Cinématique inverse d'un robot redondant à 7DDL par l'intelligence artificielle

Rayan BOUISSOU
Étudiant en maîtrise de la production automatisée
École de Technologie Supérieure (ÉTS)
Montréal, QC, Canada
rayan.bouissou.1@ens.etsmtl.ca

Louis BOIROT
Étudiant en maîtrise de la production automatisée
École de Technologie Supérieure (ÉTS)
Montréal, QC, Canada
louis.boirot.1@ens.etsmtl.ca

***Résumé*—Cet article couvre la mise en place d'une résolution de cinématique inverse par deep learning. Sur un robot 7 degrés de libertés, le traitement d'un problème de cinématique inverse est trop long pour être mis en place dans une production automatisée. Pour utiliser un réseau, des milliers de données valides pour entrainer et tester le robot ont été créés. Plusieurs architectures neuronales ont été testés et comparés dans ce problème jusqu'à en trouver une adéquate. Une relation avec d'autres articles scientifiques sera également effectuée pour valider nos résultats.**

Keywords— 7 DDL–DOF, deep learning, inverse kinematics.

I. INTRODUCTION

Lorsqu'une entreprise souhaite automatiser sa production, elle se penche souvent vers la solution la plus évidente : le robot articulé. Ce choix est justifié par 2 raisons : il coute moins cher qu'une ligne de production entière et il est capable d'être reprogrammé pour effectuer une tout autre fonction que celle à laquelle il avait été assignée au départ.

Ensuite vient un choix plus délicat : combien de degrés de libertés (DDL) le robot devrait posséder ? La réponse parait simple sur le papier car plus il a de degrés de liberté et plus il peut faire des mouvements complexes mais plus il est susceptible de commettre des erreurs. En effet, si on rajoute des degrés de libertés à un robot, l'algorithme devra s'assurer du placement de chacun d'entre eux donc il a statistiquement plus de chance de se tromper.

La frein qui retenait les entreprises à investir dans ces « super » robots à 7DDL était surtout la longue durée des temps de calcul dûent à des nombreux axes à étudier à chaque itérations. Néanmoins, : les réseaux de neurones sont venus régler ce problème. Capable de résoudre des problèmes tout en apprenant (learning), les algorithmes de deep learning s'entrainent avec des millions de données déjà validé par un autre algorithme puis créent des relations entre les différents paramètres d'entrée, c'est le principe d'entée (Input) – sortie (Output). Les neural network (NN – réseaux de neurones) créé comme le cerveau humain un ensemble de liens entre des couches de neurones (Layers), chaque layers est composé de plusieurs neurones et chaque neurone possède un liens avec les neurones du layers précédent, ce liens possède un poids de probabilité généralement appeler W. Le principe d'apprentissage dépend de la fonction d'activation qui est en quelque sorte le garde-fou de la valeur que transmet notre neurone, si la valeur qui arrive aux neurones coïncide ou satisfait les conditions de la fonction d'activation alors la valeur est transmise au neurones suivant, si la valeur ne satisfait pas la fonction d'activation alors la valeur n'est pas transmisse et le poids de probabilité est modifié pour s'assurer que les valeurs obtenue en sortie de notre réseaux soit ceux désiré. Il existe plusieurs fonctions d'activations ainsi que plusieurs types de layers, chacun avec une propriété spécifique et des calculs bien précis que nous détaillerons dans la suite de l'article.

Enfin après avoir créé des milliers de relations, le réseau de neurones vas apprendre et éventuellement retourner des résultats attendu, dans cette articles nous allons donc essayer d'utiliser différentes architectures et étudier celles qui nous ont permis d'obtenir le meilleure résultats. Nous allons voir dans ce papier comment entrainer un réseau intelligent pour qu'il effectue une cinématique inverse d'un robot de 7 degrés de liberté le plus précisément possible.

II. INSPIRATION DES SOURCES

L'optimisation des mouvements robotiques étant un sujet fréquent dans les entreprises, des chercheurs du monde entier se sont penchés sur le sujet. La solution qui revient serait d'utiliser un réseau de neurones, comme l'ont fait M Aruna Devi et son équipe. Dans leur article [1] ils vont essayer de planifier une trajectoire optimale pour le robot en limitant les à-coups. Pour cela ils se servent d'outils comme les réseaux de neurones artificiels (ANN), les heuristiques de plus proches voisins (KNN) et la régression linéaire (LR). Ils ont utilisé un algorithme développé sur une combinaison linéaire de splines positives à support compact minimal (B splines) adapté en cubique pour le robot. Cette opération mathématiques permet de limiter les à coup et de faire des trajectoires plus lisses. Pour diminuer le temps de réflexion, les algorithmes de machine learning

sont utilisés. La difficulté est de trouver le meilleur algorithme pour éviter un obstacle dans le mouvement. Il semblerait que l'algorithme KNN est le meilleur avec 100% de précision et une meilleure efficacité dans l'évitement de l'obstacle.

