Aurélia Autem Gibran Chevalley

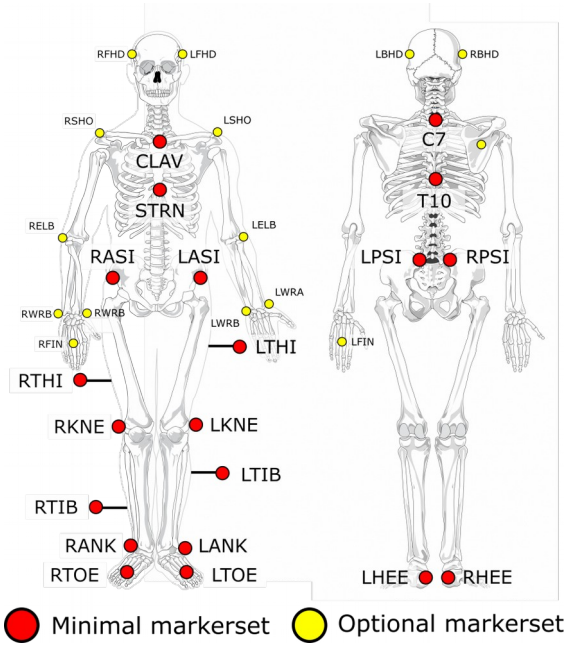
Projet SOFAMEHACK

Sommaire

I. Introduction

Ce travail vise à répondre au problème lancé par le challenge SOFAMEHACK 2019. Le but est de développer un algorithme pour détecter des évènements liés à la marche, comme le pied gauche/droit est posé ou le pied gauche/droit est levé.
Dans un milieu médical, ce type d'outils permettrait l'étude de la marche de patients malades et faciliterait le travail des médecins.

Pour détecter les différents étapes de la marche, nous disposons de données récoltées par les HUG qui sont récupérées via des capteurs sur le corps et des plaques de pression sur le sol qui détecte si un pied se pose ou se lève. Voici l'ensemble des capteurs possibles :

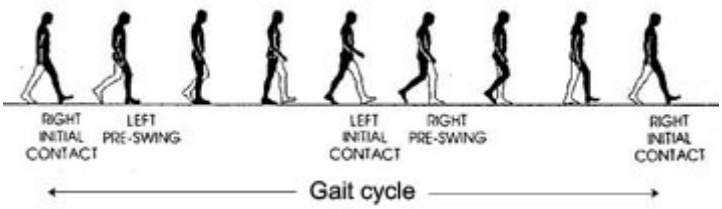


Les données liées aux capteurs et aux plaques de pression sont encodés dans des fichiers c3d.

Pour les plaques de pression, les évènements sont encodés selon deux paramètres :
Foot_Strike / Foot_Off et Right / Left.

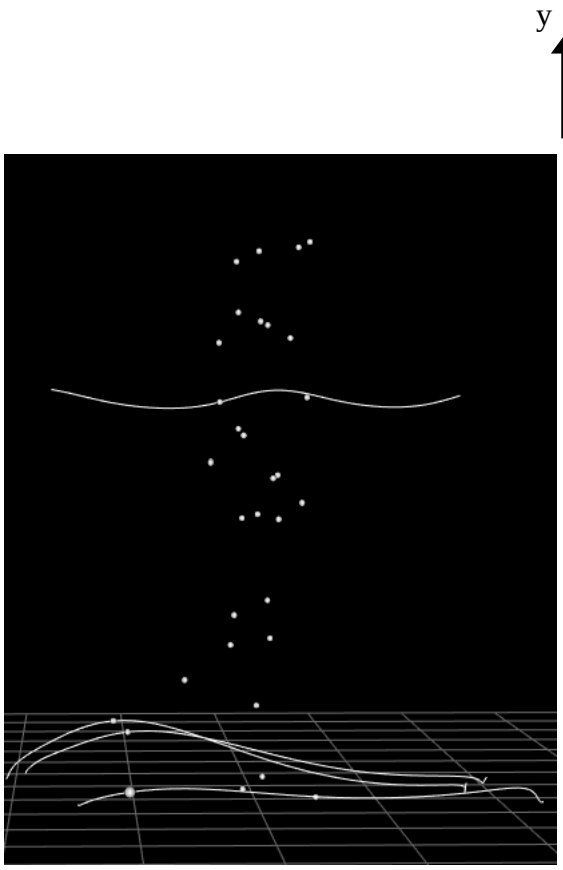
Source : http://sofamehack2019.sofamea.org/data

Le but de ce projet est donc d'utiliser ces données pour automatiquement décrire les différentes étapes de la marche.



