UNIVERSIDAD NACIONAL AUTÓNOMA
DE MÉXICO

FACULTAD DE CIENCIAS

Generación de potenciales de interacción basados en Inteligencia Artificial para predecir propiedades de materiales.

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

Físico

PRESENTA:

Carlos Emilio Camacho Lorenzana

TUTOR

Dr. J. Guadalupe Pérez Ramírez

CD. MX. 2023

Agradecimientos

El fin más importante cuando se escoge una carrera universitaria es poder generar una perspectiva de la realidad que le permita a los individuos crear su propio criterio de cómo tomar decisiones a lo largo de su vida. La enseñanza más grande que me llevo de haber estudiado Física es que no hay nada mejor que trabajar en equipo para afrontar los problemas de la vida. Dicho esto, debo hacer mención de los maravillosos equipos que me han acompañado hasta este punto de mi vida y a los que les debo este logro importante en mi vida:

A mi familia nuclear, pues me han apoyado y han creído en mí en todo momento. La elección de carrera no fue nada fácil para mí, pero tengo la suerte de contar con mi papá Juan Carlos Camacho, a mi hermana Karla Edith Camacho, a mi abuela Emelia Quintero, y a mi madre, que se adelantó en el camino en el 2020, Emelia Lorenzana, quienes nunca dejaban de asombrarse de todo lo que aprendía en la carrera y quienes siempre me apoyaron en todo. Al fin, logramos este título.

A mi novia, Martha Méndez, por su infinito amor y su deseo de acompañarme en cada paso que daba en la carrera. Ella fue la principal motivación de decidir estudiar Física y me mostró la valentía requerida para tomar decisiones importantes en la vida. Muchas gracias, amor, lo logramos.

A mis mejores amigos, Rodrigo González Oro, Humberto Fonseca y Jesús Mancillas, porque a pesar de los diferentes caminos que tomamos en nuestras vidas universitarias, siempre hacíamos lo posible para mantenernos juntos. A pesar de que ellos eran las personas que más sacrificaba en cuanto a reuniones y salidas por cuestiones escolares, nunca dejaron de quererme y apoyarme en todo lo que hacía y en lo que me proponía alcanzar. Hermanos, lo logramos.

A mis amigos en la carrera, quienes me mostraron la importancia de trabajar juntos y de tener una visión crítica de todo lo que aprendimos durante la licenciatura. Karla Cecilia Gómez, Ethan Campos, Ana Castelán, Diego Morales, Rodrigo Moreno, Yazareth Peña, Maximiliano Ponce, Erasmo Hinojosa y Luis Fernando Ramos, muchas gracias por sus enseñanzas y risas, lo logramos.

A mis amigos que han perdurado a lo largo de mi vida, pues me han forjado en carácter para llegar a mis objetivos y son parte esencial de cómo me desenvuelvo en la vida. Hugo Pérez, Jairo Patiño, Giselle Lazos, Daniela Palomo y Diego Cortés, gracias por seguir siendo parte de mi vida, lo logramos.

A mis compañeros y amigos del trabajo, por las anécdotas, risas y enseñanzas que tengo con ustedes día con día y por el apoyo para concluir la tesis. Celeste Castro y Arturo García, gracias por tantas experiencias y por enriquecer mi vida de una manera que nunca imaginé, lo logramos.

Finalmente, al doctor Bokhimi y a los miembros del laboratorio LIACEI del Instituto de Física de la UNAM, en especial, a Diego Peña, Karen Daniela Cruz, Alejandra Hinostroza, Luis Ariel Fuentes y Juan Arnulfo Baltazar, por el camino que hemos recorrido para adentrarnos en el mundo de la Inteligencia Artificial y los grandes proyectos que hemos desarrollado a lo largo de dos años de convivencia y aprendizaje. Muchas gracias por compartir conmigo todo su conocimiento, lo logramos.

Índice

1. Resumen

Existen actualmente dos metodologías para encontrar potenciales de interacción en sistemas moleculares: aquellos que utilizan cálculos cuánticos para definir la energía del sistema para cada distribución atómica que conforma una trayectoria de un sistema molecular de interés, y con esa información proponen modelos cuyas predicciones se ajusten a esos cálculos realizados. Esta metodología es la más utilizada actualmente en la comunidad científica. No obstante, los potenciales obtenidos estarán limitados a la precisión con que se realizaron los cálculos que guiaron a las predicciones de dichos potenciales. Por otro lado, a partir de resultados experimentales, se pueden proponer potenciales de interacción, cuyo ajuste de parámetros debe ser de tal modo que se puedan reproducir las observables del experimento reportado. Esta metodología no ha sido tan utilizada como la anterior mencionada, ya que el ajuste de parámetros se realiza, en la mayoría de las veces, de manera manual y la sobre simplificación del potencial de interacción carece de complejidad suficiente para describir correctamente el fenómeno estudiado.

Profundizando con el tema de los potenciales simplificados, la mayoría utiliza sólo la contribución entre pares para describir el potencial de interacción global del sistema. Dichas representaciones no incluyen interacciones de tercias de elementos, ya que implican un costo computacional elevado y, hasta no hace mucho, no se contaba con las herramientas para acelerar dichos cálculos.

En este trabajo se presenta una forma de describir el potencial de interacción con resultados experimentales, utilizando Inteligencia Artificial como propuesta de potencial, con la cual se pueden incorporar los términos de tripletes.

Utilizando como ejemplo de dicha metodología un sistema de moléculas de agua, el algoritmo es capaz de predecir con alta precisión la distribución radial y angular del sistema. Aunado a estos resultados, también se introduce una teoría que justifica la creación de redes neuronales desde el punto de vista de la física teórica y con la cual se crea la red de grafos DimeNet++, la cual es capaz de recabar la información de la interacción entre los diferentes elementos moleculares para construir el potencial de interacción.