Une équipe de la faculté d'ingénierie d'électronique et d'information à Shaanxi a optimisé une méthode de Levenberg–Marquardt qui consiste à obtenir une solution à la minimisation d'une fonction se présentant souvent sous la forme d'une régression non linéaire et qu'on appellera DLS (Damped Least Squared). Leur algorithme va calculer une trajectoire entre la position actuelle du robot et une destination pour que le robot puisse aller le plus vite possible avec le moins de mouvement des articulations. Ils ont remarqué que parmi les nombreuses données d'entrées (angles de chaque degré de liberté, le seuil d'erreur de convergence, l'erreur de direction, etc) certaines avaient une plus grande influence sur le résultat.

Le deuxième angle de liberté à la base du robot ainsi que le dernier sont les plus importants car ce sont ceux qui vont bouger à chaque mouvements :

- Le 2e angle va modifier la hauteur et la profondeur de tout le robot en le levant ou le baissant
- Le dernier va s'assurer que l'outil soit parfaitement placé pour l'opération suivante

Certains facteurs d'erreurs jouent aussi un rôle important dans l'optimalité du résultat mais la relation entre eux est très floue. C'est pourquoi l'apprentissage profond est très utile dans ce genre de cas. Après les tests sur les différentes données, le réseau de neurones est capable de répondre en 5 millisecondes et une dizaine d'itérations alors que la méthode de convergence la plus fiable le Gaussian DLS (GDLS) prend environ 75 millisecondes et 151 itérations.

On remarque que la méthode d'optimisation par réseau de neurone écrase toutes les techniques mathématiques connu car elle créé des relations incompréhensibles entre les variables clés mais qui fonctionnent très bien.

(Mettre les graphiques de l'article)

Si on se penche plus en détail dans le problème d'un 7DDL on remarque que très peu de robots existent en 7DDL, certains chercheurs vont même jusqu'à créer leur propre robot pour effectuer leurs expérimentations. L'équipe de H. Ashraf a créé son propre robot pour avoir des données de cinématique inverse [2]. Ainsi ils peuvent « nourrir » leur réseau de neurones avec des données dont ils sont certaines qu'elles vont fonctionner. Ils comparent deux architectures de réseaux de neurones pour trouver l'optimal.

Pour les créer ils ont sorti la matrice DH de leur robot puis ils ont analysé les résultats avec Matlab avant de l'incorporer dans leur architecture profonde avec la librairie Keras dans Python.

Nous nous sommes inspirés de leur démarche pour effectuer notre propre réseau de neurones.

III. Manipulation effectues / Chemin de pensee

Grâce à la librairie Keras, nous avons créé notre propre réseau de comme dans le dernier article.

L'objectif de trouver un mouvement optimal pour une trajectoire était extrêmement difficile. Comme nous n'étions que 2 et que nous n'avions que très peu de connaissance dans Matlab et les réseaux de neurones, nous avons réfléchis comment se rapprocher au maximum de notre objectif.

Les articles ou ils arrivent à optimiser un mouvement demande des connaissances en mathématiques informatisées gigantesques. En revanche, le réseau de neurone de H. Ashraf va avoir des données d'entrées qui sont les limites d'angles de chaque articulation et la position à atteindre. La sortie sera la position angulaire de chaque articulation. L'objectif est qu'il soit le plus proche possible de la position désiré. Nous étions tout à fait capables de reproduire cela.

Pour réaliser notre algorithme il nous fallait la matrice DH d'un robot avec 7 degrés de liberté. Nous avons décidé de prendre le Gen3 de Kinova qui possède une version 6 ou 7 DDL. Ce choix est principalement du au 183 pages explicatives du robot disponible gratuitement sur lesquels ont a pu se baser pour effectuer nos calculs de vérifications.
La librairie Keras de Tensorflow est extrêmement puissante et nous allons nous en servir pour notre réseau de neurones.

