L’utilisation de machine learning dans mécanique quantique avec réseaux de neurones - rapport de stage

Étudiant : Kanlai Peng, Encadrants : Jérôme Margueron et Hubert Hansen

Dans ce rapport, nous étudions dans un premier temps la capacité des réseaux de neurones artificiels à approximer des fonctions continues à support compact. Nous décrivons ensuite un programme original de minimisation de l’énergie d’une particule dans un puits de potentiel reposant sur cette capacité d’approximation qu’ont les réseaux et essayons de trouver l’état fondamental de ce système. Tous les programmes que nous écrivons sont basés sur Python.

Table des matières

1. Introduction

Le calcul des états et des énergies propres pour un potentiel quelconque est un problème général en mécanique quantique qui s’étend de la physique atomique à l’étude des quarks. On trouve ces états et leurs énergies par la résolution de l’équation de Schrödinger indépendante du temps mais l’absence de solution analytique pour la plupart des potentiels nous contraint à recourir à des solveurs numériques.

Mais pourquoi on utilise le machine learning? En fait, il existe de nombreuses autres façons de résoudre ce problème. Mais si on considère un système très compliqué, malheureusement ces méthodes peuvent devenir inefficaces. Pour trouver une méthode la plus efficiente, on essaie d'utiliser le machine learning avec réseaux de neurones. L'idée d'utiliser les réseaux de neurones comme solveurs de l'équation de Schrödinger indépendante du temps nous est venu de l'article *Machine learning the deuteron* écrit par JWT. Keeble et A. Rios. **[1]**

Afin de trouver les états et les énergies propres d'un potentiel V(x), on doit résoudre l'équation de Schrödinger indépendante du temps (à une dimension ici) où m est la masse d'une particule :

$$\left[\frac{-\hbar^2}{2m}\frac{d^2}{dx^2} + V(x)\right]\psi(x) = E\psi(x) \tag{1}$$

Si ψ(x) est un état propre, son énergie peut être obtenue en multipliant (1) à gauche par ψ(x)

$$\psi(\mathrm{x})\left[\frac{-\hbar^2}{2m}\frac{d^2}{dx^2} + V(x)\right]\psi(x) = E\psi(x)^2 \tag{2}$$

Et on trouve bien :

$$\frac{-\hbar^2}{2m}\,\psi(\mathrm{x})\psi''(\mathrm{x}) + \mathrm{V}(\mathrm{x})|\psi|^2 = E|\psi|^2 \tag{3}$$

Intégrale par partie :

$$\int_a^b \psi(\mathrm{x})\psi''^{(\mathrm{x})}dx = [\psi\psi']_a^b - \int_a^b |\psi'|^2 dx \tag{4}$$

Insérer (4) dans (3) on obtient :

$$\frac{-\hbar^2}{2m}\left([\psi\psi']_a^b - \int_a^b |\psi'|^2 dx\right) + \int_a^b \mathrm{V}(\mathrm{x})|\psi|^2 dx = E\int_a^b |\psi|^2 dx \tag{5}$$

Ainsi :

$$E = \frac{\frac{-\hbar^2}{2m}\left([\psi\psi']_a^b - \int_a^b |\psi'|^2 dx\right) + \int_a^b \mathrm{V}(\mathrm{x})|\psi|^2 dx}{\int_a^b |\psi|^2 dx} \tag{6}$$

Pour la solution analytique, on intègre de $-\infty$ à $+\infty$ et donc on obtient :

$$E = \frac{\frac{-\hbar^2}{2m}\left([\psi\psi']_{-\infty}^{+\infty} - \int_{-\infty}^{+\infty} |\psi'|^2 dx\right) + \int_{-\infty}^{+\infty} \mathrm{V}(\mathrm{x})|\psi|^2 dx}{\int_{-\infty}^{+\infty} |\psi|^2 dx} \tag{7}$$

Une fonction d'onde doit vérifier que : $\int_{-\infty}^{+\infty} |\psi|^2 dx = 1$, donc on trouve :

L'énergie cinétique $E_c = \frac{-\hbar^2}{2m}\left([\psi\psi']_{-\infty}^{+\infty} - \int_{-\infty}^{+\infty} |\psi'|^2 dx\right)$ et

L'énergie potentielle $E_p = \int_{-\infty}^{+\infty} \mathrm{V}(\mathrm{x})|\psi|^2 dx$

