

PROJET SEMESTRIEL

Apprentissage profond pour l'escalade



FIGURE 1 – Photo d'une escaladeuse

Réalisé par :

Ariane DEPONTHIEUX

Roxane LEDUC

Anouk ANDRE

Encadré par : M. KNIPPEL

Janvier 2023 — Juin 2023

Table des matières

1	Introduction	2
2	Présentation des données	3
2.1	Appellation des fichiers de données	3
2.2	Description des variables	4
3	Analyse statistique descriptive	6
3.1	Chargement des données	6
3.2	Représentation des données X/Y	6
3.3	Analyse de la donnée Jerk_pos	7
3.4	Analyse des données Reduced_state	8
3.5	Analyse des données Hip_roll	9
3.6	Analyse de la vitesse de grimpe	10
3.7	Dendrogramme	11
3.8	Méthode Elbow	11
3.9	Kmeans	12
3.10	Analyse par Classe	12
4	Réseaux de neurones	15
4.1	Qu'est ce qu'un réseau de neurones ?	15
4.2	Réseaux de neurones à propagation avant (Feedforward)	17
4.3	Réseaux de neurones récurrents (Recurrent Neural Networks - RNN)	18
4.4	Auto-encodeurs et auto-encodeurs variationnels	21
4.4.1	Auto-encodeurs	21
4.4.2	Auto-encodeurs varitationnels	21
4.5	Réseaux de neurones convolutifs (Convolutional Neural Networks - CNN)	23
4.6	Réseaux antagonistes génératifs (Generative Adversarial Networks - GAN)	24
4.7	Réseau de l'État d'Écho (Echo State Networks - ESN)	26
4.7.1	Introduction	26
4.7.2	Equation	26
4.7.3	Phase d'échantillonnage	26
4.7.4	Phase de calcul des poids	27
4.8	Implémentation avec Tensorflow	28
5	Conclusion	36
6	Annexe	37
7	Bibliographie	40

1 Introduction

L'escalade est un sport qui implique des compétences physiques et mentales, telles que la force, l'endurance, la flexibilité, la technique et la prise de décision rapide. Les sportifs d'escalade doivent s'entraîner régulièrement pour améliorer leurs compétences et atteindre leurs objectifs de performance. Cependant, il peut être difficile pour les athlètes de mesurer leurs performances en raison de la nature complexe et variée de l'escalade.

Dans ce contexte, l'apprentissage statistique ou machine learning peut offrir une solution pour aider les sportifs à mieux comprendre leurs performances et à atteindre leurs objectifs d'entraînement. En appliquant des techniques de machine learning à des données provenant de sportifs d'escalade dans des conditions variées, il est possible d'extraire des informations utiles sur les performances, les habitudes d'entraînement et les facteurs clés de succès.

Dans ce projet, nous proposons de découvrir et mettre en œuvre des techniques d'apprentissage statistique appliquées à des données provenant de sportifs faisant de l'escalade dans des conditions variées. Nous allons explorer des techniques telles que la régression, la classification, les algorithmes d'apprentissage non supervisés et les réseaux de neurones pour analyser les données et extraire des informations pertinentes pour les sportifs.

Une première grosse partie sera donc consacrée à une analyse statistique descriptive des données et une seconde à la compréhensions et l'utilisation des réseaux de neurones, grâce notamment à la bibliothèque **Tensorflow**.

2 Présentation des données

La base de données (105 fichiers `.mat`) qui nous a été fournie comporte un grand nombre d'interrogations (détaillées ci-dessous). Nous avons considéré (après longue réflexion) que chaque fichier correspondait à la montée d'un grimpeur sur une voie (on suppose que toutes les voies sont les mêmes).

L'objectif de ce projet sera donc de pouvoir estimer le niveau d'un nouveau grimpeur à partir de certaines données récupérées lors de son escalade de la voie.

Nous n'avons pas pu avoir de renseignements quant au niveau de chaque grimpeur (pas de labels). Il s'agit donc là d'une classification non supervisée. Une grosse partie du travail consistera donc à commencer par effectuer une classification par groupe de niveaux à l'aide par exemple de la méthode de partitionnement `k-means`.

L'expérience permet 100 relevés par secondes. Ainsi, pour chaque fichier `".mat"`, chaque mesure sont séparées par un intervalle de temps de :

$$\Delta t = 0.01 \text{ s} = 10 \text{ ms}$$

Selon le nombre de données relevées, on peut ainsi déduire la durée d'escalade de la voie de chaque grimpeur. À noter qu'un grimpeur a entre 5000 et 9000 données, ce qui correspond à entre 50" à 1'30" de grimpe environ.

2.1 Appellation des fichiers de données

La première grosse interrogation a été de chercher à comprendre le sens de l'appellation des fichiers. Ils ont la forme suivante :

`G1_P<numéro de la personne>_S<1|2>_V<1|2|3|4>_E<1|2|3|4>_<date DDMMYYYY>.mat`

Exemple de nom de fichier : `G1_P11_S1_V1_E1_21032014.mat`

Nous avons émis l'idée que les capteurs de mesure étaient positionnés sur 31 personnes (P00, P01, P02, ... , P32 ; sans P16 et sans P22) avec :

- P00 à P11 : 4 enregistrements pour une montée sur chaque voie : V1, V2, V3 et V4
- P12 à P32 (pas de P16, P22) : 3 enregistrements pour une montée sur chaque voie : V2, V3 et V4

Cependant, après analyse, nous ne sommes pas certaines de ce que nous avançons et nous considérerons donc dans la suite de l'étude que toutes les voies sont les mêmes.

Pour chaque grimpeur et à chaque montée, les données suivantes sont relevées :

Workspace	
Name	Value
Afford_count	7797x1 double
Hip_local_var	7297x1 double
Hip_roll	7797x1 double
Jerk_pos	1.0613e+14
Jerk_rot	1.6511e+15
Left_foot	7797x1 double
Left_hand	7797x1 double
Neck_local_var	7297x1 double
Neck_roll	7797x1 double
Reduced_state	7797x1 double
Right_foot	7797x1 double
Right_hand	7797x1 double
Roll_correl	7297x1 double
transition	5x5 double
X	7797x1 double
Y	7797x1 double

FIGURE 2 – Noms et formats des 16 variables contenus dans un fichier `.mat` quelconque

2.2 Description des variables

Jerk_rot/Jerk_pos : valeurs réelles

Le **jerk** est un terme utilisé en escalade qui correspond au changement d'accélération au cours du temps. C'est la variation temporelle de l'accélération, autrement dit, c'est la dérivée de l'accélération par rapport au temps.

Plus l'accélération change, plus le jerk augmente. Ainsi, un escaladeur qui a un petit jerk, signifie une grande fluidité lors de la montée. A l'inverse, un grand jerk indique un mouvement très saccadé.

transition : matrice 5×5 ou 1×1

La matrice transition a été construite à partir des autres données contenues dans les fichiers `.mat`. Elle contient donc des informations non pertinentes et ne sera à priori pas utile dans le cas de notre étude (on ne l'utilisera pas).

X/Y : matrice à n lignes et 1 colonne de valeurs réelles

Les coordonnées (X,Y) correspondent aux mesures de la position de la hanche du grimpeur au cours du temps (X en horizontale et Y en vertical), avec $X \in [-50, 400]$ et $Y \in [0, 950]$ environ.

Hip_roll

Le `hip_roll` correspond à l'orientation de la hanche du grimpeur par rapport au mur. Les valeurs sont comprises entre -1.80 et 1.80 en respectant la règle suivante :

- **0** : hanches du grimpeur face au mur
- **$[-1.80, 0]$** : hanches du grimpeur tournée vers la gauche
- **$[0, 1.80]$** : hanches du grimpeur tournée vers la droite
- **-1.80 ou 1.80** : hanches du grimpeur dos au mur

En pratique, les valeurs sont comprises dans $[-0.9, 0.9]$ car le grimpeur n'a pas d'intérêt à tourner ses hanches à 180° du mur.

Neck_roll

De manière analogue, le **Neck_roll** correspond à l'orientation de la nuque du grimpeur par rapport au mur. Les valeurs sont comprises entre -1.80 et 1.80 en respectant la règle suivante :

- **0** : tête du grimpeur face au mur
- **[-1.80, 0]** : tête du grimpeur tournée vers la gauche
- **[0, 1.80]** : tête du grimpeur tournée vers la droite
- **-1.80 ou 1.80** : tête du grimpeur dos au mur

En pratique, les valeurs sont comprises dans $[-0.9, 0.9]$ car le grimpeur n'a pas d'intérêt à tourner sa tête à 180° du mur.

Roll_correl : vecteur de taille m

Ce vecteur contient les valeurs de corrélation entre **Hip_roll** et **Neck_roll** au cours du temps. Ses valeurs sont entre -1 et 1 . Ainsi, si le grimpeur a une rotation de nuque et de hanche identique, la valeur de **Roll_correl** sera de 1 à un certain temps. À l'inverse si le grimpeur oriente sa hanche dans le sens opposé à la nuque, la valeur de **Roll_correl** sera de -1 .

Hip_local_var/ Neck_local_var : vecteurs de taille m

Left_foot/ Left_hand/ Right_foot/ Right_hand : matrices à n lignes et 1 colonne de valeurs entières (comprises entre 0 et 3 inclus)

Chacun de ses vecteurs contient des entiers entre 0 et 3 qui correspond à l'"état" du pied/de la main gauche/droit(e) du grimpeur avec :

- **0** : immobile
- **1** : en exploration
- **2** : en changement
- **3** : en utilisation (traction)

À noter que chaque valeur du vecteur caractérise des valeurs d'observations extérieures (nature de donnée différente) il y a donc des risques d'erreurs. Cependant, étant donnée que l'observateur est un expert en escalade, les potentielles erreurs seront négligées.

Reduced_state : matrice de taille $n \times 1$

Le vecteur **Reduced_state** (de même nature que les vecteurs immédiatement ci-dessus) contient des valeurs de 0 à 4, correspondant à l'état général du grimpeur comme suit :

- **0** : immobile
- **1** : en réglage
- **2** : en changement
- **3** : en traction
- **4** : en exploration

3 Analyse statistique descriptive

3.1 Chargement des données

Afin de pouvoir analyser correctement notre jeu de données, nous avons construit des DataFrames à l'aide de nos fonctions `load_datasets()` et `clean_datasets()` : voir Annexe 1.

On obtient finalement le DataFrame suivant :

```
In [2]: %run import_jeu_donnes.py

In [3]: df
Out[3]:
```

	personne	...	Right_foot
0	P01	...	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...
1	P10	...	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...
2	P02	...	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...
3	P08	...	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...
4	P10	...	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...
..
100	P23	...	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...
101	P03	...	[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, ...
102	P01	...	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...
103	P28	...	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...
104	P25	...	[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, ...

[105 rows x 21 columns]

FIGURE 3 – Dataframe df

3.2 Représentation des données X/Y

Les données X et Y correspondent à la position du grimpeur au cours du temps. Nous avons créé une fonction `printTrajectoires(df)`, qui permet de visualiser la trajectoire d'un grimpeur au cours du temps : voir Annexe 2.

On obtient ainsi par exemple la trajectoire suivante :

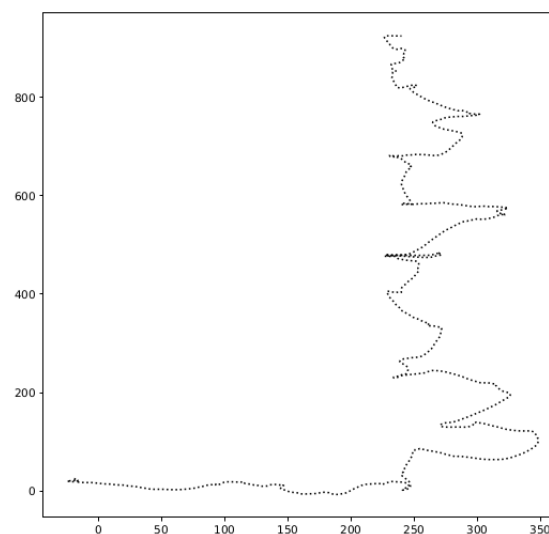


FIGURE 4 – Représentation en coordonnées (X,Y) de la montée de la voie 2 de la personne P06

3.3 Analyse de la donnée Jerk_pos

En comparant les valeurs des jerk de toutes les personnes, on voit que la personne 27 a la plus grande valeur de `jerk_pos`, avec une valeur de l'ordre de 10^{16} contre une valeur de l'ordre de 10^{13} pour la personne avec la plus petite `jerk_pos` pour la personne 6. En regardant les visuels d'escalade des voies au complet comme ci-dessous, on voit effectivement que la personne 27 semble avoir un rythme de grimpe assez saccadé et que la personne 6 quant à elle a une trajectoire bien plus lisse, ce qui est significatif d'un rythme de grimpe plus homogène dans le temps.