IV. Cinematique Directe et inverse

Nous nous sommes directement inspirés d'un exemple disponible sur Matlab pour l'ensemble de l'obtention des données

Afin d'étudier la cinématique inverse nous avions besoin d'un modèle de robot à 7 DDL. Dans un premier temps nous avons essayé d'utiliser la cinématique directe pour générer l'espace de travail (workspace environment) du robot, or cette manipulation prend du temps à simuler et nous n'arrivions pas à obtenir l'ensemble de l'espace de travail.

$$ {}^{i-1}T_i = \begin{bmatrix} cos(\theta_i) & -cos(\alpha_i)sin(\theta_i) & sin(\alpha_i)sin(\theta_i) & a_i cos(\theta_i) \\ sin(\theta_i) & cos(\alpha_i)cos(\theta_i) & -sin(\alpha_i)cos(\theta_i) & a_i sin(\theta_i) \\ 0 & sin(\alpha_i) & cos(\alpha_i) & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

Figure 1 : Paramètres DH pour une transformation classique ([6], p163)

i	α_i (radians)	a_i (m)	d_i (m)	θ_i (radians)
0 (from base)	π	0.0	0.0	0
1	$\pi / 2$	0.0	- (0.1564 + 0.1284)	q_1
2	$\pi / 2$	0.0	- (0.0054 + 0.0064)	$q_2 + \pi$
3	$\pi / 2$	0.0	- (0.2104 + 0.2104)	$q_3 + \pi$
4	$\pi / 2$	0.0	- (0.0064 + 0.0064)	$q_4 + \pi$
5	$\pi / 2$	0.0	- (0.2084 + 0.1059)	$q_5 + \pi$
6	$\pi / 2$	0.0	0.0	$q_6 + \pi$
7 (to interface)	π	0.0	- (0.1059 + 0.0615)	$q_7 + \pi$

Figure 2 : Paramètres DH classiques sphériques ([6], p163)

Nous avons donc pris le cas opposé et décider d'utiliser la cinématique inverse pour obtenir nos données et d'utiliser la position de l'effecteur dans l'espace cartésien donc x, y et z comme entrée de notre réseau de neurones. Nous avons étudié plusieurs robots industriels à 7 DLL et nous avons choisie de prendre le Kino Gen3 pour notre projet. La fonction Matlab '*inversekinematic'* (IK) permet d'étudier la cinématique inverse et notamment d'obtenir les matrices de nos angles (theta) qui sont associés au 7 axes de notre robot pour un totale de 7 theta. Ce qui est pratique avec la fonction IK c'est que le Kinova Gen3 est lui aussi présent dans la base de données de la fonction 'loadrobot', cette fonction permet d'importer un modèle rigide de robot avec des joints, il est possible de réaliser son propre robot ou bien d'importé un robot qui existe pour l'étudier.

Le fait de passer par cette étape nous permettais de nous assurer que notre robot soit mécaniquement réaliste, que les limites soit respecté et aussi de nous assurer que son environnement de travail reste le même pour chaque trajectoire (cf la figure suivante).

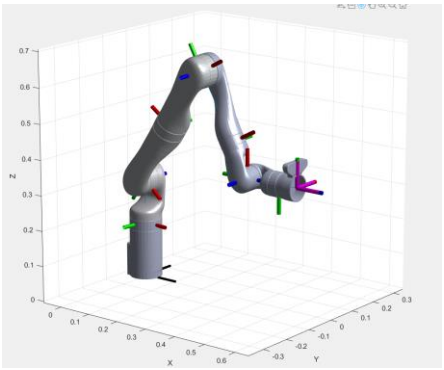


Figure 3 – Représentation du Kinova Gen3 sous matlab

Pour réaliser la trajectoire de notre robot nous avons indiqué un ensemble de point former autour d'un cercle, chaque points étant séparé par un angle nous avions donc plusieurs outils pour faire varier les trajectoires effectué tel que la position du cercle, le rayon du cercle et l'angle entre les différents points. Nous avons utilisé 3 configuration différente au niveau de la position du cercle dans l'espace et aussi 2 rayon différent de cercle pour avoir plus de donnés possible. Avec le tableau en annexes qui représente les différents cercles de trajectoire nous avons pu obtenir 80 000 données pour les angles et les points donc un totale de 160 000 données sous forme de matrices 3x10 000 pour la position de l'effecteur et 7x10 000 pour les angles des joints.

