# UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

## FACULTAD DE INGENIERÍA

Inteligencia artificial en la predicción de fuerzas interatómicas

**TESIS**

Que para obtener el título de

**Ingeniero en computación**

**P R E S E N T A**

Yardiel Bonilla Galicia

**DIRECTOR(A) DE TESIS**

Dr. Ruben Satamaria Ortiz

**Ciudad Universitaria, Cd. Mx., 2023**

# Índice general

# Índice de figuras

# Índice de cuadros

# Capítulo 1

# Introducción

La obtención de las fuerzas internas que sufren los fluidos, siempre ha sido una de gran interés para todas las áreas de la ciencia que emplean algún sistema con este tipo de materia. El agua al ser una sustancia con una configuración química relativamente simple, resulta fácil emplearla en cualquier medio, ya sea que, se utilice para generar energía, como medio de disolución, para cambiar su estado o simplemente estudiar su comportamiento ante diferentes condiciones. Aunque existen diferentes elementos químicos con una configuración atómica aun sencilla, el agua es considerada como el disolvente universal por la capacidad de formar puentes de hidrógeno con otras sustancias, como lo son los alcoholes, azúcares, proteínas, etc.

Conocer el comportamiento del agua en diferentes condiciones siempre depende de un sistema computacional que calcule el movimiento molécula a molécula, de esta forma se asegura un conducta congruente. Dentro de este análisis siempre tendremos un limite a la hora de analizar sistemas de grandes dimensiones ya sea que, el tiempo requerido sea imposible de tolerar, o bien, el hardware empleado no soporte esta carga. Por ello es necesario que se desarrollen nuevas tecnologías que resuelvan este tipo de problemas de una forma mas eficiente.

Este trabajo consiste en la creación de una red neuronal artificial que sea capaz de agilizar el calculo de las fuerzas de Coulomb y las fuerzas de Lennard-Jonnes, prediciendo estas fuerzas, en vez de calcularlas. Por ello, en este trabajo se abarcan temas que van desde como interaccionan los átomos de agua y por ende la obtención de las fuerzas que producen estas interacciones, la forma en como evoluciona el modelo matemático de una neurona artificial hasta convertirse en una red de neuronas artificiales que es capaz de guardar conocimiento e incluso optimizar su comportamiento, también temas que explican algoritmos que tienen la capacidad de extraer información con la máxima relevancia posible (algoritmo de dividir y conquistar, desplazamiento al origen, desplazamiento sobre los ejes x,y, cambio de coordenadas, filtro de suavizado, filtro resalte de bordes y conversión posicional) para así agilizar en entrenamiento de una red neuronal artificial.

El objetivo general consta en disminuir el tiempo requerido que toma el calculo de las fuerzas ínter-atómicas cuando se emplea un contenedor con un número de átomos inmenso, así como la dependencia de un equipo de computo con un hardware de alto rendimiento (servidor).

# Capítulo 2

# Interacciones Atómicas

## 2.1. Fuerzas de Coulomb

Cada cuerpo en el espacio esta sujeto a una interacción con los demás objetos que lo rodean. La forma en como se relacionan dichos objetos depende de la carga eléctrica que posea cada uno en ese momento.

Coulomb establece que la fuerza eléctrica de interacción entre dos partículas cargadas en reposo tiene las siguientes propiedades:

- La fuerza entre las dos partículas es inversamente proporcional al cuadrado de la separación entre ellas, denotado con la letra r.
- La fuerza es proporcional al producto de las cargas q1 y q2 de las partículas.
- La fuerza atrae a las partículas si las cargas son de signo opuesto y las rechaza si las cargas tienen el mismo signo.

La ecuación de la fuerza Coulombiana entre la partícula 1 y 2 es:

$$F(r) = (1/4\pi\epsilon)(q_1 q_2 / r_{12}^2) \tag{2.1.1}$$

Donde $\epsilon$ se le conoce como la constante eléctrica, la cual depende del medio donde se encuentre.

En su forma vectorial, la fuerza de Coulomb se puede escribir como:

$$\begin{aligned} &F(r) = \tfrac{1}{4\pi\epsilon}\tfrac{q_1 q_2}{|r_{12}|^2}\,\hat{\mathbf{r}}_{12} \quad ; \quad \hat{\mathbf{r}}_{12} = \tfrac{\vec{r}_{12}}{|\vec{r}_{12}|} \quad ; \quad \vec{r}_{12} = (\vec{r}_1 - \vec{r}_2) \quad ; \quad \vec{r}_1 = (x_1, y_1, z_1) \\ &\therefore \quad F(r) = \tfrac{1}{4\pi\epsilon}\tfrac{q_1 q_2}{|r_{12}|^3}\,\vec{\mathbf{r}}_{12} \\ &F_x(r) = \tfrac{1}{4\pi\epsilon}\tfrac{q_1 q_2}{|r_{12}|^3}\,x_{12} \quad ; \quad F_y(r) = \tfrac{1}{4\pi\epsilon}\tfrac{q_1 q_2}{|r_{12}|^3}\,y_{12} \quad ; \quad F_z(r) = \tfrac{1}{4\pi\epsilon}\tfrac{q_1 q_2}{|r_{12}|^3}\,z_{12} \end{aligned} \tag{2.1.2}$$

En la mayoría de las ocasiones un sistema contiene muchas partículas. Entre estas partículas la fuerza electromagnética, presenta una forma similar a la de la gravedad, lo que quiere decir que tiene un alcance infinito el cual, se debilita con la distancia pero, es lo suficientemente

lento como para que las partículas muy lejanas aun lo sientan, lo que nos obliga a tomar en cuenta cada una de las partículas que se encuentren en el medio. A pesar de ello, las interacciones de Coulomb mantienen un carácter aditivo por pares de las interacciones. Por ejemplo, la fuerza de Coulomb sobre la partícula 1 queda definida por la fuerza 1, 2, etc.

$$\begin{aligned} F_1(r) &= \tfrac{1}{4\pi\epsilon}\tfrac{q_1q_2}{|r_{12}|^3}\ \vec{\mathbf{r}}_{12} + \tfrac{1}{4\pi\epsilon}\tfrac{q_1q_3}{|r_{13}|^3}\ \vec{\mathbf{r}}_{13} + \tfrac{1}{4\pi\epsilon}\tfrac{q_1q_4}{|r_{14}|^3}\ \vec{\mathbf{r}}_{14} + \cdots \\ F_2(r) &= \tfrac{1}{4\pi\epsilon}\tfrac{q_1q_2}{|r_{21}|^3}\ \vec{\mathbf{r}}_{21} + \tfrac{1}{4\pi\epsilon}\tfrac{q_2q_3}{|r_{23}|^3}\ \vec{\mathbf{r}}_{23} + \tfrac{1}{4\pi\epsilon}\tfrac{q_2q_4}{|r_{24}|^3}\ \vec{\mathbf{r}}_{24} + \cdots \end{aligned} \tag{2.1.3}$$

El desplazamiento de la partícula 1 esta dada por la fuerza $\mathbf{F}_1$. De igual forma se obtiene el desplazamiento de la partícula 2,3,etc. Todo esto ocurre al tiempo, digamos $t_1$. Para denotar la dependencia en el tiempo escribimos la ecuación 2.1.3 en términos de $t_1$.

$$F_1(r(t_1)) = \tfrac{1}{4\pi\epsilon}\tfrac{q_1q_2}{|r_{12}|^3}\ \vec{\mathbf{r}}_{12}(t_1) + \tfrac{1}{4\pi\epsilon}\tfrac{q_1q_3}{|r_{13}|^3}\ \vec{\mathbf{r}}_{13}(t_1) + \tfrac{1}{4\pi\epsilon}\tfrac{q_1q_4}{|r_{14}|^3}\ \vec{\mathbf{r}}_{14}(t_1) + \cdots \tag{2.1.4}$$

En un instante de tiempo posterior $t_2$ el nuevo desplazamiento de la partícula 1 esta dada por $F_1(r(t_2))$.

$$F_1(r(t_2)) = \tfrac{1}{4\pi\epsilon}\tfrac{q_1q_2}{|r_{12}|^3}\ \vec{\mathbf{r}}_{12}(t_2) + \tfrac{1}{4\pi\epsilon}\tfrac{q_1q_3}{|r_{13}|^3}\ \vec{\mathbf{r}}_{13}(t_2) + \tfrac{1}{4\pi\epsilon}\tfrac{q_1q_4}{|r_{14}|^3}\ \vec{\mathbf{r}}_{14}(t_2) + \cdots \tag{2.1.5}$$

Y así mismo para las demás partículas.

## 2.2. Fuerzas de Lennard-Jones

Las partículas experimentan fuerzas y energías en un cúmulo de moléculas. Dependiendo de la carga de las partículas, se establecerá una atracción o repulsión electrostática entre ellas. Así para evitar que, por ejemplo, un par de partículas lleguen a contacto pleno cuando sus cargas son opuestas, o para evitar que se alejen hasta el infinito una de otra cuando sus cargas son iguales, se introduce el potencial de interacción de Lennard - Jones (LJ).

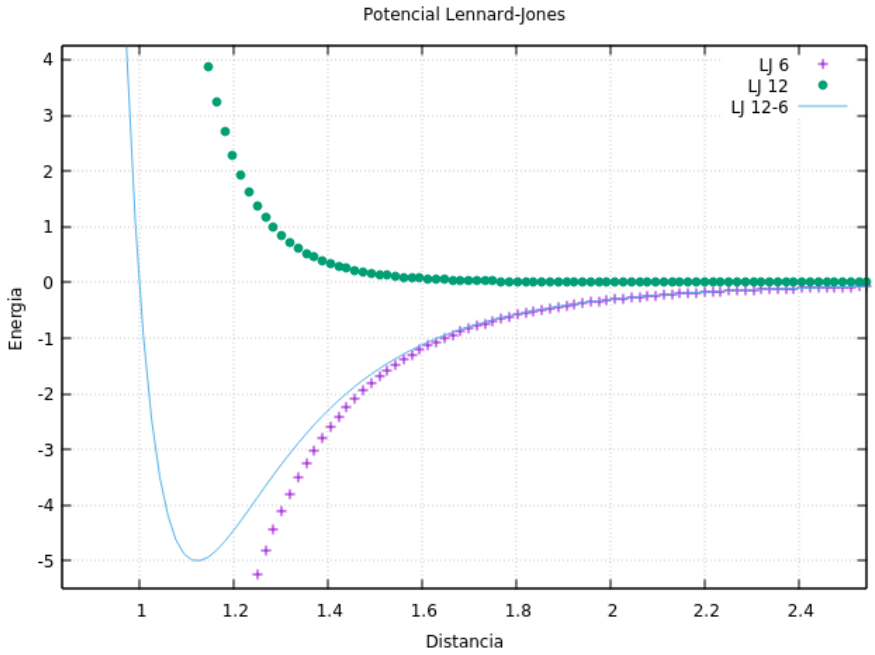
La función que describe el comportamiento suave de repulsión y atracción es la siguiente

$$\mathbf{V}_{LJ}(r) = 4e[(\sigma/r)^{12} - (\sigma/r)^6] \tag{2.2.1}$$

Donde r es la distancia que existe entre un par de partículas, $\sigma$ es una distancia de referencia, y $e$ es una energía de referencia, cuyos valores dependerán del tipo de átomos interactuantes con los que se estén trabajando. El primer termino de la ecuación introduce una energía de repulsión entre las partículas. Dicho termino va disminuyendo de forma gradual hasta aproximarse a cero:

$$\sigma/r^{12} \tag{2.2.2}$$

Figura 2.2.1: Gráfica de las energías de Lennard-Jones




El segundo termino describe la energía de atracción debido al signo negativo de esta contribución. Dicho termino provoca que las partículas se aproximen entre ellas.

$$-\sigma/r^6 \tag{2.2.3}$$

La unión de las contribuciones anteriormente mencionadas, es decir $\sigma/r^{12}$ y $-\sigma/r^6$, describe el equilibrio electrostático entre las dos partículas. Este equilibro se consigue a partir de la intersección de las energías de los cuerpos cuando se encuentran en un mínimo de energía, si una partícula se repele está al mismo tiempo generara una energía de atracción que evitara su desprendimiento, en el caso contrario si una partícula se atrae está generara una energía de repulsión que evitara el choque. La grafica subsecuente describe este comportamiento en términos numéricos.

Para obtener la fuerza que ejercen las partículas entre si es necesario derivar la ecuación 2.2.1 en términos de la distancia. por regla de la cadena

$$F = \frac{dV(r)}{dx} = -\frac{dV(r)}{dr}\ \frac{dr}{dx} \tag{2.2.4}$$

calculando el primer termino de la ecuación y reescribiendo la ecuación 2.2.1

$$\begin{aligned} V(r) &= 4e[\sigma^{12}r^{-12} - \sigma^6 r^{-6}] \\ \frac{dV(r)}{dr} &= 4e[-12\sigma^{12}r^{-13} + 6\sigma^6 r^{-7}] = \frac{24}{r}e[-2(\sigma/r)^{12} + (\sigma/r)^6] \end{aligned} \tag{2.2.5}$$

Para resolver el segundo termino, se denota que r es una distancia con posiciones (x, y, z):

$$\begin{aligned} \frac{dr}{dx} &= \sqrt{(x_2-x_1)^2+(y_2-y_1)^2+(z_2-z_1)^2}/dx \;\; = \;\; (x_2-x_1)/r_x \\ \frac{dr}{dy} &= (y_2-y_1)/r_y \quad ; \quad \frac{dr}{dz} = (z_2-z_1)/r_z \end{aligned} \tag{2.2.6}$$

Uniendo las ecuaciones 2.2.6 y 2.2.7 para cada termino, la fuerza se expresa como:

$$\begin{aligned} F_x &= \tfrac{24}{r^2}e[-2(\sigma/r)^{12}+(\sigma/r)^6]\cdot(x_2-x_1) \\ F_y &= \tfrac{24}{r^2}e[-2(\sigma/r)^{12}+(\sigma/r)^6]\cdot(y_2-y_1) \\ F_z &= \tfrac{24}{r^2}e[-2(\sigma/r)^{12}+(\sigma/r)^6]\cdot(z_2-z_1) \end{aligned} \tag{2.2.7}$$

El potencial de Lennard - Jones solo puede ser empleado para la interacción de dos partículas, por lo que, los sistemas complejos deberán ser divididos a una expresión de pares para poder implementar este potencial.

## 2.3. Fuerzas externas entre moléculas de agua

Una vez discutidas las fuerzas Coulombianas y de LJ entre átomos, en esta sección describimos las fuerzas entre moléculas, las cuales están dadas entre átomos de diferentes moléculas de agua y nunca entre átomos de la misma molécula.

Supongamos que tenemos dos moléculas. La molécula uno esta compuesta por los átomos, $O^{(1)}, H_1^{(1)}, H_2^{(1)}$, mientras que la molécula dos está compuesta por los átomos $O^{(2)}, H_1^{(2)}, H_2^{(2)}$. La interacción entre los oxígenos $O^{(1)} - O^{(2)}$ esta dada en nuestra aproximación molecular por la interacción Coulombiana y la de LJ. Por otro lado, las interacciones $O^{(1)} - H_1^{(2)}$, $O^{(1)} - H_2^{(2)}$ están dadas por las interacciones de Coulomb. Finalmente, las interacciones de $H_1^{(1)}$ y $H_2^{(1)}$ con $H_1^{(2)}$ y $H_2^{(2)}$ están dadas por la interacción de Coulomb.

Cuadro 2.1: Parámetros de los modelos de interacción

| | |
|---|---|
| Parámetros armónicos | |
| $r_0 = 0{,}957$ Å | ; $k_r = 1{,}914 E_h/$ Å$^2$ |
| $\theta_0 = 104{,}52°$ | ; $K_\theta = 0{,}239 E_h/rad^2$ |
| | |
| Cargas eléctricas | |
| $q_{A_u} = 0{,}000e$ | |
| $q_O = -0{,}834e$ | |
| $q_H = +0{,}417e$ | |
| | |
| LJ parámetros | |
| $\xi_{A_uA_u} = 8{,}4301 \times 10^{-3} E_h$ ; | $\sigma_{A_uA_u} = 2{,}629$ Å |
| $\xi_{A_uO} = 1{,}4293 \times 10^{-3} E_h$ ; | $\sigma_{A_uO} = 2{,}8781$ Å |
| $\xi_{OO} = 2{,}4234 \times 10^{-4} E_h$ ; | $\sigma_{OO} = 3{,}1507$ Å |

Todas estas interacciones se resumen en la siguiente tabla.

Cuadro 2.2: Interacciones atómicas entre átomos

| Átomo | Átomo | Interacción |
|---|---|---|
| $O^{(1)}$ | $O^{(2)}$ | C, LJ |
| $O^{(1)}$ | $H_1^{(2)}$ | C |
| $O^{(1)}$ | $H_2^{(2)}$ | C |
| $H_1^{(1)}$ | $O^{(2)}$ | C |
| $H_1^{(1)}$ | $H_1^{(2)}$ | C |
| $H_1^{(1)}$ | $H_2^{(2)}$ | C |
| $H_2^{(1)}$ | $O^{(2)}$ | C |
| $H_2^{(1)}$ | $H_1^{(2)}$ | C |
| $H_2^{(1)}$ | $H_2^{(2)}$ | C |

Con objeto de ilustrar las interacciones antes descritas, a continuación describimos las ecuaciones para la interacción Coulombiana entre dos moléculas de agua.

$$
\begin{aligned}
F_1(r(t_1)) = {} & \tfrac{1}{4\pi\epsilon}\tfrac{q_{O^{(1)}}q_{O^{(2)}}}{|r_1|^3}\ \vec{\mathbf{r}}_1(t_1) + \tfrac{1}{4\pi\epsilon}\tfrac{q_{O^{(1)}}q_{H_1^{(2)}}}{|r_2|^3}\ \vec{\mathbf{r}}_2(t_1) + \tfrac{1}{4\pi\epsilon}\tfrac{q_{O^{(1)}}q_{H_2^{(2)}}}{|r_3|^3}\ \vec{\mathbf{r}}_3(t_1) + \\
& + \tfrac{1}{4\pi\epsilon}\tfrac{q_{H_1^{(1)}}q_{O^{(2)}}}{|r_4|^3}\ \vec{\mathbf{r}}_4(t_1) + \tfrac{1}{4\pi\epsilon}\tfrac{q_{H_1^{(1)}}q_{H_1^{(2)}}}{|r_5|^3}\ \vec{\mathbf{r}}_5(t_1) + \tfrac{1}{4\pi\epsilon}\tfrac{q_{H_1^{(1)}}q_{H_2^{(2)}}}{|r_6|^3}\ \vec{\mathbf{r}}_6(t_1) \\
& + \tfrac{1}{4\pi\epsilon}\tfrac{q_{H_2^{(1)}}q_{O^{(2)}}}{|r_7|^3}\ \vec{\mathbf{r}}_7(t_1) + \tfrac{1}{4\pi\epsilon}\tfrac{q_{H_2^{(1)}}q_{H_1^{(2)}}}{|r_8|^3}\ \vec{\mathbf{r}}_8(t_1) + \tfrac{1}{4\pi\epsilon}\tfrac{q_{H_2^{(1)}}q_{H_2^{(2)}}}{|r_9|^3}\ \vec{\mathbf{r}}_9(t_1)
\end{aligned}
\tag{2.3.1}
$$

Donde las distancias $\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3$ ... $\mathbf{r}_9$ corresponden a las distancias entre $O^{(1)}-O^{(2)}$, $O^{(1)}-H_1^{(2)}$, $O^{(1)}-H_2^{(2)}$ ... $H_2^{(1)}-H_2^{(2)}$, respectivamente, y así mismo con las siguientes distancias. De forma similar, se escriben las fuerzas LJ. Estas expresiones se deben generalizar para el caso de $n$ moléculas de agua.

Cuando se consideran $n$ moléculas de agua, los cálculos se escalan como $n^2$. Al trabajar con miles de moléculas de agua los cálculos tienden a saturar a cualquier equipo de cómputo relativamente rápido, aun cuando se haga uso, de la última tecnología tanto de hardware y como de software. Esto nos exige utilizar algoritmos de mayor eficacia, que nos permitan avanzar con pasos más firmes. Por ello se decide emplear Inteligencia artificial, mediante redes neuronales.

## 2.4. Fuerzas internas en las moléculas de agua

En el capítulo anterior se discutieron las interacciones entre diferentes moléculas. Estas interacciones se describieron mediante fuerzas electrostáticas de Coulomb y de Lennar-Jonnes. No obstante, cada una de estas moléculas está integrada por átomos. Particular-

mente una molécula de agua está formada por un oxígeno y dos hidrógenos. La estabilidad de esta molécula la describiremos en esta sección mediante el uso de fuerzas tipo resorte. Precisamente suponemos que el átomo de oxígeno está ejerciendo una fuerza sobre el primer hidrógeno la cual, es normalmente se representada por una fuerza de tipo resorte $F_{OH_1} = -K_r(\mathbf{r_{OH_1}} - \mathbf{r^0_{OH_1}})$. Así mismo, la interacción con el segundo hidrógeno con el oxigeno esta dada por $F_{OH_2} = -K_r(\mathbf{r_{OH_2}} - \mathbf{r^0_{OH_2}})$. La constante K es la misma en ambos casos. La interacción entre los enlaces $OH_1 - OH_2$ también está descrita por una fuerza de resorte $F_{OH_1-OH_2} = -K_\theta(\theta_{HOH} - \theta^0_{HOH})$. En estas ecuaciones $\vec{\mathbf{r^0}}$ y $\theta^0$ son la distancia de equilibrio y el ángulo de equilibrio respectivamente, mientras $K_r$ y $K_\theta$ describen la rigidez de los resortes. La fuerza interna de cada molécula de agua está dada por:

$$
\begin{aligned}
&r_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2} \\
&r^0_{ij} = \sqrt{(x^0_i - x^0_j)^2 + (y^0_i - y^0_j)^2 + (z^0_i - z^0_j)^2} \\
&V(r_{HOH}) = \tfrac{k_r}{2}[r_{OH1} - r^0_{OH1}]^2 + \tfrac{k_r}{2}[r_{OH2} - r^0_{OH2}]^2 + \tfrac{k_\theta}{2}(\theta_{HOH} - \theta^0_{HOH})^2 \\
&V(r_{HOH}) = \tfrac{k_r}{2}[\sqrt{(x_O - x_{H1})^2 + (y_O - y_{H1})^2 + (z_O - z_{H1})^2} - r^0_{OH2}]^2 \\
&+\tfrac{k_r}{2}[\sqrt{(x_O - x_{H2})^2 + (y_O - y_{H2})^2 + (z_O - z_{H2})^2} - r^0_{OH2}]^2 \\
&+\tfrac{k_\theta}{2}(\theta_{HOH} - \theta^0_{HOH})^2
\end{aligned}
\tag{2.4.1}
$$

Preparando las derivadas

$$U(r_{HOH}) = \tfrac{k_r}{2}(r_{OH1} - r^0_{OH1})^2 \qquad ; \tfrac{\partial U(r_{HOH})}{\partial_{x_O}} = -k_r(r_{OH1} - r^0_{OH1})\tfrac{x_O - x_{H1}}{r_{OH1}} \tag{2.4.2}$$

$$V(r_{HOH}) = \tfrac{k_r}{2}(r_{OH2} - r^0_{OH2})^2 \qquad ; \tfrac{\partial V(r_{HOH})}{\partial_{x_O}} = -k_r(r_{OH2} - r^0_{OH2})\tfrac{x_O - x_{H1}}{r_{OH2}} \tag{2.4.3}$$

$$W(\theta_{HOH}) = \frac{k_r}{2}(\theta_{HOH} - \theta^0_{HOH})^2 \qquad ; \cos\theta_{HOH} = \frac{r_{OH1}\,\cdot\, r_{OH2}}{r_{OH1}\, r_{OH2}}$$

$$u = \cos\theta_{HOH} \qquad ; \theta = \arccos(u) \qquad ; \frac{\partial\theta}{\partial u} = -\frac{1}{\sqrt{1-u^2}}$$

$$\frac{\partial u}{\partial_{x_O}} = \frac{1}{r_{OH1}r_{OH2}}\frac{\partial}{\partial x_O}[r_{OH1}\cdot r_{OH2}] + \frac{r_{OH1}\cdot r_{OH2}}{r_{OH2}}(-\frac{x_O - x_{H1}}{r^3_{OH1}} + \frac{r_{OH2}\cdot r_{OH2}}{r_{OH1}}(-\frac{x_O - x_{H2}}{r^3_{OH2}})$$

$$= \frac{(x_O - x_{H1}) + (x_O - x_{H1})}{r_{OH1}r_{OH2}} - \frac{(x_O - x_{H1})}{r^2_{OH1}}\cos\theta_{HOH} - \frac{(x_O - x_{H2})}{r^2_{OH2}}\cos\theta_{HOH}$$

$$\frac{\partial W(\theta_{HOH})}{\partial x_O} = k_\theta \frac{\theta_{HOH} - \theta^0_{HOH}}{\sqrt{1-\cos^2\theta_{HOH}}}[\frac{(x_O - x_{H1}) + (x_O - x_{H1})}{r_{OH1}r_{OH2}} - \frac{(x_O - x_{H1})}{r^2_{OH1}}\cos\theta_{HOH} - \frac{(x_O - x_{H2})}{r^2_{OH2}}\cos\theta_{HOH}] \tag{2.4.4}$$

Entonces la derivada de $V(r_{HOH})$ queda como la suma de la fuerzas $F_{x_O}(r_{HOH}) + F_{x_{H1}}(r_{HOH}) + F_{x_{H2}}(r_{HOH})$:

$$\begin{aligned} F_{x_O}(r_{HOH}) &= -k_r(r_{OH1} - r^0_{OH1})\frac{x_O - x_{H1}}{r_{OH1}} - k_r(r_{OH2} - r^0_{OH2})\frac{x_O - x_{H1}}{r_{OH2}} \\ &+ k_\theta \frac{\theta_{HOH} - \theta^0_{HOH}}{\sqrt{1-\cos^2\theta_{HOH}}}[\frac{(x_O - x_{H1}) + (x_O - x_{H1})}{r_{OH1}r_{OH2}} - \frac{(x_O - x_{H1})}{r^2_{OH1}}\cos\theta_{HOH} - \frac{(x_O - x_{H2})}{r^2_{OH2}}\cos\theta_{HOH}] \end{aligned} \tag{2.4.5}$$

$$F_{x_{H1}}(r_{HOH}) = k_r(r_{OH1} - r^0_{OH1})\frac{x_O - x_{H1}}{r_{OH1}} + k_\theta \frac{\theta_{HOH} - \theta^0_{HOH}}{\sqrt{1-\cos^2\theta_{HOH}}}[-\frac{(x_O - x_{H2})}{r_{OH1}r_{OH2}} + \frac{(x_O - x_{H1})}{r^2_{OH1}}\cos\theta_{HOH}] \tag{2.4.6}$$

$$F_{x_{H2}}(r_{HOH}) = k_r(r_{OH2} - r^0_{OH2})\frac{x_O - x_{H2}}{r_{OH2}} + k_\theta \frac{\theta_{HOH} - \theta^0_{HOH}}{\sqrt{1-\cos^2\theta_{HOH}}}[-\frac{(x_O - x_{H1})}{r_{OH1}r_{OH2}} + \frac{(x_O - x_{H2})}{r^2_{OH2}}\cos\theta_{HOH}] \tag{2.4.7}$$

El cómputo de esta fuerza para el sistema de n moléculas de agua se escala linealmente con el número de moléculas de agua de ahí que no se requiera cómputo en paralelo para las fuerzas internas de las moléculas de agua. Con esta sección se da por terminada la descripción de las fuerzas moleculares del agua.