Le potentiel test que nous étudierons ici est celui d'un oscillateur harmonique à une

dimension ($V(x) = \frac{1}{2}m\omega x^2$) dont les énergies propres analytiques sont $E = \hbar\omega(\frac{1}{2} + n)$ où n est un entier naturel. Pour l'état fondamental avec la fonction d'onde $\psi_0(x) = (\frac{m\omega}{\pi\hbar})^{\frac{1}{4}} e^{-\frac{m\omega x^2}{2\hbar}}$, on obtient :

$$E_c = \frac{\hbar\omega}{4} \quad ; \quad E_p = \frac{\hbar\omega}{4} \quad ; \quad E_{totale} = \frac{\hbar\omega}{2}$$

On prendra m, ħ, c et ω égaux à 1 par la suite.

A. États propres d'un oscillateur harmonique à une dimension : méthode Runge-Kutta

n	Énergie
0	0.487663
1	1.46298
2	2.43830
3	3.41361
4	4.38893

TABLE 1. Énergies trouvées grâce à la propagation RK4 **[11]**

L'algorithme RK d'ordre 4 **[11]** est complété d'une méthode de tir nous permet de trouver les énergies (table 1) et les fonctions d'ondes (figure 1) des états liés. On observe que, contrairement à la théorie, les énergies trouvées ne sont pas 0.5, 1.5, 2.5...etc. Une erreur moyenne de 6% est commise. L'écart entre chaque niveau est constant et vaut environ 0.97532 (contre 1 théoriquement). Ces problèmes sont peut-être dus à l'accumulation d'erreurs de calcul au fil de la propagation. Cette méthode n'étant cependant pas le sujet principal du stage, nous nous sommes concentrés sur les réseaux de neurones.

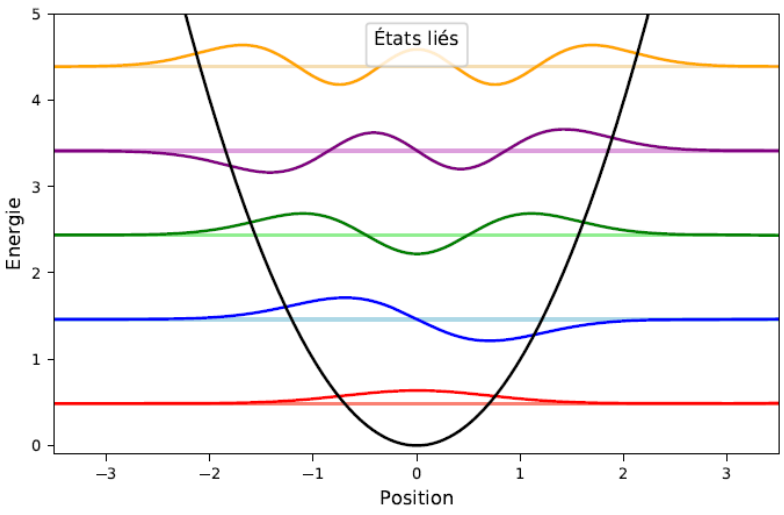

Fig. 1. États liés d'un oscillateur harmonique. Les fonctions d'onde (non normalisées) sont alignées sur leurs énergies. **[11]**

2. Réseaux de neurones – illustration du théorème d'approximation universelle

Selon le théorème d'approximation universelle **[2]**, un réseau de neurone peut reproduire n'importe quelle fonction à valeurs réelles et à support compact. Nous essayons ici d'illustrer ce théorème en approximant l'état fondamental d'un oscillateur harmonique. Des tutoriels d'initiation aux réseaux de neurones écrits par Colin Bernet sont disponibles en bibliographie : **[3]**, **[4]** et **[5]**.

A. Introduction au concept des réseaux de neurones

(1) Géométries :

On voit sur la figure 2 que les réseaux peuvent être décomposés en trois parties : la couche d'entrée (rouge) depuis laquelle les données x sont propagées vers les couches cachées (violet) puis le résultat final y_{pred} est calculé en couche de sortie (bleu). Les réseaux que nous étudions ici prennent une discrétisation de l'espace (x) en entrée et renvoient une fonction d'onde (y_{pred}) en sortie. Par la suite, j'utiliserai le mot "couche" pour désigner les couches cachées des réseaux. Aussi, un réseau 200x200 par exemple désignera un réseau à deux couches cachées, toutes deux composées de 200 neurones. Tous les réseaux que nous étudions disposent d'une entrée et d'une sortie seulement.