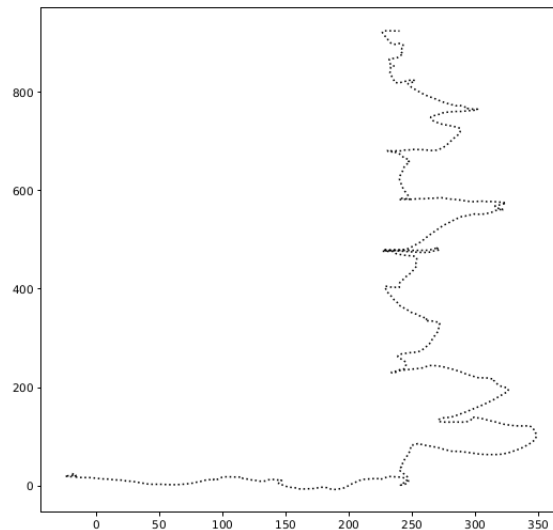


FIGURE 5 – Représentation en coordonnées (X,Y) de la montée de la voie 2 de la personne *P06*

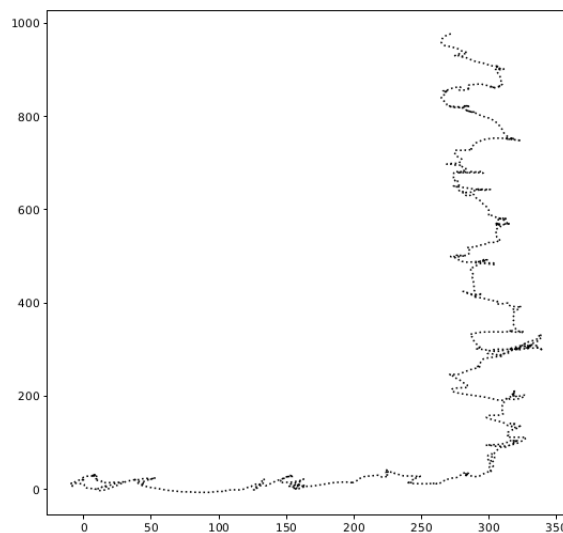


FIGURE 6 – Représentation en coordonnées (X,Y) de la montée de la voie 2 de la personne *P27*

On note une forte corrélation entre la valeur de `jerk_rot` et `jerk_pos`, avec un coefficient de corrélation de **0.99755661**, très proche de 1.

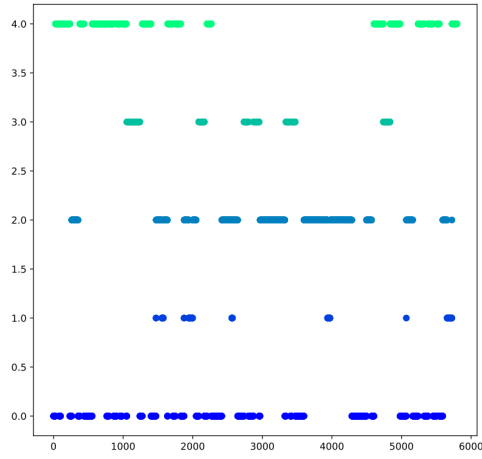
	Jerk_rot	Jerk_pos
personne		
27	3.266514e+17	1.889990e+16
18	1.144933e+17	7.462706e+15
29	5.503709e+16	3.467907e+15
32	3.348915e+16	2.178060e+15
20	2.180884e+16	1.603469e+15
19	1.507913e+16	8.572309e+14
14	9.642080e+15	7.194906e+14
25	4.732380e+15	7.072818e+14
28	1.134732e+16	7.002190e+14
17	9.440483e+15	6.547222e+14
30	5.925487e+15	5.693690e+14
15	8.699937e+15	4.741826e+14
23	4.254431e+15	4.553227e+14
13	6.591653e+15	4.076094e+14
24	3.230882e+15	3.446203e+14
05	4.695017e+15	3.173635e+14
12	2.824526e+15	2.589253e+14
11	3.186968e+15	1.981997e+14
04	2.892400e+15	1.637403e+14
26	3.028841e+15	1.625909e+14
21	1.728450e+15	1.528361e+14
07	3.117004e+15	1.281387e+14

FIGURE 7 – Début du classement des personnes selon la valeur de leur jerk_pos

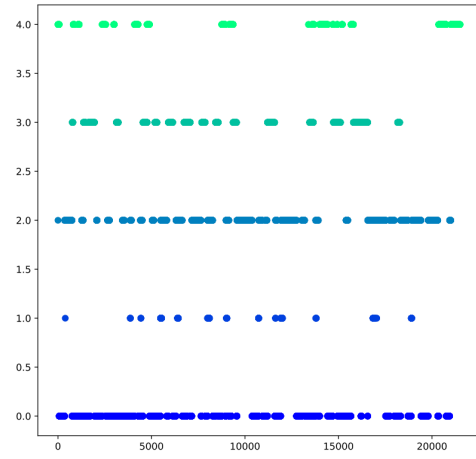
3.4 Analyse des données Reduced_state

Nous avons ensuite décidé d'analyser l'évolution de l'état général du grimpeur à l'aide du champ `Reduced_state`. Nous avons tracé, pour chaque grimpeur, un nuage de points correspondant à son "état" à chaque intervalle de temps (cf. "Description des variables" pour plus de détails quant à l'état du grimpeur correspondant aux valeurs entières), ainsi qu'un histogramme de fréquence : voir Annexe 3.

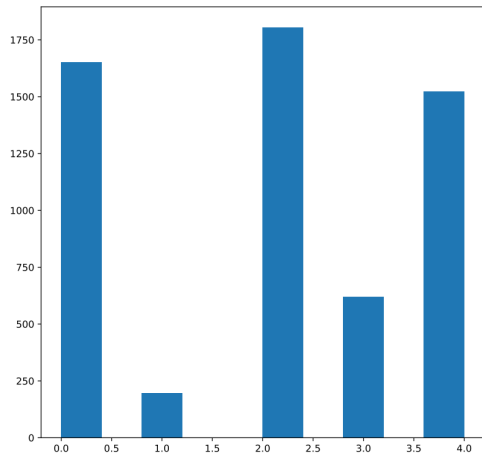
Nous avons choisi d'observer le résultat de la personne 6 et de la personne 27. Il est difficile de faire une conclusion précise. On "sait" que la personne 6 a un nettement meilleur niveau que la personne 27 (grâce au champ `jerk`). D'après les histogrammes de fréquence, il semblerait que le grimpeur expérimenté soit plus "en réglage" que immobile ce qui n'est pas le cas du moins bon grimpeur. Le plus expérimenté est nettement plus en "exploration" que l'autre. Ses observations sont à vérifier avec des données chiffrées plus générales (en observant les moyennes de chaque état pour chaque groupe après une classification par niveau par exemple).



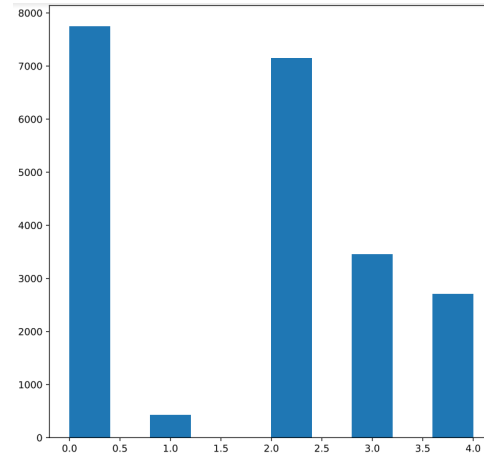
(a) Nuage de points (personne 6)



(b) Nuage de points (personne 27)



(a) Histogramme représentant la proportion des états de la personne 6

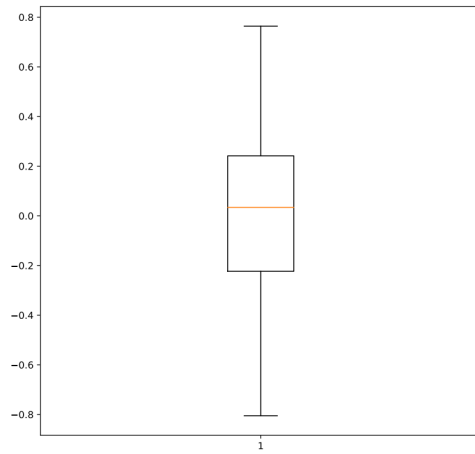


(b) Histogramme représentant la proportion des états de la personne 27

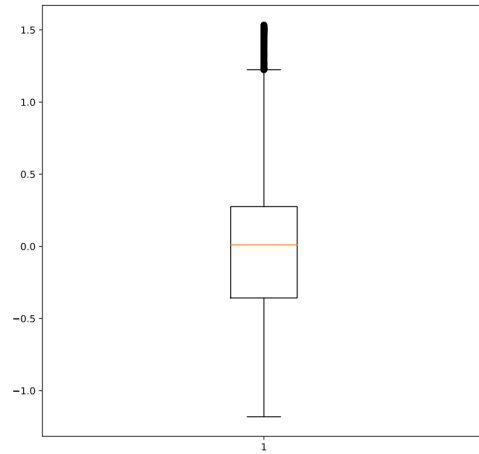
3.5 Analyse des données Hip_roll

Afin de comparer les données `hip_roll` nous avons construits des boîtes à moustaches ainsi que des nuages de points, que nous avons stockés dans les répertoires `Hip_roll` et `Box_hiproll` : voir Annexe 4.

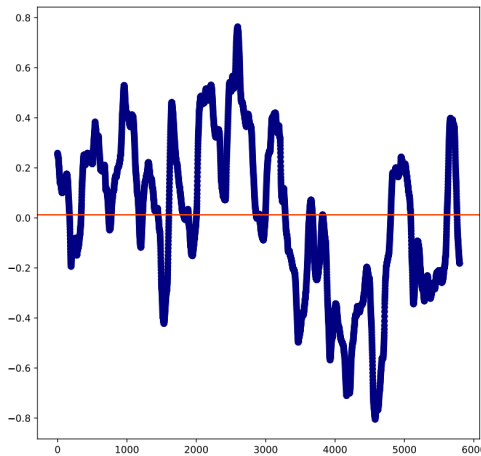
Nous obtenons les résultats suivants pour les personnes 6 et 27 :



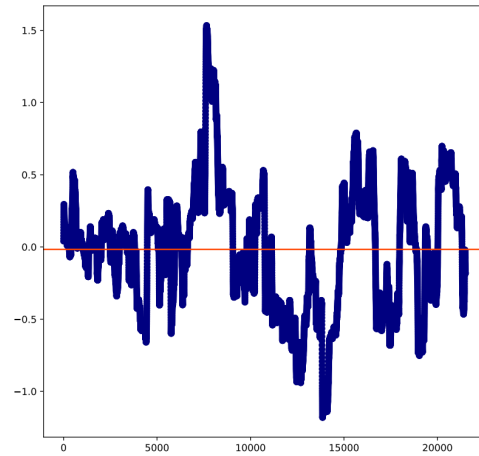
(a) Boîte à moustache `hip roll` (personne 6)



(b) Boîte à moustache `hip roll` (personne 27)



(a) Nuage de points `hipRoll` personne 6



(b) Nuage de points `hipRoll` personne 27

On observe que les valeurs de `hip roll` du moins bon grimpeur (personne 27) sont plus étalées que l'autre. Le meilleur grimpeur (personne 6) pivote donc moins les hanches que le moins bon. Il semblerait aussi que la variation de `hip roll` de la personne 6 soit plus faible.