V. TRAITEMENT DES DONNEES

Nous avons exporté les données sur Matlab dans des fichiers *'.csv'* (disponible en compressé), ces fichiers ont ensuite été importé sur l'environnement de travail *Jupyter notebook* et avons définie les données tel que X_Data représente les matrices de points et Y_Data les matrices des angles.

```
X_Data.head(3)
```

	x	y	z
0	0.5	0.1	0.400063
1	0.5	0.1	0.400126
2	0.5	0.1	0.400188

Figure 4 - représentation de X_Data

```
Y_Data.head(3)
```

	theta1	theta2	theta3	theta4	theta5	theta6	theta7
0	-0.252025	0.443977	3.096152	-2.130523	-0.342403	1.035787	1.770949
1	-0.252027	0.443828	3.096157	-2.130459	-0.342446	1.035584	1.771024
2	-0.252030	0.443679	3.096162	-2.130394	-0.342490	1.035380	1.771100

Figure 5 - Représentationde Y_Data

Après l'obtention des données nous n'avions plus qu'à les implémenté dans le réseau de neurones. D'après la littérature il existe plusieurs manières de traités des données et de les classé, ces méthodes sont notamment utilisées pour obtenir des résultats convergent rapidement quand les données de base sont très variées, or nos données sont basées sur un cercle donc même avec 80 000 données les résultats se rapproche. Néanmoins nous avons appliqué la méthode *'Normalized Data'* **(1)** pour mettre nos données à la bonne échelle (scaling).

$$\boldsymbol{Normalized\ Data} = \frac{\boldsymbol{Y - Y_{min}}}{\boldsymbol{Y_{max} - Y_{min}}}$$

(1)

Nous avons décidé d'appliqué ce scaling uniquement aux données en sortie. Par la suite nous avons établie un système d'apprentissage, sur l'ensemble de nos données nous avons utilisé 67% pour entrainer notre modèle et 33% pour vérifier notre modèle.

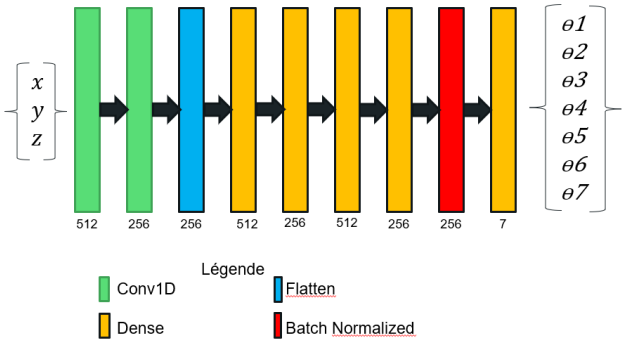
VI. RESEAUX DE NEURONES – IMPLEMENTATION

Nous avons basé l'architecture de notre réseau de neurones sur l'article proposé par Elkholy et al.[2]. Il

existe plusieurs types de réseaux de neurones envisageable et architecture, on peut ici en notés 3 les réseaux de neurones artificiel (ANN), réseaux de neurones convolutionnelle (CNN), ici nous utilisons un réseau CNN pour l'architecture de notre réseau, même s'il est plus généralement utilisé pour de la reconnaissance d'image via notamment l'utilisation de la convolution, il peut s'appliquer sur d'autres domaines telles que celui de l'IK.

Pour implémenter notre modéle nous avons utilisé la librairie Keras pour utiliser le modèle sequentiel ainsi que plusieurs layers issues de Keras et nous avons aussi utilisé l'optimiseur Adam avec une vitesse d'apprentissage de 10^{-3} et l'étude de l'erreur est selon la 'MSE' (Mean Square Error). Le modèle séquentiel sert principalement à faciliter la mise en place d'une série de layers possédant un seul tenseur en input et un seul en output. Cela permet de simplifier le code lors de son écriture et de sa lecture. L'avantage d'utiliser un modèle plutôt que de simples séries de couches c'est qu'on peut le sauvegarder et l'adapter et donc faire plusieurs tests selon plusieurs architectures ou bien avec un même modèle mais des fonctions d'activation différentes.

VII. Reseaux de neurones – Architecture



Nous n'utilisons pas l'orientation de l'effecteur car nous ne l'utilisons pas dans la génération des données. La position de l'éffecteur représente l'entré du réseau et les 7 angles sont la sortie. Chaque couche est constitué de 256 ou de 512 neurones. Pour Conv1D nous utilisons un filtres de 512 avec un kernel de 3 et une fonction d'activation linéaire. Nous utilisons une couche Flatten suivis de 4 couches Dense. Enfin nous utilisons une couche Batch Normalized suivi d'une couche dense constitué de 7 neurones qui seront respectivement nos 7 angles.

```
Model: "sequential_14"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv1d_24 (Conv1D)           (None, 3, 512)            2048
_________________________________________________________________
conv1d_25 (Conv1D)           (None, 1, 256)            393472
_________________________________________________________________
flatten_12 (Flatten)         (None, 256)               0
_________________________________________________________________
dense_58 (Dense)             (None, 512)               131584
_________________________________________________________________
dense_59 (Dense)             (None, 256)               131328
_________________________________________________________________
dense_60 (Dense)             (None, 512)               131584
_________________________________________________________________
dense_61 (Dense)             (None, 256)               131328
_________________________________________________________________
batch_normalization_12 (Batc (None, 256)               1024
_________________________________________________________________
dense_62 (Dense)             (None, 7)                 1799
=================================================================
Total params: 924,167
Trainable params: 923,655
Non-trainable params: 512
```