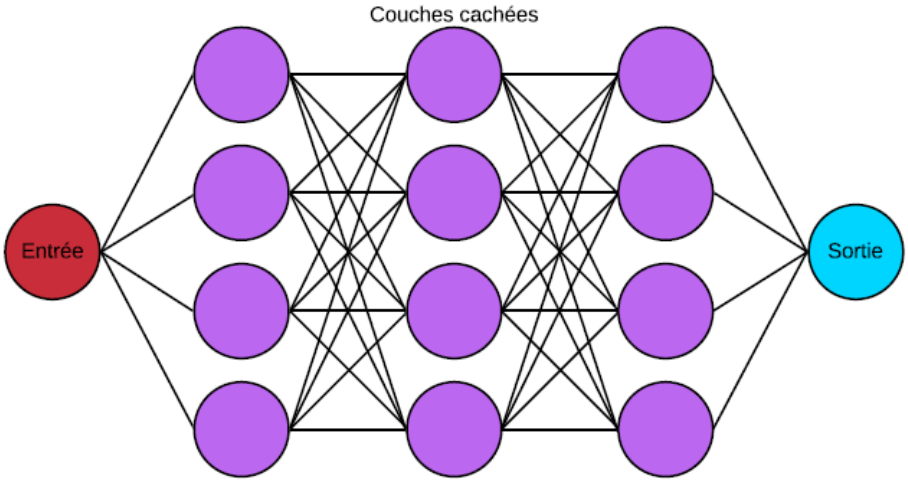

Fig. 2. Schéma d'un réseau de neurones : entrée, 3 couches cachées et sortie.

(2) Concept et vocabulaire :

Un réseau de neurones artificiels, ou réseau neuronal artificiel, est un système dont la conception est à l'origine schématiquement inspirée du fonctionnement des neurones biologiques, et qui par la suite s'est rapproché des méthodes statistiques. Les neurones artificiels composant le réseau sont tous dotés d'une fonction d'activation qui détermine l'activation ou la non activation des neurones pour une entrée donnée. La fonction d'activation que nous utilisons ici est une ReLU (Rectified Linear Unit) qui prend la forme suivante :

$$R(z) = \begin{cases} 0 & quand \quad z \leq 0 \\ z & quand \quad z > 0 \end{cases} \tag{8}$$

Un neurone doté d'une telle fonction d'activation est éteint lorsque $z \leq 0$ et est allumé pour $z > 0$. Cette variable prend la forme z = ax + b où x est la valeur qui entre dans la fonction d'activation, et où a et b sont des quantités modifiées par entrainement du réseau, respectivement le poids et le biais. Un neurone est donc composé d'une fonction d'activation qui prend en entrée x les sorties R(z) des neurones de la couche qui précède la sienne (figure 2). Prenons un exemple fictif dans lequel on cherche à ce que notre réseau reproduise la fonction discrète suivante :

$$f(x) = \begin{cases} 1 & quand \quad x = 0 \\ 7 & quand \quad x = 1 \\ 3 & quand \quad x = 2 \end{cases} \tag{9}$$

On la représente par deux listes de trois nombres : x = [0, 1, 2] et y_{cible}= [1, 7, 3] puis on entraine notre réseau à reproduire y_{cible}. Pour ce faire, on commence par initialiser le réseau avec des paramètres (a et b) aléatoires puis on lui fournit x et y_{cible}. On propage l'ensemble des données x dans le réseau qui se charge de faire une prédiction y_{pred} qu'on compare ensuite à y_{cible}. Un epoch correspond à une propagation de toutes les données x.

Entraînement :

Epoch 1 : Le réseau fait une première prédiction y_{pred} = [32, −9, 64] éloignée de la liste attendue. La fonction de coût calcule l'écart quadratique moyen entre y_{cible} et y_{pred} appelé perte (loss). Le réseau calcule ensuite le gradient de la fonction de coût par rapport à a et b pour savoir comment les modifier afin de minimiser le loss. Il les modifie en conséquence, fin du premier epoch.

Epoch 2 : Le réseau prédit y_{pred} = [11, 3, 9]. La prédiction est meilleure grâce au premier entraînement mais doit être améliorée. Le réseau calcule le loss, le gradient, puis modifie ses paramètres et recommence le processus pour l'epoch 3. Ces calculs sont gérés automatiquement par Keras **[6]**, la librairie Python que nous utilisons ici.

B. Reproduction de l'état fondamental d'un oscillateur harmonique

L'oscillateur harmonique à une dimension est un système facile à rechercher. Notre objectif est que les réseaux de neurones puissent être utilisés pour étudier les systèmes complexes, par conséquent, nous partons d'un oscillateur harmonique et examinons si cette méthode fonctionne ou pas.

