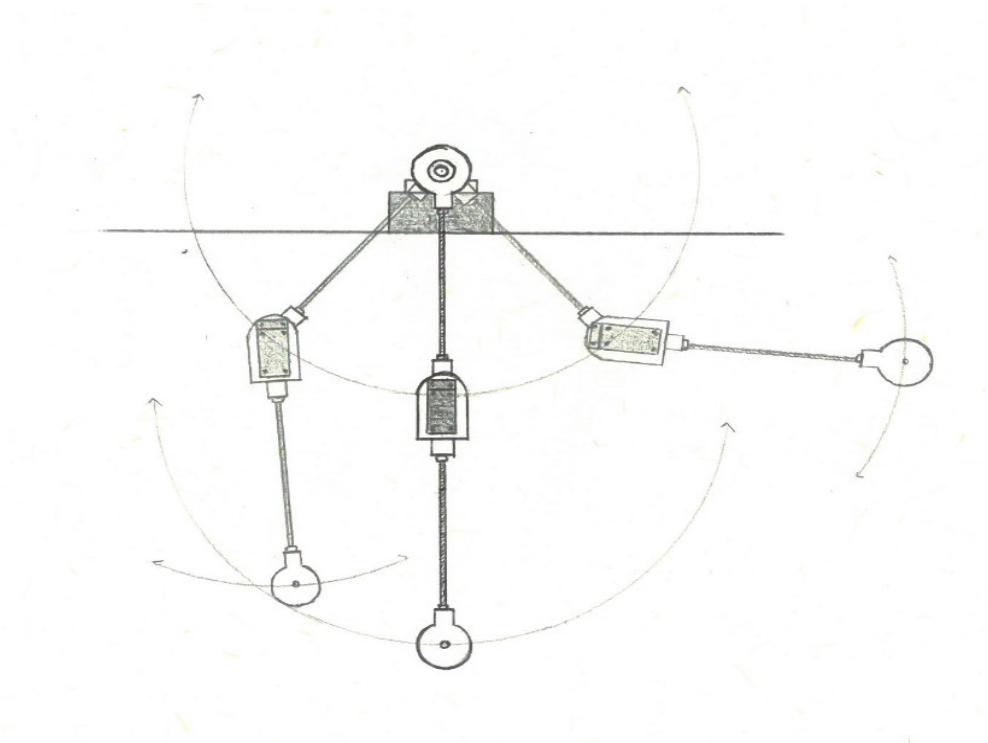
TAVIGNOT Baptiste

VIALA Martial

SENDRA Thomas

Apprentissage de l'équilibre pour un pendule inverse (Deep Learning)



Chef de projet : M. Sébastien LENGAGNE

Projet Polytech Peip 2ème année (2018-2019)

POLYTECH CLERMONT-FERRAND

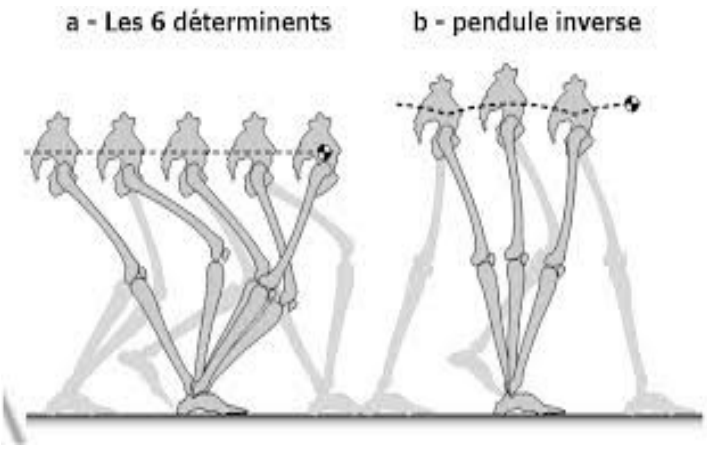
Table des matières

I. La marche bipède et le pendule inverse

Depuis quelques années, l'intelligence artificielle a pris une ampleur exponentielle. C'est devenu un enjeu pour l'avenir en matière de production, notamment industrielle. Pour arriver à un tel résultat, le plus « simple » est de créer une machine qui puisse penser et agir comme le ferait un humain. L'Homme se sert donc de ses propres connaissances pour arriver à cet objectif.

La machine, une fois fabriquée, peut être comparée à un enfant. Celui-ci ne connait rien et doit recevoir une énorme quantité de données afin qu'il puisse de lui-même créer ses propres réflexions. Le cas de la marche bipède est un exemple de cette phase d'apprentissage.

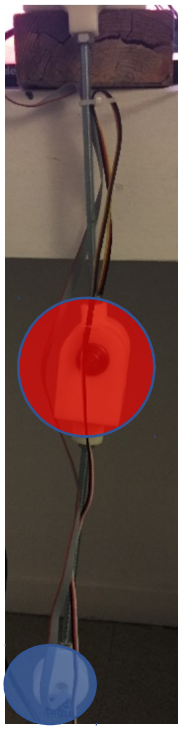
L'enfant doit comprendre qu'il faut établir un équilibre avec ses deux jambes : lorsqu'une jambe touche le sol de façon tendue, l'autre se plie vers l'avant et retombe vers le sol de façon également tendue.



Issue de la thèse «*Le mouvement segmentaire au service du déplacement dans la marche*» *soutenue à l'université Rennes 2 en 2014*

On peut modéliser la phase où la jambe se balance pour se tendre par le phénomène mécanique du pendule inverse. Ainsi, marcher revient à placer sa jambe dans un intervalle $[-\sigma, \sigma¿$ où σ est l'angle entre l'axe des ordonnées et la jambe dans un repère galiléen. Par

expérience, nous savons que cet angle doit être le plus petit possible afin de minimiser le risque de chute. C'est pourquoi, chercher à avoir un pendule complètement tendu s'avère très pertinent. Pour être au plus près de la réalité, nous avons à notre disposition le double pendule inverse ci-dessous.



Notre double pendule est composé d'un moteur (zone rouge), qui va commander la partie inférieure du bras, ainsi que d'un capteur (zone bleue), qui agit en tant que gyromètre (calcule la vitesse angulaire) et accéléromètre (calcule l'accélération de ce que nous appellerons le bras articulé).

En effet, on peut aisément le comparer à une jambe puisque nous contrôlons notre articulation du genou qui entraîne les différentes parties de celle-ci. Ici, le moteur va activer le mouvement de l'extrémité du bras, ce qui va provoquer celui de la partie non contrôlée par le moteur, que nous appellerons « lâche ». Le capteur va nous donner la position géographique du bras dans le repère et nous permettra, plus tard, d'utiliser ses informations. Le but étant, à partir de la position de repos (image ci-contre), de tendre le bras verticalement vers le haut à l'aide d'un programme basé sur le Deep-Learning. Avant de continuer dans l'aspect technique de notre projet, nous allons tout d'abord nous pencher sur la théorie concernant le Deep-Learning.

II. Introduction au Deep Learning :

i. L'histoire de l'Intelligence artificielle

Dans son article "Computing Machinery and Intelligence" publié en 1950, Alan Turing discute pour la première fois de la conscience d'une machine (Can machine think ?[1]). Il propose notamment une expérience de pensée appelée « The Imitation Game » ou « le Test de Turing » dont le principe est le suivant :

Dans deux salles séparées se trouvent :

- Un ordinateur (A)
- Une personne humaine (B)

Ainsi qu'un "interrogateur" humain (C).

Le but de C est de découvrir lequel de A ou B est un ordinateur en se basant sur une interaction verbale uniquement. Si C ne réussit pas, alors, l'ordinateur a passé le test. Cette expérience a pour but de répondre à la question suivante : L'ordinateur peut-il de façon naturelle converser avec une personne humaine ? S'ensuit alors une réflexion sur la méthode que l'on devrait adopter pour donner une conscience à une machine.

