CONCEPTION D'UN SYSTÈME DE CLASSIFICATION DE GESTES PAR RÉSEAU DE NEURONES À PARTIR DE DONNÉES INERTIELLES

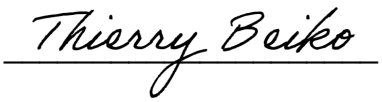
ELE3000

Présenté à

M. Raison

Par

Thierry Beiko



École Polytechnique de Montréal

9 décembre 2018

POLYTECHNIQUE MONTRÉAL

UNIVERSITÉ D'INGÉNIERIE

Table des matières

Table des figures

Introduction

Problématique

La capture et la classification du mouvement joue un rôle important au niveau du contrôle d'application. En effet, l'identification efficace d'un geste offre de nouvelles opportunités, que ce soit pour contrôler un ordinateur, un programme ou un dispositif quelconque. Il existe plusieurs méthodes de détection du mouvement telles que les systèmes de caméra, les capteurs inertiels et les signaux EMG. Cette dernière méthode est d'ailleurs celle qui est plus répandue actuellement, mais elle présente tout de même une certaine complexité. Effectivement, afin de capter les signaux musculaires du patient, de nombreux capteurs doivent être placés sur le sujet, ce qui complique le montage physique du système d'acquisition. De plus, les données générées par ces dispositifs peuvent être difficiles à interpréter et à manipuler. Elles doivent être traitées de nombreuses fois avant qu'il soit possible d'obtenir des résultats utiles. Ainsi, la complexité du montage et de l'analyse des données rend pertinent le développement d'un système plus simple et accessible, mais tout de même performant.

Objectifs

Dans cette optique, le projet cherche à développer un dispositif de détection et de classification du mouvement facilement réalisable et universel. Pour ce faire, le projet tente d'atteindre les objectifs suivants :

- Concevoir un dispositif de petite taille pouvant être porté au poignet
- Capter les données du mouvement grâce à une centrale inertielle
- Développer et entraîner un réseau de neurones pour la classification du mouvement
- Contrôler une application sur l'ordinateur selon le mouvement prédit

Les centrales inertielles sont utilisées en raison de leur petite taille et de leur précision. De plus, pour le contrôle d'application, sept mouvements ont été ciblés pour la classification, soit le cercle, la rotation horaire et antihoraire du poignet, la translation vers la droite, la gauche, le haut et le bas.

Spécifications fonctionnelles

Afin de répondre adéquatement aux objectifs ciblés, le système doit répondre à certaines spécifications fonctionnelles. Entre autres, il doit être en mesure de détecter l'initiation d'un geste pour ensuite capter les données inertielles propres au mouvement réalisé. À partir de ces données, le système doit pouvoir classifier le mouvement en moins d'une seconde grâce à un réseau de neurones. Par la suite, la commande correspondant au geste doit être effectuée. Puis, il doit être possible de détecter et de classifier plusieurs mouvements consécutifs.

Ainsi, le projet a été réalisé en se fiant à ces objectifs et développé selon l'architecture présentée à la figure 1.

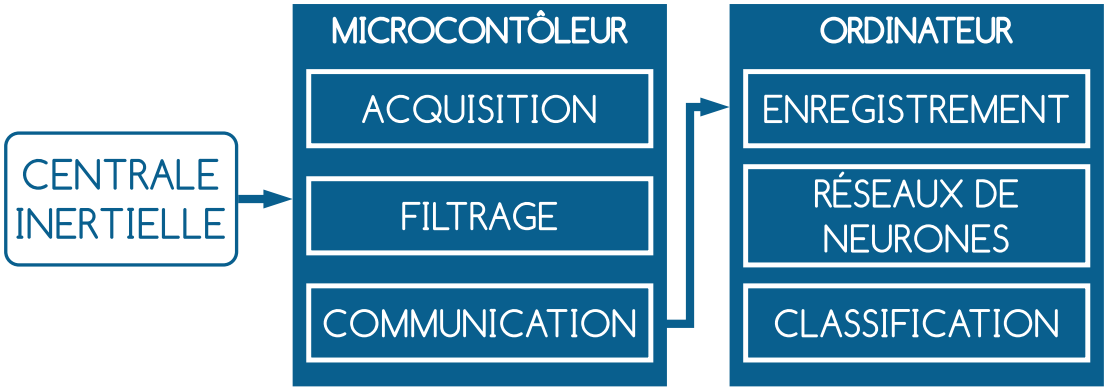


Figure 1 : Architecture du système

L'architecture peut être séparé en deux principaux modules, soit le microcontrôleur et l'ordinateur. Dans le cadre de ce rapport, le projet est séparé en 3 sections.

1. Acquisition et filtrage des données inertielles
    a. Montage du dispositif
    b. Pour l'entraînement du réseau de neurones
        i. Arduino
        ii. Matlab
    c. Pour la classification du mouvement
2. Développement et entraînement du réseau de neurones
3. Classification et contrôle d'application à partir de Python

Dans les prochaines sections, les différents algorithmes et éléments de programmation seront expliqués.

Méthodologie

Afin de détailler le fonctionnement du système, les différents fichiers utilisés seront présentés tout au long du rapport. Certains fichiers possèdent le même nom, mais puisqu'ils sont utilisés pour différentes sections du projet, leur contenu peut être différent. Pour clarifier, tous les fichiers utilisés dans le cadre du projet sont présentés dans le tableau suivant avec les indications précisant le contexte de leur utilisation ainsi que leur emplacement.

Tableau 1 : Fichiers utilisés pour la réalisation du projet

Acquisition pour l'entraînement du réseau de neurones		
Module	**Nom du fichier**	**Dossier**
1 b)	AcquisitionCercle.m	Acquisition (Matlab)
1 b)	AcquisitionRotG_D.m	Acquisition (Matlab)
1 b)	AcquisitionSwipeD_G.m	Acquisition (Matlab)
1 b)	AcquisitionSwipeH_B.m	Acquisition (Matlab)
1 b)	classementMvt.m	Acquisition (Matlab)/test
1 b)	classementMvt.m	Acquisition (Matlab)/train
Acquisition pour la classification (Arduino)		
1 c)	Compass.ino	Razor Code Cleanup/Razor_AHRS
1 c)	DCM.ino	Razor Code Cleanup/Razor_AHRS
1 c)	Math.ino	Razor Code Cleanup/Razor_AHRS
1 c)	Output.ino	Razor Code Cleanup/Razor_AHRS
1 c)	Razor_AHRS.ino	Razor Code Cleanup/Razor_AHRS
1 c)	Sensors.ino	Razor Code Cleanup/Razor_AHRS
Développement et entraînement du réseau de neurones		
2 a)	Modèle2.py	DNN
2 a)	DNN Optimizer.ipynb	DNN
2 b)	LSTM_model_good.h5	DNN
2 b)	X_train1.txt	DNN
2 b)	y_train1.txt	DNN
2 b)	X_test1.txt	DNN
2 b)	y_test1.txt	DNN
Classification et contrôle d'application à partir de Python		
3	AcquisitionPython.py	DNN

1. Acquisition et filtrage des données inertielles

a. Montage du dispositif

Pour la réalisation du montage physique, les pièces suivantes ont été utilisées :

- Microcontrôleur : Arduino/Genuine Micro
- Centrale inertielle : DFRobot 10DOF IMU

- Cable d'alimentation USB à micro-USB
- Fil de cuivre
- Petite plaque d'essai (Breadboard)
- Bracelet de fixation

Pour ce qui est du volet programmation de ce projet, les logiciels et librairies suivants ont été utilisés. Les détails de ceux-ci seront évoqués dans les sections respectives :

- Logiciel Arduino
- Code Razor_AHRS (Bartz, Razor AHRS v1.4.2, 2016)
- Tutoriel sur le code Razor_AHRS (Bartz, Razor AHRS v1.4.2, 2016)
- Librairie Arduino sur le filtre de Kalman (Chagas, 2014)
- Logiciel Python 3.7
- Librairie Tensorflow et Keras
- Logiciel Matlab 2018a

L'Arduino Micro ainsi que le centrale inertielle DFRobot ont été choisies en raison de leurs petites dimensions et de leur performance relative à leur prix. En effet, la centrale offre une bonne précision ainsi que 10 degrés de liberté, ce qui est amplement suffisant dans le cadre de ce projet. C'est également le cas pour le Arduino Micro qui répond aux exigences du projet.