Figure 6 - Exemple du modèle utilisé

La couche convolutionnelle (Conv1D) joue le rôle le plus « lourd » dans un réseau de neurones. Elle se compose d'une série de filtres auto-apprenant (noyaux). Chaque filtre a une petite taille, 1x3 dans notre cas pour conv1D. Pendant le passage à travers le calque, le filtre est glissé (convoqué) sur la largeur et la hauteur de l'entrée. Au fur et à mesure que le filtre est glissé, une carte d'activation ou de fonction est produite qui donne la réponse du filtre à chaque position spatiale. Le réseau apprendra les filtres qui s'activent lorsqu'ils voient une caractéristique spécifique, comme les bords si l'entrée est une image. Chaque couche convolutionnelle a plusieurs filtres et chaque filtre produit une activation séparée ou une carte de caractéristiques. Ces cartes d'activation sont empilées et passées à la couche suivante. [4]

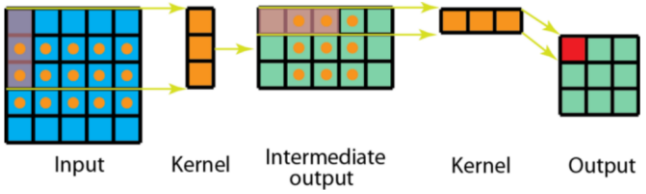


Figure 7 : Convolution avec un filtre de 3x1 (sur une image en 2D) [5]

La couche « dense » est la plus couramment utilisée dans les modèles. C'est une couche de réseau neuronal qui est profondément connectée. Chaque neurone reçoit l'entrée de tous ceux de sa couche précédente. Ceci signifie qu'il y aura n^2 relations à chaque couches pour n input. En arrière-plan, la couche dense effectue une multiplication matricielle vectorielle. Les valeurs utilisées dans la matrice sont en fait des paramètres qui peuvent être formés et mis à jour à l'aide de la rétropropagation. [3]

La couche « Flatten » permet de ramener tous les niveaux du réseau multicouche à un seul plan. Ceci permet de reprendre les modifications avec les fonctions capables de ne manipuler qu'une seule couche.

Comme son nom l'indique, la couche « batch normalization » vient normaliser toutes les données pour éviter qu'une petite valeur soit ignorée parmi une autre très grande. Dans notre cas c'est indispensable car les mouvements des premiers angles sont beaucoup moins grands que ceux des derniers mais ils ont un impact gigantesque sur les mouvements de l'outil. A titre d'exemple, en considérant que le robot soit en position initiale (tous les angles à 0°), un mouvement du 2e angle à la base du robot de 1° déplacerait l'outil de 1,5 cm. Le fait de normaliser toutes les données à une échelle identique permet de donner autant d'importance à chaque angle.

Un optimizer sert à diminuer l'erreur commise par le réseau de neurone lors de sa prédiction. En bref, lorsque le résultat final est comparé aux valeurs de départ, il y a une erreur qui ressort. L'optimizer va venir constater cette erreur et mettre à jour les paramètres des neurones pour améliorer sa prédiction ultérieure.

VIII. RESEAUX DE NEURONES – RAISONNEMENT ET RESULTATS

Chaque layers possède une fonction d'activation, nous avons utiliser plusieurs fonctions avant de trouver celle la plus adapté à notre réseau. Nous avons appliqué un raisonnement empirique pour trouver ces fonctions, nous nous sommes basé sur l'ensemble des fonctions disponible dans la librairies pour étudier nos résultats.