Source : https://www.utdallas.edu/atec/midori/Handouts/walkingGraphs.htm

II. Traitement des données



On peut visualiser les données via un logiciel Mokka. Sur l'image ci-contre, on peut voir les capteurs (points blancs) et le tracé de certains d'entre eux (les points LHEE, LANK, LTOE et le point T10).

Grâce à ce logiciel, nous avons pu visualiser les capteurs, les évènements et autres informations liés aux fichiers c3d.

Dans un premier temps, nous avons décidé de faire une sélection parmi les capteurs proposés afin de conserver seulement les données les plus significatives et faciliter l'apprentissage de nos algorithmes. Pour la sélection, comme on pouvait le penser, les capteurs proches des pieds révèlent bien les mouvements de la marche, on voit aussi que certains points au niveau du thorax (T10 / C7) peuvent apporter des informations complémentaires.
Ainsi, notre premier choix de capteurs s'est porté sur : LHEE, LANK, LTOE, RHEE, RANK, RTOE et T10.

Ensuite, il nous fallait transformer les données sous forme de vecteurs afin de les utiliser dans nos algorithmes. Grâce au format c3d, nous pouvons récupérer les coordonnées en x, y, z pour chaque frame et les frames où sont présents les évènements. Nous avons décidé de garder les coordonnées x, y et z pour chaque capteurs choisis.

⇒ $[x_1, y_1, z_1, \dots, x_m, y_m, z_m]$ pour une frame, en supposant qu'on ait m capteurs

Par la suite, nous avons remarqué que x ne fait que croître au fil du temps et pour faciliter l'apprentissage, nous avons essayé de soustraire à chaque point en x la valeur d'un point du thorax (T10) en x. Sur Mokka, on observe que cela revient à marcher sur place et diminue l'importance de la distance parcourue avant un évènement.

⇒ $[x_1 - \mathbf{x_{10}}, y_1, z_1, \dots, x_m - \mathbf{x_{10}}, y_m, z_m]$

Enfin, la dernière étape pour le traitement des données est de labelliser chaque instance. Pour cela, nous avons pensé à deux méthodes différentes : sparse et dense.

i. Méthode « sparse »

Pour la méthode « sparse », l'idée consiste à extraire les données, comme expliqué ci-dessus, qui représentent un évènement dans notre fichier c3d.
Ensuite, on donne un nombre pour chaque évènement possible :

Foot_Strike – Left	→	1
Foot_Off – Left	→	2
Foot_Strike – Right	→	3
Foot_Off – Left	→	4

Enfin, pour que notre algorithme reconnaisse aussi les frames qui ne contienne pas d'event, on extrait autant de frames avec event que sans event et on labellise les frames sans event avec 0.

ii. Méthode « dense »

Suite à la méthode « sparse », nous avons pensé qu'il serait mieux de garder la continuité des frames. Nous avons donc pensé à la méthode « dense », cette fois, nous gardons toutes les frames d'un fichier depuis le premier évènement jusqu'au dernier et on les labellise :

Pied droit posé	→	0
Pied gauche posé	→	1
Deux pieds posés	→	2

En général, le but dans le choix de la représentation des données est aussi d'avoir environ autant d'instances de chaque classe car si on a une classe prédominante, un algorithme pourrait juste classer toutes les instances avec cette classe et quand même obtenir une accuracy correcte.