Donc on cherche maintenant à reproduire l'état fondamental d'un oscillateur harmonique à une dimension. On dispose d'une liste x discrétisée de −5 à 5 sur 10001 points, et d'une liste y_{cible} calculée à partir de la solution analytique **[7]** de l'équation (1) pour l'état fondamental ψ_{EF} d'énergie E_{EF}:

$$\psi_{EF}(x) = \left(\frac{m\omega}{\pi\hbar}\right)^{\frac{1}{4}} e^{\frac{-m\omega x^2}{2\hbar}} \quad ; \quad E_{EF} = \frac{1}{2} \tag{10}$$

Les courbes noire, rouge et orange de la figure 3 illustrent une fonctionnalité de Keras permettant de mesurer le loss à chaque epoch. La courbe bleue est une moyenne sur 30 runs du programme de reproduction. Sur la courbe rouge, on voit que le minimum de la fonction de coût a été atteint vers le 45e epoch environ. Néanmoins, le réseau continue à modifier ses paramètres lors des epochs suivants et oscille autour du minimum de la fonction de coût. Le loss augmente et les prédictions s'éloignent de la cible. L'aléatoire devient important à partir du 30e epoch environ. Autrement dit, au-delà de ce seuil, deux runs du programme pourront donner des résultats très différents.

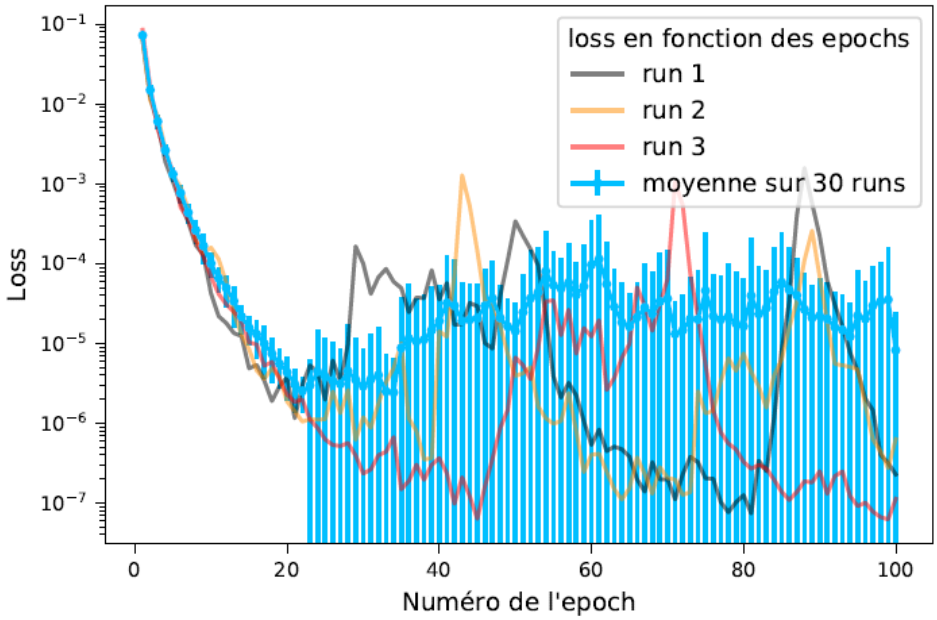

Fig. 3. Écart quadratique moyen (loss) entre ψ_{EF} et les prédictions du réseau (200x200x200) en fonction des epochs

Maintenant, on va écrire un programme qui peut trouver l'état fondamental d'un

potentiel harmonique en utilisant un réseau de neurones. Comme indiqué précédemment, on dispose d'une liste x discrétisée de −5 à 5 sur 10001 points et ensuite, on utilise une boucle simple et l'équation (10) pour calculer la vraie fonction d'onde point par point. Après, on fait une approximation de la fonction d'onde par machine learning. Ici, on utilise un réseau de neurones de la taille 200×200 et on fait les prédictions avec 45 epochs. En fin, on réussit à faire une approximation par réseau de neurones avec custom loss. Le résultat qu'on a trouvé est présenté dans la figure 4.