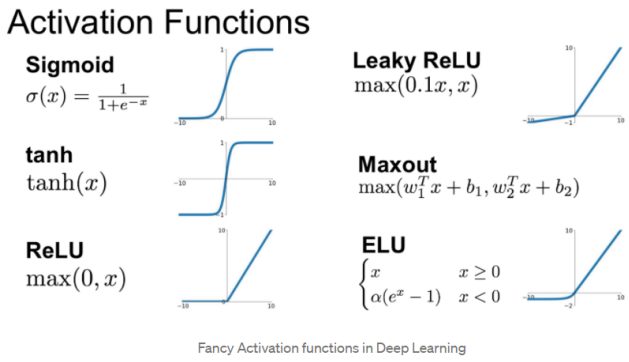


Figure 8 - Différentes fonctions d'activations

Pour savoir si notre modèle fonctionne nous utilisons nous regardons la valeur de la loss en fonction du nombre d'epochs (itération), la fonction 'mean square error' qui nous permet de déterminer la différence de l'erreur au carré, si la MSE est proche de zéros cela veut dire que notre modèle et bien entrainé et qu'il trouve une valeur cohérente par rapport à celle qu'on voulait qu'il trouve en revanche plus la valeur s'éloigner de zéros cela veut dire que la valeur en output ne correspond pas à la valeur attendu. Cet outil nous permet d'étudier le modèle en nous focalisant sur l'erreur à l'apprentissage et à l'essai avec une valeur réelle. Nous avons utilisé 2 méthodes pour vérifier l'apprentissage de notre modèle, la première méthode consiste à réutiliser une valeur qu'il à potentiellement vue pendant son apprentissage et la seconde consiste à ne pas implanter une partie des donné dans la fonction *'train_test_split'* (il ne les as techniquement jamais) pour étudier son comportement vis-à-vis de données « Inconnue ».

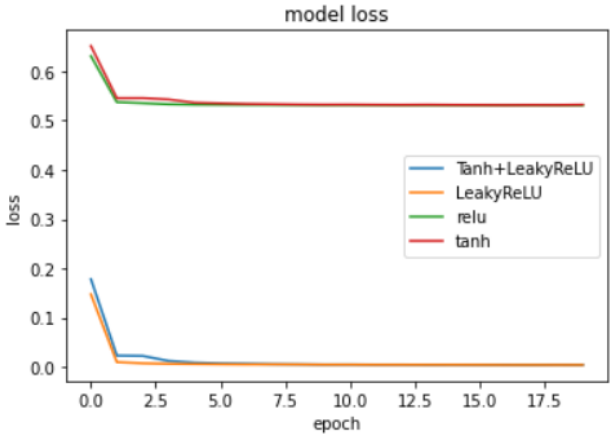


Figure 9 - Représentation des loss par epoch

Sur la figure ci-dessus nous pouvons constater que tous les modèles entrainés tendent vers une valeur plus ou moins pertinente, l'utilisation des fonction 'Tanh' et 'Relu' tendent vers une valeur loss de 0.53. L'utilisation de 'LeakyRelu' nous donne une erreur de 0.0043 et 'Tanh plus LeakyRelu' nous donne le meilleur résultat pour l'apprentissage avec une valeur loss de 0.00422. Pour la version 'Tanh plus Leakyrelu' nous avons utilisé la fonction activation 'Tanh' sur toutes les couches Dense qui suivent la couche Flatten et utilisé la couche 'LeakyRelu' pour la dernière couche Dense du modèle. L'ensemble des valeurs ne sont pas pris en compte par la fonction 'Tanh' car c'est une tengente hyperbolique et donc les grandes valeurs ne sont transmises (cf. résultats obtenus avec Tanh). Avec cette fonction d'activation le modèle peut prédire les valeurs positives et négatives aux alentours de zéros mais impossible pour lui de déduire au-dessus de la valeur '1'. En utilisant 'LeakyRelu' en dernière fonction d'activation nous permettons l'obtention d'autres valeurs que -1 et 1 et donc nous obtenons une erreur bien plus petite et un résultat cohérent. La Figure 11 - Training Tanh and LeakyReLU représente les valeurs prédites par le modèle, cela correspond à une erreur de 0.062 ce qui est un peut plus que durent l'apprentissage mais toujours correcte.