Deux algorithmes d'acquisition ont été utilisés lors de ce projet, soit un pour l'acquisition des données pour l'apprentissage du réseau de neurones et un pour l'acquisition lors de l'utilisation du dispositif pour la classification du mouvement. La connexion de la centrale et du microcontrôleur ne change pas pour les deux cas et est illustrée sur la figure 2. Le montage électrique des composantes a été réalisé sur un *Breadboard* fixé sur le bracelet. Le montage complet est présenté sur la figure 3.

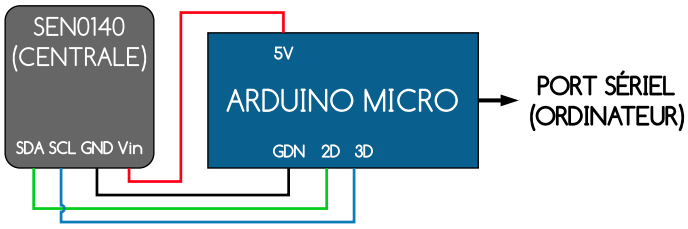


Figure 2: Schéma du montage

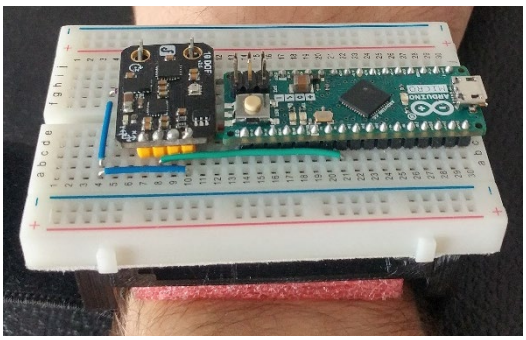


Figure 3: Montage complet

b. Pour l'entraînement du réseau de neurones

i) Arduino

Le premier code d'acquisition permet de capter les données de la centrale inertielle et de les envoyer à Matlab où elles seront enregistrées pour l'entraînement du réseau de neurones. Pour ce faire, le code Razor_AHRS qui provient de la page GitHub suivante : https://github.com/Razor-AHRS/razor-9dof-ahrs (Bartz, Razor AHRS v1.4.2, 2016) est utilisé. Ce dernier permet de capter les données provenant de la centrale à une fréquence de 50 Hz, et de les envoyer au microcontrôleur en utilisant le protocole de

communication I2C. Il contient les fichiers suivants : Razor_AHRS, Compass, DCM, Math, Output et Sensors. Puisque ce code a été réalisé pour une autre centrale inertielle (Sparkfun 9DOF), il a été nécessaire de modifier certaines sections pour l'adapter à la centrale inertielle utilisée, soit celle de DFRobot. Les deux centrales n'ont pas les mêmes repères, donc les axes de référence ont dû être modifiés. Ainsi, les axes X et Y de l'accéléromètre et du gyroscope de la centrale DFRobot correspondent à l'inverse des axes X et Y de la centrale Sparkfun. Pour y remédier, les données en question ont été multipliées par -1 dans la section Sensors du code :

```
accel[0] = -1 * (int16_t)((((uint16_t) buff[3]) << 8) | buff[2]);  // X axis (internal sensor y axis)
accel[1] = -1 * (int16_t)((((uint16_t) buff[1]) << 8) | buff[0]);  // Y axis (internal sensor x axis)
accel[2] = (int16_t)((((uint16_t) buff[5]) << 8) | buff[4]);  // Z axis (internal sensor z axis)
gyro[0] = (int16_t)(((((uint16_t) buff[2]) << 8) | buff[3]));    // X axis (internal sensor -y axis)
gyro[1] = (int16_t)(((((uint16_t) buff[0]) << 8) | buff[1]));    // Y axis (internal sensor -x axis)
gyro[2] = -1 * (int16_t)(((((uint16_t) buff[4]) << 8) | buff[5]));    // Z axis (internal sensor -z axis)
```

Figure 4: Modification des données captées

Ensuite, comme les valeurs d'accélérations sont influencées par la gravité et le bruit, il est nécessaire de calibrer les capteurs avant de les utiliser. La calibration a été effectuée selon les instructions trouvées sur cette page : https://github.com/Razor-AHRS/razor-9dof-ahrs/wiki/Tutorial#using-the-tracker (Bartz, Razor AHRS v1.4.2, 2016). Par la suite, en effectuant des tests sur les données, il a été possible de constater que l'apprentissage était plus performant lorsque les données du gyroscope étaient filtrées. Ainsi, un filtre Kalman a été appliqué sur les données du gyroscope. Pour ce faire, il faut d'abord importer la librairie trouvée sur la page : https://github.com/bachagas/Kalman (Chagas, 2014). Ensuite, il faut spécifier les paramètres du filtre afin de l'initialiser comme sur la figure 5, où q correspond à la covariance du bruit des mesures, r à la covariance du bruit des capteurs, p à la covariance de l'erreur sur l'estimation et k au gain de Kalman.

```
// kalman_init(double q, double r, double p, double k)
   Kalman filtre(0.0625,32,3,0);
```

Figure 5 : Initialisation du filtre Kalman

Ce type de filtre a été utilisé puisqu'il utilise les données antérieures pour éliminer le bruit sur les nouvelles mesures. Ainsi, cela permet de réduire la perte de précision des capteurs lorsque plusieurs mesures sont effectuées sur une longue période de temps. Les valeurs des paramètres ont été choisies après plusieurs essais en se basant sur l'allure des signaux de sortie de la centrale. Les valeurs sont filtrées dans la section Output avant d'être envoyées par le port sériel, comme le montre la figure 6.

```
gyro[0] = filtre.getFilteredValue(gyro[0]);
gyro[1] = filtre.getFilteredValue(gyro[1]);
gyro[2] = filtre.getFilteredValue(gyro[2]);
Serial.print(float((int(gyro[0])))); Serial.print(",");
Serial.print(float((int(gyro[1])))); Serial.print(",");
Serial.print(float((int(gyro[2])))); Serial.println();
```

Figure 6: Application du filtre Kalman et envoi des données du gyroscope par le port sériel

Puis, pour faciliter l'acquisition sur Matlab, les données sont envoyées sous forme de texte en continu par le port sériel. Pour ce faire, le mode et le format de la transmission sont spécifiés explicitement dans le code.

```
int output_mode = OUTPUT__MODE_SENSORS_CALIB;
int output_format = OUTPUT__FORMAT_TEXT;
```

Figure 7: Code permettant d'activer la transmission par le port sériel

De cette façon, il ne faut pas envoyer de commande d'activation pour que la transmission ait lieu. Finalement, comme les données provenant du magnétomètre ne nous intéressent pas, elles ont été retirées de la boucle de transmission dans le code Output. De plus, les données de l'accéléromètre et du gyroscope ont été transformées pour qu'elles soient envoyées sous la forme :

```
A -39.00,41.00,249.00
G -2.00,-2.00,-2.00
```

Figure 8: Format des données pour la transmission par le port sériel

ii) Matlab

Afin d'obtenir les données nécessaires pour entraîner le réseau de neurones à classifier les différents gestes, un algorithme d'acquisition a été conçu sur Matlab. Chaque type de geste (cercle, translation, rotation) possède un code légèrement différent, car les axes d'intérêts ne sont pas les mêmes pour chaque mouvement. Ainsi, pour alléger le présent rapport, seul l'algorithme d'acquisition pour les translations gauche/droite, soit le fichier AcquisitionSwipeD_G.m, sera détaillé. Dans son ensemble, ce code permet d'enregistrer les données relatives à une translation vers la droite et la gauche dans un fichier texte. Les grandes lignes de cet algorithme sont détaillées ci-dessous :