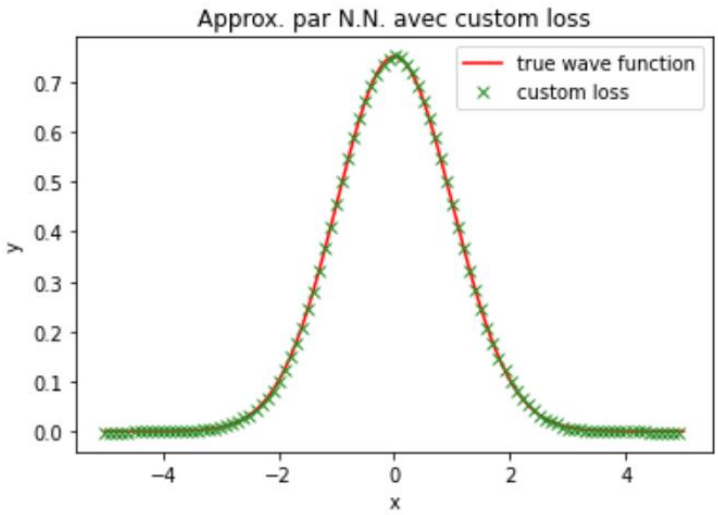

Fig. 4. Approximation par réseau de neurones

Une étude du loss en fonction des epochs a été faite pour comparer l'impact de la géométrie sur la minimisation du loss **[8]**. On y voit que l'ajout de non linéarité (plusieurs couches) permet d'atteindre un loss plus faible, et ce en moins d'epochs que pour un réseau à une couche. On y voit aussi que l'augmentation du nombre de paramètres permet d'obtenir des loss plus bas mais que les oscillations autour du minimum de la fonction de coût ont une plus grande amplitude qu'avec des réseaux moins denses.

C. Calcul de l'énergie de la reproduction

Une fois l'état fondamental reproduit par le réseau, on peut en extraire l'énergie avec l'équation (7). Une première tentative d'extraction nous avait posé problème du fait que les prédictions du réseau sont des morceaux de droites. Un bon apprentissage permet de lisser le résultat mais la dérivée de la prédiction restait discontinue. Nous avons donc extrait l'énergie à partir de splines cubiques de la prédiction dont le calcul **[9]**, la dérivation et l'intégration **[10]** sont gérés par la librairie SciPy.

Pour calculer l'énergie, on crée une fonction 'energy compute' avec les

paramètres : abscisse, fonction et le potentiel. En utilisant la librairie SciPy et l'équation (6), on peut calculer l'énergie et l'utiliser comme valeur de retour de la fonction. Ici, on est dans le cas d'un oscillateur harmonique à une dimension, donc le potentiel $V(x) = \frac{1}{2}m\omega x^2$ et la fonction d'onde du état fondamental $\psi_0(x) = (\frac{m\omega}{\pi\hbar})^{\frac{1}{4}} e^{-\frac{m\omega x^2}{2\hbar}}$. En substituant ces termes comme les paramètres dans la fonction, on trouve directment l'énergie associée à la vraie fonction d'onde.

Encore une fois, on fait une approximation de la fonction d'onde par machine learning et cette fois-ci on peut aussi calculer l'énergie associée à la fonction de prédiction approximée par le réseau de neurones et la comparer avec le résultat qu'on a trouvé précédemment. On peut voir sur la figure 5 que, les deux résultats sont très proches après entrainement.