2. Introducción

2.1. Delimitación del Objeto de Estudio

En dinámica molecular, determinar una función que describa la energía potencial de un sistema, y con la cual se puedan inferir propiedades de algún arreglo molecular, ha sido un problema muy estudiado por la comunidad científica, con diversos resultados que han dado luz a generalidades de la forma de trabajo de este problema, así como soluciones que permiten ser base para futuros nuevos proyectos de investigación. Por mencionar algunos ejemplos, existen las funciones que se modelan con la forma funcional AMBER [1], los potenciales basados en CHARMM [2], entre otros. La forma genérica que adoptan estos potenciales se presenta a continuación [3]:

$$E = \sum_{\text{enlaces}} \frac{1}{2} k_b (r - r_0)^2 + \sum_{\text{ángulos}} \frac{1}{2} (a - a_0)^2 + \sum_{\text{torsiones}} \sum_{n} \frac{V_n}{2} (1 + \cos(n\phi - \delta)) + \sum_{\text{LJ}} 4\epsilon_{ij} \left[\left(\frac{\sigma_{ij}}{r_{ij}} \right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}} \right)^{6} \right] + \sum_{\text{electrostáticas}} \frac{q_i q_j}{r_{ij}} \tag{1}$$

en donde

$$\sum_{\text{enlaces}} \frac{1}{2} k_b (r - r_0)^2$$

es un potencial de oscilador armónico que mide la interacción entre los elementos que forman un enlace,

$$\sum_{\text{ángulos}} \frac{1}{2} (a - a_0)^2$$

son potenciales de ángulos,

$$\sum_{\text{torsiones}} \sum_{n} \frac{V_n}{2} (1 + \cos(n\phi - \delta))$$

mide la torsión,

$$\sum_{\text{LJ}} 4\epsilon_{ij} \left[\left(\frac{\sigma_{ij}}{r_{ij}} \right)^{12} - \left(\frac{\sigma_{ij}}{r_{ij}} \right)^{6} \right]$$

es el potencial de Lennard-Jones; y finalmente,

$$\sum_{\text{electrostáticas}} \frac{q_i q_j}{r_{ij}}$$

mide la interacción electrostática entre pares.

Profundizando en cada uno de los componentes mencionados, los parámetros $k_b, \delta, V, \epsilon_{ij}$ deben ajustarse al sistema de estudio particular de interés. No obstante, algunos de estos parámetros se obtienen mediante cálculos cuánticos, a lo que se le denomina metodología por construcción (mejor conocida en inglés como metodología bottom-up) en química teórica. Esta metodología se caracteriza por ofrecer predicciones detalladas, según se necesiten. Así lo explica el trabajo de Sun y colaboradores en la Figura 1 de su trabajo [4], donde dicha forma de trabajo se hace notar por datos de referencia obtenidos de modelos de alta precisión, como pueden ser simulaciones con cálculos cuánticos.

Como también se hace notar en el artículo mencionado previamente, el desarrollo de la metodología por construcción tuvo su auge con el desarrollo de los algoritmos de Aprendizaje Automático [4], los cuales distinguen tres métodos de ajuste de parámetros: por medio de Aprendizaje Supervisado, Aprendizaje No Supervisado y Aprendizaje por Refuerzo. De los tres métodos mencionados, el Aprendizaje Supervisado es aquel que se ha adaptado mejor a la metodología por construcción, ya que la parte Supervisada proviene de tener a priori datos considerados como verdaderos y los algoritmos a desarrollar deben ofrecer resultados con la precisión más alta respecto a estos datos verdaderos que se le ofrecen al modelo para ajustar sus parámetros.

Recientemente, las aplicaciones del Aprendizaje Automático al campo de la Física han sido muy diversas y con un gran número de modelos propuestos para dichos fines [5]. La mayoría de los trabajos incursionan en el campo del Aprendizaje Supervisado, donde se busca que un cierto modelo sea capaz de producir datos que coincidan con el conjunto de datos verdaderos, conocido mejor en términos del Aprendizaje Automático como etiquetas, y mediante esta comparación entre las predicciones hechas por la red y los datos reales, se busca optimizar los parámetros del modelo, hasta conseguir una precisión lo más alta posible. Acto seguido, el modelo debe ser capaz de realizar inferencias sobre conjuntos de datos que no fueron utilizados como parte de su método de actualización de parámetros.

El principal problema con el ajuste de parámetros bajo la metodología supervisada, es que no se obtendrá una precisión mayor de la que se tiene, por decir, con los cálculos cuánticos utilizados como variable dependiente de referencia. Por ejemplo, si se utiliza un determinado funcional para llevar a cabo un cálculo con teoría DFT [6] y con ello generar los datos de referencia para el ajuste, entonces la precisión con las que se obtengan dichos cálculos estará limitada por la precisión intrínseca de los datos de referencia obtenidos por el mencionado funcional.

Ante esta situación, se contempla otra metodología de trabajo, en donde se busca que la simulación generada por cierto modelo, que aproximará a la energía potencial, presente resultados que empaten con el experimento. Dicha metodología se conoce como metodología por definición (top-down, en inglés), como lo aclaran en el artículo mencionado previamente sobre modelos de ADN [4] y tiene como valores de referencia cantidades macroscópicas obtenidas principalmente por rigurosos experimentos. Los modelos buscan reproducir lo que se observa directamente en el experimento. Es decir, es la misma naturaleza quien define cómo se debe comportar el modelo por medio de una observable del sistema a estudiar.

Esta forma de resolver el problema ha carecido de relevancia debido a dos factores principales:

- Dada la necesidad de explicar con mayor detalle ciertos comportamientos de diversos sistemas o arreglos moleculares, las aproximaciones clásicas con pocos parámetros ya no son suficientes para capturar la complejidad adyacente a dicho sistema de estudio.

- Si se busca ayuda de los algoritmos de Aprendizaje Automático, aquellos que podrían ofrecer resultados más precisos presentaban, hasta hace no mucho, problemas de convergencia en el ajuste de sus parámetros.

