Machine Learning pour le diagnostic de machines défectueuses

Aymen BACCAR - Loïc BAL - Antoine BROSSAS - Tom CARRERE - Florian DERLIQUE

SE4

ANNÉE UNIVERSITAIRE 2021/2022

Sommaire

Introduction

Durant les deux premiers semestres de ce projet, nous avons découvert et étudié le Machine Learning, que ce soit la théorie de différents algorithmes connus ou l'application pratique à des datasets que nous pouvons récupérer sur Internet ou bien créer nous même.

La création de notre dataset ayant été fastidieuse et peu concluante (voir résultats du semestre précédent), nous avons comme objectif pour ce semestre d'implémenter nos algorithmes (et principalement le réseau de neurones) à des machines dont nous avons accès à l'école.

Après discussion avec notre tuteur de projet Monsieur Lakhal, nous avons décidé d'utiliser des Robotino plutôt que d'utiliser la chaîne de production présente à l'école car il existe des logiciels simples permettant d'effectuer des simulations de gestion d'entrepôt par des Robotinos (Robotino View et Robotino Simulation) dont nous allons récupérer les données en temps réel, les traiter puis commander les différents Robotino à l'aide d'un algorithme de réseau de neurones.

L'objectif de ce semestre est ainsi de réaliser le scénario d'un réseau logistique autonome (à l'image d'Amazon par exemple) composé de plusieurs Robotino effectuant chacun une tâche et d'y introduire des erreurs puis de les traiter grâce au Machine Learning afin de ne pas perturber la logistique et d'obtenir un maximum d'efficacité.

Nous avons organisé ce projet en plusieurs étapes où chacun des membres a pu plus ou moins participé en fonction de ses capacités :

- Conception du Grafcet pour la commande des Robotino
- Récupération et traitement des données en temps réel
- Utilisation du Machine Learning afin de commander les Robotino

Pour cela, nous avons constitué 2 équipes : une équipe spécialisée dans la commande du Robotino constituée de Loïc et Aymen et l'autre dans le traitement des données avec le Machine Learning avec Antoine, Tom et Florian.
Nous avons ensuite organisé notre projet à l'aide du diagramme de Gantt ci-dessous.

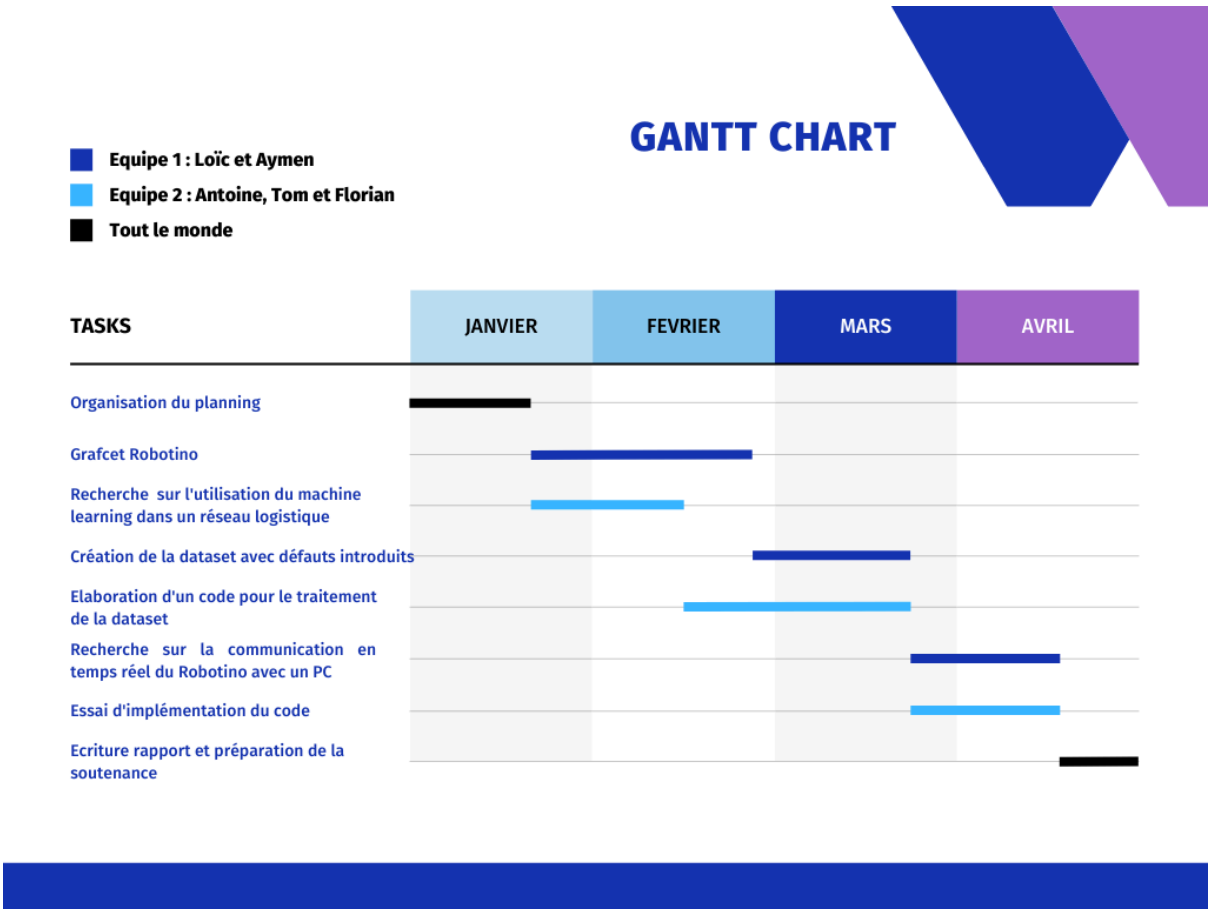
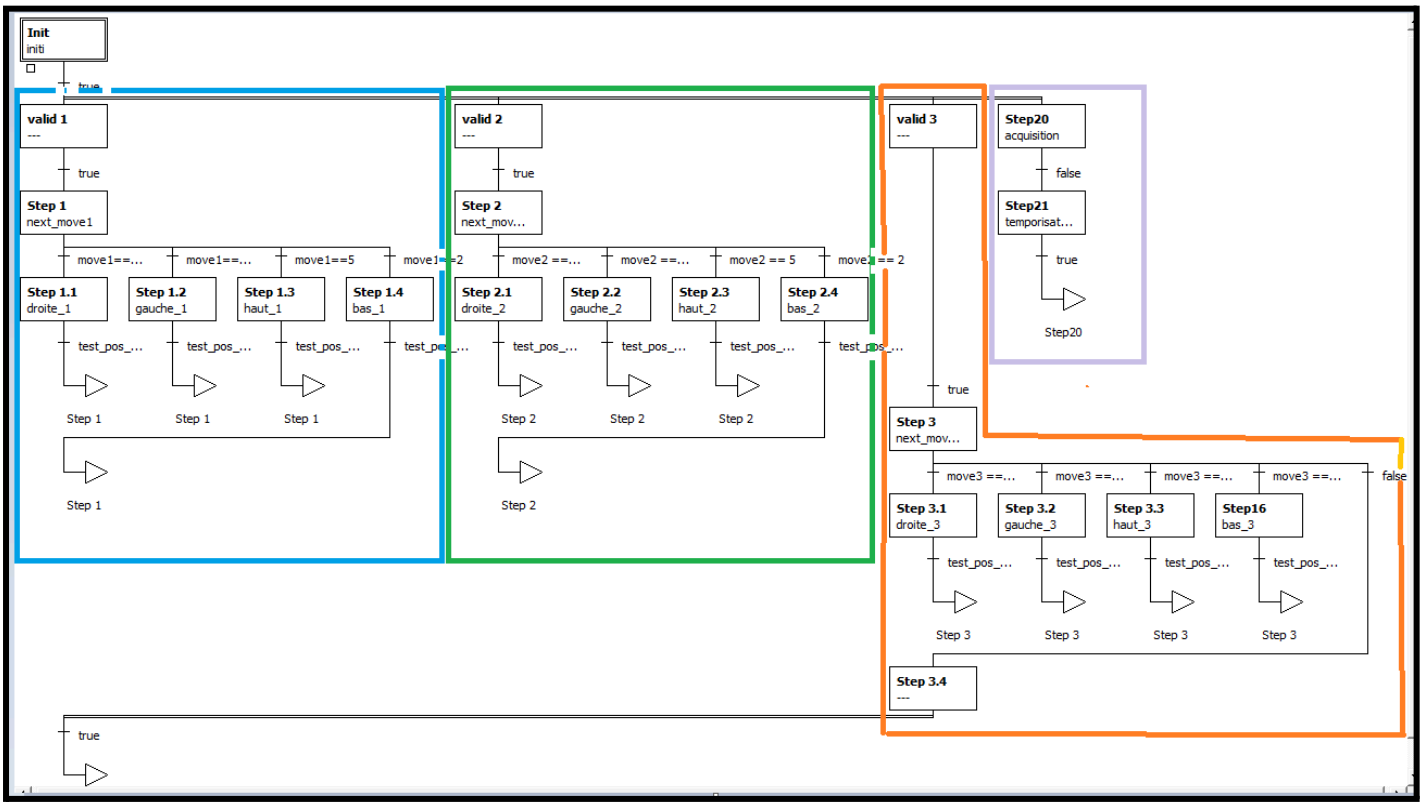