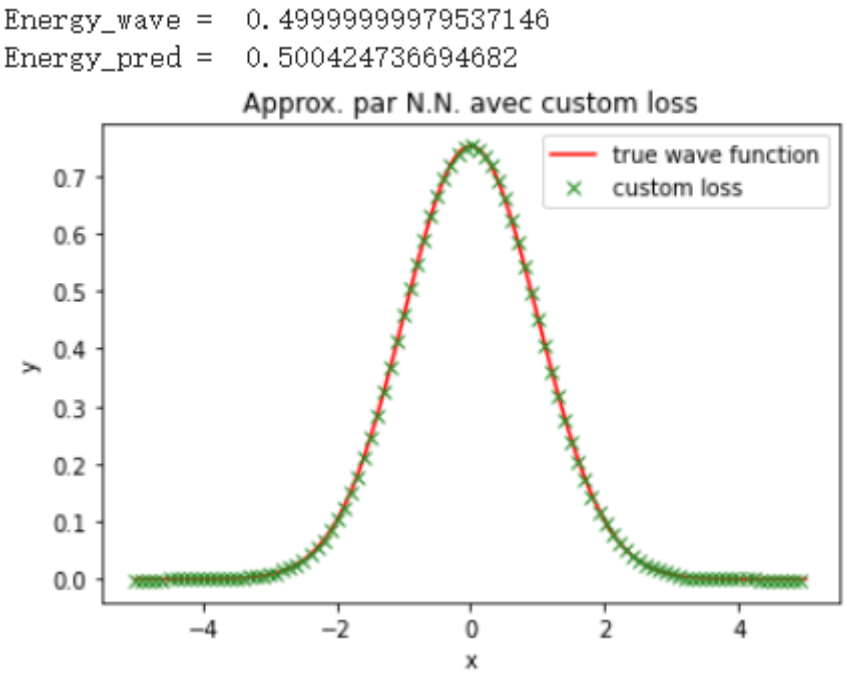

Fig.5. Calcul de l'énergie

D. L'évolution de l'énergie de prédiction au cours du entraînement et le taux de convergence

Maintenant on va étudier comment l'énergie de prédiction obtenu par réseau de neurones varie au cours de l'entraînement. Pour faire ça, il faut créer une boucle de l'entraînement dans la partie de machine learning du programme. On change le nombre d'epoch à 1 et on affiche l'énergie de prédiction pour chaque entraînement. Ainsi, on peut voir directment comment l'énergie de prédiction varie après chaque entraînement et comment elle évolue au cours de l'augmentation du nombre des

entraînements. De plus, on peut définir le taux de convergence comme la différence entre l'énergie de prédiction calculée après chaque entraînement et celle obtenu au cours de la boucle précédente. Par exemple, si on fixe le nombre de boucle à 10, nous obtiendrons le résultat indiqué dans la figure 6. On peut constater que l'énergie de prédiction se rapproche généralement de la valeur théorique lorsque le nombre de l'entraînement augmente, sauf pour les septième et dixième boucles. Dans le même temps, la valeur absolue du taux de convergence diminue au fil du temps aussi. Et si on continue à faire les entraînements, l'énergie sera très proche de la valeur théorique à partir du 15e boucle environ. Le code du programme que nous avons exécuté se trouve dans l'annexe.

```
20/20 [==============================] - 0s 2ms/step - loss: 0.0978
0 Energy_pred =  1.944539493578198 , convergence rate =  1.944539493578198
20/20 [==============================] - 0s 2ms/step - loss: 0.0249
1 Energy_pred =  1.3760642303936363 , convergence rate =  -0.5684752631845618
20/20 [==============================] - 0s 2ms/step - loss: 0.0112
2 Energy_pred =  1.0130795823094794 , convergence rate =  -0.362984648084157
20/20 [==============================] - 0s 2ms/step - loss: 0.0085
3 Energy_pred =  0.7854222639597843 , convergence rate =  -0.22765731834969505
20/20 [==============================] - 0s 2ms/step - loss: 0.0054
4 Energy_pred =  0.7192702749954685 , convergence rate =  -0.06615198896431584
20/20 [==============================] - 0s 2ms/step - loss: 0.0034
5 Energy_pred =  0.6700134145768722 , convergence rate =  -0.04925686041859623
20/20 [==============================] - 0s 2ms/step - loss: 0.0025
6 Energy_pred =  0.7491183408044697 , convergence rate =  0.07910492622759746
20/20 [==============================] - 0s 2ms/step - loss: 0.0019
7 Energy_pred =  0.6355317893232151 , convergence rate =  -0.11358655148125463
20/20 [==============================] - 0s 2ms/step - loss: 8.7510e-04
8 Energy_pred =  0.5675872755545127 , convergence rate =  -0.06794451376870236
20/20 [==============================] - 0s 2ms/step - loss: 7.9317e-04
9 Energy_pred =  0.6192276642313073 , convergence rate =  0.051640388676794546

Energy_wave =  0.49999999979537146
```

Fig.6. L'évolution de l'énergie de prédiction

3. Minimisation de l'énergie pour trouver l'état fondamental

Jusqu'ici nous avons étudié la qualité d'approximateur universel des réseaux de neurones mais notre problème n'est pas résolu. Notre but maintenant est de développer une méthode pour que le programme trouve l'état fondamental sans le connaitre au préalable.

A. Théories concernant la minimisation de l'énergie

Dans la partie 2, nous avons calculé l'énergie d'un oscillateur harmonique quantique à une dimension en utilisant la fonction d'onde et l'expression du potentiel.

Mais il existe un théorème qui nous dit que si nous utilisons la fonction d'onde fondamentale propre, l'énergie trouvée sera minimale. Pour les autres fonctions d'onde propres ou toute autre fonction d'onde, ce n'est pas le cas.