"Instead of trying to produce a programme to simulate the adult mind, why not rather try to produce one which simulates the child's?" ("Au lieu d'essayer de faire un programme qui simule l'esprit de l'adulte, pourquoi ne pas en essayer un qui simule celui d'un enfant").

Si nous reprenons le cas de la marche, lorsque l'enfant réussit à marcher une fois, il arrivera toujours à reproduire ce mouvement machinalement. En s'inspirant de cela, l'Intelligence artificielle va, en plus d'effectuer une tache qu'on lui a appris à faire, s'en souvenir et le reproduire efficacement et sans difficultés.

1Issue de « Computing Machinery and Intelligence »

À l'aide de cette article, l'idée d'un processus permettant à une machine de faire des activités comparables à celles humaines est alors née sous le nom d'intelligence artificielle donné par John McCarthy.

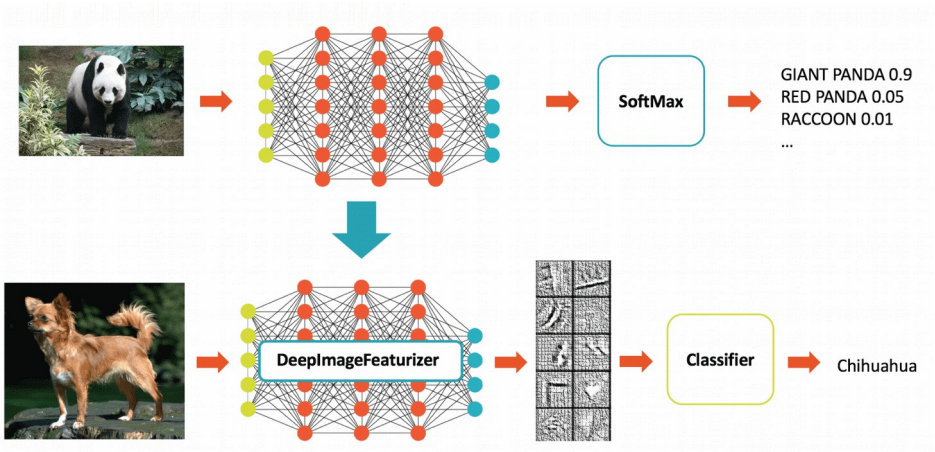
ii. Les différents types d'apprentissage

Il convient de bien choisir le type d'apprentissage qui s'adapte le mieux aux données et à l'expérience voulue. Tout d'abord, il faut savoir qu'il existe 2 types d'apprentissage : le supervisé et le non supervisé.

a) L'apprentissage supervisé

L'apprentissage supervisé consiste à fournir une énorme quantité de données à l'algorithme et lui donner la conclusion qu'il doit en tirer. Il va donc créer lui-même un lien entre les données et le résultat voulu. Il va s'entrainer à créer ce lien à l'aide d'un réseau de neurones, qui va reproduire cette phase de pensée et de connexions entre des connaissances. On va lui donner plusieurs données qui ne font pas parties de son ensemble de connaissances et il renverra un résultat compatible à la donnée suivant le lien voulu.

Pour donner un exemple concret, il y a le cas classique de la reconnaissance d'image. On donne à l'algorithme une énorme quantité d'images, par exemple de voitures. Pour chaque image, nous allons leurs attribuer une valeur de sortie qui représente le « résultat » (Par exemple reconnaitre la marque de la voiture). À l'aide des caractéristiques propres aux voitures de chaque constructeur, il va en déduire les liens existants avec la carrosserie ou le logo si celui-ci est visible et l'attribuer à la classe « nom de la marque de la voiture ».



Reconnaissance d'une image de panda et de Chihuahua[2]

Ici, l'image de panda passe par le réseau de neurones qui va utiliser tous les liens qu'il a créé pour en déduire la nature de l'animal. Plus on va vers la droite dans le réseau, plus les paramètres pris en compte seront fins pour une précision optimale. Le réseau peut s'apparenter à un filtre de plus en plus étroit pour ne faire passer que les informations qui concernent l'image, tel un entonnoir. Cette méthode est dite de classification puisqu'elle va regrouper chaque image dans une classe préexistante et entrainée.

Pour l'apprentissage supervisé, il nous faut donc déjà connaitre ses classes. Cependant, il y a un autre type d'apprentissage, ne faisant pas intervenir les classes.

b) L'apprentissage non supervisé

L'apprentissage non supervisé est le complémentaire de l'apprentissage supervisé puisqu'on ne va pas lui donner les valeurs en sortie. Il doit donc créer un lien de lui-même juste en comparant les données et ainsi rapprocher les données entre elles. Cette absence

2https://databricks.com/blog/2017/06/06/databricks-vision-simplify-large-scale-deep-learning.html?preview=true

de classe va cependant manquer, c'est donc l'algorithme qui va créer ses propres classes et placer chaque donnée dans l'une des classes correspondantes.

Si l'on reprend l'exemple de la reconnaissance d'image avec la voiture, il va déterminer les voitures qui ont le plus en commun et pouvoir dire « cette classe ressemble à d'autres que j'ai auparavant analysées, donc je peux affirmer que ce sont approximativement la même voiture ». Cette méthode est appelée le « clustering » (ou partitionnement de données en français).

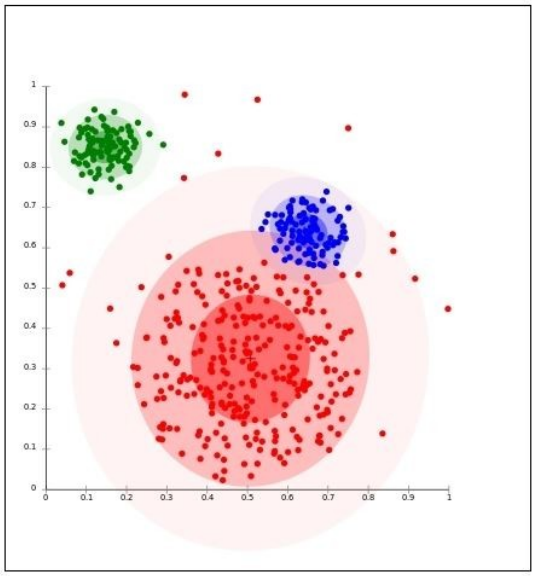


Schéma d'interprétation du procédé de clustering

On retrouve bien ici cette idée de regrouper en paquets des données en fonction de leur ressemblance avec les autres points. Les points rassemblés forment le « cœur » de la classe.

c) Le Deep Learning

Le Deep Learning (ou apprentissage profond) peut se faire de façon supervisé ou non. Il faut déjà savoir que l'ensemble des IA est composé de deux ensembles qui ne sont pas disjoints :

- La NLP (Natural Language Processing) : Cela constitue la reconnaissance de mots, de paroles permettant à un agent virtuel d'interagir avec l'homme.

- Le Machine Learning : c'est le fait de résoudre une tache demandée sans que le programmeur ne l'ait explicitement programmée pour cela.

On voit très bien que le Deep-Learning est inclut dans le Machine Learning puisque, effectivement, on veut que le cheminement pour aboutir au résultat vienne de l'algorithme et non qu'il soit superficiel dans des lignes de code brutes. L'intérêt est que l'algorithme prenne en compte une grande abstraction dans les données qu'on va lui donner et qu'il s'entraîne pour accomplir sa tâche au mieux.

Le Deep Learning est en pleine expansion actuellement, notamment avec les assistants vocaux (Google Home, Alexa, …) qui vont reconnaitre les paroles prononcées par l'utilisateur et vont pouvoir lui répondre en fonction des informations recueillies lors de la reconnaissance. De plus, à chaque utilisation, il y a un re-paramétrage du réseau de neurones toujours dans le but d'affiner les performances de l'algorithme. Cependant une question se pose :

De quoi est composé un réseau de neurones et comment effectuer son paramétrage ?

iii. Comment créer un réseau de neurones ?