III. Approches du problème

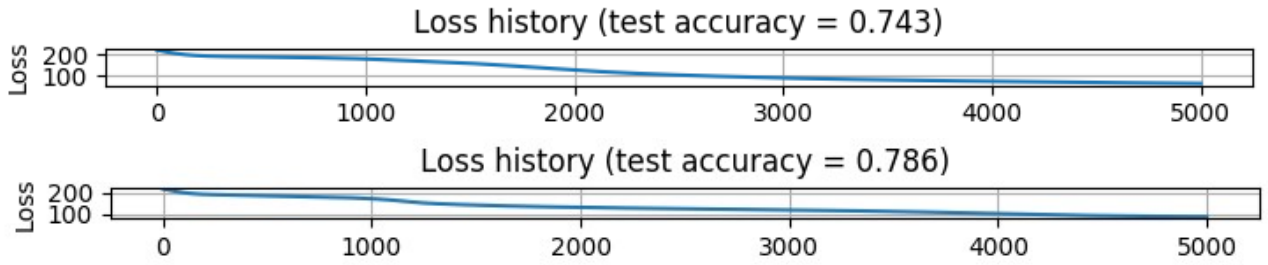
i. Réseaux de neurones simples

Méthode sparse

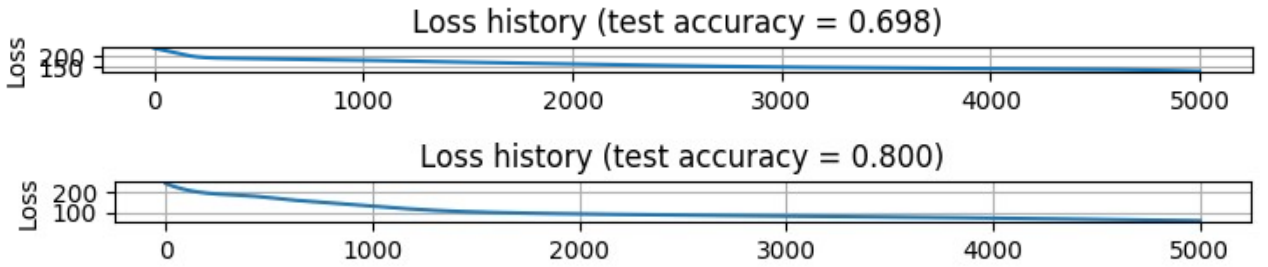
La première méthode que nous avons implémenté utilise un réseau de neurone simple et la représentation des données *sparse*.
Dans un premier temps, nous avons séparé tous les fichiers en un training set et un testing set avec un ratio 2/3 et 1/3. Nous avons séparer les fichiers séparément pour une même pathologie afin que les données soient uniformément réparties. Ensuite, nous avons labellisé toutes les frames de chaque vidéo du training set avec la représentation *sparse* des données.

Pour entraîner notre réseau, nous avons utilisé un certain nombre d'hyperparamètres comme le nombre de hidden layers, la taille de chaque layer ou le fait des soustraire ou non la coordonnée X. On pourrait aussi en rajouter d'autres comme les labels que l'on fournit au réseau.
Afin de choisir les hyperparamètres, on utilise la cross-validation, c'est-à-dire que l'on sépare le training set en 10 parties et on teste les différents paramètres sur ces ensembles. Pour finir, on choisit le meilleur et on calcule l'accuracy avec le test set.

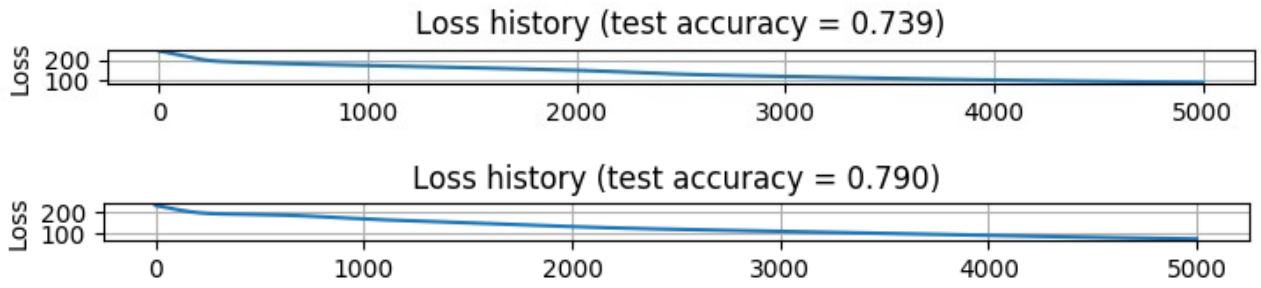
Lors de la cross-validation nous avons obtenu les résultats suivants :
(test_accuracy représente la moyenne des tests effectués pour la validation)



Premier exemple pour un réseau avec une couche cachée contenant 5 cellules. En haut, en soustrayant X et en bas, sans soustraire X.