Actualmente, estos dos puntos mencionados tienen una posible corrección: las metodologías experimentales ofrecen resultados cada vez más precisos, los cuales confirman las hipótesis formuladas por la Mecánica Cuántica, por ejemplo, en el caso de los estudios moleculares. Por otro lado, no fue hasta hace poco que con la técnica de Diferenciación Automática [7] que se expandieron las posibilidades de aplicar el Aprendizaje Automático con otras técnicas, más allá del Aprendizaje Supervisado.

Teniendo claro estos incidentes y sus posibles soluciones, es necesario retomar la discusión acerca de la ecuación 1. Los trabajos que consideran potenciales basados en dicha ecuación no toman en cuenta todos los términos que se describen en ella, pues recaen principalmente en las interacciones por pares, siendo la distancia inter atómica el principal rasgo para llevar a cabo el aprendizaje de los modelos [8, 9] . Los trabajos mencionados previamente utilizan redes neuronales de grafos [8], las cuales usan la distancia para predecir propiedades de los nodos del grafo. La principal razón por la cual no se consideran los términos de los tripletes es por el costo computacional que supondría realizar dichos cálculos [9].

Por consiguiente, es necesario encontrar una manera de aprovechar la información de los ángulos contenida en los tripletes para poder tener un potencial con redes neuronales, tal que pueda ser aplicada a un sistema molecular de interés y con ello predecir propiedades moleculares de dicho arreglo.

En este trabajo, se explica a detalle la construcción de una red neuronal de grafos, con ciertas adecuaciones a su estructura, de modo que cubra los requisitos mínimos explicados en la ecuación 1, aplicando dicho algoritmo a un arreglo de 901 moléculas de agua. La forma de entrenar la red neuronal también será explicado a detalle, con la finalidad de proporcionar una herramienta de generación de funciones potenciales, sin la necesidad de tener a priori cálculos cuánticos.

2.2. Propósito de la Investigación

Esta investigación busca hacer un aporte a la aplicación de la Inteligencia Artificial en el campo de la Física, utilizando los algoritmos de Aprendizaje Automático Profundo más modernos, que son las redes neuronales de grafos. Para ello, se trabajó en el lenguaje de programación Python, junto con la librería JAX [10].

Además, este trabajo busca responder la necesidad de mayor documentación y ejemplos de códigos con la librería mencionada, pues su interés radica en la utilización de tarjetas gráficas para acelerar el cálculo de los potenciales [11].

2.3. Objetivos

Se pretende realizar un código enfocado a la Dinámica Molecular del arreglo de moléculas de agua, en donde utilizando redes neuronales de grafos [12], integrados al proceso de aprendizaje por refuerzo profundo, se calcula el potencial para describir dicha dinámica. La investigación tiene dos objetivos centrales:

1. Integrar conceptos físicos de Mecánica Cuántica, Física Estadística, Termodinámica y Materia Condensada para la elaboración y justificación de una red neuronal de grafos, así como la determinación de la energía y las fuerzas del sistema, para con ello predecir propiedades independientes del tiempo de dicho arreglo: las distribuciones radiales y angulares de pares de oxígenos, así como la presión del sistema.
2. Diseñar y elaborar un código que contenga suficiente documentación de la adaptación de la librería JAX al problema de dinámica molecular [13] Así, se tienen además como objetivos secundarios:

   a) Implementación de la librería JAX MD como parte esencial del código propuesto.

   b) Comparar los tiempos de ejecución con resultados publicados relacionados con el tema.

   c) Comparar la precisión y el error obtenidos con trabajos similares.

   d) Implementar el uso de las tarjetas gráficas disponibles en el Laboratorio de Inteligencia Artificial en Ciencias Exactas e Ingenierías (LIACEI) del Instituto de Física de la UNAM.

2.4. Metodología de trabajo

Se trata de una investigación teórica que propone la implementación de códigos en el lenguaje de programación Python, con énfasis en la librería JAX, la cual permite al usuario desarrollar código propio para realizar cálculo científico con fácil adaptación para utilizar tarjetas gráficas en la ejecución de procesos numéricos.

Utilizando los elementos que se expondrán en el marco teórico, se propondrá una red neuronal para realizar simulaciones de dinámica molecular en moléculas de agua. Esta fase exploratoria presenta tres momentos:

- Búsqueda de librerías construidas en JAX para encontrar metodologías de implementación de códigos para elaborar redes neuronales, de modo que se realice directamente la optimización de hiperparámetros.
- Elaboración de redes neuronales con base en la teoría efectiva señalada en [14], en la cual basándose en conceptos físicos se definen parámetros para definir la arquitectura de una red neuronal.
- Hiperparametrización de las redes elaboradas para encontrar la mejor candidata para el entrenamiento completo.

  Finalmente, se calculan las energías y las fuerzas del arreglo mencionado.

2.5. Estructura del Trabajo

En la Parte 3. Desarrollo Teórico, se detallan los componentes teóricos que dan sustento y perspectiva a la investigación, en donde se pone en contexto los elementos de inteligencia artificial y de física que se emplearán durante la discusión de todo el proyecto. Esta parte se divide en cuatro secciones:

1. Inteligencia Artificial: Se definen los conceptos de Aprendizaje Supervisado y Aprendizaje por Refuerzo.

2. Redes Neuronales: Se exponen los componentes principales que conforman este tipo de algoritmos, así como reconocer su impacto en la física a través de los últimos años.

3. Teoría Efectiva de las Redes Neuronales: incluye conceptos que dan sustento a la manera en que se trabajarán los diferentes modelos de redes neuronales a lo largo del proyecto de investigación.

4. Física: de manera sucinta se esclarecen los términos necesarios para trabajar con los conceptos físicos que se abordarán en los modelos.

5. Algoritmo DiffTRe: El trabajo presentado en esta tesis se basa en el algoritmo DIffTRe [15], el cual presenta una manera de trabajar la metodología por definición a través de todos los conceptos que se trabajarán en los puntos anteriormente mencionados.

La Parte 3. Desarrollo Experimental, presenta a detalle todos los componentes necesarios para lograr el entrenamiento de las redes neuronales.

1. Diseño Experimental: En esta sección se aclaran la infraestructura, software y hardware empleados para lograr la correcta ejecución de los códigos.