Diagramme de Gantt

Grafcet pour la commande des Robotinos

Au début on a commencé par établir un Grafcet sur Robotino View afin de commander 3 Robotino et acquérir des informations sur son état de fonctionnement (courants fournis aux trois moteurs, niveau de la batterie, vitesse du robot …).

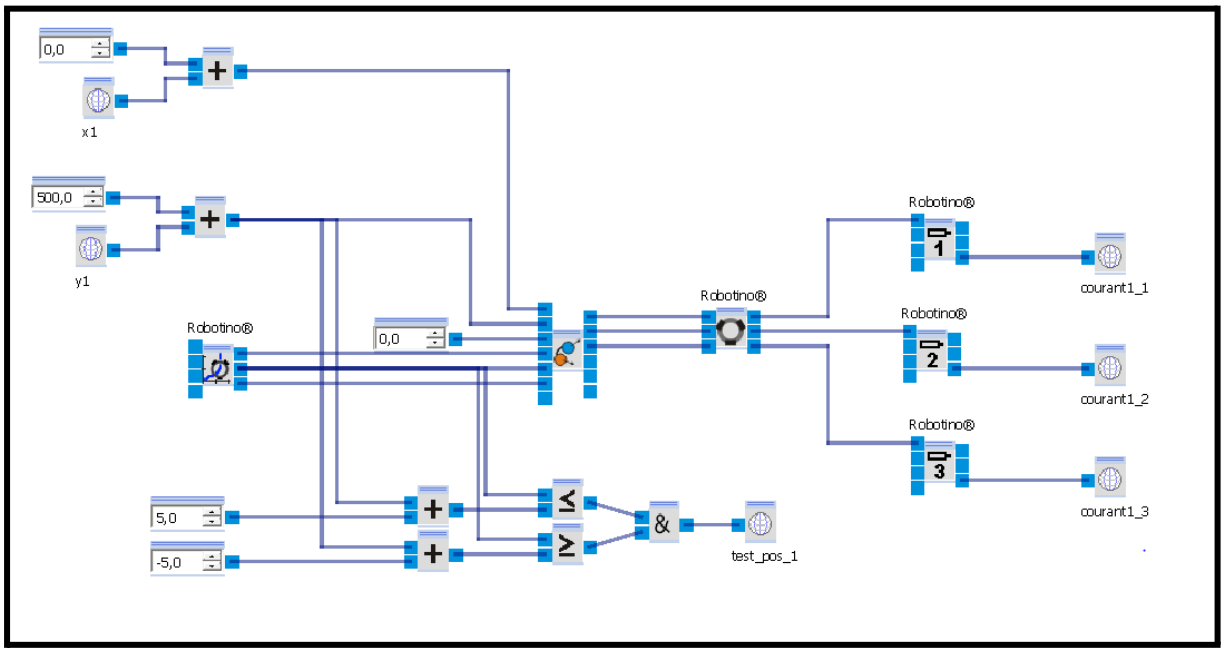


Ce grafcet contient 4 fonctions principales dont trois pour définir les trajectoires des Robotino et la dernière pour acquérir en continu les données à travers les capteurs implantés dans ces robots.