1. Ouverture du port sériel et configuration de la vitesse de transmission à 38 400 bits/seconde
2. Ouverture du fichier .txt dans lequel seront enregistrées les données du mouvement
3. Lecture du port sériel sur 50 itérations afin de permettre aux capteurs de se réchauffer
4. Détection du premier caractère d'une ligne sur le port sériel (A pour accéléromètre et G pour gyroscope)
5. Pour chaque nouvelle ligne reçue par le port sériel, les données sont enregistrées dans 3 variables distinctes pour chaque axe de chaque capteur (Accel1, Accel2, Accel3 et gyro1, gyro2, gyro3)
6. Calcule une moyenne sur 30 itérations pour chacun des axes lorsque qu'aucun mouvement n'est effectué
7. Calcule la différence entre la valeur d'accélération pour l'axe Z de l'accéléromètre afin de la comparer à un seuil (axe Z, car ce geste s'effectue le long de l'axe Z et donc différent pour les autres gestes)
8. Si l'écart est supérieur au seuil, dans ce cas 10 m/s^2, acquisition sur 70 itérations des données de tous les axes et enregistrement de la différence entre les valeurs et les moyennes, obtenues à l'étape 6, dans un tableau pour chaque axe
9. Les étapes 4 à 8 sont répétés 60 fois afin d'obtenir les données de 30 mouvements vers la droite et 30 mouvements vers la gauche

10. Écriture des données captées dans le fichier texte selon le format : Ax, Ay, Az, Gx, Gy, Gz

Pour les mouvements ayant deux directions, soit les deux types de translations et la rotation, les mouvements ont été effectués en alternance. Pour la translation droite/gauche, le premier mouvement est vers la droite, pour la translation haut/bas, le premier mouvement est vers le haut et pour la rotation, le premier mouvement est dans le sens antihoraire. Pour ce qui est du cercle, les mouvements étaient répétés un à la suite de l'autre dans le sens antihoraire. Il est possible qu'une acquisition soit lancée sans qu'un mouvement soit effectué en raison d'un geste non voulu. Dans un tel cas, le mouvement a été retiré du fichier afin d'être remplacé par un autre du même type. Pour ce faire, il est nécessaire de rester immobile lors d'une fausse acquisition, car de cette façon il est facile de la repérer puisque le fichier texte contient un ensemble de 70 données constantes. Le logiciel Notepad++ a été utilisé pour cette tâche. Le même processus a été répété pour chaque mouvement.

Afin de construire une base de données comportant tous les mouvements ainsi que l'identification de ceux-ci, un autre algorithme Matlab a été développé, soit ClassementMvt.m. Ce code permet d'assembler les données de chaque mouvement dans un seul fichier .txt (fichier *Input*) et de créer un autre fichier .txt avec l'identifiant de chaque mouvement respectif (fichier *Output*) qui serviront à l'entraînement du réseau de neurones. Les grandes lignes de cet algorithme sont détaillées ci-dessous :

1. Ouverture des fichiers .txt de tous les mouvement et création des fichiers qui contiendront toutes les données
2. Dans une boucle *for*, écriture des 70 données d'un mouvement dans le fichier *Input* et d'un seul identifiant dans le fichier *Output*, par alternance. Cette boucle est répétée pour le nombre de mouvements recueillis lors de l'acquisition

Afin de mélanger les mouvements, un ordre aléatoire a été choisi pour l'écriture des données. Comme les fichiers pour les mouvements à deux orientations comportent deux mouvements différents, les données de ces fichiers étaient transcrites en alternance pour s'assurer que l'identifiant inscrit dans le fichier *Output* correspondent bien au mouvement effectué. Pour ce faire, l'opérateur modulo (%) a été utilisé pour déterminer si l'index de la boucle *for* est pair ou impair. Afin d'identifier les mouvements de façon claire, chacun d'entres eux a été associé à un chiffre comme dans le tableau suivant :

Tableau 2 : Encodage du type de mouvements

Mouvement	**Chiffre attribué**
Cercle	0
Translation droite	1
Translation gauche	2
Rotation horaire	3
Rotation antihoraire	4
Translation haut	5
Translation bas	6

c. Pour la classification du mouvement

Le code utilisé pour l'acquisition des données qui serviront à la classification lors de l'utilisation du dispositif est encore basé sur le code Razor_AHRS. Toutefois, la section *Output* de celui-ci a été modifiée et ressemble davantage au code d'acquisition Matlab présenté plus haut. Contrairement au code présenté précédemment, ce code ci nécessite une commande d'activation afin d'entamer la transmission par le port sériel. Ainsi, pour activer l'acquisition, la commande '#on' doit être envoyée via le port sériel. Grâce à un booléen, celle-ci permet d'accéder dans la boucle d'acquisition située dans la fonction ***void output_sensors_text(char raw_or_calibrated)*** qui se trouve dans la section Output.ino. Il est également possible d'envoyer la commande '#of' afin de désactiver l'acquisition. Les grandes lignes de cette fonction sont détaillées ci-dessous :

1. Calcule la moyenne des accélérations pour les trois axes de l'accéléromètre et du gyroscope sur 30 itérations
2. Une fois la moyenne calculée, envoie le caractère 'G' par le port sériel, signifiant que l'algorithme est prêt à enregistrer un mouvement
3. Calcule la différence entre les valeurs des capteurs et les moyennes respectives obtenues à l'étape 1, afin de la comparer pour chaque axe des deux capteurs avec le seuil défini
4. Si l'écart est supérieur au seuil, soit une valeur de 10 m/s^2, acquisition sur 70 itérations des données du mouvement
5. Conversion des valeurs captées en type *string*
6. Envoie les données des 6 axes par le port sériel sur une seule ligne selon la forme : Ax, Ay, Az, Gx, Gy, Gz
7. Répète les étapes 1 à 6 en continu ou jusqu'à ce que la commande '#of' soit envoyée

L'étape 1 permet d'obtenir des nouvelles valeurs de calibration pour chaque mouvement dépendant de sa position dans l'espace. En effet, il est nécessaire de calibrer les capteurs à chaque acquisition, car lorsque le dispositif est immobile, les données transmises par les capteurs ne sont pas nécessairement nulles, mais elles demeurent constantes. Cependant, les valeurs dépendent de la position et de l'orientation des capteurs. Ainsi, les données envoyées par le port sériel correspondent aux valeurs brutes des capteurs auxquelles on soustrait la moyenne obtenue à l'étape 1.

2. Développement et entraînement du réseau de neurones

a. Développement

Afin de classifier les mouvements à partir de données inertielles, un réseau de neurones artificiels a été développé. Ce dernier a été implémenté avec Keras (https://keras.io/) qui est une librairie *open source* pour les réseaux de neurones sur Python. Dans le cadre du projet, la librairie *open source* Tensorflow (https://www.tensorflow.org/) a été utilisée comme *backend*. Grâce à Keras, le développement d'un modèle spécifique devient très explicite et ne nécessite pas de comprendre les concepts mathématiques complexes derrière l'apprentissage machine. Ainsi, dans la prochaine section le modèle développé sera présenté et détaillé, mais les notions concernant son fonctionnement ne seront pas abordées. Pour utiliser ces deux librairies, elles doivent être installées dans le bon répertoire et importées lors de l'implémentation sur Python. Le code nécessaire pour importer les librairies et les fonctions nécessaires est présenté à la figure 9.

```
import tensorflow as tf
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import LSTM,Dense,Dropout
```

Figure 9: Code permettant d'importer les librairies de développement de réseaux de neurones

Afin de classifier un mouvement, la dépendance temporelle des données est très importante, car le mouvement est représenté par une séquence de données dont l'ordre est important. Ceci signifie que pour prédire un mouvement, le modèle doit utiliser les données actuelles ainsi que les données antérieures. Les réseaux de neurones réguliers ne prennent pas en compte les données précédentes. Pour cette raison, il est nécessaire d'utiliser un réseau récurrent qui permet une certaine persistance de l'information. Pour ce faire, le modèle utilise des couches de cellules LSTM (*Long short-term memory*) standards présentées à la figure 10.