# Capítulo 3

# Redes Neuronales Artificiales

Los primeros científicos en presentar un sistema artificial con la capacidad de imitar el aprendizaje humano fueron el psiquiatra Warren McCulloch y el matemático Walter Pitts en 1943. El modelo que presentan aunque rústico da origen a las primeras *redes neuronales artificiales (RNA)* (Isasi Viñuela, 2004 [2]).

Para la creación de una neurona artificial analicemos primero su contraparte biológica. Una neurona es la unidad celular fundamental del sistema nervioso humano, la cual, se compone de un cuerpo, una sinapsis, una dendrita y un axón. Mediante este conjunto de elementos la neurona recibe y crea señales que dan origen a un estímulo físico-químico. Por medio del sistema anterior una neurona tiene la facultad de trabajar con otras para generar una estructura única, la cual, es capaz de almacenar nuestros recuerdos, pensamientos o aprendizajes como impulsos eléctricos.

## 3.1. Principios de una red neuronal artificial

Una neurona artificial entonces copia la estructura neuronal humana mediante un modelo matemático, con el fin de imitar su funcionamiento. El cuerpo de la neurona es representado por una suma, $\sum$, la sinapsis mediante pesos, w, las dendritas son los estímulos de entrada, x, la respuesta a los estímulos de entrada z y, finalmente, un componente sin equivalente biológico el cual refuerza el estímulo de entrada, b.

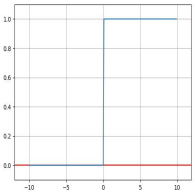
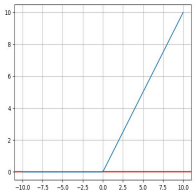
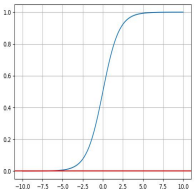
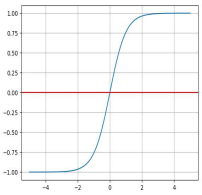
$$z = \sum_{i=0}^{n} w_i x_i + b \tag{3.1.1}$$

El modelo anterior si es entrenado con un numero determinado de iteraciones este obtendrá la capacidad de guardar información por medio de la actualización de sus pesos sinápticos $w$, sin embargo aún es incapaz de diferenciar entre las características únicas de algún conjunto de datos y por ende no puede generar una clasificación de los mismos. Para este fin se emplean las *funciones de activación* las cuales, tienen la facultad de limitar el valor del dominio de la función respecto a un umbral dado, así las salidas generadas serán parte de una serie de conjuntos con una división evidente entre ellos. Una característica muy valorada de estas funciones es la no linealidad, la cual, responde a la realidad de los

impulsos de entrada que tienen una configuración de patrones no rectilíneos, de ahí que sean ampliamente utilizadas (I. Foodfellow, 2016 [8]).

Generalmente se emplean las siguiente funciones de activación:

Cuadro 3.1: Funciones de activación

| Nombre | Ecuación | Grafica |
|---|---|---|
| Binaria | $\phi(z) = \begin{cases} 0, z < 0 \\ 1, z \geq 0 \end{cases}$ |  |
| ReLu | $\phi(z) = max\{0, z\} = \begin{cases} 0, z \leq 0 \\ z, z > 0 \end{cases}$ |  |
| Sigmoide | $\phi(z) = \dfrac{1}{1 + e^{-z}}$ |  |
| Tangente hiperbólica | $\phi(z) = tanh(z) = \dfrac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$ |  |

Si aplicamos cualquier función de activación al modelo 3.1.1 tendremos una función $f$ en términos de $z$, esta $f(z)$ da origen al llamado *perceptrón* como el que se muestra el la ilustración 3.1.1

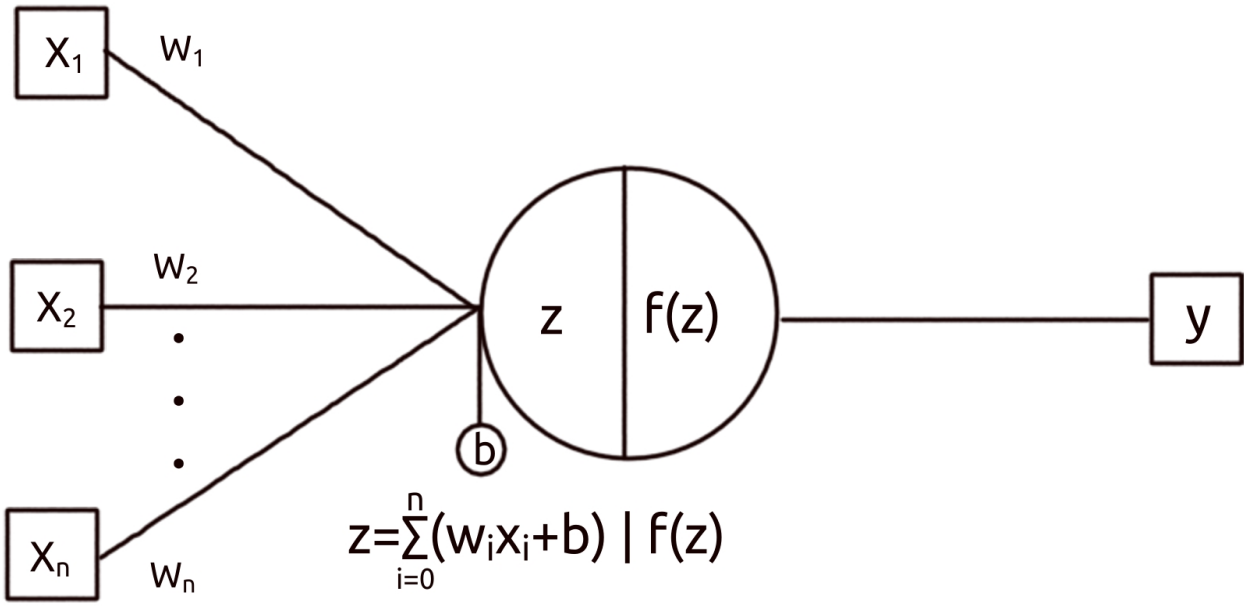



Figura 3.1.1: Representación gráfica de un perceptrón

A manera de ejemplo usemos la función sigmoidea para generar un perceptrón en forma:

$$f(z) = (1 + e^{-z})^{-1} \rightarrow f(z) = [1 + \exp(-(\sum_{i=0}^{n} w_i x_i + b))]^{-1} \tag{3.1.2}$$

Para llegar aun resultado idóneo en el entrenamiento del perceptrón es necesario dar valores iniciales aleatorios a nuestros pesos $w$ en la primera iteración de nuestro modelo. El ajuste para llegar a la respuesta esperada $z$ se generará con la iteración de la función $F(z)$.

A manera de ejemplo probemos este modelo de perceptrón para predecir la compuerta lógica OR, mediante una función de activación tipo escalón 3.1, y con unos pesos iniciales en cero:

Cuadro 3.2: Valores iniciales

| $x_1, x_2$ | $w_1, w_2$ | refuerzo $b$ | Resultado esperado |
|---|---|---|---|
| $0, 0$ | $0, 0$ | $-\frac{1}{2}$ | $0$ |
| $0, 1$ | $0, 0$ | $-\frac{1}{2}$ | $1$ |
| $1, 0$ | $0, 0$ | $-\frac{1}{2}$ | $1$ |
| $1, 1$ | $0, 0$ | $-\frac{1}{2}$ | $1$ |

Sustituyendo estos valores en la ecuación 3.1.1 y aplicando la función de activación

$$\begin{aligned} &z = w_1 \cdot x_1 + w_2 \cdot x_2 + b \qquad \rightarrow \qquad f(z) \\ &(0)(0) + (0)(0) + (-\tfrac{1}{2}) = -\tfrac{1}{2} \rightarrow -\tfrac{1}{2} \leq 0 \qquad \therefore \qquad y \text{ no se activa} \\ &\text{y es la respuesta esperada} \end{aligned} \tag{3.1.3}$$

Ahora encontremos la respuesta de la entrada $0, 1$

$$(0)(0) + (0)(1) + (-\tfrac{1}{2}) = -\tfrac{1}{2} \rightarrow -\tfrac{1}{2} \geq 1 \qquad \therefore \qquad y \text{ no se activa}$$

por lo debemos de actualizar algun peso para que se consiga el valor superior o igual a 1

$$(0)(0) + (1)(1) + (-\tfrac{1}{2}) = \tfrac{1}{2} \rightarrow \tfrac{1}{2} \geq 1 \qquad \therefore \qquad y \text{ no se activa}$$

nuevamente actualicemos los pesos

$$(0)(0) + (2)(1) + (-\tfrac{1}{2}) = 1{,}5 \rightarrow 1{,}5 \geq 1 \qquad \therefore \qquad y \text{ se activa} \tag{3.1.4}$$

hagamos lo mismo para las demás variables $1, 0$ y $1, 1$ obviando el proceso de iteración

$$(2)(1) + (2)(0) + (-\tfrac{1}{2}) = 1{,}5 \rightarrow 1{,}5 \geq 1 \qquad \therefore \qquad y \text{ se activa} \tag{3.1.5}$$

$$(2)(1) + (2)(1) + (-\tfrac{1}{2}) = 3{,}5 \rightarrow 3{,}5 \geq 1 \qquad \therefore \qquad y \text{ se activa} \tag{3.1.6}$$

finalmente usando estos pesos en la ecuación 3.1.3 y la ecuación 3.1.4

$$\begin{aligned} &(2)(0) + (2)(0) + (-\tfrac{1}{2}) = -\tfrac{1}{2} \rightarrow -\tfrac{1}{2} \leq 0 \qquad \therefore \qquad y \text{ no se activa} \\ &(2)(0) + (2)(1) + (-\tfrac{1}{2}) = 1{,}5 \rightarrow 1{,}5 \geq 1 \qquad \therefore \qquad y \text{ se activa} \end{aligned} \tag{3.1.7}$$

Así los valores finales quedan como:

Cuadro 3.3: Valores finales

| $x_1, x_2$ | $w_1, w_2$ | refuerzo $b$ | f(z)=y |
|---|---|---|---|
| $0, 0$ | $2, 2$ | $-\frac{1}{2}$ | $0$ |
| $0, 1$ | $2, 2$ | $-\frac{1}{2}$ | $1$ |
| $1, 0$ | $2, 2$ | $-\frac{1}{2}$ | $1$ |
| $1, 1$ | $2, 2$ | $-\frac{1}{2}$ | $1$ |

Estos pesos funcionan por lo tanto, se conservan. Notar que f(z)=y, representa la separación de nuestros datos. Por ende la clasificación de nuestra compuerta lógica OR.

## 3.2. Redes neuronales monocapa

Una *capa de procesamiento* se crea a partir del encadenamiento de múltiples perceptrón es mediante un único conjunto de entradas, donde cada perceptrón aprenderá una característica única respecto a nuestros datos, así generara una serie de respuestas de salida que dividirá nuestros datos. Generalmente se emplea una misma función de activación para toda la capa mediante la cual busca generar practicidad a la hora de implementar el modelo. De esta forma la única variación se encuentra en los pesos iniciales.

En ocasiones nos encontramos con un problema que puede tener múltiples soluciones, o bien múltiples clasificaciones. En estos casos un único perceptrón resulta incapaz de resolver el problema de clasificación. Para dar solución a este contratiempo se crearon las *redes neuronales mono capa* que como su nombre indica son redes que constan de una única capa de procesamiento (Jeffrey A, 1997 [10]). Teniendo el siguiente modelo matemático:

$$\begin{aligned} y_1 &= f(\textstyle\sum_{i=0}^{n} w_i^{(1)} x_i + b_1) \\ y_2 &= f(\textstyle\sum_{i=0}^{n} w_i^{(2)} x_i + b_2) \\ y_3 &= f(\textstyle\sum_{i=0}^{n} w_i^{(3)} x_i + b_3) \end{aligned} \tag{3.2.1}$$

Para entender mejor entendimiento de las capas ocultas resulta más sencillo analizarlo por medio de su representación matricial y su representación grafica:

$$\underset{\text{Pesos w}}{\begin{bmatrix} 0{,}47 & 0{,}03 & ... & 1{,}02 \\ 0{,}43 & 1{,}06 & ... & 2{,}78 \\ 2{,}43 & 2{,}90 & ... & 1{,}93 \\ ... & ... & ... & ... \\ 3{,}37 & 0{,}33 & ... & 0{,}33 \end{bmatrix}} \cdot \underset{\text{Entradas x}}{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ .. \\ x_n \end{bmatrix}} + \underset{\text{Refuerzo } b}{\begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ .. \\ b_n \end{bmatrix}} \rightarrow \underset{\text{Salidas z}}{\begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ .. \\ z_n \end{bmatrix}} \rightarrow \underset{\text{F. activación}}{\begin{bmatrix} y_1 = f(z_1) \\ y_2 = f(z_2) \\ y_3 = f(z_3) \\ .. \\ y_n = f(z_n) \end{bmatrix}}$$

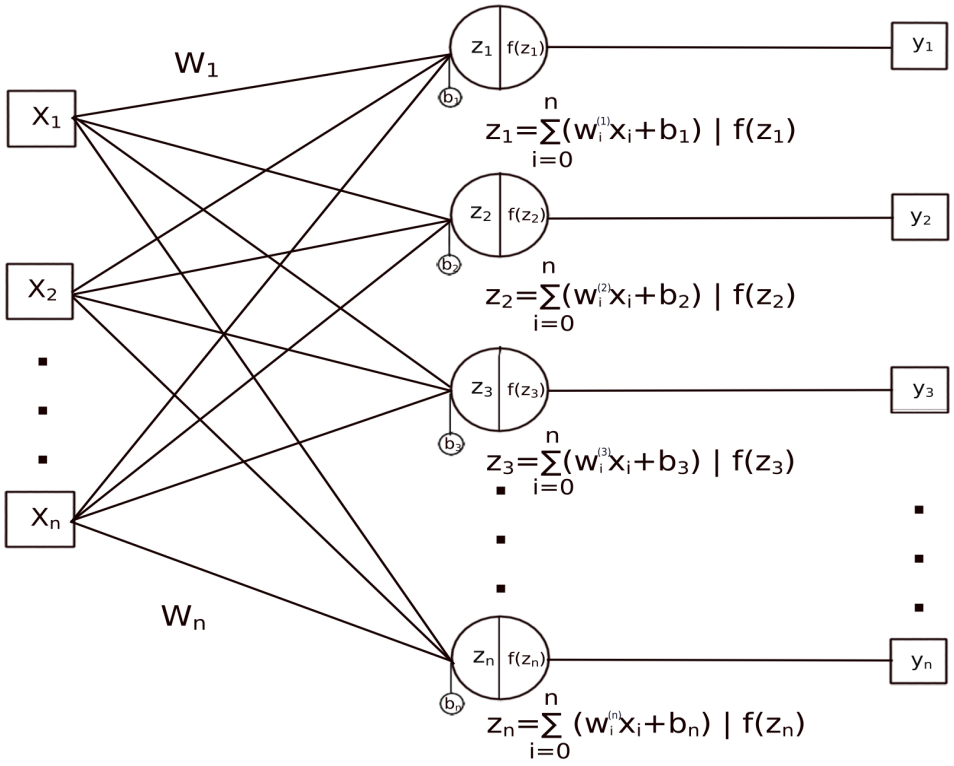



Figura 3.2.1: Representación gráfica de un perceptrón monocapa

Notar que no es necesario que el número de neuronas se mayor al numero de entradas. Lo que se busca en realidad es usar el menor numero de neuronas, para tener un mejor rendimiento a la hora de entrenar el modelo.

## 3.3. Redes neuronales multicapa

Dentro de un modelo monocapa tenemos la capacidad de guardar todas las características únicas de nuestros estímulos de entrada, no obstante, si nuestros datos comparten características tales como, pelaje, orejas, tamaño, color, etc. no podremos asegurar que mediante estos adjetivos encontremos una clasificación determinante. Como muestra de ello y a manera de ejemplo que haríamos ente un perro que parece gato o bien un gato que parece perro. La identificación de las particularidades en los modelos vistos hasta ahora resulta lo suficientemente complicado como para no implementarlos. Para conseguir una correcta clasificación con estas dificultades es necesario implementar un modelo *multicapa*.

La configuración de las *redes neuronales multicapa* parten de la base del modelo monocapa. Se crea una nueva cadena de neuronas de cualquier tamaño, estas neuronas tomaran las salidas $f(z_m)$ de la base para convertirse en nuevos impulsos de entrada $x_m$. Así cada nueva cadena generara lo que se le conoce como *capas ocultas* o capas de procesamiento. Las capas que vayamos agreguemos a la estructura base no necesariamente compartirán tamaño, función de activación ni refuerzo $b$, sino más bien, lo que se busca es crear una capa distinta a la anterior, para generar una separación característica. Los datos que comparten peculiaridades estarán guardados intrínsecamente dentro de estas capas ocultas. De esta forma, alguna neurona de nuestras capas ocultas tendrá la capacidad de discernir entre sí es perro, o bien un gato ya que, deberá pasar por una serie de funciones de activación antes de poder tomar una elección.

Es hasta este momento resulta evidente que la ultima capa de procesamiento $f(z_m^{(n)})$ no puede cambiar de tamaño respecto a la clasificación $y_n$ que deseemos hacer. El numero de neuronas de salida esta casado con nuestro sistema clasificatorio. En el caso de las redes monocapa no es evidente debido a que el número de neuronas de entrada es el mismo que las de salida.

Las ecuaciones para este tipo de redes esta dada de la siguiente manera:

$$\begin{aligned} y_1 &= f(\textstyle\sum_{i=0}^{n} w_i^{(1,n)} \cdot .... \cdot f(\sum_{i=0}^{n} w_i^{(1,2)} \cdot f(\sum_{i=0}^{n} w_i^{(1,1)} \cdot x_i + b_1^{(1)})_1^{(1)} + b_2^{(1)})_1^{(2)} + .... + b_n^{(1)})_1^{(n)} \\ y_2 &= f(\textstyle\sum_{i=0}^{n} w_i^{(2,n)} \cdot .... \cdot f(\sum_{i=0}^{n} w_i^{(2,2)} \cdot f(\sum_{i=0}^{n} w_i^{(2,1)} \cdot x_i + b_1^{(2)})_2^{(1)} + b_2^{(2)})_2^{(2)} + .... + b_n^{(2)})_2^{(n)} \end{aligned} \tag{3.3.1}$$

La diferencia entre las ecuaciones previas sera establecida por el refuerzo $b$ que elijamos, al igual que por los pesos $w$ que normalmente son aleatorios al inicio de nuestro entrenamiento.

Nuevamente observemos su representación matricial para obtener un mejor entendimiento de este modelo:

$$\begin{matrix} \text{w capa n} \end{matrix} \quad \begin{matrix} \text{Entradas x} \end{matrix} \quad \begin{matrix} \text{Refuerzo } b \end{matrix} \quad \begin{matrix} \text{Salidas z} \end{matrix} \quad \begin{matrix} \text{F. activación} \end{matrix}$$

$$\begin{bmatrix} 0{,}37 & 0{,}73 & ... & 0{,}02 \\ 1{,}03 & 1{,}56 & ... & 0{,}78 \\ 1{,}43 & 2{,}93 & ... & 0{,}93 \\ ... & ... & ... & ... \\ 1{,}37 & 2{,}33 & ... & 0{,}22 \end{bmatrix} \cdot \begin{bmatrix} f(z_1^{(n)}) \\ f(z_2^{(n)}) \\ f(z_3^{(n)}) \\ .. \\ f(z_m^{(n)}) \end{bmatrix} + \begin{bmatrix} b_1^{(n)} \\ b_2^{(n)} \\ b_3^{(n)} \\ .. \\ b_m^{(n)} \end{bmatrix} \rightarrow \begin{bmatrix} z_1^{(n+1)} \\ z_2^{(n+1)} \\ z_3^{(n+1)} \\ .. \\ z_m^{(n+1)} \end{bmatrix} \rightarrow \begin{bmatrix} y_1^{(n)} = f(z_1^{(n+1)}) \\ y_2^{(n)} = f(z_2^{(n+1)}) \\ y_3^{(n)} = f(z_3^{(n+1)}) \\ .. \\ y_m^{(n)} = f(z_m^{(n+1)}) \end{bmatrix}$$

Un perceptrón multicapa es una función de la forma $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$. También representado como n-L-m donde n representan las entradas, L las capas ocultas y m las salidas. De manera particular puede existir una RNA multicapa con una sola neurona por capa de procesamiento, si ese el caso se tendría la siguiente ecuación:

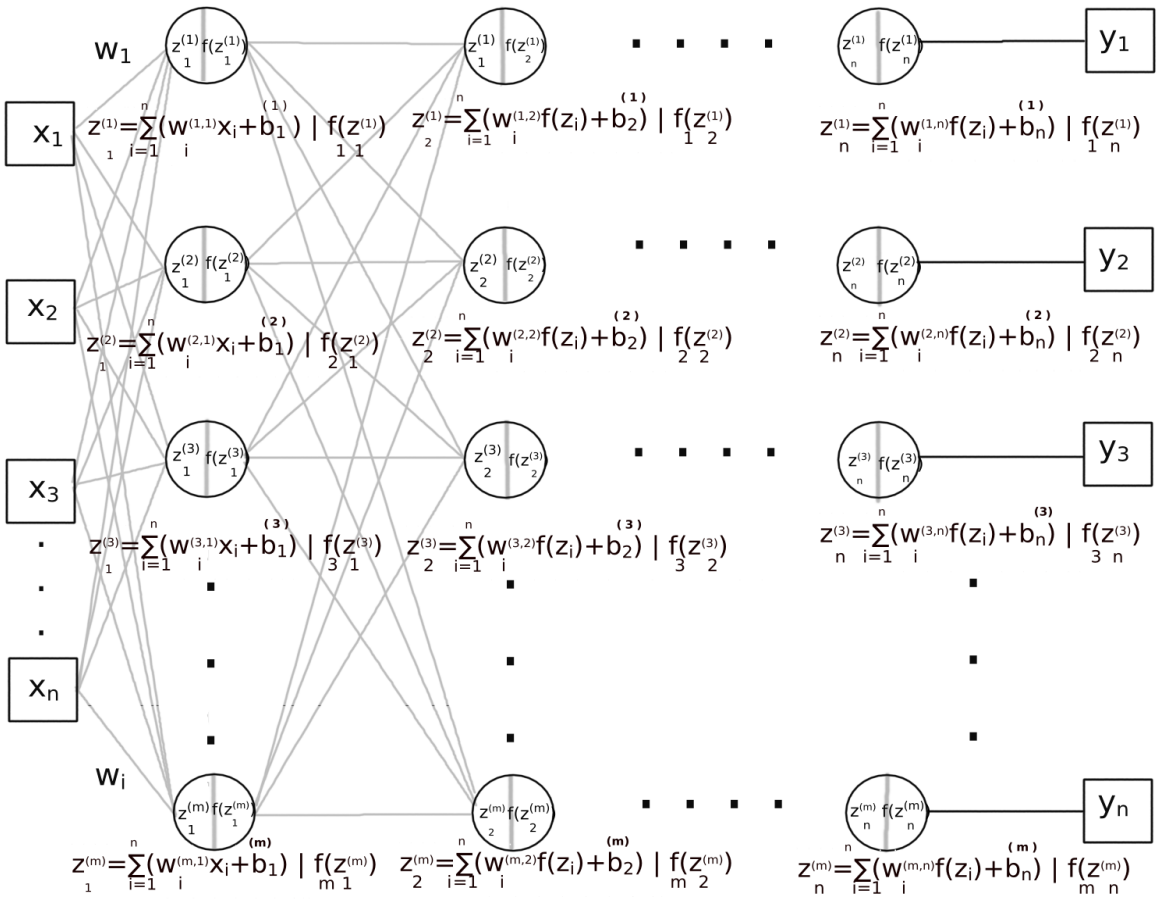



Figura 3.3.1: Representación gráfica de un perceptrón multicapa

$$f(z_t) = f(f_L(f_{L-1}(....f_1(z)))) \tag{3.3.2}$$

## 3.4. Algoritmo de retro-propagación

Si un perceptrón tiene un error ya sea que utilizo un peso demasiado grande o por el contrario demasiado pequeño, este se propagara por la red hasta llegar al resultado final. Para poder corregir este error con cualquiera de las estructuras vistas en las secciones anteriores se deben emplear múltiples iteraciones esperando que en alguna de ellas se encuentre este error y se corrija. Una forma más eficiente para dar solución a este problema es emplear la *retro propagación* la cual, emplea la estructura de las RNA multicapa para realizar el calculo del error cuadrático medio del resultado $y_n$ final, después propagar este cálculo a las neuronas anteriores, de esta forma la neurona con menor contribución a la respuesta total sera encontrada y actualizada sin necesidad de generar una nueva iteración. Para poder recorrer una RNA multicapa de forma inversa es necesario realizar la derivada de todos los componentes de cada una de las neuronas (Garcia Cabello, 2002 [3]).

En términos de funciones lo que estamos efectuando es la aplicación del gradiente descendente para encontrar el mínimo global por lo que, se deben cumplir los siguientes teoremas:

**Teorema 1**: Sea $f : \mathbb{R}^n \to \mathbb{R}$ una función diferenciable en el vecindario de algún punto $w = (w_1, ...., w_i)$. Entonces, el gradiente de $f$ en $w = (w_1, ...., w_i)$ denotado $\bigtriangledown f(w_1, ...., w_i)$.

1. representa la pendiente de la recta tangente a la función $f$ en el punto w.

2. Apunta en la dirección en la que la función $f$ crece más rápidamente. Por lo tanto, $-\bigtriangledown F$ indica la dirección de disminución más rápida.

3. Es ortogonal a las superficies de nivel (generalización del concepto de curva de nivel para una función de dos variables) de $f$, es decir, las de la forma $f(w_1, ...., w_i) = k$ para una constante $k$

**Teorema 2**: Sea $f : \mathbb{R}^n \to \mathbb{R}$ una función diferenciable. Existe un mínimo local que se puede alcanzar mediante la actualización iterativa de acuerdo con el sistema.