1) Définition de la trajectoire du robotino

Nous avons défini pour la commande des robots des fonctions de déplacement pour chaque direction (Droite, Gauche, Haut, Bas) et pour le faire nous avons utilisé l'odométrie du moteur pour l'asservir en position.

Ce fonctionnement repose sur l'addition entre une constante qui représente la distance à parcourir dans la direction concernée et les coordonnées initiales du Robotino qu'on enregistre d'une manière continue chaque fois qu'on parcourt la boucle, ensuite nous avons introduit les résultats de ces additions dans le parcoureur de positions afin d'envoyer une commande au robotino qui va imposer les valeurs des vitesses pour chaque moteur
Cette fonction compare aussi la position actuelle du robot avec des valeurs seuils et définit la variable test-pos propre au robot qui sera utilisé pour indiquer si un déplacement a été achevé.
Le schéma de câblage ci-dessous donne la structure de la commande pour déplacer le robot vers la droite:



Fonction du déplacement à droite

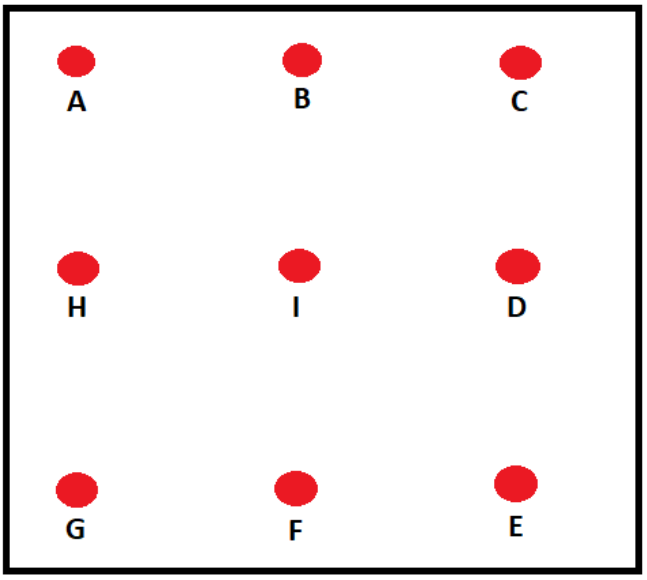
On définit de la même façon les fonctions propres aux autres déplacements en:

- Soustrayant 500 de l'ordonnée de la position actuelle du robot pour le déplacer vers la gauche.

- Ajoutant 500 à l'abscisse de la position actuelle du robot pour le déplacer vers l'avant.

- Soustrayant 500 de l'abscisse de la position actuelle du robot pour le déplacer vers l'arrière.

En tenant compte que le du fait que le robot ne peut effectuer que des déplacements sur une distance constante, Nous avons défini les trajectoire de cette manière:

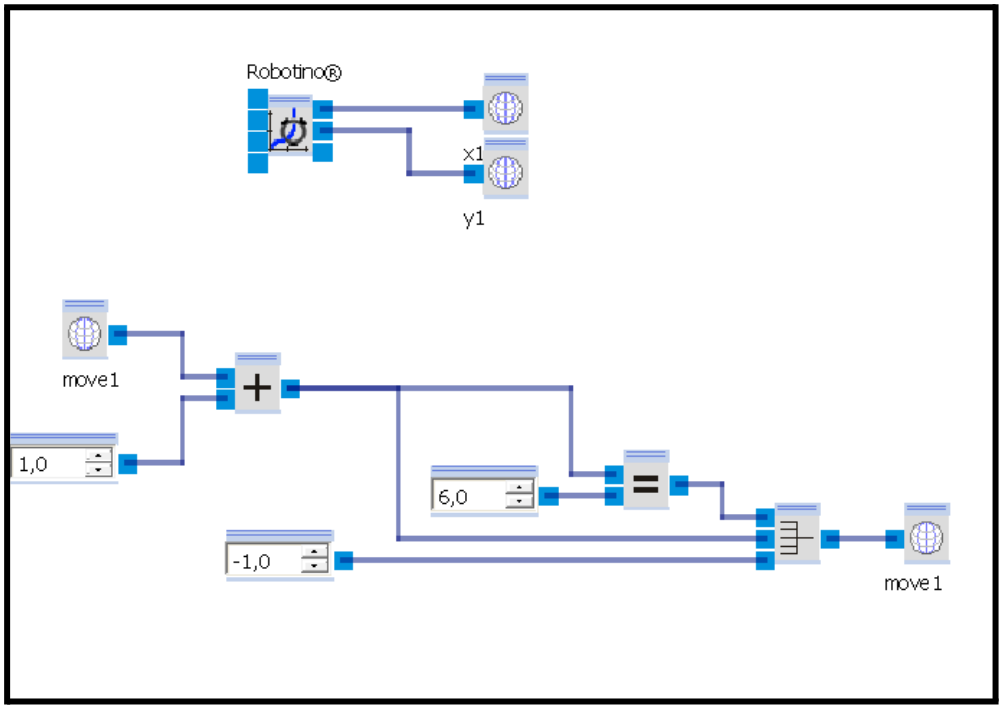


La distance entre 2 points voisins représente la constante qu'on ajoute aux coordonnées du robot qui est égale à 500 dans notre cas.
Le robotino ne peut se déplacer que d'un point donné vers son voisin et ne peut pas se déplacer en diagonal, donc pour aller de A vers C ce dernier doit aller d'abord vers B puis passer de B vers C.
Dans notre cas le premier robotino doit parcourir le chemin EFGHIDE donc il passe d'un point à l'autre dans cet ordre en allant à partie de sa position initiale (E) deux fois à gauche pour arriver à G puis une fois en haut pour atteindre H ensuite deux fois à droite pour arriver D et enfin une fois en bas pour arriver à sa destination (E).