2. Resultados: Se muestran tablas y gráficas del desarrollo expuesto en apartados anteriores.

La Parte 4. Conclusiones y Discusión: Se realiza un análisis de los resultados obtenidos y se reflexiona sobre las preguntas de investigación a la luz del marco teórico.

3. Desarrollo teórico

En este apartado se presentan los aspectos teóricos a considerar para la realización del código. Se sientan las bases de los campos de la Física y de las Ciencias de la Computación necesarias para entender el desarrollo experimental que se presenta en el apartado de Metodología. Finalmente, se conjuntan todos los puntos para la explicación detallada de cómo funciona el algoritmo DiffTRe.

3.1. Inteligencia Artificial

3.1.1. Origen y definición

El término de Inteligencia Artificial fue concebido por primera vez en 1955 con el proyecto de verano realizado por el colegio Darmouth [16]. Esta rama de la ciencia busca reproducir la inteligencia humana a través de la imitación de las acciones que se llevan a cabo para dichos fines [17].

Dentro del campo de la inteligencia artificial, existen dos escuelas de pensamiento principales, las cuales se derivan de comprender los siguientes conceptos: [18]

1. Deducción vs. Inducción: La deducción parte de lo general a lo particular; la inducción busca generalizar, a partir de lo particular. Las deducciones ofrecen conclusiones generalmente muy precisas; sin embargo, las inducciones permiten obtener conclusiones no triviales, por medio de procesos estadísticos, y que pueden ser de carácter general y preciso.

2. Razonamiento vs. Aprendizaje: Dado un conjunto de afirmaciones, el razonamiento busca establecer conexiones entre ellas para llegar a una conclusión. Por otro lado, los métodos de aprendizaje utilizan inferencia estadística, dados muchos ejemplos, para obtener conclusiones. Entre mayor sea la cantidad de datos que se recolectan, mejor será el poder predictivo realizado por aprendizaje.

Con lo anterior, tenemos entonces las siguientes ramas de pensamiento principal en inteligencia artificial:

1. Razonamiento deductivo: dado un conjunto de hipótesis, cuyos elementos son considerados como verdaderos, se busca llegar a una conclusión. Los algoritmos desarrollados bajo esta línea de pensamiento suele tener código en bruto que utiliza las hipótesis verdaderas dadas para obtener un resultado. Ejemplos del razonamiento deductivo se encuentran en los sistemas de recomendación [19].

2. Aprendizaje inductivo: Esta metodología crea las hipótesis a partir de numerosos ejemplos. Teniendo estas hipótesis, se procede a realizar predicciones en un nuevo conjunto de datos. Con este método, se busca entonces crear un algoritmo de aprendizaje que exprese las hipótesis como una función matemática, la cual calcula la probabilidad de que un cierto dato de entrada cumpla o no con dichas hipótesis:

$$\mathrm{P}(X = x) = f(\text{rasgos}) \tag{2}$$

   donde los rasgos son las características que describen a los datos de entrada, y la función f puede ser un algoritmo de Aprendizaje Automático, como pueden ser las redes neuronales que se describirán con detalle más adelante. El algoritmo se inicializa con una cierta forma genérica, que representa una primera suposición, y se ajustan los parámetros según se aprende de los ejemplos de entrada.

En el presente trabajo, se utilizará el enfoque de aprendizaje inductivo como estrategia para trabajar con los algoritmos de Aprendizaje Automático.

Una vez seleccionado el método de trabajo, se escoge ahora la manera en que se va a realizar el aprendizaje de los algoritmos:

3.1.2. Aprendizaje supervisado

Comenzando desde una perspectiva histórica, en los años 50 ya se contaba con la posibilidad de crear computadoras capaces de aprender a través de prueba y error. Pero el siguiente tópico a tratar fue el de aprendizaje generalizado y reconocimiento de patrones [20]. El trabajo original de la implementación de un Perceptrón fue de los pioneros en el desarrollo del aprendizaje supervisado [21], en donde dado un conjunto de datos con su respectivo valor a predecir, un algoritmo genera predicciones y se comparan con los valores reales de cada muestra.

En la actualidad, prevalece en la aplicación de la Inteligencia Artificial a la Física este tipo de aprendizaje, como se puede apreciar en el trabajo de Behler y Parrinello [22]. En este trabajo, así como en muchos otros casos, los valores de referencia, que utilizará el modelo para aprender, son generados principalmente mediante cálculos cuánticos llevados a cabo por la teoría DFT, que en español significa Teoría del Funcional de la Densidad, cuya técnica permite obtener soluciones aproximadas al problema de valores propios asociados al Hamiltoniano del sistema.

Para generar los valores de referencia se utiliza la Teoría del Funcional de la Densidad para generar los cálculos de dichos valores, considerando principalmente moléculas en el vacío.

Esta manera de aplicar el aprendizaje supervisado presenta dos problemas:

1. Las métricas del modelo a obtener están limitadas por la precisión con la que se generan los valores reales a través de los funcionales.
2. Si se quiere realizar un estudio con escalas mayores, como las macromoléculas, no se pueden obtener datos utilizando cálculos cuánticos, por lo que sería imposible realizar aprendizaje supervisado [15].

3.1.3. Aprendizaje no Supervisado

En este campo no existe un valor de referencia para comparar los resultados; los algoritmos buscan crear representaciones de las entradas, y con ello generar automáticamente clases o agrupamientos basados en el conjunto de datos que se les proporciona [23]. La principal característica de este aprendizaje es que busca encontrar una estructura o forma inherente de los datos de entrada que le permita realizar los agrupamientos que mejor describan al sistema.

El diseño de materiales es un ejemplo de la aplicación del aprendizaje no supervisado a la Física [24]. En este caso, se utilizan modelos de aprendizaje automático generativos para poder predecir materiales inorgánicos hipotéticos a partir de una base de datos de compuestos dada. El algoritmo aprende la estructura intrínseca detrás de los materiales que se encuentran en el conjunto de datos, y a partir de dicha estructura se proponen nuevos materiales, cumpliendo con las reglas que el algoritmo encuentra a partir del estudio de los datos de entrada.