La idea de este método consta en encontrar los valores mínimos de una función dada $F(w_n)$ para ello, derivamos cada una de las funciones que se encuentran como salida a nuestra red neuronal.

$$\bigtriangledown f(w_1, ..., w_i) = (\partial f(w)/\partial w_1, ...., \partial f(w)/\partial w_i) \tag{3.4.1}$$

Realizando este proceso mediante la regla de los cuatro pasos

$$\begin{aligned} &\lim_{h\to 0}[f(w+h) - f(w)]/h \\ \approx &(\lim_{h\to 0}[f(w_1+h) - f(w_1)]/h, ...., \lim_{h\to 0}[f(w_i+h) - f(w_i)]/h) \end{aligned} \tag{3.4.2}$$

Despegando la diferencia del caso general $i$ obtenemos

$$\begin{aligned} \bigtriangledown f(w_i) &= \lim_{h\to 0}[f(w_i+h) - f(w_i)]/h \\ f(w_i+h) &\approx f(w_i) + h \bigtriangledown f(w_i) \end{aligned} \tag{3.4.3}$$

Si consideramos h como un decremento en dirección opuesta a la red neuronal como $h = -\epsilon \bigtriangledown f(w_n)$, donde épsilon es no negativo y es una variable similar a la taza de aprendizaje. Aplicando esto a la ecuación 3.4.3 entonces tenemos:

$$f(w_i - \epsilon \bigtriangledown f(w_i)) \approx f(w_i) - \epsilon \bigtriangledown f(w_i) \bigtriangledown f(w_i) \approx f(w_i) - \epsilon(\bigtriangledown f(w_i))^2 \leq f(w_i) \tag{3.4.4}$$

por lo tanto la función que actualiza el valor mínimo donde ya aplicamos la derivada es la siguiente:

$$w_i^{(n+1,m)} = w_i^{(n,m)} - \epsilon \bigtriangledown f(w_i^{(n,m)}) \tag{3.4.5}$$

Si aplicamos este procedimiento al refuerzo $b$ en vez de los pesos $w$ obtendríamos

$$b_i^{(n+1,m)} = b_i^{(n,m)} - \epsilon \bigtriangledown f(b_i^{(n,m)}) \tag{3.4.6}$$

Resolviendo la ecuación 3.4.5 respecto a $w$ empleando la regla de la cadena y siguiendo el procedimiento del articulo de Kiprono Elijah [4].

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u}\frac{\partial u}{\partial x} \tag{3.4.7}$$

tenemos:

$$\frac{\partial f}{\partial w_i^{(n,m)}} = \frac{\partial f}{\partial f(z_m^{(n)})} \cdot \frac{\partial f(z_m^{(n)})}{\partial z_m^{(n)}} \cdot \frac{\partial z_m^{(n)}}{\partial w_i^{(n,m)}} \tag{3.4.8}$$

Obtengamos primero la derivada de z respecto a w

$$\begin{aligned} \frac{\partial z}{\partial w_i^{(n,m)}} &= \frac{\partial}{\partial w_i^{(n,m)}}(f^{(1)} \cdot w_1^{(1,1)} + f^{(2)} \cdot w_2^{(1,2)} + .... + f^{(n)} \cdot w_i^{(n,m)}) \\ \frac{\partial z}{\partial w_i^{(n,m)}} &= f^{(n)} \cdot 1 \end{aligned} \tag{3.4.9}$$

siendo que $f^{(n)}$ es la función de activación anterior a la capa de salida y $w^{(n,m)}$ los pesos de la capa que deseamos actualizar

Ahora derivemos la función de activación respecto de z, utilizando como ejemplo a la sigmoidea:

$$\begin{aligned} \frac{\partial f(z_m^{(n)})}{\partial z_m^{(n)}} &= \frac{\partial}{\partial z_m^{(n)}}(f(z_m^{(n)})) = \frac{\partial}{\partial z_m^{(n)}}[1/(1+\epsilon^{-z_m^{(n)}})] \\ \frac{\partial f(z_m^{(n)})}{\partial z_m^{(n)}} &= f(z_m^{(n)})(1 - f(z_m^{(n)})) \end{aligned} \tag{3.4.10}$$

Ahora la derivada de la función de perdida:

$$\begin{aligned} \frac{\partial f}{\partial f(z_m^{(n)})} &= \frac{\partial}{\partial f(z_m^{(n)})}(-[t \cdot ln \cdot f(z_m^{(n)}) + (1-t)ln(1 - f(z_m^{(n)}))]) \\ \frac{\partial f}{\partial f(z_m^{(n)})} &= \frac{-t}{f(z_m^{(n)})} + \frac{1-t}{1-f(z_m^{(n)})} \end{aligned} \tag{3.4.11}$$

sustituyendo todas las derivadas en la principal 3.4.8 tenemos:

$$\begin{aligned} &\frac{\partial f}{\partial w_i^{(n,m)}} = \frac{-t}{f(z_m^{(n)})} + \frac{1-t}{1-f(z_m^{(n)})} \cdot f(z_m^{(n)})(1 - f(z_m^{(n)})) \cdot f^{(n)} \\ &= \frac{-t}{f(z_m^{(n)})} \cdot f(z_m^{(n)})(1 - f(z_m^{(n)}) + \frac{1-t}{1-f(z_m^{(n)})} \cdot f(z_m^{(n)})(1 - f(z_m^{(n)}) \cdot f^{(n)} \\ &= -t + f(z_m^{(n)})t + f(z_m^{(n)}) - f(z_m^{(n)})t \cdot f^{(n)} \\ &= (f(z_m^{(n)}) - t) \cdot f^{(n)} \end{aligned} \tag{3.4.12}$$

Si aplicamos este mismo proceso para el refuerzo $b$ con el que obtenemos el sesgo entonces tenemos:

$$\frac{\partial f}{\partial b_n} = (f(z_m^{(n)}) - t) \cdot 1 \tag{3.4.13}$$

Finalmente resolviendo las ecuaciones 3.4.5 y 3.4.6 donde ya estaremos actualizando los pesos y el refuerzo $b$ entonces tendríamos:

$$\begin{aligned} w_i^{(n,m)} &= w_i^{(n,m)} - \epsilon \frac{\partial f}{\partial w_i^{(n,m)}} \\ &= w_i^{(n,m)} - \epsilon \cdot (f(z_m^{(n)}) - t) \cdot f^{(n)} \\ b_n &= b_n - \epsilon \frac{\partial f}{\partial b_n} \\ &= b_n - \epsilon \cdot (f(z_m^{(n)}) - t) \cdot 1 \end{aligned} \tag{3.4.14}$$

Donde $t$ normalmente tiene el valor de 1 tanto para los pesos como para los refuerzos b. Este valor afirma si es o no es una contribución adecuada.

Para encontrar la contribución menos favorable se deberá realizar una nueva derivada. Esto nos llevará a recorrer al modelo de forma inversa. Este recorrido inverso se realiza de la siguiente manera:

$$\frac{\partial f}{\partial w_i^{(n,m-1)}} = \frac{\partial f}{\partial f(z_m^{(n)})} \cdot \frac{\partial f(z_m^{(n)})}{\partial z_m^{(n)}} \cdot \frac{\partial z_m^{(n)}}{\partial f(z_m^{(n-1)})} \cdot \frac{\partial f(z^{(n-1)})}{\partial z_m^{(n-1)}} \cdot \frac{\partial z_m^{(n-1)}}{\partial w_i^{(n,m-1)}} \tag{3.4.15}$$

El número de derivadas efectuadas dependerá del número de neuronas y capas ocultas que se estén empleando en nuestro modelo. Una vez que se encontró la neurona dentro de estas capas con la contribución menos favorable, se procederá a actualizar su peso y su refuerzo $b$. Al efectuarse una nueva época en el modelo, el procedimiento de retropropagación deberá de realizarse nuevamente.

# Capítulo 4

# Decodificando las interacciones atómicas

De manera particular se analizó un recipiente el cual contiene 10496 moléculas de agua ( ver imagen 4.0.1) $O, H_1, H_2$, estas se encuentran sometidas a una temperatura y presión constantes con el fin de mantener su estado liquido. Cada elemento $O, H_1, H_2$ en el contenedor tiene las siguientes componentes $x, y, z$:

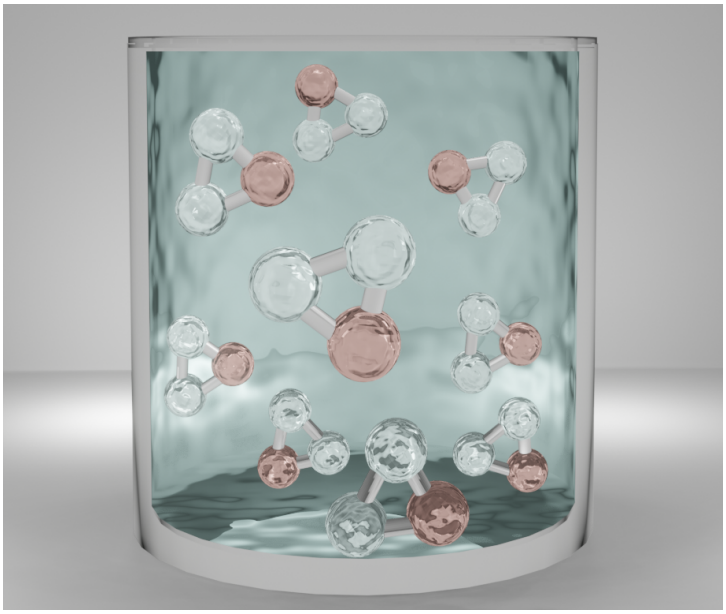
$$O_x^0, O_y^0, O_z^0, H_{1x}^0, H_{1y}^0, H_{1z}^0, H_{2x}^0, H_{2y}^0, H_{2z}^0$$

$$O_x^1, O_y^1, O_z^1, H_{1x}^1, H_{1y}^1, H_{1z}^1, H_{2x}^1, H_{2y}^1, H_{2z}^1 \tag{4.0.1}$$

$$O_x^{10496}, O_y^{10496}, O_z^{10496}, H_{1x}^{10496}, H_{1y}^{10496}, H_{1z}^{10496}, H_{2x}^{10496}, H_{2y}^{10496}, H_{2z}^{10496}$$

Donde el superíndice representa el identificador de cada molécula de agua.

Figura 4.0.1: Contenedor de agua



Como se vio en el capitulo dos para obtener las fuerzas de Coulomb y las fuerzas LJ se deberán considerar cada uno de los átomos, debido a que cada uno de estos percibe su entorno de una forma diferente. En nuestro caso los $10496 * 3 = 31488$ átomos de agua. La fuerza 1 $F_{O_x^1}$ se calculara mediante los 31487 átomos restantes. La fuerza 2 $F_{O_y^1}$ sera obtenida por los 31486 átomos restantes. Este comportamiento se describe con la siguiente ecuación:

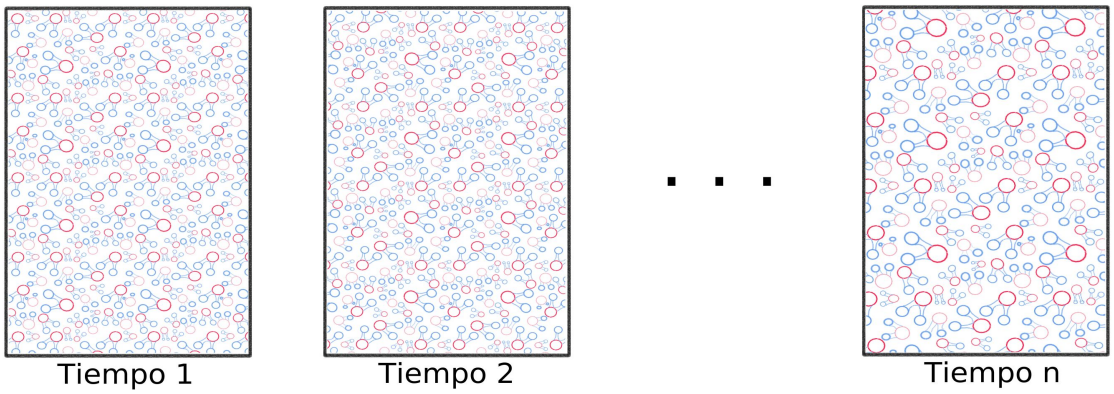
$$N_{interacciones} = \frac{n(n-1)}{2} \tag{4.0.2}$$

Donde $n$ es numero de átomos dentro del contenedor.

Debido a la fuerza interna entre los átomos, siempre existirá movimiento dentro nuestro contenedor por lo que, el análisis que generemos en un tiempo $t_0$ siempre sera diferente a un tiempo $t_1, t_2, ..., t_n$ (ver imagen 4.0.2).

Para los 31488 átomos de agua tendríamos 495731328 interacciones por calcular. Si agregamos las $n$ capturas de tiempo a nuestro análisis , esteremos ante un sistema cuyo comportamiento esta descrito por $N_{interacciones} \cdot N_{capturas}$.

Figura 4.0.2: Diferentes cuadros de tiempo $t_1, t_2, ..., t_n$

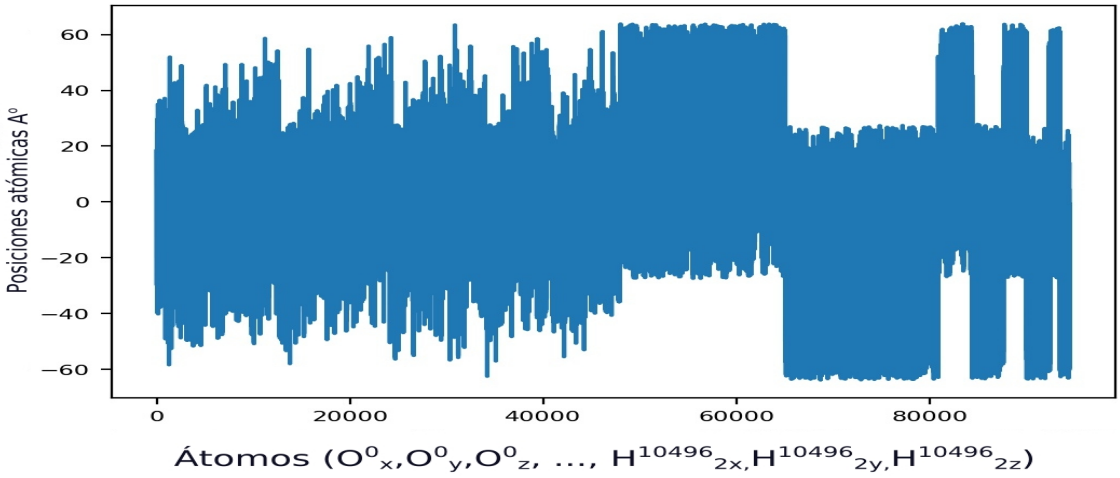



El número de cálculos a realizar tiene un comportamiento similar al de una función de $n^2$ por lo que, al aumentar considerablemente el número de átomos o prolongar las capturas de tiempo de los análisis, sera imposible realizar todos los cálculos requeridos. La imposibilidad esta directamente relacionada con la potencia computacional (ver apéndice 8.3), así como con el tiempo necesario para realizar el cálculo de cada una las interacciones atómicas. Una mejor alternativa es implementar una RNA, la cual mejora tanto en tiempo como en potencia al no calcular, si no, predecir estas fuerzas.

Antes de poder emplear una RNA primero es necesario encontrar una relación entre las posiciones iniciales $x, y, z$ de los 31488 átomos respecto a las fuerzas resultantes $Fx, Fy, Fz$ de cada molécula de agua es decir, decodificar la información intrínseca de los átomos de agua dentro del contenedor. Comencemos entonces por analizar el comportamiento de los

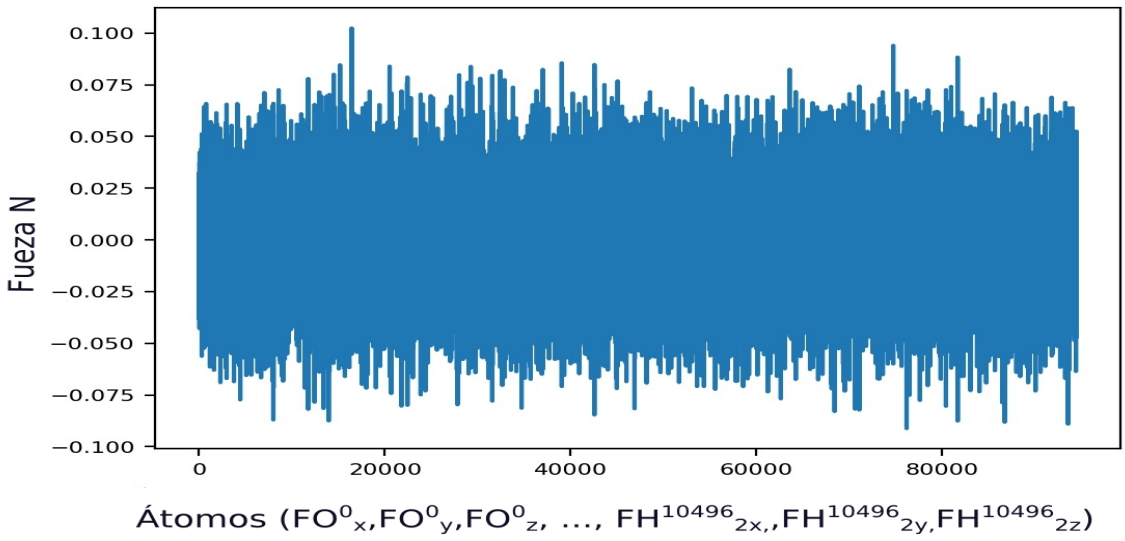
átomos, para lo cual, la siguiente gráfica 4.0.3 muestra su distribución espacial en el primer cuadro tiempo.

Figura 4.0.3: Distribución de las posiciones iniciales de los átomos a $t = 0$




A partir de estas posiciones iniciales y tras realizar los cálculos correspondientes se obtienen las siguientes distribución de las fuerzas totales $Fx, Fy, Fz = F_{Coulomb} + F_{LJ}$. Asimismo veamos su distribución espacial en la grafica 4.0.4

Figura 4.0.4: Distribución de las fuerzas resultantes a $t = 0$



Como podemos apreciar en las gráficas anteriores no existe ningún patrón evidente entre las posiciones iniciales y las fuerzas resultantes, por lo que, es necesario quitar aleatoriedad a los datos para así facilitar en entrenamiento de la RNA, así como asegurar una correcta predicción.

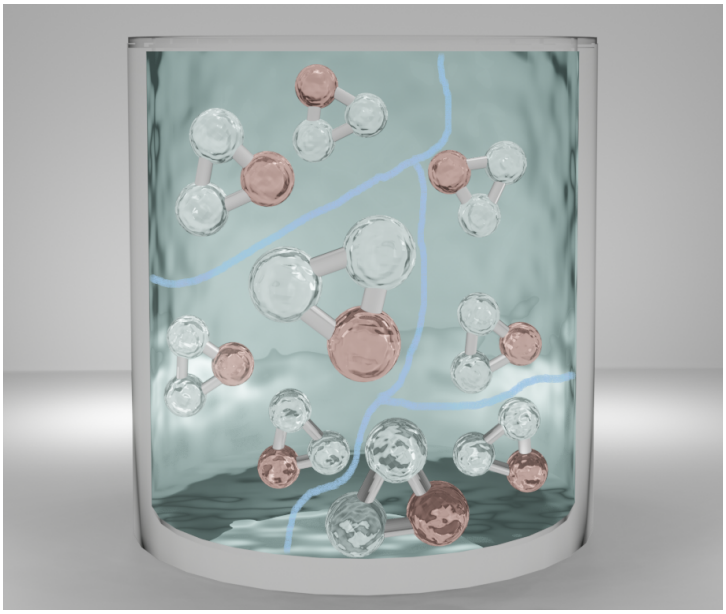
Una de las técnicas mas comunes para este tipo de problemas es la de dividir y conquistar, la cual, consta en separar un sistema en pequeños fragmentos tanto como sea posible, con el fin de simplificar el medio, de tal forma que la resolución de este se torne simple. La idea de aplicar este método es encontrar una serie de cúmulos de agua los cuales tengan la capacidad de describir todo el medio con un número limitado de moléculas como se representa en la imagen 4.0.5.

La técnica de dividir y conquistar en este caso se realizo por medio de seleccionar un átomo de oxigeno con el cual, se calcularon las distancias respecto a los demás oxígenos (ver ecuación 4.0.3). Estas distancias se ordenaron de menor a mayor con el fin de solo seleccionar las primeras distancias mas cercanas. Se tomaron a los oxígenos como referencia debido a que se buscaba obtener cúmulos de aguas completos $O - H - H$.

$$
\begin{aligned}
d_1(O_c, O^0) &= \sqrt{(O_x^0 - O_{x_c})^2 + (O_y^0 - O_{y_c})^2 + (O_z^0 - O_{z_c})^2} \\
d_2(O_c, O^1) &= \sqrt{(O_x^1 - O_{x_c})^2 + (O_y^1 - O_{y_c})^2 + (O_z^1 - O_{z_c})^2} \\
d_{10496}(O_c, O^{10496}) &= \sqrt{(O_x^{10496} - O_{x_c})^2 + (O_y^{10496} - O_{y_c})^2 + (O_z^{10496} - O_{z_c})^2}
\end{aligned}
\tag{4.0.3}
$$

Donde $O_c$ es el oxigeno seleccionado, $O^n$ el recorrido de todos los elementos del contenedor.

Figura 4.0.5: Partición del contenedor de agua donde, cada división simboliza un vecindario



Cada vecindario creado por cada átomo de oxigeno requiere mucho tiempo en *unidad central de proceso (CPU)* por lo que, se migro este mismo procedimiento en una *unidad de procesamiento de gráficos (GPU)* con el lenguaje de programación *Compute Unified Device Architecture (CUDA)*. La siguiente tabla muestra el tiempo necesario para calcular todos los vecindarios de 100 moléculas de agua en un cuadro de tiempo con 10496 oxígenos, utilizando CPU, multi-CPU mediante *Message Passing Interface* MPI, GPU, multi-GPU:

Cuadro 4.1: Tiempo de procesamiento por 10496 átomos de oxigeno

| Tecnologías | tiempo |
|---|---|
| $CPU - Python$ | $10_{\text{días}}$ |
| $Multi - CPU - MPI$ | $40_{segundos}$ |
| $GPU - CUDA$ | $16_{segundos}$ |
| $Multi - GPU - CUDA$ | $16_{segundos}$ |
| $GPU - NUMBA - Python$ | $20_{segundos}$ |

Para entender mejor como es que se llegaron a estos tiempos de procesamiento hagamos un pequeño paréntesis para hablar de la implementación con el programa con CUDA ya que, este fue el lenguaje con el que se obtuvieron los mejores resultados. Para utilizar CUDA, primero es necesario conocer el hardware del GPU con la que se esta trabajando. Aunque no todos los GPU son iguales todos comparten una configuración como la que se ve en el apéndice 8.2

Ya que se consulto la estructura del funcionamiento de cada uno de los componentes de una GPU, es posible programar un núcleo en CUDA que agrupe un conjunto de 300 átomos es decir 100 moléculas de agua en un vecindario, donde se utilizaron a los oxígenos para calcular todas las distancias. Considerar a los 100 vecinos como una hipótesis con la cual, suponemos que seremos capases de describir todas las fuerza que percibe un átomo respecto a todo el contenedor.

Para comenzar a utilizar los procesadores de nuestra GPU primeramente debemos conocer en que hilo de procesamiento nos encontramos. Como en CUDA cada procesador trabaja en paralelo y de manera asíncrona la forma de encontrar nuestro hilo es la siguiente:

$$O_{indice} = blockIdx.x * blockDim.x + threadIdx.x \tag{4.0.4}$$

Donde $O_{indice}$ es el hilo $1, 2, 3, ...n$ donde nos encontramos, blockIdx es identificador del bloque $1, 2, 3, ..., n$, blockDim es el tamaño de nuestro bloque CUDA que normalmente es de 256 casillas y threadIdx es identificador del hilo $1, 2, 3, ..., 256$.

Al obtener este identificador en el que nos encontramos es posible emplearlo como índice para seleccionar a un oxigeno como central. El oxigeno central sera utilizado como referencia

para encontrar un vecindario:

$$\begin{aligned} O_{x_c} &= atomos_{contenedor}[H_{indice} * 9 + 0] \\ O_{y_c} &= atomos_{contenedor}[H_{indice} * 9 + 1] \\ O_{z_c} &= atomos_{contenedor}[H_{indice} * 9 + 2] \end{aligned} \tag{4.0.5}$$

De esta forma si nos encontramos en el indice $1, 21, 13, ..., n$ también, nos encontraremos en el átomo central $1, 21, 13, ..., n$. Como solo queremos utilizar a los oxígenos como centrales se realiza una multiplicación por 9 ya que cada molécula esta compuesta por 9 posiciones (ver ecuación 4.0.1).

Calculando las distancias por medio de la ecuacion 4.0.3 nos queda la siguiente notación:

$$\begin{aligned} x_{temporal} &= (atomos_{contenedor}[i + 0] - O_{x_c})^2 \\ y_{temporal} &= (atomos_{contenedor}[i + 1] - O_{y_c})^2 \\ z_{temporal} &= (atomos_{contenedor}[i + 2] - O_{z_c})^2 \end{aligned} \tag{4.0.6}$$

$$Distancia = \sqrt{x_{temporal} + y_{temporal} + z_{temporal}}$$

Donde $i$ serán los saltos de 9 en 9 hasta los $31488 * 3 = 94464$ coordenadas del vector.

Dentro de un núcleo CUDA no es posible generar vectores temporales que afecten al vector principal $atomos_{contenedor}$, por lo que no es posible guardar todas las distancias en un vector temporal, estas sean de calcular conforme se haga un recorrido en vector principal. Para saber si esta distancia calculada es la menor de todas se utilizo la condición 4.0.7. Para hacer uso de esta condición en primera instancia el limite superior e inferior se les tendrá que asignar un valor considerablemente grande y considerablemente pequeño respectivamente. Los limites se actualizaran de tal forma que solo sea tomada la menor distancia. Cuando queramos encontrar una distancia diferente solo debemos configurar el limite inferior para que este sea igual a la distancia anterior, así no la volveremos a considerar.

$$Limite_{inferior} < Distancia \quad \&\& \quad Distancia <= Limite_{superior} \tag{4.0.7}$$

Un inconveniente de aplicar los limites de la condición previa es no poder detectar átomos que se encuentren a una misma distancia respecto al oxigeno central. Esta dificultad se resolvió volviendo a calcular las distancias, verificando si existen repeticiones pero, con un identificador diferente al ya encontrado.

Aunque se intento aplicar un método de ordenamiento mas eficiente como listas ligadas, listas doblemente ligadas, push y pop, etc. el lenguaje CUDA impido esta implementación.

Para utilizar al núcleo programado con las condiciones previas, antes de todo es necesario definir el número de de hilos y el número de bloques que queramos utilizar. El producto de multiplicar estas variables dará como resultado el numero de procesadores que utilizaremos

(ver ecuación 4.0.8). Como cada hilo ejecuta un núcleo y cada hilo se procesa en paralelo, el tiempo de procesamiento es mínimo (ver cuadro 4.0.5).