Deuxième exemple, pour un réseau avec deux couches cachées la première avec 5 cellules et la deuxième avec 15 cellules. En haut, en soustrayant X et en bas, sans soustraire X.



Troisième exemple, pour un réseau avec trois couches cachées avec 8 cellules chacune. En haut, en soustrayant X et en bas, sans soustraire X.

Ensuite, l'algorithme repère le meilleur modèle d'après la cross-validation et teste ce modèle avec le testing set qui contient des données que le modèle n'a jamais vu.

```
Test accuracy for the best model :  0.6292134831460674
```

Conclusion :

On remarque que les loss histories sont décroissantes et se rapprochent de zéro vers la fin, ce qui est un comportement assez normal.
On a aussi vu que les accuracies sont assez proches les unes des autres donc on peut se demander si notre réseau ne choisit pas toujours la même classe, qui serait prédominante parmi toutes les classes. Néanmoins, avec plusieurs tests, on a remarqué que le réseau choisit des classes différentes donc on peut écarter le problème. On peut donc conclure que les tailles choisies pour les hyperparamètres sont toutes correctes.

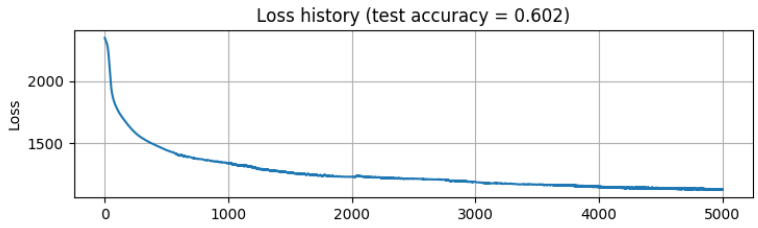
On a aussi remarqué que entre différentes exécutions de training, la validation accuracy pouvait varier d'environ 0.10 points.

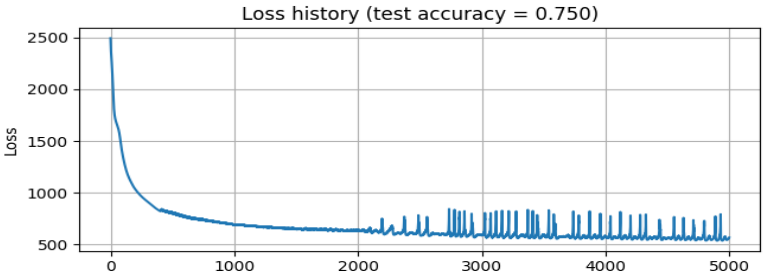
Concernant la test accuracy finale, on voit qu'elle est plus faible que la validation accuracy. On peut donc se demander si on n'a pas une situation d'overfitting.

Méthode dense

La seconde méthode que nous avons implémenté est la même que la première sauf que cette fois-ci on a utilisé la représentation *dense* des données.

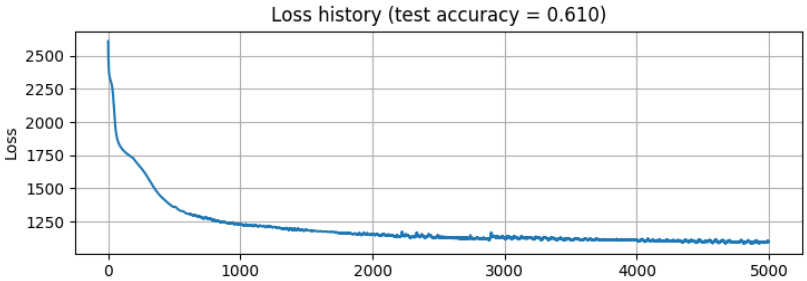
Lors de la cross-validation nous avons obtenu les résultats suivants :
(test_accuracy représente la moyenne des tests effectués pour la validation)

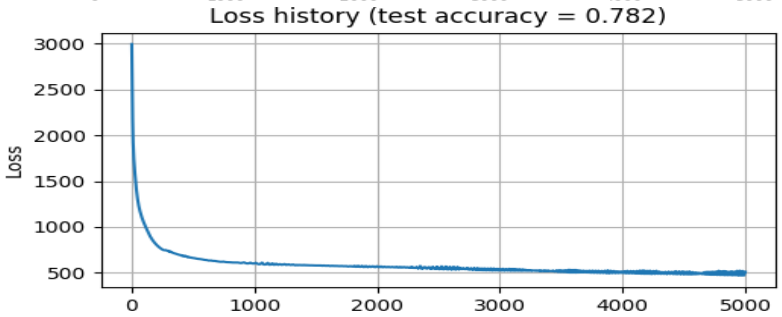




Premier exemple pour un réseau avec une couche cachée contenant 5 cellules.