Cette succession de déplacements est réalisée grâce à la fonction next move qui comme son nom l'indique donne le prochain mouvement à effectuer. Ceci est fait en incrémentant la variable move qui est comprise entre 0 et 6 et selon la variable de celle-ci le robot réalise un des 4 déplacements possibles.
La variable move est calculée en début de la commande de chaque robotino et une divergence en ou est introduite pour traiter les différents cas en vérifiant la valeur de move.

Toujours en partant de l'exemple du premier robot, la variable move est initialisée à 0 et le robot procède de la manière suivante :

1 - Effectue deux fois le déplacement vers la gauche (move = 0 ou move = 1).
2 - Part en haut (move = 2).
3 - Effectue deux fois le déplacement vers la droite (move = 3 ou move = 4).
4 - Exécute la fonction de déplacement vers le bas .(move = 5)



Fonction next_move

Voici deux vidéos qui montrent le fonctionnement de deux Robotino :

commande robotino
commande simultanée de deux robots

2) Définition de la récupération des données

Afin de récupérer les données du capteur nous avons créé un bloc dans le grafcet qui réalise cette tâche en continu.

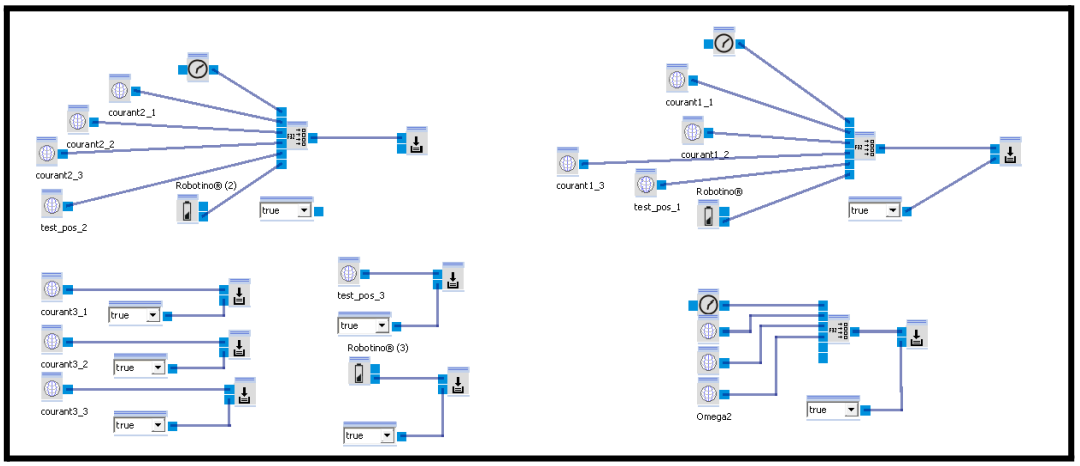


Schéma de la récupération de données.

Afin de récupérer les données nous avons rajouté un bloc permettant d'écrire chaque valeur des données en entrée sur une ligne.

Nous sauvegardons dans des fichier différents les valeurs des différents robotino:

- Le 1er fichier contient le courant et la valeur de la batterie du premier robot.
- Le 2ème fichier contient le courant et la valeur de la batterie du second robot.
- Le 3ème fichier contient le courant et la valeur de la batterie du second robot.
- Enfin le dernier contient les valeurs des vitesses des différents moteurs.

Le bloc de sauvegarde doit avoir une valeur true sur une de ses entrées. Ainsi tant que le bloc est enclenché, on continue de récupérer la valeur des différentes grandeurs.
Des données ici du même échantillon sont écrites en colonnes, deux lignes différentes représentent deux échantillonnages à des durées différentes.

Communication avec les Robotinos

Maintenant que nous avons réalisé notre parc de Robotino, il est primordial de communiquer avec ces derniers de façon continue et en temps réel, la communication avec le réseau de Robotino nous permettra de :

- Récupérer les données en continu pour s'assurer du bon fonctionnement des robots avec notre algorithme de réseau de neurones (nous viendrons sur ce point plus tard)
- Envoyer des données au Robotino afin soit de l'arrêter si un problème survient, ou soit de commander un robotino si l'un du réseau est défectueux
- Faire la communication entre chaque Robotino pour que ces derniers soient au courant des différents états des autres robots.'r(

Nous devons alors programmer une solution qui pourra :
- Récupérer et envoyer des données au Robotino
- Entraîner les données de différents Robotino sans que le temps d'exécution soit grand (dans le cas de parc avec des milliers de Robotinos)

1) Récupération et envoi de données

Il existe plusieurs méthodes pour récupérer les données à travers différents langages : Langage C++, Python, Matlab, LabView, ect

La méthode la plus connue à ce jour, est d'utiliser le serveur Restful du Robotino, cependant à quoi consiste cette méthode ?

Avec le dernier package robotino-daemons, l'interface Web de Robotino est basée sur un serveur Restful écoutant sur le port 80 de Robotino. Le serveur met non seulement les données à la disposition de l'interface Web de Robotino, mais peut également être utilisé pour accéder aux capteurs et aux acteurs de Robotino. Rest Api fait partie de toutes les images de V4 et versions ultérieures.

Pour tester l'existence de cette interface serveur, nous avons essayé de récupérer des données de la simulation de notre parc à Robotino à l'aide de Matlab, à l'aide cette fonction :

```
img=webread('http://192.168.0.1/cam0')
imshow(img)
```

Cependant, cette commande existe seulement sur les versions supérieures de MATLAB 2014, nous avons des versions antérieures à 2014. Comme depuis le début du projet, nous avons entraîné des bases de données à l'aide des algorithmes de machine learning uniquement Python, nous avons opté pour la méthode de récupération de données sur le serveur RestFul avec Python.

Pour récupérer les données dans le serveur Restful à l'aide de Python nous allons importer les librairies suivantes :

- **Requests** : permet d'envoyer très facilement des requêtes HTTP/1.1 sur des serveurs
- **Json** (JavaScript Object Notation) : spécifié par la RFC 7159 et par l'ECMA-404, est un format d'échange de données léger inspiré de la syntaxe littérale d'objet JavaScript.