Pour donner suite aux travaux d'Alan Turing, de nombreux scientifiques se sont penchés sur le sujet en essayant de retranscrire cette « humanité » dans une machine. Pour donner une conscience à celle-ci, il est évident que se baser sur le cerveau humain pour l'adapter informatiquement est la solution la plus facile à réaliser. C'est le principe du biomimétisme : on copie un phénomène naturel pour développer de nouvelles technologies.

C'est donc en reproduisant le processus neuronal de pensée que le réseau de neurones fut créé. Un réseau de neurones est composé d'au moins deux couches (la couche d'entrée et la couche de sortie). Il peut également y avoir des couches intermédiaires appelées couches *cachées*. C'est lorsqu'il y a des couches cachées dans le réseau que l'on parle de Deep Learning (Apprentissage profond).

Chaque couche est composée d'un nombre fini de neurones indépendants les uns les autres. Cependant, chaque neurone d'une couche (hors couche d'entrée), est dépendant des neurones de la couche précédente et cette liaison se traduit par un poids (similaire au rôle d'une synapse).

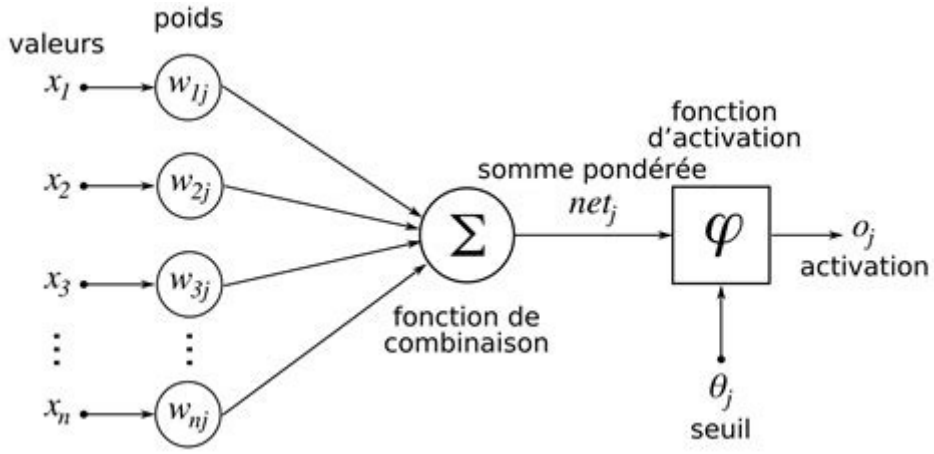


Schéma du calcul d'un neurone au sein du réseau

Pour passer de la couche n à n+1, on applique une fonction dite d'activation appelée sigmoïde.

$$\sigma(x)=\frac{1}{1+e^{-x}}$$

x est défini comme étant la somme des neurones précédentes multipliées respectivement par leur poids associés (x = σ).

On introduit alors une valeur arbitraire appelée le seuil. Si l'image de x, par la fonction sigmoïde, est supérieure ou égale au seuil, cela va « activer » le neurone (traduit par une valeur non nulle pour le neurone). Un neurone non activé va donc influencer le calcul, pour activer ou non, les neurones de la couche suivante.

Lorsque l'on regarde ce modèle, il peut paraître simple, cependant nous sommes face à un problème très difficile à résoudre : nous devons créer ce réseau alors que nous ne connaissons pas les poids associés à chaque neurone.

Par quel moyen parvenir à connaître ces valeurs ?

iv. L'optimisation

L'intérêt du réseau de neurones ici est que nous connaissons la couche d'entrée (données lors de l'initialisation de l'expérience) et la couche de sortie (données que l'on veut obtenir ; avec le bras, il s'agit de l'interprétation mathématique de la position tendue haute verticale).

Nous devons donc créer le réseau de neurones avec des valeurs de poids prises totalement aléatoirement. Ainsi, le réseau va retourner les valeurs de la couche de sortie, cependant, nous savons à l'avance que ces valeurs ne vont pas du tout correspondre à celles que l'on voudrait. Ainsi, en introduisant une fonction qui calcule l'erreur et en essayant de l'analyser, nous pouvons parvenir à réduire cet écart. Cette fonction est appelée la *fonction de coût*.

a) La fonction de coût

La fonction de coût est la fonction qui représente l'écart entre les valeurs retournées par le réseau et les valeurs théoriques. Cela se traduit par :

$$C(x,w)=\frac{1}{2}\left(y_j(x)-t_j(x)\right)^2$$

Où $y_j(x)$ est la valeur du x^{ième} neurone sortie par le réseau de neurones et $t_j(x)$est la valeur à laquelle nous voulons aboutir au x^{ième} neurone.

Le carré va servir à accentuer l'erreur et pouvoir étendre la fonction pour rendre le minimum plus facile à trouver. De plus, les poids redeviennent des variables puisque nous avons le résultat pour une combinaison très particulière de poids qui ne correspond pas à l'expérience.

Cela nous donne ainsi une fonction linéaire continue que l'on peut analyser pour trouver le minimum. Pour cela, nous allons utiliser le fait que nous voulons nous rapprocher de nos valeurs de sorties connues et ainsi procéder dans le sens inverse en adaptant les poids de la couche n à la couche 1 : cette méthode est appelée l'algorithme de rétropropagation.

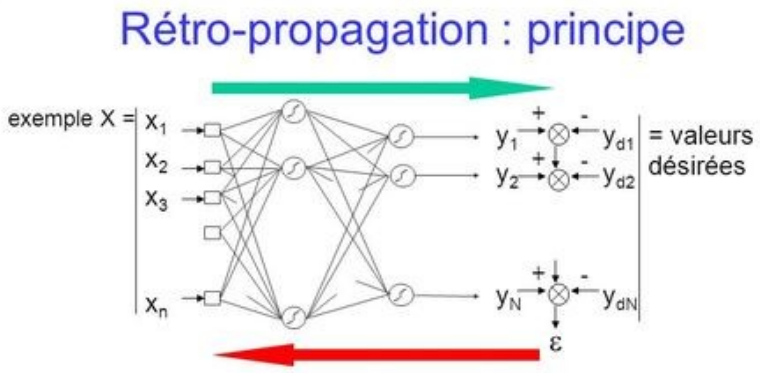


Schéma présentant l'idée générale de la rétropropagation

b) L'algorithme de rétropropagation du gradient

Pour cela, nous allons introduire la notion de gradient. Pour illustrer, le gradient peut être vu comme le coefficient de la pente d'une fonction. Ainsi, chercher le minimum revient à chercher la pente la plus « raide ». Schématiquement, nous pouvons imaginer un homme au sommet d'une chaine de montagne cherchant à arriver au point le plus bas possible. Instinctivement, le marcheur sait que plus la pente sera raide, plus il aura de chances de descendre rapidement. À chaque descente, il s'arrête et se demande à nouveau dans quelle direction la pente est la plus raide autour de lui. Il répète un nombre fini de fois cette étape et finira par atteindre le bas.

Cette analogie est la façon de procéder pour l'algorithme : Il prend un point au hasard et va chercher la pente la plus raide, ce qui correspond mathématiquement à la pente la plus petite (négativement) autrement dit l'opposé du gradient de la fonction de coût.

Le gradient est la somme des dérivées partielles d'une fonction (ici celle de coût). Seul le poids est une variable dans C(x,w) car nous connaissons déjà les valeurs que doivent prendre les x ièmes neurones.

$$\nabla C(x,w)=\frac{\partial C(x,w)}{\partial w_{ij}}$$

Afin de réajuster tous les poids, nous utilisons la formule suivante :

$\sum poids_i x\, neurone_i$ où $0 \leq \alpha \leq 1$ influence la vitesse de l'apprentissage

Plus α s'approche de 0, plus cela va retarder l'apprentissage puisque qu'il ne tient pas compte du gradient.

