UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

REDES NEURONALES ARTIFICIALES EN SIMULACIONES DE DINÁMICA MOLECULAR

T E S I S

QUE PARA OBTENER EL TÍTULO DE

F Í S I C A

PRESENTA:

EVELYN SALAZAR FLORES

TUTOR:

DR. JOSÉ GUADALUPE PÉREZ RAMÍREZ

CIUDAD DE MÉXICO 2021

Universidad Nacional
Autónoma de México

Biblioteca Central
Dirección General de Bibliotecas de la UNAM

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

Resumen

En este trabajo se presenta un método para generar la superficie de energía potencial de un sistema atómico a partir de redes neuronales artificiales. El modelo que se utilizó fue el propuesto por Jörg Behler y Michele Parrinello (2007), el cual usa un tipo de coordenadas llamadas funciones de simetría. Posteriormente se utiliza esa superficie para calcular las fuerzas atómicas y con ellas realizar una simulación de dinámica molecular.

Con base en este método de redes neuronales artificiales se creó un programa (escrito con Python y Keras) capaz de llevar a cabo simulaciones de dinámica molecular. Este programa se aplicó en un sistema compuesto por átomos de hidrógeno, oxígeno y aluminio. El código se puede encontrar en un repositorio de github o en un notebook de Kaggle.

Por último se hizo una comparación entre los resultados obtenidos con el programa presentado en este trabajo y los resultados obtenidos con *n2p2*, una paquetería creada por Andreas Singraber (2019) que también aplica redes neuronales artificiales para hacer simulaciones de dinámica molecular.

Índice

1. Introducción **1**

2. Simulaciones de dinámica molecular **3**

2.1. Dinámica molecular *ab initio* . . . 3

2.2. Algoritmo de la velocidad de Verlet . . . 5

2.3. Superficie de energía potencial . . . 7

3. Redes neuronales artificiales **9**

3.1. Aprendizaje automatizado y redes neuronales artificiales . . . 9

3.2. Modelo matemático de una red neuronal artificial . . . 11

3.3. Entrenamiento de una red neuronal artificial . . . 13

4. Potencial generado con una red neuronal artificial **19**

4.1. Creación del conjunto de datos . . . 19

4.2. Funciones de simetría . . . 20

4.3. Arquitectura de la red neuronal artificial . . . 26

5. Dinámica molecular con redes neuronales **28**

5.1. Cálculo de las fuerzas atómicas . . . 28

5.2. Derivada de la energía respecto a las funciones de simetría . . . 32

5.3. Derivada de las funciones de simetría respecto a las coordenadas cartesianas . . . 34

6. Código **37**

6.1. Código para el entrenamiento de la red neuronal . . . 39

6.2. Código para las simulaciones de dinámica molecular . . . 44

6.3. n2p2 y LAMMPS . . . 49

7. Conclusión y trabajo futuro **52**

8. Apéndice **54**

8.1. Formato de los archivos *.smf* 54

8.2. Instalación de LAMMPS y n2p2 55

1. Introducción

Desde hace tiempo las simulaciones computacionales se han convertido en una parte integral de la ciencia, junto al desarrollo de modelos matemáticos y la experimentación con la naturaleza. La importancia de las simulaciones radica en que con ellas es posible estudiar fenómenos que por sus condiciones no podrían ser analizados directamente a través de experimentos (Griebel et al., 2007).

En el caso particular de la dinámica molecular, las simulaciones pueden realizarse a partir de cálculos cuánticos *ab initio* (Car & Parrinello, 1985). Con este método se pueden estudiar procesos físicos y químicos con una gran precisión, pero el costo computacional de resolver estos cálculos es bastante elevado. Esto significa que para llevar a cabo una simulación *ab initio* se requiere un sistema computacional potente y una enorme cantidad de memoria, y aún si se cumple con estos requerimientos puede que las simulaciones estén limitadas a sistemas atómicos pequeños (Likun Tan, 2016). Otro método para llevar a cabo una simulación de dinámica molecular es suponer que el sistema actúa bajo el efecto de un campo de fuerza representado por una función paramétrica, por lo que en este caso únicamente se deben hallar los parámetros con los que la función se ajusta mejor a los datos experimentales (Scheerschmidt, 2006). Este método tiene un costo computacional menor pero tiene la desventaja de que los resultados no son tan precisos.

Una alternativa que tiene un costo computacional menor al de las simulaciones *ab initio* y que al mismo tiempo ofrece precisión en los resultados son las simulaciones que incorporan redes neuronales artificiales en sus cálculos. Desde un punto de vista muy general, una red neuronal artificial se puede describir como una función de ajuste no lineal altamente flexible, por lo que en principio se podría ajustar a cualquier función continua (Hornik et al., 1989). En el caso de las simulaciones de dinámica molecular la red neuronal se deberá ajustar a la superficie de energía potencial del sistema atómico bajo estudio, la cual es una técnica que se ha utilizado desde hace tiempo para estudiar sistemas moleculares (Blank et al., 1995) y que recientemente se aplica en el estudio y desarrollo de nuevas sustancias y materiales (Elias et al., 2016).

En la primera parte de este trabajo se aborda la teoría detrás de las simulaciones de dinámica molecular *ab initio*, seguida de la explicación del modelo detrás de las redes neuronales artificiales y del algoritmo de entrenamiento cuyo objetivo es conseguir que la red neuronal ”aprenda” a partir de un conjunto de datos. En la siguiente parte se analizará un modelo de redes neuronales artificiales capaz de generar la superficie de energía potencial de un sistema atómico a partir de las funciones de simetría (Behler & Parrinello, 2007), que son un tipo de coordenadas que dependen de las posiciones de los átomos, y después se mostrarán los cálculos necesarios para obtener las fuerzas atómicas a partir de la superficie de energía potencial con el fin de crear una simulación de dinámica molecular (Singraber et al., 2019). Por último se presentará un código escrito en Python (y Keras para las redes neuronales artificiales) donde se incorporan todos estos conceptos y cálculos para generar la superficie de energía potencial de un sistema compuesto por

átomos de aluminio, hidrógeno y oxígeno.

2. Simulaciones de dinámica molecular

2.1. Dinámica molecular *ab initio*

De acuerdo con las leyes de Newton, si a un cuerpo se le aplica una fuerza lo suficientemente grande entonces la posición q de ese cuerpo va a cambiar conforme pase el tiempo. Esto mismo se puede extender a un sistema o ensamble integrado por múltiples cuerpos, en cuyo caso van a cambiar las posiciones de los cuerpos sobre los que se ejerza alguna fuerza. Por esta razón es común que para describir un sistema de cuerpos o partículas en un tiempo determinado t se utilice al conjunto de todas las posiciones $\{q\}$. Se dice que cada conjunto $\{q\}$ determina un estado o configuración del sistema.

Por otro lado, una simulación es un algoritmo que se utiliza para modelar la evolución de un sistema a partir de un estado inicial (Allen, 2004). Para describir la evolución de un sistema de partículas se requieren tres componentes: **(a)** la posición y velocidad de todas las partículas en un tiempo determinado, **(b)** unas ecuaciones de movimiento que describan cómo cambian las posiciones con el paso del tiempo, y **(c)** un método de integración para resolver esas ecuaciones.

En el caso de un sistema atómico cada configuración del sistema queda completamente determinada por las posiciones de todos los electrones y núcleos que lo componen. La evolución de este conjunto de átomos se puede determinar a través de la ecuación de Schródinger (Griffiths, 2004), que para una sola partícula que se mueve en una dimensión tiene la siguiente forma:

$$\left(-\frac{\hbar^2}{2m}\frac{d^2}{dx^2}+V(x,t)\right)\Psi = i\hbar\frac{d\Psi}{dt}, \tag{1}$$

donde Ψ es la función de onda que representa a la partícula, $\hbar$ es la constante de Planck reducida y m es la masa de dicha partícula.

En el lado izquierdo de la ecuación (1) se encuentra la suma de dos términos: el primero es un operador diferencial que determina la energía cinética de la partícula y el segundo término $V(x,t)$ es la energía potencial que actúa sobre la partícula. A esta suma se le conoce comúnmente como Hamiltoniano y se denota como H (Jolicard, 1995). Si la partícula se mueve en un espacio tridimensional entonces el Hamiltoniano correspondiente es:

$$H = -\frac{\hbar^2}{2m}\left(\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}+\frac{\partial^2}{\partial z^2}\right)+V(\vec{x},t) = -\frac{\hbar^2}{2m}\nabla^2+V(\vec{x},t) \tag{2}$$

Como el Hamiltoniano de la ecuación (2) es lineal, se puede asumir que la energía total de un sistema de partículas es igual a la suma de las energías de

todas las partículas. Por lo tanto, el Hamiltoniano de un sistema compuesto por múltiples electrones y núcleos es (Griebel et al., 2007):

$$\begin{aligned} H = & -\frac{\hbar^2}{2m_e}\sum_{i=1}^{n}\nabla^2_{\mathbf{r_i}} - \frac{\hbar^2}{2}\sum_{i=1}^{N}\frac{1}{M_i}\nabla^2_{\mathbf{R_i}} \\ & + \frac{e^2}{4\pi\epsilon_0}\sum_{i=1}^{n}\sum_{j>i}^{n}\frac{1}{r_{ij}} + \frac{e^2}{4\pi\epsilon_0}\sum_{i=1}^{N}\sum_{j>i}^{N}\frac{Z_iZ_j}{R_{ij}} - \frac{e^2}{4\pi\epsilon_0}\sum_{i=1}^{n}\sum_{j=1}^{N}\frac{Z_j}{||\mathbf{r_i}-\mathbf{R_j}||}, \end{aligned} \tag{3}$$

donde m_e es la masa de un electrón, n es el número total de electrones en el sistema, $\nabla^2_{\mathbf{r_i}}$ es el Laplaciano respecto a la posición del i-ésimo electrón y $r_{ij} = ||\mathbf{r_i} - \mathbf{r_j}||$ es la distancia entre el electrón-i y el electrón-j. Asimismo, M_i y Z_i son la masa y el número atómico del i-ésimo núcleo respectivamente, N es el número total de núcleos en el sistema, $\nabla^2_{\mathbf{R_i}}$ es el Laplaciano respecto a la posición del i-ésimo núcleo y $R_{ij} = ||\mathbf{R_i} - \mathbf{R_j}||$ es la distancia entre el núcleo-i y el núcleo-j.

El primer par de términos en la ecuación (3) son la energía cinética electrónica y la energía cinética nuclear respectivamente, luego están los términos de repulsión entre electrones y repulsión entre núcleos, y por último la atracción entre electrones y núcleos.

Resolver la ecuación de Schrödinger con el Hamiltoniano de la ecuación (3) es un problema bastante difícil. Se puede hallar una solución analítica para el átomo de hidrógeno (compuesto por un electrón y un núcleo) (Griffiths, 2004), pero para sistemas con más partículas el problema se vuelve tan complejo que hallar una solución analítica es casi imposible. Por esta razón se acostumbra resolver la ecuación de Schrödinger de manera numérica, sin embargo este método está limitado a sistemas atómicos pequeños porque el costo computacional es considerable.

Para aligerar un poco los cálculos es necesario hacer aproximaciones o suposiciones que simplifiquen el problema. En el caso de los sistemas atómicos es común considerar la aproximación de Born-Oppenheimer (Born & Oppenheimer, 1927), de acuerdo con la cual el movimiento de los núcleos respecto a los electrones es casi nulo. La justificación física tras esta suposición es que la masa de los núcleos es mucho mayor que la masa de los electrones, por lo que el movimiento de los núcleos respecto al de los electrones es tan lento que pareciera que permanecen fijos. Como resultado de esta aproximación la ecuación de Schrödinger se puede separar en una parte que depende únicamente de los núcleos y otra parte que depende sólo de los electrones. En este caso la ecuación de Schrödinger nuclear se reemplaza por la segunda ley de Newton y el efecto de los electrones se incluye en un potencial efectivo que actúa sobre los núcleos (Griebel et al., 2007).

La energía potencial que se obtiene al separar la parte electrónica de la parte nuclear está dada por la solución de la ecuación de Schrödinger electrónica, y para llegar a esa solución se pueden aplicar varias técnicas. Una de las más utilizadas es aproximarse a la solución a través de la teoría del funcional de densidad o teoría DFT (*Density Functional Theory*) (Kohn & Sham, 1965).

2.2. Algoritmo de la velocidad de Verlet

Una vez que se ha determinado la energía potencial total del sistema atómico, lo siguiente es resolver la ecuación de Newton de los núcleos. Esto quiere decir que se debe resolver un sistema de ecuaciones diferenciales de segundo orden. Sin embargo, en una simulación no es posible resolver estas ecuaciones cada vez que transcurre una cantidad infinitesimal de tiempo por el costo computacional que esto implicaría. En su lugar lo que se hace es discretizar el tiempo, por lo que ahora las ecuaciones se deberán resolver cada vez que transcurra un intervalo de tiempo δt (Allen, 2004).

Así pues, para una simulación de dinámica molecular se debe definir algún algoritmo con el que a partir de las posiciones, velocidades y fuerzas atómicas en un tiempo t se puedan calcular las mismas variables para el tiempo $t + \delta t$. Uno de los algoritmos más utilizados es el algoritmo de la velocidad de Verlet (Omelyan et al., 2007).

Supóngase que la posición de una partícula-i en el tiempo t_n está dada por la función $\mathbf{r_i}(t_n)$. Para calcular cuál será la posición de la partícula después de un intervalo temporal δt se hace una expansión de Taylor:

$$\mathbf{r_i}(t_n + \delta t) = \mathbf{r_i}(t_n) + \delta t \frac{d\mathbf{r_i}}{dt}(t_n) + \frac{\delta t^2}{2}\frac{d^2\mathbf{r_i}}{dt^2}(t_n) + O(\delta t^3) \tag{4}$$

La primera derivada de $\mathbf{r_i}$ es igual a la velocidad $\mathbf{v_i}$ de la partícula, y de la segunda ley de Newton se tiene que

$$\mathbf{F_i} = m_i \frac{d^2\mathbf{r_i}}{dt^2} \Rightarrow \frac{d^2\mathbf{r_i}}{dt^2} = \frac{\mathbf{F_i}}{m_i} \tag{5}$$

donde m_i es la masa de la partícula.

Si se sustituyen estas variables en la ecuación (4) entonces se obtiene una expresión donde $\mathbf{r_i}$ en el tiempo $t_n + \delta t$ queda determinada por la posición, velocidad y fuerza en el tiempo t_n:

$$\mathbf{r_i}(t_n + \delta t) = \mathbf{r_i}(t_n) + \delta t \mathbf{v_i}(t_n) + \frac{\delta t^2}{2}\frac{\mathbf{F_i}}{m_i}(t_n) + O(\delta t^3) \tag{6}$$

Lo siguiente es hallar una expresión para $\mathbf{v_i}(t_n + \delta t)$.

Las expansiones de Taylor de $\mathbf{r}_i(t_n + \delta t)$ y $\mathbf{r}_i(t_n - \delta t)$ hasta segundo orden en la notación de Newton son:

$$\mathbf{r}_i(t_n + \delta t) = \mathbf{r}_i(t_n) + \dot{\mathbf{r}}_i(t_n)\delta t + \frac{\ddot{\mathbf{r}}_i(t_n)}{2}\delta t^2 \tag{7}$$

$$\mathbf{r}_i(t_n - \delta t) = \mathbf{r}_i(t_n) - \dot{\mathbf{r}}_i(t_n)\delta t + \frac{\ddot{\mathbf{r}}_i(t_n)}{2}\delta t^2 \tag{8}$$

Si se suman las dos ecuaciones anteriores y se despeja a $\mathbf{r}_i(t_n + \delta t)$ se llega a que

$$\mathbf{r}_i(t_n + \delta t) = 2\mathbf{r}_i(t_n) - \mathbf{r}_i(t_n - \delta t) + \ddot{\mathbf{r}}_i(t_n)\delta t^2 \tag{9}$$

Si ahora a la ecuación (8) se le resta la ecuación (7) y se despeja a $\mathbf{r}_i(t_n + \delta t)$ se obtiene la siguiente expresión

$$\mathbf{r}_i(t_n + \delta t) = \mathbf{r}_i(t_n - \delta t) + 2\dot{\mathbf{r}}_i(t_n)\delta t \tag{10}$$

Al sustituir la ecuación (9) en la ecuación (10) y despejar a $\dot{\mathbf{r}}_i(t_n)\delta t$ se obtiene que

$$\dot{\mathbf{r}}_i(t_n)\delta t = \mathbf{r}_i(t_n) - \mathbf{r}_i(t_n - \delta t) + \frac{\ddot{\mathbf{r}}_i(t_n)}{2}\delta t^2 \tag{11}$$

La ecuación anterior en el tiempo $t + \delta t$ es:

$$\dot{\mathbf{r}}_i(t_n + \delta t)\delta t = \mathbf{r}_i(t_n + \delta t) - \mathbf{r}_i(t_n) + \frac{\ddot{\mathbf{r}}_i(t_n + \delta t)}{2}\delta t^2 \tag{12}$$

La suma de las ecuaciones (11) y (12) es

$$\dot{\mathbf{r}}_i(t_n)\delta t + \dot{\mathbf{r}}_i(t_n + \delta t)\delta t = \mathbf{r}_i(t_n + \delta t) - \mathbf{r}_i(t_n - \delta t) + \frac{\ddot{\mathbf{r}}_i(t_n) + \ddot{\mathbf{r}}_i(t_n + \delta t)}{2}\delta t^2 \tag{13}$$

$$\Rightarrow \dot{\mathbf{r}}_i(t_n + \delta t)\delta t = -\dot{\mathbf{r}}_i(t_n)\delta t + \mathbf{r}_i(t_n + \delta t) - \mathbf{r}_i(t_n - \delta t) + \frac{\ddot{\mathbf{r}}_i(t_n) + \ddot{\mathbf{r}}_i(t_n + \delta t)}{2}\delta t^2 \tag{14}$$

Si en esta última expresión se sustituye a la ecuación (10) se llega a que

$$\dot{\mathbf{r}}_i(t_n + \delta t)\delta t = \dot{\mathbf{r}}_i(t_n)\delta t + \frac{\ddot{\mathbf{r}}_i(t_n) + \ddot{\mathbf{r}}_i(t_n + \delta t)}{2}\delta t^2 \tag{15}$$

Por lo tanto, la velocidad $\mathbf{v_i}$ en el tiempo $t_n + \delta t$ es

$$\mathbf{v_i}(t_n + \delta t) = \mathbf{v_i}(t_n) + \frac{\mathbf{F_i}(t_n) + \mathbf{F_i}(t_n + \delta t)}{2m_i}\delta t \tag{16}$$

Las ecuaciones (6) y (16) constituyen el algoritmo de la velocidad de Verlet.

2.3. Superficie de energía potencial

Hasta ahora se tiene un algoritmo con el que se pueden actualizar las posiciones y velocidades de todas las partículas en un sistema después de transcurrido un tiempo δt (Allen, 2004). Esa actualización está dada por las ecuaciones (6) y (16), y lo que se puede observar es que tanto $\mathbf{r_i}$ como $\mathbf{v_i}$ dependen de la fuerza $\mathbf{F_i}$. Lo que se va a discutir a continuación es un método para calcular las fuerzas atómicas a partir de la energía potencial del sistema (Lewars, 2011).

La configuración de un sistema está determinada por la posición de todas las partículas que lo constituyen, por lo que en el caso de un sistema atómico cada configuración depende de la posición de todos los electrones y neutrones en un tiempo determinado. Anteriormente se vio que para facilitar los cálculos se considera la aproximación de Born-Oppenheimer (Born & Oppenheimer, 1927), con la cual se separa el movimiento de los núcleos del movimiento de los electrones. Por esta razón de ahora en adelante sólo se van a considerar las coordenadas nucleares. Así pues, la configuración de un sistema compuesto por N átomos se define como el conjunto de todas las posiciones nucleares:

$$Q = \{\mathbf{r}_i\}_{i\in[1,N]}, \tag{17}$$

donde $\mathbf{r}_i = (x_i, y_i, z_i)$ es la posición del i-ésimo núcleo en coordenadas cartesianas.

Por otro lado, a cada una de las configuraciones posibles del sistema le corresponde un valor escalar E denominado como energía potencial del sistema o energía potencial total. Esto quiere decir que la energía potencial total es una función que depende de todas las coordenadas nucleares:

$$E = E(Q) \tag{18}$$

Geométricamente, la función E se puede representar como una hipersuperficie conocida como superficie de energía potencial (Figura 1), donde cada punto sobre esta superficie está relacionado con un único conjunto de coordenadas nucleares.