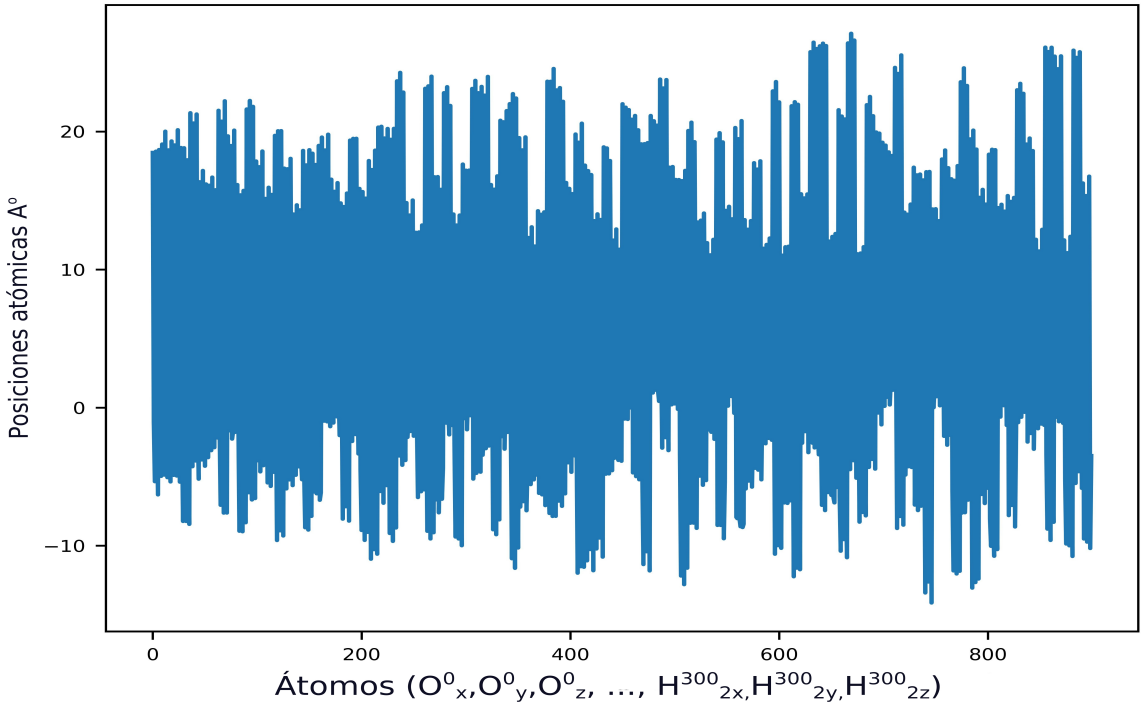
$$N_{procesadores} = N_{Bloques} * N_{Hilos} \tag{4.0.8}$$

En nuestro caso 128 bloques $*82$ hilos $= 10496$ nucleos CUDA para los 10496 átomos de oxigeno.

Aunque este código se implemento en un principio en $Fortran+CUDA$ (ver apéndice 8.1.2), también se realizo en $Python + CUDA$ donde la implementación del código se torna mas sencilla (ver apéndice 8.1.1).

Al aplicar el tratamiento anterior a las aguas se genero un aumento al número de datos por cien. Pese al incremento se consiguió un orden mas estable como se muestra en la grafica 4.0.6, donde las posiciones están contenidas entre los valores 0 y 10 Armstrong .

Figura 4.0.6: Distribución del primer vecindario respecto al oxigeno central en $t = 0$



## 4.1. Descripción uniforme de los vecindarios respecto a aun átomo de oxígeno central

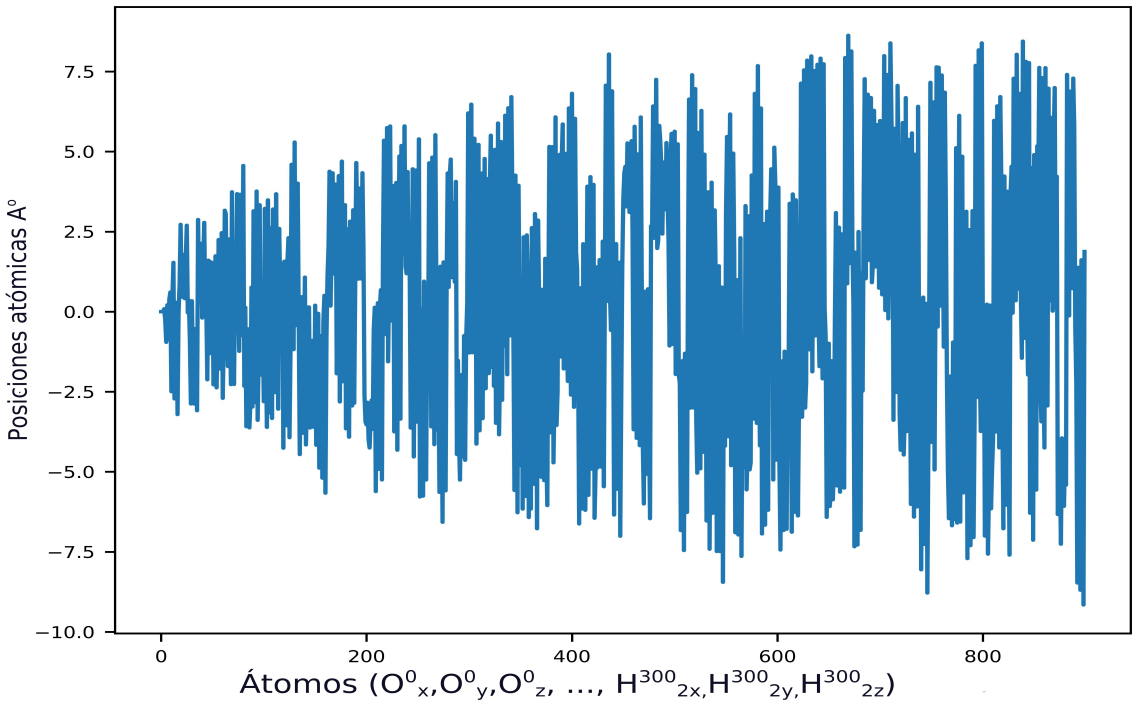
Consideremos ahora la forma individual de cada vecindario, si bien, la estructura es parecida, en ningún caso es igual. Esta forma característica aunque en un principio dependía del contenedor ahora no lo hace, lo cual, otorga la posibilidad de simplificar el cúmulo realizando un desplazamiento al origen (ver ecuación 4.1.1), sin perder su estructura característica. El código empleado en esta sección se encuentra en el apéndice 8.1.3

$$\begin{aligned} x_0 &= x_n - Ox_c \\ y_0 &= y_n - Oy_c \\ z_0 &= z_n - Oz_c \end{aligned} \tag{4.1.1}$$

Donde $O_{x_c}, O_{y_x}, O_{z_c}$ es la posición del oxigeno central, $x_n, y_n, z_n$ son las coordenadas de los demás átomos del vecindario y $x_0, y_0, z_0$ las coordenadas respecto al origen $0, 0, 0$.

La siguiente grafica 4.1.1 muestra el comportamiento de los datos después de aplicar el desplazamiento. Notemos ahora que, los datos presentan un ordenamiento similar a un hipérbola horizontal pero, unicamente sobre los átomos de oxigeno debido al procedimiento de los 100 mas cercanos. Aunque los hidrógenos generan ruido no alteran de ninguna manera esta nueva estructura característica de las aguas. A modo de comparación con la grafica 4.0.6 no pensemos que la forma del vecindario cambio si no que, simplemente se ordeno.

Figura 4.1.1: Distribución y posicionamiento al origen del primer vecindario en $t = 0$



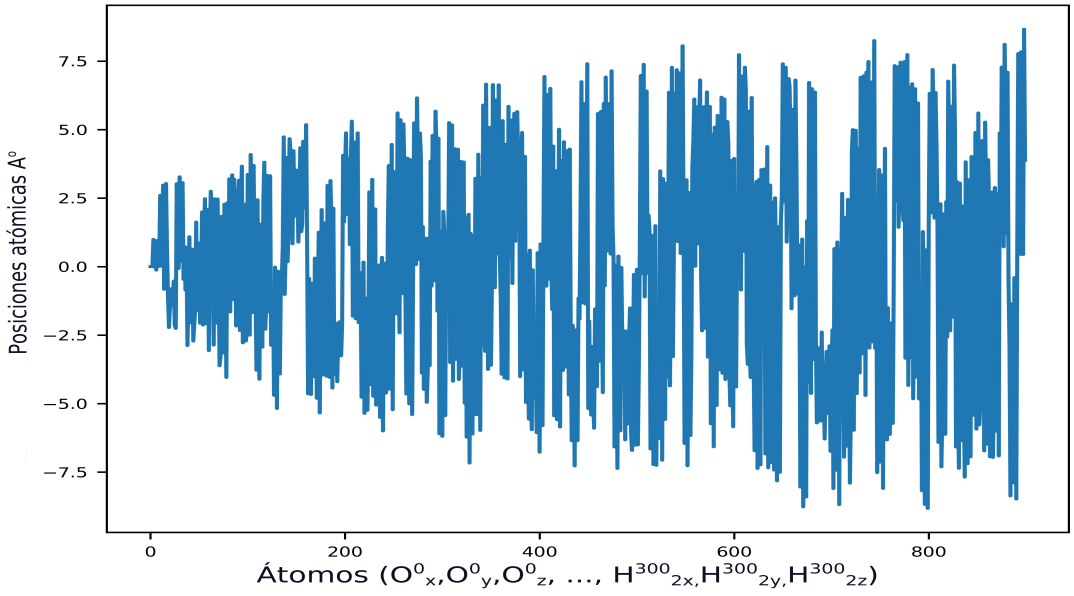
Para simplificar aun mas el sistema se efectuó una rotación sobre los ejes coordenados $y, z$, de modo que, los hidrógenos se encuentren sitúen sobre el plano $x, y$. Esta rotación genera que los primeros átomos tengan un valor igual a cero (ver resultados 4.1.2). Empleando las matrices de rotación 4.1.3 (Goldstein H, 2014 [5]), y considerando el octeto donde se encuentra el átomo se obtuvo una mejor distribución como se muestra en la imagen 4.1.2.

$$\begin{aligned} &O_{x_0^0} = 0, O_{y_0^0}, O_{z_0^0} \rightarrow \text{Desplazamiento al origen} \\ &H^0_{1y_0} = 0, H^0_{1z_0} = 0 \rightarrow \text{Rotacion sobre los planos } y, z \\ &H^0_{2z_0} = 0 \end{aligned} \tag{4.1.2}$$

De este modo la componentes $O_{z_0^0}, H^0_{2z_0} = 0$ en la primer molécula de agua en todos los vecindarios siempre serán nulas, los que en principio mejoraría la predicción al no depender de estas coordenadas.

$$\begin{aligned} ROT(x,y,\theta) &= \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} \begin{aligned} x' &= x\cos(\theta) + y\sin(\theta) \\ = y' &= -x\sin(\theta) + y\cos(\theta) \\ z' &= z \end{aligned} \\ ROT(x,z,\theta) &= \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} \begin{aligned} x' &= x\cos(\theta) + z\sin(\theta) \\ = y' &= y \\ z' &= -x\sin(\theta) + z\cos(\theta) \end{aligned} \\ ROT(y,z,\theta) &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) \\ 0 & -\sin(\theta) & \cos(\theta) \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} \begin{aligned} x' &= x \\ = y' &= y\cos(\theta) + z\sin(\theta) \\ z' &= -y\sin(\theta) + z\cos(\theta) \end{aligned} \end{aligned} \tag{4.1.3}$$

Figura 4.1.2: Distribución tras la rotación sobre los ejes $y, z$ del primer vecindario a $t = 0$



Si bien, la grafica anterior ya cuenta con un comportamiento relativamente ordenado al siempre tener un punto de origen en $0, 0, 0$, al siempre tener un incremento controlado de las distancias, aun no podremos decir que al entrenar una RNA con estos datos obtengamos una predicción correcta, por ello, se aplico un cambio de coordenadas cartesianas $x, y, z$, a un sistema de coordenadas de coordenadas polares $r, \theta, \phi$ (ver ecuaciones 4.1.4, 4.1.5 , 4.1.6)

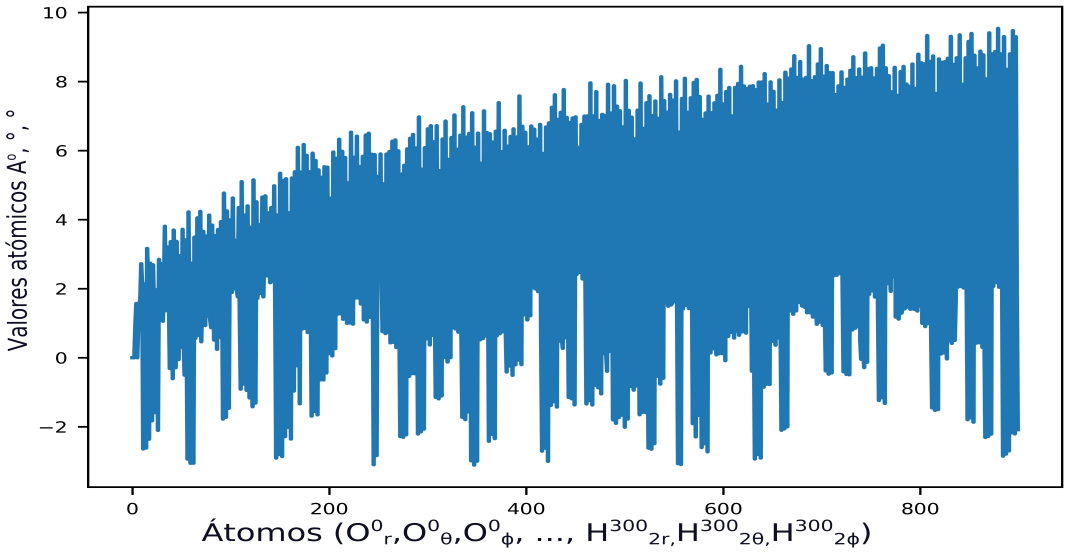
$$r = \sqrt{x^2 + y^2 + z^2} \tag{4.1.4}$$

$$\theta = \arccos(\frac{z}{\sqrt{x^2+y^2+z^2}}) = \arccos(\frac{z}{r}) = \begin{cases} \arctan\frac{\sqrt{x^2+y^2}}{z} & si \quad z > 0 \\ \pi + \arctan\frac{\sqrt{x^2+y^2}}{z} & si \quad z > 0 \\ +\frac{\pi}{2} & si \quad z = 0 \ _y \ x, y \neq 0 \\ Indefinido & si \quad x = y = z = 0 \end{cases} \tag{4.1.5}$$

$$\phi = \arccos(\frac{x}{\sqrt{x^2+y^2}}) = \begin{cases} \arctan(\frac{y}{x}) & si \quad x > 0 \\ \arctan(\frac{y}{x}) + \pi & si \quad x < 0 \ _y \ y \geq 0 \\ \arctan(\frac{y}{x}) - \pi & si \quad x < 0 \ _y \ y < 0 \\ +\frac{\pi}{2} & si \quad x = 0 \ _y \ y > 0 \\ -\frac{\pi}{2} & si \quad x = 0 \ _y \ y < 0 \\ Indefinido & si \quad x = 0 \ _y \ y = 0 \end{cases} \tag{4.1.6}$$

El cambio a este sistema de coordenadas polares genero la siguiente distribución.

Figura 4.1.3: Distribución tras la transformación $r, \theta, \phi$ del primer vecindario a $t = 0$




Salta a la vista que sean desplazado la mayor parte de los datos negativos y a diferencia de la gráfica 4.1.2 los datos ahora generan una aglomeración positiva, lo cual, nos habla de que los datos tienen muy poca variación entre si y por ende menor cantidad de ruido.

Dado que $r$ representa la distancia, si separamos esta componente de los datos anteriores (ver grafica 4.1.4) observaremos que acabamos de obtener un aproximado de las mismas distancias que calculamos al obtener los 100 vecinos mas cercanos (ver grafica 4.1.5).

Figura 4.1.4: Distribución en r del primer vecindario a $t = 0$

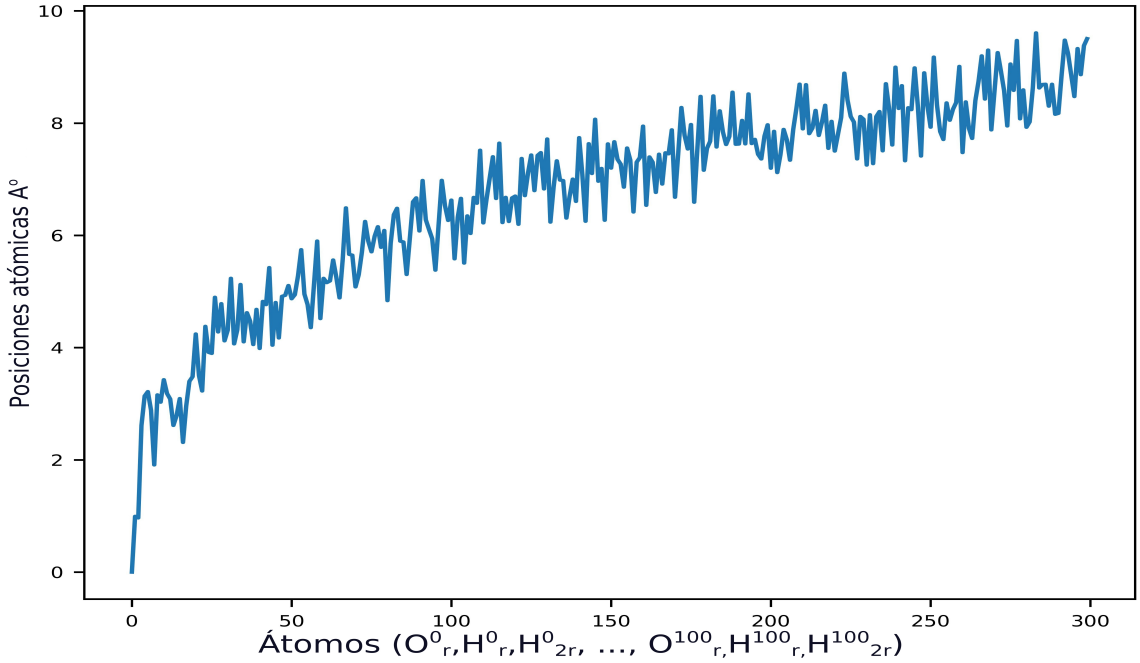
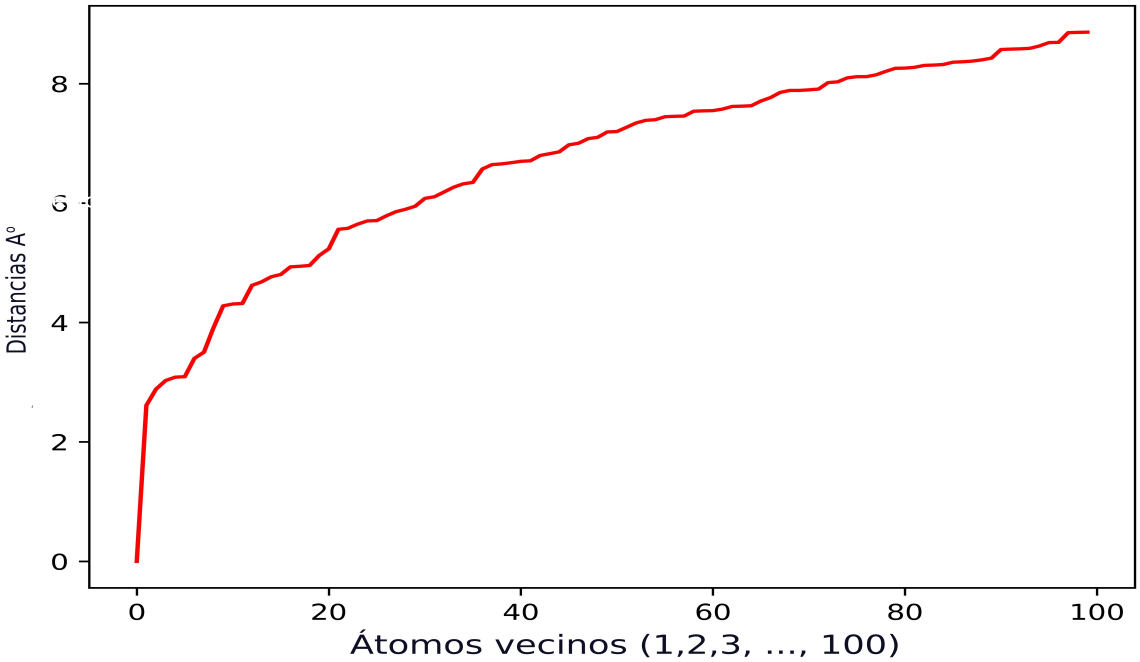



Figura 4.1.5: Distribución de las distancias del primer vecindario




Obtengamos entonces la distribución de la separación tanto de $\theta$ (ver grafica 4.1.6) como de $\phi$ (ver grafica 4.1.7).

Figura 4.1.6: Distribución en $\theta$ del primer vecindario a $t = 0$

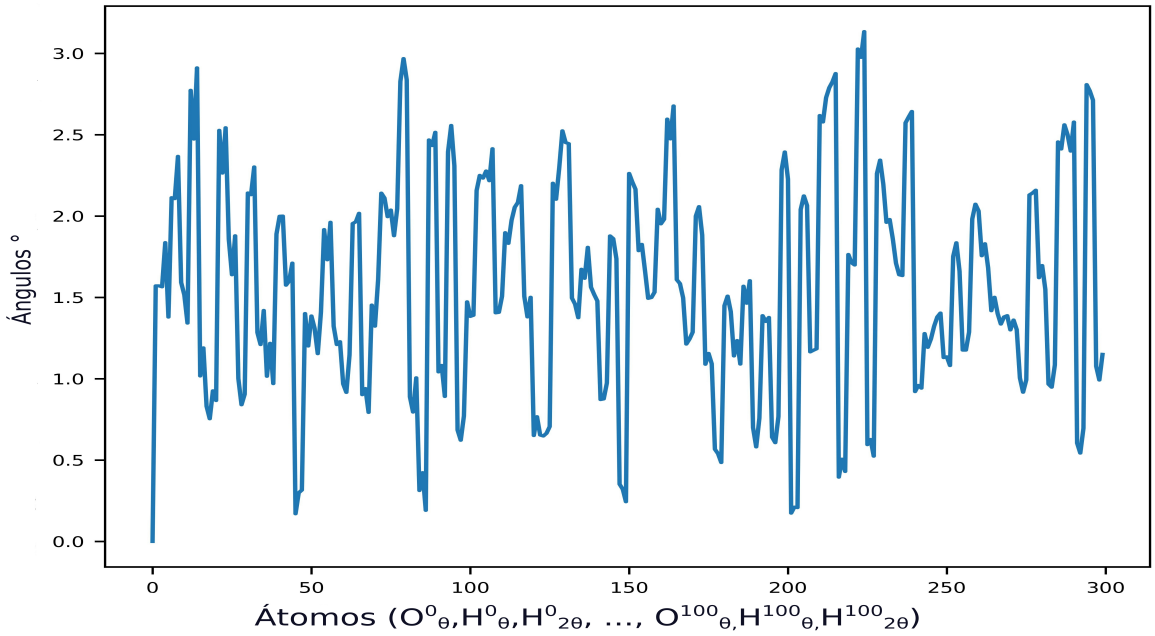
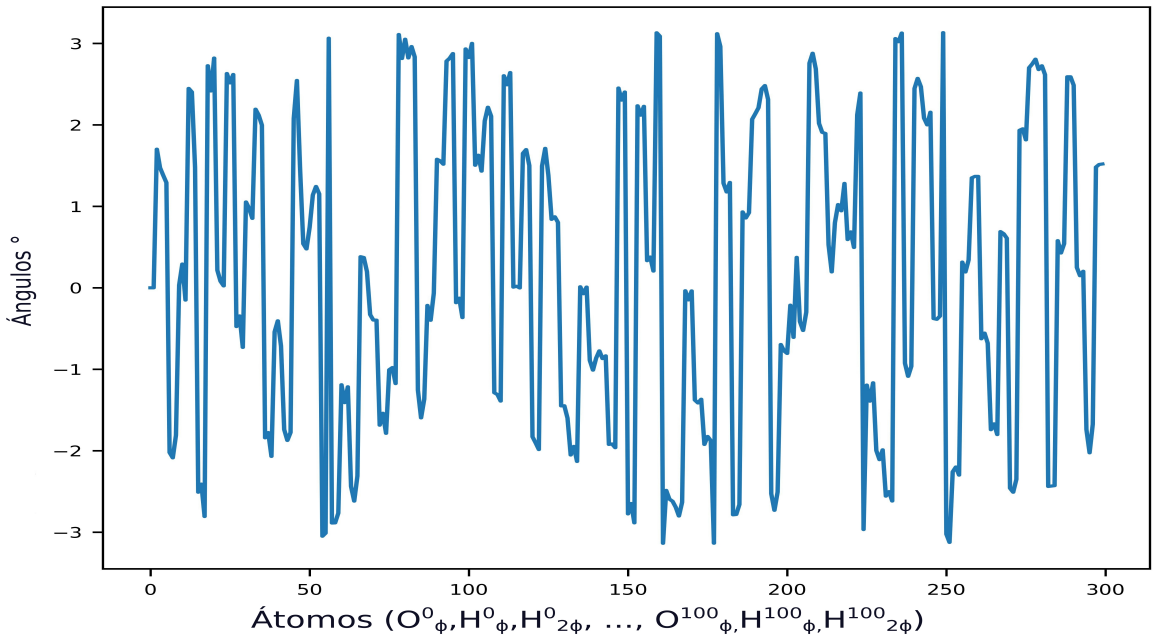



Figura 4.1.7: Distribución en $\phi$ del primer vecindario a $t = 0$



## 4.2. Filtro para corrección de errores

En toda la sección anterior se analizaron los posiciones de cada molécula haciendo una transformación una y otra vez intentado encontrar un patrón en estos datos sin embargo, no se obtuvo nada concreto. Siendo este el caso no intentemos cambiar la estructura de los datos si no, la forma en como vemos. Si cada separación de $r, \theta, \phi$ tiene la misma longitud de 300 valores numéricos, pensemos entonces que podemos usar estos valores como vectores y convertilos en matrices de $15 \times 20$, que a su vez, si les asignamos una escala de color estas pasaran a hacer imágenes (ver transformación 4.3.2).

