que l'on récupére du dictionnaire sont :

mean_train_score : moyenne des scores des modèles en apprentissage.

std_train_score : écart-type des scores des modèles en apprentissage.

mean_loss : moyenne du loss des modèles en apprentissage.

std_loss : écart-type du loss des modèles en apprentissage.

mean_test_score : moyenne des scores des modèles en test.

mean_val_score : moyenne des scores des modèles en validation.

std_val_score : écart-type des scores des modèles en validation.

mean_val_loss : moyenne du loss des modèles en validation.

std_val_loss : écart-type du loss des modèles en validation.

Toutes ces mesures sont également disponibles pour chacun des sous-ensembles (*split*) construit en cross-validation. Sont également listés, l'ensemble des paramètres entrés dans les builders. Un exemple d'affichage de fichier html est donné dans la figure 16, après élimination des colonnes inintéressantes.

References

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[2] H. Abdi, *Les réseaux de neurones*. Presses universitaires de Grenoble Grenoble, 1994.

[3] B. NUSANTARA, "Convolutional neural network,"

[4] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[5] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting.," *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[6] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[7] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[8] F. Chollet *et al.*, "Keras." https://github.com/fchollet/keras, 2015.

[9] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: A system for large-scale machine learning," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Savannah, Georgia, USA*, 2016.

	param_listDims	mean_test_score	std_test_score	mean_train_score	std_train_score	mean_val_loss	std_val_loss	mean_val_score	std_val_score
30	[128]	0.782864	0.007231	0.995888	0.002208	1.740486	0.059455	0.746251	0.005881
38	[128, 128, 128, 128, 128, 128, 128, 128, 128]	0.781690	0.013674	0.991205	0.005727	2.167765	0.580534	0.746251	0.010696
32	[128, 128, 128]	0.781690	0.009520	0.998244	0.001432	1.765031	0.117168	0.770473	0.007110
22	[64, 64, 64]	0.774648	0.011326	0.998241	0.001432	1.905443	0.252605	0.748558	0.014498
51	[512, 512]	0.772300	0.009544	0.998824	0.000831	1.820853	0.278032	0.768166	0.014950
42	[256, 256, 256]	0.772300	0.019442	0.997068	0.002190	1.593673	0.303800	0.764706	0.012315
45	[256, 256, 256, 256, 256, 256]	0.771127	0.012330	0.995886	0.003632	2.153534	0.383313	0.755479	0.010696
54	[512, 512, 512, 512, 512]	0.769953	0.010709	0.993538	0.005808	1.983928	0.075692	0.755479	0.008631
44	[256, 256, 256, 256, 256]	0.769953	0.016698	0.993547	0.004615	1.886855	0.181378	0.740484	0.040746
33	[128, 128, 128, 128]	0.768779	0.014614	0.997072	0.002189	1.809718	0.065895	0.761246	0.015730
41	[256, 256]	0.768779	0.013631	0.999415	0.000827	1.809885	0.127171	0.757785	0.012947
37	[128, 128, 128, 128, 128, 128, 128, 128]	0.767606	0.014201	0.998826	0.000830	2.235130	0.269790	0.770473	0.011418
46	[256, 256, 256, 256, 256, 256, 256]	0.766432	0.003651	0.997650	0.000837	2.925199	0.551360	0.730104	0.011301
35	[128, 128, 128, 128, 128, 128]	0.766432	0.019535	0.963613	0.050215	1.700875	0.236586	0.753172	0.020825
31	[128, 128]	0.766432	0.000835	0.999415	0.000827	1.918443	0.083934	0.770473	0.012740
57	[512, 512, 512, 512, 512, 512, 512, 512]	0.765258	0.015224	0.992371	0.003320	2.427089	0.148573	0.743945	0.011301
24	[64, 64, 64, 64, 64]	0.765258	0.018842	0.999411	0.000833	1.840670	0.202300	0.752018	0.025479
50	[512]	0.764085	0.016359	0.997653	0.001660	1.820632	0.177329	0.761246	0.019777
34	[128, 128, 128, 128, 128]	0.762911	0.010789	0.999411	0.000833	1.982441	0.187275	0.764706	0.004893
26	[64, 64, 64, 64, 64, 64, 64]	0.762911	0.005491	0.997070	0.001650	1.911304	0.160433	0.746251	0.030209
40	[256]	0.761737	0.022925	0.997066	0.000830	1.721360	0.059320	0.756632	0.010696
43	[256, 256, 256, 256]	0.760563	0.010123	0.994716	0.003803	1.815388	0.232116	0.764706	0.002825
48	[256, 256, 256, 256, 256, 256, 256, 256, 256]	0.759390	0.013807	0.993559	0.006766	2.823719	0.156266	0.734717	0.011418
47	[256, 256, 256, 256, 256, 256, 256, 256]	0.759390	0.020193	0.990019	0.009240	2.389447	0.371365	0.747405	0.010187
20	[64]	0.759390	0.015030	0.992949	0.005770	1.625154	0.100492	0.754325	0.012947
36	[128, 128, 128, 128, 128, 128, 128]	0.757042	0.023964	0.990600	0.005815	1.719665	0.298366	0.734717	0.020044
39	[128, 128, 128, 128, 128, 128, 128, 128, 128, ...	0.755869	0.013956	0.995305	0.006640	2.089898	0.067194	0.754325	0.015730
52	[512, 512, 512]	0.755869	0.018802	0.996479	0.000010	2.066320	0.141065	0.761246	0.015730
49	[256, 256, 256, 256, 256, 256, 256, 256, 256, ...	0.754695	0.002209	0.977096	0.018351	2.808232	0.158577	0.728950	0.023010
23	[64, 64, 64, 64]	0.751174	0.037041	0.976468	0.026191	2.017541	0.277492	0.738178	0.028440
2	[16, 16, 16]	0.751174	0.007377	0.976488	0.013117	1.673965	0.213145	0.743945	0.017644

Figure 16: Tableau HTML obtenu pour les variations du nombre de couches/neurones du MLP