A partir de esta superficie de energía se puede obtener la fuerza $\mathbf{F_i}$ que actúa sobre el i-ésimo átomo del sistema, para lo cual se debe calcular el vector gradiente de la energía potencial E:

$$\mathbf{F_i} = -\nabla_i E \tag{19}$$

donde $\nabla_i E$ es el gradiente de la energía respecto a la posición del átomo-i.

Por lo tanto, el problema de calcular las fuerzas atómicas se reduce a encontrar algún método con el que se pueda obtener la superficie de energía potencial (Unke et al., 2020). Algunos de estos métodos son aplicar la teoría DFT (Kohn & Sham, 1965) o resolver la ecuación de Schrödinger (Griffiths, 2004) de manera numérica con lo cual se puede alcanzar una gran precisión, pero resolver esos cálculos cuánticos tiene un costo computacional demasiado elevado y sólo es accesible para sistemas pequeños y para simulaciones de tiempos cortos (Likun Tan, 2016).

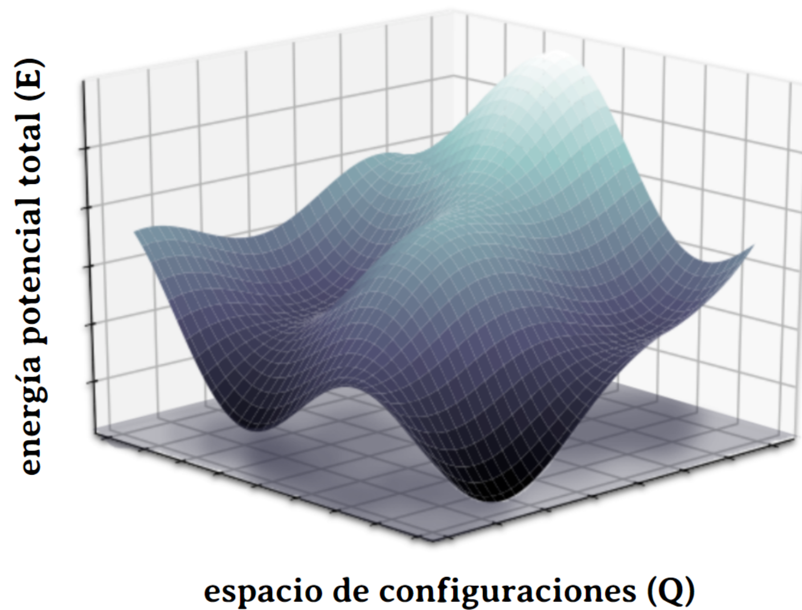


Figura 1: Visualización de la superficie de energía potencial de un sistema atómico.

Otra alternativa es suponer que la energía potencial obedece una función paramétrica, en cuyo caso sólo se tendrían que hallar los parámetros con los que la función se aproxime más a las mediciones físicas. Este método tiene un costo computacional menor que el de los cálculos cuánticos, pero se pierde precisión.

Lo ideal sería encontrar alguna manera con la que se pueda determinar la superficie de energía potencial que no esté limitada a sistemas pequeños y tiempos cortos, pero que al mismo tiempo dé resultados precisos. Una forma en la que esto puede ser posible es a través de la generación de superficies de energía potencial a partir de redes neuronales artificiales (Behler & Parrinello, 2007), la cual se discutirá más adelante. Pero para saber cómo se hace esa generación de la superficie de energía primero es necesario entender qué son las redes neuronales artificiales y cómo funcionan.

3. Redes neuronales artificiales

3.1. Aprendizaje automatizado y redes neuronales artificiales

La inteligencia artificial es el área de la Informática que busca la creación de máquinas que puedan imitar comportamientos inteligentes, y al campo dentro de la inteligencia artificial que se dedica a desarrollar algoritmos que doten a las máquinas de la capacidad de aprendizaje se le conoce como aprendizaje automatizado (Wang & Siau, 2019). Dentro del aprendizaje automatizado existen diversas técnicas que pueden clasificarse en tres grupos: aprendizaje supervisado, aprendizaje no supervisado y aprendizaje reforzado (Zhang et al., 2020).

El aprendizaje supervisado (Mueller et al., 2016) consiste en hacer que un modelo encuentre cuál es la relación entre unas variables independientes (también llamadas rasgos o características) y una variable dependiente (también llamada etiqueta). Si el modelo logra encontrar esa relación entonces podrá predecir la etiqueta de cualquier conjunto de rasgos, incluso si el modelo se aplica sobre datos que no se utilizaron durante su aprendizaje. Una de las técnicas principales del aprendizaje supervisado es la de las redes neuronales artificiales (Ng, 2021), la cual se basa en gran medida en el funcionamiento de las redes neuronales dentro del cerebro.

Las neuronas son células compuestas por un núcleo central del que salen varias extremidades o ramificaciones llamadas dendritas y axones, y la manera en la que las neuronas se comunican entre ellas es a través de impulsos eléctricos que pasan de neurona a neurona a través de estas ramificaciones (Zhang, 2019). Como se puede observar en la Figura 2, el impulso eléctrico entra por una de las dendritas, luego se procesa en el núcleo de la neurona y después sale por el axón hacia otras neuronas.

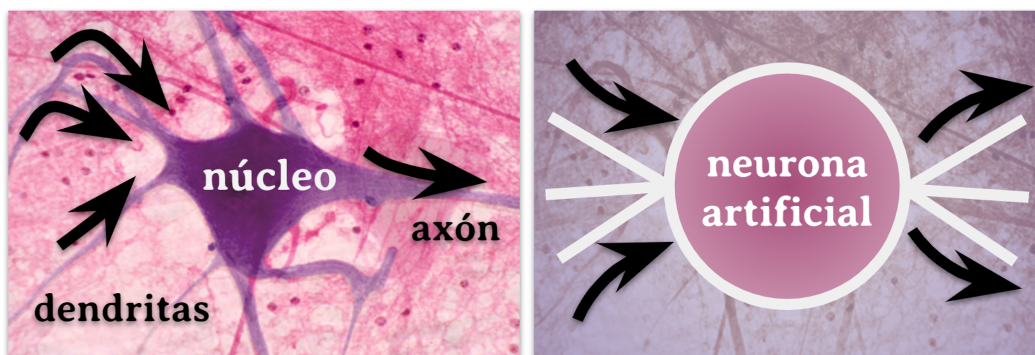


Figura 2: Comparación entre una neurona biológica y una neurona artificial.

De manera análoga al caso biológico, una red neuronal artificial se refiere a un conjunto estructurado de neuronas artificiales, donde cada neurona artificial se

representa como un nodo del que salen conexiones que transfieren información hacia otras neuronas. En cada neurona artificial se llevan a cabo cálculos con los datos que ingresaron por un lado, y después se transmite el resultado hacia las neuronas artificiales que están en el lado opuesto. De ahora en adelante se referirá a las neuronas artificiales únicamente como neuronas.

Las neuronas de una red neuronal artificial generalmente se acomodan por capas. A la primer capa de una red se le denomina como capa de entrada, y es ahí donde se ingresan los rasgos o variables de entrada del conjunto de datos. A la última capa de la red se le conoce como capa de salida y de ahí sale el resultado final, el cual se denomina como inferencia o predicción. Normalmente entre la capa de entrada y la capa de salida hay más capas llamadas capas ocultas (Zhang et al., 2020).

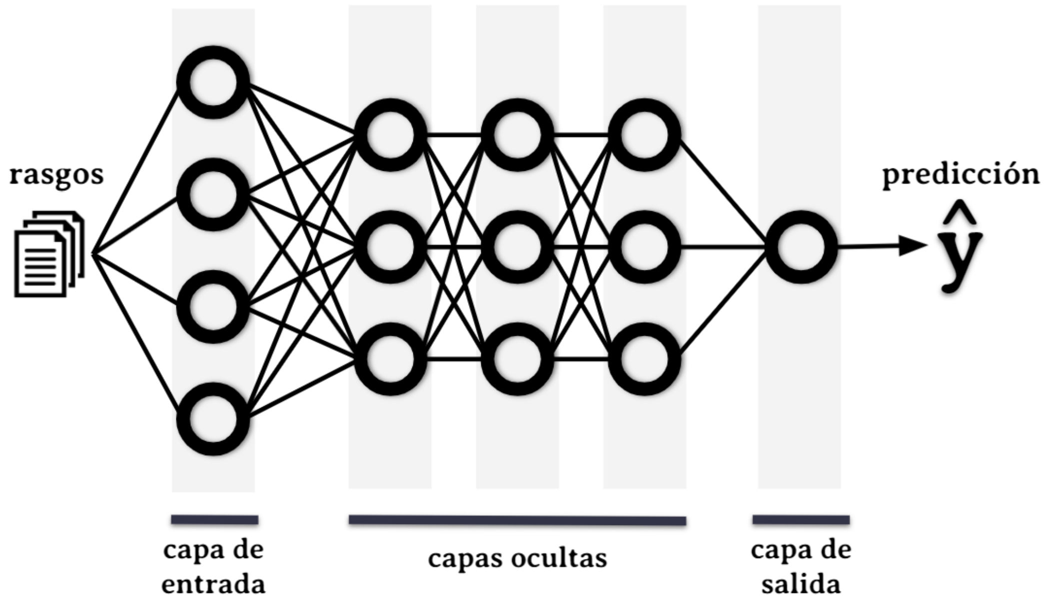


Figura 3: Estructura de una red neuronal artificial. Los rasgos se introducen en la capa de entrada y el resultado generado por la red sale por la capa de salida.

En la Figura 3 se muestra la estructura de una red artificial, y lo que se puede observar es que la información que contiene cada neurona de la capa de entrada se transmite a todas las neuronas de la siguiente capa. La conexión que hay entre cada par de neuronas tiene un valor asociado que determina cuánta importancia tiene esa conexión para la red en su totalidad. Una vez que una neurona ha recibido toda la información de la capa anterior, la neurona realiza algún tipo de transformación sobre esa información y transfiere el resultado a todas las neuronas de la siguiente capa. El mismo proceso se repite hasta llegar a la capa de salida, de donde sale la predicción de la red neuronal (Zhang et al., 2020).

3.2. Modelo matemático de una red neuronal artificial

En realidad, las redes neuronales artificiales son modelos matemáticos cuya base se encuentra en el modelo de regresión lineal. Los modelos de regresión se utilizan para encontrar la relación entre una variable que puede ser multidimensional y un valor numérico. Dicho de otra forma, el modelo de regresión es una función $f : \mathbb{R}^n \to \mathbb{R}$ con la que se puede aproximar la relación que hay entre un vector x y un escalar y. En un modelo de regresión lineal se supone que f es la combinación lineal de todas las componentes x_i del vector x:

$$\hat{y} := f(\mathbf{x}) = w_1 x_1 + ... + w_n x_n + b = \mathbf{w} \cdot \mathbf{x} + b, \tag{20}$$

donde $\hat{y}$ es la estimación del modelo.

Los términos $\mathbf{w} = (w_1, ..., w_n)$ en la ecuación anterior se conocen como pesos y determinan la importancia que cada componente de x tiene sobre $\hat{y}$. El coeficiente b se denomina como bias o valor de sesgo, e indica el valor que $\hat{y}$ debería tener si todas las componentes x_i fueran iguales a 0.

Este modelo se puede representar con una red neuronal artificial (Zhang et al., 2020) compuesta por una capa de entrada con n neuronas y una capa de salida con una sola neurona, tal y como se puede observar en la Figura 4. En este caso, las componentes del vector $\mathbf{x} = (x_1, ..., x_n)$ son los rasgos que se ingresan en la capa de entrada de la red; a cada neurona le corresponde un sólo rasgo. Además a la conexión que hay entre la i-ésima neurona de entrada y la neurona de salida se le asocia el peso w_i. Finalmente, en la neurona de la capa de salida se calcula el producto interno $\| \cdot \|$ entre los valores de entrada x y los pesos w, más el bias b, con lo cual se llega a que la predicción de la red neuronal es la estimación $\hat{y}$ del modelo de la ecuación (20).

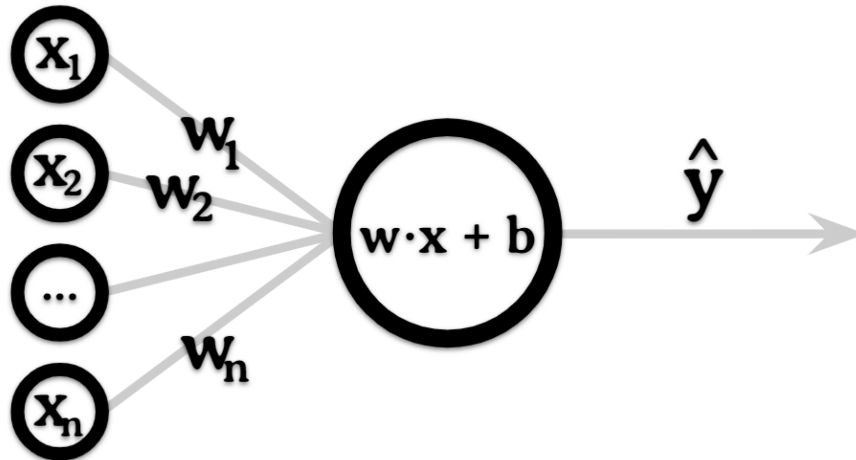


Figura 4: El modelo de regresión lineal aplicado en una red neuronal artificial.

Con la red neuronal de la Figura 4 se pueden hacer predicciones precisas si la relación entre las variables independientes x y la variable dependiente y a la

que el modelo se debe aproximar es lineal. Sin embargo, la mayoría de las veces la relación que hay en un grupo de variables no es tan simple. Para que este modelo pueda ajustarse a funciones más complejas que la lineal es necesario agregar otro elemento al modelo: una función de activación (Feng & Lu, 2019).

Las funciones de activación a son transformaciones no lineales que se aplican sobre la combinación lineal $\mathbf{w} \cdot \mathbf{x} + b$. Para simplificar la notación, se introduce la siguiente variable:

$$z = \mathbf{w} \cdot \mathbf{x} + b \tag{21}$$

Algunas de las funciones de activación más populares en el campo del aprendizaje automático son la función sigmoide σ, la tangente hiperbólica y la función ReLU (Goodfellow et al., 2016):

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{22}$$

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{23}$$

$$ReLU(z) = \text{máx}\left(0, z\right) \tag{24}$$

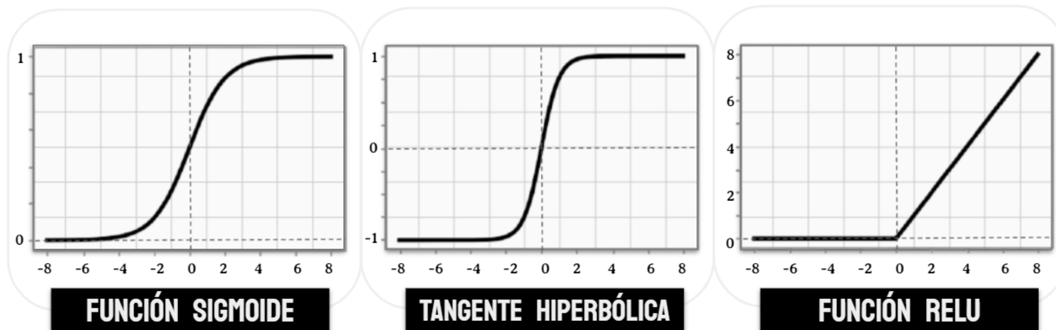


Figura 5: Algunos ejemplos de funciones de activación.

En la Figura 5 se muestran las gráficas de estas funciones de activación. Algunas de las características que se pueden observar son:

- La función sigmoide toma cualquier número real y lo manda a algún número contenido en el intervalo (0,1). Esta función es muy utilizada cuando se necesita interpretar la predicción $\hat{y}$ como una probabilidad, por ejemplo en un clasificador binario.
- La tangente hiperbólica es parecida a la función sigmoide, con la diferencia de que para ésta el rango es (-1,1).
- La función ReLU (*Rectified Linear Unit*) se comporta como la función lineal cuando $z > 0$ y los valores negativos $z < 0$ los manda al cero.

Por lo tanto, ahora el resultado de la red neuronal está dado por la siguiente expresión:

$$\hat{y} = a(z) = a(\mathbf{w} \cdot \mathbf{x} + b) \tag{25}$$

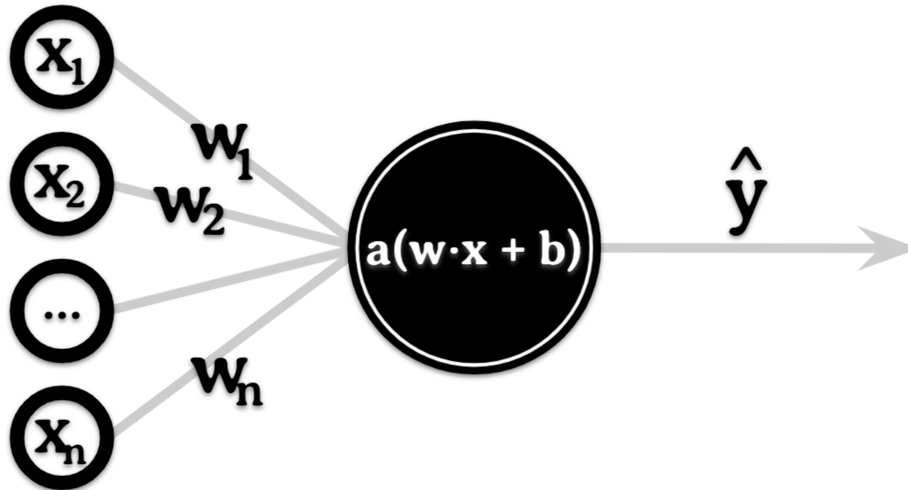


Figura 6: La función de activación se aplica en la neurona después de haber calculado la combinación lineal.

La red neuronal de la Figura 6 es una representación del modelo de regresión lineal con la función de activación a, donde se observa que a se aplica en el mismo nodo donde se lleva a cabo la combinación lineal $\mathbf{w} \cdot \mathbf{x} + b$.

El siguiente paso es elegir el valor de los pesos $\mathbf{w}$ y el bias b a través de un proceso de optimización conocido como entrenamiento.

3.3. Entrenamiento de una red neuronal artificial

En el aprendizaje supervisado es necesario contar con una colección de ejemplos o muestras a partir de las cuales la red neuronal deberá "aprender". Cada muestra está compuesta por un conjunto de rasgos ($\mathbf{x}$) y una etiqueta (y), donde los rasgos son características o propiedades que describen algún objeto o evento y la etiqueta es una variable que depende de esas características (Zhang et al., 2019).

Ya que se tiene un conjunto de datos, se toma una de las muestras y entonces se pasa el conjunto de rasgos $\mathbf{x}^{(\mathbf{i})}$ por la red neuronal para obtener una predicción $\hat{y}^{(i)}$, que en el caso de una red que sólo tiene una capa de entrada y una capa de salida está dada por la siguiente ecuación:

$$\hat{y}^{(i)} = a(\mathbf{w} \cdot \mathbf{x}^{(i)} + b) \tag{26}$$

Lo que se tiene que hacer ahora es construir un algoritmo que dé como resultado el conjunto de parámetros $\{\mathbf{w}, b\}$ con los cuales la predicción $\hat{y}^{(i)}$ será lo más parecida a la etiqueta $y^{(i)}$. Este algoritmo de optimización, comúnmente conocido como entrenamiento de la red neuronal, es un proceso iterativo en el que cada iteración se divide en tres etapas: **(a)** propagación hacia delante, **(b)** propagación hacia atrás y **(c)** actualización de los parámetros.

En la propagación hacia delante se selecciona una muestra $\{\mathbf{x}^{(i)}, y^{(i)}\}$ y se transmite a $\mathbf{x}^{(i)}$ desde la capa de entrada de la red neuronal hasta la capa de salida para obtener la predicción $\hat{y}^{(i)}$.

Una vez hecho esto, lo siguiente es medir de alguna manera qué tan diferente es la predicción $\hat{y}^{(i)}$ de la etiqueta $y^{(i)}$. Por lo tanto lo que se tiene que hacer es elegir una función de pérdida, que es una métrica que mida la distancia que hay entre esas dos variables. Otra manera de interpretar a la función de pérdida es que ésta es un función que mide la probabilidad de que $\hat{y}^{(i)}$ sea igual a $y^{(i)}$.