L'algorithme est le suivant :

- Nous allons construire l'url où sont stockées les données à recevoir
- Nous allons faire une requête HTTP (de récupération de données) sur le serveur RestFul en spécifiant l'adresse IP du robotino : `r=requests.get()`
- Si la connexion a réussi, alors on réalise un échange de données à l'aide de la librairie Json :`r.json()`
- Sinon on propage une erreur "RunTimeError" si la connexion à échoué

Voici un exemple pour la récupération des informations sur la batterie

```
# Recupere le niveau de la batterie
def get_festoolcharger(ip):
  #Obtenir l'adresse de données
  festoolcharger_url = "http://" + ip + "/data/festtoolcharger"
  #Obtenir les données de l'adresse
  r = requests.get(url = festoolcharger_url, params = PARAMS)
  #Si non erreur
  if r.status_code == requests.codes.ok:
    #Stockage des données dans data
    data = r.json()
    return data
  #Si erreur -> RuntimeError
  else:
     raise RuntimeError("Error: Récupération de %s avec les paramètres %s
ratée", festoolcharger_url, PARAMS)
```

Pour l'envoi de données, nous allons réaliser le même schéma d'algorithme sauf qu'au lieu de réaliser une requête d'obtention de données `requests.get()`, nous allons faire une requête HTTP d'envoi de données : `requests.post()`

Cependant, nous n'avons pas réussi à récupérer les données de cette façon, le problème ne provenant pas du code, mais du serveur RestFul. Nous obtenons une `RuntimeError` si nous nous trompons sur l'adresse ip du robotino (surtout au niveau du port 80), mais en spécifiant le bon port, la fonction était en boucle infinie sur l'échange de données. De plus que nous n'avons pas réussi à nous connecter sur le serveur avec la simulation de robots. C'est entre autres à cause de ce problème que nous avons essayé de travailler avec de vrais Robotino (ce qui rallonge le temps de conception du parc). Il se peut aussi que les robotinos que nous utilisons ne soient pas mis à jour ou que ces derniers ne créent pas de

serveur. Ce qui est problématique pour la partie où nous devons arrêter le robot en panne ou bien commander un robot qui l'un des robots est en panne.

Pour la suite du projet, nous allons alors récupérer les données directement sur RobotinoView (ce que nous faisons déjà pour créer la database), et nous allons adapter notre code pour que le programme ouvre toutes les "x" secondes, un fichier contenant une base de valeurs actualisée.

2) Gestion de l'entraînement des données pour le machine learning

Dans cette partie, il est question de trouver une solution pour entraîner les données de différents Robotino sans que le temps d'exécution soit grand.

La première méthode est la méthode itérative, qui consiste à entraîner les paquets de données robots par robots, bien que cette méthode soit facile à programmer, son temps d'exécution est extrêmement long (si on prend l'exemple un parc de 1000 robots, si l'algorithme met 10 secondes pour entraîner un paquet de données, notre programme mettra 2 heures et 47 minutes pour tester juste une fois l'ensemble des robots, d'ici là nous avons largement dépassé la limite avant que la production soit en arrêt total et que le robot défectueux soit totalement hors d'usage)

La deuxième méthode consiste à lancer un programme de machine learning s'occupant en continu d'un robot sans interruption dans un PC, bien que cette méthode soit le plus rapide possible, cette méthode est très coûteuse en matériel et en consommation d'énergie (si 1000 robots alors 1000 pc doit être utilisé). De plus,il sera compliqué de communiquer entre les PC pour échanger des informations.

La troisième méthode, celle que nous avons réalisée est une programmation multithread, c'est-à-dire que 1 thread = 1 sous-programme, et 1 sous-programme = 1 programme de machine learning. La communication entre les threads est facile (utilisation de variables globales.

Principe de fonctionnement :

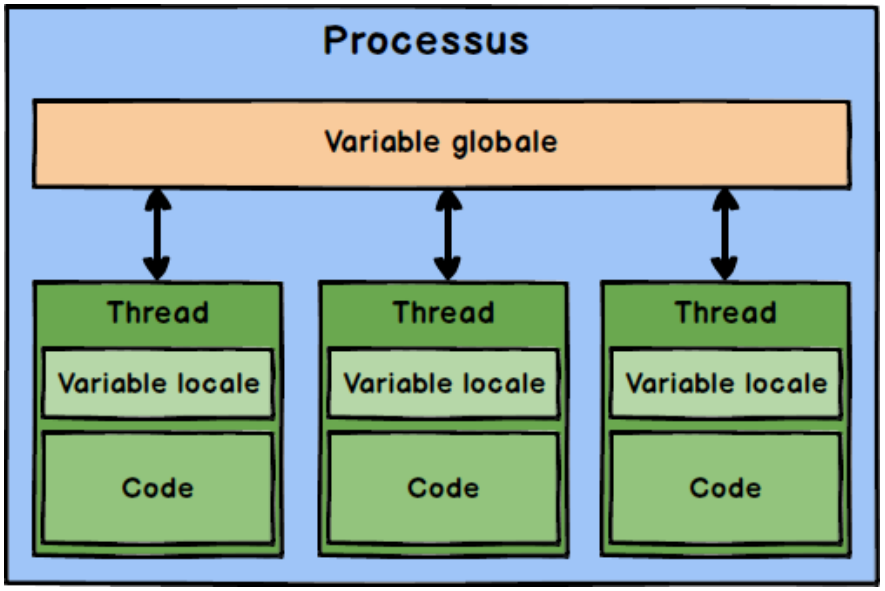
Les threads sont généralement contenus dans les processus. Plusieurs threads peuvent exister dans le même processus. Ces threads partagent la mémoire et l'état du processus. En d'autres termes: ils partagent le code ou les instructions et les valeurs de ses variables.

Chaque processus a au moins un thread, c'est-à-dire le processus lui-même. Un processus peut démarrer plusieurs threads. Le système d'exploitation exécute ces threads comme des « processus » parallèles. Sur une machine à processeur unique, ce parallélisme est obtenu par la programmation des threads ou le découpage temporel.