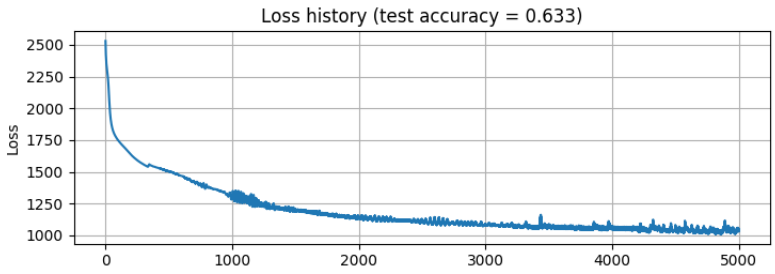
En haut, en soustrayant X et en bas, sans soustraire X.



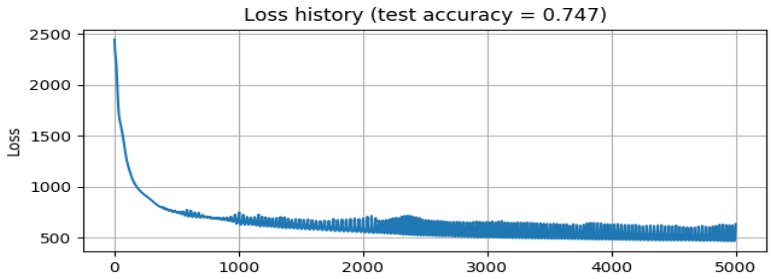


Deuxième exemple, pour un réseau avec deux couches cachées la première avec 5 cellules et la deuxième avec 15 cellules.

En haut, en soustrayant X et en bas, sans soustraire X.



Troisième exemple, pour un réseau avec trois couches cachées avec 8 cellules chacune.



En haut, en soustrayant X et en bas, sans soustraire X.

On voit que le meilleur modèle est celui avec deux couches cachées et la test accuracy est :

```
Test accuracy for the best model :  0.5881866997108632
```

Conclusion :

Pour la méthode dense, on remarque que la loss history est plus élevée au départ par rapport à la méthode sparse mais dans les deux cas, elles diminuent fortement. Pour la méthode dense, elle est plus proche de 200 – 500 que de zéro, donc elle est plus élevée au final par rapport à la méthode précédente.

Concernant l'accuracy, on peut faire les mêmes remarques que précédemment, elle est contenue entre 0.6 et 0.8 et la test accuracy finale est plus basse que la validation accuracy.

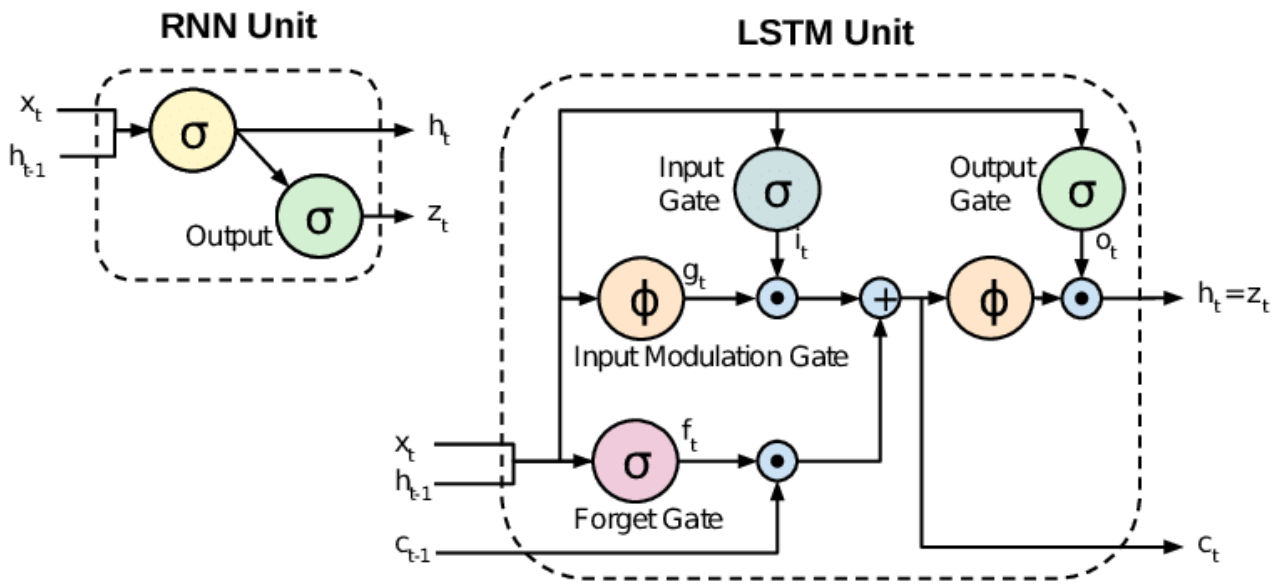
On remarque aussi que la loss history fait des pics à partir de 2000 epochs, on peut alors se demander si c'est lié au potentiel problème d'overfitting.