Una de las funciones de pérdida más utilizadas es la del cuadrado de la distancia entre la etiqueta y la predicción:

$$l_i = \frac{1}{2}\left(\hat{y}^{(i)} - y^{(i)}\right)^2 \tag{27}$$

cuya representación gráfica se puede observar en la Figura 7.

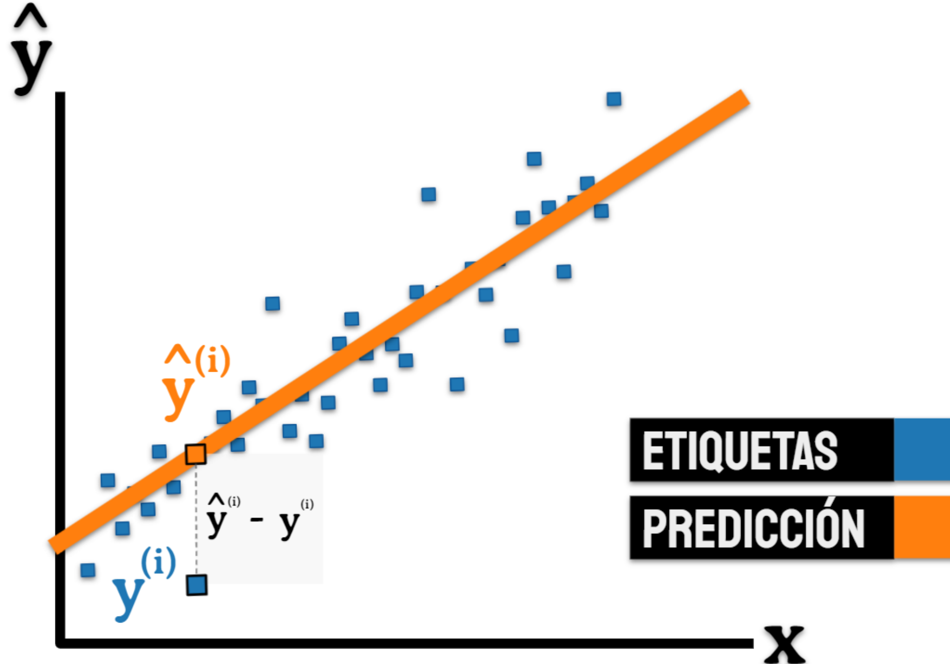


Figura 7: El cuadrado de la distancia entre la etiqueta de una muestra y su predicción es una función de pérdida.

La propagación hacia delante y el cálculo de la función de pérdida es un proceso que se debe llevar a cabo sobre cada una de las muestras del conjunto de datos. Una vez hecho esto, el siguiente paso es calcular el promedio de las funciones de pérdida de todas las muestras:

$$\begin{aligned} J = \frac{1}{m}\sum_{i=1}^{m} l_i &= \frac{1}{m}\sum_{i=1}^{m}\frac{1}{2}\big(\hat{y}^{(i)} - y^{(i)}\big)^2 \\ &= \frac{1}{m}\sum_{i=1}^{m}\frac{1}{2}\Big(a\big(z^{(i)}\big) - y^{(i)}\Big)^2 \\ &= \frac{1}{m}\sum_{i=1}^{m}\frac{1}{2}\Big(a\big(\mathbf{w}\cdot\mathbf{x}^{(i)} + b\big) - y^{(i)}\Big)^2 \end{aligned} \tag{28}$$

Al promedio de las funciones de pérdida se le conoce como función de costo J aunque también se le suele llamar simplemente como costo, y esa cantidad se puede interpretar como el error del modelo de la red neuronal. Por lo tanto, para conseguir las predicciones más precisas lo que se debe hacer es encontrar el conjunto de parámetros $\{\mathbf{w}, b\}$ con el que el costo alcance su valor mínimo. Por lo general en un inicio a $\mathbf{w}$ y b se les otorgan valores aleatorios, así que es de esperar que inicialmente el costo J sea grande.

Para reducir el costo del modelo se tienen que seleccionar otros parámetros con los que J esté más cerca de su valor mínimo, para lo cual se puede utilizar el gradiente ∇J. Una de las propiedades del vector gradiente es que siempre apunta en la dirección en la que la función aumenta más abruptamente, así que el negativo del gradiente apunta hacia donde hay un mayor descenso en la función. Si en el modelo hay M pesos y N biases entonces el gradiente ∇J estará dado por la siguiente expresión:

$$\nabla J = \left(\frac{\partial J}{\partial w_1}, ..., \frac{\partial J}{\partial w_M}, \frac{\partial J}{\partial b_1}, ..., \frac{\partial J}{\partial b_N}\right) \tag{29}$$

En la ecuación (28) se puede ver que J es una función compuesta, por lo que para calcular la derivada parcial de J respecto al peso w_i se tiene que utilizar la regla de la cadena:

$$\frac{\partial J}{\partial w_i} = \frac{\partial J}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial w_i} \tag{30}$$

Dado que para calcular la derivada parcial de J respecto a cualquiera de los parámetros $\{\mathbf{w}, b\}$ se tiene que calcular una derivada respecto a la activación a, luego una derivada respecto a la combinación lineal z y finalmente la derivada respecto a un peso o bias, pareciera que para calcular el gradiente ∇J se tiene que recorrer la red neuronal en la dirección opuesta: desde la activación en la última capa de la red hasta los parámetros que están en la capa de entrada. Por esta razón este proceso se conoce como propagación hacia atrás. Una vez que se han calculado las derivadas parciales de J respecto a todos los parámetros, se puede pasar a la última etapa: la actualización de parámetros.

La actualización de los parámetros consiste en conseguir un nuevo conjunto de parámetros $\{\mathbf{w}', b'\}$ a partir de los parámetros que inicialmente se tenían mediante la siguiente regla

$$\{\mathbf{w}', b'\} = \{\mathbf{w}, b\} - \alpha \nabla J, \tag{31}$$

donde α es un coeficiente llamado razón de aprendizaje.

De la ecuación anterior se sigue que la actualización de cualquier parámetro individual θ está dada por

$$\theta' = \theta - \alpha \frac{\partial J}{\partial \theta}, \tag{32}$$

y con la transformación de todos los parámetros de la red neuronal es que finaliza una iteración del proceso de optimización.

Como en el caso de la superficie de energía potencial, el costo J también se puede representar como una hipersuperficie en la que cada punto corresponde a un conjunto específico de parámetros $\{\mathbf{w}, b\}$. A partir de esta superficie, cada paso en una iteración del entrenamiento se puede interpretar de la siguiente manera:

- Al aplicar la propagación hacia delante y calcular la función de costo a partir de todas las muestras en el conjunto de datos se obtiene un punto sobre la superficie del costo J.
- Con la propagación hacia atrás se calcula el vector $-\nabla J$, que apunta en la dirección donde la superficie desciende más rápido.
- En la actualización de parámetros se da un paso en la dirección de $-\nabla J$, y el tamaño de ese paso está determinado por la razón de aprendizaje α.

En el lado izquierdo de la Figura 8 está la visualización de este procedimiento, donde inicialmente se estaba en el punto $\mathbf{p}$ y después de una iteración del entrenamiento se llega al punto $\mathbf{p} - \alpha \nabla J$, que está más cerca del mínimo de la superficie de costo. Si esto se repite varias veces entonces se podrá llegar al punto mínimo de la superficie tal y como se puede ver en el lado derecho de la Figura 8, que es donde se encuentran los parámetros $\{\mathbf{w}, b\}$ con los que la red neuronal va a realizar las predicciones más exactas.

Como el componente principal de este procedimiento es el vector gradiente, a este algoritmo de optimización se le conoce como descenso del gradiente. Existen diversas variantes del descenso del gradiente, como Momentum (Polyak, 1964) (Goh, 2017), Adagrad (*Adaptative Gradient Algorithm*) (Duchi et al., 2011), RMSProp (*Root Mean Square Propagation*) (Tieleman & Hinton, 2012) y Adam (*Adaptive Moment Estimation*) (Kingma & Ba, 2014), entre otros.

Los parámetros en una red neuronal son los pesos de cada conexión y los biases que se suman en cada capa de la red. Sin embargo hay muchos otros parámetros (como la razón de aprendizaje o el número de capas) que afectan en mayor o menor medida la duración del entrenamiento y la eficiencia del modelo final. A estos parámetros se les denomina como hiperparámetros, y entre los más importantes

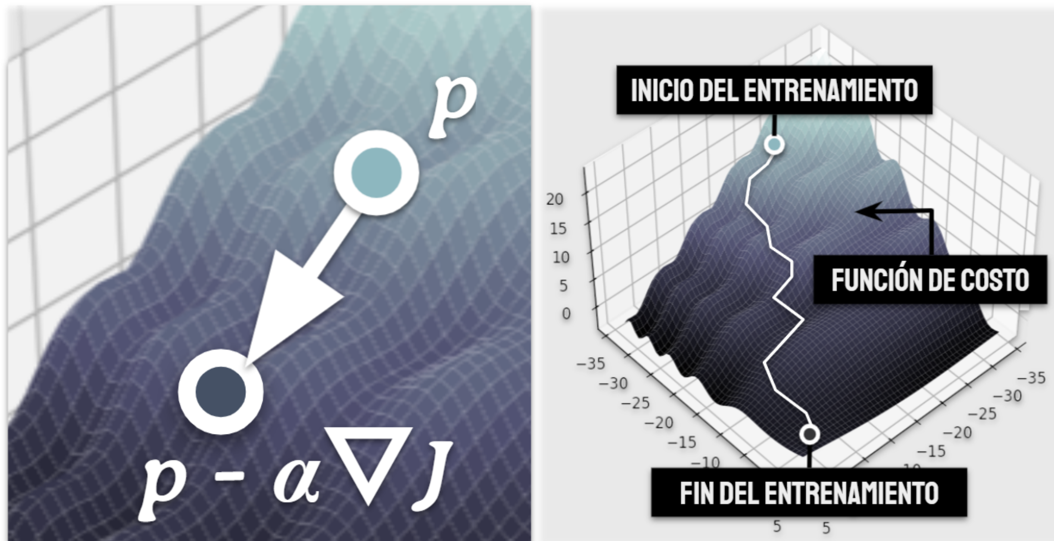


Figura 8: En la imagen de la izquierda se muestra una iteración del descenso del gradiente. En la imagen de la derecha se observa el resultado de aplicar ese algoritmo múltiples veces.

se encuentran el número de épocas y el número de muestras que se utilizan en cada época del entrenamiento.

El término ”época” se refiere a una iteración del algoritmo de optimización, lo que significa que en un entrenamiento de 500 épocas se realizarán 500 iteraciones del descenso del gradiente (o cualquier otro algoritmo de optimización) para reducir el costo del modelo.

Por otro lado, el número de muestras que se introducen en la red neuronal para calcular el costo puede variar (Zhang et al., 2019). En el planteamiento que se ha seguido hasta ahora se supone que se utiliza todo el conjunto de muestras, pero hay casos en los que un conjunto de datos puede llegar a tener miles o incluso millones de muestras, por lo que procesar toda esa información para realizar un sólo paso del descenso del gradiente podría tomar demasiado tiempo. Por esta razón es que se acostumbra que en cada época del entrenamiento se seleccione un subconjunto aleatorio del conjunto de datos, lo cual acelera notablemente los cálculos.

Con esto termina el algoritmo de entrenamiento, aunque en realidad el proceso de entrenar una red neuronal es mucho más complejo. Como este no es el tema central de este trabajo no se va a entrar en detalle, sin embargo es importante remarcar que todos los aspectos involucrados en la construcción y entrenamiento de una red neuronal afectan la eficiencia del modelo final. La selección de los hiperparámetros de la red e incluso la obtención del conjunto de datos no es una tarea trivial. Si no se tiene el suficiente cuidado se podría terminar con un modelo que en el entrenamiento haya conseguido un costo muy pequeño, pero que al aplicarlo en

el ”mundo real”, con datos nuevos y desconocidos, arroje predicciones imprecisas. Por esta razón es que en la práctica se acostumbra a aplicar la red neuronal en un conjunto de datos que no se hayan utilizado para su entrenamiento y entonces se comparan las predicciones con los valores esperados. A este conjunto de datos se le conoce como conjunto de validación. El error que se obtenga en este conjunto de datos es el que va a indicar si los hiperparámetros seleccionados fueron acertados o no. Por último se aplica el modelo en un conjunto de datos desconocidos y el error que se obtenga con esos datos es el que determina la eficiencia de todo el modelo.

4. Potencial generado con una red neuronal artificial

Las redes neuronales son modelos que, si tienen los parámetros adecuados, son capaces de predecir la relación entre una variable independiente x y una variable dependiente y. A continuación se presenta una red neuronal que puede predecir la energía potencial E de un sistema atómico a partir de las posiciones de las partículas (Behler & Parrinello, 2007).

4.1. Creación del conjunto de datos

Lo primero que se tiene que hacer es conseguir un conjunto de datos a partir de los cuales la red neuronal "aprenderá" cuál es la relación entre las posiciones atómicas y la energía potencial total. Para esto se utilizó el software de química *TeraChem* (Seritan et al., 2020), con el que pueden hacerse simulaciones de dinámica molecular con base en la teoría DFT (Kohn & Sham,1965). Durante la ejecución de una simulación se guardan algunos datos en un archivo de salida. Entre esos datos se encuentran las posiciones y energías atómicas, cargas eléctricas y la energía total del sistema. En este trabajo al archivo que contiene toda esa información se le llamó *input.data*, y a los datos correspondientes a una sola iteración de la simulación se le denominó como *frame*.

Los datos en *input.data* están separados en frames, donde cada frame empieza con una línea que dice `begin` y termina con una línea que dice `end`. Entre esas dos líneas se despliega la información del sistema atómico. Para cada átomo en el sistema hay una línea que empieza con la palabra `atom` y después viene la información de ese átomo que es **(a)** la posición en coordenadas cartesianas (x, y, z), **(b)** símbolo atómico, **(c)** energía, **(d)** carga de Mulliken y **(e)** las componentes (f_x, f_y, f_z) de la fuerza que actúa sobre el átomo en cuestión. Después de la información atómica se agrega una línea con la energía potencial total y otra con la carga eléctrica de todo el sistema. Luego se repite este mismo formato con la información del siguiente frame. En Código 1 se muestra un par de frames correspondientes a un sistema compuesto por un átomo de helio, un átomo de hidrógeno y un átomo de azufre.

En este punto podría surgir la pregunta de cuál es el beneficio de utilizar redes neuronales si inevitablemente se va a requerir una simulación hecha con cálculos cuánticos, y la respuesta es que sólo se necesita hacer una vez esa simulación para generar los datos necesarios para el entrenamiento de la red. Después se podrán hacer simulaciones con la red neuronal entrenada en mucho menos tiempo y con un costo computacional menor que el de un simulador de principios cuánticos (Singraber, 2019).

```
begin
 atom   1.5431384368  -1.1191295940  -17.0485278693   He  0.027094   0.00
    0.0157537280  -0.0181938513  -0.0642373160
 atom  -0.5894251469  -1.8160947081  -17.0515873198   H   0.018746   0.00
    -0.0100892178  -0.0238464092  -0.0624572719
 atom  -2.6832042561  -5.5809488944   12.2070706198   S   0.193836   0.00
    -0.0009933754  -0.0016049359   0.0023441287
energy -63281.8689480191
charge   0.000000
end
begin
 atom  -1.5073285374  -0.5329226948   13.7525316551   He -0.089117   0.00
    -0.0003132353  -0.0006831209   0.0038223120
 atom   1.5431384368  -1.1191295940  -17.0485278693   H   0.019072   0.00
    0.0192380329  -0.0173428512  -0.0517227537
 atom  -0.5894251469  -1.8160947081  -17.0515873198   S   0.014154   0.00
    -0.0075349067  -0.0246254451  -0.0533427212
energy -63278.6549433765
charge  0.000000
end
```

Código 1: Formato del archivo *input.data*.

Ya se tiene un conjunto de datos con toda la información necesaria para entrenar a la red neuronal, pero antes de pasar al entrenamiento se deben procesar los datos.

4.2. Funciones de simetría

Si se va a utilizar una red neuronal artificial para modelar un sistema físico, es necesario que esa red neuronal sea capaz de reproducir las propiedades físicas del sistema en cuestión. En este caso lo que se requiere es que la red que genere la superficie de energía potencial obedezca el principio de conservación de la energía, pero si se utilizan las coordenadas cartesianas de *input.data* para entrenar a la red neuronal lo que se va a observar es que la energía no se conserva bajo traslaciones ni bajo rotaciones.

Una solución a este problema es intercambiar las coordenadas cartesianas por otras coordenadas llamadas funciones de simetría (Behler & Parrinello, 2007), las cuales son funciones que dependen de las distancias y ángulos interatómicos entre las partículas del sistema. Estas coordenadas capturan apropiadamente la información sobre el entorno de un átomo y además son invariantes bajo rotaciones y traslaciones.

Lo primero que se debe hacer para calcular la función de simetría de un átomo es colocar una esfera de radio R_c alrededor de ese átomo. El átomo que está en el centro de la esfera se llama átomo central y al radio R_c se le conoce como radio de corte. Los átomos que se encuentran dentro de la esfera se definen como los átomos vecinos del átomo central. En la Figura 9 se muestra una representación de la esfera de corte alrededor de un átomo central.

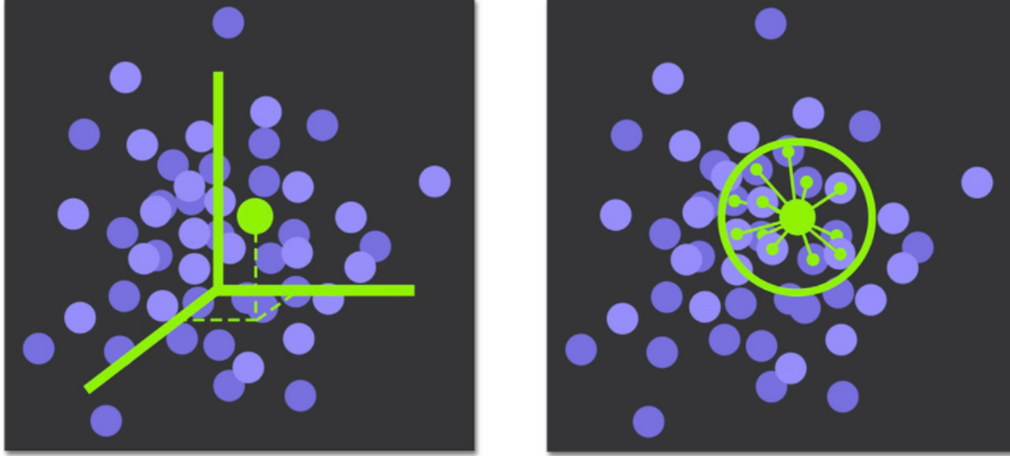


Figura 9: Sistema atómico descrito con coordenadas cartesianas (izquierda) y con las funciones de simetría (derecha).

Una función de simetría es una transformación $G : \mathbb{R}^{3n} \to \mathbb{R}$ que depende de las distancias que hay entre el átomo central y sus n átomos vecinos. La razón por la cual sólo se toman en cuenta los átomos dentro de la esfera de corte para calcular a G es que se supone que las únicas interacciones significativas son las que un átomo tiene con sus átomos cercanos.