Ejemplo de la transformación de los vectores $r, \theta, \phi$ a matrices

$$\underset{\text{Vector } r}{\begin{bmatrix} 0{,}55 & 0{,}99 & 5{,}3 & 3{,}0 & 5{,}235 & .. & .. & 7{,}34 \end{bmatrix}} \rightarrow \underset{\text{Matriz } r}{\begin{bmatrix} 0{,}55 & 0{,}99 & ... & 4{,}34 \\ 2{,}53 & 3{,}45 & ... & 6{,}7 \\ 6{,}55 & 3{,}99 & ... & 3{,}0 \\ ... & ... & ... & ... \\ 4{,}66 & 4{,}99 & ... & 7{,}34 \end{bmatrix}}$$

$$\underset{\text{Vector } \theta}{\begin{bmatrix} 0{,}2 & 0{,}9 & 1{,}3 & 1{,}0 & 0{,}235 & .. & .. & 1{,}34 \end{bmatrix}} \rightarrow \underset{\text{Matriz } \theta}{\begin{bmatrix} 0{,}2 & 0{,}9 & ... & 2{,}34 \\ 0{,}3 & 4{,}5 & ... & 3{,}7 \\ 0{,}55 & 2{,}9 & ... & 1{,}0 \\ ... & ... & ... & ... \\ 0{,}66 & 4{,}9 & ... & 1{,}34 \end{bmatrix}}$$

$$\underset{\text{Vector } \phi}{\begin{bmatrix} 1{,}2 & 2{,}9 & 1{,}09 & 0{,}05 & 0{,}005 & .. & .. & 0{,}14 \end{bmatrix}} \rightarrow \underset{\text{Matriz } \phi}{\begin{bmatrix} 1{,}2 & 2{,}9 & ... & 2{,}34 \\ 1{,}3 & 3{,}5 & ... & 0{,}7 \\ 0{,}05 & 0{,}91 & ... & 0{,}03 \\ ... & ... & ... & ... \\ 0{,}06 & 0{,}9 & ... & 0{,}14 \end{bmatrix}}$$

Esta es solo una representación de como seria la transformaron de las matrices reales debido a que es imposible colocar los 300 valores de estas. EL código de como se realizo esto se encuentra en el apéndice 8.1.5 cuadro 5, donde se emplea la función reshape para hacer esta acción.

Las matrices conseguidas si las consideramos como imágenes nos permitirán aplicar el método de suavizado el cual consta en transformar una imagen que presenta cierta inconsistencia en los datos (desface o un ruido), en una imagen sin defectos bruscos. La inconsistencia en los datos de una imagen está dada por la continuidad de cada píxel respecto a sus vecinos.

Para toda imagen f(x,y) donde $y$ son las columnas y $x$ el numero de filas en una imagen, se le aplicará un operador T, de tal forma que el resultado g(x,y) sea una imagen ya procesada (suavizada) (González R, 2018 [6]).

$$g(x,y) = T(f(x,y)) \tag{4.2.1}$$

El operador $T$ consta de una matriz cuadrada de tamaño $n \times n$. Este operador realizará un recorrido por toda la imagen que queramos suavizar mediante una serie de convoluciones, dando así como resultado una nueva imagen $g(x,y)$ (ver ecuación 4.2.2). El operador que usemos nunca deberá exceder en tamaño a nuestra imagen.

$$g(x,y) = \sum_{u=0}^{n} \sum_{v=0}^{n} T(u,v)F(x+u, y+v) \tag{4.2.2}$$

Donde $n$ es el tamaño de la matriz del operador $T$

En esta ocasión se prefirió emplear al operador $T$ como un filtro promedio el cual, consta de una matriz con todos sus valores iguales. El objetivo del filtro consta en remplazar una pequeña sección de la imagen por la suma promedio de esta sección. El nuevo valor generado dará como resultado una imagen con continuidad osea suavizada. Aunque el resultado tras aplicar el operador pareciera que altera el tamaño de la imagen original, este método no lo hace, ya que, es capaz de generar ceros en los contornos de la imagen original para así no alterar su tamaño. A esta técnica se le conoce como relleno.

Para la imagen construida a partir de la matriz $r$ se utilizo el siguiente operador $T = [4 \times 4] \cdot 1/15$.

| Matriz ejemplo | | | | | Operador T | | | | | Suavizado | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0,55 | 0,99 | ... | 4,34 | | 1/15 | 1/15 | 1/15 | 1/15 | | 0,37 | 0,73 | ... | 1,02 |
| 2,53 | 3,45 | ... | 6,7 | | 1/15 | 1/15 | 1/15 | 1/15 | | 2,03 | 0,56 | ... | 1,78 |
| 6,55 | 3,99 | ... | 3,0 | $\times$ | 1/15 | 1/15 | 1/15 | 1/15 | $\rightarrow$ | 1,43 | 0,93 | ... | 3,93 |
| ... | ... | ... | ... | | 1/15 | 1/15 | 1/15 | 1/15 | | ... | ... | ... | ... |
| 4,66 | 4,99 | ... | 7,34 | | | | | | | 3,37 | 2,33 | ... | 1,0 |

Para la imagen construida a partir de la matriz $\theta$ se utilizo el siguiente operador $T = [5 \times 5] \cdot 1/13$.

| Matriz ejemplo | | | | | Operador T | | | | | | Suavizado | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0,05 | 3,99 | ... | 2,34 | | 1/13 | 1/13 | 1/13 | 1/13 | 1/13 | | 1,7 | 0,3 | ... | 0,2 |
| 1,53 | 0,45 | ... | 5,7 | | 1/13 | 1/13 | 1/13 | 1/13 | 1/13 | | 1,03 | 0,6 | ... | 2,78 |
| 2,55 | 0,99 | ... | 1,0 | $\times$ | 1/13 | 1/13 | 1/13 | 1/13 | 1/13 | $\rightarrow$ | 2,43 | 0,93 | ... | 0,93 |
| ... | ... | ... | ... | | 1/13 | 1/13 | 1/13 | 1/13 | 1/13 | | ... | ... | ... | ... |
| 3,66 | 1,99 | ... | 0,34 | | 1/13 | 1/13 | 1/13 | 1/13 | 1/13 | | 2,37 | 0,33 | ... | 0,01 |

Aunque se intento aplicar un filtro de suavizado sobre la imagen $\phi$, no se consiguió ningún resultando favorable sobre los datos, por lo que, no se le aplico.

Los datos resaltados en rojo de las matrices anteriores solo son un ejemplo de como es que funciona la convolución, no son los datos reales. Por otro lado, el operador $T$ tanto para $r$ como para $\theta$ si son los operadores reales empleados en las imágenes $r, \theta, \phi$ originales (ver la imagen 4.3.2)

## 4.3. Filtro de resalte de bordes

Prosiguiendo con la extracción de la información se decide aplicar un filtro conocido como resalte de bordes, este consta de emplear dos métodos para su aplicación:

El primer método se trata de un filtro Gaussiano, que suavizara la imagen promedio total.

La función Gaussiana es aproximada a los filtros binomiales que se obtienen a partir del triángulo de pascal. Esta función tiene la siguiente forma:

$$f(x) = N\epsilon^{-ax^2} \tag{4.3.1}$$

Donde $N$ es una constante de normalización que depende de $a$ que en su forma estándar normalmente tiene un valor de $a = 1/2\sigma^2$

Para obtener un filtro binomial es necesario aplicar la siguiente función binomial:

$$f_N(x) = \begin{bmatrix} N \\ x \end{bmatrix} \frac{N!}{x!(N-x)!} \tag{4.3.2}$$

Donde $N$ en este caso es el orden del filtro deseado.

Los filtros binomiales son separables y convolucionales, por lo que podemos generar un filtro primero en dirección de $x$ y luego en dirección de $y$. Si convolucionamos un filtro $x$ consigo mismo este generara un filtro de segundo orden.

$$f_2(x) \cdot f_2(x) = f_4(x) \tag{4.3.3}$$

Un filtro binomial de dos dimensiones se genera a partir de:

$$[f_N(x)]^T \times [f_N(x)] \tag{4.3.4}$$

por ejemplo un filtro de orden 2:

$$\begin{bmatrix}1\\2\\1\end{bmatrix} \times \begin{bmatrix}1 & 2 & 1\end{bmatrix} = \begin{bmatrix}1 & 2 & 1\\2 & 4 & 2\\1 & 2 & 1\end{bmatrix} \tag{4.3.5}$$

Estos filtros normalmente se normalizan para no alterar la imagen:

$$\frac{1}{16}\begin{bmatrix}1 & 2 & 1\\2 & 4 & 2\\1 & 2 & 1\end{bmatrix} \tag{4.3.6}$$

Este núcleo binomial lo podemos aplicar como un filtro gaussiano por la similitud. Un filtro gaussiano en forma depende de su dimensión y tiene la siguiente forma

para una dimensión

$$f(x) = \frac{1}{\sqrt{2\pi}\cdot\sigma}e^{-(\frac{x^2}{2\sigma^2})} \tag{4.3.7}$$

para dos dimensiones

$$f(x,y) = \frac{1}{2\pi\cdot\sigma^2}e^{-(x^2+y^2/2\sigma^2)} \tag{4.3.8}$$

para N dimensiones

$$f(\vec{x}) = \frac{1}{(\sqrt{2\pi}\cdot\sigma)^N}e^{-(|x|^2/2\sigma^2)} \tag{4.3.9}$$

Donde $\sigma$ es el valor de suavizado de imagen, entre mayor sea este valor mayor será el suavizado, por lo contrario entre menor sea, menor será la reducción del ruido.

El segundo método es el de la derivada digital. Este filtro asigna la variación de una imagen con el valor de 1 y por el contrario cuando existen valores contantes asigna el valor de 0. Como la dirección de la imagen puede estar dada por el eje $x$ o por el eje $y$ es necesario realizar una derivada parcial respecto a estos ejes $x, y$. (ver ecuación 4.3.10)

$$\bigtriangledown f(x,y) = \begin{bmatrix}g_x\\g_y\end{bmatrix} \quad \begin{matrix}g_x = \dfrac{\partial f(x,y)}{\partial x} = f(x,y) - f(x-1,y)\\ g_y = \dfrac{\partial f(x,y)}{\partial y} = f(x,y) - f(x,y-1)\end{matrix} \tag{4.3.10}$$

O bien podemos aplicar su forma matricial de la misma forma como se aplico el operador $T$ en la sección anterior:

Derivada digital

Matriz ejemplo en dirección de $x$ Resalte de bordes

$$\begin{bmatrix} 0{,}37 & 15{,}73 & ... & 7{,}02 \\ 8{,}03 & 1{,}56 & ... & 3{,}78 \\ 4{,}43 & 2{,}93 & ... & 2{,}93 \\ ... & ... & ... & ... \\ 13{,}37 & 12{,}33 & ... & 17{,}0 \end{bmatrix} \cdot \begin{bmatrix} 0 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 0 & ... & 0 \\ 1 & 1 & ... & 0 \\ 0 & 0 & ... & 0 \\ ... & ... & ... & ... \\ 1 & 1 & ... & 1 \end{bmatrix}$$

Derivada digital

Matriz ejemplo en dirección de $y$ Resalte de bordes

$$\begin{bmatrix} 0{,}37 & 15{,}73 & ... & 7{,}02 \\ 8{,}03 & 1{,}56 & ... & 3{,}78 \\ 4{,}43 & 2{,}93 & ... & 2{,}93 \\ ... & ... & ... & ... \\ 13{,}37 & 12{,}33 & ... & 17{,}0 \end{bmatrix} \cdot \begin{bmatrix} 0 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 1 & ... & 0 \\ 0 & 1 & ... & 0 \\ 0 & 0 & ... & 0 \\ ... & ... & ... & ... \\ 1 & 1 & ... & 1 \end{bmatrix}$$

Una vez calculado el vector gradiente, calculamos su modulo como se muestra en la siguiente ecuación:

$$| \bigtriangledown f(x,y)| = \sqrt{g_x^2 + g_y^2} \approx |g_x| + |g_y| \tag{4.3.11}$$

Esta suma generará una matriz donde ya habrá obtenido los datos de mayor intensidad, siendo estos los bordes de la imagen.(ver imagen 4.3.1)

Figura 4.3.1: Ejemplo de la aplicación de filtro binomial - derivada digital en una imagen común

La teoría explicada en toda esta sección no fue posible representarla paso por paso con nuestros datos reales debido a que la implementación en el lenguaje Python solo abarca unas cuantas lineas. (ver apéndice 8.1.5, cuadro 7)

Las imágenes resultantes de aplicar el filtro de suavizado y el filtro Gaussiano + la derivada digital, mediante Python fueron las siguientes:

Figura 4.3.2: Aplicación de filtros

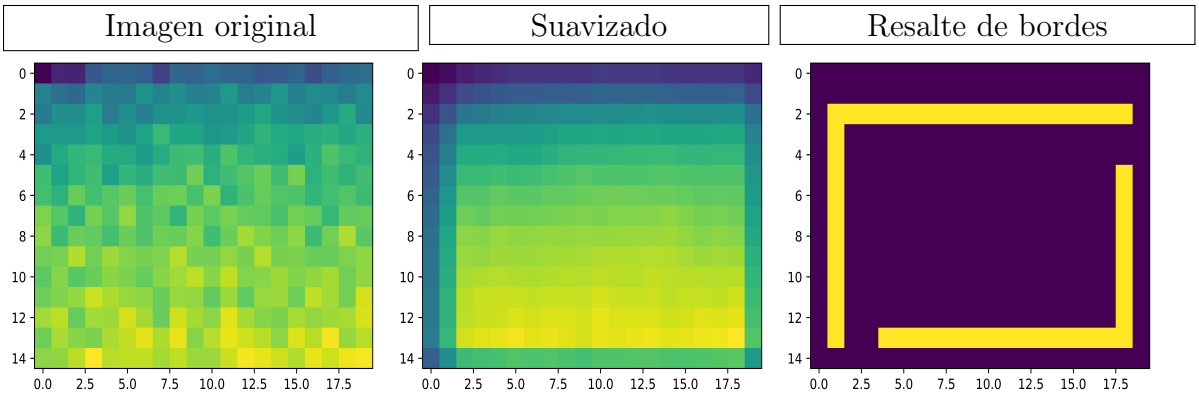



Imagen real $r$ con sus filtros correspondientes.

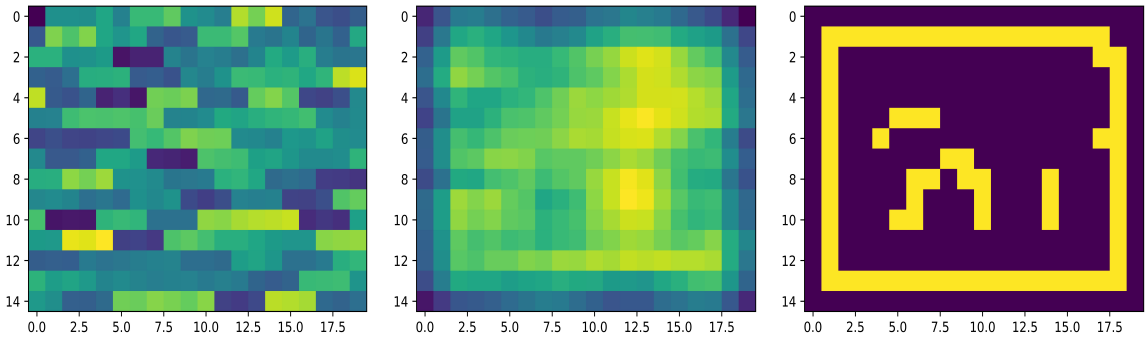


Imagen real $\theta$ con sus filtros correspondientes.


Imagen real $\phi$ con sus filtros correspondientes

Finalmente y antes de implementar una RNA se aconseja que se genere una distribución aleatoria normal, con nuestros datos de entrenamiento, en virtud de obtener un mejor resultado a la hora de entrenar nuestro modelo.

## 4.4. Conversión binaria

Para poder entrenar una RNA sin importar sus configuración son necesarias dos cosas, los impulsos iniciales (Entradas) y la respuesta de estos impulsos (Salidas), sin embargo, en las secciones anteriores solo se a trabajado con las posiciones iniciales de nuestras aguas, dejando de lado a las fuerzas totales (fuerzas de Coulomb + fuerzas de LJ), esto es así debido a que el objetivo de una RNA es manipular los menos posible los resultados esperados, con el único propósito de efectuar la menor cantidad de procesos, por lo que, no es recomendable no alterar estos datos.

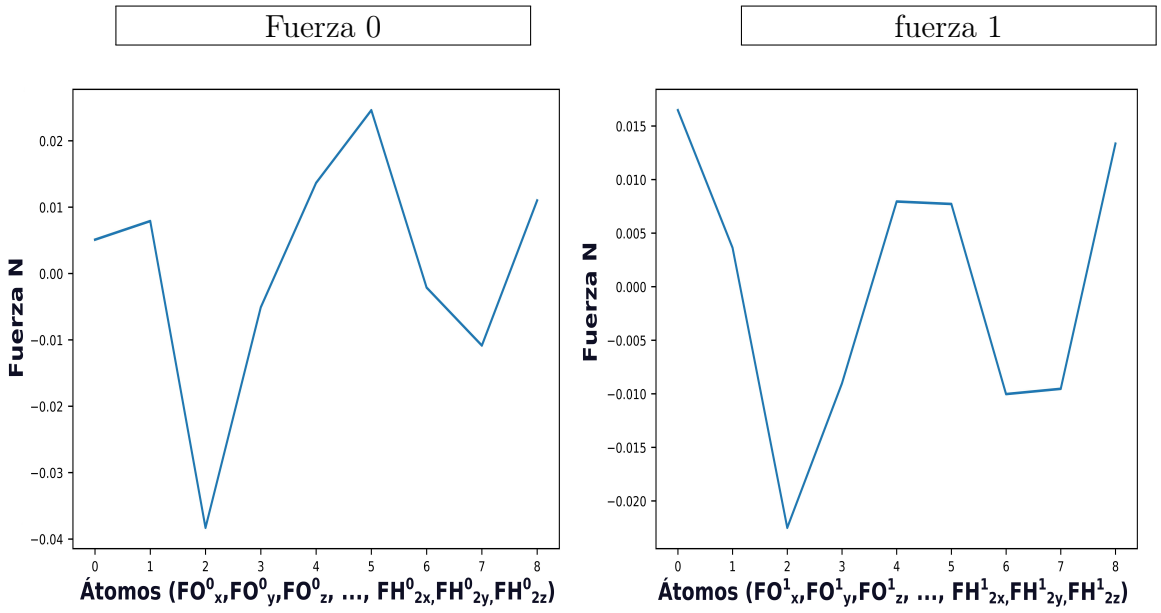
Al revisar las fuerzas resultantes así como su distribución (ver grafica 4.0.4), estas fuerzas básicamente son aleatorias. Al no poder modificar estas fuerza mejor analicemos su relación respecto a sus posiciones atómicas (ver la configuración 4.4.1), al igual que sus distribuciones espaciales ( ver las graficas 4.4.1).

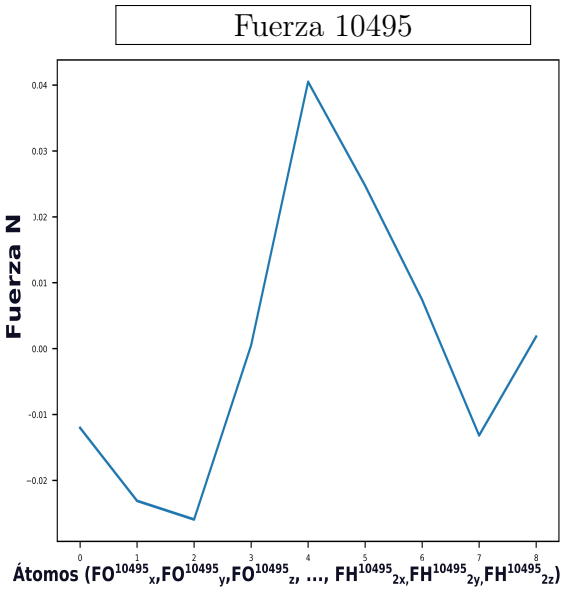
$$FO_x^0, FO_y^0, FO_z^0, FH_{1x}^0, FH_{1y}^0, FH_{1z}^0, FH_{2x}^0, FH_{2y}^0, FH_{2z}^0$$

$$FO_x^1, FO_y^1, FO_z^1, FH_{1x}^1, FH_{1y}^1, FH_{1z}^1, FH_{2x}^1, FH_{2y}^1, FH_{2z}^1$$

$$FO_x^{10496}, FO_y^{10496}, FO_z^{10496}, FH_{1x}^{10496}, FH_{1y}^{10496}, FH_{1z}^{10496}, FH_{2x}^{10496}, FH_{2y}^{10496}, FH_{2z}^{10496} \tag{4.4.1}$$

Figura 4.4.1: Distribución de las fuerzas totales en una molécula de agua en $t = 0$

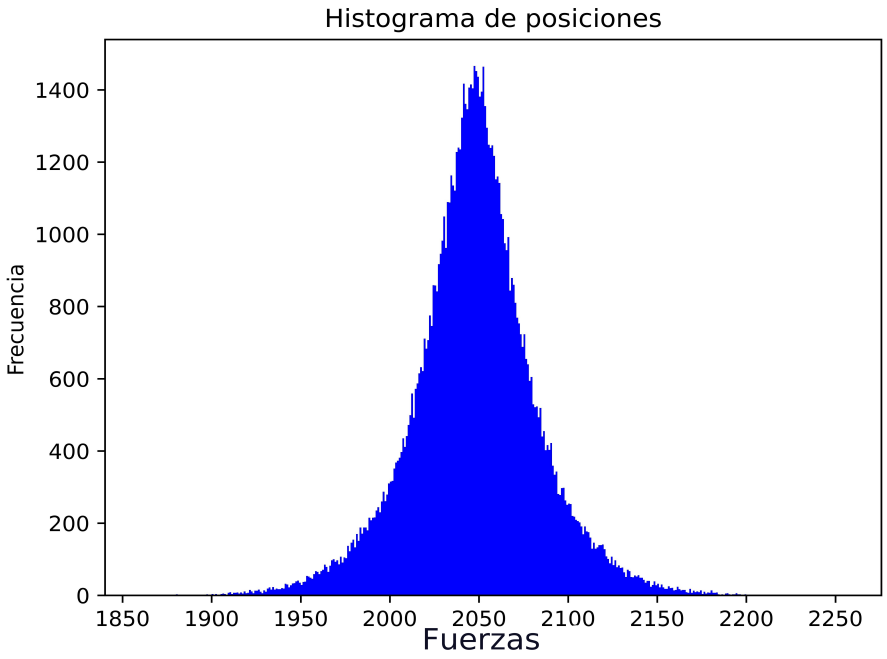





Aunque solo se tratan de nueve fuerzas en ningún caso se repiten, siendo este un problema a la hora de entrenar nuestra RNA. Si cada fuerza resultante es diferente la RNA nunca podrá generar una convergencia. Si dos entradas son parecidas la respuesta a estas entradas no serán parecidas si no totalmente diferentes por lo que, nunca podrá predecir correctamente. Examinemos entonces que tan similares pueden ser estas fuerzas mediante un histograma. Como la fuerza en ningún momento sobrepasa el rango que va de -1 a 1 newtons, segmentemos este rango en 4095 (ver ecuación 4.4.2) partes para generar el histograma.

$$\text{Saltos de frecuencia} = 2/4095 = 0{,}0004884 \tag{4.4.2}$$

Figura 4.4.2: Distribución de las fuerzas totales de forma individual




La máxima concentración de los fuerzas se haya entre los valores 2040-2060 y su pico sobre los 2050 que si realizamos la conversión tendremos:

$$\begin{aligned} &0{,}0004884 \cdot 2040 = 0{,}996336 \rightarrow 0{,}996336 - 1 = -0{,}003664 \\ &0{,}0004884 \cdot 2050 = 1{,}001221001 \rightarrow 1{,}001221001 - 1 = 0{,}001221001 \\ &0{,}0004884 \cdot 2060 = 1{,}006104 \rightarrow 1{,}006104 - 1 = 0{,}006104 \end{aligned} \tag{4.4.3}$$

Por lo que, casi toda nuestra información esta cercana al cero, y que a su vez es la menos relevante. Por ello aunque tengamos un sistema casi aleatorio aun podemos predecir las fuerzas de mayor intensidad, que son las que mas afectan a nuestro sistema. Como entre cada sección del histograma no existen mas de 200 datos y la variación entre ellos es mínima podemos tomar sus posiciones dentro de este para hacer una transformación posicional de nuestros datos sin afectarlos bruscamente. Por ejemplo si la fuerza $FO_x^0$ fuera igual a $-0{,}45055$ entonces su posición en el histograma seria de 1125. El código para realizar esta transformación se encuentra en el apéndice 8.1.4.

$$\begin{bmatrix} FO_x^n & FO_y^n & FO_z^n & FH_{1x}^0 & FH_{1y}^n & FH_{1z}^n & FH_{2x}^n & FH_{2y}^n & FH_{2z}^n \end{bmatrix}_{[1,9]}$$

$$\begin{bmatrix} 1125 & 1155 & 1135 & 1600 & 2620 & 2630 & 2200 & 1260 & 2175 \end{bmatrix}_{[1,9]}$$

Los vectores anteriores muestra un ejemplo de como serian las posiciones de las fuerzas de una molécula de agua dentro del histograma.

Para facilitar la perdición de estas fuerzas con forma de Gaussiana, se aplico una conversión que transforma la posición numérica decimal de la fuerza dentro del histograma, en un

sistema de recorrido binario. Se toman las 4095 separaciones por cada fuerza y se genera un vector de ceros de este tamaño, después se toma el valor de la posición dentro de la Gaussiana para activar con unos solo hasta este valor. El siguiente cuadro da un ejemplo de como se realizaría esta transformación.

$$\begin{aligned} &\text{si } FO_x^1 = 3 \rightarrow [1\ 1\ 1\ 0\ 0\ 0\ 0\ ...\ 0\ 0\ 0\ 0\ 0\ 0]_{[1,4095]} \\ &\text{si } FO_x^8 = 6 \rightarrow [1\ 1\ 1\ 1\ 1\ 1\ 0\ ...\ 0\ 0\ 0\ 0\ 0\ 0]_{[1,4095]} \end{aligned} \tag{4.4.4}$$

Aplicando este procedimiento paras las 9 fuerzas de una agua, tendríamos la siguiente transformación:

$$\begin{bmatrix} 1125 & 1155 & 1135 & 1600 & 2620 & 2630 & 2200 & 1260 & 2175 \end{bmatrix}_{[1,9]}$$

$$\begin{bmatrix} 11..,00 & 11..,00 & 11..,00 & 11..,00 & 11..,00 & 11..,00 & 11..,00 & 11..,00 & 11..,00 \end{bmatrix}_{[1,36855]}$$

Es decir tendríamos un vector de salida de $9 * 4095 = 36855$ elementos con puros unos y ceros.

# Capítulo 5

# Especificaciones de la red neuronal

Antes de construir una RNA, primero debemos entender que este procedimiento pese a lo definido en el capitulo 3 es totalmente incierto. Entonces se deberán encontrar a prueba y error, el numero de capas ocultas, el numero de neuronas, las funciones de activación e incluso la taza de aprendizaje. Sin que esto garantice que la distribución encontrada sea la mas optima para nuestro tipo de datos.

Atendiendo lo anterior se empleo un modelo estructural general proporcionado por Tensor-Flow y Keras. Estos programas eliminan la construcción manual de los modelos de RNA, dejando a nuestra disposición todas las posibles combinaciones de los parámetros anteriores que pudieran tener estas estructuras.