Les avantages des threads sont :

- Les programmes multithread peuvent s'exécuter plus rapidement sur les systèmes informatiques avec plusieurs processeurs, de manière simultanée.
- Un programme peut rester réactif à la saisie.
- Les threads d'un processus peuvent partager la mémoire des variables globales. Si une variable globale est modifiée dans un thread, cette modification est valide pour tous les threads. Un thread peut avoir des variables locales.

La gestion des threads est plus simple que la gestion des processus pour un système d'exploitation. C'est pourquoi ils sont parfois appelés processus légers.



Fonctions liées au threads :

```
Thread(target=<nom_fct>, args=(<mesargs>,));
```

→ Créer un thread avec comme paramètre le nom de fonction du thread (la fonction est à créer), et les arguments

```
start()
```

→ Débute le thread

```
join()
```

→ Le programme attend que le thread soit fini pour passer à la suite

Il est possible aussi d'ajouter des sémaphores, utile lorsque nous avons besoin de beaucoup de thread, un thread t1 prendra une ressource, cette ressource sera alors utilisée par un thread t1, si un thread t2 a besoin de cette ressource, alors le thread se mettra en file d'attente tant que le thread t1 ne rend pas la ressource.

Algorithme de la fonction de thread :

Il y a autant de threads que de robotino, chaque thread lancera cet algorithme en boucle infini suivant avec en paramètre les informations du robot en question, et le nom de fichier des données (qui sera remplacé par la récupération des données par serveur RestFul)

- Récupération de données
- Faire la prédiction avec les données obtenues
- On parcourt notre prédiction et :
    - Si nous avons un défaut, alors on augmente un compteur
    - Si nous avons un résultat ok, alors on reset le compteur
- Si le compteur atteint un temps fixe (déterminée par le programmeur), alors l'algorithme détecte un problème, il met fin à la boucle infinie, et met à jour le statut du robot défectueux
- Sinon, il n'y a pas de problème, alors tout va bien on reset le compteur

On met en place ce système de compteur en place puisqu'il est possible que l'algorithme de machine learning se trompe ou alors que la machine subisse un léger problème qui n'est pas critique. Si l'algorithme détecte des problèmes pendant un temps donné, alors dans ce cas, la machine est défectueuse.

Le programme s'arrête si et seulement si tous les threads sont terminés (soit quand tous les robots sont incapable de travailler)

Test unitaire du programme principale + threads :

Comme lorsque quand nous avons programmé le code, notre dataset n'était pas encore disponible, nous avons réalisé des tests unitaires sur notre code :

- Tous les threads se lancent correctement
- Les threads tournent toujours lorsqu'un s'est arrêté
- Lorsqu'un thread s'est arrêté, les données sont bien actualisées
- Le programme s'arrête bien quand tous les threads sont finis

Les tests ont été réalisés en simulant une prédiction de données provenant d'une machine défectueuse (un tableau rempli de "1"), puis de prédiction de données provenant d'une machine saine (un tableau rempli de "0")

3) Améliorations possibles du notre code et possibilités de perspectives

L'idéal pour notre code, serait d'utiliser les fonctions de récupération en utilisant le serveur Restful, si aucune solution n'est trouvée à ce problème, il faudra alors concevoir un nouveau système de récupération de données dans un autre langage, la façon dont nous récupérons les données n'est pas optimale l'heure actuelle.

Nous n’avons pas eu le temps de tester notre code en situation (problème de temps et de récupération de données) mais notre code fonctionne d’après les tests que nous avons effectués.

Réseau de neurones

Durant les semestres précédents, nous avons principalement utilisé les algorithmes classiques de Machine Learning tels que la régression linéaire, logistique ou la méthode Naive-Bayes. Pour ce semestre, nous allons utiliser un sous-domaine du Machine Learning qui est le Deep Learning et qui possède certains avantages par rapport aux algorithmes étudiés précédemment, notamment dans l'application à un réseau logistique autonome.

1) Pourquoi utiliser un réseau de neurones ?

Suite à nos recherches bibliographiques, nous avons retenu certains avantages du réseau de neurones pour notre application :

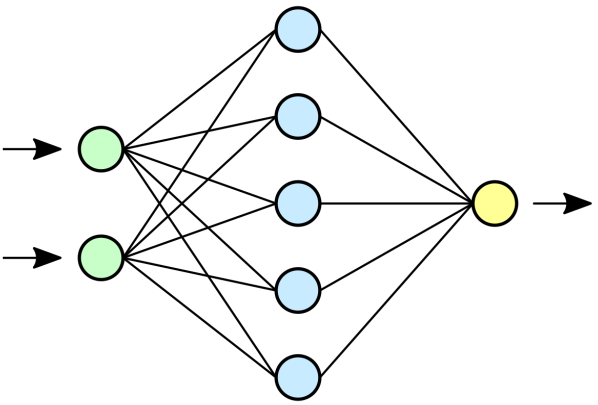
- Il existe de nombreux outils (librairie sklearn notamment) permettant de mettre en place un réseau de neurones facilement
- Il est très pratique pour des applications de récupération de données en temps réel car se met à jour au fur et à mesure que de nouvelles données apparaissent
- Il peut utiliser des variables discrètes et continues (contrairement à la régression logistique par exemple)
- Il peut avoir plusieurs variables de sortie
- Il est très flexible et peut être utilisé dans énormément de domaines

Il existe beaucoup de types de réseau de neurones, nous avons choisi d'utiliser le **MLP Classifier** qui est un algorithme de classifications afin de pouvoir trier les données saines (robot fonctionnel) et malsaines (robot dysfonctionnel).

2) Fonctionnement du réseau de neurones

Le fonctionnement d'un réseau de neurones est inspiré du comportement d'un cerveau humain. A l'image d'un cerveau humain qui va apprendre en observant énormément, un réseau de neurones va apprendre en analysant un grand nombre d'échantillons, afin de les classer en fonction de leurs similarités et de leurs différences.

On schématise un réseau de neurones simple (une seule couche cachée) comme cela :



Réseau de neurone simple

- Chaque neurone est connecté à un autre neurone en entrée et en sortie (sauf les neurones situés aux extrémités)
- Le neurone reçoit en entrée une information numérique x_i $(1 \leq i \leq N)$ avec N le nombre de données reçues par le neurone, chaque information est valorisée par un coefficient ω_i.
- Le neurone effectue la moyenne pondérée S de ses entrées puis passe cette valeur dans une fonction d'activation f qui permet d'adapter cette valeur aux caractéristiques de la sortie désirée.
- Le résultat $y = f(S)$ est ensuite passée en sortie du neurone.