Una desventaja de los modelos generativos es su inestabilidad para poder realizar apropiadamente los entrenamientos. En el caso de las Redes de Confrontación Generativa, el creador de dichos modelos menciona que se debe tener cuidado en ciertos detalles para lograr los resultados deseados [25]. Dichos detalles muchas veces son omitidos en los reportes de resultados, y, por lo tanto, no se logra reproducibilidad.

3.1.4. Aprendizaje por Refuerzo

La base del aprendizaje por refuerzo se dio en 1969 con el trabajo de Arthur Samuel [26], donde desarrolló juegos inspectores para la empresa IBM. Esta disciplina surge tras la convergencia de dos ramas principales:

- Una concerniente al estudio del aprendizaje a prueba y error, surgida del estudio de la psicología del aprendizaje animal. El trabajo de Thorndike es pionero en establecer este estudio con lenguaje concerniente a aprendizaje [27].

- La otra rama concierne al problema del control óptimo y su solución utilizando programación dinámica y funciones de valor [28].

La esencia que caracteriza a este tipo de aprendizaje es que se basa completamente en la interacción con el entorno. Esa característica es fundamental para aquellos problemas en física, como el que se presenta en esta tesis, en los cuales se desea describir interacciones entre elementos de un cierto sistema. El aprendizaje por refuerzo se distingue de los dos tipos de aprendizaje anteriores al no tener un agente externo que le indique los valores de referencia a los que debe ajustarse el algoritmo; además, su finalidad no es encontrar una estructura inherente de los datos de entrada o realizar agrupaciones de los diferentes elementos presentes en el conjunto de datos.

En el aprendizaje por refuerzo se define una función de costo, la cual es la acumulación de los distintos errores que se adquieren al llevar a cabo ciertas acciones. El sistema, a su vez, recibe retroalimentación positiva o negativa, según decida un cierto observador [29].

En este campo de la Inteligencia Artificial se distinguen dos entes principales: un *agente* y un *entorno*. A continuación se expondrán los diferentes componentes que regulan la interacción entre estos dos elementos.

El comportamiento del agente estará regido por una política, que corresponde a un mapeo entre estados percibidos del entorno, hacia acciones que puede llevar a cabo el agente cuando se encuentre en dichos estados. Como referencia, en el ámbito de la psicología, la política es lo que se conoce como reglas de estímulo-respuesta. [28]

El entorno se comunica con el agente a través de recompensas por cada acción llevada a cabo por el agente. Este punto es responsable de modificar la política del agente, ya que el principal objetivo del aprendizaje por refuerzo es maximizar la recompensa. En biología, la recompensa es sentir placer o dolor al realizar ciertas acciones. [28]

Si bien la recompensa se adquiere al tomar acciones a corto plazo, también se tiene en este planteamiento la acumulación de recompensas a largo plazo, a través de una función de valor, la cual asigna a cada acción la cantidad de recompensa que se podrá obtener a futuro, en caso de que se comience a partir de un cierto estado. El estado es una señal que le dice al agente cómo se encuentra el entorno a un determinado tiempo. Esta señal es generada por un sistema que se encuentra en el entorno. [28]

Finalmente, se debe contemplar un cuarto elemento, llamado modelo de entorno, que permite decidir sobre el curso de una acción, considerando situaciones futuras previas a la experimentación por parte del agente. [28]

Aparte de los elementos ya mencionados, también se tienen observaciones, que son distintas del estado. El estado contienen toda la información acerca del entorno, mientras que una observación contiene aspectos de ciertos elementos pertenecientes al entorno [30].

El principal problema a resolver en el aprendizaje por refuerzo es encontrar el balance entre explotar el conocimiento adquirido por el agente para adquirir mayor recompensa a corto plazo, y la exploración de nuevas acciones que pueden resultar o no en mejores recompensas futuras para el agente. [28].

Habiendo descrito a detalle los elementos de esta rama de la Inteligencia Artificial, se describen a continuación los elementos que compondrán al sistema de estudio de esta tesis y se indicará puntualmente qué representan dichos elementos bajo el esquema del planteamiento del aprendizaje por refuerzo, descrito en los párrafos anteriores.

3.2. Redes Neuronales

Una red neuronal artificial es una máquina con habilidad de adaptación, la cual está diseñada a través de la interconexión de neuronas artificiales que actúan como procesadores [29]. A través de un proceso de aprendizaje de su entorno, la red actualiza los pesos asociados a cada conexión entre neuronas hasta conseguir una cierta precisión [23]. Algunas de las características de las redes neuronales son: linealidad, transformación de las entradas a las salidas, adaptabilidad, uniformidad en diseño y análisis, y analogía con las redes biológicas.

Una neurona (artificial) es la unidad fundamental de procesamiento con la que se forman las redes neuronales. Las neuronas se conforman de los siguientes elementos: [31]

1. Sinapsis: Estos puentes de conexión se localizan en los elementos receptores, conocidos como dendritas, de la neurona artificial. Se parametrizan por los pesos w a actualizar en cada paso del aprendizaje.
2. Adición: Se recopila la información aportada por los elementos de entrada, a través de la suma de los elementos de la forma $x_i w_{ni}$. El resultado de dicha suma es conocido como pre-activación.
3. Función de activación: Define la salida de la neurona. Toma como argumento la suma definida anteriormente.
4. Sesgo: Es un desplazamiento que se le añade a la pre-activación.

Para que el modelo ajuste adecuadamente los pesos, la red necesita un proceso de adaptación, que se define a través de un algoritmo de aprendizaje [32]. Como se explicó en la sección anterior, las diferentes maneras en las que se puede llevar a cabo este aprendizaje es mediante el aprendizaje supervisado, el aprendizaje no supervisado y el aprendizaje por refuerzo.

3.2.1. Redes Neuronales de Grafos

Los grafos permiten establecer relaciones entre pares de nodos, a través de la conexión entre dichos pares, la cual se denomina arista. Esta manera sencilla de relacionar información permite representar diversas fuentes de información con un grafo. Las redes neuronales de grafos trabajan sobre estas representaciones, ya sea a nivel grafo o a nivel nodo [33].