Las RNA pre-fabricadas siempre requieren una configuración previa, ya sea la instalación de los controladores, la configuración de nuestro entorno de trabajo, etc. Por ello si quiere utilizar el código implementado en el apéndice 8.1.5 cuadro 16, primero deberá instalar CUDA, Python, keras,tensorflow, tensorflow-gpu, cuDNN y si trabaja con Linux configurar su .bashrc agregando el path de CUDA.

## 5.1. Definición de parámetros

Aunque no exista un procedimiento a seguir si existen ciertas recomendaciones que podrían facilitar la búsqueda de una configuración idónea. lo primero es definir un número de neuronas limitado así como emplear solo tres capas ocultas, la de los datos de entrada, la de los datos de salida y solo una capa oculta. Después utilizar una taza de aprendizaje muy grande respecto a nuestros datos. Finalmente aplicar solo una función de activación para todas las capas ocultas. El objetivo es generar un sobre entrenamiento. Al tener un red sobre entrenada, todos aquellos datos que se encontraban fuera del conjunto de entrenamiento generaran una predicción errada. El único fin para generar esto es conocer los limites de nuestros datos, tanto como si son requeridas mas neuronas, mas capas ocultas o bien no hemos excedido en nuestra configuración.

En nuestro caso se uso una función sigmoidea como inicial. 100 neuronas en nuestra en nuestra capa intermedia. Una taza de aprendizaje de 0.1. Debido al tamaño de nuestros datos no se alcanzo el sobre entrenamiento en el primer intento. Utilizando esta red como pivote fue posible modificar al modelo para encontrar uno que fuera capaz generar un predicción lo suficientemente satisfactoria con 10 cuadros de tiempo como datos de entrenamiento .

la siguiente tabla muestra cada una de las características de la red neuronal encontrada.

Cuadro 5.1: Parámetros utilizados en el modelo de neuronal

| Capa | Función de activación | Número de Neuronas | Taza de aprendizaje | Datos de entrada | Datos de salida | Épocas |
|---|---|---|---|---|---|---|
| 1 | *tanh/entrada* | 900 | 0,0001 | 94464000 | 3868300800 | 10000 |
| 2 | *tanh* | 4000 | | | | |
| 3 | *tanh* | 5500 | | | | |
| 4 | *tanh* | 3200 | | | | |
| 5 | *sigmoid* | 2000 | | | | |
| 6 | *tanh* | 1950 | | | | |
| 7 | *tanh* | 900 | | | | |
| 8 | *tanh/salida* | 36855 | | | | |

* Donde los datos de entrada = 31488 (Número de datos del contenedor) ∗3 (Número de coordenadas x,y,z) ∗100 (Número de vecinos) ∗10 (Cuadros de tiempo) = 94464000

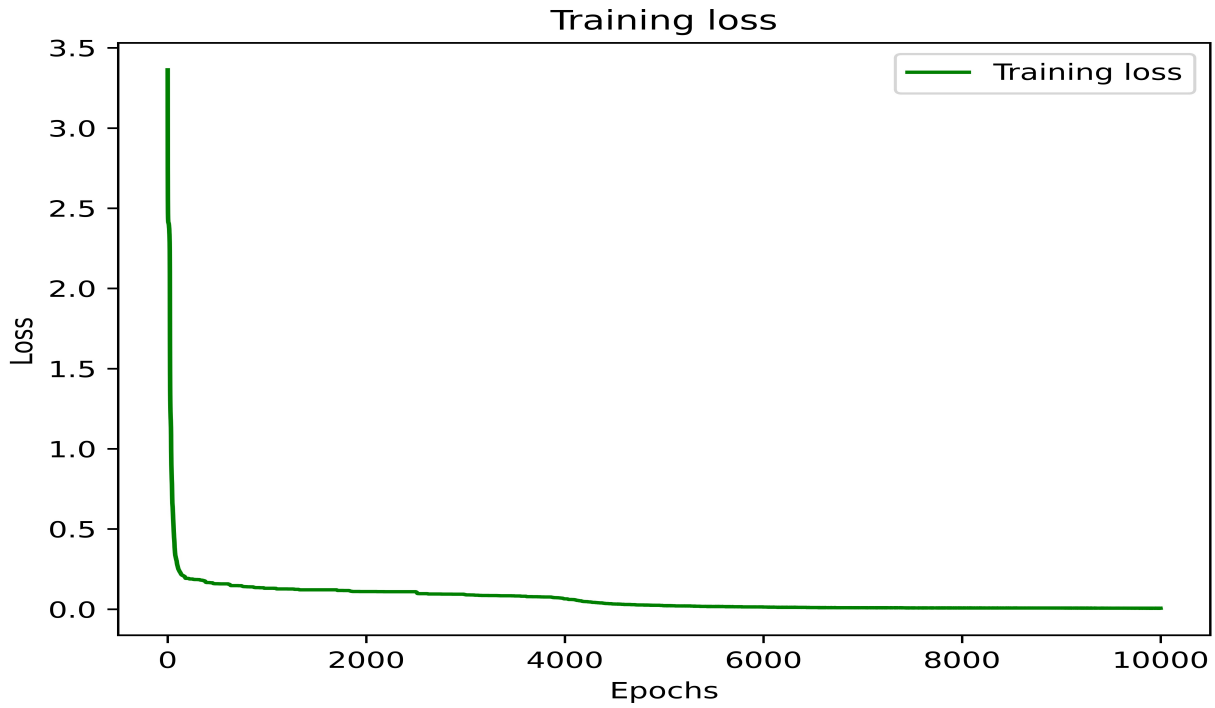
* Los datos de salidas = 31488 (Número de fuerzas resultantes) ∗3(Número de coordenadas) ∗4095 (Número de intervalos en la Gaussiana) ∗10 (Cuadros de tiempo) = 3868300800

* Capa de salida = 4095 (Número de intervalos de partición de las fuerzas acotadas encontradas [-1,1] ) ∗9 (Coordenadas x,y,z de los 3 átomos de la molécula de agua)= 36855

* Capa de entrada = 100 (Vecinos mas cercanos convertidor en r,theta,phi) ∗9 (Coordenadas x,y,z de los 3 átomos de la molécula de agua)= 900

Al entrenar este modelo se llegó a la máxima capacidad de nuestro equipo de computo al emplear toda la Random Access Memory (RAM) disponible y el máximo procesamiento tanto de las GPUs como de los procesadores. Asi se consiguió la siguiente taza de error. 5.1.1

Figura 5.1.1: Resultado del entrenamiento




A manera de resumen el siguiente diagrama 5.1.2 muestra el recorrido por el que pasaron los datos hasta llegar a este punto.

Figura 5.1.2: Recorrido de los datos

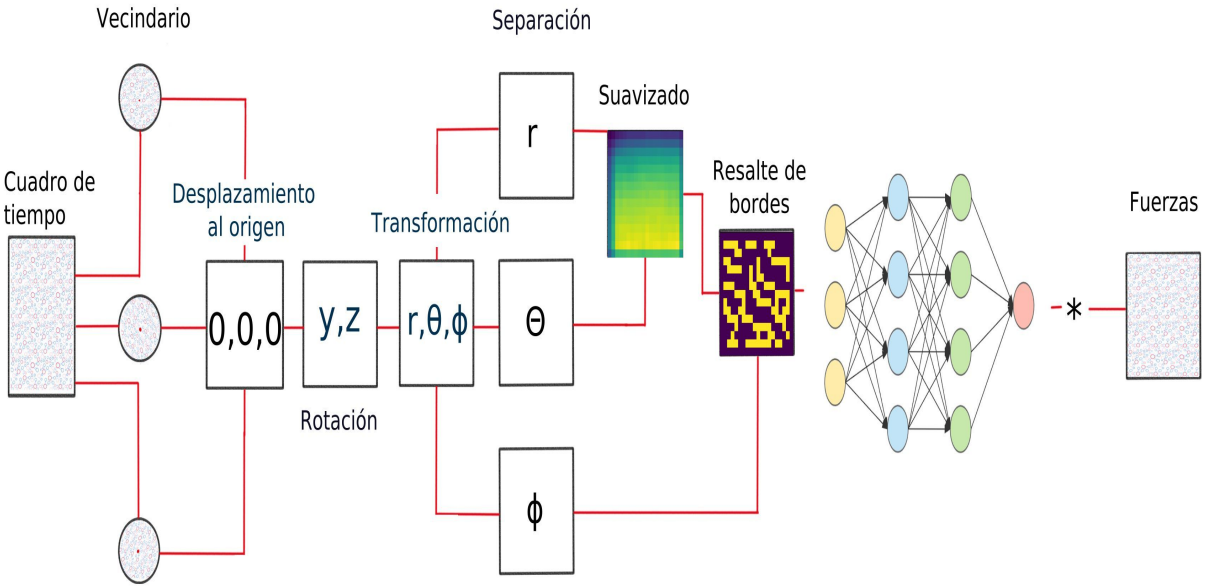
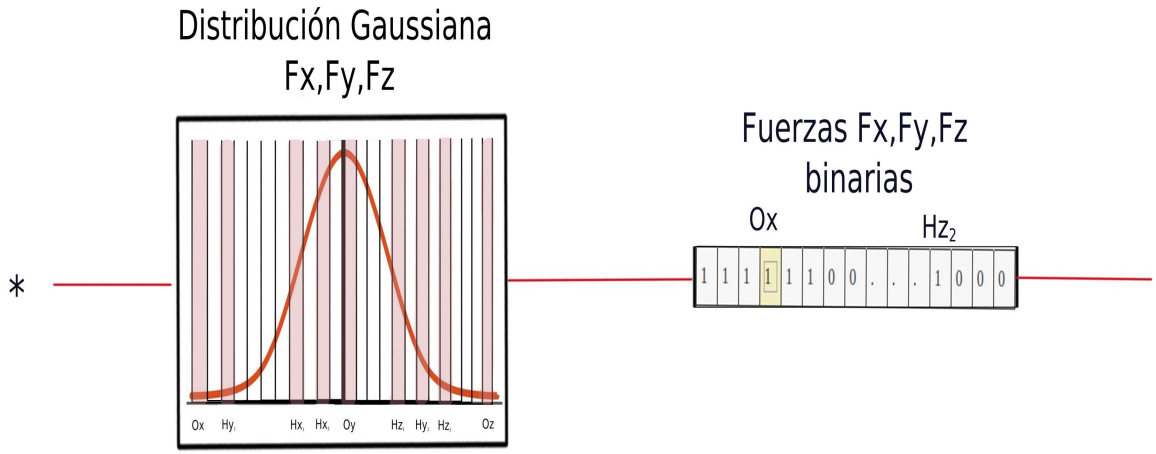



Figura 5.1.3: Recorrido de los datos de salida



## 5.2. Predicciones

Una vez se entreno el modelo anterior de RNA, se requiere verificar que el aprendizaje adquirido en el conjunto de neuronas sea el correcto. La predicción de la RNA con datos fuera del conjunto de entrenamiento genero las siguientes graficas.

Figura 5.2.1: Fuerzas obtenidas con cálculos clásicos y transformadas a un sistema de posicionamiento binario $t = 11$

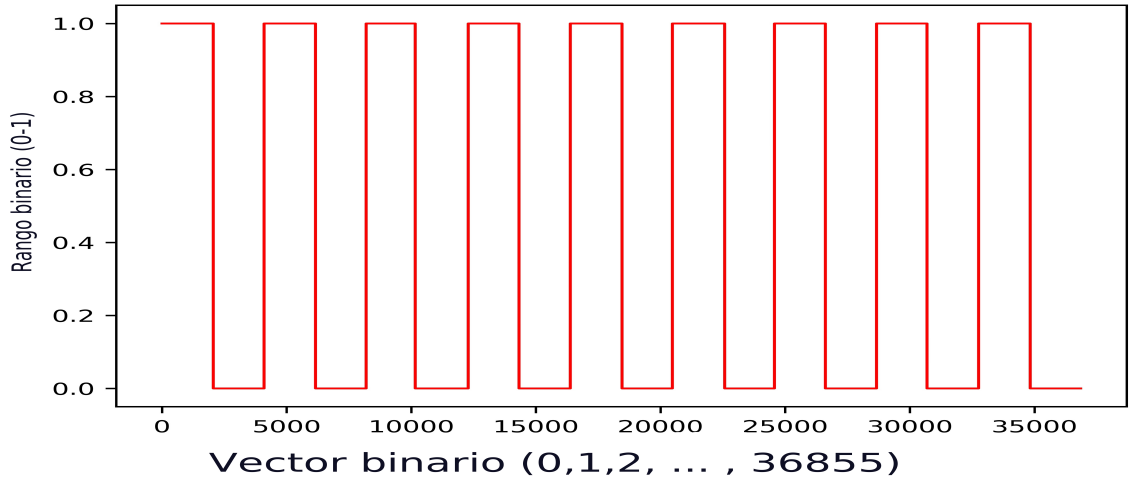
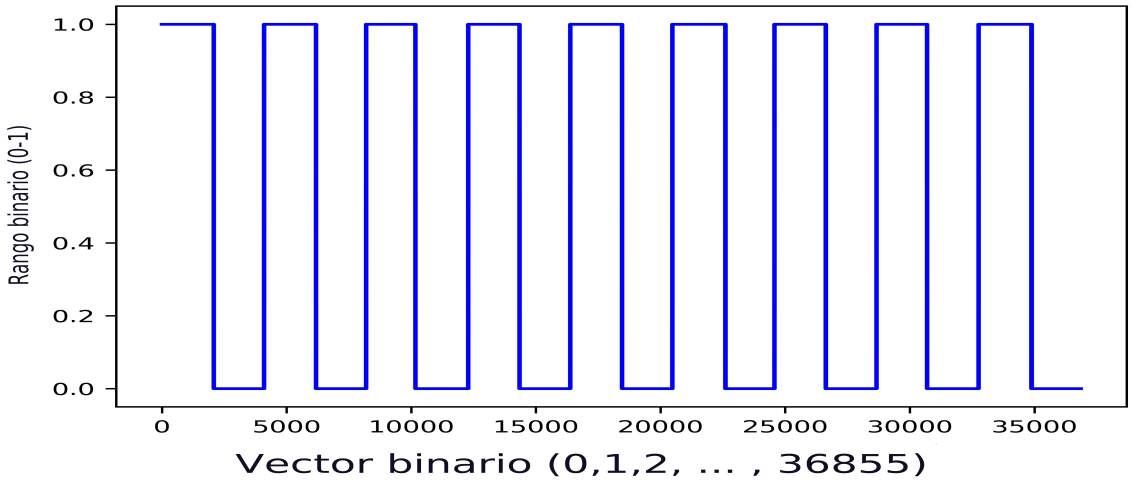
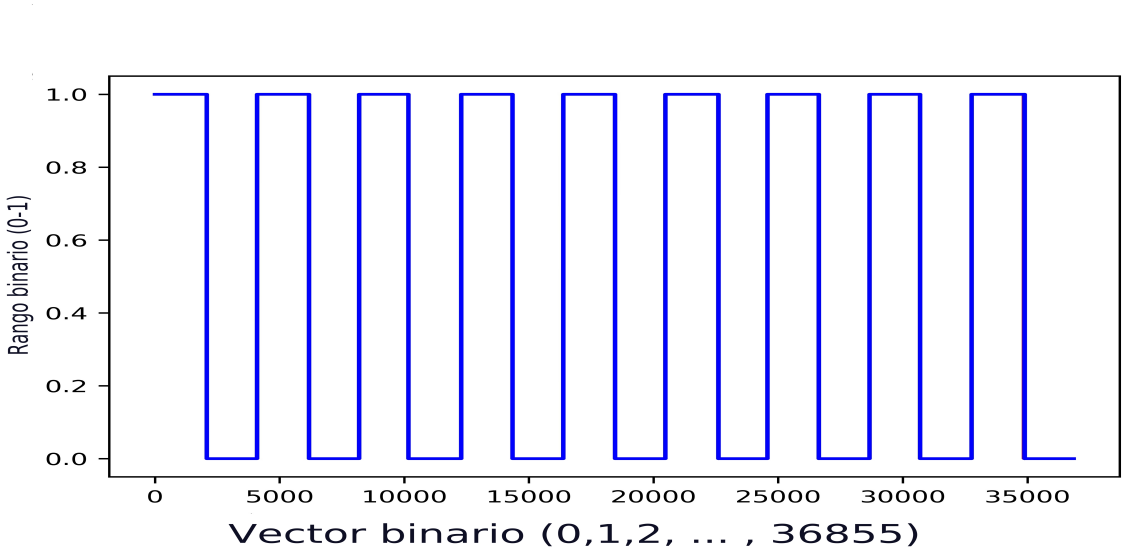



Figura 5.2.2: Fuerzas predichas con la RNA en su transformación de posicionamiento binario $t = 11$



Para comparar la diferencia entre los datos predichos y los datos calculados se unieron las graficas anteriores en una sola obteniendo así sus diferencias (ver grafica 5.2.3)

Figura 5.2.3: Comparación de los datos predichos contra los datos calculados




A la vista resalta que nos encontramos ante una predicción que podríamos denominar casi perfecta sin embargo, como aplicamos una conversión posicional binaria, ahora debemos aplicar de forma inversa este procedimiento para obtener con ello los datos reales. El primer paso es transformar cada una de las tramas de unos y ceros en vectores numéricos para lo cual solo es necesario contar el numero de unos existentes dentro del vector. Al finalizar este proceso para los 9 elementos se obtienen los valores posicionales dentro de la Gaussiana. Un ejemplo de ello se muestra a continuación:

$$\begin{bmatrix} 11..,00 & 11..,00 & 11..,00 & 11..,00 & 11..,00 & 11..,00 & 11..,00 & 11..,00 & 11..,00 \end{bmatrix}_{[1,36855]}$$

Pasamos de un vector de posiciones binarias, a un vector de enteros los cuales representan el posicionamiento dentro de la Gaussiana.

$$\begin{bmatrix} 1125 & 1155 & 1135 & 1600 & 2620 & 2630 & 2200 & 1260 & 2175 \end{bmatrix}_{[1,9]}$$

Después se tomaran estos números posicionales para realizar el siguiente procedimiento:

$$\begin{aligned} &1125 \cdot 0{,}0004884 = 0{,}54945 \\ &\text{Dato real}=0{,}54945 - 1 = -0{,}45055 \end{aligned} \tag{5.2.1}$$

Donde el 0,0004884 corresponde a la separación del histograma del sección 4.4, ecuación 4.4.2.

Este procedimiento se realizo para las 10496 tramas de unos y ceros. Obteniendo así los datos que predigo la red, cabe destacar que la precisión de la predicción de los datos se ve afectada por la separación del histograma como del mismo error de la predicción de la RNA. La siguiente tabla muestra los primeros datos obtenidos con el procedimiento anterior.

Cuadro 5.2: Datos reales comparados con los datos predichos

| | Datos reales | Datos RNA |
|---|---|---|
| $FO_x^0$ | 0,005089264538 | 0,006667 |
| $FO_y^0$ | 0,007895352043 | 0,018001 |
| $FO_z^0$ | −0,038309211518 | 0,017334 |
| $FH_{1x}^0$ | −0,005100089552 | −0,004 |
| $FH_{1y}^0$ | 0,013636783711 | 0,012667 |
| $FH_{1z}^0$ | 0,024596679766 | 0,007334 |
| $FH_{2x}^0$ | −0,002122767949 | 0,004667 |
| $FH_{2y}^0$ | −0,010854328721 | 0,020001 |
| $FH_{2z}^0$ | 0,010996639129 | 0,006667 |

Debido a que los resultados obtenidos con la predicción anterior obtuvo buenos resultados en la mayoría de los casos, se decide emplear un conjunto de datos totalmente separado de los datos. Se uso un cuadro de tiempo con una separación de 20 minutos respecto a los datos de entrenamiento.

.

Figura 5.2.4: Fuerzas obtenidas con cálculos clásicos y transformadas a un sistema de posicionamiento binario $t = 100$

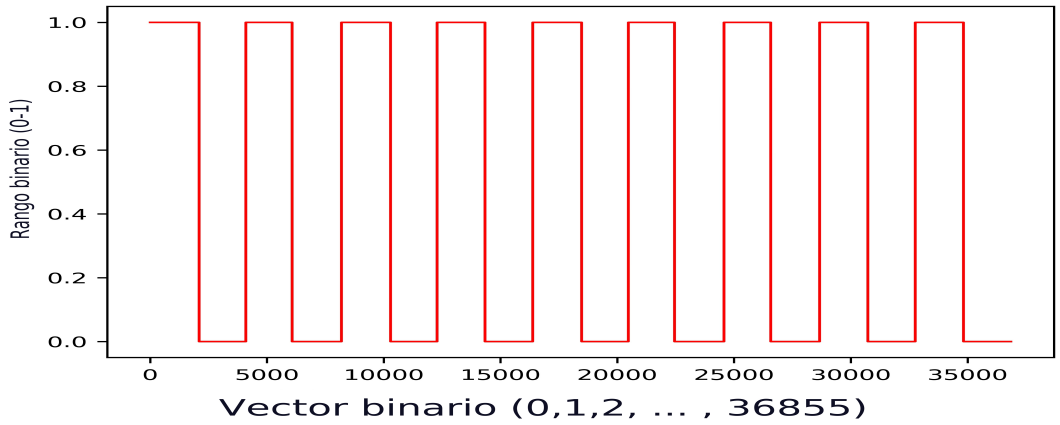
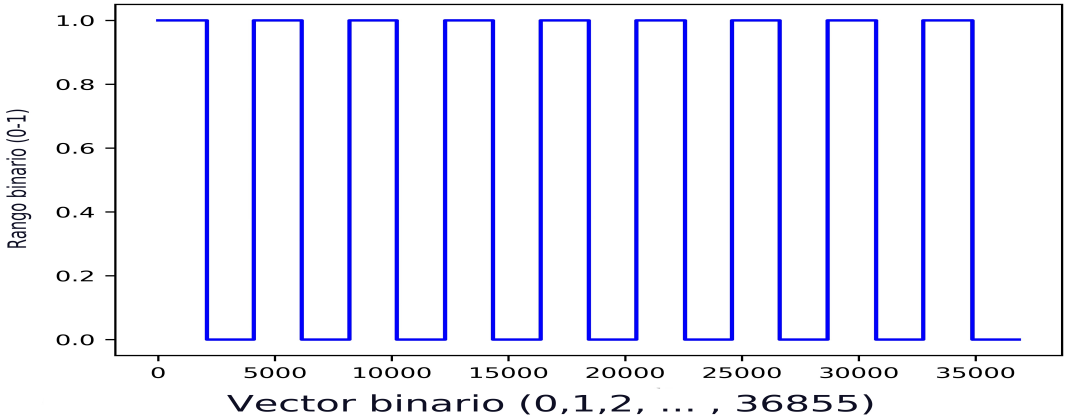
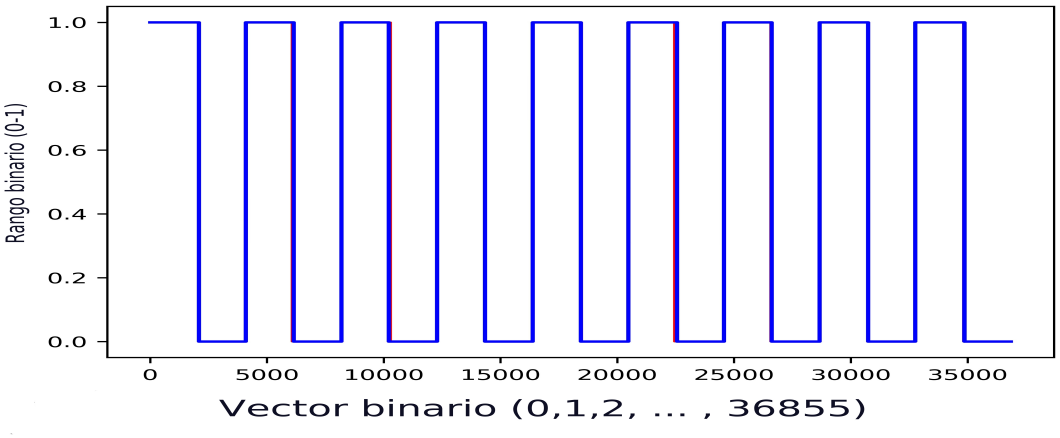



Figura 5.2.5: Fuerzas predichas con la RNA en su transformación de posicionamiento binario $t = 100$




Nuevamente veamos la unión de las dos graficas anteriores.

Figura 5.2.6: Comparación de los datos predichos contra los datos calculados




Aunque nuevamente se ve una predicción muy buena sin embargo, como se demostró en el cuadro anterior esto no quiere decir que la exactitud de los datos sea perfecta.

# Capítulo 6

# Resultados

En virtud de que en el capitulo anterior se obtuvieron buenos resultados a la hora de predecir las fuerzas totales de una molécula de agua, entonces utilicemos esa metodología sobre todo un cuadro de tiempo.