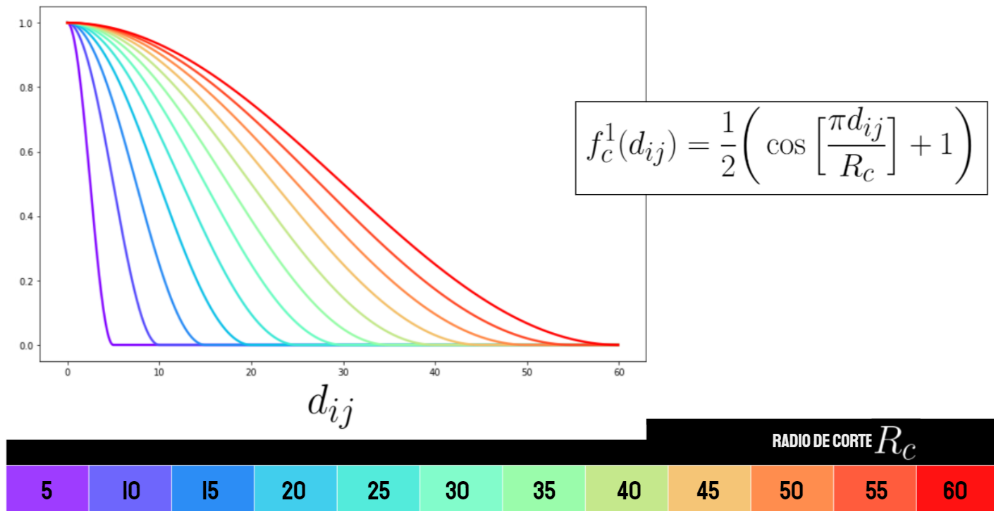
Las funciones de simetría se componen de dos partes, una distribución gaussiana y una función de corte (Behler, 2011). Las funciones de corte determinan qué tanta influencia tiene un átomo vecino sobre el átomo central dependiendo de la distancia que hay entre esos dos átomos. Algunas funciones de corte f_c son (Behler & Parrinello, 2007):

$$f_c^1(d_{ij}) = \begin{cases} \frac{1}{2}\left(\cos\left[\frac{\pi d_{ij}}{R_c}\right] + 1\right) & \text{, para } d_{ij} < R_c \\ 0 & \text{, para } d_{ij} > R_c \end{cases} \tag{33}$$

y

$$f_c^2(d_{ij}) = \begin{cases} \tanh^3\left[1 - \frac{d_{ij}}{R_c}\right] & \text{, para } d_{ij} < R_c \\ 0 & \text{, para } d_{ij} > R_c \end{cases} \tag{34}$$

donde d_{ij} es la distancia que hay entre el i-ésimo átomo central y su j-ésimo átomo vecino. En la Figura 10 se muestra la gráfica de ambas funciones de corte para distintos valores del radio de corte R_c.



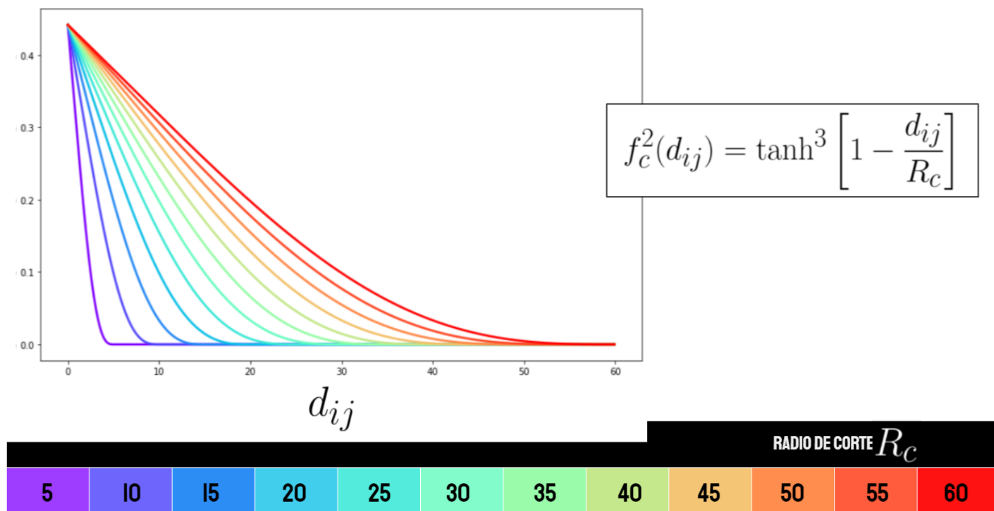


Figura 10: Arriba está la gráfica de la función de corte f_c^1, y debajo la gráfica de f_c^2 para diferentes valores del parámetro R_c. Las unidades del radio de corte son angstroms.

Existen dos tipos de funciones de simetría: radiales y angulares (Behler, 2011). La función de simetría radial del i-ésimo átomo está dada por la siguiente expresión:

$$G_i^{rad} = \sum_{j=1}^{n} e^{-\eta(d_{ij}-R_s)^2} \cdot f_c(d_{ij}), \tag{35}$$

donde η y R_s son parámetros que determinan el ancho y el centro de la distribución Gaussiana, respectivamente. En la Figura 11 se muestran las gráficas de esta función de simetría.

En cuanto a las funciones de simetría angulares, existen dos de ellas: la función de simetría angular *aumentada*

$$G_i^{ang.w} = 2^{1-\zeta} \sum_{j=1}^{n} \sum_{k=j+1}^{n} \left(1 + \lambda \cos\theta_{ijk}\right)^{\zeta} \cdot e^{-\eta(d_{ij}^2+d_{ik}^2)} \cdot f_c(d_{ij}) f_c(d_{ik}) \tag{36}$$

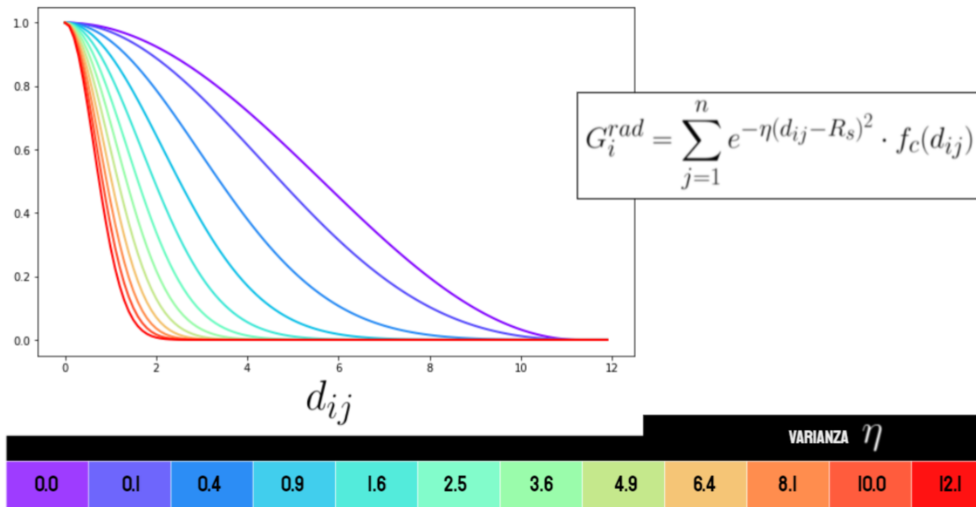
y la función de simetría angular *reducida*

$$G_i^{ang.n} = 2^{1-\zeta} \sum_{j=1}^{n} \sum_{k=j+1}^{n} \left(1 + \lambda \cos\theta_{ijk}\right)^{\zeta} \cdot e^{-\eta(d_{ij}^2+d_{ik}^2+d_{jk}^2)} \cdot f_c(d_{ij}) f_c(d_{ik}) f_c(d_{jk}), \tag{37}$$

donde λ es un parámetro que determina si el máximo del coseno está en $0°$ ($\lambda = +1$) o en $180°$ ($\lambda = -1$), ζ es un parámetro que controla la resolución angular y θ_{ijk} es el ángulo que hay entre el i-ésimo átomo central, el j-ésimo átomo vecino y el k-ésimo átomo vecino. En la Figura 12 se pueden observar las gráficas de las funciones de simetría angulares aumentadas.

La diferencia entre estas dos funciones es que en la función aumentada las distancias entre el átomo central y los átomos vecinos deben ser menores que R_c pero la distancia entre el vecino-j y el vecino-k puede ser mayor que R_c, mientras que en la función reducida las tres distancias d_{ij}, d_{ik} y d_{jk} deben ser menores que R_c.

Por lo tanto, se tienen que utilizar las ecuaciones (35), (36) y (37) para transformar los datos del archivo *input.data*. Los parámetros de las funciones de simetría deben seleccionarse con base en las características del sistema físico que se esté estudiando (Bircher et al., 2021). Las funciones de simetría que resulten de ese cambio de coordenadas se guardan en un archivo llamado *function.data*, en el que la información también se divide por frames. Ahora cada frame está conformado por las funciones de simetría de todos los átomos del sistema, además de la energía potencial total. Con los datos de *function.data* es que se hará el entrenamiento de la red neuronal, lo que significa que *function.data* es el conjunto de datos y cada frame es una muestra donde las funciones de simetría son los rasgos y la energía potencial total es la etiqueta (Witkoskie & Doren, 2005). Nótese que las funciones de simetría no son las únicas coordenadas con las que se mantienen las simetrías del sistema y que al mismo tiempo ofrecen una descripción adecuada del ambiente local de los átomos. Existen otras alternativas, por ejemplo las funciones de simetría ponderadas (Gastegger et al., 2018). Sin embargo, aquí se van a considerar únicamente a las funciones de simetría para ser consistentes con el modelo de Behler y Parrinello.



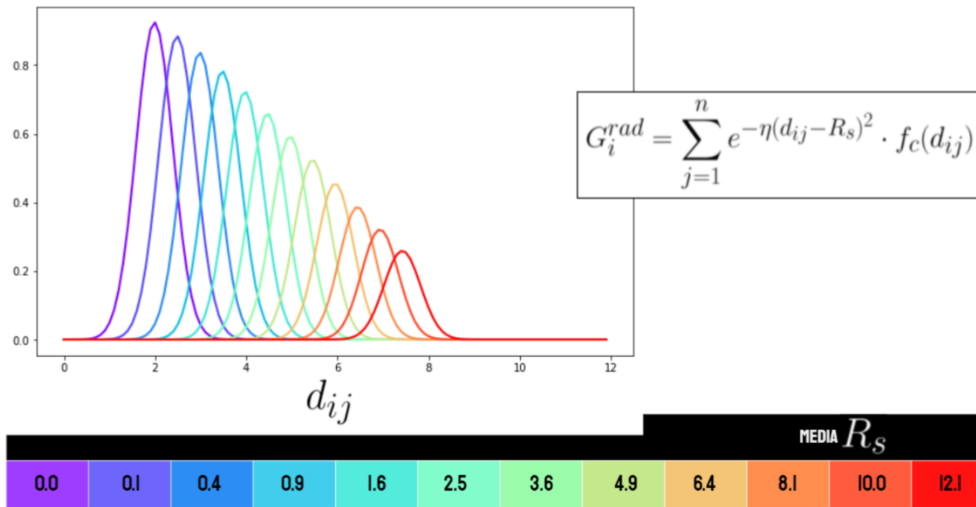
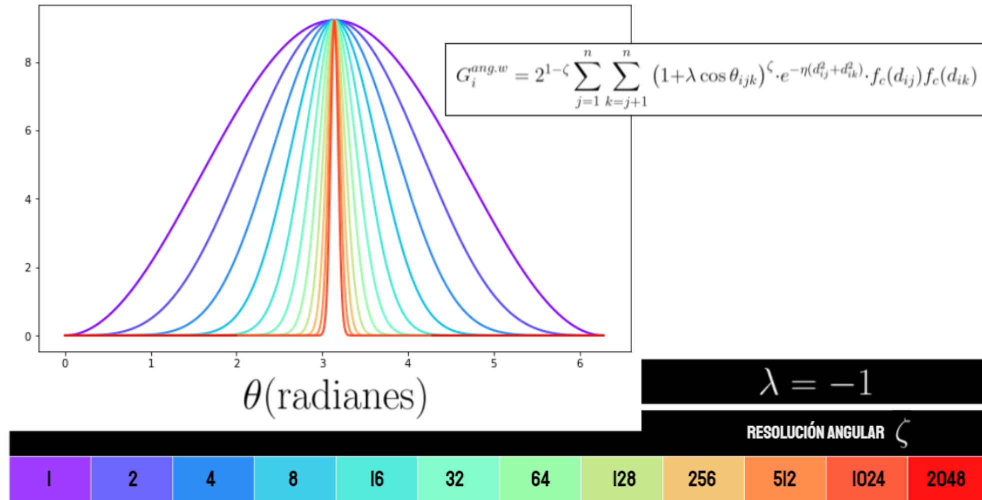


Figura 11: Arriba están las gráficas de las funciones de simetría radiales para diferentes valores del parámetro η y abajo para diferentes valores del parámetro R_s.



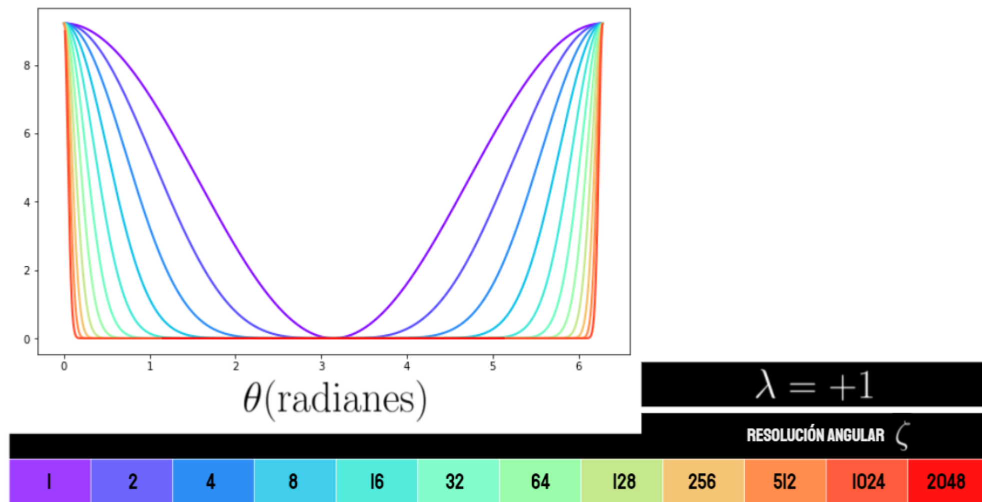


Figura 12: Arriba están las gráficas de las funciones de simetría angulares aumentadas para diferentes valores del parámetro ζ cuando $\lambda = -1$ y abajo cuando $\lambda = +1$.

4.3. Arquitectura de la red neuronal artificial

Antes de pasar a la descripción de la arquitectura de la red, se mencionan algunas observaciones:

- Anteriormente se vio que una muestra es un par {rasgos, etiqueta}. En este caso cada muestra está compuesta por un conjunto de funciones de simetría (rasgos) y una energía potencial total (etiqueta).
- Por cada átomo hay un conjunto de funciones de simetría que lo describen.
- El número de funciones de simetría que describen a un átomo α_1 puede ser distinto al número de funciones que describen al átomo α_2 si los átomos pertenecen a tipos diferentes. En caso de que α_1 y α_2 pertenezcan al mismo tipo atómico, ambos átomos deberán tener el mismo número de funciones de simetría.

Supóngase un sistema compuesto por N_1 átomos del tipo T1 y N_2 átomos del tipo T2, donde a cada átomo T1 y a cada átomo T2 les corresponden n_1 y n_2 funciones de simetría respectivamente. Por lo tanto, cada muestra tendría un total de $(N_1 \times n_1) + (N_2 \times n_2)$ rasgos. Se podría construir una red neuronal como la que se ha visto hasta ahora que tenga $(N_1 \times n_1) + (N_2 \times n_2)$ neuronas en la capa de entrada, pero con una arquitectura así no se conservaría la energía si se fuera a intercambiar la posición de dos átomos del mismo tipo (Behler, 2011). Por esta razón es que se va a proponer una arquitectura diferente.

La arquitectura de la red neuronal (Behler, 2011) con la que se va a generar la superficie de energía potencial se puede describir como una red de redes neuronales individuales. Por cada átomo en el sistema va a haber una red individual, y los rasgos que se introduzcan en esa red serán las funciones de simetría que describen al átomo en cuestión; el resultado de esa red individual se puede interpretar como una "energía atómica", aunque en realidad ese valor no tiene un significado físico.

La capa de salida de todas las redes individuales correspondientes a los átomos T1 se van a conectar a un nodo y ahí se van a sumar todas esas contribuciones. El resultado de esa suma representa la energía potencial total de los átomos del tipo T1. Lo mismo se hace con los átomos del tipo T2. Por último se conectan estas dos neuronas en un nodo donde se van a sumar las dos energías por tipo de átomo para dar como resultado la energía potencial de todo el sistema. En la Figura 13 se muestra la red neuronal con esta descripción para un sistema compuesto por tres tipos de átomos.

Dado que cada átomo del tipo T1 cuenta con n_1 funciones de simetría, todas las redes individuales del tipo T1 van a tener n_1 neuronas en la capa de entrada. Asimismo, todas las redes que pertenecen a los átomos del tipo T2 tienen n_2 neuronas en su capa de entrada.

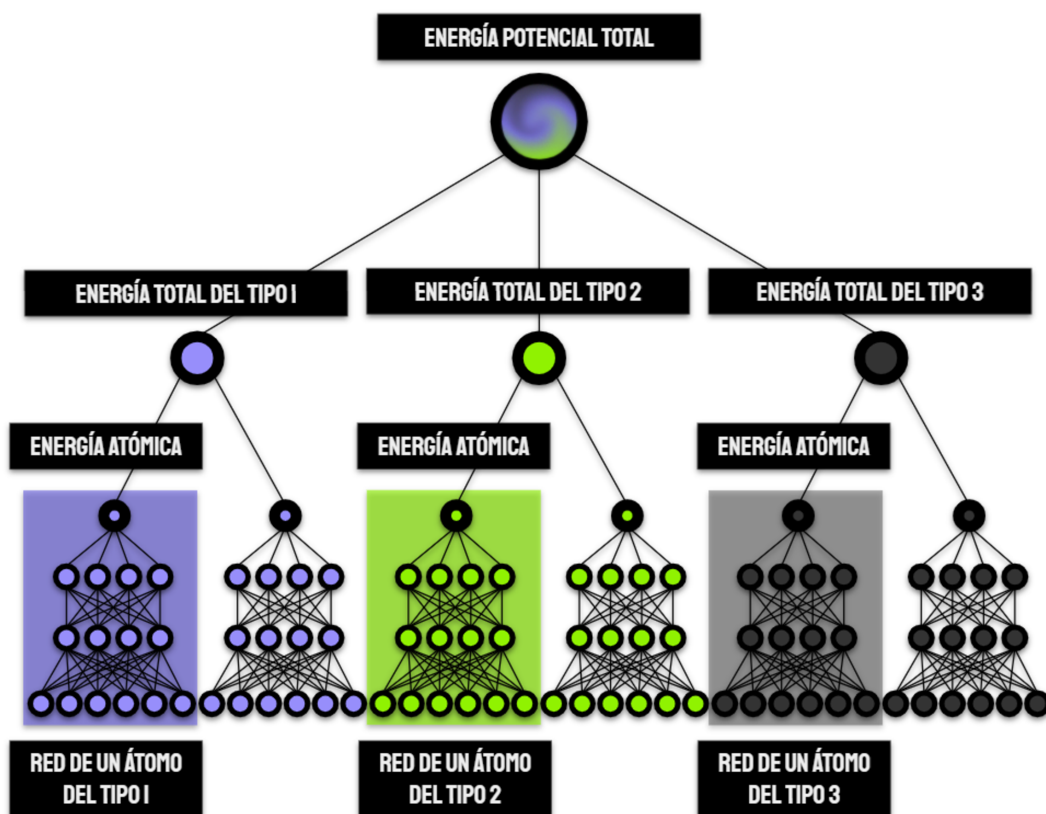


Figura 13: Red neuronal artificial que predice la energía potencial de un sistema compuesto por tres tipos de átomos.

Una ventaja de esta arquitectura es que todas las redes individuales que pertenecen a un mismo tipo de átomo comparten los mismos parámetros (pesos y biases). Esto significa que si una red individual del tipo T1 tiene w_1 parámetros entonces durante el entrenamiento de la red neuronal sólo se deberán entrenar a esos w_1 parámetros, sin importar si el sistema está compuesto por diez o por miles de átomos T1. Así pues, la cantidad de parámetros que se tienen con esta arquitectura es considerablemente menor que la cantidad que se tendría si la red neuronal tuviera una estructura convencional.

Ya que se tiene un conjunto de datos y una red neuronal, lo que sigue es entrenar al modelo con el algoritmo de optimización que se presentó anteriormente para hallar un conjunto de parámetros adecuado. Así es como se genera la superficie de energía potencial total de un sistema atómico y una vez obtenida esa superficie, lo siguiente es calcular a partir de ella las fuerzas atómicas (Singraber et al., 2019).

5. Dinámica molecular con redes neuronales

5.1. Cálculo de las fuerzas atómicas

La fuerza que actúa sobre un cuerpo se puede calcular a partir de la energía potencial E:

$$\mathbf{F} = -\nabla E \tag{38}$$

En el caso de un sistema integrado por múltiples átomos, la energía total es igual a la suma de las energías de todas los átomos que componen al sistema (Behler et al., 2008):

$$E = \sum_{i=1}^{N} E_i, \tag{39}$$

donde N es el número de átomos.