En aplicaciones a nivel grafo, la meta es proponer una función τ que realice una determinada predicción con base en la información estructurada como grafo, siendo esta función independiente de cada uno de los nodos. Poniendo el ejemplo en la química, se modela un compuesto químico a través de un grafo $\mathbf{G}$, donde los nodos representan los átomos, mientras que las aristas de estos representan enlaces químicos. La función $\tau(\mathbf{G})$ indica la probabilidad de que dicho compuesto ocasione una determinada enfermedad [34].

En aplicaciones a nivel nodo, la función τ sí depende de las propiedades de cada nodo. Por ejemplo, se puede obtener una clasificación de nodos, como pueden ser documentos, a partir de un grafo que represente una red de citas bibliográficas [35].

En Aprendizaje Automático, un paso importante para poder trabajar con grafos es llevar a cabo un preprocesamiento, el cual permita mapear el grafo en una estructura más sencilla, como un vector [36]. A pesar de que dicha representación debe ser cuidadosa, de modo que no se pierdan propiedades fundamentales del grafo, esto representa un cambio de paradigma en la manera tradicional en la que se predecían propiedades moleculares en física y química, pues se tenían que hacer representaciones a mano de la vecindad atómica. Por mencionar un caso, está la representación con redes neuronales que proponen Behler y Parrinello en su trabajo [22].

Las redes neuronales de grafos representan a la molécula como un grafo, cuyos nodos son los átomos y las aristas se definen a través de un grafo molecular predefinido o, simplemente, se conectan átomos entre sí que se encuentren dentro de una vecindad de radio c. Luego, las aristas codifican la distancia entre átomos conectados por la representación de grafo.

3.2.2. DimeNet++: red de grafos para la descripción de potenciales de interacción

La principal disyuntiva para utilizar grafos en problemas de Física es que únicamente toman en cuenta la distancia entre pares para crear la representación del sistema. Dicho esto, la representación generada es insuficiente para poder predecir la energía potencial del sistema, ya que la función que describe esta energía considera cuatro aportaciones [37]:

$$E = E_{\text{enlaces}} + E_{\text{ángulo}} + E_{\text{torsión}} + E_{\text{no enlaces}} \tag{3}$$

Si solo se toman en cuenta las distancias, entonces hacen falta los términos referentes a los ángulos, así como información direccional que se deba incluir en el término de torsión [38].

Ahora, no es fácil extender el grafo para que tome en cuenta la información de los ángulos, ya que la dependencia exclusiva de la distancia permite invariancia traslacional, rotacional y de inversión de la molécula. Para poder tomar en cuenta estos elementos, se propone una red neuronal llamada "Directional Message Passing Neural Network", que se nombrará a partir de ahora como DimeNet++ [39], y cuya propuesta formal es representar a los átomos como un conjunto de mensajes que se transmiten entre sí, y que al juntar un cierto cúmulo de ellos, pasan un mensaje global a otro átomo, y así sucesivamente.

A continuación, se clarifican algunos puntos relacionados con la red DimeNet:

- Incrustaciones: Los átomos se representan como un vector de una determinada dimensión, el cual se denomina en la literatura como incrustaciones (embedding, en inglés). La propuesta de la manera de actualizar los valores de dichas representaciones es a través de la información que los átomos vecinos pueden proporcionar al átomo de interés.

- Inspiración de la red DimeNet++: Como se hizo notar en la ecuación 3, es necesario describir mejor la interacción entre los elementos del sistema. Para ello, el algoritmo de propagación de mensajes permite hacer la inferencia sobre propiedades en los nodos de una red, con base en la información de los nodos vecinos [40]. Este algoritmo manda mensajes entre los nodos conectados de un cierto grafo, de modo que se pueda predecir la probabilidad de que un nodo pertenezca a un cierto estado. En términos físicos, los mensajes representarán la interacción de más de dos cuerpos de los átomos o moléculas involucrados en el sistema, siendo éstos los nodos que se marcan en el grafo.

- Simetrías e invariancias: Cualquier predicción molecular realizada con los algoritmos de grafos debe respetar leyes básicas de la Física. En el caso de dinámica molecular, las simetrías principales a considerar son las siguientes: homogeneidad e isotropía, que conducen a invariancia rotacional y traslacional; las partículas son indistinguibles, por lo que son invariantes por permutación; finalmente, la simetría por paridad, que induce invariancia bajo un cambio de signos en las coordenadas espaciales.

- Campo conservativo: En dinámica molecular, el campo de fuerzas debe ser un campo vectorial conservativo, de modo que satisfaga la conservación de la energía (que es consecuencia directa de pedir homogeneidad temporal). La manera más fácil de lograr esto, es obteniendo el gradiente de una función que funja como la energía potencial del sistema. Entonces, el objetivo es predecir dicha función potencial, de modo que se puedan obtener las fuerzas por medio de la ecuación:

$$\vec{F}(\vec{x}) = -\frac{\partial}{\partial x_i} f_\theta(\vec{x}) \tag{4}$$

  donde θ son los parámetros de la función a ajustar. Más aún, dado que en dinámica molecular se tiene especial enfoque en obtener las fuerzas, cuando se calcula el costo del entrenamiento del algoritmo propuesto para este fin, se agregan las derivadas de la función potencial propuesta como parte del cálculo del correspondiente error:

$$L = |f_\theta(\vec{X}) - \hat{t}(\vec{X})| + \frac{\rho}{3N} \sum_{i=1}^{N} \sum_{\alpha=1}^{3} \left| -\frac{\partial f_\theta(\vec{X})}{\partial_{x_{i\alpha}}} - \hat{F}_{i\alpha} \right| \tag{5}$$

en donde $\hat{t}(\vec{X})$ es la variable objetivo del entrenamiento, que en el caso presente, es la energía potencial del sistema; así mismo, $\hat{F}_{i\alpha}$ son las fuerzas de las moléculas, en las tres componentes espaciales habituales. Para esta exposición, se tiene la hipótesis de que el entrenamiento se hace con cálculos cuánticos, y por eso sabemos a priori las fuerzas y energías del sistema molecular. El hiperparámetro ρ se escoge de tal manera que escale la aportación de las fuerzas al costo total.