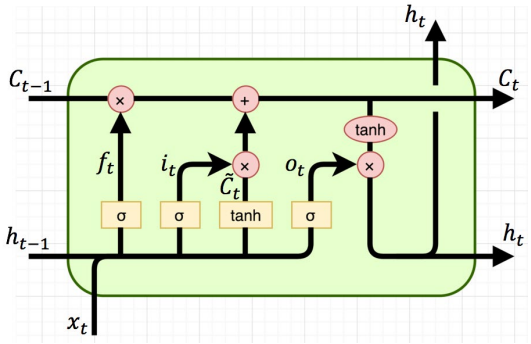


Figure 10: Représentation d'une cellule LSTM

Les cellules LSTM fonctionnent avec 3 fonctions définies dans la cellule selon différents poids. Les fonctions sont : la fonction d'oubli (*forget gate*) f_t, la fonction input (*input gate*) i_t et la fonction output (*output gate*) o_t. Chaque i-ème cellule représente le i-ème pas de temps d'une séquence de données. Chaque cellule reçoit X_t comme entrée pour le t-ème pas de temps et produit en sortie l'état actuel de la cellule, C_t, et une valeur de sortie, h_t. De plus, chaque cellule reçoit C_{t-1} qui contient l'état précédent de la cellule ainsi que toute l'information préservée jusqu'au pas de temps t-1. La cellule reçoit également h_{t-1} qui correspond à la sortie du pas de temps précédent (Drumond, Marques, Vasconcelos et Clua, 2018).

Une couche de LSTM correspond donc à une chaîne de cellules LSTM qui se partage l'information provenant de différents pas de temps.

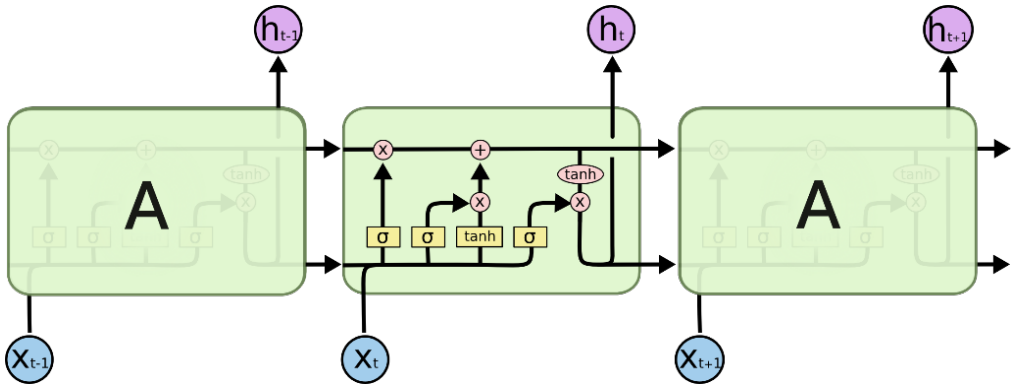


Figure 11 : Structure d'une couche LSTM

Puisque chaque cellule prend en entrée un pas de temps de la séquence et que les mouvements sont enregistrés sur 70 pas de temps, la première couche du modèle est une couche LSTM comportant 70 cellules. Après avoir analysé les résultats qui seront présentés dans la prochaine section, un *Dropout* de 59.66% a été introduit. La deuxième couche est également composée de cellules LSTM, mais celle-ci possède 50 cellules et un *Dropout* de 28.54%. Puis la troisième et dernière couche, est une couche *Dense* composée de 7 unités qui utilise l'activation *softmax*. Une couche *Dense* est une couche dans laquelle chaque entrée est connectée à toutes les sorties avec un certain poids. Finalement, le modèle est entraîné à évaluer la perte de l'entropie croisée en utilisant le *Adam optimizer*. Les paramètres utilisés pour implémenter le modèle ont été sélectionnés à la suite d'une optimisation utilisant le *Wrapper Hyperas* pour Keras (http://maxpumperla.com/hyperas/). Ce module détermine les valeurs optimales à utiliser pour les paramètres spécifiés. Il y parvient en effectuant des essais avec toutes les combinaisons possibles de paramètres et en testant les configurations avec les données d'apprentissage et de validation. Les paramètres optimisés pour ce modèle sont les suivant :

- Le nombre de cellules pour la première couche LSTM parmi les choix : {20, 50, **70**, 100}
- Le *Dropout* de la première couche entre 0 et 1 → **(0.5966)**
- Le nombre de cellules pour la deuxième couche LSTM parmi les choix : {20, **50**, 70, 100}
- Le *Dropout* de la première couche entre 0 et 1 → **(0.2854)**
- L'ajout ou **non** d'une couche *Dense* additionnelle
- Pour la couche additionnelle : nombre d'unités parmi les choix : {10, 50, 70} et le *Dropout* de la couche entre 0 et 1
- Grandeur du lot (*batch size*) lors de l'entraînement parmi les choix : {5, **7**, 10, 35}

Les résultats obtenus avec l'optimisation sont en caractère gras dans la liste ci-dessus. La structure du modèle développé est présentée à la figure 12.

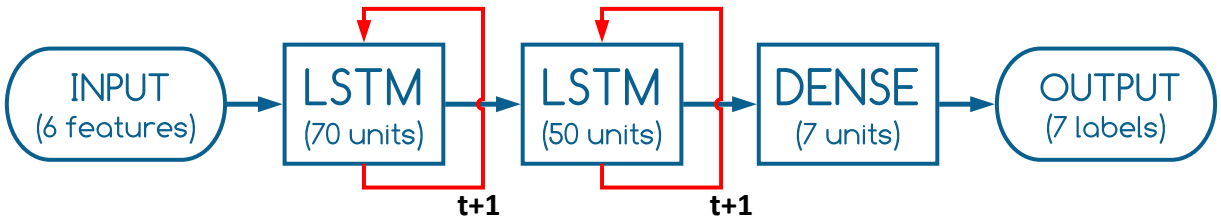


Figure 12 : Structure du réseau de neurones

b. Entraînement

Pour entraîner le modèle, les données obtenues avec l'acquisition sur Matlab, présentée à la section 1.b.ii), sont utilisées. Cependant, avant d'entraîner le modèle avec celles-ci, certaines modifications doivent être apportées.

Tout d'abord, les données sont lues à partir des fichiers .txt et elles sont enregistrées dans une variable. Ensuite, elles sont redéfinies en tant que *numpy array*.

```
X_train = pd.read_csv("X_train1.txt",header=None)
x_train = np.array(X_train)
```

Figure 13: Code pour la lecture et la transformation en tableau

Par la suite, il est nécessaire de modifier la forme des données pour qu'elles soient compatibles avec les cellules LSTM. En effet, la couche LSTM prend en entrée un tableau comportant trois dimensions. Ainsi la forme des entrées doit être la suivante : (nombre d'échantillons/mouvements, nombre de pas de temps par échantillons, nombre d'observations/données à chaque pas de temps). Puisque la banque de données d'entraînement contient 210 mouvements comportant chacun 70 pas de temps composés des données des 6 axes, les données d'entrées pour l'entraînement doivent être envoyées au modèle avec la structure (210, 70, 6). Les données de validation comportent seulement 70 mouvements, donc la structure des ces données est (70, 70, 6). Pour ce qui est des sorties, elles sont présentement sous la forme d'un chiffre entre 0 et 6 selon le mouvement qu'elles identifient. Cependant, le modèle fonctionne en calculant la probabilité que le mouvement effectué soit un tel geste, et ce, pour tous les gestes. Ainsi, il faut que la sortie comporte autant de valeurs que de nombre de gestes pouvant être classifiés. Pour ce faire, la fonction **to_categorical(y, num_classes=None)** de la librairie Keras est utilisée. Cette fonction crée un tableau 1D de **num_classes** éléments et associe chaque valeur présente dans **y** a un indice du tableau. Par exemple, pour un cercle identifié par un 0, la sortie transformée correspond à [1 0 0 0 0 0 0]. Pour une rotation antihoraire, identifié par un 4, la sortie transformée est [0 0 0 0 1 0 0] et ainsi de suite pour chaque geste.

```
x_train = x_train.reshape((210, 70,6))
ytrain = to_categorical(ytrain)
```