Por lo tanto, la fuerza que actúa sobre el átomo a con coordenadas $\mathbf{r}_a = (x_a, y_a, z_a)$ es:

$$\mathbf{F}_a = -\nabla_a E = -\sum_{i=1}^{N} \nabla_a E_i, \tag{40}$$

donde ∇_a es el operador gradiente respecto a las coordenadas $\mathbf{r_a}$.

Sin embargo, la energía potencial E que se obtiene a través de una red neuronal no depende de las coordenadas cartesianas sino que de las funciones de simetría (Behler & Parrinello, 2007). Además, de acuerdo con la arquitectura de la Figura 13, la energía total E es igual a la suma de las energías por tipo de átomo E_t, y a su vez cada una de estas energías se obtiene a partir de la suma de las energías de todos los átomos que pertenecen al tipo t:

$$E = \sum_{t=1}^{T} E_t = \sum_{t=1}^{T} \sum_{i=1}^{N_t} E_i^t, \tag{41}$$

donde T es el número de tipos de átomo en el sistema, N_t es el número de átomos que pertenecen al tipo t y E_i^t es la energía del i-ésimo átomo de tipo t.

La relación entre las distintas energías se puede observar en la Figura 14, que es la arquitectura de la red neuronal donde también se muestra la dependencia entre las funciones de simetría G y las posiciones atómicas $\mathbf{r}$. Como se puede observar, la trayectoria que se sigue en esta estructura es la siguiente:

$$
\begin{array}{c}
\text{ENERGÍA POTENCIAL TOTAL } E \\
\uparrow \\
\text{ENERGÍA POR TIPO DE ÁTOMO } E^t \\
\uparrow \\
\text{ENERGÍA ATÓMICA } E_i^t \\
\uparrow \\
\text{FUNCIONES DE SIMETRÍA } G_{il}^t \\
\uparrow \\
\text{POSICIONES ATÓMICAS } \mathbf{r}_i
\end{array}
$$

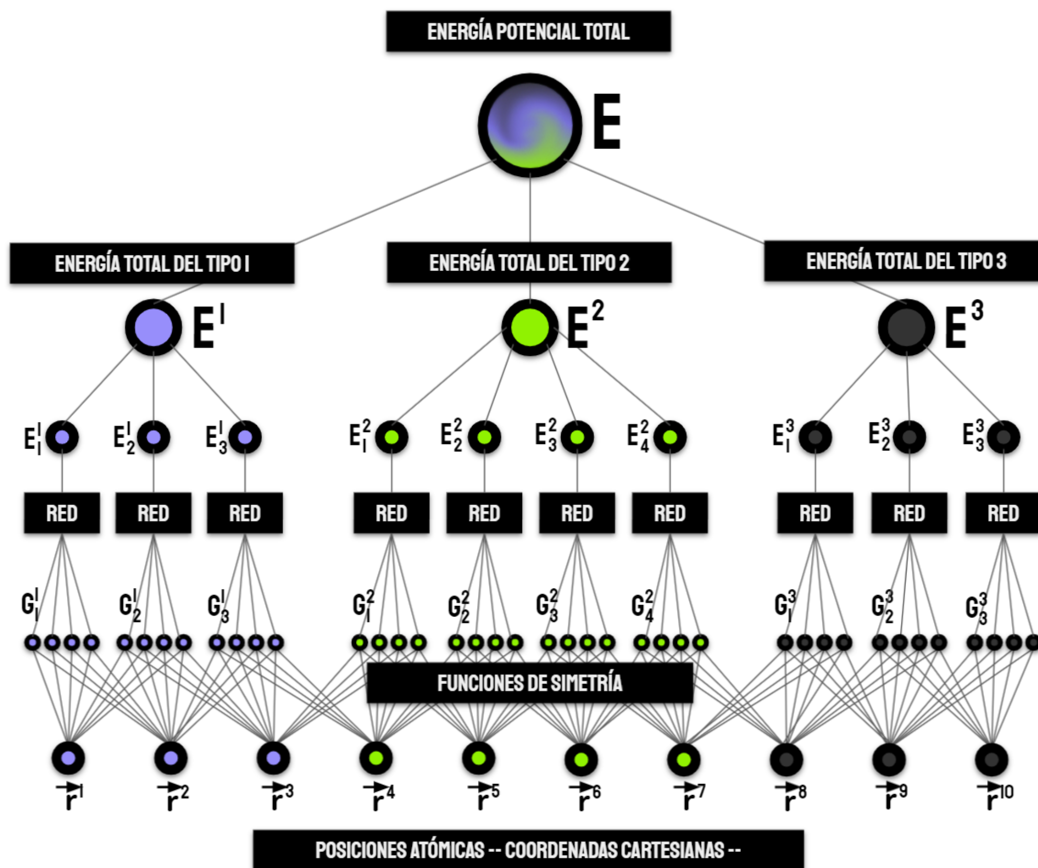


Figura 14: Arquitectura de la red neuronal donde también se considera la dependencia entre las funciones de simetría y las posiciones atómicas.

Considérese un sistema compuesto por seis átomos donde el átomo α es el átomo central. En la Figura 15 se muestra la red neuronal correspondiente a este sistema (incluyendo la dependencia entre funciones de simetría y posiciones atómicas). Como se puede observar, la energía atómica E_α^t depende de las funciones de simetría G_α^t, y estas funciones dependen de la posición $\mathbf{r}_\alpha$ del átomo α. Esto quiere decir que un cambio en la posición de este átomo implica un cambio en la energía E_α^t, el cual está dado por la siguiente expresión

$$\nabla_\alpha E_\alpha^t = \sum_{l=0}^{M_\alpha} \frac{\partial E_\alpha^t}{\partial G_{\alpha l}^t} \frac{\partial G_{\alpha l}^t}{\partial \mathbf{r}_\alpha}, \tag{42}$$

donde $G_{\alpha l}^t$ es la l-ésima función de simetría del átomo α y M_α es el número total de funciones de simetría que describen a dicho átomo.

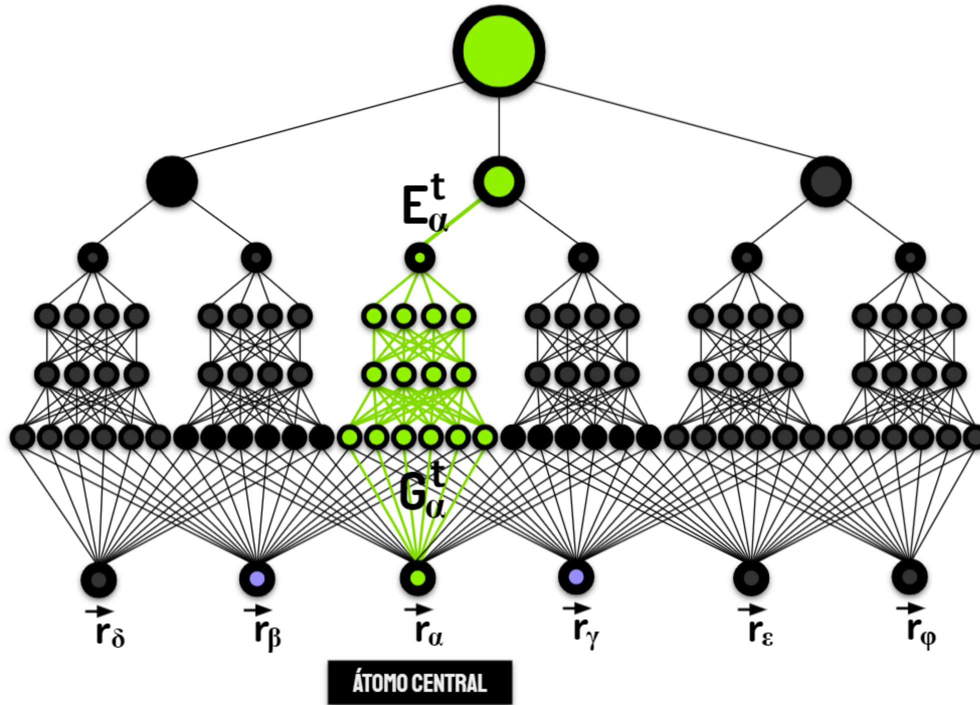


Figura 15: El cambio en la posición del átomo central se propaga hacia la energía atómica correspondiente a ese átomo y llega hasta la energía potencial total.

Pero las funciones de simetría no sólo dependen de la posición del átomo central, sino que también dependen de las posiciones de sus átomos vecinos. Supongamos que el átomo β y el átomo γ son los únicos vecinos del átomo central α. Si se mueve al átomo α entonces sus funciones de simetría van a cambiar, y como α es vecino de β entonces las funciones de simetría del átomo β también van a sufrir un cambio. Lo mismo pasa con el átomo γ. En consecuencia, el cambio en la posición del átomo α modificará a las energías E_α^t, $E_\beta^{t'}$ y E_γ^t tal y como se puede observar

en la Figura 16. Esos cambios están dados por las siguientes ecuaciones:

$$\nabla_\alpha E^t_\alpha = \sum_{l=0}^{M_\alpha} \frac{\partial E^t_\alpha}{\partial G^t_{\alpha l}} \frac{\partial G^t_{\alpha l}}{\partial \vec{r}_\alpha} \ , \ \nabla_\alpha E^{t'}_\beta = \sum_{l=0}^{M_\beta} \frac{\partial E^{t'}_\beta}{\partial G^{t'}_{\beta l}} \frac{\partial G^{t'}_{\beta l}}{\partial \vec{r}_\alpha} \ , \ \nabla_\alpha E^t_\gamma = \sum_{l=0}^{M_\gamma} \frac{\partial E^t_\gamma}{\partial G^t_{\gamma l}} \frac{\partial G^t_{\gamma l}}{\partial \vec{r}_\alpha} \ . \tag{43}$$

Que los subíndices t y t' sean diferentes significa que los átomos vecinos no necesariamente deben ser del mismo tipo que el átomo central.

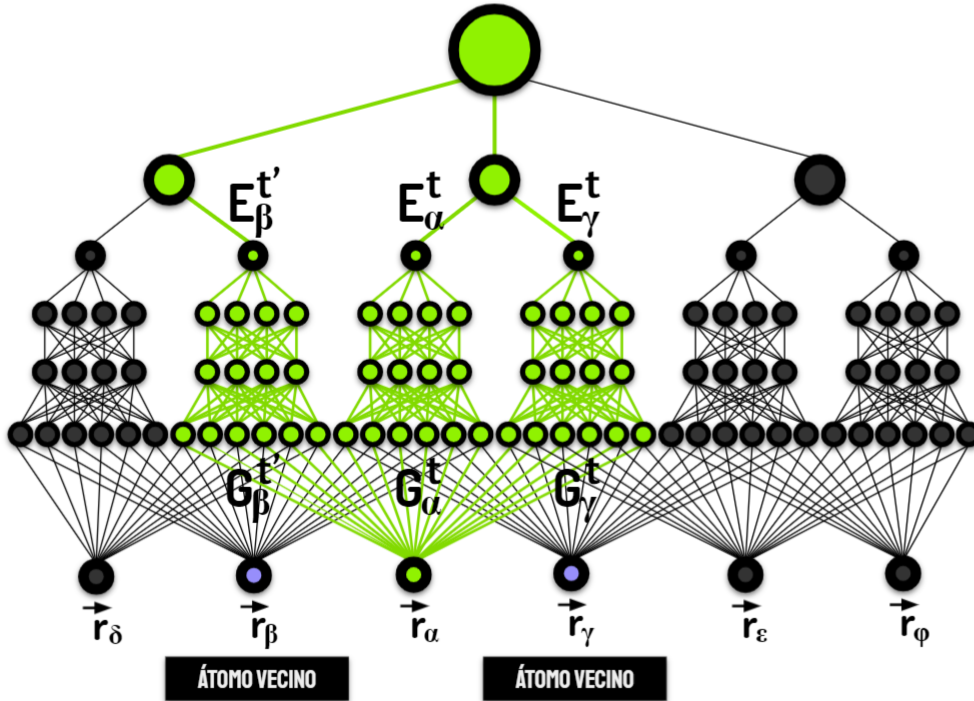


Figura 16: El cambio en la posición del átomo central se propaga hacia las energías atómicas de sus átomos vecinos.

Si un átomo δ no es vecino del átomo central entonces sus funciones de simetría (y por tanto, la energía $E^{t'}_\delta$) no se van a alterar si cambia la posición del átomo α. Esto es, $\nabla_\alpha E^{t'}_\delta = 0$ para átomos que no son vecinos del átomo central.

Por lo tanto, el cambio en la energía total E respecto a la posición del átomo α es (Behler, 2011):

$$\begin{aligned} \nabla_\alpha E &= \sum_{t=1}^{T} \sum_{i=1}^{N_t} \nabla_\alpha E^t_i \\ &= \sum_{t=1}^{T} \sum_{i=1}^{N_t} \sum_{l=0}^{M_i} \frac{\partial E^t_i}{\partial G^t_{il}} \frac{\partial G^t_{il}}{\partial \vec{r}_\alpha}. \end{aligned} \tag{44}$$

De aquí se sigue que la fuerza que actúa sobre el átomo α es:

$$\vec{F}_\alpha = -\nabla_\alpha E = -\sum_{t=1}^{T}\sum_{i=1}^{N_t}\sum_{l=0}^{M_i} \frac{\partial E_i^t}{\partial G_{il}^t}\frac{\partial G_{il}^t}{\partial \vec{r}_\alpha} \tag{45}$$

De acuerdo con esta última ecuación, para calcular la fuerza que actúa sobre el átomo α se tiene que hacer uso de la regla de la cadena. Es decir, se deben calcular las derivadas parciales de las energías E_i^t respecto a las funciones de simetría G_{il}^t y también las derivadas parciales de esas funciones de simetría respecto a la posición del átomo α.

5.2. Derivada de la energía respecto a las funciones de simetría

Para calcular la derivada parcial de la energía E_i^t respecto a la función de simetría G_{il}^t se tiene que recorrer a la red individual del átomo i desde la última capa hasta la primera, en un proceso parecido a la propagación hacia atrás en el algoritmo de entrenamiento.

Supóngase que se tiene una red neuronal para generar la superficie de energía potencial donde cada red individual de tipo t está compuesta por una capa de entrada, dos capas ocultas y una capa de salida. En la Figura 17 se muestra una red neuronal individual con esta descripción.

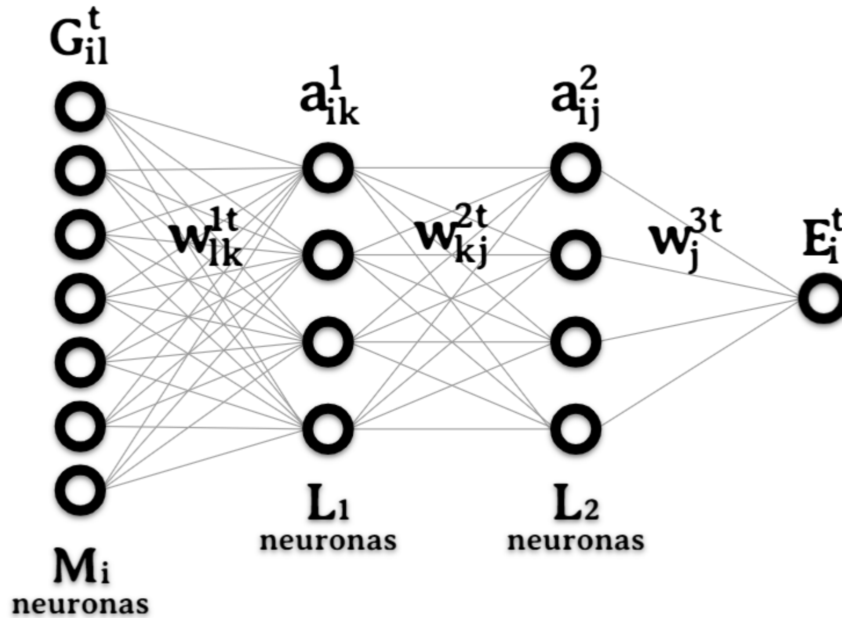


Figura 17: Red neuronal individual correspondiente a un átomo de tipo t.

E_i^t es el valor de la neurona en la última capa de la red, y está dado por esta expresión:

$$E_i^t = a^3(z_i^3), \text{ con } z_i^3 = \sum_{j=1}^{L_2} a_{ij}^2 w_j^{3t} + b^{3t}, \tag{46}$$

donde a^3 es la función de activación que se aplica en la capa de salida, L_2 es el número de neuronas en la segunda capa oculta y w_j^{3t} es el peso que une a la j-ésima neurona en la segunda capa con la única neurona en la tercer capa de la red.

El término a_{ij}^2 en la ecuación (46) corresponde al valor de la j-ésima neurona en la segunda capa oculta de la red neuronal:

$$a_{ij}^2 = a^2(z_{ij}^2), \text{ con } z_{ij}^2 = \sum_{k=1}^{L_1} a_{ik}^1 w_{kj}^{2t} + b^{2t}, \tag{47}$$

donde a^2 es la función de activación que actúa sobre la segunda capa oculta, L_1 es el número de neuronas en la primera capa oculta y w_{kj}^{2t} es el peso que une a la k-ésima neurona en la primer capa con la j-ésima neurona en la segunda capa.

De manera similar, a_{ik}^1 es el valor de la k-ésima neurona en la primer capa oculta de la red:

$$a_{ik}^1 = a^1(z_{ik}^1), \text{ con } z_{ik}^1 = \sum_{l=1}^{M_i} G_{il}^t w_{lk}^{1t} + b^{1t}, \tag{48}$$

donde a^1 es la función de activación de la primer capa oculta, M_i es el número de neuronas en la capa de entrada y G_{il}^t es la l-ésima función de simetría del átomo i de tipo t.

Al reunir todas estas ecuaciones se obtiene la siguiente expresión:

$$\begin{aligned} E_i^t &= a^3\Big(\sum_{j=1}^{L_2} a_{ij}^2 w_j^{3t} + b^{3t}\Big) \\ &= a^3\Big(\sum_{j=1}^{L_2} a^2\Big(\sum_{k=1}^{L_1} a_{ik}^1 w_{kj}^{2t} + b^{2t}\Big) w_j^{3t} + b^{3t}\Big) \\ &= a^3\Big(\sum_{j=1}^{L_2} a^2\Big(\sum_{k=1}^{L_1} a^1\big(\sum_{l=1}^{M_i} G_{il}^t w_{lk}^{1t} + b^{1t}\big) w_{kj}^{2t} + b^{2t}\Big) w_j^{3t} + b^{3t}\Big) \end{aligned} \tag{49}$$

Como se puede ver en la ecuación (49) la energía atómica E_i^t está dada por varias composiciones de funciones, por lo que para calcular su derivada parcial respecto a la función de simetría G_{il}^t se tiene que usar la regla de la cadena.

Si se modifica el valor de G_{il}^t entonces todas las neuronas en la primer capa de la red individual del átomo i también van a cambiar. Esa variación está dada por

la siguiente expresión:

$$\sum_{k=1}^{L_1} \frac{\partial a_{ik}^1}{\partial G_{il}^t} = \sum_{k=1}^{L_1} \frac{\partial z_{ik}^1}{\partial G_{il}^t} \frac{\partial a^1}{\partial z_{ik}^1} = \sum_{k=1}^{L_1} w_{lk}^{1t} \frac{\partial a^1}{\partial z_{ik}^1} \tag{50}$$

Ahora, si una sola de las activaciones a^1 cambia entonces todas las activaciones de la segunda capa también van a cambiar. El cambio de la segunda capa respecto a la activación a_{ik}^1 es:

$$\sum_{j=1}^{L_2} \frac{\partial a_{ij}^2}{\partial a_{ik}^1} = \sum_{j=1}^{L_2} \frac{\partial z_{ij}^2}{a_{ik}^1} \frac{\partial a^2}{\partial z_{ij}^2} = \sum_{j=1}^{L_2} w_{kj}^{2t} \frac{\partial a^2}{\partial z_{ij}^2} \tag{51}$$

De la misma manera, si cualquiera de las activaciones a^2 varía entonces la única neurona de la tercer capa también va a alterarse, y como la tercer capa de las redes individuales es la capa de salida entonces la energía E_i^t es la que va a cambiar. El cambio de E_i^t respecto a la activación a_{ij}^2 es:

$$\frac{\partial E_i^t}{\partial a_{ij}^2} = \frac{\partial z_i^3}{\partial a_{ij}^2} \frac{\partial a^3}{\partial z_i^3} = w_j^{3t} \frac{\partial a^3}{\partial z_i^3} \tag{52}$$

Con las ecuaciones anteriores se llega a que la derivada parcial de la energía del átomo i respecto a la l-ésima función de simetría es:

$$\frac{\partial E_i^t}{\partial G_{il}^t} = \left[\sum_{j=1}^{L_2} \left[\sum_{k=1}^{L_1} w_{lk}^{1t} \frac{\partial a^1}{\partial z_{ik}^1} w_{kj}^{2t} \right] \frac{\partial a^2}{\partial z_{ij}^2} w_j^{3t} \right] \frac{\partial a^3}{\partial z_i^3} \tag{53}$$

5.3. Derivada de las funciones de simetría respecto a las coordenadas cartesianas

La l-ésima función de simetría radial del i-ésimo átomo de tipo t está dada por la siguiente expresión:

$$\left[G_{il}^t\right]^{rad} = \sum_{m=1}^{N_i} e^{-\eta^l (d_{im} - R_s^l)^2} \cdot f_c(d_{im}), \tag{54}$$

donde N_i es la cantidad de átomos vecinos correspondientes al átomo i, f_c es la función de corte y $d_{im} = ||\vec{r_i} - \vec{r_m}||$ es la distancia entre el i-ésimo átomo central y su m-ésimo átomo vecino.