3.6 Analyse de la vitesse de grimpe

Intuitivement, nous avons pensé que la vitesse pouvait constituer un bon indicateur du niveau d'un grimpeur. Nous avons donc calculé le temps de montée de chaque grimpeur pour chaque voie. Pour ce faire, nous avons récupéré la taille d'un vecteur.

Cependant, il est important de noter qu'il existe principalement "deux types" de relevés. Un relevé plutôt mécanique pour les `X`, `Y`, `*_roll` et un relevé humain (entiers entre 0 et 4 pour décrire l'état du grimpeur) : `*_foot`, `*_hand` et `Reduced_state`. Il y a donc deux types de vecteurs et ils ne font pas tous la même taille. On l'observe ci-dessous :

```

Les tailles des colonnes Left_foot et Reduced_state sont identiques : True
Les tailles des colonnes Left_foot et Right_hand sont identiques : True
Les tailles des colonnes Left_foot et Left_foot sont identiques : True
Les tailles des colonnes Left_foot et Left_hand sont identiques : True
Les tailles des colonnes Left_foot et Afford_count sont identiques : True
Les tailles des colonnes Left_foot et Right_foot sont identiques : True
Les tailles des colonnes Neck_roll et Hip_roll sont identiques : True
Les tailles des colonnes Neck_roll et Neck_roll sont identiques : True
Les tailles des colonnes Neck_roll et Y sont identiques : True
Les tailles des colonnes Neck_roll et X sont identiques : True

```

FIGURE 12 – Taille des vecteurs

Pour comparer la vitesse des grimpeurs on peut donc le faire en regardant la taille d'un vecteur de l'une des deux catégories :

```

df["vitesse"] = df.X.apply(lambda v: v.shape[0])
df["vitesse"] = df.Reduced_state.apply(lambda v: v.shape[0])

```

Nous avons ensuite calculé le coefficient de corrélation du temps de montée et du `jerk_pos` qui est de 0.77, ce qui est cohérent puisqu'il paraît logique qu'un bon grimpeur aille vite dans une voie.

3.7 Dendrogramme

Pour pouvoir analyser correctement les données et faire des déductions quant aux paramètres qui influent sur le niveau, nous avons décidé de faire de la classification. Nous avons d'abord réalisé un dendrogramme en nous servant uniquement de la vitesse de grimpe : voir Annexe 5.

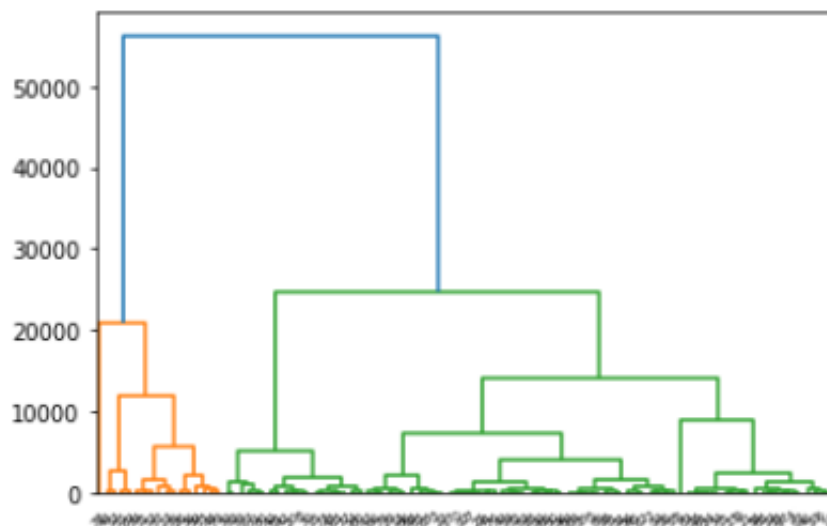


FIGURE 13 – Dendrogramme

Le dendrogramme nous permet de détecter majoritairement deux catégories de grimpeurs.

3.8 Méthode Elbow

La méthode Elbow va nous permettre de déterminer le nombre optimal de cluster à prendre en compte. Cette fois-ci, en plus de la vitesse nous allons prendre en compte l'état général du

grimpeur à l'aide du champ `Reduced_state`. Nous allons compter le nombre de 0, de 1, de 2, de 3 et de 4 (afin de contourner le problème que pose les séries temporelles). Nous créons un nouveau dataframe `_df` : voir Annexe 6.

On va donc pouvoir appliquer la méthode sur ce nouveau dataframe après avoir appliqué une standardisation : voir Annexe 7.

On obtient la figure suivante :

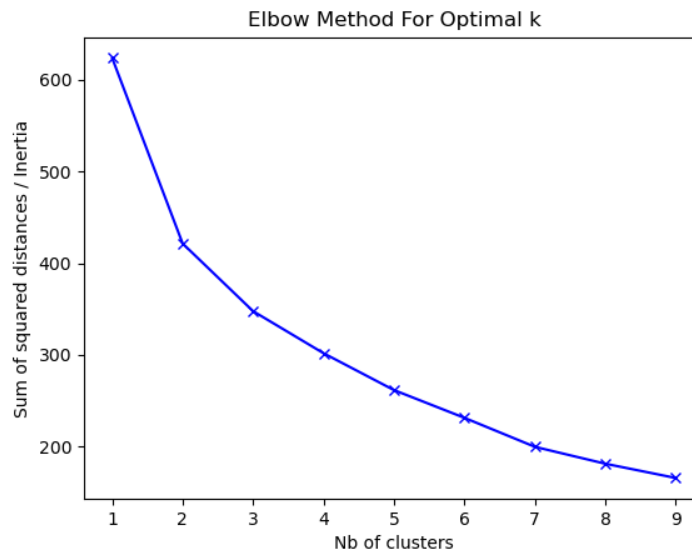


FIGURE 14 – Méthode Elbow

On conclut donc que le nombre de clusters "optimal" est de 3.

3.9 Kmeans

Après avoir effectué la méthode Elbow nous choisissons donc de garder 3 clusters, soit 3 groupes de niveaux. Nous allons donc maintenant pouvoir enfin labeliser nos données, et ce, à l'aide d'un k-means (nous créons une nouvelle colonne "CategorieGrimpeur") : voir Annexe 8.

Nous allons donc procéder à une analyse de données par classe.

3.10 Analyse par Classe

1. `Reduced_state` :

Pour cette donnée nous avons décidé de regarder la part de chaque état lors d'une ascension pour chaque groupe. Voici les résultats :

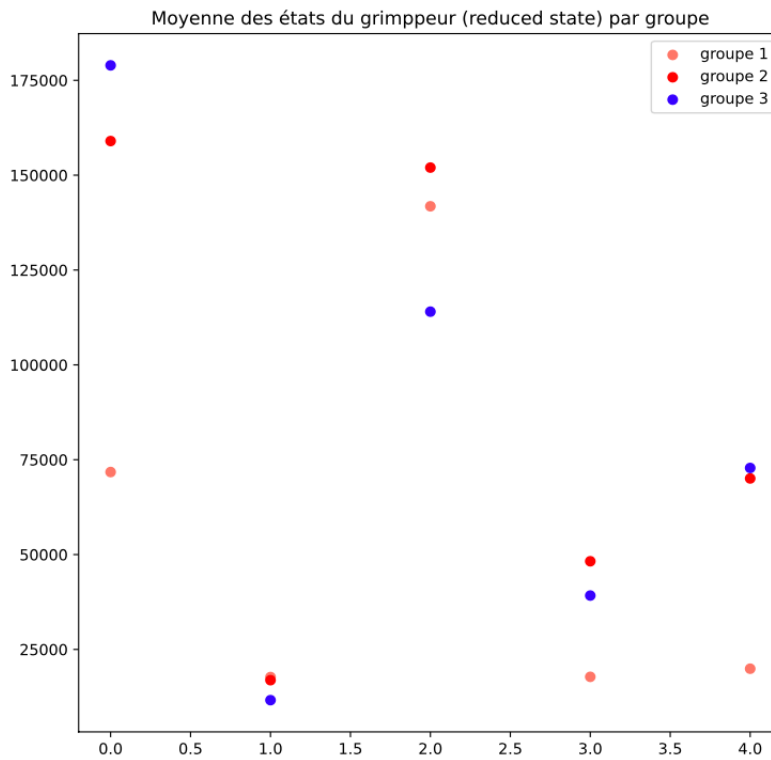


FIGURE 15

Nous pouvons remarquer que le premier groupe est beaucoup moins immobile que les deux autres, et un peu moins en traction et en exploration. Nous pouvons également remarquer que le troisième groupe est plus immobile et moins "en changement" que les autres. Nous pourrions donc penser que les grimpeurs du groupe 3 ont un moins bon niveau, et les grimpeurs du groupe 1, un meilleur niveau.

2. Temps de grimpe :

```
Vitesse moyenne du premier groupe : 9605.75
Vitesse moyenne du deuxième groupe : 8418.622641509433
Vitesse moyenne du troisième groupe : 18114.434782608696
```

FIGURE 16

Nous pouvons remarquer que le temps d'ascension des grimpeurs du groupe 3 est en moyenne très nettement supérieur aux deux autres. Cela rejoint l'hypothèse de conjecture selon laquelle le troisième groupe aurait un moins bon niveau que les autres.

3. Hip roll :

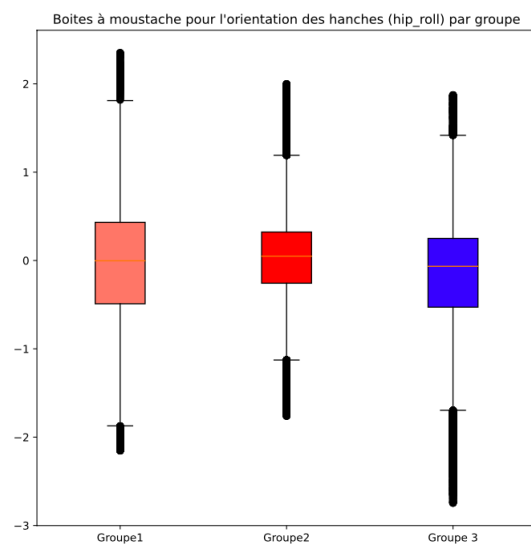


FIGURE 17

Il semblerait, d'après les quartiles, que les grimpeurs du groupe 1 pivotent bien plus les hanches que les autres. Nous pouvons supposer qu'un bon grimpeur est à l'aise à s'orienter comme il le souhaite et donc plus apte à bouger les hanches. Nous pourrions alors penser que le premier groupe a un meilleur niveau que les autres.

4 Réseaux de neurones

4.1 Qu'est ce qu'un réseau de neurones ?

Avant de passer à l'implémentation, nous avons commencé par réaliser une petite étude sur ce qu'est un réseau de neurones et quels sont les existants.

Un réseau de neurones est un modèle mathématique inspiré du fonctionnement du cerveau humain. Il est utilisé pour résoudre des problèmes complexes d'apprentissage automatique, de classification, de régression et de reconnaissance de motifs.

Il est composé de couches de neurones interconnectés. La première couche est appelée couche d'entrée et reçoit les données en entrée. La dernière couche est appelée couche de sortie et produit les résultats finaux. Entre ces deux couches, il peut y avoir une ou plusieurs couches cachées, qui aident à extraire et à représenter les caractéristiques importantes des données.