Figure 14: Code permettant de modifier les données d'entrée et de sortie

Une fois le modèle développé et les données modifiées, il est désormais possible d'entraîner le réseau de neurones. Pour ce faire, la fonction **model.fit()** est utilisée. Elle prend en argument les données d'entrée

et de sortie, le nombre de *epochs* et la grandeur du lot (*batch size*). Un *epoch* correspond à une itération lors de laquelle toutes les données sont passées dans le réseau. Donc pour 10 *epochs* le modèle va passer au travers de toutes les données à 10 reprises. La grandeur du lot correspond au nombre d'échantillons qui doivent être traités avant que les paramètres internes du modèle soient mis à jour.

```
epochs = 5
batch_size = 7
model.fit(x_train,ytrain,epochs=epochs,batch_size=batch_size)
```

Figure 15: Code permettant d'entraîner le modèle

Le nombre de *epochs* choisi est de 5, mais en observant les résultats il est possible de remarquer que le modèle converge après seulement 4 itérations. Les résultats de l'optimisation du modèle ont fourni une valeur de 7 comme étant la valeur optimale pour la grandeur du lot. Par la suite, il est possible d'évaluer le modèle sur les données de validation grâce à la fonction **model.evaluate()** qui prend en argument les données de validation d'entrée et de sortie ainsi que la grandeur du lot. Les résultats de l'évaluation seront présentés dans la section résultats. Finalement, le modèle entraîné est enregistré en format .h5 grâce à la fonction **model.save()** qui prend en argument le nom et l'extension du fichier. L'ensemble du code servant à créer et entraîner le modèle se retrouve dans le fichier Modèle2.py.

3. Classification et contrôle d'application à partir de Python

Maintenant que le modèle est développé et qu'il a été entraîné avec un nombre suffisamment élevé de données (210 X 70 = 14 700 données), il est possible d'effectuer des prédictions sur des nouvelles données avec le modèle. Afin d'acheminer les données vers le modèle, il faut d'abord établir la communication entre le microcontrôleur et le programme contenant le réseau de neurones sur Python. La communication s'effectue par le port sériel à une vitesse de transmission de 38 400 bits par seconde. Ainsi, il faut configurer le port sériel dans le programme pour que celui-ci puisse bien capter les données émises par le microcontrôleur comme sur la figure 16.

```
serPort = serial.Serial(port ='COM8', baudrate=38400, timeout = 0)
```

Figure 16 : Code de configuration du port sériel

Un *timeout* nul a été choisi, car ceci empêche le blocage de la lecture lorsque aucune donnée n'est envoyée au port sériel. Ainsi, la lecture retourne toujours quelque chose, soit *None* lorsqu'il n'y a rien soit les données lorsqu'il en a. Pour que la communication fonctionne adéquatement, il est nécessaire d'utiliser le module **.io** de Python qui facilite le traitement de plusieurs types de *input/output*. Puisque les données du microcontrôleur sont envoyée sous forme de texte, la classe dédiée au *stream* de texte est utilisée, soit **io.TextIOWrapper**. Ce type de protocole de communication permet de placer les données provenant du port sériel dans un *buffer*. De cette façon, il est possible d'obtenir des données qui ont été envoyées lorsqu'il n'y avait pas de lecture sur le port sériel. Le code permettant de bien configurer le protocole de communication se trouve à la figure 17.

```
serioText = io.TextIOWrapper(io.BufferedRWPair(serPort, serPort, 1), encoding='ascii')
```

Figure 17 : Configuration du protocole de communication

Afin d'entamer le début de l'acquisition du mouvent, il faut d'abord envoyer la commande d'activation au microcontrôleur grâce au code de la figure 18.

```
serioText.write('#on')
```

Figure 18: Code d'activation de l'acquisition sur le microcontrôleur

Ensuite, comme expliqué dans la section 1 c), lorsque le programme est prêt à enregistrer un nouveau mouvement, le caractère 'G' est envoyé. Il faut donc capter ce caractère avant de commencer l'enregistrement des données. Pour ce faire, une boucle vérifie d'abord s'il y a des données à être lues dans le tampon de lecture. S'il n'y en a pas, un délai de 0.02 seconde est forcé entre chaque vérification. Ce délai laisse le temps au microcontrôleur de transmettre de nouvelles données puisque l'acquisition de la centrale inertielle s'effectue à une fréquence de 50 Hz.

```
while serPort.in_waiting < 0:
    time.sleep(0.02)
```

Figure 19: Code de vérification du tampon de lecture

Si le tampon de lecture contient des valeurs, le programme sort de la boucle et commence la lecture ligne par ligne des données. La présence du caractère 'G' est vérifiée pour chacune d'entre elle. Une fois qu'il

est capté, le programme entre dans une autre boucle de lecture ligne par ligne. Lorsque aucun mouvement n'est effectué, le microcontrôleur ne transmet rien, donc la lecture retourne *None* et le programme demeure dans la même boucle en attendant le début d'un geste.

```
char2 = None
while not char2:
    char2 = serioText.readline()
```

Figure 20: Code de la boucle d'attente

Une fois qu'un mouvement est amorcé, le programme entame une boucle *for* de 70 itérations pour l'enregistrement des données inertielles. Cette boucle comporte les étapes suivantes :

1- Lecture d'une ligne complète
2- Conversion de la ligne de texte en tableau *numpy* de *float* grâce à la fonction **np.fromstring()**
3- Transformation de la forme du tableau pour qu'elle soit compatible avec le réseau de neurones
4- Enregistrement des données dans un tableau contenant les 70 ensembles de données

Une fois l'acquisition terminée, la commande '#of' est envoyée au port sériel pour désactiver l'acquisition. Afin d'effectuer des prédictions, il faut d'abord charger le modèle avec la fonctions **model.load_model('LSTM_model_good.h5')**. Par la suite, on fournit les données au modèle pour qu'il puisse calculer la probabilité de chaque *output*.

```
output = model.predict(data,batch_size = 1, verbose =1)
```

Figure 21: Code permettant d'effectuer des prédictions sur des données (data)

Ici, la variable *output* est un tableau qui contient la probabilité de chaque mouvement selon leur index respectif présenté dans le tableau 2. De façon générale, le mouvement effectué correspond au mouvement qui possède la probabilité maximale. Ainsi, en utilisant la fonction **np.argmax(*output*)** il est possible d'obtenir l'indice de la valeur maximale dans la tableau. Puisque les mouvements correspondent à des indices prédéfinis, il est possible d'associer des actions à chaque indice lorsque celui-ci et considéré maximal. Dans le cadre de ce projet, l'application musicale *Spotify* a été sélectionnée comme exemple de contrôle d'application. Afin de contrôler cette application, les commandes du clavier prédéfinies sur *Spotify* ont été utilisées. Pour simuler l'appui de touches, la librairie *keyboard* de Python a été utilisée. Ainsi, chaque action est réalisée en simulant l'appui de touche spécifiques. Les actions réalisées par chaque mouvement sont définies dans le tableau suivant :

Tableau 3 : Actions associées aux gestes et à leur indice respectif ainsi que les commandes de clavier

Indice	Geste	Action	Commandes clavier (touches)
0	Cercle	Met la musique en pause	Espace
1	Translation droite	Prochaine chanson	Ctrl+right
2	Translation gauche	Chanson précédente	Ctrl+left
3	Rotation horaire	Augmente le volume	B (Maj)
4	Rotation antihoraire	Diminue le volume	C (Maj)
5	Translation haut	Avance de 10 secondes	Shift+right
6	Translation bas	Recule de 10 secondes	Shift+left

Résultats

Avant d'obtenir le modèle final pour le réseau de neurones plusieurs essais ont dû être réalisés pour déterminer sa performance. Les différents essais, les paramètres utilisés et les conclusions tirées seront présentés ci-dessous. Pour chaque modèle, les graphiques de la précision et de la perte en fonction des *epochs* ont été générés.