La derivada parcial de la función de simetría $\left[G_{il}^t\right]^{rad}$ respecto a la posición del átomo central i (Singraber et al., 2019) es:

$$\frac{\partial \left[G_{il}^t\right]^{rad}}{\partial \vec{r}_i} = \sum_{m=1}^{N_i} \kappa_l(d_{im}) \vec{r}_{im} \text{ , con } \vec{r}_{im} = \vec{r}_i - \vec{r}_m, \tag{55}$$

y la derivada respecto a la posición del m-ésimo átomo vecino es:

$$\frac{\partial\left[G_{il}^{t}\right]^{rad}}{\partial\vec{r}_{m}} = -\kappa_{l}(d_{im})\vec{r}_{im}. \tag{56}$$

donde

$$\kappa_{l}(r) = \frac{1}{r}e^{-\eta^{l}(r-R_{s}^{l})^{2}}\left[\frac{df_{c}(r)}{dr} - 2\eta^{l}(r-R_{s}^{l})f_{c}(r)\right]. \tag{57}$$

En cuanto a las funciones angulares, la l-ésima función de simetría angular aumentada del átomo i de tipo t es:

$$\left[G_{il}^{t}\right]^{ang.w} = 2^{1-\zeta^{l}}\sum_{m=1}^{N_i}\sum_{n=m+1}^{N_i}\left(1+\lambda^{l}\cos\theta_{imn}\right)^{\zeta^{l}}\cdot e^{-\eta^{l}(d_{im}^{2}+d_{in}^{2})}\cdot f_{c}(d_{im})f_{c}(d_{in}) \tag{58}$$

La derivada parcial de esta función respecto a la posición del átomo central i es (Singraber et al., 2019):

$$\begin{aligned}\frac{\partial\left[G_{il}^{t}\right]^{ang.w}}{\partial\vec{r}_{i}} &= 2^{1-\zeta^{l}}\sum_{m=1}^{N_i}\sum_{n=m+1}^{N_i}\xi_{l}(\vec{r}_{im},\vec{r}_{in}) \\ &\cdot\left[\left(\phi_{l}(\vec{r}_{im},\vec{r}_{in})-2\eta^{l}+\chi_{l}(d_{im})\right)\vec{r}_{in}+\left(\phi_{l}(\vec{r}_{in},\vec{r}_{im})-2\eta^{l}+\chi_{l}(d_{im})\right)\vec{r}_{in}\right]\end{aligned} \tag{59}$$

y la derivada respecto a la posición del átomo vecino m es:

$$\begin{aligned}\frac{\partial\left[G_{il}^{t}\right]^{ang.w}}{\partial\vec{r}_{m}} &= 2^{1-\zeta^{l}}\sum_{m=1}^{N_i}\sum_{n=m+1}^{N_i}\xi_{l}(\vec{r}_{im},\vec{r}_{in}) \\ &\cdot\left[-\left(\phi_{l}(\vec{r}_{im},\vec{r}_{in})-2\eta^{l}+\chi_{l}(d_{im})\right)\vec{r}_{im}+\psi_{l}(\vec{r}_{im},\vec{r}_{in})\vec{r}_{mn}\right],\end{aligned} \tag{60}$$

donde

$$\xi_{l}(\vec{r}_{im},\vec{r}_{in}) = \left(1+\lambda^{l}\cos\theta_{imn}\right)^{\zeta^{l}}e^{-\eta^{l}(d_{im}^{2}+d_{in}^{2})}f_{c}(d_{im})f_{c}(d_{in}), \tag{61}$$

$$\phi_{l}(\vec{r}_{im},\vec{r}_{in}) = \frac{\lambda^{l}\zeta^{l}}{1+\lambda^{l}cos\theta_{imn}}\left[\frac{1}{d_{im}d_{in}}-\frac{1}{d_{im}^{2}}\right], \tag{62}$$

$$\chi_{l}(r) = \frac{1}{rf_{c}(r)}\frac{df_{c}(r)}{dr}, \tag{63}$$

$$\psi_{l}(\vec{r}_{im},\vec{r}_{in}) = -\frac{1}{d_{im}d_{in}}\frac{\lambda^{l}\zeta^{l}}{1+\lambda^{l}cos\theta_{imn}}. \tag{64}$$

Por otro lado, la l-ésima función de simetría angular reducida del i-ésimo átomo de tipo t es:

$$\left[G_{il}^{t}\right]^{ang.n} = 2^{1-\zeta^{l}}\sum_{m=1}^{N_i}\sum_{n=m+1}^{N_i}\left(1+\lambda^{l}\cos\theta_{imn}\right)^{\zeta^{l}}\cdot e^{-\eta^{l}(d_{im}^{2}+d_{in}^{2}+d_{mn})}\cdot f_{c}(d_{im})f_{c}(d_{in})f_{c}(d_{mn}), \tag{65}$$

cuya derivada respecto a la posición del i-ésimo átomo central es (Singraber et al., 2019):

$$\frac{\partial\big[G_{il}^{t}\big]^{ang.n}}{\partial\vec{r}_i} = 2^{1-\zeta^l}\sum_{m=1}^{N_i}\sum_{n=m+1}^{N_i}\xi_l^{ang.n}(\vec{r}_{im},\vec{r}_{in}) \cdot \Big[\big(\phi_l(\vec{r}_{im},\vec{r}_{in}) - 2\eta^l + \chi_l(d_{im})\big)\vec{r}_{im} + \big(\phi_l(\vec{r}_{in},\vec{r}_{im}) - 2\eta^l + \chi_l(d_{in})\big)\vec{r}_{in}\Big] \quad (66)$$

y la derivada respecto a la posición del m-ésimo átomo vecino es:

$$\frac{\partial\big[G_{il}^{t}\big]^{ang.n}}{\partial\vec{r}_m} = 2^{1-\zeta^l}\sum_{m=1}^{N_i}\sum_{n=m+1}^{N_i}\xi_l^{ang.n}(\vec{r}_{im},\vec{r}_{in}) \cdot \Big[-\big(\phi_l(\vec{r}_{im},\vec{r}_{in}) - 2\eta^l + \chi_l(d_{im})\big)\vec{r}_{im} + \big(\psi_l(\vec{r}_{im},\vec{r}_{in}) - 2\eta^l + \chi_l(d_{mn})\big)\vec{r}_{mn}\Big], \quad (67)$$

donde

$$\xi_l^{ang.n}(\vec{r}_{im},\vec{r}_{in}) = \big(1 + \lambda^l\cos\theta_{imn}\big)^{\zeta^l} \cdot e^{-\eta^l(d_{im}^2+d_{in}^2+d_{mn}^2)} \cdot f_c(d_{im})f_c(d_{in})f_c(d_{mn}). \quad (68)$$

Con esto finalizan los cálculos de las derivadas parciales. Por lo tanto ya quedó determinado el método que se debe seguir para obtener las fuerzas atómicas a partir de una superficie de energía potencial generada con redes neuronales artificiales. Lo siguiente es implementar la red neuronal y las ecuaciones que se han visto hasta ahora en un código para poder realizar una simulación de dinámica molecular.

6. Código

Recapitulando, para realizar una simulación de dinámica molecular se requieren las posiciones iniciales de todos los átomos del sistema bajo estudio y un algoritmo con el que se puedan calcular las fuerzas que actúan sobre esos átomos. Una manera con la que se pueden obtener esas fuerzas es a través de calcular las derivadas de la superficie de energía potencial del sistema, la cual se puede obtener de varias maneras. Una de ellas consiste en entrenar una red neuronal artificial para que se ajuste a la superficie de energía potencial del sistema. Una vez que se han obtenido las fuerzas atómicas, lo siguiente es utilizarlas en un algoritmo con el que se puedan determinar las posiciones de los átomos después de transcurrido un intervalo de tiempo dt. La repetición iterativa de este proceso es lo que da lugar a una simulación de dinámica molecular.

Así pues, en esta parte se presenta un código donde se incorporan todos los conceptos y ecuaciones vistas en secciones anteriores para generar una simulación de dinámica molecular a partir de redes neuronales artificiales. El código está escrito en Python 3.6 y fue probado en un ambiente con la siguiente paquetería:

- `numpy 1.19.5`
- `keras 2.4.3`
- `tensorflow 2.4.1`
- `pandas 1.1.5`

También se requieren los programas `pydot` y `graphviz` para visualizar algunas gráficas.

El código para las simulaciones de dinámica molecular está distribuido entre un programa principal llamado `molecular_dynamics.py` y los módulos `frame`, `network_activation_functions` y `custom_layer`.

`molecular_dynamics.py` es un programa que contiene varias funciones que funcionan como un *pipeline*, lo que significa que el output de una función es el input de otra. De todas esas funciones, las únicas que están disponibles para el usuario son `training` y `molecular_dynamics`. Además, `molecular_dynamics.py` también cuenta con tres clases: `SymFunc_G2`, `SymFunc_G3` y `Atom`.

A cada átomo del sistema le corresponde un conjunto de funciones de simetría que lo describen, y cada una de esas funciones está determinada por un conjunto de parámetros. La clase `SymFunc_G2` se utiliza para almacenar los parámetros de las funciones de simetría radiales (ecuación 35). Un objeto de esta clase contiene los parámetros correspondientes a una sola función de simetría, y son: **(1)** el índice de la función de simetría, **(2)** el símbolo atómico del átomo central, **(3)** el símbolo atómico del átomo vecino, **(4)** η, **(5)** R_s y **(6)** R_c.

Con la clase `SymFunc_G3` se guardan los parámetros de las funciones de simetría angulares, ya sean aumentadas (ecuación 36) o reducidas (ecuación 37), donde cada objeto de esta clase contiene los siguientes parámetros: **(1)** el índice de la función de simetría, **(2)** el símbolo atómico del átomo central, **(3)** el símbolo atómico del j-ésimo átomo vecino, **(4)** el símbolo atómico del k-ésimo átomo vecino, **(5)** η, **(6)** λ, **(7)** ζ, **(8)** R_c y **(9)** R_s.

Por último, la clase `Atom` se usa para llevar un registro de la información de todos los átomos que integran el sistema. Cada objeto de esta clase almacena los datos de un sólo átomo, los cuales son: **(1)** índice, **(2)** posición en coordenadas cartesianas, **(3)** símbolo atómico, **(4)** velocidad, **(5)** fuerza, **(6)** número atómico, **(7)** masa atómica, **(8)** lista de átomos vecinos, **(9)** funciones de simetría, **(10)** derivada de la energía respecto a las funciones de simetría, **(11)** derivada de las funciones de simetría del átomo central respecto a la posición del átomo central y **(12)** derivada de las funciones de simetría de los átomos vecinos respecto a la posición del átomo central. Algunas de estas variables se leen de algún archivo y otras se obtienen (y actualizan) durante la ejecución del programa.

Por su parte, los módulos `frame`, `network_activation_functions` y `custom_layer` contienen funciones o clases que `molecular_dynamics.py` requiere para su funcionamiento. En `network_activation_functions` hay una lista de funciones de activación con sus respectivas derivadas. La lista incluye a la función sigmoide, la tangente hiperbólica, la tangente hiperbólica con giro lineal y la función identidad. En `custom_layer` se define una capa personalizada para la red neuronal artificial y en el módulo `frame` se encuentra una clase llamada `Frame` que se utiliza para procesar a todas las funciones de simetría antes de aplicarlas en el entrenamiento de la red neuronal.

A continuación se enlistan todas las funciones contenidas en el programa `molecular_dynamics.py`:

- `read_parameters`: Lee un archivo de texto llamado *control.in*, el cual contiene los parámetros de la arquitectura de la red neuronal, los hiperparámetros para el entrenamiento de la red y la ubicación de los archivos con los parámetros de las funciones de simetría y las posiciones atómicas.
- `read_SymFunc_parameters`: Lee los archivos con los parámetros de las funciones de simetría, que son archivos de texto con extensión *.smf*. Hay uno de estos archivos por cada tipo de átomo en el sistema.
- `read_positions`: Lee al archivo *input.data*, el cual contiene las posiciones atómicas y la energía potencial total del sistema.
- `read_data`: Llama a las funciones anteriores para obtener toda la información requerida para ejecutar el programa.
- `distance`: Calcula la distancia entre un par de átomos.
- `create_lists`: Genera una lista de listas vacías.
- `cut_function`: Funciones de corte.

- `cut_function_derivative`: Derivada de las funciones de corte.
- `compute_neighbors_list`: Dado un átomo del sistema, con esta función se obtiene una lista con los índices de sus átomos vecinos.
- `features`: Dado un átomo del sistema, con esta función se calculan sus funciones de simetría (y las derivadas).
- `process_training_data`: Normalización de las funciones de simetría. El resultado de esta función son los tensores X (con los rasgos/funciones de simetría) y Y (con las etiquetas/energías potenciales totales).
- `create_model`: Construye el modelo de la red neuronal.
- `training`: Ejecuta el entrenamiento de la red neuronal.
- `save_loss`: Guarda el costo del modelo obtenido durante el entrenamiento en los archivos *loss.out* y *val_loss.out*.
- `plot_loss`: Genera la gráfica del costo obtenido durante el entrenamiento.
- `get_weights_file`: Guarda los parámetros optimizados en archivos de texto.
- `get_weights`: Carga los parámetros optimizados del archivo *weights.h5*, el cual se genera cuando se entrena a la red neuronal.
- `energy_derivative`: Calcula la derivada de la energía potencial total respecto a las funciones de simetría.
- `compute_forces`: Calcula las fuerzas atómicas.
- `molecular_dynamics`: Ejecuta una simulación de dinámica molecular.

Nota: Es necesario que en la ubicación donde se corra el programa haya una carpeta llamada *outputs*, que es donde se guardarán los archivos que el programa genere durante la ejecución.

6.1. Código para el entrenamiento de la red neuronal

La función `training` es la que se utiliza para crear y entrenar redes neuronales, lo que significa que es esta función con la que se van a generar las superficies de energía potencial. `training` se puede utilizar de la siguiente manera:

```
import molecular_dynamics
molecular_dynamics.training('control.in', True)
```

`training` requiere dos argumentos para su ejecución. El primero es la ubicación del archivo de texto *control.in* y el segundo argumento es una variable booleana que determina si se van a guardar las funciones de simetría en un archivo llamado

function.data. En *control.in* es donde se encuentra la información sobre el sistema atómico del que se va a crear la superficie de energía potencial, los parámetros de la arquitectura de la red neuronal artificial y los hiperparámetros del entrenamiento, además de la ubicación de los archivos *.smf* con los parámetros de las funciones de simetría y la ubicación del archivo *input.data* con las posiciones atómicas. En Código 2 se muestra un ejemplo del formato que *control.in* debe tener para ejecutar a `training`. La información en la primer parte de este archivo se organiza en dos columnas: en la columna de la izquierda están los nombres de los parámetros y en la columna del lado derecho se encuentran sus valores correspondientes.

```
 1 ################################################################
 2 # Col  Description
 3 ################################################################
 4 # 1    Atomic symbol
 5 # 2    Atomic number
 6 # 3    Atomic mass
 7 # 4    Path of the file with the symmetry functions parameters
 8 ################################################################
 9
10 cut_function_type             2
11 angular_sf                    wide
12 atomic_data_path              data/input.data
13 length_unit                   Ang
14 number_of_atom_types          3
15 neurons_per_layer             [15, 15, 1]
16 activation_function           hyperbolic_tangent_with_linear_twist
17 weight_initialization         random_uniform
18 bias_initialization           zeros
19 random_seed                   10
20 optimization_method           Adagrad
21 learning_rate                 0.005
22 epsilon                       1e-08
23 decay_value                   0.0
24 clipvalue                     1.0
25 loss_function                 mse
26 number_of_epochs              600
27 minibatch_size                10
28 validation_fraction           0.1
29
30 ################################################################
31 #              1   2    3      4
32 ################################################################
33
34 atom_type_1     H   1   1.0008    H-SymFunc.smf
35 atom_type_2     O   8   15.999    O-SymFunc.smf
36 atom_type_3     Al  13  26.982    Al-SymFunc.smf
```

Código 2: Formato que el archivo *control.in* debe tener para crear y entrenar a la red neuronal artificial.

El parámetro `cut_function_type` se refiere al tipo de función de corte:

$$f_c(r) = \begin{cases} \frac{1}{2}\left[cos\left(\frac{\pi r}{R_c}\right) + 1\right] & , \text{ si } \texttt{cut_function_type} = 1 \\ tanh^3\left(1 - \frac{r}{R_c}\right) & , \text{ si } \texttt{cut_function_type} = 2 \end{cases} \tag{69}$$

`angular_sf` indica el tipo de función de simetría angular que se utilizará. Si es igual a `wide` se usarán las funciones de simetría aumentadas, pero si es igual a `narrow` entonces se usarán las funciones reducidas. Luego está `atomic_data_path`, que es la ubicación del archivo donde se encuentran las posiciones atómicas y las energías potenciales totales. `length_unit` se refiere a las unidades de longitud y `number_of_atoms_types` indica cuántos tipos de átomo hay en el sistema.

`neurons_per_layer` indica el número de neuronas que hay en cada capa de las redes individuales. Por ejemplo, si ese parámetro es igual a $[15, 15, 1]$ eso significa que cada red tiene dos capas ocultas, cada una con 15 neuronas, y una capa de salida con una sola neurona. `activation_function` es la función de activación que se aplica en las capas ocultas de las redes individuales. `weight_initialization` y `bias_initialization` se refieren a los métodos para inicializar los valores de los pesos y los biases, respectivamente. Las demás variables son hiperparámetros que determinan el algoritmo de optimización que se utilizará para entrenar a la red neuronal.

En la parte inferior de *control.in* se enlista la información de cada tipo de átomo en el sistema, además de la ubicación de los archivos que contienen los parámetros de las funciones de simetría. En el Apéndice se muestra un ejemplo del formato que los archivos *.smf* deben tener.

En Código 3 se muestra el código de `training`. El procedimiento que se sigue al ejecutar esta función es el siguiente:

- Primero se llama a `read_data` para obtener todos los datos necesarios.
- Luego se inicializa una instancia de la clase `Atom` por cada átomo que hay en el sistema.
- Después se obtienen las listas de átomos vecinos con la función `compute_neighbors_list`.
- Con `process_training_data` se calculan y normalizan las funciones de simetría.
- Luego se crea la arquitectura de la red neuronal con `create_model` y entonces se llama a la función `fit` de Keras para entrenar a la red neuronal.
- Al finalizar el entrenamiento se llama a la función `save_weights` de Keras para guardar los pesos y biases optimizados en un archivo HDF5 llamado *weights.h5*.
- Con `save_loss` se guardan el error de entrenamiento y el error de validación en los archivos *loss.out* y *val_loss.out*, y con la función `plot_loss` se hacen las gráficas del error obtenido durante el entrenamiento.
- Por último, con `get_weights_file` se guardan los pesos y biases optimizados en archivos de texto llamados *weights.x.data*, donde x es un número atómico. Como se mencionó antes, cada tipo de átomo en el sistema tiene su propia red neuronal y por tanto su propio conjunto de pesos y biases.

```python
def training(control_file, save):

    # leer parametros y posiciones iniciales
    read control parameters, SF parameters and initial positions
    params, frames, G = read_data(control_file)

    # inicializar atomos y generar listas de atomos vecinos
    framesList = []
    for frame in frames:
        AtomsList = []
        for i in range(len(frame[:-1])):
            temp_atom = Atom(i, frame[i]) # atomo central
            for g in range(len(G)):
                if temp_atom.symbol == G[g][0]:
                    temp_atom.atomic_number(params['atomic_number'][g])
                    temp_atom.atomic_mass(params['atomic_mass'][g])
            AtomsList.append(temp_atom)

        # listas de atomos vecinos
        AtomsList = compute_neighbors_list(AtomsList, params, G)
        AtomsList.append(frame[-1])
        framesList.append(AtomsList)

    # obtener tensores y factores de escala
    data_x, data_y, scale_outputs = process_training_data(framesList, params,
     G, save)

    # crear modelo de la red neuronal
    model = create_model(params, G)

    # entrenar red neuronal
    history = model.fit(data_x, data_y,
                        epochs = int(params['number_of_epochs']),
                        batch_size = int(params['minibatch_size']),
                        validation_split = float(params['validation_fraction'
    ]) )
    print('Training of neural network: Done')

    # salvar modelo y parametros optimizados
    model.save_weights('outputs/weights.h5')
    save_model(model, 'outputs/trained_model.h5')

    # salvar y graficar el error
    loss, val_loss = save_loss(history, scale_outputs)
    plot_loss(loss, val_loss, model)

    # crear archivos con los parametros optimizados
    get_weights_file(model, params, G)

    return
```
Código 3: Función *training*.