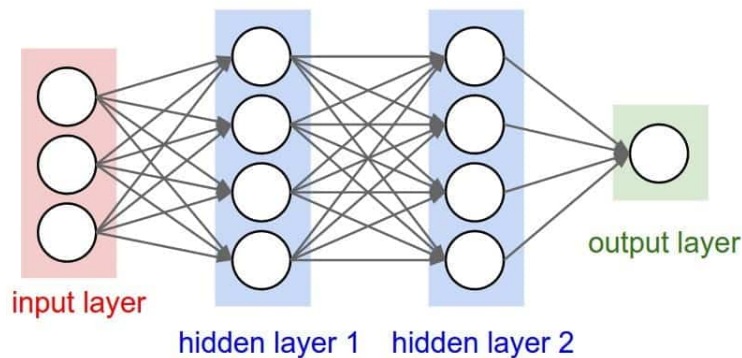


FIGURE 18

Chaque neurone dans le réseau reçoit une ou plusieurs entrées pondérées. Chaque entrée est multipliée par un poids correspondant, qui détermine l'importance de cette entrée pour le neurone. Les poids sont des paramètres ajustables qui sont appris pendant la phase d'entraînement du réseau.

Après la pondération des entrées, chaque neurone applique une fonction d'activation à la somme pondérée de ses entrées. La fonction d'activation introduit une non-linéarité dans le réseau, ce qui lui permet de capturer des relations complexes entre les données d'entrée.

Le signal se propage à travers les couches du réseau de neurones de manière séquentielle. À chaque couche, les neurones calculent leurs activations en fonction des entrées pondérées et des fonctions d'activation. Ce processus se poursuit jusqu'à ce que le signal atteigne la couche de sortie, produisant ainsi une prédiction ou une estimation.

Une fois que la prédiction est générée, une fonction de perte ou de coût est utilisée pour mesurer l'écart entre la prédiction du réseau et la valeur réelle attendue. L'objectif est de minimiser cette erreur pendant la phase d'entraînement du réseau.

Une fois l'erreur calculée, elle est propagée de la couche de sortie vers la couche d'entrée en utilisant un algorithme appelé rétropropagation de l'erreur. Cet algorithme ajuste les poids de manière itérative en fonction de la contribution de chaque poids à l'erreur globale du réseau. La rétropropagation de l'erreur permet d'optimiser les poids du réseau afin de minimiser l'erreur de prédiction.

Le réseau de neurones passe par plusieurs itérations de propagation avant et de rétropropagation de l'erreur pour ajuster progressivement les poids et améliorer les prédictions. Pendant cette phase d'apprentissage, les données d'entraînement sont présentées au réseau, et les poids sont mis à jour en fonction des erreurs calculées.

Une fois le réseau de neurones entraîné, il peut être utilisé pour effectuer des prédictions sur de nouvelles données.

Les réseaux de neurones sont particulièrement efficaces pour traiter des données non linéaires ou hautement complexes, ce qui en fait une technique d'apprentissage automatique puissante pour résoudre des problèmes qui ne peuvent pas être facilement résolus avec des méthodes traditionnelles de programmation.

Il existe plusieurs types de réseaux de neurones, chacun étant conçu pour répondre à des problématiques spécifiques. Nous en présenterons quelques uns plus en détail dans ce qui va suivre. Ces différents types de réseaux de neurones peuvent être combinés pour créer des modèles plus complexes et plus performants pour des tâches spécifiques. Ils peuvent être également sujets à des problèmes tels que le surapprentissage et le biais algorithmique, ce qui nécessite une attention particulière lors de leur utilisation.

4.2 Réseaux de neurones à propagation avant (Feedforward)

Les réseaux de neurones à propagation avant, également connus sous le nom de réseaux de neurones feedforward, sont une classe de réseaux de neurones artificiels qui sont utilisés pour des tâches d'apprentissages supervisés. Ils sont appelés "feedforward" parce que l'information se propage à travers le réseau de manière unidirectionnelle, de l'entrée vers la sortie, sans rétroaction ou boucles de rétroaction.

Chaque neurone dans les couches cachées et de sorties effectue les étapes suivantes :

- Calcule la combinaison linéaire des valeurs d'entrée pondérées par les poids associés à chaque connexion.
- Applique une fonction d'activation à la somme pondérée.
- Transmet la sortie calculée aux neurones de la couche suivante jusqu'à ce que la sortie finale soit générée par la couche de sortie.

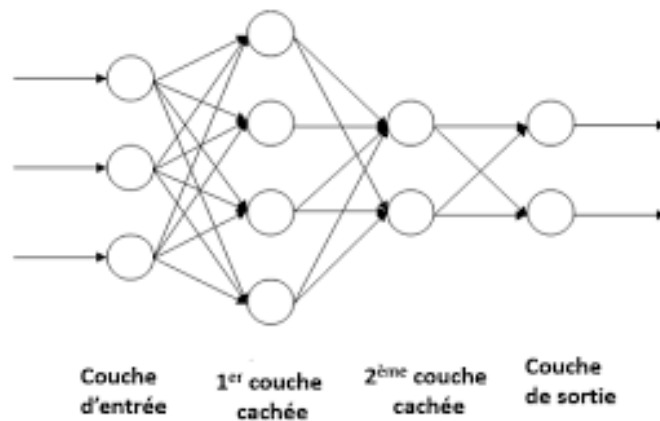


FIGURE 19 – Exemple de réseau de neurones *feed-forward*

4.3 Réseaux de neurones récurrents (Recurrent Neural Networks - RNN)

Les réseaux de neurones récurrents sont couramment utilisés pour la modélisation de séquences telles que le traitement du langage naturel et la reconnaissance de la parole, ainsi que pour la prédiction de séries chronologiques. Contrairement aux réseaux de neurones classiques, les RNN ont une architecture qui leur permet de prendre en compte les informations contextuelles des données séquentielles en utilisant des connexions récurrentes.

La cellule récurrente est la composante fondamentale d'un RNN. Elle maintient un état interne (ou une mémoire) qui est mis à jour à chaque étape temporelle en fonction de l'entrée courante et de l'état précédent. La cellule récurrente permet de capturer et de mémoriser les dépendances à court terme dans la séquence.

À chaque étape temporelle, l'entrée courante est combinée avec l'état précédent pour calculer l'état courant de la cellule récurrente. Cet état courant est ensuite utilisé pour générer la sortie à cette étape temporelle et est également rétropropagé à l'étape suivante pour influencer le calcul de l'état futur.

Les connexions récurrentes permettent aux informations de circuler à travers les différentes étapes temporelles. Cela permet aux RNN de prendre en compte les dépendances à long terme dans les séquences en utilisant les informations des étapes précédentes. Les connexions récurrentes aident également à réduire la quantité de paramètres du modèle, car les mêmes poids sont partagés à chaque étape temporelle.

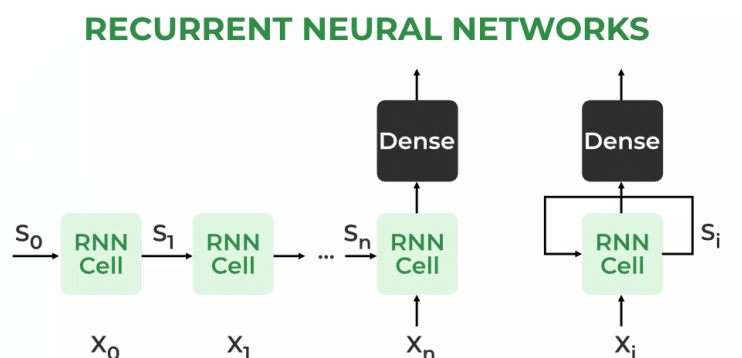


FIGURE 20 – RNN

Bien qu'ils soient très pratiques comparés à une architecture de réseau de neurones classique pour le traitement des données séquentielles, il s'avère qu'ils sont extrêmement difficiles à entraîner pour gérer la dépendance à long terme en raison du problème de la disparition du gradient (Gradient Vanishing). Une explosion de gradient peut également se produire mais très rarement. Pour surmonter ces lacunes, de nouvelles variantes RNN ont été introduites comme le Long short term memory (LSTM) ou réseau récurrent à mémoire court et long terme et le Gated Recurrent Unit (GRU) ou réseau récurrent à portes.

Le LSTM

Le réseau récurrent à mémoire court et long terme est l'architecture de réseau de neurones récurrents la plus utilisée en pratique puisqu'il permet de répondre au problème de disparition de gradient. Il a été proposé par Sepp Hochreiter et Jürgen Schmidhuber en 1997.

Ici, chaque cellule de mémoire est dotée de trois portes : la porte d'entrée (détermine quelles nouvelles informations doivent être stockées dans la cellule de mémoire, elle prend en compte l'entrée courante et l'état caché précédent pour générer un vecteur d'activation qui met à jour la cellule de mémoire), la porte d'oubli (décide quelles informations stockées dans la cellule de mémoire doivent être oubliées ou effacées, elle utilise l'entrée courante et l'état caché précédent pour générer un vecteur de pondération qui contrôle la suppression sélective des informations de la cellule de mémoire) et la porte de sortie. Ces portes contrôlent le flux d'informations dans et hors de la cellule de mémoire, permettant ainsi de réguler la propagation du gradient et de préserver les informations pertinentes sur de longues périodes de temps.

La porte d'entrée et la porte d'oubli sont combinées pour mettre à jour la cellule de mémoire. Les nouvelles informations pertinentes sont ajoutées à la cellule de mémoire en fonction de la porte d'entrée, tandis que les informations obsolètes sont supprimées en fonction de la porte d'oubli.

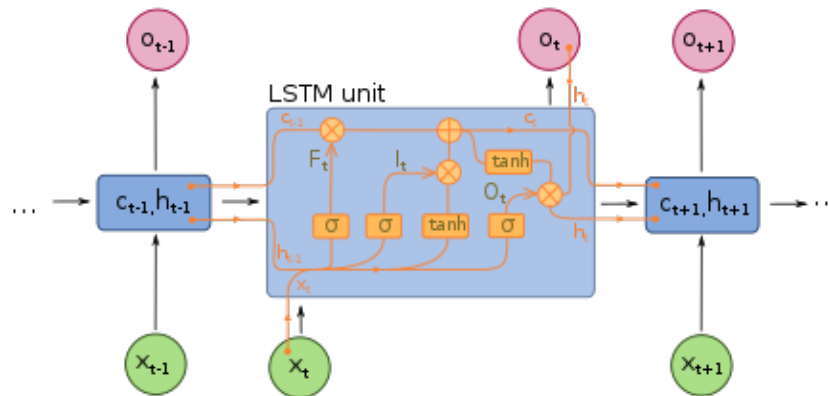


FIGURE 21 – LSTM

Le GRU

Un réseau de neurones récurrents à portes, est une variante des LSTM introduite en 2014 par Kyunghyun Cho et Al. Les réseaux GRU ont des performances comparables aux LSTM pour la prédiction de séries temporelles. Une unité requiert moins de paramètres à apprendre qu'une unité LSTM. Un neurone n'est associé plus qu'à un état caché et les portes d'entrée et d'oubli de l'état caché sont fusionnées (update gate). La porte de sortie est remplacée par une porte de réinitialisation (reset gate).

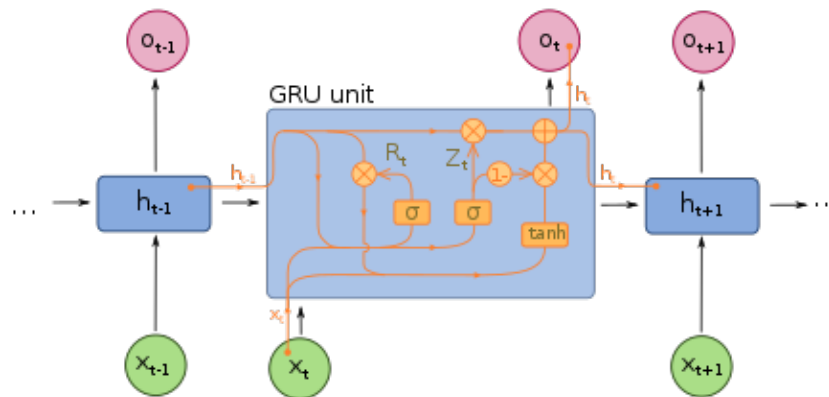


FIGURE 22 – GRU

Remarque : Qu'elle soit détaillée ou simplifiée, la représentation d'un réseau récurrent n'est pas aisée, car il est difficile de faire apparaître la dimension temporelle sur le schéma. C'est notamment le cas pour les connexions récurrentes, qui utilisent l'information du temps précédent. Pour solutionner ce problème, on utilise souvent une représentation du réseau "déplié dans le temps", afin de faire apparaître explicitement celui-ci.