Tout d'abord, le premier modèle a été entraîné avec un ensemble de données différents de celui utilisé pour le modèle final. Le premier ensemble de données contenait seulement 10 échantillons de chaque mouvement, mais chaque mouvement est enregistré sur 100 pas de temps.

Les paramètres utilisés pour le premier essai sont les suivants :

- 1ère couche LSTM : 20 cellules, *return_sequences = True*, Dropout = 0.143258
- 2e couche LSTM : 80 cellules, Dropout = 0.25254
- 3e couche Dense : 50 unités, activation = relu, Dropout = 0.69785
- 4e couche Dense : 7 unités, activation = softmax
- 20 epochs
- Grandeur de lot = 7

Le modèle a été évalué avec un ensemble de données de validation comportant 35 mouvements, soit 5 de chaque type.

Après l'entraînement, les résultats suivants ont été obtenus :

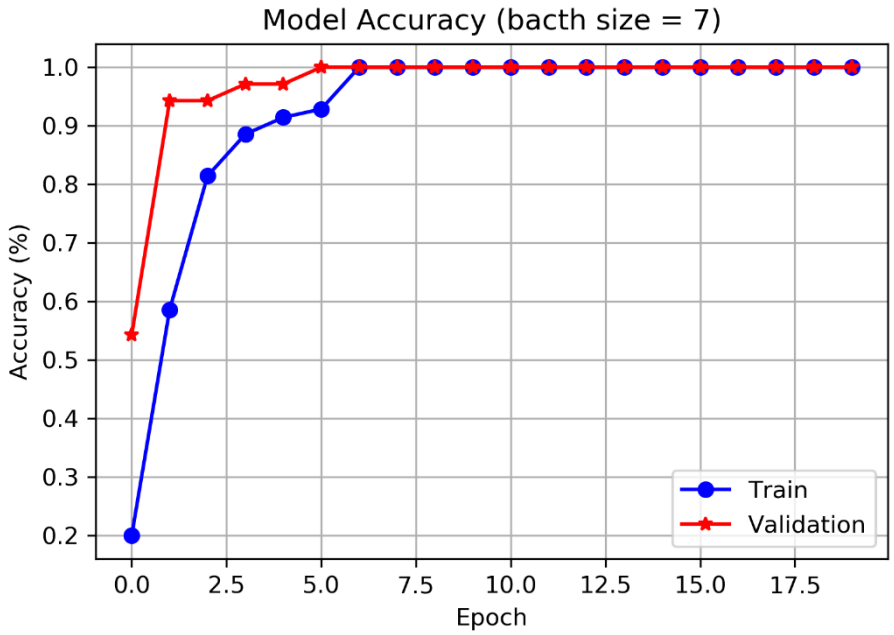


Figure 22: Précision du modèle #1 lors du premier essai

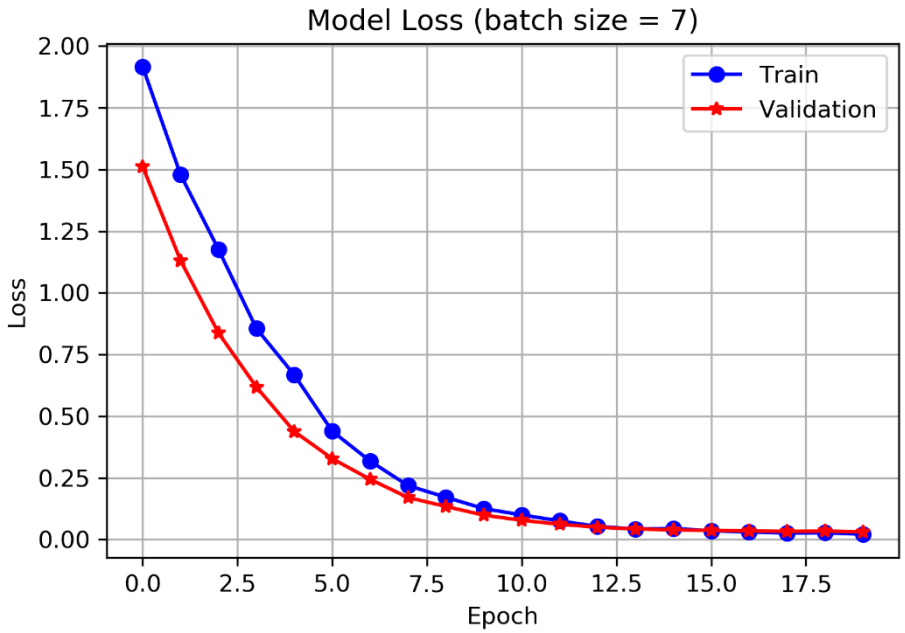


Figure 23: Perte du modèle #1 lors du premier essai

L'allure de la courbe de la figure 22 laisse croire que le modèle généré est très performant puisqu'il converge vers une précision de 100% après seulement 6 itérations. Bien que le modèle performe adéquatement sur les données de validation, il n'en est pas de même lors des tests. Ceci s'explique par le phénomène de *overfitting* ou surapprentissage qui survient lorsque l'ensemble de données d'apprentissage utilisé est trop limité. Le surapprentissage se produit lorsque le modèle interprète les détails et le bruit des données d'apprentissage comme étant des caractéristiques fondamentales du signal. Ainsi, lorsqu'on effectue des prédictions sur de nouvelles données, ces caractéristiques ne sont pas nécessairement présentes, donc la performance du modèle est affectée.

Pour surmonter ce problème, un nouvel ensemble de données à été recueilli pour l'entraînement du deuxième modèle. Celui-ci est plus grand et contient 210 mouvements, soit 30 pour chaque type, enregistrés sur 70 pas de temps.

Les paramètres du modèle pour le second essai sont les suivants :

- 1ère couche LSTM : 100 cellules, *return_sequences = True*, Dropout = 0.5
- 2e couche LSTM : 100 cellules, Dropout = 0.5
- 3e couche Dense : 7 unités, activation = softmax
- 20 epochs
- Grandeur de lot = 70

Après l'entraînement, les résultats suivants ont été obtenus :