	0	1	2	3	4	5	6
0	0.152139	0.592787	1.000000	-1.000000	0.089003	0.975114	1.000000
1	0.341986	0.359104	1.000000	-0.999999	0.381666	0.629154	0.999999
2	0.977339	0.642160	0.999996	-0.999996	0.534110	0.529027	0.999996
3	-0.360254	0.341899	1.000000	-1.000000	-0.646734	0.550183	1.000000
4	0.211947	0.573000	1.000000	-1.000000	0.165111	0.966850	0.999999

Figure 10 - Trainning Tanh

	0	1	2	3	4	5	6
0	0.171454	0.596483	3.122550	-2.052816	0.112463	1.067716	1.568439
1	0.299186	0.335870	3.111772	-1.863007	0.367795	0.656043	1.329394
2	1.003469	0.559058	2.344673	-1.752550	0.481157	0.535697	1.592981
3	-0.367104	0.348873	3.257638	-1.701052	-0.639161	0.471422	2.189844
4	0.220201	0.569458	3.132968	-2.041996	0.189991	1.029921	1.528129

Figure 11 - Training Tanh and LeakyReLU

La figure suivante représente l'erreur obtenue par chaque modèle lorsqu'on a demandé aux modèles de prédire des valeurs en fonction de données test. Nous pouvons constater la même tendance pour l'ensemble des fonctions d'activations avec notamment la fonction 'Tanh et LeakyReLU' qui obtient une erreur très faible comparé aux autres modèles

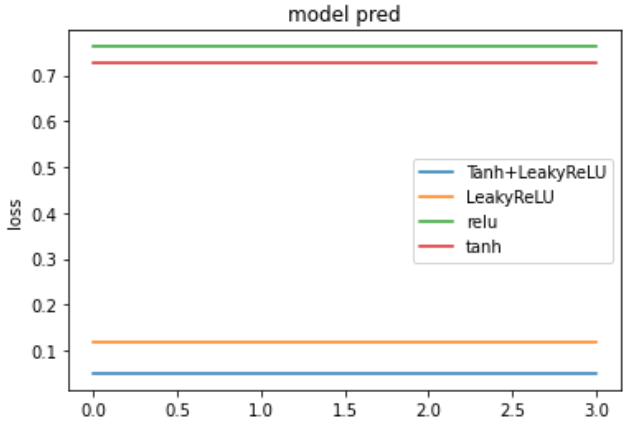


Figure 12 - Résultats de prédiction avec données connues

Pour notre second type de modèle nous avons décider d'appliquer la normalisation des données sur les angles, nous avons donc la fonction précédemment évoquer (1) et avons par la suite entrainé nos différents modèles selon les mêmes principes. Nous avons utilisé la fonction d'activation sigmoïde pour les couches dense afin de comparer les résultats avec la fonction Tanh qui jusqu'à présent nous as donné les meilleurs résultats. La dernière couche 'Dense' est toujours appelée avec 'LeakyReLU' en fonction d'activation pour l'ensemble des tests qui suivent. Nous pouvons constater que pendant l'apprentissage que c'est avec la fonction sigmoïde qu'on obtient le meilleur résultat avec une erreur à 0.00125 et les fonction 'Tanh and Relu' et 'Tanh et LeakyReLU' approche le même résultat avec une erreur à 0.0015, nous constatons aussi que sur cette apprentissage 'LeakyReLU' n'a pas tout de suite convergé vers un résultat probant mais cela peut varier selon chaque session d'apprentissage.

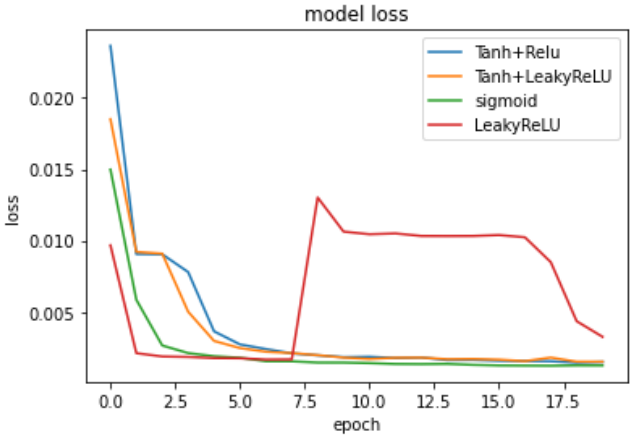


Figure 13 - Erreur par epoch selon chaque fonction d'activation

Pour l'étude de la prédiction du modèle, ici nous n'avons pas utilisé une valeur de test distribué dans la fonction 'train_test_split' mais nous avons gardé une partie des données que nous n'avons jamais montré au modèle afin de voir son comportement fasse à des données qu'il n'a encore jamais vues. L'ensemble des modèles ont su trouver des solutions proches de ce que nous avions lors de l'apprentissage mais le modèle avec la fonction sigmoïde quant à lui a beaucoup plus de difficulté à trouver les bonnes valeurs avec une erreur de 0.8 voir encore plus sur certain essais.