4.4 Auto-encodeurs et auto-encodeurs variationnels

4.4.1 Auto-encodeurs

Un réseau de neurones auto-encodeurs est un type de réseau de neurones artificiels qui apprend à représenter des données d'entrée de manière comprimée en utilisant une structure d'encodage-décodage.

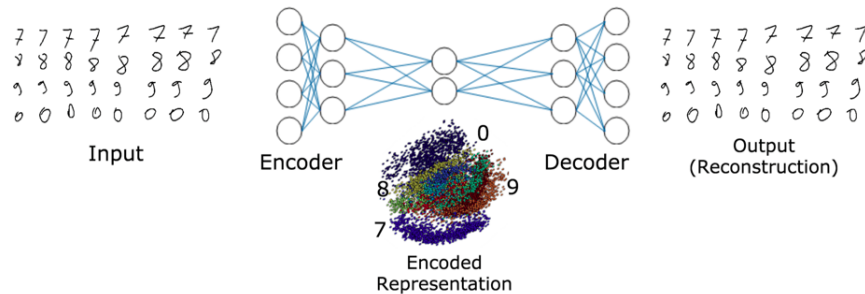


FIGURE 23 – Auto-encodeurs

Un auto-encodeur peut être représenté comme une fonction f qui prend en entrée un vecteur de données \mathbf{x} et renvoie un vecteur de sortie \mathbf{y} de même dimension. Il possède une structure interne (bottleneck - espace latent) implémentée par les couches cachées (le nombre de neurones dans ces couches est inférieur aux nombres de neurones des couches d'entrée/sortie). L'objectif de l'auto-encodeur est de minimiser la différence entre l'entrée et la sortie, également appelée erreur de reconstruction.

Un réseau d'auto-encodeurs est composé de plusieurs couches d'encodeurs et de décodeurs, où chaque couche représente une transformation non linéaire de la couche précédente. Les encodeurs réduisent la dimensionnalité de l'entrée tandis que les décodeurs la restaurent à sa dimensionnalité d'origine. Un auto-encodeur s'entraîne à extraire les parties les plus importantes d'une entrée afin de générer une sortie qui présente moins de descripteurs. En d'autres termes, le réseau ignore le bruit, généralement pour réduire la dimensionnalité de l'entrée. Le réseau entier est formé en minimisant la somme des erreurs de reconstruction pour toutes les couches.

Les auto-encodeurs peuvent être utilisés pour des tâches telles que la réduction de dimensionnalité, la compression de données, la détection d'anomalies et la génération de données. Ils ont également été utilisés avec succès pour la préparation de données en vue d'une utilisation ultérieure dans des modèles d'apprentissage en profondeur plus avancés. A la différence d'un grand nombre de réseaux de neurones, les auto-encodeurs peuvent être entraînés de manière non-supervisée, ce qui permet d'appliquer ces méthodes à des jeux de données non annotés.

4.4.2 Auto-encodeurs variationnels

Un auto-encodeur variationnel (VAE) est un type particulier de réseau d'auto-encodeurs qui ajoute une contrainte de régularisation pour générer des représentations latentes qui suivent une distribution spécifique, généralement une distribution normale multivariée.

La principale utilisation d'un VAE est la génération de données. En effet, en apprenant une représentation latente qui suit une distribution connue, le VAE peut être utilisé pour générer de nouvelles données en échantillonnant à partir de cette distribution. Ces données générées

peuvent être utilisées dans diverses applications telles que la synthèse de données pour l'entraînement de modèles de deep learning, la génération de contenu pour des applications de réalité virtuelle, ou même la création de nouvelles œuvres d'art numériques.

Un autre avantage des VAE est qu'ils permettent également d'interpoler entre des exemples existants dans l'espace latent, ce qui permet de générer des données intermédiaires qui ne sont pas nécessairement présentes dans l'ensemble de données d'origine. Cette fonctionnalité peut être utilisée pour la création de nouvelles images ou de nouveaux sons en combinant des caractéristiques de plusieurs exemples existants.

En résumé, les VAEs sont utilisés pour générer de nouvelles données à partir d'une distribution latente connue, ce qui peut être utile dans de nombreuses applications, notamment la synthèse de données pour l'entraînement de modèles de deep learning, la création de contenu pour la réalité virtuelle et la création artistique.

4.5 Réseaux de neurones convolutifs (Convolutional Neural Networks - CNN)

Les réseaux de neurones convolutifs (Convolutional Neural Networks ou CNN) sont des réseaux de neurones artificiels qui sont particulièrement efficaces pour traiter des données de haute dimensionnalité. Les CNN sont inspirés de la façon dont le cortex visuel du cerveau traite les informations visuelles.

Voici les principales étapes du fonctionnement d'un CNN :

Convolution : La première couche d'un CNN est une couche de convolution, qui applique des filtres (également appelés noyaux ou kernels) sur l'image d'entrée. Chaque filtre extrait une caractéristique spécifique de l'image, telle que les bords, les textures ou les formes. La sortie de la couche de convolution est une carte de caractéristiques.

Fonction d'activation : Après chaque couche de convolution, une fonction d'activation est appliquée à la sortie de la couche pour introduire de la non-linéarité dans le modèle. La fonction d'activation la plus couramment utilisée dans les CNN est la fonction ReLU (Rectified Linear Unit).

Pooling : La couche de pooling réduit la dimensionnalité de la carte de caractéristiques en appliquant une opération de sous-échantillonnage (par exemple, le max-pooling). Cette opération extrait les valeurs les plus importantes de chaque zone de la carte de caractéristiques.

Couches fully connected : Les couches fully connected (ou couches denses) sont similaires à celles des réseaux de neurones traditionnels. Elles relient les sorties de la couche précédente à une sortie finale qui est une prédiction pour la tâche donnée, telle que la classification d'images.

Rétropropagation : Après l'évaluation de la sortie, le modèle utilise la rétropropagation pour ajuster les poids des filtres et des connexions pour minimiser l'erreur de prédiction.

Ces étapes sont généralement répétées plusieurs fois dans le réseau pour améliorer les performances du modèle.

Les réseaux de neurones convolutifs sont spécialement conçus pour travailler sur des données d'images et de vidéos. Ils utilisent des couches de convolution pour extraire des caractéristiques à partir de l'image, suivies de couches de pooling pour réduire la dimensionnalité des caractéristiques extraites. Les réseaux de neurones convolutifs sont également capables d'apprendre des hiérarchies de caractéristiques à partir des images, ce qui leur permet de reconnaître des motifs complexes tels que des formes, des textures et des contours.

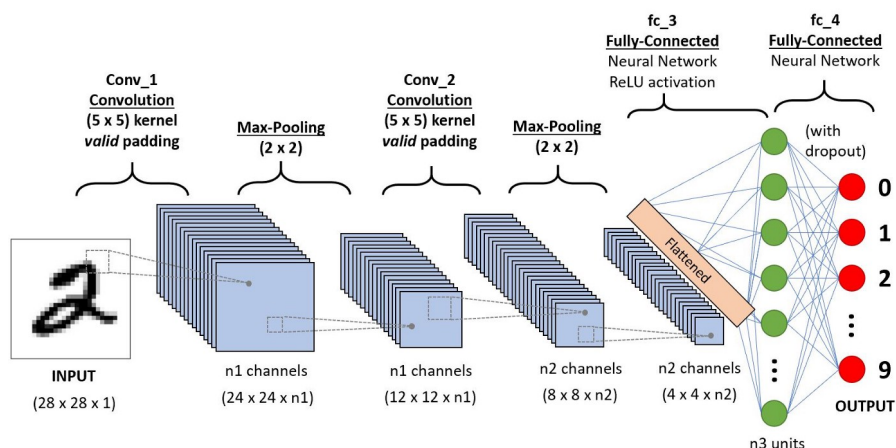


FIGURE 24 – Réseaux de neurones convolutifs

4.6 Réseaux antagonistes génératifs (Generative Adversarial Networks - GAN)

Les réseaux antagonistes génératifs (en anglais, generative adversarial networks ou GAN) sont des modèles de réseaux de neurones non supervisés qui fonctionnent en mettant en concurrence deux modèles de réseaux de neurones. Par exemple, dans le cas du jeu de données en dessous, le premier réseau (appelé le générateur) générerait une image de chiffre. Le second, le discriminateur, reçoit la sortie du générateur ou un exemple d'une véritable image de chiffre. Sa sortie est sa propre estimation de la probabilité que l'entrée qui lui a été fournie provienne d'un exemple créé par le générateur. La décision du discriminateur agit alors comme un signal d'erreur adressé aux deux modèles mais dans des "directions opposées", en ce sens que s'il est certain que sa décision est correcte, cela implique une erreur importante pour le générateur (pour ne pas avoir réussi à tromper le discriminateur), tandis que si le discriminateur a été largement trompé, la perte importante est pour le discriminateur.

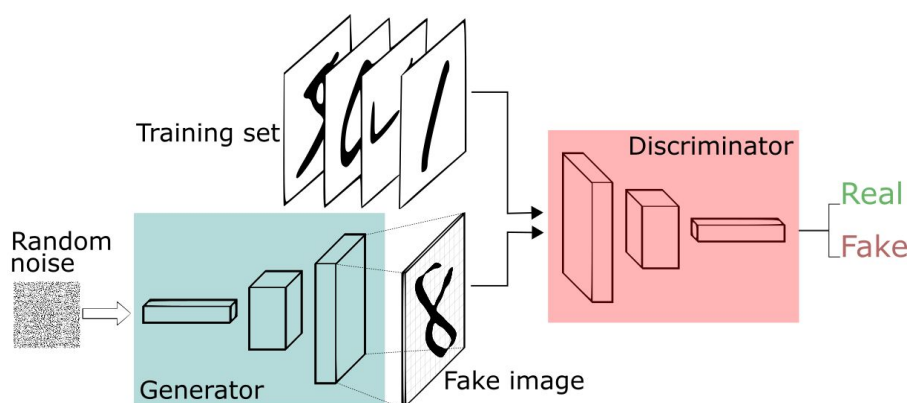


FIGURE 25 – GAN

L'approche d'apprentissage du GAN

Un générateur a été défini pour générer des chiffres manuscrits, un discriminateur pour déterminer si les chiffres manuscrits sont réels ou non, et un jeu de données de chiffres manuscrits réels. Alors comment faire l'entraînement ?

À propos des générateurs

Pour le générateur, l'entrée doit être un vecteur à n dimensions et la sortie une image de la taille du pixel de l'image exemple. Nous devons donc d'abord obtenir le vecteur d'entrée.

Conseils : Le générateur peut être n'importe quel modèle capable de produire des images, comme le réseau neuronal entièrement connecté le plus simple, ou un réseau déconvolutionnel.

Le vecteur d'entrée est considéré ici comme porteur de certaines informations sur la sortie, comme le nombre de chiffres dans l'écriture, le degré de gribouillage de l'écriture, etc. Nous n'avons pas besoin d'informations spécifiques sur les chiffres de sortie, mais seulement qu'ils ressemblent le plus possible aux chiffres manuscrits réels (pour tromper le discriminateur). Nous utilisons donc un vecteur généré aléatoirement comme entrée, où l'entrée aléatoire devrait idéalement satisfaire les distributions courantes telles que la distribution moyenne, la distribution gaussienne, etc.

Astuce : Si nous devons obtenir des nombres de sortie spécifiques par la suite, nous pouvons analyser la sortie générée par le vecteur d'entrée et avoir une idée des dimensions utilisées pour contrôler la numérotation des nombres, c'est-à-dire la sortie spécifique. Souvent, cela n'est pas précisé avant la formation.

À propos des discriminateurs

Il va sans dire que les discriminateurs sont souvent des discriminateurs communs, où l'entrée est l'image et la sortie est l'étiquette d'authenticité de l'image.