Dans le cas des deux modèles, on peut dire que l'accuracy n'est pas mauvaise ~60 % mais ce n'est pas suffisant pour notre problème. On a besoin d'un modèle beaucoup plus précis pour qu'il soit utile aux médecins.

ii. Réseaux de neurones récurrents (RNN et LSTM)

Une méthode à laquelle on s'est intéressé est ce que l'on appelle les réseaux de neurones récurrents (RNN) et leur version améliorée les long short-term memory (LSTM). Ces réseaux de neurones sont efficaces lors qu'ils utilisent des instances de training qui sont des séquences. Par exemple une suite de mots qui forme une phrase, une suite de frame qui forme une vidéo ou bien encore une suite de sons qui forment l'enregistrement d'une voix. Les RNN prennent en compte les données précédentes dans une séquence grâce à des poids qui partent d'une couche de neurone et qui reviennent sur elle même.

La différence entre les LSTM et les RNN classiques est que chaque node du LSTM est plus complexe que celle du RNN classique en utilisant un système de gates comme on peut le voir ci-dessous. Cette amélioration a pour avantage de permettre au LSTM d'apprendre plus efficacement.



Source : https://www.researchgate.net/profile/Aliaa_Rassem/publication/317954962/figure/fig2/AS:667792667860996@1536225587611/RNN-simple-cell-versus-LSTM-cell-4.png