À noter que $C(x,w)=\frac{1}{2}(y_j(x)-t_j(x))^2$ est la valeur de poids après la nième rétropropagation.

Pour résumer, on entre des poids aléatoires dans le réseau. On calcule la fonction de coût issue de la différence entre la valeur sortie et celle prédite (ou voulue). Ensuite on réadapte les poids en utilisant la formule du gradient descendant. Cela forme la première étape, et ensuite on repasse dans le réseau de neurones avec les nouveaux, on recalcule l'erreur…etc. À la fin, nous voulons que l'erreur soit infime. Il faut donc de nombreux passage dans le réseau afin d'obtenir celui qui correspond à l'expérience voulue. Cela est appelé la phase d'entrainement.

Un fois l'entrainement fini, nous avons plus qu'à mettre les données d'initialisation. (car nous pouvons entrainer le réseau avec d'autres données, dites d'entrainement tant que l'on connait le résultat que l'on veut obtenir par l'expérience).

v. Un exemple : la fonction XOR

La fonction XOR (ou exclusif) est un opérateur booléen défini par la table de vérité suivante :

A	B	A **XOR** B
0	0	0
0	1	1
1	0	1
1	1	0

Tableau de vérité de la fonction XOR[3]

0 représente la valeur booléenne False et 1 la valeur True. Ici, on voit que l'algorithme va pouvoir s'entrainer avec 4 couples de neurones dans la couche d'entrée. En créant le réseau de neurone sur python, nous avons commencé par implémenter les 4 cas possibles avec le couple (0,1) et dans une autre variable le résultat attendu dans la couche de sortie.

Ici, nous avons créé, non pas un réseau de neurones, mais un perceptron. Un perceptron est « l'ancêtre » du réseau de neurones. C'est la version la plus simple d'un réseau avec aucune couche cachée.

3https://www.dyclassroom.com/logic-gate/xor-and-xnor-logic-gate

Exécution de notre programme python XOR

Il y a donc 2 neurones sur la couche d'entrée et nous initialisons deux poids aléatoirement reliés à un seul neurone sur la couche de sortie. Ainsi, avec la fonction sigmoïde et la rétropropagation, nous allons, avec de l'entrainement, arriver à des poids donnant des valeurs très proches de celle attendues (0.005 environ à la fin de notre programme).

Cette fonction est très simple et aurait très bien pu être faite sans perceptron cependant cela nous a permis de mieux appréhender toutes ces recherches théoriques afin de les voir de façon plus concrète. Cela nous amène à la prochaine étape, concrétiser nos connaissances dans le but de faire bouger ce bras articulé.

III. Les étapes de notre projet

Pour tendre le bras articulé, nous avons décomposé notre projet pour en tirer clairement les priorités :

- Faire bouger le bras articulé
- L'amener vers le haut
- Le garder en position tendue

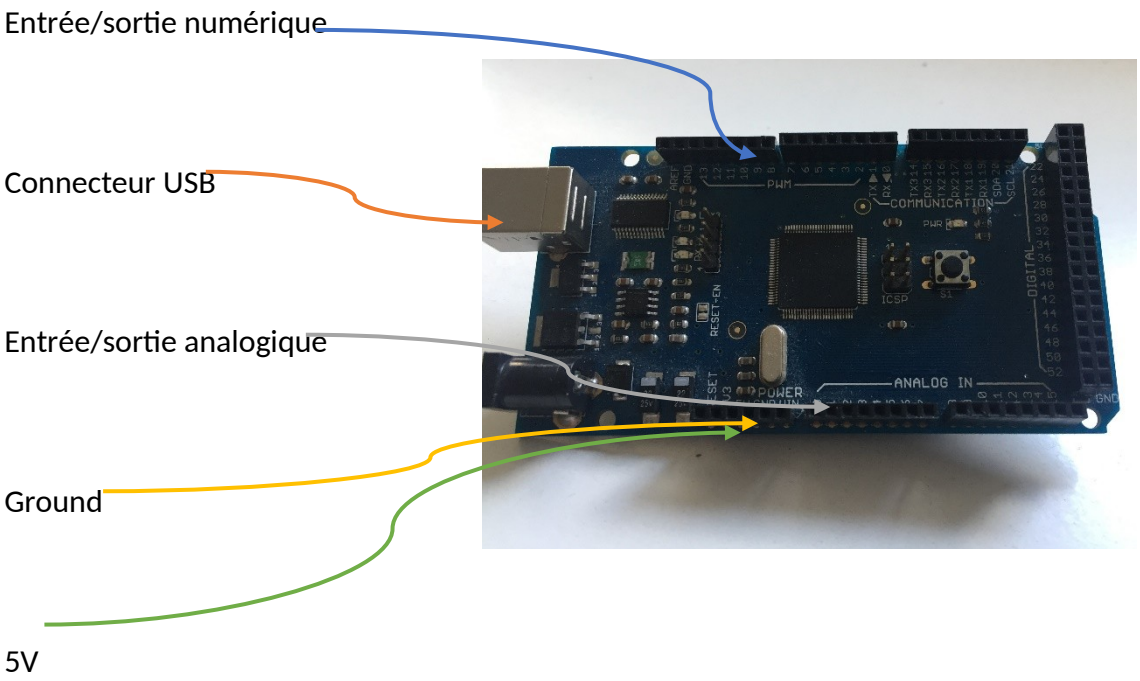
i. Le déplacement de notre bras

Avec notre bras articulé, il nous a également été fourni une carte Arduino. Arduino est une plateforme électronique open-source : elle est de licence libre, seule la carte est payante. Le grand atout d'Arduino est sa simplicité d'utilisation et sa diversité d'application ; nous pouvons allumer des diodes LED comme le fait une guirlande et surtout, ce qui va nous intéresser, commander un servomoteur. Il nous fallait donc connaitre comment fonctionnait Arduino pour l'utiliser correctement.

a) La carte Arduino

Ce n'est pas une véritable carte Arduino mais une copie. Elle fonctionne très bien cependant nous verrons plus tard que cela va avoir des conséquences sur les paramètres à régler.

Voici le modèle de carte Arduino qui nous a été fourni :



L'Arduino est un **circuit imprimé** où est intégré un **connecteur USB** qui va nous servir plus tard pour faire tourner notre programme. Le **microcontrôleur** est l'équivalent d'un

ordinateur à échelle très petite et agit en tant que mémoire vive, mémoire morte, entrée et sortie de valeurs. C'est le microprocesseur qui va intervenir dans l'exécution du programme et va se charger de le faire tourner.

Il faut également noter que la carte Arduino fonctionne par signal électrique à l'intérieur du circuit imprimé et c'est l'ordinateur auquel la carte est branchée qui va fournir cette électricité. Ce qui est particulièrement intéressant est qu'il peut tout aussi bien envoyer que recevoir un signal.

On appelle **pin** les entrées/sorties de la cartes Arduino qui peuvent être numériques ou analogiques. Les sorties numériques peuvent uniquement recevoir un signal : ils sont en mode OUTPUT. Les sorties analogiques peuvent aussi bien recevoir qu'envoyer un signal : ils peuvent être en mode INPUT ou OUTPUT.

Nous pouvons de plus ajouter des composants en les branchant à la carte par des fils électriques dans les différentes entrées/sorties de la carte suivant son type :

- Si c'est un moteur ou équivalent, il faut mettre la **Terre** sur le **ground** et la source de tension au **5V** et un autre fil à brancher sur l'un des pins**.**
- Sinon comme c'est le cas pour notre capteur, il faut juste le brancher à un pin suivant ce que l'on veut (numérique ou analogique), le chiffre n'importe pas, il faut se rappeler dans quel pin le composant est connecté pour le programme.

b) La programmation Arduino

Pour programmer la carte Arduino, nous avons dû apprendre les bases du langage C puisque Arduino est codé en pseudo-C. Cela nous a donc pris un peu de temps afin d'acquérir assez de connaissances pour faire des programmes intéressant en C. Puis, nous avons appris toutes les spécificités d'Arduino le distinguant d'un programme en C.