Conseils : De même, les discriminateurs, comme les générateurs, peuvent être n'importe quel modèle de discriminateur, comme un réseau entièrement connecté, ou un réseau contenant une convolution, etc.

4.7 Réseau de l'État d'Écho (Echo State Networks - ESN)

4.7.1 Introduction

Un ESN est composé de trois principales composantes : une couche d'entrée, une réserve de neurones appelée "réservoir", et une couche de sortie. La couche d'entrée reçoit les données d'entrée, la réserve de neurones est une couche récurrente non linéaire avec des connexions aléatoires, et la couche de sortie génère les prédictions ou les sorties du réseau.

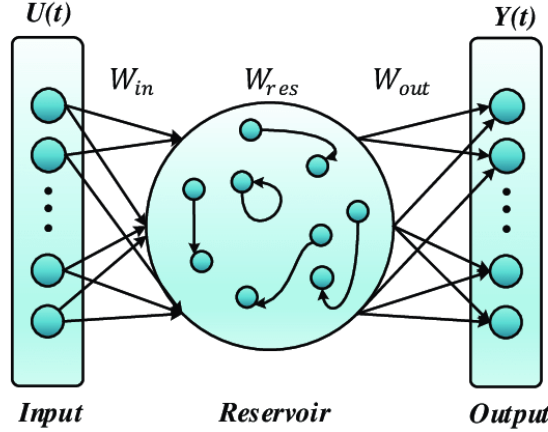


FIGURE 26 – ESN

4.7.2 Equation

À chaque entrée $u(t)$, la réserve met à jour son état, et son équation de mise à jour d'état est la suivante : $x(t+1) = f(W_{in} * u(t+1) + W_{back} * x(t))$

L'équation d'état de sortie de l'ESN est la suivante : $y(t+1) = f_{out}(W_{out}(u(t+1), x(t+1)))$

Le processus d'apprentissage de l'ESN est le processus de détermination de la matrice de poids de connexion de sortie du coefficient W_{out} sur la base des échantillons d'apprentissage donnés. L'apprentissage est divisé en deux phases : la phase d'échantillonnage et la phase de calcul des poids.

4.7.3 Phase d'échantillonnage

La phase d'échantillonnage commence par une sélection arbitraire de l'état initial du réseau, mais généralement l'état initial du réseau est choisi comme étant nul.

(1) Les échantillons d'apprentissage ($u(t)$, $t=1,2,\dots,P$) sont ajoutés à la réserve après la matrice de poids des connexions d'entrée W_{in} .

(2) Le calcul et la collecte de l'état du système et de la sortie $y(t)$ sont effectués tour à tour selon les deux équations d'état précédentes.

Afin de calculer la matrice des poids de connexion de sortie, les variables d'état internes doivent être collectées (échantillonnées) à partir d'un certain moment m et former une matrice $B(P-m+1, N)$ avec des vecteurs comme lignes, tandis que les données d'échantillon correspondantes $y(t)$ sont également collectées et forment un vecteur colonne $T(P-m+1, 1)$.

4.7.4 Phase de calcul des poids

La phase de calcul des poids consiste à calculer les poids de connexion de sortie W_{out} sur la base de la matrice d'état du système et des données collectées pendant la phase d'échantillonnage. Comme la relation entre la variable d'état et la sortie prédite est linéaire, l'objectif à atteindre est d'approcher la sortie désirée $y(t)$, en utilisant la sortie prédite.

4.8 Implémentation avec Tensorflow

Le module keras de tensorflow permet de définir facilement des réseaux de neurones en les décrivant couche par couche. Nous nous concentrerons ici sur le modèle LSTM (réseau récurrent à mémoire court et long terme), couramment utilisé pour la modélisation de séquences. Nous allons définir l'architecture générale d'un tel réseau.

Nous avons du commencer par préparer les données que nous allions fournir au modèle. Nous avons choisis d'entraîner notre réseau de neurones en nous servant des caractéristiques suivantes : "Reduced_state", "Left_foot" et "Hip_roll" et nous avons pris comme label y la colonne "CategorieGrimpeur".

Il a fallu commencer par "padder" (compléter de zéros) nos colonnes caractéristiques afin que ces dernières fassent toutes la même taille puis ensuite les assembler sous la forme d'une liste (X). La fonction np.pad() est une fonction de la bibliothèque NumPy en Python qui permet de réaliser un rembourrage (padding) sur les tableaux multidimensionnels (arrays). Elle permet d'ajouter des valeurs (zéros, constantes ou valeurs spécifiées) autour des bords du tableau existant pour obtenir une taille de tableau souhaitée.

```
df2.X1 = df2.X1.apply(lambda t: np.pad(t, (0, nTimeStep - len(t))))
df2.X2 = df2.X2.apply(lambda t: np.pad(t, (0, nTimeStep - len(t))))
df2.X3 = df2.X3.apply(lambda t: np.pad(t, (0, nTimeStep - len(t))))
df2["X"] = df2.apply(lambda row: np.asarray([row.X1, row.X2, row.X3]),
axis=1)
X = df2.apply(lambda row: np.asarray([row.X1, row.X2, row.X3]).T,
axis=1).to_list()
```

Ensuite, comme nous prévoyons de faire de la classification multi-classe (3 classes de niveau), il a fallu convertir les labels en codage one-hot :

```
y = tf.keras.utils.to_categorical(y, num_classes=3)
```

Nous avons ensuite divisé les données en ensembles d'entraînement et de validation :

```
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2,
shuffle=True, random_state=42)
```

Puis s'est posé un autre problème : convertir les tableaux NumPy en tenseurs TensorFlow.

```
X_train = tf.convert_to_tensor(X_train)
X_val = tf.convert_to_tensor(X_val)
y_train = tf.convert_to_tensor(y_train)
y_val = tf.convert_to_tensor(y_val)
```

Après que les données aient été préparées nous avons pu définir notre modèle LSTM :

```
model = tf.keras.models.Sequential([
    tf.keras.layers.LSTM(6, input_shape=(nTimeStep, 3)),
    tf.keras.layers.Dense(3, activation='softmax')
])
```

Ce modèle est de type Sequential, c'est-à-dire qu'il va être décrit par une suite de couches les unes à la suite des autres. La première est une couche LSTM (Long Short-Term Memory) et la seconde une couche Dense (entièrement connectée). La couche LSTM a 6 neurones et prend une

entrée de forme (nTimeStep, 3), où nTimeStep est le nombre de pas de temps dans une séquence et 3 est la dimension de chaque étape de temps. La couche Dense a 3 neurones et utilise la fonction d'activation Softmax pour produire une sortie de probabilités pour 3 classes différentes.

Pour vérifier que tout va bien jusque là, on peut exécuter la commande `modele.summary()` qui affiche un résumé des couches et du nombre de poids à définir.

Nous compilons maintenant le modèle avec une fonction de perte, un optimiseur et une métrique d'évaluation.

```
model.compile(loss='categorical_crossentropy', optimizer='adam',  
              metrics=['accuracy'])
```

La fonction de perte 'categorical_crossentropy' est couramment utilisée pour la classification multiclasse lorsqu'il y a plus de deux classes. Elle mesure la différence entre les valeurs prédites et les valeurs réelles en utilisant la log vraisemblance et pénalise les prédictions incorrectes.

L'optimiseur 'adam' est un algorithme d'optimisation couramment utilisé dans l'apprentissage en profondeur pour ajuster les poids du modèle en fonction de la fonction de perte. Il utilise une combinaison de méthodes de descente de gradient stochastique et adaptative pour ajuster les poids.

La métrique d'évaluation 'accuracy' mesure la précision globale du modèle sur les données de test en calculant le nombre de prédictions correctes par rapport au nombre total de prédictions.

Nous entraînons le modèle sur les données d'entraînement :

```
history = model.fit(X_train, y_train, epochs=3, batch_size=32,  
                   validation_data=(X_val, y_val))
```

Le modèle est entraîné sur les données X_train et y_train pendant 3 epochs ou itérations, avec une taille de lot (batch_size) de 32.

Le paramètre validation_data est utilisé pour évaluer les performances du modèle sur des données de validation distinctes fournies par les jeux de données X_val et y_val.

Nous évaluons ensuite le modèle sur les données de test :

```
loss, accuracy = model.evaluate(X_val, y_val)
```

La fonction evaluate renvoie deux métriques : la perte (loss) et l'exactitude (accuracy).

La perte est une mesure de l'erreur moyenne du modèle sur l'ensemble de données de validation, calculée en comparant les valeurs prédites du modèle aux valeurs réelles. L'exactitude est une mesure de la précision du modèle sur l'ensemble de données de validation, exprimée en pourcentage.

Les valeurs de perte et d'exactitude renvoyées par `evaluate` peuvent être utilisées pour évaluer la performance globale du modèle sur les données de validation, afin de vérifier si le modèle est en train de sur-entraîner ou de sous-entraîner. Si la perte est faible et l'exactitude est élevée,

cela indique que le modèle est performant sur les données de validation et est probablement généralisable à de nouvelles données.

Dans notre cas, nous obtenons une "accuracy" (précision) de 62%, ce que nous estimions être assez concluant compte tenu du peu que nous n'avions que très peu d'informations pour la labellisation.

```
Epoch 1/3
3/3 [=====] - 27s 9s/step - loss: 1.0979 - accuracy: 0.4458 - val_loss: 1.0941 - val_accuracy: 0.6190
Epoch 2/3
3/3 [=====] - 26s 9s/step - loss: 1.0957 - accuracy: 0.4458 - val_loss: 1.0889 - val_accuracy: 0.6190
Epoch 3/3
3/3 [=====] - 26s 9s/step - loss: 1.0932 - accuracy: 0.4458 - val_loss: 1.0835 - val_accuracy: 0.6190
1/1 [=====] - 1s 821ms/step - loss: 1.0835 - accuracy: 0.6190
```

FIGURE 27 – Accuracy

Mais nous avons constaté un problème lorsque nous avons réalisé des prédictions. En effet, malgré une précision de 62%, nous obtenons des résultats complètement incohérents : Nous sommes censés obtenir un tableau à 3 colonnes où chaque ligne correspond à un grimpeur et contient deux zéros et un un. (Le 1 indiquant l'appartenance à un groupe), Comme suit :

```
[ [0. 1. 0.]
  [0. 0. 1.]
  [1. 0. 0.]
  [0. 1. 0.]
  [1. 0. 0.]
  [0. 1. 0.]
  [0. 1. 0.]
  [0. 1. 0.]
  [0. 0. 1.]
  [0. 0. 1.]
  [0. 1. 0.]
  [0. 0. 1.]
  [0. 1. 0.]
  [1. 0. 0.]
  [0. 1. 0.]
  [0. 0. 1.]
  [1. 0. 0.]
  [1. 0. 0.]
  [0. 1. 0.]
  [0. 1. 0.]
  [0. 0. 1.] ]
```

FIGURE 28 – classement des grimpeurs tests par groupe de niveau

Mais nous obtenons un tableau qui comporte sur chacune des lignes 0.33... sur chaque colonne (ci-après) :

```

[[0.3301807  0.34251386 0.32730538]
 [0.3301807  0.34251386 0.32730538]
 [0.3301807  0.34251386 0.32730538]
 [0.3301807  0.34251386 0.32730538]
 [0.3301807  0.34251386 0.32730538]
 [0.3301807  0.34251386 0.32730538]
 [0.3301807  0.34251386 0.32730538]
 [0.3301807  0.34251386 0.32730538]
 [0.3301807  0.34251386 0.32730538]
 [0.3301807  0.34251386 0.32730538]
 [0.3301807  0.34251386 0.32730538]
 [0.3301807  0.34251386 0.32730538]
 [0.3301807  0.34251386 0.32730538]
 [0.3301807  0.34251386 0.32730538]
 [0.3301807  0.34251386 0.32730538]
 [0.33018073 0.3425139  0.3273054  ]
 [0.33018073 0.3425139  0.3273054  ]
 [0.33018073 0.3425139  0.3273054  ]
 [0.33018073 0.3425139  0.3273054  ]
 [0.33018073 0.3425139  0.3273054  ]]

```

FIGURE 29 – Predictions du niveau des grimpeurs

Ainsi, non seulement notre réseau de neurones renvoie exactement les même sorties pour chaque grimpeur mais en plus ces résultats ne sont pas interprétables. En regardant plus en détail, nous avons finalement remarqué que l'accuracy stagnait au cours des époques ce qui semble vouloir dire que notre réseau n'apprend pas. Cela peut avoir de nombreuses causes : cela peut être dû à la complexité de notre modèle. Si ce dernier est trop complexe par rapport à la quantité de données d'entraînement, il peut conduire à des résultats incohérents. Un modèle complexe peut avoir tendance à sur-ajuster les données et ne pas généraliser correctement. Il est aussi possible que notre modèle ait convergé vers un minimum local de la fonction de perte. Cela peut être dû à une initialisation inappropriée des poids du modèle, un taux d'apprentissage trop élevé ou trop faible, ou un nombre d'epochs d'entraînement insuffisant.