L'objectif de l'entraînement de l'algorithme est de trouver les bons paramètres ω_i tels que $y = f(X)$ soit correct avec X l'ensemble des paramètres passés en entrée du réseau de neurones.

On commence donc par initialiser des ω_i aléatoirement puis l'algorithme les corrigera par principe de "backpropagation" qui consiste à commencer par corriger les ω_i en commençant par les neurones de la dernière couche et en finissant avec ceux de la première couche.

Dans le cas d'une classification binaire, nous disposons de 2 neurones en sortie, un correspondra à "robot fonctionnelle" et l'autre à "robot dysfonctionnelle".

3) Le programme réalisé et les tests

Dans un premier temps, nous devons intégrer les données récupérées du logiciel RobotinoView puis les mettre en forme pour les intégrer dans notre base d'entraînement de notre algorithme de Machine Learning.

Pour ce faire, on lance la fonction `load_data()` programmée par nos soins :

```
def load_data(data,data2,build_db):
  fichier = open(data,'r')
  first=1
  for str in fichier:
    liste = str.split()
    for j in range(len(liste)):
      liste[j]=float(liste[j])
    liste = np.array(liste)
    if(build_db):liste = np.hstack([liste,0])
    if(not first):tab = np.vstack([tab,liste])
    else:tab=liste
    first=0
  if(build_db):
    fichier = open(data2,'r')
    first=1
    for str in fichier:
```

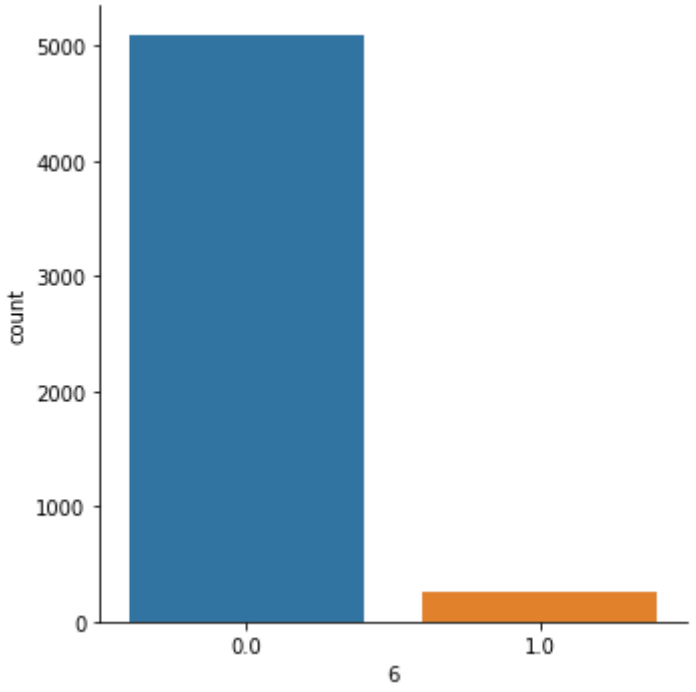
```
      liste = str.split()
      for j in range(len(liste)):
        liste[j]=float(liste[j])
      liste = np.array(liste)
      liste = np.hstack([liste,1])
      tab = np.vstack([tab,liste])
      first=0
  return tab
```

Voici les quelques fonctions utiles pour comprendre ce code : la fonction prend en paramètre les deux noms de fichiers et des flags (si cette fonction est appelée pour créer la database ou non)

- `split()` permet de ranger dans une liste, les mots séparés d'espaces
- `float(<str>)` permet de transformer en flottant une chaîne de caractères
- `hstack()` de la librairie Numpy permet de concaténer horizontalement (permet d'ajouter des colonnes par exemple
- `vstack()` de la librairie Numpy permet de concaténer verticalement (permet d'ajouter des lignes par exemple

On regarde si nos données comportent des données manquantes et on regarde le nombre de données saines et malsaines (le "6" correspond à la colonne de la target)

```
sns.factorplot(6,data=data,kind='count')
```



On remarque que nous avons plus de données saines que malsaines, l'idéal serait d'avoir un taux équilibré entre les deux. Il est possible d'ajouter des données malsaines mais cela va induire en erreur notre algorithme. On laisse comme ça pour l'instant, on regardera la précision de notre système si l'algorithme peut détecter des erreurs

```
import missingno as msno
msno.matrix(data)
```

L'absence de lignes blanches montre que nous n'avons pas de données manquantes.

La base ainsi faite, on isole les target des features, et on supprime la colonne correspondant au temps d'acquisition à l'aide de la fonction `drop()` de la librairie Numpy

Puis on entraîne notre base à l'aide de la fonction suivante, nous utilisons les algorithmes de réseaux de neurones de la librairie Scikit-learn :

```
def fit_model(Xtrain,Ytrain):
  #Modèle réseau de neurones
  modele_resNeur = MLPClassifier(random_state=1, max_iter=300)

  #training
  modele_resNeur.fit(Xtrain,Ytrain)

  return modele_resNeur
```

Il est toujours possible de splitter notre base en données d'entraînement et de test pour réaliser des tests de performance de la database et de l'algorithme :

```
#Séparation des données vers des ensembles de test et d'entrainement (ici
20% test et 80% d'entrainement)
Xtrain, Xtest, ytrain, ytest = train_test_split(X,y, test_size = 0.4,
random_state = 100)
```

Nous pouvons tester notre modèle en splittant nos valeurs comme expliqué ci-dessous, on entraîne la base de test en fonction des données de la base d'entrainement :

```
def pred_model(Xtest,modele_resNeur):
  y_pred = modele_resNeur.predict(Xtest)
  return y_pred
```