L'une d'elle; dès le démarrage du logiciel Arduino dédié à la programmation, lorsque l'on ouvre, nous voyons cela :

Une fonction **void** est une fonction qui ne renvoie aucune valeur. Ici la void **setup ()** va être la première fonction du programme à s'exécuter une et une seule fois, contrairement à la void **Loop (),** qui est une fonction qui va s'exécuter à l'infini et sans repasser dans le setup.

Le bouton est la compilation du programme : une fois le programme vérifié sans erreur (sinon message d'erreur), il va être compilé, c'est-à-dire traduit du langage humain vers le langage machine. En effet, la machine est capable uniquement de lire des valeurs binaires et c'est en cela que consiste la compilation.

Le bouton va lui permettre de téléverser : il s'agit d'envoyer le programme compilé vers la carte Arduino (et plus particulièrement au microprocesseur).

Cependant , lors de notre première utilisation d'Arduino, nous avons essayé de téléverser un programme vide uniquement, afin de tester si le microprocesseur recevait bien le programme. Un message d'erreur s'est affiché et durant de nombreuses heures, nous avons réfléchis à ce problème qui ne venait ni du programme (puisqu'il était vide) , ni du branchement de la carte (puisque la carte Arduino était reconnue par l'ordinateur). Seulement, elle était reconnue par défaut comme une carte Arduino Uno qui est l'une des cartes Arduino les plus utilisées. Or, cette carte qui , nous l'avons dit, est une copie, n'est pas comparable à celle d'un Uno mais d'un Arduino/Genuino Mega. Une fois ce changement fait dans la section « outils », et sélectionné le port série « COM 5 » qui est le nom du port USB de notre PC sur lequel est branché l'Arduino, le message d'erreur était toujours présent. En

cherchant dans de nombreux forums, nous avons vus que c'était un problème de driver. Il fallait donc tout simplement installer le driver correspondant à la carte. C'est donc avec cela que nous pu envoyer à terme notre programme au microprocesseur.

Cela nous permet donc de commencer à attaquer notre première étape .

c) Le premier déplacement du bras

Comme nous l'avons dit précédemment, le moteur du bras va être branché sur le ground , 5V et sur un pin quelconque.

Nous allons nous servir de la bibliothèque **Servo** qui a été spécialement créée pour activer un moteur connecté à une carte Arduino. Il s'agit donc simplement de spécifier le pin auquel est lié le moteur à l'aide de la commande **moteur.attach() .**

Il faut savoir que le bras articulé, tel un coude, ne peut bouger que sur l'intervalle [0,π], cependant nous nous servons davantage de l'angle en degré (donc l'intervalle [0,180]).

Avec la commande **moteur.write(),** il suffit de mettre l'angle (entre 0 et 179) en argument.

En testant une suite de commande write (donc en changeant l'angle), on a pu voir que le bras n'a pas forcément le temps d'aller jusqu'à l'angle desirée puisque les commande s'enchainent. Il faut dons intégrer un temps de pause grâce à **delay()** où l'argument est le temps voulu en millisecondes.

Ainsi le problème est quelque peu résolu mais il faut connaître le temps moyen que met le bras pour aller de 0 à 179 degrés (le 179 degré vient du fait qu'ammener le bras à 180 degré va faire forcer le moteur). Nous sommes donc capables de bouger le bras et connaissons le temps que celui-ci met pour effectuer un mouvement d'un degré (environ 0,5 ms).

Lorsque nous avons fait tourné le programme, nous avons remarqué un phénomène physique qui va beaucoup nous aider par la suite. L'énergie mécanique, apportée par le

moteur pour bouger le bras, va permettre de faire monter le bras de plus en plus haut grâce à un balancement engendré à chaque mouvement (Conservation de l'énergie mécanique). Ainsi, réussir à augmenter ce balancement peut amener le bras à être au dessus du socle constituant l'origine de notre repère.

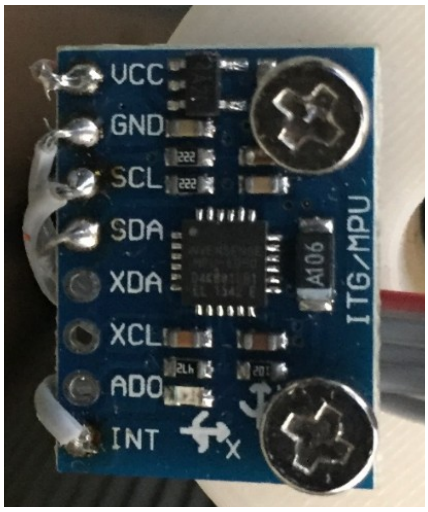
ii. L'utilisation du capteur

Maintenant que nous savons faire bouger notre bras, il nous faut utiliser le capteur pour en tirer les informations qui nous aideront à ce que le bras soit identifiable dans un repère galiléen. Il est tout d'abord intéressant de noter que seul un repère à 2 dimensions nous suffit puisque le bras nous permet seulement un mouvement de gauche à droite.

Nous devons donc avant tout savoir quelles données le capteur va être capable d'envoyer qui pourront nous être utiles pour la suite.

Ce capteur est un MPU-6050. En lisant la datasheet, nous avons appris qu'il faisait à la fois accéléromètre et gyroscope. Ainsi nous pouvons récupérer 7 valeurs grâce à lui :

- L'accélération en x, y et z
- La température
- La position angulaire en x, y et z



Nous nous intéressons donc uniquement à l'accélération et la position angulaire en x et y puisque nous travaillons en 2D. Cependant, il faut arriver à récupérer ces données.

Nous avons appris que le capteur fonctionnait en I2C. C'est un bus d'information (transfert de données) permettant la communication entre le microprocesseur et d'autres composants. Ici, nous parlons du microprocesseur de la carte Arduino. La communication se fait sur le modèle maître/esclave.

Sur ce modèle, il y un maître qui va donner les commandes à suivre (celui qui communique) tandis que l'/les esclave(s) doivent se charger de les effectuer (celui qui reçoit la communication).

Le fil connecté au SDA (voir photo capteur) est celui qui contient la communication. Il va y avoir deux types de SDA : esclave et maître.

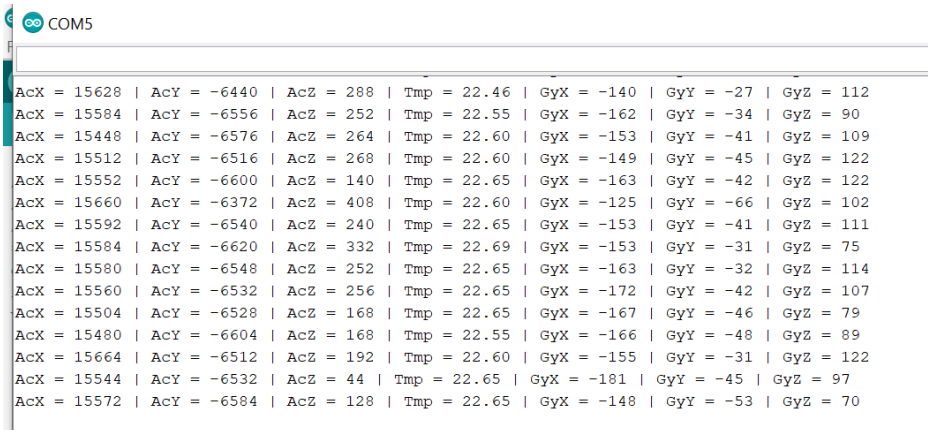
Pour débuter, le maître va donner l'adresse de l'esclave (7 bits) auquel il veut instaurer une communication et un 8ème bit va être mis pour préciser s'il s'agit d'une lecture ou écriture de données. Ensuite, l'esclave va lui confirmer qu'il a bien compris sa tâche. Le maître envoie les données et enfin, l'esclave confirme qu'il les a bien reçues.