Figura 6.0.1: Distribución de las fuerzas obtenidas mediante cálculos clásicos en $t = 15$

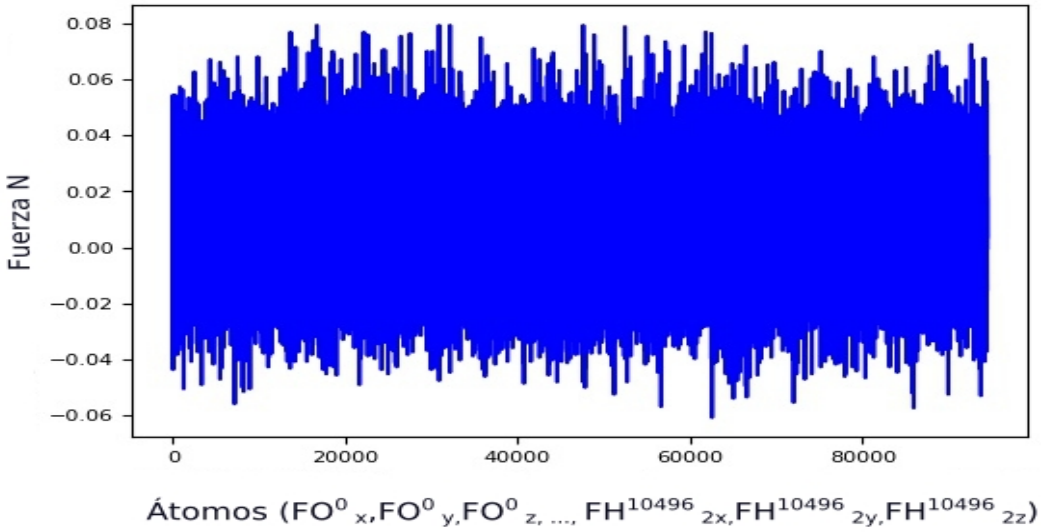


Figura 6.0.2: Distribución de las fuerzas predichas por la RNA en $t = 15$

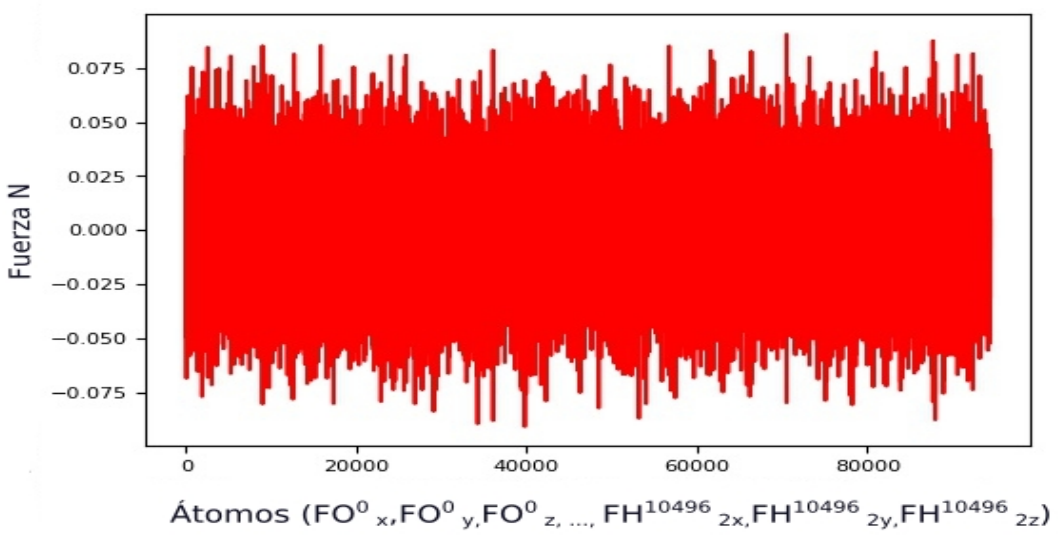
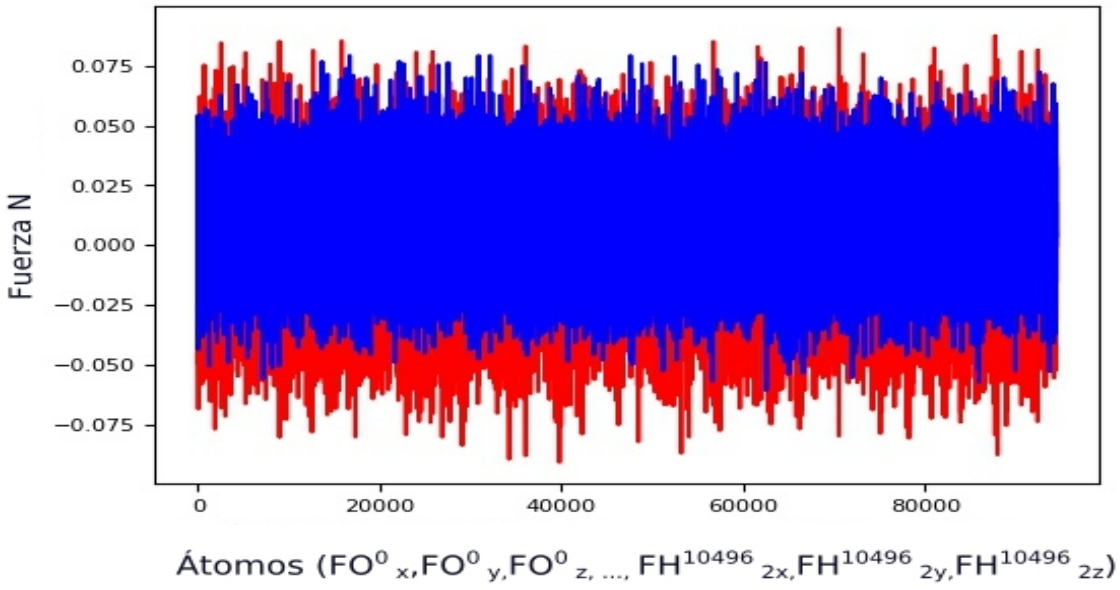



Figura 6.0.3: Comparación entre los datos predichos contra los datos calculados



Si obtenemos el error cuadrático medio de estos datos mediante la formula 6.0.1, obtenemos un error de 0.0005907620854341286

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{6.0.1}$$

Ahora si calculamos la desviación cuadrática media mediante la formula 6.0.2, obtenemos una desviación de 0.02430559782095739.

$$MSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{6.0.2}$$

Estos datos nos dicen que la predicción que estamos obteniendo no es del todo favorable ya que, el valor de 0,024... aun es muy grande para este tipo de datos, sin embargo, como lo denotan las graficas, los datos predichos no pierden estructura manteniendo un rango que no supera los limites que van de $-1, 1$, lo cual nos dice que este es el camino correcto a seguir para un encontrar un modelo que obtenga una predicción mucho mas precisa.

# Capítulo 7

# Conclusiones

Al realizar este procedimiento similar al de una RNA convolucional podemos decir que la predicción alcanzada en la totalidad de los datos aunque carece de una precision absoluta es lo suficientemente capaz de describir cualquier conjunto de fuerzas dentro de cualquier contenedor sometido a las misma condiciones que las que se emplearon en esta tesis.

El error que obtenemos sobre los datos esta dado por diferentes situaciones. La primera de ellas es debido al numero de neuronas empleado, ya que, este es un numero limitado por el hardware con el que contamos. Otra dificultad consta en la separación realizada en el histograma, al no ser exacta sobre todos los datos. Si bien, es posible corregir estas dificultades ya sea que, se tenga un mejor equipo de computo, se aumenten el numero de neuronas empleado, se aumente la resolución de la separación del histograma o se cree un mejor modelo de RNA, no podemos asegurar que esto se disminuya el error.

Aunque el método empleado en esta tesis consume mucho tiempo en limpiar los datos de entrada, al igual que, los de salida, aun podemos aseverar que supera al sistema de calculo clásico, ya que, toda esta metodología tiene un comportamiento puramente lineal. Para que exista un claro ganador entre el calculo de las fuerzas de forma clásica y las predicciones de las fuerzas, primero de han de aumentar el numero de moléculas de agua que a su vez aumentara el tamaño del contenedor, si bien esto no representa un gran cambio también es posible generar un aumento en el numero de cuadros de tiempo, de esta forma se fuerza a alcanzar un limite computacional y de tiempo, donde resultaría como vencedor este modelo de RNA.

Una ventaja mas de esta RNA es la facilidad de compartir el modelo entrenado, para que este se ejecute en cualquier computadora que tenga la capacidad de ejecutar CUDA y con una Memoria gráfica de acceso aleatorio VRAM (Video Random Access Memory) superior a los 8 gigabytes.

# Capítulo 8

# Apendices

## 8.1. Programas realizados

### 8.1.1. Vecindario de 100 moléculas de agua código Python-Cuda

El siguiente código resuelve el problema visto el capitulo 4. Aquí se implementan las formulas de las distancias respecto al oxigeno central y se guarda un vecindario de 100 moléculas de agua. Para ello se realiza una implementación mediante Python que utiliza el kernel de CUDA. La librería @cuda.jit se encarga de la comunicación de estos dos lenguajes de promoción. Al finalizar el procedimiento el código genera un archivo que guarda cada vecindario encontrado

[1]:
```
# Librerias para cuda,tensorflow,conversion de vectores,
# funciones matematicas
import numpy as np
from numba import cuda,jit
import math
import tensorflow.compat.v1 as tf
import os
```

[2]:
```
# Se activan las GPU que esten dispobles
# Para este caso solo una

os.environ ['CUDA_VISIBLE_DEVICES'] = '0'
config = tf.ConfigProto()
config.gpu_options.allow_growth = True
sess = tf.Session (config = config)
```

[3]:
```
#Se activa la libreria cuda
@cuda.jit
```

```
#Se define le nombre del núcleo en cuda
#Vector de entrada= entrada
#Salida del metodo= vecinos

def cien_cercanos_kernel(entrada, vecinos):

    #hilo en el que se ejecuta el núcleo
    hilo_actual= cuda.threadIdx.x + cuda.blockDim.x * cuda.blockIdx.x

    hilo_temporal=hilo_actual

    #salto de los datos Ox,Oy,Oz, Hx_0,Hy_0,Hz_0, Hx_1,Hy_1,Hz_1 = 9
    coordenada_xyz=hilo_temporal*9

    # 100 elementos mas cercanos al oxigeno,
    # y sus respectivos hidrogenos= 900
    n_cantidad_datos=coordenada_xyz*100

     # Coordenada Ox,Oy,Oz central

    componente_x=entrada[0][hilo_actual*9]
    componente_y=entrada[0][hilo_actual*9+1]
    componente_z=entrada[0][hilo_actual*9+2]

    #Distancia inicial
    distsalida=0

    # Posición del dato xyz encontrado
    identificador=0;

    # se inicializa el limite inferior para comparar la distancia
    lim_inferior=-100

    #Recorrido sobre los 100 atomos mas cercanos el central
    for l in range(100):

        # Se inicializa el limite superior para comparar la distancia
        lim_superior=1000

        # Se recorre el cuadro de tiempo de tamaño 31488*3
        for m in range(0,94464,9):

            # se calcula la distancia respecto al atomo central
```

```
            distancia=(entrada[0][m]-componente_x)**2
                                                    ␣
↪+(entrada[0][m+1]-componente_y)**2+(entrada[0][m+2]-componente_z)**2
            distancia=math.sqrt(distancia)

            if (distancia<lim_superior and distancia>lim_inferior):

                # Guardado del minimo local mas cercano
                # cada uno con 3 coordenada xyz. Al terminar
                # el ciclo se obtiene el minimo global

                coordenadax=entrada[0][m]
                coordenaday=entrada[0][m+1]
                coordenadaz=entrada[0][m+2]
                coordenadax1=entrada[0][m+3]
                coordenaday1=entrada[0][m+4]
                coordenadaz1=entrada[0][m+5]
                coordenadax2=entrada[0][m+6]
                coordenaday2=entrada[0][m+7]
                coordenadaz2=entrada[0][m+8]

                # Actualización el limite superior
                lim_superior=distancia

                # Guardado de la distancia del minimo global
                distsalida=distancia

                # Guardado de la posicion del vecino encontrado
                identificador=m/9

        # Actualización del limite inferor

        lim_inferior=lim_superior

        # Guardado del vecino mas cercano
        # encontrado

        vecinos[0][n_cantidad_datos]=coordenadax
        vecinos[0][n_cantidad_datos+1]=coordenaday
        vecinos[0][n_cantidad_datos+2]=coordenadaz
        vecinos[0][n_cantidad_datos+3]=coordenadax1
        vecinos[0][n_cantidad_datos+4]=coordenaday1
        vecinos[0][n_cantidad_datos+5]=coordenadaz1
        vecinos[0][n_cantidad_datos+6]=coordenadax2
```

```
        vecinos[0][n_cantidad_datos+7]=coordenaday2
        vecinos[0][n_cantidad_datos+8]=coordenadaz2

        # Incremento de 9 para guardar las siguientes coordenadas
        n_cantidad_datos+=9

        if(l>=100):
            break

        # Limpieza de la distancia para encontrar distancias repetidas
        distancia=0

        # Se realiza nuevamente el procedimiento pero  esta vez
        # comparando si exite una distancia igual a la encontrada
        # con anterioridad

        for m in range(0,94464,9):
            distancia=(entrada[0][m]-componente_x)**2+
            (entrada[0][m+1]-componente_y)**2+(entrada[0][m+2]
            -componente_z)**2
            distancia=math.sqrt(distancia)

            #dato=abs(compara-dato)

            if (distancia==distsalida and (m/9)!=identificador):
                vecinos[0][n_cantidad_datos]=entrada[0][m]
                vecinos[0][n_cantidad_datos+1]=entrada[0][m+1]
                vecinos[0][n_cantidad_datos+2]=entrada[0][m+2]
                vecinos[0][n_cantidad_datos+3]=entrada[0][m+3]
                vecinos[0][n_cantidad_datos+4]=entrada[0][m+4]
                vecinos[0][n_cantidad_datos+5]=entrada[0][m+5]
                vecinos[0][n_cantidad_datos+6]=entrada[0][m+6]
                vecinos[0][n_cantidad_datos+7]=entrada[0][m+7]
                vecinos[0][n_cantidad_datos+8]=entrada[0][m+8]
                n_cantidad_datos+=9
                l=l+1
                if(l>=99):
                    break
        if(l>=100):
            break
```

[4]:
```
# Se define la funcion que llamara al nucleo al núcleo
def cuda_gpu(coordenadas):

    # Se envia la informacion a la gpu
    gpu_cuda = cuda.to_device(coordenadas)

    # Se crea un espacio temporal para los resultados
    resultado_temporal = np.zeros(shape=(1,9446400), dtype=np.float32)

    # Se envia la informacion  a la gpu
    gpu_cuda_res = cuda.to_device(resultado_temporal)

    # Número de bloques
    hilos_por_bloque = 128

    # Número de hilos
    bloques_por_malla = 82
    # La multiplicación de estos dos elementos deben dar como resultado
    # el tamaño de los elementos a trabajar 128*82=10496
    # en este caso 31488/3

    # Se llama al núcleo y se le envia la información
    cien_cercanos_kernel[bloques_por_malla, hilos_por_bloque](gpu_cuda,␣
 ↪gpu_cuda_res)

    # Se copian los datos y se sincronizan
    resultado = gpu_cuda_res.copy_to_host()

    # Se devuelve el vector
    return resultado
```

[5]:
```
# Se lee el archivo de las posiciones iniciales
movie='movie.xyz'
movied=open(movie, 'r')
dato=movied.read().split()

# Se limpian los datos de informacion inecesaria
separador=4

# Se define el vector para mandarlo a la gpu
entrada_cuda=[]

#Número de frames
frames=2
```

```
for i in range(frames):
    entrada_cuda.append([])

    # Se recorre el cuadro dato por dato
    for j in range(0,31488*3,3):

        # Se llena el vector
        entrada_cuda[i].append(float(dato[separador]))
        entrada_cuda[i].append(float(dato[separador+1]))
        entrada_cuda[i].append(float(dato[separador+2]))

        # Informacion inecesaria
        separador=separador+4
    separador=separador+3
```

[6]:
```
# Se define un vector de salida al núcleo cuda
resultado=[]

# Se recorren los frames
for i in range(frames):

    # Se tranforma el frame en un vector
    ent1_cuda=np.asarray([entrada_cuda[i]])

    # Se llena el vector
    resultado.append(cuda_gpu(ent1_cuda))
```

[7]:
```
h="h"
o="o"
# Guardado de los datos en formato x y z
file = open("salida_cuda.xyz", "w")
file.write("300" + os.linesep)          # Ox, Oy, Oz
file.write("XYZ" + os.linesep)          # Hx_0, Hy_0, Hz_0
cont=0                                  # Hx_1, Hy_1, Hz_1
for i in range(frames):
    for j in range(0,9446400,9):
        file.write(o+"   "+ str(resultado[i][0][j])+"  ,
        " +str(resultado[i][0][j+1])+"  ,  "+ str(resultado[i][0][j+2])+
        os.linesep)
        file.write(h+"   "+ str(resultado[i][0][j+3])+
        "  ,  " +str(resultado[i][0][j+4])+"  ,
        "+ str(resultado[i][0][j+5])+ os.linesep)
        file.write(h+"   "+ str(resultado[i][0][j+6])+
```

```
        "   ,  " +str(resultado[i][0][j+7])
        +"   ,  "+ str(resultado[i][0][j+8])+ os.linesep)
        cont+=1
        if cont==100:
            cont=0
            file.write("300" + os.linesep)
            file.write("XYZ" + os.linesep)
file.close()
```

### 8.1.2. Codigo en Fortran-CUDA

El siguiente programa utiliza una sistema de programación estructurada utilizando el lenguaje C, sin embargo, se le llama CUDA debido a que emplea métodos que solo se pueden emplear en este lenguaje. La parte de Fortran solo se utilizo para enviar los datos posicionales del agua así como escribir los vecindarios después del calculo en paralelo.

[ ]:
```
// ********************
// Obteniendo 0-99 vecinos por cada átomo
// de un conjunto de N átomos
// ********************


# include <stdio.h>
# include <math.h>
# include <assert.h>
# include <iostream>
# include <cuda_runtime.h>


typedef struct

  {int Ndatos;
   int DatosInicio, DatosFin;           // inicio & fin de los datos
   float *Datos_L, *Datos_D;            // local & dispositivo de entrada
   double *resultados_L, *resultados_D; // local & dispositivo de salida
   cudaStream_t stream;
  } TGPUarray;

   const int num_GPU_max= 32;      // maximo número de GPUs


// **********************
    __global__ static void
```

```
    reduceKernel (float *xx, double *resultados_D, int nAtoms, int␣
 ↪vecinos)
// *********************

  {const int indice= blockIdx.x * blockDim.x + threadIdx.x;

  float aa, bb, cc;         // coordenadas con indice i
  double dist, distOut;     // variables temporales de distancia
  double LimInf, LimSup;    // limites iniciales
  double indentificador;    // Átomo J

  int cont= 0.0;             // contador
  int nAtCoords= 9*nAtoms;   // numero de atomos por coordenada
  int knt= 9*indice*vecinos; // tamaño de los saltos

// coordenadas por átomo con indice

  aa= xx[indice*9+0];
  bb= xx[indice*9+1];
  cc= xx[indice*9+2];

  LimInf= -1.0;


  for (int ll=0; ll < vecinos; ll++)   //
                                       //
// inicializando el limite superior    //
  {LimSup= 1234.0;                     //
                                       //
// inicializando el contador           //
   cont=0;                             //
                                       //
// find vecinos 'll' of atom 'indice'     // ----
                                          //
  for (int mm=0; mm < nAtCoords; mm= mm+9)   //
                                          //
 {dist= powf ((xx[mm+0]-aa),2) +          //
        powf ((xx[mm+1]-bb),2) +          //
        powf ((xx[mm+2]-cc),2);           //
                                          //
  dist= sqrtf(dist);                      //
  cont++;                                 //
                                          //
                                          //
```

```
  if (LimInf < dist &&                          //
               dist <= LimSup)                  //
     {LimSup= dist;                             //
      indentificador= cont;                     //
      distOut= dist;}                           //
  }
                                  // --------
                                  //
// guardando y actulizando limites //
   LimInf= LimSup;                //
   resultados_D[knt+0]= distOut;
   resultados_D[knt+1]= indentificador;
   resultados_D[knt+2]= indice;
                                  //
   knt= knt + 3;                  //

  if(ll>=vecinos){break;}

 dist=0;
 cont=0;
 for (int mm=0; mm < nAtCoords; mm= mm+3)   //
                                            //
 {dist= powf ((xx[mm+0]-aa),2) +            //
        powf ((xx[mm+1]-bb),2) +            //
        powf ((xx[mm+2]-cc),2);             //
                                            //
  dist= sqrtf(dist);                        //
  cont++;                                   //

  if ( dist==distOut && indentificador!=(cont*1.0))
     {
      resultados_D[knt+0]= distOut;
      resultados_D[knt+1]= (cont*1.0);
      resultados_D[knt+2]= indice;
      knt= knt + 3;
      if(ll>=vecinos){break;}
      ll=ll+1;
     }
  }
  if(ll>=vecinos){break;}

 }

 } // cierre de kernel
```

```
  extern "C" void

   kernel_ia_(int *nPar, int *Vecinos, int *Mdata,
              float *coords, double *buf)

  {
// data-structured with name array
   TGPUarray array[num_GPU_max];

// data for the gpu parallel module
   int nAtoms= *nPar;         // número de átomos
   int vecinos= *Vecinos;     // Átomo central + 99 vecinos
   int nDatos= *Mdata;        // 3*Natoms*100
   int i, j, nGPUs;

   int nBlocks= 256  ;        // número de bloques
   int nThreads= 123;         // número de hilos por bloque

// Número de disponible GPU
   cudaGetDeviceCount(&nGPUs);

// Número de datos por GPU
   int nDatosPorGPU= nDatos/nGPUs;


   int primero= 0, fin= nDatosPorGPU-1;

// número de particulas por GPU
   for (i= 0; i < nGPUs; i++)

  {array[i].Ndatos= nDatosPorGPU;}


   for (i=0; i < nGPUs; i++)

// limites por GPU
  {array[i].DatosInicio= primero;
   array[i].DatosFin= fin;

   printf (" %d:   %d  %d\n",
             i+1, primero, fin);
```

```
 primero= fin + 1;
 fin=   fin + nDatosPorGPU;
}



 for (i= 0; i < nGPUs; i++)

{cudaSetDevice(i);
 cudaStreamCreate (&array[i].stream);

 cudaMalloc ((void **) &array[i].Datos_D,    nDatos*sizeof(float));
 cudaMalloc ((void **) &array[i].resultados_D, nDatos*sizeof(double));

 cudaMallocHost ((void **) &array[i].Datos_L,    nDatos*sizeof(float));
 cudaMallocHost ((void **) &array[i].resultados_L,␣
↪nDatos*sizeof(double));

 for (int ij= 0; ij < nDatos; ij++)

{array[i].Datos_L[ij]= coords[ij];}

}


 for (i= 0; i < nGPUs; i++)

{cudaSetDevice(i);

 cudaMemcpyAsync (array[i].Datos_D,
                  array[i].Datos_L,
                    nDatos*sizeof(float),
                    cudaMemcpyHostToDevice,
                  array[i].stream);

 reduceKernel <<< nBlocks, nThreads, 0, array[i].stream >>>
                                        (array[i].Datos_D,
                                         array[i].resultados_D,
                                          nAtoms,
                                          vecinos);


 cudaMemcpyAsync (array[i].resultados_L,
```

```
                    array[i].resultados_D,
                      nDatos*sizeof(double),
                      cudaMemcpyDeviceToHost,
                    array[i].stream);

}


int knt1= 0, knt2= 0;


for (i= 0; i < nGPUs; i++)

{cudaSetDevice(i);
 cudaStreamSynchronize (array[i].stream);

for (j= knt2; j < array[i].Ndatos + knt2; j++)
{buf[knt1]= array[i].resultados_L[j];
 knt1 +=1;}

 knt2= knt1;
}


 for (i= 0; i < nGPUs; i++)

{cudaSetDevice(i);
 cudaStreamSynchronize(array[i].stream);

 cudaFreeHost (array[i].Datos_L);
 cudaFree     (array[i].Datos_D);

 cudaFreeHost (array[i].resultados_L);
 cudaFree     (array[i].resultados_D);

 cudaStreamDestroy(array[i].stream);
 cudaDeviceReset();
}


}
```

### 8.1.3. Desplazamiento al origen, rotación sobre los ejes y,z y conversión del plano x,y,z al r,theta,fhi

Aplicando las formulas vistas en el capitulo 4.1, se desarrollo en siguiente código que obtiene el desplazamiento al origen de cada vecindario. Después crea una rotacion sobre el los ejes x,y. Finalmente realiza un cambio de coordenadas $x, y, z$ a $r, theta, fhi$. Destacar que se toma en cuenta en que cuadrante se encuentra cada coordenada $x, y, z$ para poder así aplicar el procedimiento adecuado.

[1]:
```
# Leer datos

Datos_cuda='salida_cuda.xyz
tem=open(Datos_cuda, 'r')

# Separacion del archivo

datos=tem.read().split()
separador=3
contador=0

# Vector del vecindario

datos_100=[]

frames=10496*15

for i in range(frames):
    datos_100.append([])
    for j in range(300*3):
        datos_100[i].append(datos[separador])
        separador=separador+2
    separador=separador+2
```

[2]:
```
# Desplazamiento al origen 0,0,0

desplazamiento=[]
for i in range(frames):
    desplazamiento.append([])
    xcentral=float(datos_100[i][0])
    ycentral=float(datos_100[i][1])
    zcentral=float(datos_100[i][2])
    for j in range(0,len(datos_100[i]),3):
        desplazamiento[i].append(float(datos_100[i][j])-xcentral)
        desplazamiento[i].append(float(datos_100[i][j+1])-ycentral)
```

```
        desplazamiento[i].append(float(datos_100[i][j+2])-zcentral)
```

[3]:
```
# Rotación sobre los ejes coordenados y,z

import math

datos2_100=[]
datos3_100=[]
datos4_100=[]
for i in range(frames):
    datos2_100.append([])
    datos3_100.append([])
    datos4_100.append([])

    x1=float(desplazamiento[i][3])
    y1=float(desplazamiento[i][4])

    if (x1==0):
        tetaz=3.0*(math.pi/2)
    else:
        v1=math.atan(abs(y1/x1))
        if(x1>0.0 and y1==0.0):
            tetaz=0

        if(x1>0.0 and y1>0.0):
            tetaz=v1

        if(x1==0.0 and y1>0.0):
            tetaz=math.pi/2

        if(x1<0.0 and y1>0.0):
            tetaz=math.pi-v1

        if(x1<0.0 and y1==0.0):
            tetaz=math.pi

        if(x1<0.0 and y1<0.0):
            tetaz=math.pi+v1

        if(x1==0.0 and y1<0.0):
            tetaz=3.0*(math.pi/2)
```

```
        if(x1>0.0 and y1<0.0):
            tetaz=2.0*math.pi-v1


    for j in range(0,len(desplazamiento[i]),3):
        x=float(desplazamiento[i][j])
        y=float(desplazamiento[i][j+1])
        z=float(desplazamiento[i][j+2])
        datos2_100[i].append(x*math.cos(tetaz) + y*math.sin(tetaz))
        datos2_100[i].append(-x*math.sin(tetaz) + y*math.cos(tetaz))
        datos2_100[i].append(z)


    x2=float(datos2_100[i][3])
    z2=float(datos2_100[i][5])
    v1=math.atan(abs(z2/x2))

    if(x2>0.0 and z2==0.0):
        tetay=0

    if(x2>0.0 and z2>0.0):
        tetay=v1

    if(x2==0.0 and z2>0.0):
        tetay=math.pi/2.0

    if(x2<0.0 and z2>0.0):
        tetay=math.pi-v1

    if(x2<0.0 and z2==0.0):
        tetay=math.pi

    if(x2<0.0 and z2<0.0):
        tetay=math.pi+v1

    if(x2==0.0 and z2<0.0):
        tetay=3.0*(math.pi/2)

    if(x2>0.0 and z2<0.0):
        tetay=2.0*math.pi-v1


    for j in range(0,len(datos2_100[i]),3):
        x=float(datos2_100[i][j])
```

```
        y=float(datos2_100[i][j+1])
        z=float(datos2_100[i][j+2])

        datos3_100[i].append(x*math.cos(tetay) + z*math.sin(tetay))
        datos3_100[i].append(y)
        datos3_100[i].append(-x*math.sin(tetay) + z*math.cos(tetay))

    if(datos3_100[i][3]<0.0):
        for j in range(0,len(datos3_100[i]),3):
            x=float(datos3_100[i][j])
            y=float(datos3_100[i][j+1])
            z=float(datos3_100[i][j+2])

            datos3_100[i][j]=x*math.cos(math.pi) + y*math.sin(math.pi)
            datos3_100[i][j+1]=-x*math.sin(math.pi) + y*math.cos(math.pi)
            datos3_100[i][j+2]=z

    y11=float(datos3_100[i][7])
    z11=float(datos3_100[i][8])
    v1=math.atan(abs(z11/y11))

    if(y11>0.0 and z11==0.0):
        tetax=0

    if(y11>0.0 and z11>0.0):
        tetax=v1

    if(y11==0.0 and z11>0.0):
        tetax=math.pi/2

    if(y11<0.0 and z11>0.0):
        tetax=math.pi-v1

    if(y11<0.0 and z11==0.0):
        tetax=math.pi

    if(y11<0.0 and z11<0.0):
        tetax=math.pi+v1

    if(y11==0.0 and z11<0.0):
        tetax=3.0*(math.pi/2)

    if(y11>0.0 and z11<0.0):
        tetax=2.0*math.pi-v1
```

```
    for j in range(0,len(datos3_100[i]),3):
        x=float(datos3_100[i][j])
        y=float(datos3_100[i][j+1])
        z=float(datos3_100[i][j+2])

        datos4_100[i].append(x)
        datos4_100[i].append(y*math.cos(tetax) + z*math.sin(tetax))
        datos4_100[i].append(-y*math.sin(tetax) + z*math.cos(tetax))

    if(datos4_100[i][7]<0.0):
        for j in range(0,len(datos3_100[i]),3):
            x=float(datos4_100[i][j])
            y=float(datos4_100[i][j+1])
            z=float(datos4_100[i][j+2])

            datos4_100[i][j]= x
            datos4_100[i][j+1]=y*math.cos(math.pi) + z*math.sin(math.pi)
            datos4_100[i][j+2]=-y*math.sin(math.pi) + z*math.cos(math.pi)
```

[4]:
```
# Método de comprovación

def convertidor(theta, phi,r):
    x = r*math.cos(phi) * math.sin(theta)
    y = r*math.sin(phi) * math.sin(theta)
    z = r*math.cos(theta)
    return x, y, z
```

[5]:
```
#transformacion de x,y,z a r,theta,phi

datarthetafhi=[]
for i in range(frames):
    datarthetafhi.append([])
    for j in range(0,900,3):
        if (j<2):
            datarthetafhi[i].append(0.0)
            datarthetafhi[i].append(0.0)
            datarthetafhi[i].append(0.0)

        else:
            a=round(datos4_100[i][j],6)
            b=round(datos4_100[i][j+1],6)
```

```
            c=round(datos4_100[i][j+2],6)
            temp_r=math.sqrt(a**2 + b**2 + c**2)
            r=temp_r

            phi=math.atan2(b, a)

            theta= math.atan2(math.sqrt(a**2 + b**2),c)

            phi=round(phi,6)
            theta=round(theta,6)

            datarthetafhi[i].append(r)
            datarthetafhi[i].append(theta)
            datarthetafhi[i].append(phi)
```

[6]:
```
h="h"
o="o"
import os
#se guarda los datos en formato x y z
file = open("angulos.rtf", "w")
file.write("300" + os.linesep)                    #O, H, H
file.write("molec" + os.linesep)
file.write("------------" + os.linesep)
cont=0

for i in range(frames):
    for j in range(0,900,9):
        file.write(o+"   "+ str(round(datarthetafhi[i][j],3))
        +"  ,  " +str(round(datarthetafhi[i][j+1],3))+"  ,  "+ str(round(
        datarthetafhi[i][j+2],3))+ os.linesep)
        file.write(h+"   "+ str(round(datarthetafhi[i][j+3],3)) +"  ,  "
        +str(round(datarthetafhi[i][j+4],3))+
        "  ,  "+ str(round(datarthetafhi[i][j+5],3))+ os.linesep)
        file.write(h+"   "+ str(round(datarthetafhi[i][j+6],3
        )) +"  ,  " +str(round(datarthetafhi[i][j+7],3))+"  ,  "+
        str(round(datarthetafhi[i][j+8],3))+ os.linesep)
    file.write("300" + os.linesep)
    file.write("molec" + os.linesep)
    file.write("------------" + os.linesep)

file.close()
```

### 8.1.4. Conversión de las fuerzas C, LJ a un vector posicional binario

Mediante la segmentación de la distribución Guassina de las fuerzas se obtiene un posición. Esta genera un vector de 4095 posiciones por cada coordenada. El llenado de este vector depende de la posición. Si se encuentra en la posición 2000, entonces existirán 2000 unos y 2095 ceros dentro del vector que lo conforma.