Nous avons donc, par la suite, essayé de modifier de nombreux paramètres pour remédier à ce problème et avoir des résultats interprétables.

1. Fonction d'activation :

Une des premières pistes que nous avons suivi est celle des fonctions de transfert. Effectivement, souvent lorsqu'un réseau n'apprend pas cela peut être du au fait que tous les poids (ou un trop grand nombre) sont nuls. Nous avons donc testé les fonctions "elu" et "leakyrelu" qui semblent plus adaptées notamment car elles permettent d'avoir des valeurs lisses lorsque $x < 0$.

2. Changement nombre de couches, nombre de neurones par couche

Parfois le nombre de couche est insuffisant et le réseau n'est pas assez complexe pour permettre une bonne classification, c'est pourquoi nous avons essayé d'ajouter une puis deux couches LSTM.

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 33980, 10)	560
lstm_1 (LSTM)	(None, 10)	840
dense (Dense)	(None, 3)	33

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 33980, 10)	560
lstm_1 (LSTM)	(None, 33980, 10)	840
lstm_2 (LSTM)	(None, 10)	840
dense (Dense)	(None, 3)	33

FIGURE 30

Le temps d'exécution devient alors très important (jusqu'à 15min) et les résultats ne changent pas, l'accuracy stagne toujours. Pour les tests suivants nous avons gardé le modèle à 3 couches.

3. Standardisation des données

Finalement, nous avons pensé que les données pouvaient avoir besoin d'être "mises à échelle". En effet, les mesures du temps de grimpe ont un ordre de grandeur de 10^5 , les données d'états sont des entiers entre 0 et 4 et la rotation des hanches est comprises entre -1.8 et 1.8. Pour cela, nous avons utilisé la fonction `MaxAbsScaler()` qui permet de standardiser des données qui ne sont pas en répartition normale.

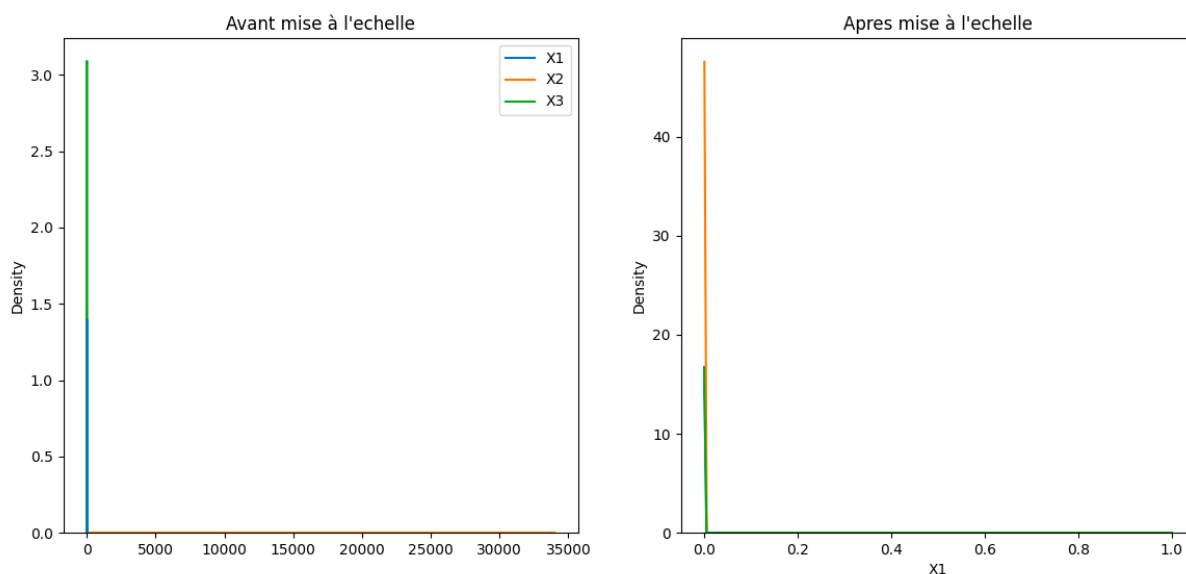


FIGURE 31

Mais malheureusement, même après cela, toujours aucun résultat concluant.

4. Utilisation du temps de grimpe ?

Finalement, nous avons remis en question l'utilisation du temps de grimpe dans le réseau de neurones. En effet ces données ne sont pas des séries temporelles et les couches LSTM traitent les dépendances au temps. Nous nous sommes donc dit que cela pouvait poser problème.

A) Nous avons fait un réseau de neurone avec séries temporelles.

B) Un réseau de neurones sans séries temporelles Au final, les résultats sont les mêmes, l'accuracy continue à stagner et nous ne comprenons pas pourquoi.

A partir de ce moment la nous avons réalisé tous nos tests avec en entrée le temps de grimpe et le jerk. Le réseau de neurones est le suivant :

Layer (type)	Output Shape	Param #
dense (Dense)	(None, None, 30)	90
dense_1 (Dense)	(None, None, 30)	930
dense_2 (Dense)	(None, None, 3)	93

FIGURE 32 – modèle utilisé pour les tests suivants

Pour essayer de comprendre mieux nous avons essayé d'analyser l'évolution de l'accuracy et de la fonction de perte au cours des époques.

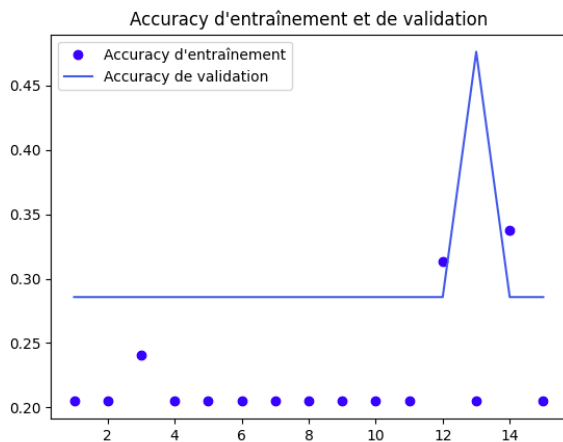


FIGURE 33 – 15 époques, 10 neurones sur les couches cachées (précision)

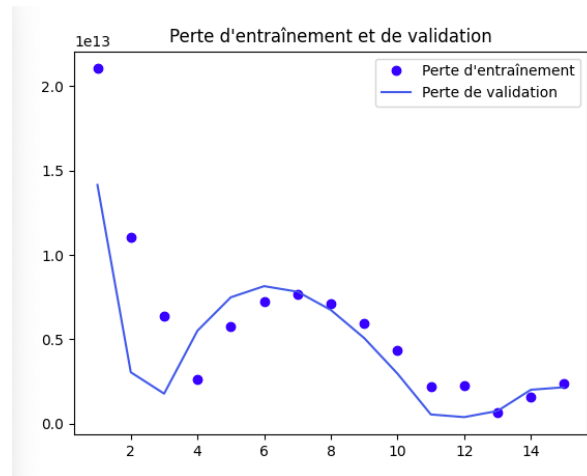


FIGURE 34 – 15 époques, 10 neurones sur les couches cachées (perte)

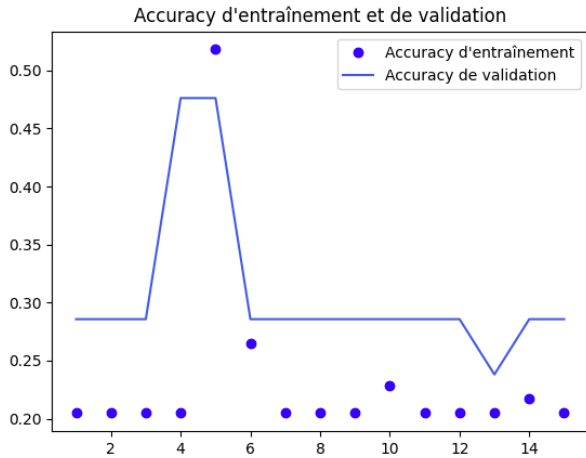


FIGURE 35 – 15 époques, 30 neurones sur les couches cachées (précision)

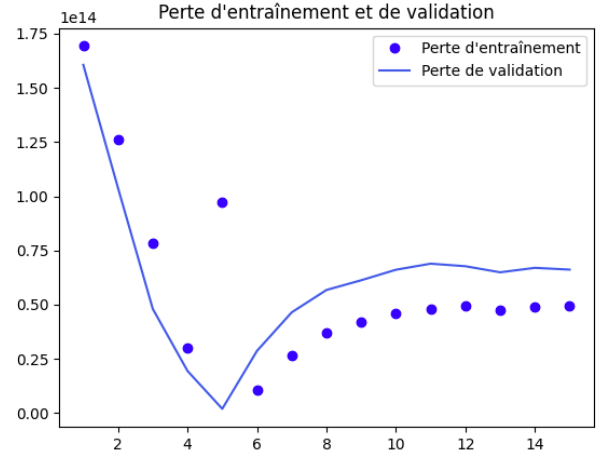


FIGURE 36 – 15 époques, 30 neurones sur les couches cachées (perte)

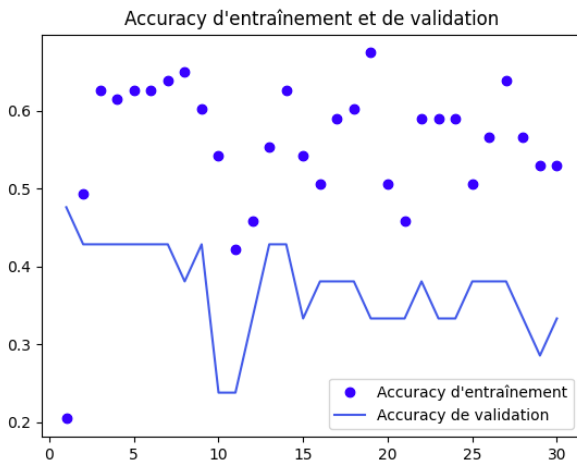


FIGURE 37 – 30 époques, 30 neurones sur les couches cachées (précision)

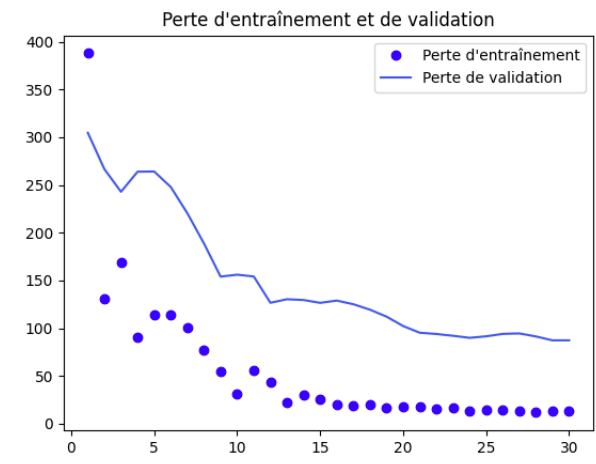


FIGURE 38 – 30 époques, 30 neurones sur les couches cachées (perte)

Nous avons fait varier le nombre d'époques ainsi que le nombre de neurones sur la couche cachée. Nous pouvons constater que, pour chacun des tests, l'accuracy d'entraînement est supérieure à l'accuracy de validation et que la perte d'entraînement est inférieure à celle de validation. Cela traduit parfois un phénomène d'overfitting, seulement, comme cette observation est valide dès la première époque, il semblerait que le problème se situe autre part. D'autre part, l'accuracy semble stagner ou régresser, ce que nous avons du mal à expliquer.