Maintenant que nous avons compris ce système de communication, il faut le retranscrire dans un programme Arduino. Nous avons eu besoin de la bibliothèque Wire. On commence par déclarer que la communication se fait avec le capteur à l'aide de la commande **Wire.beginTransmittion(MPUadress)**.

Ensuite, le maitre (la carte Arduino) va demander au capteur (l'esclave) les données : c'est la commande **Wire.requestFrom(MPUadress,14).** Le 14 fait référence au nombre de registres que l'on veut : chaque valeur donnée par le capteur est partagée dans 2 registres (le capteur nous donne 7 informations au total donc cela fait 2x7 registres pour l'intégralité des données). Pour préciser à quel registre nous commençons, il faut la commande **Wire.write(registre)**. La commande **Wire.read() <<8|Wire.read()** va ensuite nous servir pour définir chaque type d'information suivant la datasheet (il s'agit de la concaténation des valeurs retournées par deux registres consécutifs).

Par exemple, si nous commençons le registre par 0x3b alors la commande précédente permet d'avoir l'accélération en x. Enfin, il suffit d'enchainer la même commande 7 fois pour définir chaque valeur retournée par le capteur puis de demander d'afficher les valeurs à

l'aide de **Serial.print** sur le moniteur série. Une fois exécuté, voici ce que nous avons obtenu :



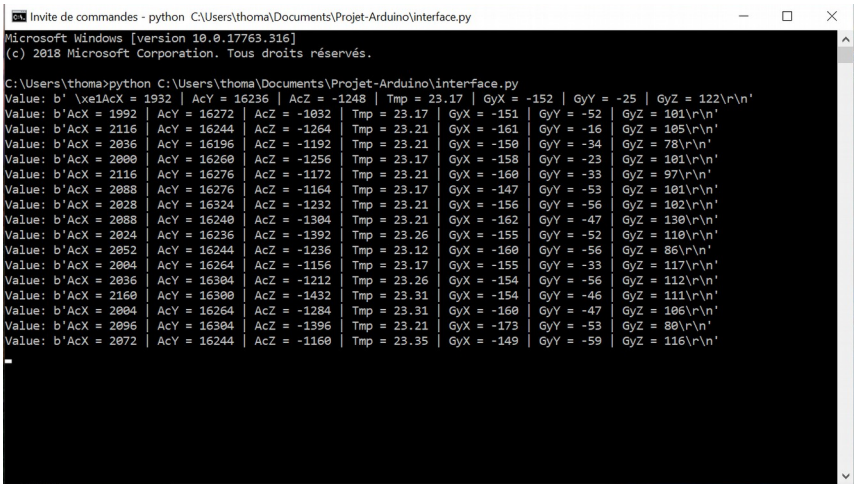
Affichage des valeurs du capteur d'après notre programme de récupération de données

Maintenant que nous avons les valeurs que retourne le capteur, il nous reste à nous attaquer à l'étape du réseau de neurones. Cependant, avant de nous lancer dans celui-ci, il nous a fallu trouver un moyen de récupérer les valeurs obtenues pour qu'elles puissent être utilisées au sein d'un programme Python. Ainsi, nous avons dû programmer une interface jouant le lien entre Arduino et le réseau de neurones.

iii. L'interface Arduino-Python

C'est grâce au câble USB que nous avons réussi à communiquer avec l'Arduino. Ici, la bibliothèque Python adéquate pour lire des valeurs issues d'une carte Arduino est **Pyserial**. Une fois la bibliothèque importée, la commande **serial.Serial('port série', badauds)** va permettre de donner le bon chemin pour accéder aux données. Pour nous, le port série est COM5, qui correspond tout simplement au nom de l'entrée USB de l'ordinateur auquel est branchée la carte. Le nombre de badauds est de 9600 par défaut, c'est la vitesse des données en bit/secondes.

Ensuite la commande **ser.readline()** permet de récupérer les valeurs que retourne Arduino.



Exécution de notre interface python

Cependant, il y a problème qui semble anodin mais qui est essentiel pour ne pas compliquer le réseau : chaque ligne renvoie « bêtement » ce que le moniteur affiche. Ainsi, nous ne pouvons pas utiliser telle qu'elle la ligne retournée par le programme. Nous avons dû la transformer en liste puis la subdiviser en 4 sous listes pour ensuite les transformer en flottant afin d'être sûr que la position peut être intégrée dans les calculs sans poser un problème.

De plus, nous avons tester un autre programme nous permettant de faire vaciller à droite et à gauche notre bras à notre convenance afin d'augmenter le mouvement de balancement et ce, pour réussir à effectuer un tour complet au bras.

Pour cela, nous nous sommes servis du début du précédent programme afin de connecter l'Arduino et l'interface. Nous avons ensuite, à l'aide de la bibliothèque *msvcrt*, réussit à faire en sorte que si la touche « a » du clavier était activée, alors cela était traduit dans le programme par la valeur 0, et si c'était la touche « z », alors par la valeur 1. Nous envoyons ensuite la valeur à l'Arduino à l'aide de la commande **ser.write().** Si la valeur reçue par l'Arduino est 0, alors nous avons choisi arbitrairement que cela correspondrait à un mouvement vers la gauche, et de même, avec la valeur 1, le bras effectue un mouvement vers la droite.

Nous avons pu grâce à cela effectuer une simulation expérimentale du double pendule qui, cependant, n'était pas forcément un gain de temps puisque les touches doivent être activées de façon très précise et de plus en plus espacées de façon à augmenter le balancement, tout en faisant en sorte d'accompagner le mouvement (sinon l'énergie est perdue et le bras perd de son balancement). Nous n'avons réussi qu'à dépasser l'axe horizontale à l'aide du balancement mais pas d'avantage.

Maintenant que nous pouvons récupérer les valeurs, nous pouvons nous occuper du réseau de neurones qui va utiliser ces données. Cependant, il nous reste une question avant de le commencer :

Quel apprentissage allons-nous utilisé pour créer le réseau ? Sera-t-il supervisé ou non ?

iv. L'apprentissage de notre réseau

Nous avons opté pour un système de récompense pour l'algorithme avec l'apprentissage par renforcement. Le principe de cet apprentissage est que l'algorithme qui va jouer le rôle d'agent, choisira l'action la plus appropriée à effectuer en fonction de l'environnement dans lequel il se situe. À chaque type d'action, nous allons donner une récompense en fonction de la pertinence de l'action par rapport au but voulu. L'algorithme doit simuler le plus de choix possibles pour avoir la récompense la plus grande.

Par exemple, prenons un algorithme devant simuler le chemin le plus rapide pour aller d'un point à l'autre. Ainsi, nous allons définir un système de récompenses pour les actions correspondantes à la droite passant par les 2 points. Nous pouvons également définir une récompense moins grande pour des fonctions en escaliers de cette droite. Cet exemple n'est pas très pertinent par rapport au Deep learning puisqu'on aurait très bien pu faire sans.