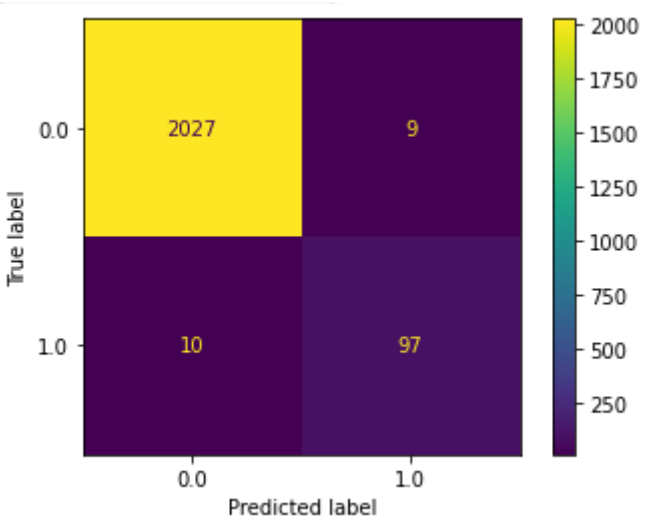
Cette fonction retourne la liste des targets ainsi prédits, on l'a compare avec les vraies valeurs

```
#précision du modèle
precision = accuracy_score(ytest, ypred)
precision
```

```
0.9911339244050397
```

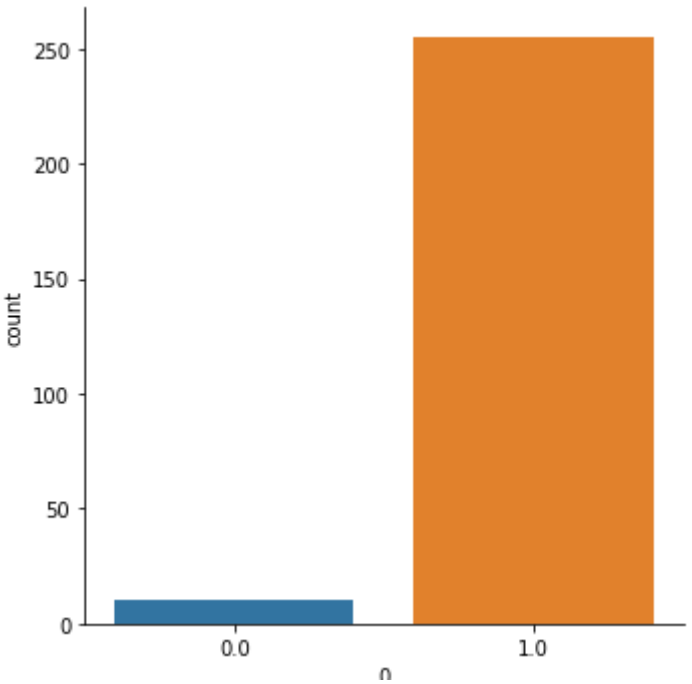
:

```
from sklearn.metrics import plot_confusion_matrix
plot_confusion_matrix(modele_resNeur, Xtest, ytest)
plt.show()
```



On remarque un score de précision de 0,99, ce qui est honorable, on pourrait croire que notre algorithme ne prédit que des targets "Machine saine" mais ce n'est pas le cas, c'est ce qu'on voit dans la matrice de confusion.

On peut également essayer d'entraîner des valeurs provenant d'une machine défectueuse (nous avons récupéré des données en ajoutant de défaut manuellement sur le logiciel RobotinoView :



On voit alors que dans ce cas, notre modèle remarque que les données d'entrée proviennent d'un robot possédant un défaut (ce qui est le cas)

Notre database est très bien, il manque juste plus de valeurs malsaines pour qu'elle soit parfaite, mais le score tel qu'il est excellent.

L'intégralité du code (2 parties confondues) est disponible à ce lien : https://colab.research.google.com/drive/1LZIlqdimblozKZAoeTKNBlbeWL3EUiSF?usp=sharing

Conclusion

En conclusion, notre projet a pris une direction toute autre de celle prise lors des deux semestres précédents. En effet, lors de ce semestre nous nous sommes attelés à la partie pratique en travaillant directement sur les Robotino afin de créer un système de gestion d'entrepôt autonome. On a mis de côté les fonctions et les algorithmes que l'on utilisait lors des semestres précédents au profit d'un réseau de neurones codé sur python. C'était en effet la technologie adéquate pour ce que l'on voulait faire sachant qu'on ne l'avait encore jamais exploité.

Dans un premier temps on a dû définir une situation ainsi qu'un environnement pour notre parc de Robotinos travaillant ensemble et ainsi nous avons défini un scénario dans un entrepôt de logistique avec des chemins prédéfinis pour les robots pour qu'ils puissent travailler ensemble et communiquer afin de venir à bout de leurs tâches. Nous avons donc dans un même temps développer le grafcet des Robotino tout en cherchant des informations sur le machine learning dans un réseau logistique (nécessaire pour notre application avec les Robotino).

Puis, lorsque que la simulation fonctionnait nous nous sommes mis à créer notre propre base de données avec les informations à l'aide du logiciel RobotinoView. Mais, c'est ici que vint notre premier problème de ce semestre, il était impossible de récupérer les données nécessaires pour construire une base de données fonctionnelle uniquement à l'aide de la solution logicielle. On a donc dû très vite passer sur un système réel, c'est-à-dire de vrais Robotino pour pouvoir générer cette base de données.

Une fois cette base de données générée correctement à l'aide de deux Robotino, nous devions maintenant y inclure des défauts pour pouvoir les traiter par la suite à l'aide de notre algorithme de réseau de neurones. Dans l'idée de notre simulation de gestion d'entrepôt, on se devait de traiter cette base de donnée en temps réel, or on a eu un autre problème qui ici. Le problème était qu'on ne pouvait pas se connecter au port 80 des Robotino directement, il était donc impossible de leur donner des ordres en fonction de la base de données qu'ils génèrent en temps réel. On a donc malheureusement dû simplifier notre idée de base, on ne traite ainsi la base qu'uniquement après que le robot ait fini sa tâche et non pendant comme le suggérait le modèle temps réel que l'on souhaitait mettre en place.

On a donc encore une marge d'amélioration possible pour atteindre notre objectif final de l'entrepôt de logistique auto-géré à l'aide de la communication entre les Robotino ainsi que de l'analyse en temps réel de la dataset générée par ses derniers.