- Paso de mensajes: Como se mencionó previamente, los átomos están representados mediante una incrustación $\vec{h_i} \in \mathbf{R}^H$, con H un hipérparámetro de la red. La misma estrategia de incrustaciones se utiliza para codificar los mensajes que los átomos vecinos pueden llegar a pasar a otro átomo, análogo al caso. La idea de la red DimeNet++ es que los átomos sean codificados mediante dos representaciones: la representación inicial consistirá de un vector asociado al número atómico del elemento del sistema. Los elementos que conforman esta primera representación se escogen de manera aleatoria de una distribución uniforme limitada por ciertos valores que se especificarán más adelante. La creación del primer mensaje consistirá de la concatenación entre pares de sus representaciones iniciales mencionadas. Los siguientes mensajes se formarán a partir de la interacción de dichos pares con sus entornos respectivos. La actualización de dichas representaciones se lleva a cabo mediante la siguiente fórmula:

$$\mathbf{m}_{ji}^{(l+1)} = f_{\text{actualización}}\left(\mathbf{m}_{ji}^{(l)}, \sum_{k \in \mathcal{N}_j \setminus \{i\}} f_{\text{interacción}}\left(\mathbf{m}_{kj}^{(l)}, \mathbf{e}^{(ji)}, \mathbf{a}^{(kj,ji)}\right)\right) \tag{6}$$

donde las funciones de actualización y de interacción son redes neuronales, que se especifican más adelante de esta sección, al igual que las funciones base de distancia y ángulos $\mathbf{e}^{(ji)}$ y $\mathbf{a}^{(kj,ji)}$.

- Funciones base: Parte fundamental de poder convertir un grafo en un vector de entrada es la representación que se utilice para las interacciones codificadas en las aristas. Como se mencionó previamente, la extensión que se desea integrar en el grafo es lo concerniente a los ángulos. Para ello, se presenta la siguiente argumentación para crear dichas bases:

  Primero, hay que estudiar el espacio de soluciones posible a la ecuación de Schrödinger independiente del tiempo:

$$\left[-\frac{\hbar^2}{2m}\nabla^2 + V(d)\right]\psi(d) = E\psi(d) \tag{7}$$

  donde $\psi(d)$ representa la función de onda del electrón, $d = x_k - x_j$, E y m son constante. Para el potencial, se escoge que tome el valor de 0 dentro de la vecindad de radio c, y que valga ∞ en cualquier otro lugar fuera de la vecindad. Haciendo pasos algebraicos, se llega a la ecuación de Helmholtz:

$$\left[\nabla^2 + k^2\right]\psi(d) = 0 \tag{8}$$

  con el número de onda $k = \frac{\sqrt{2mE}}{\hbar}$. Utilizando coordenadas esféricas $(\mathrm{d}, \theta, \phi)$ y usando el método de separación de variables, obtenemos la siguiente solución:

$$\psi(d,\theta,\phi) = \sum_{l=0}^{\infty}\sum_{m=-l}^{l}\left[a_{lm}j_l(kd) + b_{lm}y_l(kd)\right]Y_l^m(\theta,\phi) \tag{9}$$

  donde j_l y y_l son las funciones de Bessel de primera y segunda especie, y Y_l^m son los armónicos esféricos. Luego, se pide que la solución no diverja en el origen, por lo que b_{lm}=0. Otra característica de la solución es que dependa exclusivamente de la distancia y del ángulo θ, por lo que se toma m=0, y se tiene la solución

$$\psi(d,\theta) = a_l j_l(kd) Y_l^0(\theta) \tag{10}$$

  Para satisfacer las condiciones de frontera, es necesario pedir que

$$k = \frac{z_{ln}}{c} \tag{11}$$

donde z_{ln} es la n-ésima raíz del l-ésimo orden de la función de Bessel de primera especie. Después, normalizando la función resultante, se obtiene la solución final, dada por

$$\hat{a}(d,\theta) = \sqrt{\frac{2}{c^3 j_{l+1}^2(z_{ln})}} j_l\left(\frac{z_{ln}}{c}\right) Y_l^0(\theta) \tag{12}$$

con $l \in [0,...,N_S-1]$ y $n \in [0,...,N_r]$, donde N_S y N_R son el orden de las funciones de Bessel y el número de raíces a considerar, los cuales los define el usuario del código.

Ya que se tiene una representación que incluye a los ángulos, se tiene que trabajar en una representación de la distancia. Entonces, se tiene que quitar la dependencia del ángulo θ y ϕ, por lo que se toma l=m=0, y se tiene la solución

$$\psi(d) = a j_0\left(\frac{z_{0,n}}{c} d\right) \tag{13}$$

donde

$$z_{0,n} = n\pi \tag{14}$$

$$j_0(\alpha) = \frac{\sin(\alpha)}{\alpha} \tag{15}$$

Normalizando, se obtiene la representación final

$$\hat{e}(d) = \sqrt{\frac{2}{c}} \frac{\sin\left(\frac{n\pi}{c} d\right)}{d} \tag{16}$$

Nótese que 12 y 16 se escribieron con notación diferente a la expresada en la ecuación 6, esto debido a que las ecuaciones mencionadas no son doblemente diferenciables debido a la discontinuidad que se presenta en el radio de corte. Para lograr la continuidad, se debe proponer una función tal que tenga una raíz en d=c con multiplicidad 3 y que su primera y segunda derivada se vayan a cero en el radio de corte. Entonces, las representaciones finales quedarían como:

$$a(d,\theta) = u(d)\hat{a}(d,\theta) \tag{17}$$

$$e(d) = u(d)\hat{e}(d) \tag{18}$$

La función $u(d)$ que cumple estos requisitos es de la forma

$$u(d) = 1 - \frac{(p+1)(p+2)}{2} d^p + p(p+2)d^{p+1} - \frac{p(p+1)}{2} d^{p+2} \tag{19}$$

con p $\in \mathbb{N} \cup \{0\}$. Se toma el valor de p=6, como se indica en el trabajo original.