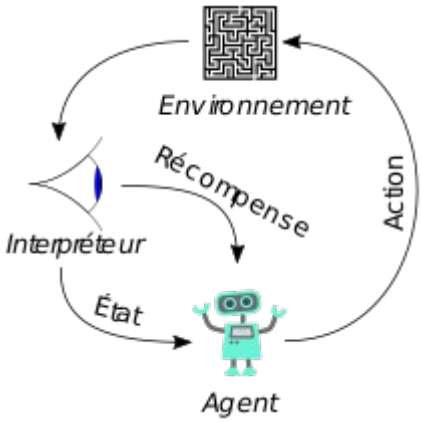


Schéma du fonctionnement de l'apprentissage par renforcement

On débute dans un environnement précis qui va être donné à un interpréteur, le transformant en une récompense et un état précis. L'agent doit, en fonction de ces deux données, choisir la meilleure action, ce qui va changer l'environnement et donc la récompense ainsi que l'état donné par l'interpréteur. Le processus va se reproduire tant que la tâche n'est pas effectuée. On voit aisément sur le schéma que chaque acteur dans ce processus va influencer les autres.

Il faut bien comprendre que le calcul de récompense se fait suivant le choix des actions futures : il va chercher à avoir la meilleure récompense **future**. C'est la formule suivante qui va nous permettre de le modéliser dans notre programme :

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r t_{+2} + \ldots + \gamma^{n-1} r_n$$

$y_j(x)$ correspond à la récompense future à l'instant t , r_t est la récompense actuelle à l'instant t.

γ est une valeur comprise entre 0 et 1 jouant un rôle dans l'apprentissage. Plus gamma s'approche de 0, plus on va vouloir une récompense dite **proche** (c'est-à-dire la meilleure action directe) et plus il s'éloigne, plus on va avoir cette fameuse récompense future.

Cela ne nous donne cependant pas le rapport avec le réseau de neurones.

C’est avec une autre fonction, la Q-value, qui détermine la meilleure récompense après une action a, dans l’état s, que l’on va pouvoir observer le lien :

$$Q(s_t, a_t) = max\ (R_{t+1})$$

Ce qui est intéressant, pour la Q-value c’est que, pour l’action a et l’état s, elle se calcule avec la Q-value à l’instant t+1.

Nous allons alors pouvoir construire un réseau de neurones où la couche d’entrée sera la Q-value à l’instant t et la Q-value à l’instant t+1, la couche de sortie. Ensuite, avec la dernière équation, nous pouvons calculer l’erreur entre la Q-value à l’instant t et le résultat de l’équation avec la Q-value à l’instant t+1 qui doivent être égaux. Ainsi, nous pouvons procéder à l’entrainement du réseau de neurones, afin d’avoir les meilleurs poids et donc un réseau de neurone correspondant à la meilleure combinaison de choix d’actions.

Nous pouvons cependant avoir un problème lorsque l’algorithme se contente d’une récompense maximale locale et non globale. Pour cela, nous devons introduire de l’aléatoire dans le choix des actions pour pousser l’agent à exploiter les moindres actions possibles dans l’environnement.

C’est avec la stratégie d’ε-greedy que nous allons réussir à le faire. Le principe est simple : on choisit arbitrairement un nombre entre 0 et 1 que l’on appellera ε. Le choix ε va dépendre de la part d’aléatoire que nous donnerons dans l’algorithme. Ensuite, lorsque l’agent va devoir choisir une action, on va tirer aléatoirement un nombre entre 0 et 1 : si celui-ci est inférieur à ε, alors l’action choisie sera aléatoire, sinon, c’est l’agent qui va choisir l’action la plus pertinente.

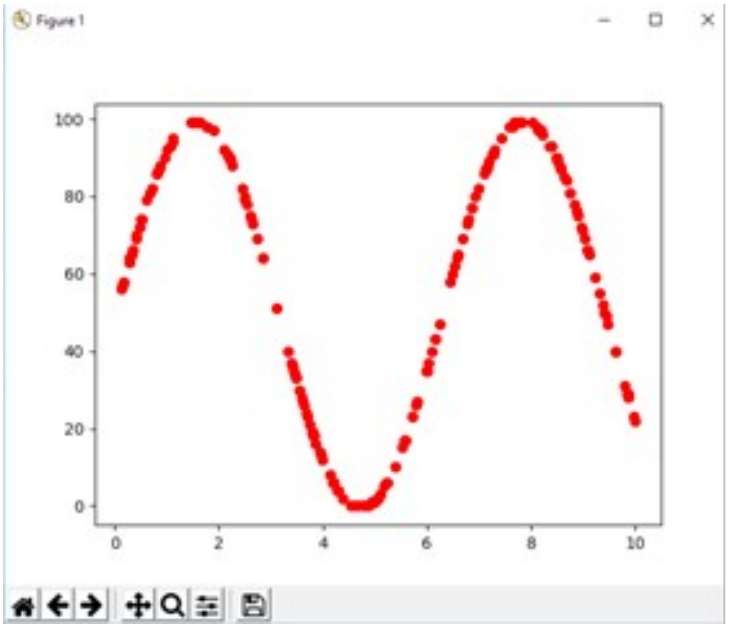
Cela permet d’étendre les possibilités et d’empêcher l’algorithme de se contenter d’une récompense qui peut ne pas être la récompense totale de l’environnement.

Nous pouvons maintenant commencer le réseau de neurones.

v. Le réseau de neurones

Sous les conseils de notre chef de projet, nous avons dans un premier temps tenter de *fiter* la fonction sinus, c'est-à-dire se rapprocher au maximum de la fonction sinus en utilisant un réseau de neurones.
Pour ce faire, nous allons utiliser le module *Pytorch* de python. C'est un module qui permet de créer des réseaux de neurones plus facilement et de les optimiser plus facilement en utilisant une carte graphique si possible.

Pytorch utilise majoritairement des tenseurs, ce sont des matrices spécifiques à pytorch qui permettent le calcul du gradient descendant ainsi que la back propagation.
On crée un réseau composé de 2 couches cachées. Ce nombre est choisi arbitrairement, de sorte que le réseau de neurones soit tout de même profond sans que le calcul ne soit trop compliqué. C'est aussi le nombre de couches cachées que nous pensons utiliser pour le double pendule.
Une fois ceci fait, on définit comme 'cible' de notre réseau un tenseur composé de valeurs pseudo-aléatoires de la fonction sinus pour x compris entre 0 et 10. On obtient le graphique suivant :

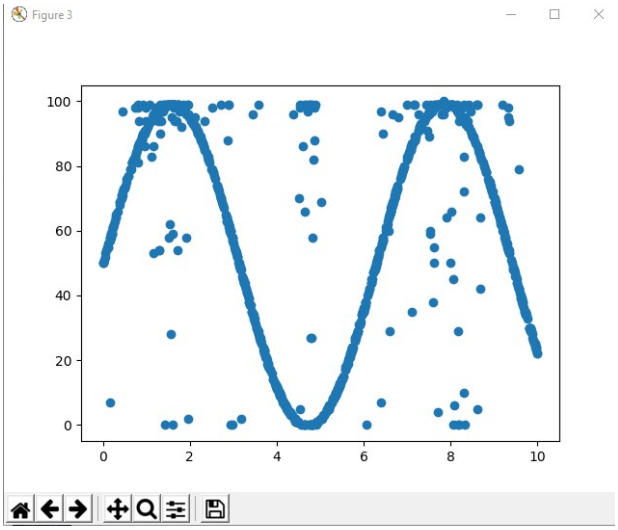


Représentation graphique de la fonction sinus

Nous remarquons que l'abscisse est comprise entre 0 et 100, on ajoute tout simplement 1 à toutes les valeurs du tenseur. Puis, nous multiplions ces valeurs par 50 pour obtenir ce graphique.

Cette manipulation est nécessaire car la fonction "criterion", qui correspond au calcul de l'erreur a besoin d'avoir en entrée des valeurs supérieures à zéro et un tenseur "long". Or, lorsque l'on transforme un tenseur normal en "long", les valeurs sont arrondies donc on a besoin d'augmenter artificiellement nos valeurs pour conserver les écarts.