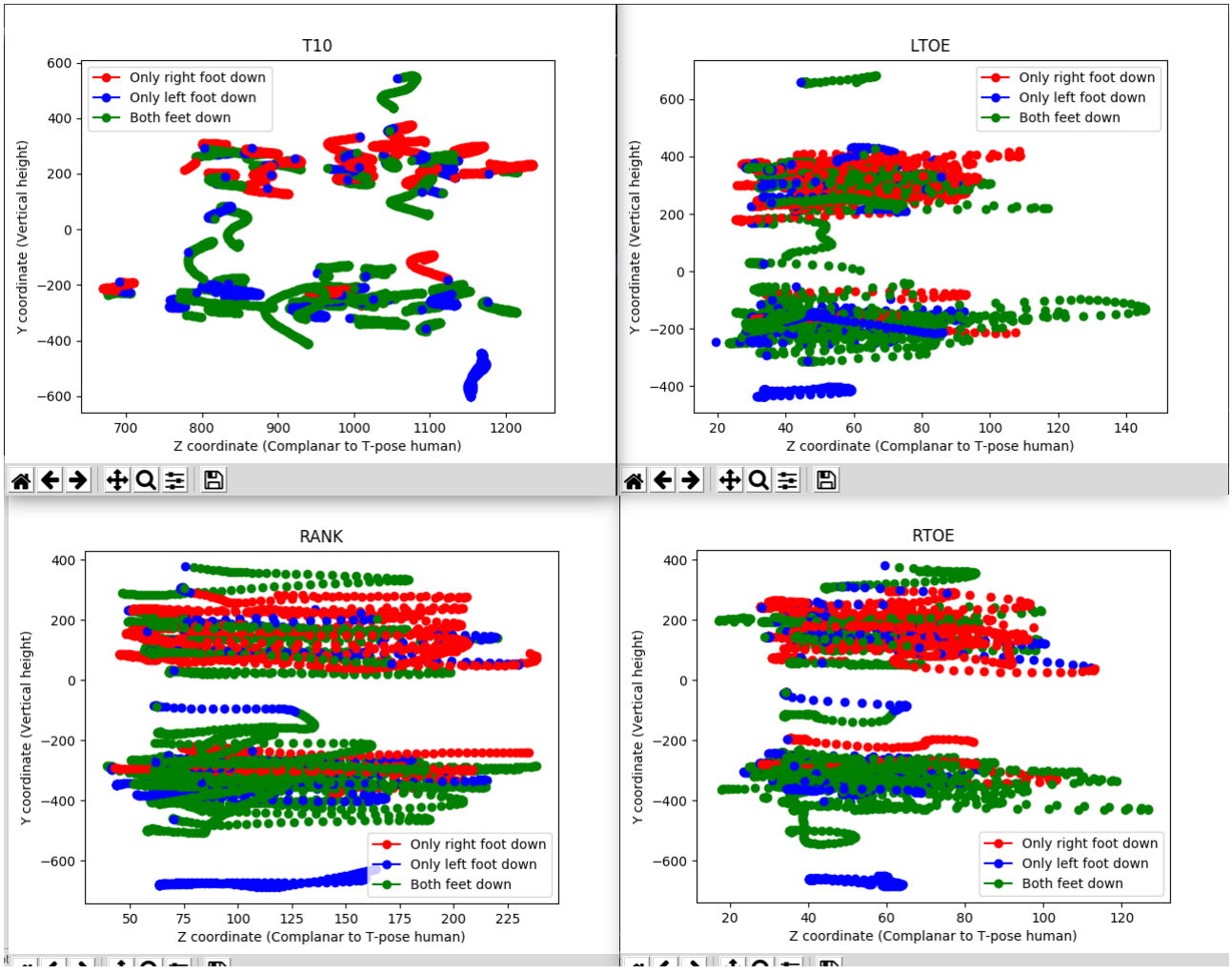
Les RNN et LSTM permettent deux choses lors d'un surpervised learning.
Premièrement, un LSTM peut prédire la suite d'une séquence qu'on lui donne. Par exemple dans le cas où on donne le début d'une phrase à un LSTM il pourra prédire une fin possible de la phrase.
Deuxièmement, un LSTM permet de classifier une séquence. Par exemple si on donne à un LSTM des vidéos de football ou des vidéos de publicités il pourra faire la différence entre les deux lorsqu'on lui donnera un nouvelle vidéo.

Dans notre situation, on souhaite aussi classifier mais nous voulons classifier chaque frame de notre vidéo et non pas tout une vidéo en entière. Pour régler partiellement ce problème on pourrait tricher

un peu en classifiant des petites extraits de vidéos mais on a besoin d'être précis dans notre projet et donc classifier des extrait de vidéos resterait trop vague. C'est pour ces raisons que l'on ne s'est pas servit de RNN / LSTM.

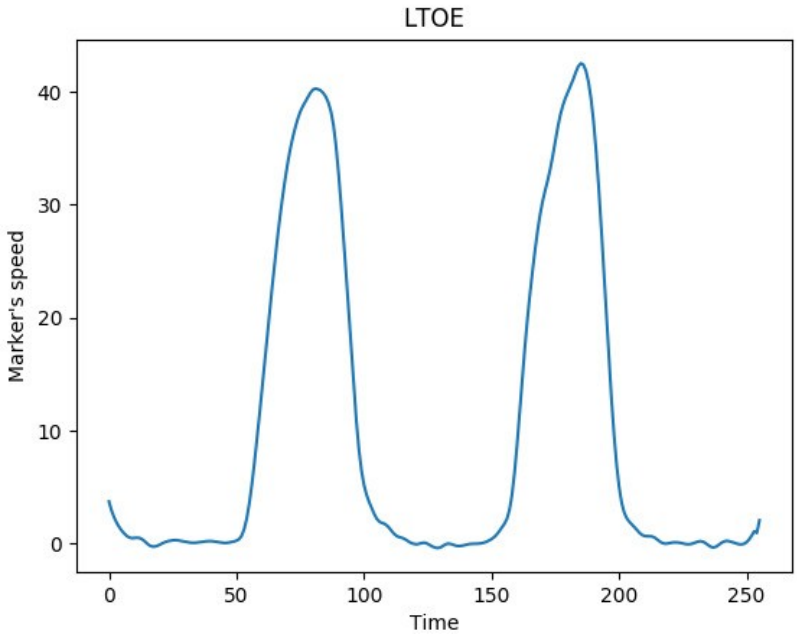
iii. Séparabilité des évènements (kNN)