Dans cette partie, nous supposons que nous ne connaissons pas la fonction d'onde propre et faisons comme si le potentiel était n'importe quel potentiel, c'est-à-dire le système pourrait être représenté par n'importe quel potentiel. Et c'est un test de la méthode sur un résultat connu.

Pour tous les ψ dans un ensemble W qui représente l'ensemble de toutes les fonctions d'onde possibles (même les fonctions d'onde impropres), on peut calculer l'énergie E(ψ). Si nous trouvons la fonction d'onde ψ_0 avec l'énergie la plus faible $E_0 = E(\psi_0)$, alors il y a un théorème qui nous dit que c'est la fonction d'onde fondamentale propre.

B. Création d'une fonction de perte

Pour résoudre ce problème, il suffit de créer une fonction de perte ('loss function' en anglais) qui prend l'énergie de la fonction d'onde représentée par le réseau comme la valeur de retour. En faisant cela, le réseau de neurone va minimiser l'énergie au cours de l'entraînement et la fonction de prédiction qu'on obtient se rapprochera de plus en plus de la fonction d'onde fondamentale propre. Et finalement, on peut réussir à trouver l'état fondamental de ce système.

Tout d'abord, on essaie de créer une fonction de perte pour minimiser la différence entre la vraie fonction d'onde et la fonction approximée par le réseau de neurone. Pour faire ça, on prend ces deux termes comme les arguments de la fonction de perte. Ensuite, on calcul leur erreur quadratique moyenne et la prend comme la valeur de retour de cette fonction. Ici, il faut faire attention que le type des arguments de cette fonction est un tenseur et donc on doit faire le calcul avec les tenseurs. Finalement, nous obtiendrons le même résultat que dans la deuxième partie.

Maintenant, l'étape suivante consiste à changer la valeur de retour de la fonction perte à l'énergie calculée a partir de la fonction de prédiction. L'idée n'est pas très compliquée, mais on rencontre beaucoup de problèmes quand on essaie de faire une transformation du type tenseur au type tableau numpy ('numpy array' en anglais). Malheureusement, on n'a pas encore trouvé une solution correcte pour cette transformation et c'est une piste de travail non terminé car manque de temps.

4. Conclusion

Nous avons étudié deux méthodes numériques de résolution de l'équation de

Schrödinger indépendante du temps permettant de trouver la fonction d'onde et l'énergie de l'état fondamental d'un oscillateur harmonique. Un algorithme déterministe comme RK4 demande trop de calcul quand la dimension N du problème devient très grande. Et le programme utilisant les réseaux de neurones permet de reproduire l'état fondamental d'un oscillateur harmonique et calculer l'énergie de reproduction, mais cette méthode n'est pour l'instant pas du tout efficace. Dans le cas de problème beaucoup plus compliqué, on espère qu'elle deviendra efficace et c'est pourquoi nous avons choisi les réseaux de neuronaux comme le sujet. Mais en ce qui concerne la dernière partie de la minimisation de l'énergie, il reste encore des problèmes non résolus : comment faire une transformation correcte de tenseur au tableau numpy? Y a-t-il une autre solution pour réaliser la minimisation de l'énergie, par exemple changer les arguments de la fonction 'energy compute' aux tenseurs et faire les calculs directment en tenseurs? De plus, quelles sont des performances des réseaux de neurones sur des temps plus longs? Quelles sont les performances du programme à deux et trois dimensions ? Nous réfléchissons aussi à une méthode permettant de trouver les états excités, éventuellement en travaillant dans un sous-espace de fonctions orthogonales.

Bibliographie

[1] J. W. T. KEEBLE et A. RIOS. "Machine learning the deuteron". In : (2019). URL : arXiv:1911.13092.

[2] Balázs Csanád CSÁJI. "Approximation with Artificial Neural Networks, MSc Thesis". In: Eötvös Loránd University (ELTE), Budapest, Hungary 24 :48 (2001). URL :https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.101.2647&rep=rep1&type=pdf

[3] Colin BERNET. Handwritten Digit Recognition with scikit-learn. URL : https://thedatafrog.com/en/articles/handwritten-digit-recognition-scikit-learn/

[4] Colin BERNET. Le réseau à un neurone : régression logistique. URL : https://thedatafrog.com/fr/articles/logistic-regression/

[5] Colin BERNET. Premier réseau de neurones avec keras. URL : https://thedatafrog.com/fr/articles/first-neural-network-keras/

[6] KERAS. Model training. URL : https://keras.io/api/models/model_training_apis/

[7] Claude COHEN-TANNOUDJI, Bernard DIU et Franck LALOË. Mécanique quantique Tome 1. EDP Sciences, 2018, p. 371-378. ISBN : 9782759822874.