El siguiente código implementa lo anterior guardado los vectores posicionales en conjuntos de 4095*9=36855 que son los que conforman una molécula de agua.

[1]:
```
# Libreria para la manipulación de vectores
import numpy as np

# Lectura del archivo
Datos_fuerzas='fuerzas-totales'
tem=open(Datos_fuerzas, 'r')
datos=tem.read().split()
```

[2]:
```
# Separación por renglón y acomodo de las fuerzas C + LJ

separador=3
contador=0
datos_fuerzas=[]

frames=1

for i in range(frames):
    datos_fuerzas.append([])
    for j in range(0,31488*3,3):
        datos_fuerzas[i].append(float(datos[separador]))
        datos_fuerzas[i].append(float(datos[separador+1]))
        datos_fuerzas[i].append(float(datos[separador+2]))
        separador=separador+4
    separador=separador+2

# Liberación de memoria
# limpieza de la variable datos
del datos
```

[3]:
```
# Función que convierte un número decimal
# en un vector de unos y ceros
# decimal=21 -> 21 unos y 4074 ceros

def binario_poscicional(decimal):
    activador=0
```

```
    binario=[]
    for i in range(4095):
        if(i<decimal):
            activador=1
        else:
            activador=0
        binario.append(activador)
    return binario
```

[4]:
```
# Procedimiento para averiguar la posición
# de la fuerza respecto a la gaussina en
# un rango de -1 a 1

incremento=0.0004884
vector_posicion=[]
for i in range(frames):
    vector_posicion.append([])
    for j in range(len(datos_fuerzas[i])):
        contador_posicion=0
        limite_inferior=-1
        for k in range(4095):
            limite_superior=limite_inferior+incremento
            if limite_inferior<=datos_fuerzas[i][j] and␣
 ↪datos_fuerzas[i][j]<=limite_superior:
                decimal=binario_poscicional(contador_posicion)
                vector_posicion[i].append(decimal)
                break;
            limite_inferior=limite_superior
            contador_posicion+=1
```

[5]:
```
# limpieza de la variable datos

del datos_fuerzas
```

[6]:
```
# Acomodo de los datos, 4095 elementos por coordenada
# 4095 en x, 4095 en y, 4095 en z

salida_red=[]
for i in range(frames):
    salida_red.append([])
    for j in range(len(vector_posicion[i])):
        salida_red[i].extend(vector_posicion[i][j])
```

[7]:
```
# Guardado de los frames de 9 posiciones
# 4095 * 9 = 36855

import os
file = open("rangos.tf", "w")
for i in range(frames):
    for j in range(0,len(salida_red[i]),36855):
        for k in range(36855):
            file.write(str(salida_red[i][k+j]))
        file.write(os.linesep)
```

### 8.1.5. Entrenamiento de la red neuronal keras-tensorflow

El siguiente procedimiento convierte un vector de tamaño n en una imagen de n*m. Después suaviza la imagen mediante un kernel y aplica un filtro llamado resalte de bordes. Estas imágenes creadas actuaran como entradas para la red neuronal. También este programa lee las fuerzas transformadas en unos y ceros. Estas fuerzas actúan como respuesta de salida a la red neuronal. Finalmente se entrena el modelo de Keras-Tensorflow así como se guarda el modelo, la taza de aprendizaje y las épocas.

[1]:
```
# Libreria para manejo de vectores

import numpy as np

# Librerias necesarias para la aplicación
# de los filtro

# Restaura la imagen

from skimage import restoration

# Realiza una convolucion con el filtro

from scipy.signal import convolve2d

# Filtro que contiene el resalte de bordes

from skimage import feature

# Algoritmo de revoltura

```

```
import random
```

[2]:
```
#lectura del archivo de angulos

ruta='angulos.rtf'
```

[3]:
```
abrir_archivo=open(ruta, 'r')

# separar datos por renglon

dato=abrir_archivo.read().split()

separador=4
coordenadas_xyz=[]

# número de frames a entrenar

frames=10496*10

for i in range(frames):
    coordenadas_xyz.append([])
    for j in range(300*3):
        coordenadas_xyz[i].append(float(dato[separador]))
        separador=separador+2
    separador=separador+3
```

[4]:
```
# Separación de los datos en vectores r,theta,fhi

r=[]
t=[]
f=[]

for i in range(frames):
    r.append([])
    t.append([])
    f.append([])
    for j in range(0,900,3):
        r[i].append(coordenadas_xyz[i][j])
        t[i].append(coordenadas_xyz[i][j+1])
        f[i].append(coordenadas_xyz[i][j+2])
```

[5]:
```
# Conversión de los vectores r,theta,fhi en matrices
# en un formato lo mas cuadrado posible
```

```
vector_imagen_r=[]
vector_imagen_t=[]
vector_imagen_f=[]

r= np.array(r, "float32")
t= np.array(t, "float32")
f= np.array(f, "float32")
for i in range(frames):
    vector_imagen_r.append(np.asarray(r[i]).reshape(15,20))
    vector_imagen_t.append(np.asarray(t[i]).reshape(15,20))
    vector_imagen_f.append(np.asarray(f[i]).reshape(15,20))
```

[6]:
```
#función para convertir true a 1 y false a 0

def convertir (vector):
    vector=vector.reshape(1,300)
    vector1=[]
    for i in range(300):
        if vector[0][i] == False:
            vector1.append(0)
        if vector[0][i] == True:
            vector1.append(1)
    return vector1
```

[7]:
```
# Kernel normal para r

filtror = np.ones((4,4)) / 15

# Kernel normal para theta

filtrot = np.ones((5,5)) / 13

entrada_red=[]

temporal=[]
for i in range(frames):

    # Convolucion entre la imagen y el kernel

    img = convolve2d(vector_imagen_r[i], filtror, 'same')

    # Aplicación del filtro resalte de bordes en r

    edger = feature.canny(img)
```

```
    # Convolucion entre la imagen y el kernel

    img = convolve2d(vector_imagen_t[i], filtrot, 'same')

    # Aplicación del filtro resalte de bordes en theta

    edget = feature.canny(img)

    # Aplicación del filtro resalte de bordes en fhi

    edgef = feature.canny(vector_imagen_f[i])


    # se aplica la funcion convertir

    vector_unos_r=convertir(edger)

    vector_unos_t=convertir(edget)

    vector_unos_f=convertir(edgef)

    vector_unos_r=np.asarray(vector_unos_r)
    vector_unos_t=np.asarray(vector_unos_t)
    vector_unos_f=np.asarray(vector_unos_f)

    temporal=[]
    temporal.extend(vector_unos_r)
    temporal.extend(vector_unos_t)
    temporal.extend(vector_unos_f)

    # se transforma de matriz a vector
    entrada_red.append(temporal)
```

[8]:
```
# Lectura del impuso de respuesta

salidas='rangos.tf'
```

[9]:
```
# abrir y leer el archivo
salida=open(salidas, 'r')

# separar por renglon las fuerzas
salida1=salida.read().split()
```

```
separador=0

# definir vector salida que contendra una matriz
salidas_red=[]
contador=0

for i in range(frames):
# transformar el vector sal en matriz
    salidas_red.append([])
    for a in range(1):
# guardar las fuerzas
        x = [int(a) for a in str(salida1[separador])]
        salidas_red[i].extend(x)
# satar un renglon
        separador+=1
```

[10]:
```
# Guardado de las posiciones vectoriales

x = range(frames)
ran=list(x)
```

[11]:
```
# Desordenamiento de las posiciones vectoriales

random.shuffle(ran)
```

[12]:
```
# Guardado de datos revueltos

entrada_red_random=[]
salidas_red_random=[]
for i in range(frames):
    entrada_red_random.append(entrada_red[ran[i]])
    salidas_red_random.append(salidas_red[ran[i]])
```

[13]:
```
# Conversión de los datos a flotantes

ent= np.array(entrada_red_random, "float32")
sal= np.array(salidas_red_random, "float32")
```

[14]:
```
# Libreria de tensorflow
import tensorflow.compat.v1 as tf

# Aditamento para seleccionar el gpu
```

```
import os
os.environ ['CUDA_VISIBLE_DEVICES'] = '0'
config = tf.ConfigProto()
config.gpu_options.allow_growth = True
sess = tf.Session (config = config)
```

[15]:
```
# Librerias necesarias para la red neuronal

from keras.models import Sequential
from keras.layers import Dense
from tensorflow import keras
import tensorflow as tf
from keras import models
from keras import layers
```

[16]:
```
# Definición del modelo de la red neuronal

model = models.Sequential()
model.add(layers.Dense(900, activation='tanh', input_shape=(900,)))
model.add(layers.Dense(4000, activation='tanh'))
model.add(layers.Dense(5500, activation='tanh'))
model.add(layers.Dense(3200, activation='tanh'))
model.add(layers.Dense(2000, activation='sigmoid'))
model.add(layers.Dense(1950, activation='tanh'))
model.add(layers.Dense(900, activation='tanh'))
model.add(layers.Dense(36855, activation='tanh'))
```

[17]:
```
from keras import losses
from keras import metrics
from tensorflow.keras import optimizers

model.compile(optimizer=optimizers.RMSprop(learning_rate=0.00001),
              loss=losses.binary_crossentropy,
              metrics=[metrics.binary_accuracy])
```

[18]:
```
# Guardado de las epocas y de los pesos.

from datetime import datetime
from tensorflow import keras
logdir="logs/fit/" + datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = keras.callbacks.TensorBoard(log_dir=logdir)
```

[19]:
```
# Se entrena el modelo y se define el tamaño de procesamiento
```

```
model.fit(ent,sal,epochs=10000,batch_size
=2048,callbacks=[tensorboard_callback])
```

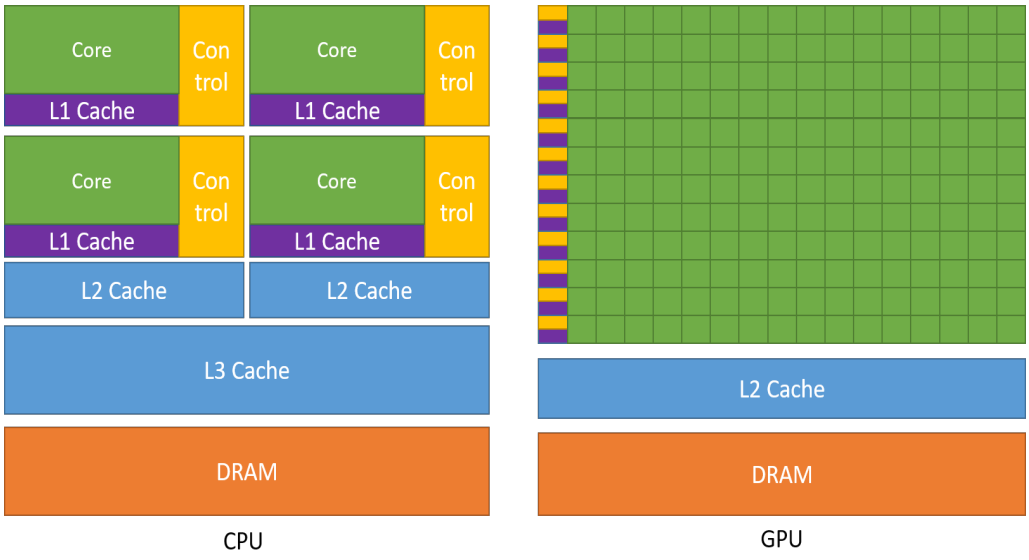
[20]:
```
# Se salva el modelo

model.save('modelrtf.h5')
```

## 8.2. Apendice: CUDA(Compute Unified DeviceArchitecture)

Para poder implementar cualquier código con este lenguaje primero es necesario tener un hardware de tipo GPU. Este tipo de dispositivos realiza cálculos de propósito general de forma paralela, rápida y con mayor capacidad que una CPU. Las GPU compatibles con Cuda deben contar con unidades de ejecución multiprocesadores de transmisión(SM), que se conectan entre sí mediante una memoria cache. Cada SM está compuesto por núcleos denominados transmisión de procesos(SP) o núcleos CUDA (NVIDIA Corporation & affiliates, 2023 [7]).

CUDA es el lenguaje de alto rendimiento que controla el hadware de una GPU facilitando el control de los componentes para una mejor aprovechamiento de los recursos del sistema. Sin embargo antes de programar cualquier rutina es necesario conocer los componentes de nuestra GPU. 8.2.1

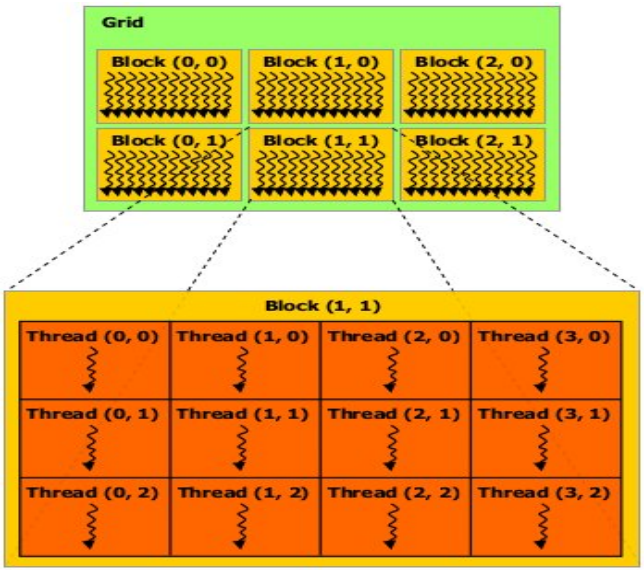
Figura 8.2.1: Diferencia entre una CPU / GPU




Cada procesador de una GPU representado en verde en la imagen anterior esta compuesto de una serie de hilos de procesamiento los cuales tienen la capacidad de ejecutar una función

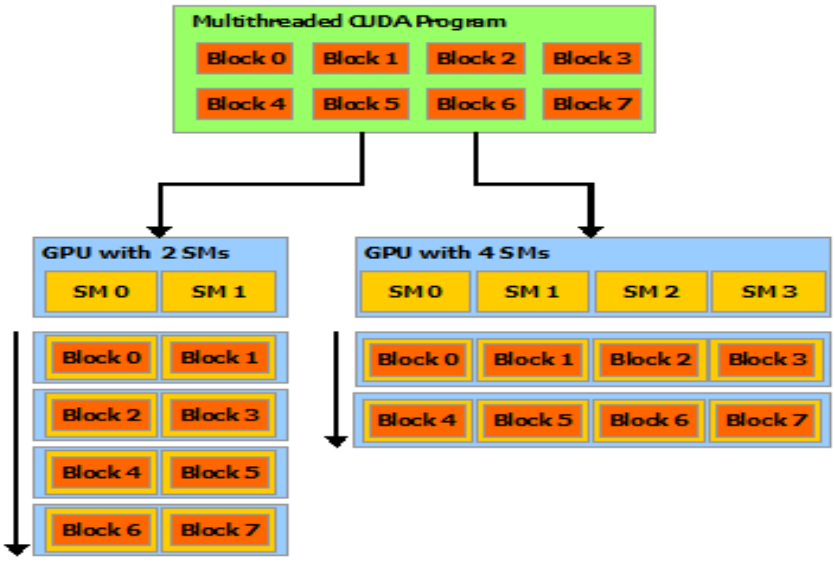
también llamada núcleo. A fin de aumentar la capacidad de procesamiento, es posible agrupar estos procesadores es una serie de bloques y estos a su vez en una cuadricula.

Figura 8.2.2: Cuadricula



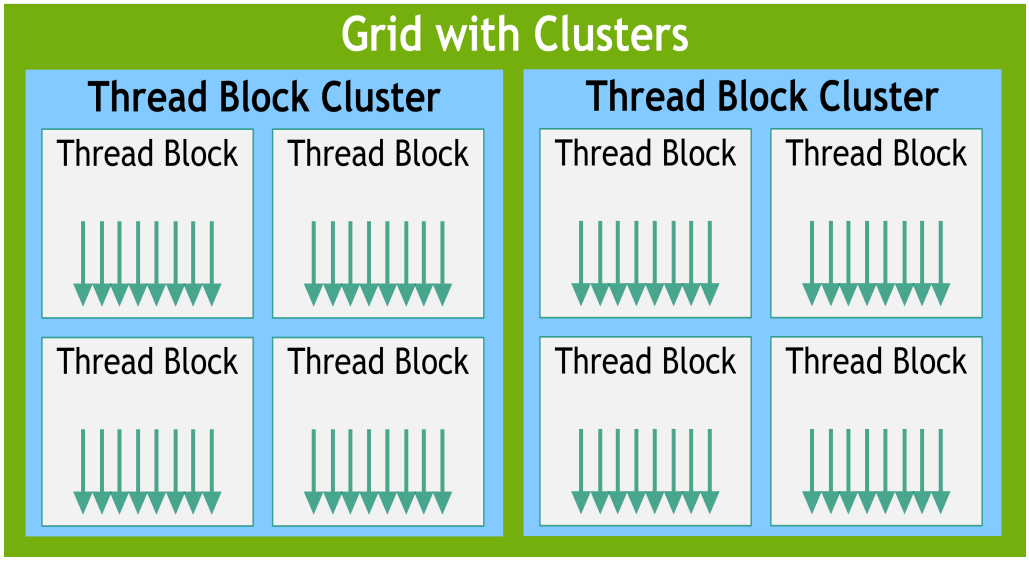

De esta manera es posible escalar el núcleo de la siguiente manera:

Figura 8.2.3: Escalamiento GPU




Al llamar un núcleo CUDA este se ejecutara N veces en paralelo por N subprocesos CUDA diferentes, sin embargo, CUDA nos permite llamar mas de un núcleo obteniendo así el siguiente modelo.

Figura 8.2.4: Grupo de núcleos CUDA




## 8.3. Apendice: especificaciones técnicas del equipo empleado

para poder implementar los códigos generados con anterioridad se empleó un equipo hp con las siguientes características:

- 2 GPU 2080 ti
- 2 GPU 1080 ti
- 90 Gigas de RAM
- 1 procesador xeon
- 1 disco duro de 1tb mecánico

# Bibliografía


[1] Ruben Santamaria O. (2023,12). *Molecular Dynamics*. Springer Cham. Primera edición

[2] Isasi Viñuela P. y Galván León I.M.(2004). *Redes de neuronas artificiales*. Pearson educacion. S.A. Madrid

[3] García Cabello, J. (2002,11). *Mathematical Neural Networks*. Axioms. https://doi.org/10.3390/axioms11020080. [Consultado el 20 de marzo de 2022]

[4] Kiprono Elijah Koech . (2002,07). *How Does Back Propagation Work in Neural Networks?*. https://towardsdatascience.com/how-does-back-propagation-work-in-neural-networks-with-worked-example-bc59dfb97f48. [Consultado el 15 de julio de 2022]

[5] Goldstein .H, Poole .C y Safko .J (2014). *Classical mechanics*. Tercera edición. 150,154

[6] Gonzalez .R y Woods .R. (2018) *Digital image processing*. Pearson. Cuarta edición

[7] NVIDIA Corporation & affiliates. (2023. Noviembre) *CUDA C++ Programming Guide*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#notice. [Consultado el 25 de noviembre de 2023]

[8] I.Foodfellow, Y. Bengio y A. Courville. (2016) *Deep Learning*. MIT press, Cambridge, Reino unido

[9] Stanford University *Guiá de los planes de estudio*. https://cs224d.stanford.edu/syllabus.html. [Consultado el 12 de febrero de 2022]

[10] May, Jeffrey A (06-1997). *Using artificial neural networks to identify unexploded ordnance*. Monterey, Calif. : Naval Postgraduate School ; Springfield. Pag (23-34)

[11] International Work-Conference on Artificial and Natural Neural Networks, Mira, J. Prieto A. (2001). *Connectionist Models of Neutons, Learning Processes and Artifical Intelligence*. Berlin ; New York : Springer. Pag 712-718

[12] Nii O. Attoh-Okine y Bilal M. Ayyub. (2005). *Applied research in uncertainty modeling and analysis*. New York, NY : Springer. Pag(158-164, 226-240)

[13] Harvey, Robert L (1994). *Neural network principles*. Englewood Cliffs, NJ : Prentice Hall. Pag(158-164, 226-240)

[14] Korn, Granino A.(1991). *Neural network experiments on personal computers and workstations*. Cambridge, Mass. : MIT Press. Pag(142-146)

[15] Josh Paterson y Adam Gibson(2017). *Deep learning : a practitioner's approach*. Sebastopol, CA : O'Reilly Media, Inc. Pag(40-80, 124-140)

[16] Raschka, Sebastian(2015). *Python machine learning : unlock deeper insights into machine learning with this vital guide to cutting-edge predictive analytics*. Birmingham, UK : Packt Publishing Ltd. Pag(17-47)

[17] Bernd Jähne(2002). *Digital image processing*. Berlin ; New York : Springer. Pag(328-330)

[18] Scott E. Umbaugh(2011). *Digital image processing and analysis : human and computer vision applications with CVIPtools*. Boca Raton : Taylor & Francis. Pag(50-62, 144-176)

[19] Louis J. Galbiati jr.(1990). *Machine vision and digital image processing fundamentals*. Englewood Cliffs, N.J. : Prentice Hall. Pag(113-123)

[20] Aurélien Géron(2017). *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems*.Sebastopol, CA : O'Reilly Media. Pag(253-272)

[21] Nikhil Buduma(2017). *textsFundamentals of deep learning : designing next-generation machine intelligence algorithms*. Sebastopol, CA : O'Reilly Media. Pag(17-62)

[22] Bernhard Mehlig(2021). *Machine learning with neural networks*. Department of Physics UNIVERSITY OF GOTHENBURG Göteborg, Sweden. Pag(1-12, 73-90)

[23] Martin T. Hagan,Howard B. Demuth,Orlando De Jesús y Mark Hudson Beale (2014). *Neural Network Design*. 2nd Edition, eBook. Pag(36-70). https://hagan.okstate.edu/NNDesign.pdf

[24] LISA Lab(2015). *Deep Learning Tutorials*. University of Montreal. Pag(17-24, 35-62). https://www.cs.virginia.edu/yanjun/teach/2014f/lecture/L20-handout-deeplearningTutorial.pdf

[25] James McCaffrey. (2018). *Keras Succinctly*. Syncfusion Inc.

[26] jeff heaton(2005). *Introduction to Neural Networks with Java*. Heaton Research, Inc. Pag(49-125). https://www.heatonresearch.com/book/

[27] Arnulf Jentzen, Benno Kuckuck y Philippe von Wurstemberger(2023). *Mathematical Introduction to Deep Learning: Methods, Implementations, and Theory*. eBook. Pag(21-29). https://arxiv.org/pdf/2310.20360.pdf

[28] Mukesh D. Patil, Gajanan K. Birajdar y Sangita S Chaudhari (2023). *Computational Intelligence In Image And Video Processing*. Chapman & Hall Book /CRC Press. Pag(1-33)

[29] Yunji Chen, Ling Li, Wei Li, Qi Guo, Zidong Du y Zichen Xu(2017). *AI Computing Systems An Application-Driven Perspective*.Morgan Kaufmann Publishers an imprint of Elsevier.