Este código se utilizó para generar la superficie de energía potencial total de un sistema compuesto por 30 átomos de aluminio (Al), 94 átomos de hidrógeno (H) y 98 átomos de oxígeno (O). Para cada átomo se calculó un conjunto de 78 funciones de simetría. El algoritmo de optimización elegido para el entrenamiento de la red fue Adagrad, que es una variante del descenso del gradiente (Anil et al., 2019). Para entrenar a la red neuronal se utilizó un conjunto de datos compuesto por 2000 muestras.

En la Figura 18 se observan las gráficas del error (o costo) obtenidas durante el entrenamiento. Como se pude observar, el error de entrenamiento (color azul) desciende desde el inicio del entrenamiento mientras que el error de validación (color naranja) oscila alrededor de los 200 meV/átomo. Puede que hayan aplicaciones en las que un error de 0.2 eV/átomo sea aceptable, pero lo más deseable es reducir el error lo más que sea posible. Para conseguir eso se podrían seguir varias técnicas, como aumentar el tamaño del conjunto de datos o utilizar un algoritmo de optimización distinto. Otra manera en la que se podría reducir el error del modelo es conseguir un conjunto de datos apropiado para entrenar a la red, para lo cual es necesario realizar un análisis *a priori* del sistema bajo estudio (Lorenz et al., 2004). Esto quiere decir que se deben seleccionar datos que contengan información sobre las propiedades más relevantes del sistema.

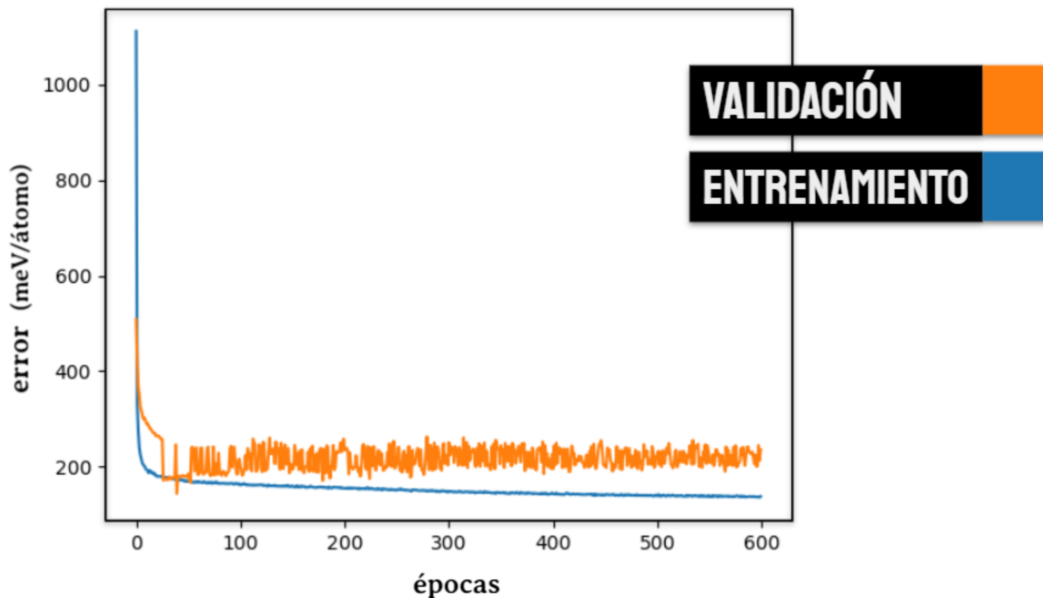


Figura 18: Gráficas del error (o costo) obtenidas durante el entrenamiento y validación de la red neuronal.

Por otro lado, el tiempo que el entrenamiento tardó en finalizar fue de alrededor de 20 horas. Al final de este trabajo se mencionan algunas sugerencias con las que se podría reducir el tiempo de ejecución considerablemente.

6.2. Código para las simulaciones de dinámica molecular

Después de haber entrenado a la red neuronal artificial con `training` se obtiene un conjunto de pesos y biases optimizados, y estos parámetros son los que determinan a la superficie de energía potencial del sistema. Así pues, lo que sigue es utilizar esta superficie para calcular las fuerzas sobre los átomos y posteriormente obtener las nuevas posiciones atómicas. La función que se encarga de esto es `molecular_dynamics` y se utiliza de la siguiente manera:

```
import molecular_dynamics
molecular_dynamics.molecular_dynamics('control.in')
```

El único argumento que esta función requiere es la ubicación de un archivo *control.in* parecido al que se utilizó con `training`. En Código 4 se muestra un ejemplo del formato que debe tener este archivo cuando se va a utilizar para ejecutar una simulación de dinámica molecular. La variable `model_path` se refiere a la ubicación del archivo donde se guardaron los parámetros optimizados obtenidos al finalizar el entrenamiento de la red neuronal. `iteration` se refiere a las iteraciones que se realizarán durante la simulación y `timestep` indica cada cuánto tiempo se deberán actualizar las posiciones atómicas.

```
1  ##########################################################################
2  # Col  Description
3  ##########################################################################
4  # 1    Atomic symbol
5  # 2    Atomic number
6  # 3    Atomic mass
7  # 4    Path of the file with the symmetry functions parameters
8  ##########################################################################
9
10 cut_function_type              2
11 angular_sf                     wide
12 atomic_data_path               data/input.data
13 length_unit                    Ang
14 number_of_atom_types           3
15 model_path                     outputs/trained_model.h5
16 neurons_per_layer              [15, 15, 1]
17 activation_function            hyperbolic_tangent_with_linear_twist
18 iterations                     20
19 timestep                       5
20
21 ##########################################################################
22 #                1    2    3      4
23 ##########################################################################
24
25 atom_type_1     H    1    1.0008    H-SymFunc.smf
26 atom_type_2     O    8    15.999    O-SymFunc.smf
27 atom_type_3     Al   13   26.982    Al-SymFunc.smf
```

Código 4: Formato que *control.in* debe tener para correr una simulación de dinámica molecular.

La función `molecular_dynamics` llama repetitivamente a otra función llamada `compute_forces` (Código 5), la cual realiza lo siguiente:

- Obtiene las listas de átomos vecinos con `compute_neighbors_list`
- Calcula las funciones de simetría y sus respectivas derivadas con `features`.
- Con `get_weights` se leen los parámetros optimizados de *weights.h5* y con ellos se reconstruye la superficie de energía potencial. A partir de esa superficie se calculan las derivadas de la energía con `energy_derivative`.
- Finalmente se utilizan las derivadas para calcular las fuerzas atómicas.

```
1  def compute_forces(atoms, params, G):
2
3      # listas de atomos vecinos
4      atoms = compute_neighbors_list(atoms, params, G)
5
6      # funciones de simetria G y sus derivadas dGdr
7      atoms = features(atoms, params, G, derivatives=True)
8
9      # derivadas de la energia dEdG
10     atoms = energy_derivative(atoms, params, G)
11
12     # calcular fuerzas atomicas
13     for atom_i in atoms:
14         temp_forces = np.zeros(3)
15         # sumar contribucion del atomo central
16         for l in range(len(atom_i.G)):
17             temp_forces -= atom_i.dEdG[l] * atom_i.dGdr[l]
18         # sumar contribucion de atomos vecinos
19         for j in range(len(atoms)):
20             atom_j = atoms[j]
21             if atom_j.index in atom_i.neighbors:
22                 for i in range(len(atom_j.neighbors)):
23                     if atom_i.index == atom_j.neighbors[i]:
24                         n_index = i
25                         for l in range(len(atom_j.G)):
26                             temp_forces -= atom_j.dEdG[l] * atom_j.dGdn[
       n_index][l]
27         atom_i.force = temp_forces
28
29     return atoms
```

Código 5: Función *compute_forces*

En Código 6 se muestra el código de `molecular_dynamics`. Lo primero que sucede al ejecutar esta función es que se llama a `read_data` para obtener todos los datos necesarios y luego se inicializa una instancia de la clase `Atom` por cada átomo que hay en el sistema. De aquí en adelante los siguientes pasos se repiten iterativamente.

- Actualización de las posiciones atómicas.
- Cálculo de las fuerzas atómicas con `compute_forces`.
- Actualización de las velocidades atómicas.

Más específicamente, en una iteración de `molecular_dynamics` se tienen que calcular los siguientes términos para cada átomo-i:

1. G: Funciones de simetría del átomo central (radiales y angulares).
2. $dGdr$: Derivada de las funciones de simetría del átomo central respecto a las coordenadas $\vec{r}_i = (x_i, y_i, z_i)$.
3. $dGdn$: Derivada de las funciones de simetría de los átomos vecinos respecto a las coordenadas $\vec{r}_i$.
4. $dEdG$: Derivada de la energía respecto a las funciones de simetría del átomo central.

Luego se calculan las fuerzas atómicas con esos datos (Behler, 2011)

$$\vec{F}_\alpha = -\nabla_\alpha E = -\sum_{t=1}^{T}\sum_{i=1}^{N_t}\sum_{l=0}^{M_i} \frac{\partial E_i^t}{\partial G_{il}^t}\frac{\partial G_{il}^t}{\partial \vec{r}_\alpha} \tag{70}$$

y finalmente se calculan las ecuaciones de Verlet para obtener las nuevas posiciones atómicas:

$$\mathbf{r_i}(t_n + \delta t) = \mathbf{r_i}(t_n) + \delta t \mathbf{v_i}(t_n) + \frac{\delta t^2}{2}\frac{\mathbf{F_i}}{m_i}(t_n) \tag{71}$$

$$\mathbf{v_i}(t_n + \delta t) = \mathbf{v_i}(t_n) + \delta t \frac{\mathbf{F_i}(t_n + \delta t) + \mathbf{F_i}(t_n)}{2m_i} \tag{72}$$

```
def molecular_dynamics(control_file):

    # leer parametros y posiciones iniciales
    params, positions, G = read_data(control_file)


    # inicializar atomos
    AtomsList = []
    for i in range(len(positions[0][:-1])):
        temp_atom = Atom(i, positions[0][i]) # atomo central
        for g in range(len(G)):
            if temp_atom.symbol == G[g][0]:
                temp_atom.atomic_number(params['atomic_number'][g])
                temp_atom.atomic_mass(params['atomic_mass'][g])
        AtomsList.append(temp_atom)

    # simulacion de dinamica molecular
    dt = float(params['timestep'])

    with open('outputs/traj.data', 'w') as f:
        for it in range(1,params['iterations']+1):
            f.write('{}\n'.format(it))

            # actualizar posiciones
            f_t = []
            for atom in AtomsList:
                f_t.append(atom.force)
                atom.position = atom.position + atom.velocity*dt + atom.force
    /2/atom.mass*dt*dt
                r = atom.position
                f.write('{}\t'.format(atom.index))
                f.write('{}\t'.format(r[0]))
                f.write('{}\t'.format(r[1]))
                f.write('{}\n'.format(r[2]))

            # actualizar fuerzas
            AtomsList = compute_forces(AtomsList, params, G)

            # actualizar velocidades
            for i in range(len(AtomsList)):
                AtomsList[i].velocity = AtomsList[i].velocity  + (AtomsList[i
    ].force + f_t[i])/2/AtomsList[i].mass*dt

            print('Iteration {}/{}: Done'.format(it, params['iterations']))


    return
```

Código 6: Función *molecular_dynamics*

Con la función `molecular_dynamics` se crea un archivo de texto llamado *traj.data* en el que se guardan las posiciones atómicas obtenidas en cada iteración de la simulación.

Este programa se utilizó para hacer una simulación con el mismo sistema atómico de aluminio, hidrógeno y oxígeno. Los parámetros optimizados que se utilizaron para reconstruir a la superficie de energía potencial son los que se obtuvieron tras entrenar a la red con `training`. Esta simulación tuvo una duración de 20 iteraciones y el intervalo de tiempo dt que se eligió fue de 5 segundos. En la Figura 19 se muestran algunas capturas de esta simulación de dinámica molecular, las cuales se obtuvieron al graficar las posiciones atómicas del archivo *traj.data*. Las partículas de color verde representan a los átomos de aluminio, las partículas moradas a los átomos de oxígeno y las partículas grises a los átomos de hidrógeno.

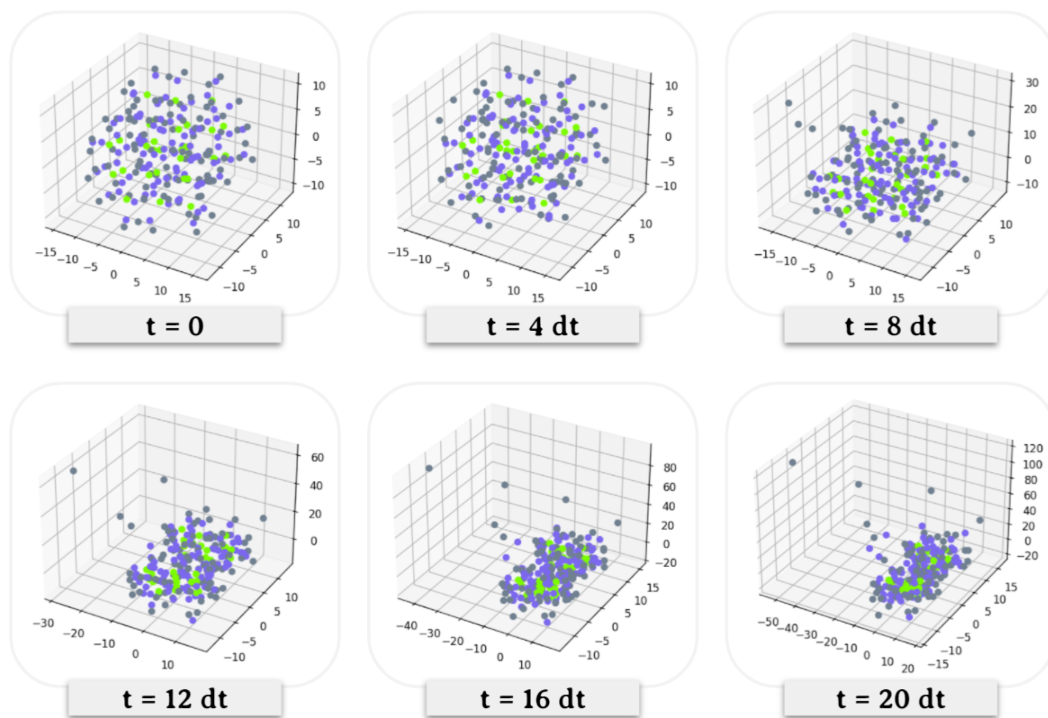


Figura 19: Capturas de una simulación de dinámica molecular realizada sobre un sistema de aluminio, oxígeno e hidrógeno.

6.3. n2p2 y LAMMPS

Las redes neuronales artificiales se han aplicado en la realización de simulaciones de dinámica molecular desde hace algunos años, por lo que no es de sorprender que ya existan algunos programas con los que se puedan llevar a cabo simulaciones de este tipo. Algunos ejemplos son RuNNER (Behler, 2018), ænet (Artrith & Urban, 2018) y Amp (Peterson & Khorshidi, 2017). Otro programa es n2p2, una paquetería desarrollada por Andreas Singraber (Singraber, 2019). Esta paquetería, escrita en C++, cuenta con varias funciones cuya finalidad es la de generar una superficie de energía potencial con redes neuronales artificiales a partir de funciones de simetría (Behler & Parrinello, 2007), justo de la misma manera que se presentó en este trabajo. Adicionalmente n2p2 cuenta con una interfaz que le permite conectarse con el simulador de dinámica molecular LAMMPS (*Large-scale Atomic/Molecular Massively Parallel Simulator*) (Plimpton et al., 2004), desarrollado en Sandia National Laboratories. A través de esta interfaz LAMMPS puede utilizar la superficie de energía potencial generada por n2p2 para calcular las fuerzas atómicas y con ellas actualizar las posiciones atómicas, de esta manera se puede llevar a cabo una simulación de dinámica molecular utilizando estos dos programas. En el Apéndice se incluye una sección donde se describen los pasos que se deben seguir para instalar a n2p2 y a LAMMPS.

Para generar una superficie de energía potencial con n2p2 se necesita un archivo *input.data* con las posiciones atómicas y las energías totales del sistema bajo estudio (exactamente igual al archivo *input.data* que se utilizó a lo largo de este trabajo) y un archivo de control llamado *input.nn*, donde se especifican los parámetros tanto del sistema atómico como de la arquitectura de la red neuronal y del algoritmo de optimización con el que se entrenará a la red.

Tras la instalación y construcción de n2p2 se deben crear varios ejecutables, cada uno de los cuales realiza un paso en el proceso para generar la superficie de energía potencial. Los pasos de este procedimiento son (Singraber, 2019):

1. Normalizar los datos que se utilizarán en el entrenamiento de la red. En una carpeta donde se encuentren los archivos *input.data*, *input.nn* y el ejecutable `nnp-norm` se debe correr el siguiente comando desde una terminal: `./nnp-norm`

2. Calcular las funciones de simetría. El ejecutable que realiza esta acción es `nnp-scaling`. Durante la ejecución de este programa se irán creando histogramas con las distribuciones de las funciones de simetría. Cuando se ejecute a `nnp-scaling` se debe especificar el número de contenedores en los histogramas, por ejemplo: `./nnp-scaling 500`

3. Entrenar a la red neuronal. El ejecutable para este paso es `nnp-train` y se debe usar en la carpeta donde estén *input.data*, *input.nn* y el archivo *scaling.data*, que contiene el máximo, mínimo, promedio y desviación estándar de las funciones de simetría. Ese último archivo se debió haber creado en el paso anterior. Para ejecutar el entrenamiento se utiliza el comando `./nnp-train`

Para comparar el desempeño del código que se presentó en este trabajo con el de n2p2 se creó con cada uno de estos códigos la superficie de energía potencial de un sistema compuesto por 30 átomos de aluminio, 94 átomos de hidrógeno y 98 átomos de oxígeno. El conjunto de datos que se utilizó para generar ambas superficies estaba compuesto por diez configuraciones del sistema atómico (es decir, se utilizaron diez muestras para entrenar a las redes neuronales).

Luego se tomaron los pesos y los biases optimizados obtenidos con el entrenamiento de n2p2 y se utilizaron (junto con el ejecutable `nnp-predict`) para predecir la energía potencial total de una configuración del sistema. Después se hizo lo mismo, pero con los pesos y biases optimizados obtenidos con el código de este trabajo. En la Figura 20 se muestran los resultados obtenidos en ambos casos.

	n2p2	tesis
duración del cálculo de las funciones de simetría	< 5 segundos	40 minutos
duración del entrenamiento	8.5 minutos	< 1 minuto
predicción de la energía potencial total (eV)	-14 409.4452	-14 409.4253
diferencia entre la energía real y la predicción (eV)	0.014241	-0.00563

Figura 20: Comparación de los resultados obtenidos al generar una superficie de energía potencial y utilizarla para predecir la energía de un sistema atómico con la paquetería n2p2 y con el código presentado en este trabajo.