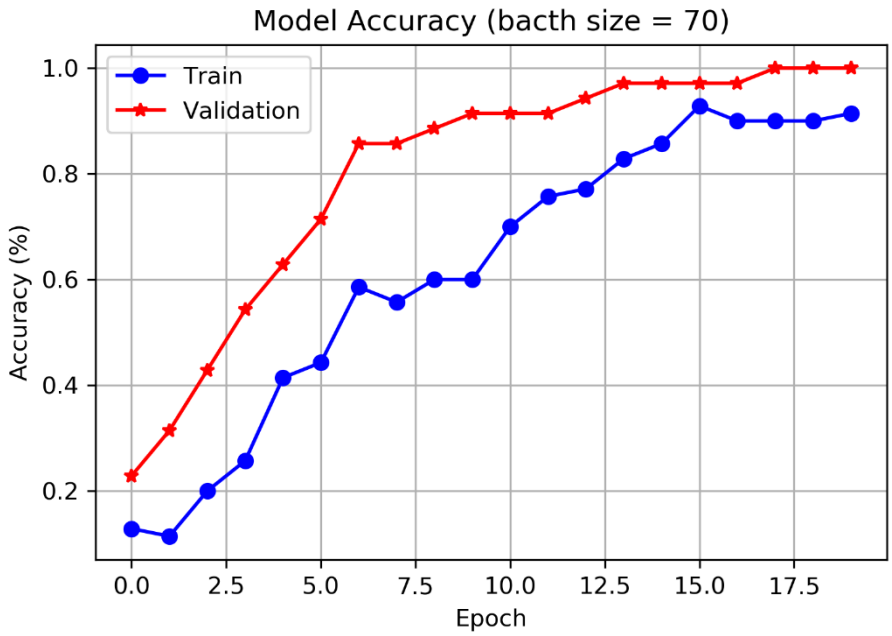


Figure 24: Précision du modèle #2 lors du second essai

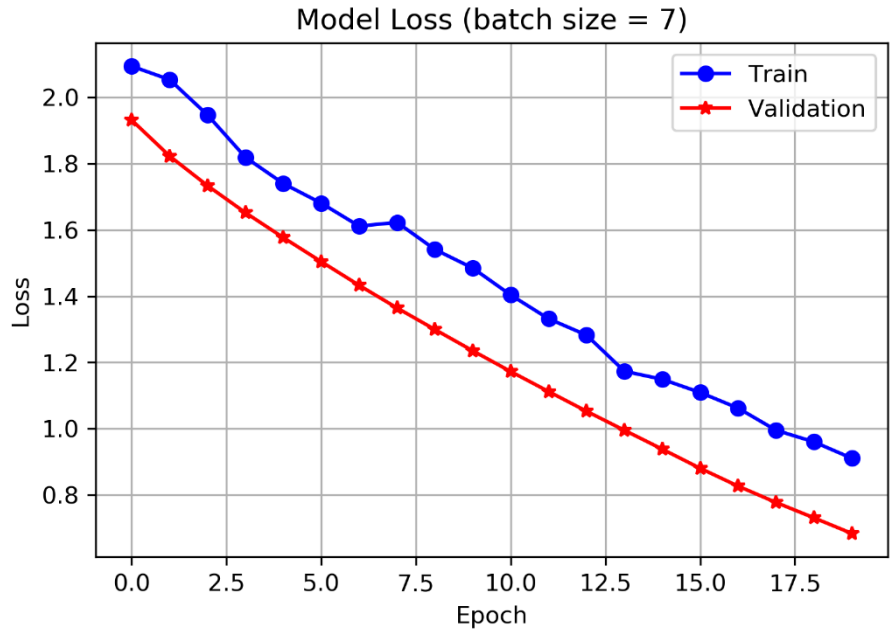


Figure 25: Perte du modèle #2 lors du second essai

Le modèle 2 ne semble pas être affecté par le surapprentissage. Au contraire, en observant la figure 24 on observe que le modèle ne converge pas parfaitement même après 20 itérations. De plus, en analysant la perte sur la figure 25, on réalise qu'elle demeure élevée après 20 itérations. Afin de régler le modèle, une optimisation des paramètres a été effectuée avant le module *Hyperas* en suivant les instructions de la page http://maxpumperla.com/hyperas/ et en l'ajustant au modèle conçu.

L'optimisation a retourné les paramètres suivants qui ont permis de définir le modèle #3.

- 1ère couche LSTM : 70 cellules, *return_sequences = True*, Dropout = 0.596602
- 2e couche LSTM : 50 cellules, Dropout = 0.2854
- 3e couche Dense : 7 unités, activation = softmax
- 5 epochs
- Grandeur de lot = 7

Après l'entraînement, les résultats suivants ont été obtenus :

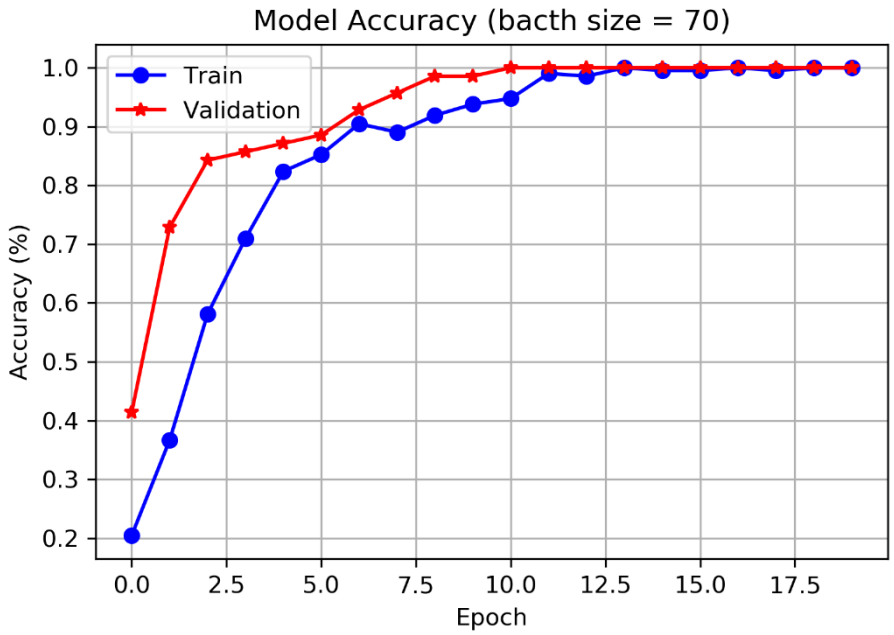


Figure 26: Précision du modèle #3 lors du troisième essai

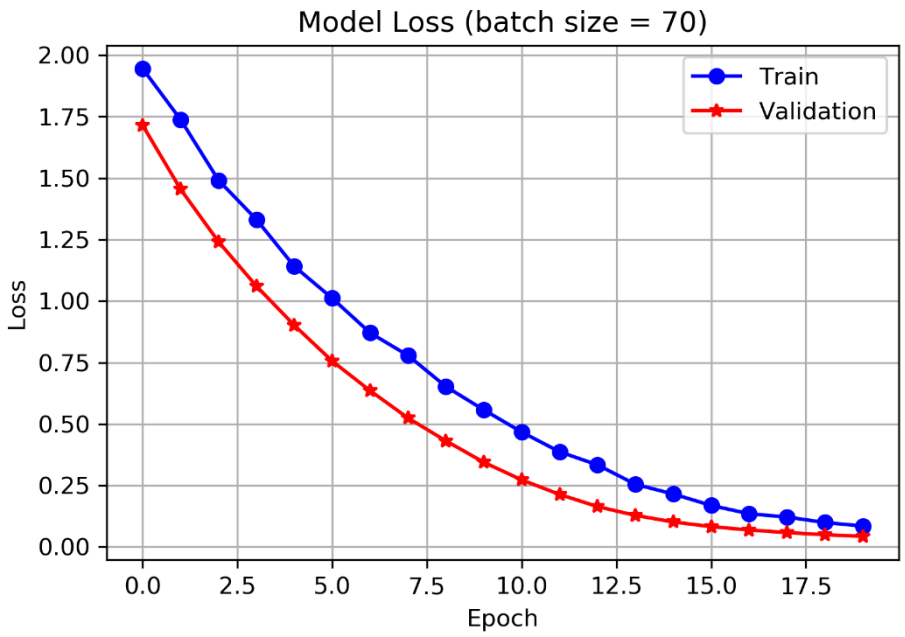


Figure 27: Perte du modèle #3 lors du troisième essai

Les figure 26 et 27 montrent que l'optimisation a permis d'augmenter la précision du modèle et de réduire sa perte. Cependant, en utilisant une grandeur de lot de 70, le modèle nécessite 13 itérations avant de converger et 20 itérations pour réellement diminuer la perte de façon minimale.