Ensuite, nous avons pensé à d'autres méthodes connues en apprentissage automatique comme kNN (k nearest neighbors). Avec cette méthode, l'idée est de grouper les données et d'attribuer la classe de chaque données selon la classe de ses k voisins.
Pour que cette méthode soit efficace, il faut que les données soient séparables. Nous avons donc afficher les coordonnées y et z pour différents capteurs, afin de voir si les données bien groupées ou pas :

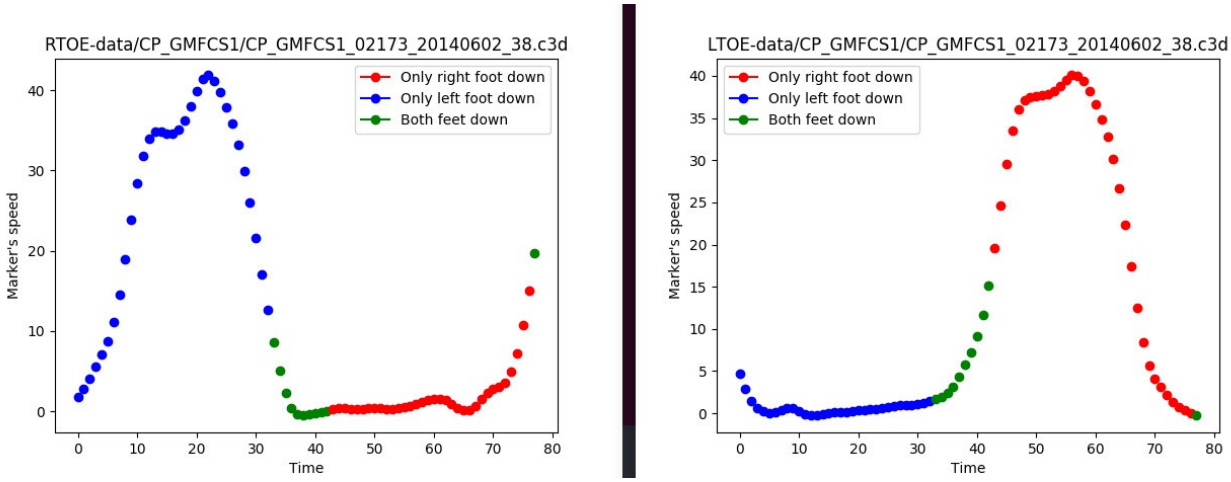


Comme on peut le voir, les données se chevauchent beaucoup. On a donc conclut que kNN ne serait pas un bon algorithme pour cette représentation des données.

Ensuite, on s’est intéressé à la coordonnée x et à sa vitesse et on a affiché la vitesse en x pour un capteur en fonction du temps. On obtient le graphique suivant :



On remarque que la courbe décrit particulièrement bien les étapes de la marche. En effet, quand le pied se lève, il accélère donc sa vitesse est plus grande et quand le pied est au sol, sa vitesse est nulle.
On a ensuite labellisé ces points et on a regardé si les évènements sont séparables. On obtient la figure suivante :



Ici, on voit que les données sont clairement séparables, on pourrait donc appliquer un réseau de neurones ou kNN pour classifier les données.

IV. Conclusion

On a vu que les méthodes avec réseaux de neurones simples et, malgré les deux représentations de données différentes, ne donnent pas de résultats satisfaisants.
L'utilisation des réseaux de neurones restent possible mais peut-être avec une autre représentation des données ou avec un autre type de réseau.

Néanmoins, par rapport aux observations que l'on a faites par rapport à la séparabilité des données, nous pensons qu'exploiter la vitesse en x est la méthode avec le plus de potentiel. Elle serait donc celle à développer en priorité.