Au milieu de l'entrainement, on obtient ceci :

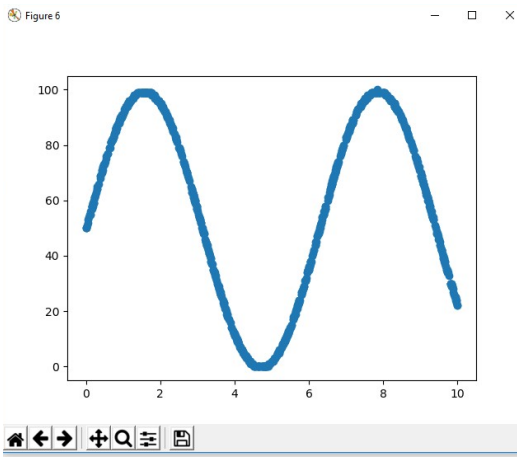


Représentation de l'approximation de la fonction sinus dans notre réseau de neurones

Nous pouvons voir que certains points ont déjà "pris place" sur la courbe mais qu'il en manque encore beaucoup.

Après entrainement, on a une courbe qui se rapproche beaucoup de notre courbe initiale avec tout de même quelques points qui se sont dispersés. C'est tout simplement dû au fait que nous n'avons pas entrainé l'algorithme assez longtemps et/ou que le taux d'apprentissage était trop grand (donc l'algorithme apprend plus vite mais moins précisément).

En faisant un apprentissage plus long, on obtient finalement :



Représentation graphique à l'issue de 2000 entrainements du réseau

Nous arrivons donc à faire fonctionner un réseau de neurones et à lui faire *fiter* une fonction. De plus, nous arrivons à faire bouger le bras articulé et nous savons transférer des données de python à la carte Arduino et vis-versa. Il ne nous reste donc qu'à adapter le réseau pour faire bouger le bras.

IV. Les limites de notre projet

i. Nos problèmes

Ayant le corps de notre réseau de neurones, nous avons été confrontés à deux problèmes :

Premier problème : La communication Arduino-Python

Nous avons rencontré une erreur lors de la conversion octet-entier. Effectivement, la commande ser.write() sur Python, nous fait renvoyer des octets de la forme : b'\x01\x02\ x00' (cela correspond au nombre 120). Or, sur Arduino, nous ne parvenons pas à le convertir en entier pour indiquer l'angle du moteur.

Second problème : Le réseau de neurones en lui-même

Nous avons remarqué que le réseau de neurones demandait beaucoup d'itérations et de temps pour des résultats assez peu précis. De plus, les données du capteur sont elles aussi assez imprécises, nous ne sommes donc même pas sûrs que le réseau aurait pu nous permettre de remonter le bras. Cependant, notre réseau arrivait à effectuer une phase d'entrainement avant d'avoir l'erreur.

ii. Notre façon de concevoir

Théoriquement, nous avions pensé à créer un réseau de neurones fonctionnant sur le modèle suivant :

Notre couche d'entrée serait la moyenne des valeurs du capteur à l'état de repos et l'objectif à atteindre, la moyenne des valeurs du capteur à la position verticale voulue.

À chaque entrainement, nous enverrions les valeurs de la couche de sortie obtenues à l'Arduino pour qu'il fasse un mouvement d'angle correspond à cette valeur. Une fois le mouvement fait, Arduino renvoie les nouvelles valeurs du capteur au programme. L'algorithme python se sert de cette nouvelle position du capteur pour entrainer son réseau en utilisant la fonction de coût (F.nll_loss) et le gradient descendant (loss.backward). On répète cela à l'infini afin d'atteindre notre objectif.

iii. Notre recul sur le projet

Nous estimons qu'il nous aurait fallu une quinzaine d'heures de travail afin de finir le réseau de neurones tel que nous l'avions pensé. Cependant, si celui-ci ne fonctionnait pas tel que nous le voulons, il nous aurait fallu beaucoup plus de temps.

De plus, nous émettons des doutes concernant la possibilité de le faire monter à la verticale étant donné que nous avons deux algorithmes pour le faire bouger manuellement (un sur Arduino et l'autre avec les touches A et Z présenté précédemment) et que nous arrivons à peine à dépasser l'horizontale. Peut-être que l'intensité donnée au courant n'est pas assez importante ? Le moteur n'est pas assez puissant pour réaliser cette tâche ? Ou alors nous n'avons pas employé la meilleure méthode pour la faire monter ?

Nous avons mal géré notre temps puisque nous avons beaucoup insisté sur la partie théorique et une énorme partie de ces connaissances ne nous a pas servie (entre autres ; l'algorithme par renforcement qui s'avère être très complexe à mettre en place ; l'apprentissage du langage C qui a été très peu utile)

Également, nous avons souffert d'un trop gros manque de connaissances pré-requises et nous pensons que des TD nous aurait grandement aidés mais notre formation de PEIP ne nous en donne pas accès.

V. Conclusion

Malgré notre échec à finaliser entièrement notre projet, nous avons remis en question et améliorer beaucoup de nos capacités individuelles, qui nous seront très utiles dans notre futur cycle d'ingénieur, telles que l'organisation, la répartition des charges de travail dans un groupe, ou encore la persévérance devant la difficulté.

Ce projet nous a permis d'approfondir nos connaissances en matière d'informatique et d'électronique. Nous avons appris et compris le fonctionnement d'un réseau de neurones et

plus généralement, nous avons appris à appréhender un domaine plus poussé et complexe qu'est le Deep Learning.

Ce que nous avons retenu est que la théorie peut être comprise sans pour autant que la mise en pratique ne soit évidente. Le fait que ce projet concentre à la fois de l'électronique, de l'informatique, de la physique et des mathématiques est un atout indéniable pour une polyvalence des connaissances.

Si nous avions terminé le projet à temps, le fait de pouvoir simuler l'équilibre à l'aide de l'intelligence artificielle fonderait les prémisses d'un projet théorique sur la fabrication d'un robot bipède. À notre échelle, notre projet pourrait sembler peu pertinent en tenant compte de la technologie actuelle, cependant il amène à des réflexions plus concrètes et utiles pour l'avenir :

Peut-on se servir de l'intelligence artificielle pour rendre la mobilité à des personnes physiquement diminuées ?

L'intelligence artificielle est dans la continuité du développement de la technologie et va très probablement prendre une place encore plus importante dans notre quotidien.

VI. Bibliographie

Réseau de neurones artificiels :
https://fr.wikipedia.org/wiki/R%C3%A9seau_de_neurones_artificiels

Test de Turing :
https://www.csee.umbc.edu/courses/471/papers/turing.pdf

L'apprentissage supervisé :
https://www.math.univ-toulouse.fr/~agarivie/mydocs/apprentissageSupervise.pdf

L'apprentissage non supervisé :
https://fr.wikipedia.org/wiki/Apprentissage_non_supervisé

HEDIN, Jean-Claude : *Comprendre le Deep Learning*, 2016

Conférence de Yann Lecun (USI Event 2015) :
https://www.youtube.com/watch?v=RgUcQceqC_Y

Build A Neural Net in 4 minutes (Siraj Raval) :
https://www.youtube.com/watch?v=h3l4qz76JhQ

Programmez vos premiers montages Arduino (cours OpenClassroom)

Communication Arduino-Python :
https://www.pobot.org/Arduino-pilotee-en-Python.html

Récupérer les valeurs du capteur :
https://knowledge.parcours-performance.com/ajouter-gyroscope-a-robot-arduino/

How I2C Communication Works and How To Use It with Arduino (How to mechatronics):
https://www.youtube.com/watch?v=6IAkYpmA1DQ

Conférence sur l'algorithme par renforcement (Xebia : mois du data mai 2017)
https://www.youtube.com/watch?v=6nEJSmC_6cE

Documentation Pytorch pour le réseau de neurone :
https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html#sphx-glr-beginner-blitz-neural-networks-tutorial-py