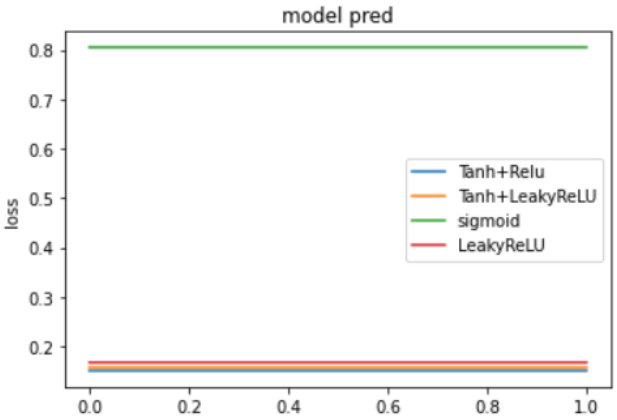


Figure 14 - Erreur fasse à des donnés inconnues

IX. Conclusion

Le modèle étant fonctionnelle et nous donnants des résultats relativement corrects nous pouvons dires que nous sommes satisfaits du résultat, à savoir que ni moi ni mon collège avions des notions en deep learning ou machines learning, les résultats sont convainquant et obtenue avec un temps de calcule en moyennes de 5 secondes pour le modèle avec données non-normaliser et 4 secondes pour le modèle avec les données normalisé, le modèle nous retour des valeurs cohérentes même s'il n'a jamais vu les données. Néanmoins plusieurs points reste à améliorer, tout d'abord nous ne prenons pas l'orientation de l'effecteur comme données

d'entrée de notre modèle ce qui laisse donc une liberté non négligeable aux modèle en terme de solution, en effet même si nos erreurs sont très faibles se paramètre n'existant pas dans la première couche 'Conv1D' il est difficile de connaitre son impacte. Il existe d'autres type de réseaux de neurones nous avons montré ici la version CNN mais le modèle pourrait aussi fonctionner sous une autre structure. Enfin bien que nous ayons un nombre suffisant de donnés, cela ne nous permet de travailler sur l'ensemble de l'espace de travail du robot et donc restreint l'obtention de la cinématique inverse. Pour une amélioration du modèle nous pouvons garder la structure de CNN et ANN mais plus développer le modèle ANN afin d'obtenir de meilleurs résultats, aussi l'obtention de l'ensemble de l'espace de travail permettra une meilleur étude de cinématique et donc de pouvoir l'implémenté sur n'importe quelle robot à 7 degrés de liberté.

Acknowledgment. Les auteurs souhaiteraient remercier le professeur Ph.D. Maarouf SAAD ainsi que PhD. Yassine Kali pour leurs aides et leurs conseils sur ce projet.

X. REFERENCES

[1] A. Devi, P. D. Jadhav, N. Adhikary, P. S. Hebbar, M. Mohsin et K. S. Shashank, «Trajectory Planning & Computation of Inverse Kinematics of SCARA using Machine Learning,» 12 Avril 2021. [En ligne]. Available: https://ieeexplore.ieee.org/document/9395927/metrics. [Accès le 07 2021].

[2] H. A. Elkholy, A. S. Shahin, A. W. Shaarawy, H. Marzouk et M. Elsamanty, «Solving Inverse Kinematics of a 7-DOF Manipulator using convolutional Neural Network,» 24 Mars 2020. [En ligne]. Available: https://link.springer.com/chapter/10.1007%2F978-3-030-44289-7_32. [Accès le 06 2021].

[3] P. Sharma, «Keras Dense Layer Explained for Beginners,» MLK, 20 Octobre 2020. [En ligne]. Available: https://machinelearningknowledge.ai/keras-dense-layer-explained-for-beginners/#What_is_a_Dense_Layer_in_Neural_Network. [Accès le 01 05 2021].

[4] L. M. Kitchell, «Convolutional Neural Networks,» github.io, [En ligne]. Available: https://kitchell.github.io/DeepLearningTutorial/4cnnsinkeras.html. [Accès le 07 2021].

[5] K. Bai, «A Comprehensive Introduction to Different Types of Convolutions in Deep Learning,» Towards Data Science, 11 Février 2019. [En ligne]. Available: https://towardsdatascience.com/a-comprehensive-introduction-to-different-types-of-convolutions-in-deep-learning-669281e58215. [Accès le 08 2021].

[6] Kinova, «Kinova® Gen3 Ultra lightweight robot - User Guide,» 2020. [En ligne]. Available: kinovarobotics.com. [Accès le 21 07 2021].

XI. LISTES DES FIGURES