El sistema atómico que se consideró para hacer las pruebas estaba compuesto por 222 átomos, a cada uno de los cuales le correspondían 78 funciones de simetría. Eso significa que las funciones de simetría que se tuvieron que calcular para las 10 muestras que se utilizaron en el entrenamiento de la red fueron $222 \times 78 \times 10 = 173,160$. El tiempo que a n2p2 le tomó realizar esos cálculos fue de sólo unos pocos segundos, mientras que el código aquí presentado se tardó 40 minutos. Dado que n2p2 está escrito en C++ y el código aquí presentado fue hecho en Python es de esperar que n2p2 sea notablemente más veloz. Una de las razones de por qué sucede esto es que se debe acceder constantemente a la memoria para

actualizar los datos de los átomos, y para esa tarea C++ es mucho más eficiente que Python (Zehra et al., 2020). En el caso del tiempo de entrenamiento, la duración fue menor que la de n2p2 porque se utilizó a Keras, que es una librería escrita especialmente para trabajar con redes neuronales artificiales con la mayor eficiencia.

Por otro lado, las predicciones que se realizaron tanto con n2p2 como con el código de este trabajo fueron bastante precisas a pesar de haber utilizado pocas muestras en el entrenamiento. El valor real de la energía potencial era de -14 409.4309 eV, por lo que las predicciones obtenidas con ambos métodos tuvieron una precisión de casi el 100 %.

7. Conclusión y trabajo futuro

Las redes neuronales artificiales han encontrado una aplicación en la química, física y en la ciencia de materiales, particularmente en la generación de superficies de energía potencial (Behler, 2016), lo que ha permitido estudiar materiales con una precisión de cálculos *ab initio* (Morawietz et al., 2016). Aquí se presentó un código basado en la generación de superficies de energía potencial a partir de redes neuronales artificiales y funciones de simetría (Behler & Parrinello, 2007), el cual demostró predecir la energía potencial total de un sistema con una gran precisión pero a costa de un tiempo de ejecución considerable. De los resultados obtenidos al hacer la comparación con n2p2 (Singraber, 2019) es evidente que un programa escrito en C++ para el cálculo de las funciones de simetría es más veloz que uno escrito en Python. Sin embargo una razón por la que se debe insistir en crear un código con Python es la posibilidad de utilizar librerías como Keras (Chollet et al., 2015) y Tensorflow (Abadi et al., 2015), con las que construir y entrenar redes neuronales es mucho más simple y sencillo que con C++. Esto tiene además la ventaja de que se pueden experimentar con diferentes arquitecturas e incluso con diferentes tipos de redes neuronales (por ejemplo, las redes convolucionales).

El procedimiento que se expuso para generar superficies de energía potencial existe desde hace varios años (Behler & Parrinello, 2007) y desde entonces ha seguido evolucionando para hacerse más rápido y eficiente. Entre las modificaciones que se han sugerido está la de sustituir las funciones de simetría por funciones de simetría polinomiales (Bircher et al., 2021) con lo que se lograría aumentar la velocidad y precisión de los cálculos. Otra sugerencia es la de implementar filtros Kalman durante el entrenamiento de la red neuronal para conseguir que las fuerzas atómicas resultantes sean más precisas (Singraber et al., 2019). Estas y otras modificaciones se podrían aplicar en el código aquí presentado para volverlo más rápido y eficiente. Otra forma con la que se podría acelerar el código es utilizar librerías como CuPy, Numba o CUDA (nvidia, 2021) con las que se puedan llevar a cabo múltiples cálculos de manera paralela, lo cual tiene el potencial de acelerar drásticamente los cálculos cuando se ejecute en un GPU.

Otra adición que se le puede hacer al código es una interfaz que pueda conectarse a un programa de dinámica molecular, como LAMMPS (Plimpton et al. 2004) o HOOMD-blue (The Glotzer Group HOOMD-blue, 2018), para aprovechar todas las funcionalidades que ese tipo de programas ofrecen, pues de esa manera se podrían analizar las propiedades macroscópicas de los sistemas sobre los que se hagan las simulaciones.

Y por último, una de las motivaciones detrás de este trabajo es la de demostrar e incentivar la aplicación de técnicas de la inteligencia artificial en la ciencia. Existen múltiples ejemplos de esta unión entre ciencia e inteligencia artificial no sólo en las simulaciones de dinámica molecular, sino también en otras áreas de la física, química, astrofísica, biología y medicina. Hace algunos años la inteligencia artificial tuvo un resurgimiento ocasionado por el avance en la capacidad de cómputo (e.g. GPUs) y desde entonces ha estado en un estado constante de crecimiento y evolución que la ha hecho llegar hasta un punto en el que es capaz de

realizar cosas que antes sólo parecían ficción. Espero que con este trabajo el lector se haya podido dar una idea del potencial que la inteligencia artificial tiene para impulsar y acelerar el desarrollo de la ciencia.

8. Apéndice

8.1. Formato de los archivos *.smf*

Los archivos con la extensión *.smf* contienen los parámetros de las funciones de simetría. Por cada tipo de átomo debe haber uno de estos archivos. A continuación se muestra un ejemplo del archivo *.smf* con los parámetros para los átomos de hidrógeno. De acuerdo con este ejemplo, cada átomo de hidrógeno tiene 12 funciones de simetría radiales (G2) y 24 funciones de simetría angulares (G3).

```
# CentralAtom H

# Type G2
# j-atom   eta   Rs   Rc
H   0.005000  0.0  12.00
H   0.008000  0.0  12.00
H   0.013000  0.0  12.00
H   0.021000  0.0  12.00

O   0.005000  0.0  12.00
O   0.008000  0.0  12.00
O   0.013000  0.0  12.00
O   0.021000  0.0  12.00

Al  0.005000  0.0  12.00
Al  0.008000  0.0  12.00
Al  0.013000  0.0  12.00
Al  0.021000  0.0  12.00

# Type G3
# j-k-atoms   eta   lambda   zeta   Rc
H   O   0.0      1.0    1.0     12.0
H   O   0.0     -1.0    1.0     12.0
H   O   0.0      1.0    2.0     12.0
H   O   0.0     -1.0    2.0     12.0
H   H   0.0      1.0    1.0     12.0
H   H   0.0     -1.0    1.0     12.0
H   H   0.0      1.0    2.0     12.0
H   H   0.0     -1.0    2.0     12.0
H   Al  0.0      1.0    1.0     12.0
H   Al  0.0     -1.0    1.0     12.0
H   Al  0.0      1.0    2.0     12.0
H   Al  0.0     -1.0    2.0     12.0
Al  Al  0.0      1.0    1.0     12.0
Al  Al  0.0     -1.0    1.0     12.0
Al  Al  0.0      1.0    2.0     12.0
Al  Al  0.0     -1.0    2.0     12.0
Al  O   0.0      1.0    1.0     12.0
Al  O   0.0     -1.0    1.0     12.0
Al  O   0.0      1.0    2.0     12.0
Al  O   0.0     -1.0    2.0     12.0
O   O   0.0      1.0    1.0     12.0
O   O   0.0     -1.0    1.0     12.0
O   O   0.0      1.0    2.0     12.0
O   O   0.0     -1.0    2.0     12.0
```

8.2. Instalación de LAMMPS y n2p2

Para realizar simulaciones con LAMMPS y n2p2 es indispensable que las librerías GSL y Eigen estén instaladas en el sistema. En caso de que alguna de estas librerías no esté instalada, se deben ejecutar los siguientes comandos en una terminal:

a) Para instalar GSL:

```
wget http://ftp.gnu.org/pub/gnu/gsl/gsl-2.3.tar.gz
tar -zxvf gsl-2.3.tar.gz
cd gsl-2.3/
./configure
make
make check
sudo make install
```

Nota: Si el enlace desde el que se descarga a GSL con `wget` ya no existe en el momento en el que se este leyendo este trabajo, entonces se tiene que descargar el paquete desde la página https://ftp.wayne.edu/gnu/gsl/

b) Para instalar Eigen:

```
sudo apt-get install libeigen3-dev
```

Cuando se inició este trabajo, la instalación y construcción de la interfaz entre LAMMPS y n2p2 se debía realizar manualmente. Sin embargo, después se agregó en n2p2 una funcionalidad para crear esa interfaz de manera automática. Los pasos de la instalación y la construcción de la interfaz siguiendo este último método son:

```
git clone https://github.com/CompPhysVienna/n2p2.git n2p2
cd n2p2/src
make libnnp
make lammps-nnp
make libnnptrain
cd application
make
```

Tras la instalación de los programas, en la carpeta `n2p2/bin` se creará el ejecutable `lmp_mpi` de LAMMPS para correr simulaciones de dinámica molecular y los ejecutables de todos los programas contenidos en n2p2, entre los cuales están:

- `nnp-norm`: Normaliza los datos
- `nnp-scaling`: Calcula las funciones de simetría

- `nnp-train`: Entrena una red neuronal para generar una superficie de energía potencial

Referencias

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E. & otros. (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.* https://www.tensorflow.org/

[2] Allen, M. P. (2004) Introduction to Molecular Dynamics Simulation. *Computational Soft Matter: From Synthetic Polymers to Proteins, Lecture Notes*. Vol. 23, ISBN 3-00-012641-4, pp. 1-28.

[3] Artrith, N., & Urban, A. (2016). An implementation of artificial neural-network potentials for atomistic materials simulations: Performance for TiO2. *Computational Materials Science, Volume 114, 2016, Pages 135-150*. DOI: https://doi.org/10.1016/j.commatsci.2015.11.047.

[4] Artrith, N., & Urban, A. (2018) *aenet: Atomic Inter-action Potentials Based on Artificial Neural Networks,Version 2.0.3.* http://ann. atomistic.net

[5] Behler, J., & Parrinello, M. (2007). Generalized Neural-Network Representation of High-Dimensional Potential-Energy Surfaces. *Phys. Rev. Lett. 98, 146401*. DOI: 10.1103/PhysRevLett.98.146401

[6] Behler, J. (2011). Atom-centered symmetry functions for constructing high-dimensional neural network potentials. *The Journal of Chemical Physics* 134, 074106. DOI: 10.1063/1.3553717

[7] Behler, J. (2011). Neural network potential-energy surfaces at chemistry: a tool for large-scale simulations. *Phys. Chem. Chem. Phys.*, 13, 17930-17955.

[8] Behler, J., Lorenz, S., & Reuter, K. (2013). Representing molecule-surface interactions with symmetry-adapted neural networks. *J Chem Phys*. 2007 Jul 7;127(1):014705. DOI: 10.1063/1.2746232.

[9] Behler, J. (2014). Representing potential energy surfaces by high-dimensional neural network potentials. *J. Phys.: Condens. Matter* 26 183001

[10] Behler, J. (2015). Constructing High-Dimensional Neural Network Potentials: A Tutorial Review. *International Journal of Quantum Chemistry*. DOI: 10.1002/qua.24890

[11] Behler, J. (2016). Perspective: Machine learning potentials for atomistic simulations. *The Journal of Chemical Physics* 145, 170901 https://doi.org/10.1063/1.4966192

[12] Behler, J. (2018) *RuNNer - A Neural Network Code for High-Dimensional Potential-Energy Surfaces*. http://www.uni-goettingen.de/de/560580.html

[13] Berger, R. (2017). *Using Python in LAMMPS*. LAMMPS Workshop, Temple University.

[14] Bircher, M. P., Singraber, A., & Dellago, C. (2021). Improved Description of Atomic Environments using Low-cost Polynomial Functions with Compact Support. arXiv:2010.14414

[15] Blank, T. B., Brown, S. D., Calhoun, A. W., & Doren, D.J. (1995). Neural network models of potential energy surfaces. *J. Chem. Phys. 103, 4129*. DOI: https://doi.org/10.1063/1.469597

[16] Bontempi, G. (2021). *Handbook Statistical foundations of machine learning* (2° ed). Universidad Libre de Bruselas.

[17] Born, M., & Oppenheimer, R. (1927). Zur Quantentheorie der Molekeln. https://doi.org/10.1002/andp.19273892002

[18] Car, R., & Parinnello, M. (1985) Unified Approach for Molecular Dynamics and Density-Functional Theory. *Phys. Rev. Lett. 55, 2471*. DOI: 10.1103/PhysRevLett.55.2471

[19] Chollet, Francois & otros. *Keras*. GitHub. https://github.com/fchollet/keras

[20] Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research, 2121–2159*.

[21] Elias, J. S., Artrith, N., Bugnet, M., Giordano, L., Botton, G. A., Kolpak, A. M., & Shao-Horn, Y. (2016) Elucidating the Nature of the Active Phase in Copper/Ceria Catalysts for CO Oxidation. *ACS Catalysis 2016 6 (3), 1675-1679*. DOI: 10.1021/acscatal.5b02666

[22] Feng, J., & Lu, S. (2019). Performance Analysis of Various Activation Functions in Artificial Neural Networks. *Journal of Physics: Conf. Series 1237 022030*. DOI: 10.1088/1742-6596/1237/2/022030

[23] Gastegger, M., Schwiedrzik, L., Bittermann, M., Berzsenyi, F., & Marquetanda, P. (2018). wACSF—Weighted atom-centered symmetry functions as descriptors in machine learning potentials. *J. Chem. Phys. 148, 241709*. DOI: https://doi.org/10.1063/1.5019667

[24] Go, G. (2017). Why Momentum Really Works. *Distill*. DOI: 10.23915/distill.00006

[25] Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press, http://www.deeplearningbook.org

[26] Griebel, M., Knapek, S., & Zumbusch, G. (2007). *Numerical Simulation in Molecular Dynamics. Numerics, Algorithms, Parallelization, Applications*. Springer.

[27] Griffiths, D. J. (2004). *Introduction to Quantum Mechanics* (2° ed). Pearson.

[28] Holm, C. (2013). *Simulation Methods in Physics 1*. Instituto de Física Computacional, Universidad de Stuttgart. *Phys. Chem. 1995. 46; 83-108*.

[29] Jolicard, G. (1995). Effective Hamiltonian Theory and Molecular Dynamics.

[30] Khorshidi A., & Peterson, A. (2016). Amp: A modular approach to machine learning in atomistic simulations. *Computer Physics Communications 207:310-324*. DOI:10.1016/j.cpc.2016.05.010

[31] Kingma, D. P., & Ba, Jimmy. (2014). Adam: a method for stochastic optimization. *arXiv preprint arXiv:1412.6980.*

[32] Kohn, W., & Sham, J. (1965). Self-Consistent Equations Including Exchange and Correlation Effects. *Phys. Rev. 140, A1133.* DOI: 10.1103/PhysRev.140.A1133

[33] Kumar, H., &Maiti P. K. (2011).*Computational Statistical Physics: Lecture Notes, Guwahati SERC School. Chapter 6: Introduction to Molecular Dynamics Simulation.* DOI: 10.1007/978-93-86279-50-7

[34] Lewars, E. G. (2011). *Computational Chemistry: Introduction to the Theory and Applications of Molecular and Quantum Mechanics.* Springer.

[35] Likun Tan. (2017). Multiple Length and Time-scale Approaches in Materials Modeling. *Advances in Materials. Special Issue: Advances in Multiscale Modeling Approach. Vol. 6, No. 1-1, 2017, pp. 1-9.* DOI: 10.11648/j.am.s.2017060101.11

[36] Lorenz, S., Groß, A., & Scheffler, M. (2004). Representing high-dimensional potential-energy surfaces for reactions at surfaces by neural networks. *Chemical Physics Letters* 395(4-6):210-215 DOI:10.1016/j.cplett.2004.07.076

[37] Lorenz, S., Groß, A., & Scheffler, M. (2006). Descriptions of surface chemical reactions using a neural network representation of the potential-energy surface. *Phys. Rev. B 73, 115431.* https://doi.org/10.1103/PhysRevB.73.115431

[38] Mcelfresh, C. (2020). *Molecular Dynamics: Velocity Verlet.* Medium. https://python.plainenglish.io/molecular-dynamics-velocity-verlet-integration-with-python-5ae66b63a8fd

[39] Morawietz, T., Singraber, A., Dellago, C., & Behler, J. (2016). How van der Waals interactions determine the unique properties of water. *Proceedings of the National Academy of Sciences 113, 8368-8373.* DOI: 10.1073/pnas.1602375113

[40] Mueller, T., Kusne, A. G., & Ramprasad, R.. (2016). *Machine Learning in Materials Science: Recent Progress and Emerging Applications.* https://doi.org/10.1002/9781119148739.ch4

[41] Ng. A. *Machine Learning.* http://cnx.org/content/col11500/1.4/

[42] Omelyan, I. P., Mryglod, I.M., & Folk, R. (2002). Optimized Verlet-like algorithms for molecular dynamics simulations. *Phys Rev E Stat Nonlin Soft Matter Phys. 65(5 Pt 2):056706.* DOI: 10.1103/PhysRevE.65.056706. Epub 2002 May 17. PMID: 12059749.

[43] Plimpton, S. (1995). Fast Parallel Algorithms for Short-Range Molecular Dynamics. *J. Comput. Phys. 1995, 117, 1-19.* DOI: https://doi.org/10.1006/jcph.1995.1039

[44] Plimpton,S., Thompson, A., Moore, S., & Kohlmeyer, A. (2004) LAMMPS Molecular Dynamics Simulator. http://lammps.sandia.gov (visto en 04/08/2021)

[45] Plimpton, S. (2015). *A Quick Tour of LAMMPS*. 4th LAMMPS Workshop, Sandia National Labs.

[46] Polyak, B. T. (1964). Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics, 4(5), 1–17.*

[47] Scheerschmidt, K. (2006). Effective Hamiltonian Theory and Molecular Dynamics. *Theory of Defects in Semiconductors (pp.213-244).* DOI: http://dx.doi.org/10.1007/11690320_10

[48] Schwedek, T. (2017) *Determining the Potential Energy Surface of Atomic Clusters using Machine Learning*. Universidad Libre de Berlín

[49] Seritan, S., Bannwarth, C., Fales, B.S., Hohenstein, E.G., Isborn, C.M., Kokkila-Schumacher, S.I.L., Li, X., Liu, F., Luehr, N., Snyder, J., Song, C., Titov, A.V., Ufimtsev, I.S., Wang, L.-P., Martínez, T.J. (2020) TeraChem: A graphical processing unit-accelerated electronic structure package for large-scale ab initio molecular dynamics *WIREs Comput. Mol. Sci. 2020;e1494*. DOI: 10.1002/wcms.1494

[50] Singraber, A., Behler, J., & Dellago, C. (2019). Library-Based LAMMPS Implementation of High-Dimensional Neural Network Potentials. *Journal of Chemical Theory and Computation* 15 (3), 1827-1840 DOI: 10.1021/acs.jctc.8b00770

[51] Singraber, A., Behler, J., & Dellago, C. (2019). Supporting Information for Library-Based LAMMPS Implementation of High-Dimensional Neural Network Potentials.

[52] Singraber, A., Morawietz, T., Behler, J., & Dellago, C. (2019). Parallel Multistream Training of High-Dimensional Neural Network Potentials. *Journal of Chemical Theory and Computation*, 15 (5), 3075-3092. DOI: 10.1021/acs.jctc.8b01092

[53] Stende, J. (2017) *Constructing High-Dimensional Neural Network Potentials for Molecular Dynamics*. Universidad de Oslo.

[54] Tieleman, T., & Hinton, G. (2012). Lecture 6.5-rmsprop: divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning, 4(2), 26–31.*

[55] Tsai, S. H., Krech, M., & Landau, D. P. (2004). Symplectic Integration Methods in Molecular and Spin Dynamics Simulations. *Brazilian Journal of Physics*, 34, 384-391.

[56] Unke, O. T., Koner, D., Patra, S., Käser, S., & Meuwly, M. (2020). High-dimensional potential energy surfaces for molecular simulations: from empiricism to machine learning. *IOP Publishing Ltd Machine Learning: Science and Technology, Volume 1, Number 1.*

[57] Witkoskie, J. B., & Doren, D. J. (2005) Neural Network Models of Potential Energy Surfaces: Prototypical Examples. *J Chem Theory Comput. 2005 Jan;1(1):14-23.* DOI: 10.1021/ct049976i. PMID: 26641111.

[58] Zehra, F., Javed, M., Khan, D., & Pasha, M. (2020). Comparative Analysis of C++ and Python in Terms of Memory and Time. *Preprints 2020120516*. DOI: 10.20944/preprints202012.0516.v1

[59] Zhang, A., Lipton, Z. C., Li, M., & Smola, A. J. (2020). *Dive into Deep Learning*. 2020

[60] Zhang, J. (2019). Basic Neural Units of the Brain: Neurons, Synapses and Action Potential. *https://arxiv.org/abs/1906.01703*