[8] Clément LOTTEAU. GitHub. URL :

https://github.com/quadrivecteur?tab=repositories
[9] SCIPY. Interpolation avec SciPy. URL :
https://docs.scipy.org/doc/scipy/reference/tutorial/interpolate.html
[10]SCIPY. Integration avec SciPy. URL :
https://docs.scipy.org/doc/scipy/reference/tutorial/integrate.html
[11]Clément LOTTEAU. GitHub. URL :
https://github.com/quadrivecteur/Runge_Kutta_4

Annexe

```
import numpy as np
import matplotlib.pyplot as plt
import math
import keras
from keras import layers, models, optimizers
import tensorflow as tf
from scipy import integrate
from scipy import interpolate
from scipy.stats import norm


#energy
def energy_compute (a,b,abscissa,function,pot):
    #interpolations
    tck_true = interpolate.splrep(abscissa, function, k=3, s=0)            #W.F.
    tck_VOH = interpolate.splrep(abscissa, pot, k=3, s=0)                  #P.F
    tck_true_carre = interpolate.splrep(abscissa, function*function, k=3, s=0)      #W.F.squared
    tck_true_pot_true_carre = interpolate.splrep(abscissa, pot*function*function, k=3, s=0)   #P.F * W.F.squared
    der_true = interpolate.splev(abscissa, tck_true, der=1)                #W.F.derivative
    tck_true_der = interpolate.splrep(abscissa,der_true*der_true, k=3,s=0)     #W.F.derivative.squared
    int_true_carre = interpolate.splint(a,b,tck_true_carre)                #integral of W.F.squared
    int_pot_true_carre = interpolate.splint(a,b,tck_true_pot_true_carre)       #integral of P.F * W.F.squared
    int_true_der = interpolate.splint(a,b,tck_true_der)                    #integral of W.F.derivative.squared
    Energy = ((-pow(hbar,2)/(2*m))*(function[b]*der_true[b]-function[a]*der_true[a]
                    - int_true_der) + int_pot_true_carre) / int_true_carre
    return Energy
```

```
#oscillateur harmonique V(x)=0.5*m*omega*omega*x*x

hbar=1 #Planck constant
omega=1 #w
m=1
a=-5
b=5
pts=1000
linx = np.linspace(a,b,pts)
#print(linx)
h = (b-a)/float(pts)
VOH = np.zeros_like(linx)
wave = np.zeros_like(linx)
#oscillateur harmonique: wave function, potential function
VOH = 0.5*m*omega*omega*linx*linx #potential function of oscillateur harmonique
wave = pow(m*omega/(math.pi*hbar),0.25)*np.exp(-m*omega*(linx*linx)/(2*hbar))
#wave function of oscillateur harmonique

energy_wave = energy_compute(a,b,linx,wave,VOH)

#machine learning: predication function
model = models.Sequential([
                                                layers.Dense(200, input_shape=(1,), activation='relu'),
                                                layers.Dense(200, input_shape=(1,), activation='relu'),
                                                layers.Dense(1),
]) #keras
```

```
model.summary()  #print
opt  =  optimizers.Adam(learning_rate=0.001)
model.compile(loss='mean_squared_error',  optimizer=opt)

fits=30
energy_preds_0  =  0
for  i  in  range(0,fits):
        model.fit(linx,wave,batch_size=50,epochs=1)
        predictions  =  model.predict(linx)
        preds  =  predictions.reshape(-1)
        energy_preds  =  energy_compute(a,b,linx,preds,VOH)
        energy_rate  =  energy_preds  -  energy_preds_0
        print(i,'Energy_pred  =  ',energy_preds,',  convergence  rate  =  ',energy_rate)
        energy_preds_0=energy_preds

print('')
print('Energy_wave  =  ',energy_wave)

plt.title('Approx.  par  N.N.  avec  custom  loss')
plt.plot(linx,wave,c='r',label  =  'true  wave  function')
plt.plot(linx[0:pts-1:10],preds[0:pts-1:10],marker='x',c='forestgreen',label  =  'custom  loss',linestyle='None')
plt.ylabel('y')
plt.xlabel('x')
plt.legend()
plt.savefig('custom_loss.pdf')
```