Par conséquent, pour obtenir le modèle final, les mêmes paramètres ont été maintenus sauf la grandeur de lot qui été diminué à 7 afin de permettre au modèle de converger plus rapidement. Les résultats suivants ont été obtenus :

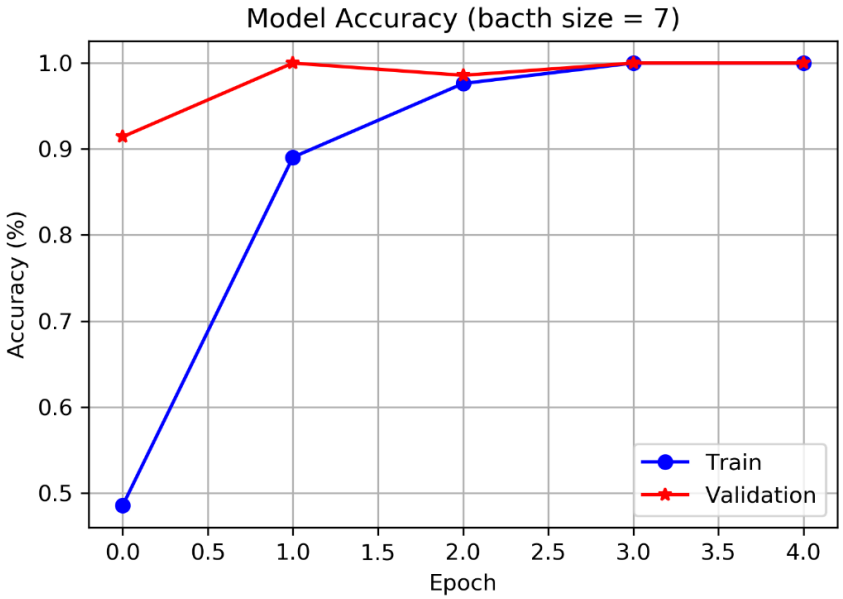


Figure 28 : Précision du modèle final lors du quatrième essai

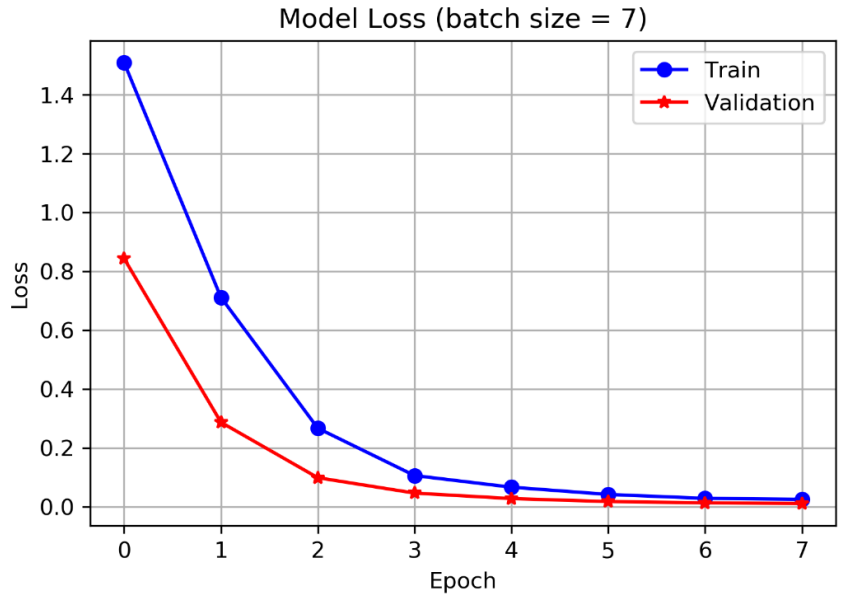


Figure 29: Perte du modèle final lors du quatrième essai

Avec ce modèle final, il a été possible de détecter et de classifier 50 mouvements distincts avec un taux de réussite de 100%. Cependant, lors de la classification de mouvements consécutifs, le taux de réussite

obtenu est plutôt de 90%. Ce dernier résultat a été obtenu en effectuant 15 séquences de 3 mouvements consécutifs.

Analyse des résultats

À la lumière de ces résultats, il est possible d'affirmer que le projet répond aux objectifs, soit de détecter et de classifier les mouvements de l'avant-bras en utilisant un réseau de neurones. De plus, les résultats de la classification ont pu être utilisés afin de contrôler adéquatement une application sur l'ordinateur, soit le logiciel *Spotify* lors des démonstrations. En effectuant la classification sur l'ordinateur plutôt que sur le microcontrôleur, le contrôle d'application s'effectue sans délai important entre le mouvement et l'action. Ainsi, le dispositif répond aux spécifications fonctionnelles à l'exception de la classification de mouvement consécutifs. En effet, un taux de réussite de 90% est observé pour celle-ci. Lorsqu'un premier mouvement est effectué et classifié, il arrive parfois que la seconde acquisition soit lancée sans que l'utilisateur n'ait initié de mouvement. Ceci fait en sorte que les données obtenues ne correspondent pas à un mouvement complet ou qu'elles correspondent à un non mouvement qui n'est pas classifié par le modèle.

Les résultats présentés ont été obtenus lorsqu'un utilisateur entraîné et connaissant le fonctionnement du programme, moi-même, utilise le dispositif. Des essais ont été réalisés avec d'autres sujets, mais les résultats n'étaient pas toujours concluants. Les données d'apprentissage et de validation ont toutes été obtenues à partir du même utilisateur. De plus, bien que le nombre de données utilisées fût suffisant dans le cadre de ce projet, il s'agit tout de même d'un faible nombre de données selon les standards de l'apprentissage machine.

Perspectives et améliorations

Montage du dispositif

Il serait intéressant d'incorporer un module *Bluetooth* au dispositif afin de le rendre sans fil. Le fil de connexion avec l'ordinateur peut nuire au mouvement et ainsi introduire certaines erreurs au niveau de l'acquisition des données.

Communication via le port sériel

Lors de l'envoi des données du microcontrôleur vers le programme de classification il arrive parfois qu'une erreur de compatibilité se produise au niveau du port sériel. Celle-ci est causé par la forme des données envoyées et la façon dont celles-ci sont lues par le programme. Il y a donc place à améliorations au niveau logiciel de la communication sérielle.

Données d'apprentissage

Afin d'améliorer la performance et l'universalité du réseau de neurones, il est primordial d'obtenir des données provenant de différents utilisateurs. Ceci permet d'adapter le modèle aux différentes amplitudes, vitesses, formes et durées des mouvements propres à chaque personne. De plus, en effectuant l'acquisition des données sur un grand nombre de personnes, la quantité de données d'apprentissage augmentera de façon considérable ce qui est favorable pour l'apprentissage machine.

Conclusion

Somme toute, le projet a permis de prouver la possibilité et la performance d'un système de détection et de classification des mouvements de l'avant-bras en utilisant des données inertielles et un réseau de neurones récurrent. Le dispositif parvient à détecter et classifier sept mouvements de l'avant-bras sans délai majeur. La classification du mouvement permet par la suite le contrôle d'application du logiciel *Spotify*. Cependant, certaines améliorations sont nécessaires afin de rendre le dispositif plus performant pour faire en sorte que son utilisation soit plus naturelle. Notamment, en rendant le dispositif sans fil et en effectuant une acquisition plus complète et exhaustive des données d'apprentissage, il serait possible de rendre ce projet plus robuste et utile. En bref, les développements effectués et les résultats obtenus lors du projet offrent une bonne base pour la continuité de ce dernier.

Bibliographie

Bartz, P. (2016). *Building an AHRS using the SparkFun "9DOF Razor IMU" or "9DOF Sensor Stick"*. Récupéré sur GitHub: https://github.com/Razor-AHRS/razor-9dofahrs/wiki/Tutorial#using-the-tracker

Bartz, P. (2016). *Razor AHRS v1.4.2*. Récupéré sur GitHub: https://github.com/Razor-AHRS/razor-9dof-ahrs

Chagas, B. A. (2014). *Kalman*. Récupéré sur GitHub: https://github.com/bachagas/Kalman

Drumond, R., Marques, B., Vasconcelos, C. and Clua, E. (2018). PEEK - An LSTM Recurrent Network for Motion Classification from Sparse Data. *In Proceedings of the 13th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications,* 1, 215-222. DOI: 10.5220/0006585202150222