5. Changement de la répartition entraînement/test

Nous avons utilisé 80% des données pour la phase d'entraînement et 20% pour le test. Nous avons essayé de faire 90-10 puis 70-30.

Il semble que le résultat le plus concluant soit pour la répartition 70-30 :

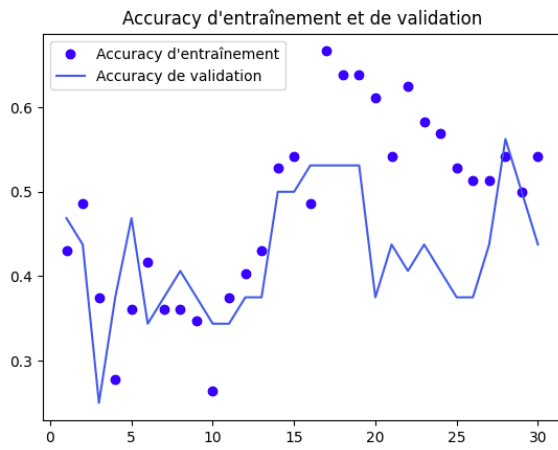


FIGURE 39 – 15   poques, 10 neurones sur les couches cach  es (pr  cision)

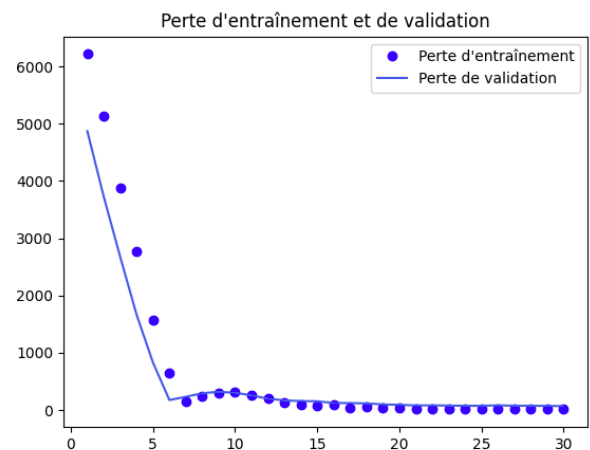


FIGURE 40 – 15   poques, 10 neurones sur les couches cach  es (perte)

L'accuracy est de 44% et le tableau de r  sultat se rapproche plus de ce que l'on attend :

```
[[1.0009147e-05 9.9998993e-01 0.0000000e+00]
 [9.1752530e-24 9.9999994e-01 0.0000000e+00]
 [9.9775743e-01 2.2424902e-03 0.0000000e+00]
 [2.1261290e-21 9.7114211e-01 2.8857861e-02]
 [5.2017826e-06 9.9999470e-01 0.0000000e+00]
 [3.3583332e-15 9.9999994e-01 0.0000000e+00]
 [9.9862033e-01 1.3795918e-03 0.0000000e+00]
 [9.9999994e-01 6.3708742e-25 2.7408628e-10]
 [0.0000000e+00 9.9999994e-01 0.0000000e+00]
 [0.0000000e+00 9.9999970e-01 2.4105620e-07]]
```

FIGURE 41 – extrait du tableau des r  sultats de sortie

Nous avons ainsi pu construire un tableau de r  sum   de la classification par notre r  seau :

Tableau de comparaison			
	G1	G2	G3
G1	1	7	0
G2	4	9	2
G3	0	5	4

FIGURE 42 – R  sultats obtenus

Nous pouvons remarquer qu'il y a beaucoup d'inversion groupe 1-groupe 2 pour la classification. Ce n'est pas forc  ment   tonnant, car d'apr  s l'analyse de donn  es les groupes 1 et 2 nous semblent assez proches en terme de niveau. Par contre, il y a une grosse confusion entre le groupe 2 et le groupe 3, ce que nous n'attendions pas forc  ment.

Malheureusement, nous avons manqu   de temps et n'avons pas r  ussi    comprendre d'o   pouvait provenir notre erreur.

5 Conclusion

En conclusion, ce projet a été une expérience enrichissante dans l'application des techniques d'apprentissage statistique à un domaine spécifique, en l'occurrence l'escalade. Nous avons pu mettre en œuvre une analyse descriptive approfondie pour comprendre les caractéristiques clés des données et les facteurs influençant les performances des sportifs. L'utilisation d'un réseau de neurones LSTM a permis d'explorer les capacités de prédiction du modèle et de capturer les dépendances temporelles dans les données.

Bien que nous ayons rencontré des défis lors de l'analyse des résultats et de l'entraînement du modèle, cela nous a offert une occasion d'apprendre et de comprendre les nuances de l'apprentissage automatique. Nous avons appris à travailler avec des bibliothèques telles que TensorFlow pour construire et entraîner des réseaux de neurones, en adaptant les modèles à nos besoins spécifiques.

Ce projet nous a également sensibilisés à l'importance d'une étape préliminaire d'analyse statistique approfondie pour mieux comprendre les données avant de les utiliser dans des modèles d'apprentissage automatique. Cette compréhension préalable des données nous a aidés à prendre des décisions éclairées tout au long du processus d'analyse et de modélisation.

En somme, ce projet a été une expérience pratique et immersive, nous permettant d'acquérir de nouvelles compétences en matière d'apprentissage automatique, d'analyse statistique et de manipulation de données. Il nous a également donné un aperçu du potentiel des réseaux de neurones LSTM dans la prédiction de performances sportives. Nous sommes confiants que les connaissances et les compétences acquises dans ce projet nous seront utiles dans nos futurs travaux d'analyse de données et de modélisation prédictive.

6 Annexe

Annexe 1 : page 6

```
def load_datasets():
    rows = []
    for filename in glob('data/G*.mat'):
        monDict = {
            "personne": filename[8:11],
            "s": filename[12:14],
            "v": filename[15:17],
            "e": filename[18:20],
            "date": filename[21:29],
        }
        monDict.update(loadmat(filename))
        rows.append(monDict)
    return DataFrame(data=rows)

def clean_datasets(df):
    df.drop(columns=['__header__', '__version__', '__globals__'],
            inplace=True)
    df.Jerk_rot = df.Jerk_rot.apply(lambda v: v[0,0])
    df.Jerk_pos = df.Jerk_pos.apply(lambda v: v[0,0])
    for fieldname in ['Reduced_state', 'Neck_local_var',
                     'Hip_local_var', 'Hip_roll', 'Right_hand', 'Neck_roll', 'Left_foot',
                     'Y', 'X', 'Roll_correl', 'Left_hand', 'Afford_count', 'Right_foot']:
        # Conversion matrice -> vecteur colonne
        df[fieldname] = df[fieldname].apply(lambda v: v.reshape(-1))
    return df
```

Annexe 2 : page 6

```
def printTrajectoires(df):
    for idx, row in df.iterrows():
        v, s, e = row.v, row.s, row.e
        fig, ax = plt.subplots(figsize=(8.26, 8.26))
        X, Y = row.X, row.Y
        ax.plot(X, Y, ':k')
        plt.savefig(f'./img/traj_{idx}_{s}_{v}_{e}.pdf',
                    bbox_inches='tight')
        # plt.show()
        plt.close(fig)
```

Annexe 3 : page 8

```
# Analyse de Reduced_state par histogramme de frequence et
# nuage de points
def printReducedState(df):
    for idx, row in df.iterrows():
        p, v, s, e = row.personne, row.v, row.s, row.e

        Y = row.Reduced_state
        X = np.arange(len(Y))
```

```
fig, ax = plt.subplots(figsize=(8.26, 8.26))
ax.scatter(X, Y, c=Y, cmap = plt.cm.Set1, marker = 'o')
plt.savefig(f'./reduced_state/{p}_{s}_{v}_{e}.pdf',
bbox_inches='tight')
plt.close(fig)
```

```
fig, ax = plt.subplots(figsize=(8.26, 8.26))
ax.hist(Y)
plt.savefig(f'./hist/{p}_{s}_{v}_{e}_hist.pdf',
bbox_inches='tight')
plt.close(fig)
```

Annexe 4 : page 9

```
def Hip_roll_nuages(df):
    for idx, row in df.iterrows():
        p, v, s, e = row.personne, row.v, row.s, row.e

        Y = row.Hip_roll
        X = np.arange(len(Y))

        fig, ax = plt.subplots(figsize=(8.26, 8.26))
        ax.scatter(X, Y, marker = '.', c = 'navy')
        ax.axhline(y = np.mean(Y), c = 'orangered')
        plt.savefig(f'./Hip_roll/{p}_{s}_{v}_{e}.pdf',
bbox_inches='tight')
        plt.close(fig)

        fig, ax = plt.subplots(figsize=(8.26, 8.26))
        ax.boxplot(Y)
        plt.savefig(f'./Box_hiproll/{p}_{s}_{v}_{e}_Box_hiproll.pdf',
bbox_inches='tight')
        plt.close(fig)
```

Annexe 5 : page 11

```
df["vitesse"] = df.X.apply(lambda v: v.shape[0])
Z = linkage(df.vitesse.to_frame(), method='ward', metric='euclidean')
d = dendrogram(Z, orientation='top', labels=df.index,
show_contracted=True, no_plot=False,
leaf_font_size=6, leaf_rotation=45.)
plt.show()
```

Annexe 6 : page 12

```
for i in range(5):
    _df[f"Reduced_state{i}"] = df.Reduced_state.apply(lambda v:
len([x for x in v if x == i]))

_df = _df[["vitesse", "Reduced_state0", "Reduced_state1",
"Reduced_state2", "Reduced_state3", "Reduced_state4"]].copy(deep=True)
```

Annexe 7 : page 12

```

scaler = StandardScaler()
_df = scaler.fit_transform(_df)

Sum_of_squared_distances, K = [], range(1, 10)
for nclusters in K:
    kmeans = KMeans(n_clusters=nclusters)
    kmeans.fit(_df)
    Sum_of_squared_distances.append(kmeans.inertia_)

plt.plot(K, Sum_of_squared_distances, "bx-")
plt.xlabel("Nb_of_clusters")
plt.ylabel("Sum_of_squared_distances_/_Inertia")
plt.title("Elbow_Method_For_Optimal_k")
plt.show()

```

Annexe 8 : page 12

```

kmeans = KMeans(n_clusters=3, n_init=50, random_state=0)
kmeans.fit(_df)
df["CategorieGrimpeur"] = kmeans.labels_
print(len(kmeans.labels_))
print(df.groupby(by="CategorieGrimpeur").vitesse.count())

```


7 Bibliographie

- *Culture physique - L'escalade et le jerk* - Arte.tv
<https://www.arte.tv/fr/videos/100100-007-A/culture-physique/ire>
- «*Generative Adversarial Networks* » : théorie et pratique
<https://www.theses.fr/2021SORUS051.pdf>
- *Explorations in Echo State Networks*
https://www.ai.rug.nl/~mwiering/Thesis_Adrian_Millea.pdf
- *Réseaux de neurones récurrents pour le traitement automatique de la parole*
<https://www.theses.fr/2017SACLS295.pdf>
- *Apprentissage de séquences et extraction de règles de réseaux récurrents : application au traçage de schémas techniques.*
<https://www.theses.fr/2018BORD0032.pdf>
- *Gate-Variants of Gated Recurrent Unit (GRU) Neural Networks*
<https://arxiv.org/pdf/1701.05923.pdf>
- *Approches neuronales pour le résumé abstraktif de transcriptions de parole*
<https://www.theses.fr/2021LEMA2022.pdf>
- *Spécification et compilation de réseaux de neurones embarqués*
https://theses.hal.science/tel-03997036/file/POMPOUGNAC_Hugo_these_2022.pdf