Teniendo la explicación de todos los elementos involucrados en esta red, se procede a explicar los bloques que conforman la arquitectura de DimeNet:

Bloque de incrustación: El comienzo de DimeNet está marcado por este bloque, en donde se generarán las primeras representaciones de los átomos, así como los primeros mensajes de cada par de elementos del arreglo molecular que forman parte de una misma vecindad.

La primera incrustación del átomo es una inicialización aleatoria de números que representan los números atómicos de los diferentes elementos que componen a la molécula. Esta representación se denota como $\vec{h}_i^{(0)}$. Entonces, el primer mensaje que se manda del átomo j al átomo i, se escribe mediante la regla

$$\vec{m}_{ji}^{(1)} = \sigma\left(\left[\vec{h}_j^{(0)} \| \vec{h}_i^{(0)} \| e(d_{ji})\right]\mathbf{W} + \mathbf{b}\right) \tag{20}$$

con σ la función de activación Swish [41], **W** y **b** parámetros de la red que se ajustarán en el entrenamiento. La función de activación Swish tiene la expresión:

$$f_{\text{Swish}} = x \cdot \frac{x}{1 + e^{-x}} \tag{21}$$

Bloque de interacción: La siguiente parte de la red contempla múltiples bloques de interacción, los cuales toman en cuenta tanto la representación de la distancia como de los ángulos, de modo que sea en esta parte del código donde se generen los cálculos que se indican en la ecuación 1.

Previo a explicar a detalle los pasos algebraicos que siguen los mensajes a través de este bloque de interacción, es conveniente explicar cualitativamente la acción que se lleva a cabo en esta parte de la red:

Primero, dado un radio de corte c, el cual tiene comúnmente dimensiones de Angstroms, se define una vecindad de radio c alrededor del i-ésimo átomo. Todo aquel átomo que se encuentre dentro de esta vecindad generará un grafo, con conexiones con todos los demás átomos que se encuentren en dicha vecindad.

Luego, para generar el nuevo mensaje del j-esimo átomo hacia el i-esimo átomo $\vec{m}_{ji}^{(l+1)}$, hay que recabar primero la información de todos los vecinos de la vecindad dada, hacia el j-esimo átomo. Teniendo generado esta recopilación de mensajes, se prosigue a hacer la propagación del mensaje hacia el i-ésimo átomo. Esto es lo que está indicado matemáticamente en la ecuación 6.

Teniendo mejor entendido el concepto detrás de este bloque, se procede a desarrollar el análisis cuantitativo que involucra a todos los elementos explicados en los párrafos anteriores:

Primero, se describe la manera de recabar toda la información concernientes a los vecinos de los átomos j e i. Dado un mensaje que se propagó en la l-ésima capa de un átomo k al átomo j, se transforma este mensaje por medio de una capa lineal, con la función de activación Swish. Se genera entonces una representación ω_1, dada por

$$\omega_{1k} = \sigma\left(\mathbf{W}\vec{m}_{kj}^{(l)} + b\right) \tag{22}$$

donde se usará indistintamente los parámetros **W** y **b** para indicar los pesos y los sesgos de las diferentes capas, aunque hay que recalcar que no representarán a un mismo conjunto de parámetros.

Por otro lado, utilizando la representación de la distancia entre dichos átomos, $e(d_{ji})$, se llevará a cabo una nueva representación, haciendo el producto de Hadamard entre los siguientes arreglos multidimensionales:

$$\omega_{2k} = \mathbf{W}e(d_{ji}) \odot \omega_{1k} \tag{23}$$

Finalmente, la tercera representación se generará utilizando la información angular, y se generará mediante la expresión

$$\omega_{3k} = \left[\mathbf{W}a(d_{kj}, \theta_{kji})\right]^T \mathbf{W}\omega_{2k} \tag{24}$$

Teniendo esta representación final, se repite el proceso para todos los k-ésimos átomos de la vecindad, y se genera finalmente el mensaje de propagación de la vecindad hacia el átomo j

$$\vec{m}_{\mathcal{N}_j \setminus \{i\}, j} = \sum_k \omega_{3k} \tag{25}$$

A todo el procedimiento elaborado para llegar a la expresión $\vec{m}_{\mathcal{N}_j \setminus \{i\}, j}$ se le conoce en el formalismo de DimeNet++ como el Pase Direccional de Mensajes.

Para concluir, falta incluir en este bloque el mensaje generado en la capa anterior, $\vec{m}_{ji}^{(l)}$. Este mensaje se transforma por medio de una capa lineal, con la activación Swish, y se suma al vector de mensaje direccional generado por los vecinos:

$$\eta_{1j} = \vec{m}_{\mathcal{N}_j \setminus \{i\},j} + \sigma\left(\mathbf{W}\vec{m}_{ji}^{(l)} + \mathbf{b}\right) \tag{26}$$

Antes de proseguir con la arquitectura, se define a continuación un bloque residual, que se repite varias veces en lo que queda del bloque. Este bloque residual sigue la idea del trabajo original de las ResNets [42], donde se concatenan dos capas lineales previo a establecer la conexión con la entrada original. La expresión matemática de este bloque está dado por

$$\text{Res}(z) = z + \sigma\left[\mathbf{W}\sigma\left(\mathbf{W}z + \mathbf{b}\right) + \mathbf{b}\right] \tag{27}$$

Con esta definición, se prosigue con las representaciones restantes que quedan dentro del bloque. A continuación, se genera η_{2j}, utilizando un bloque residual:

$$\eta_{2j} = \sigma\left(\mathbf{W}\text{Res}(\eta_{1j}) + \mathbf{b}\right) \tag{28}$$

El siguiente paso es establecer una conexión con el mensaje original, $\vec{m}_{ji}^{(l)}$, similar a que si eta_{2j} hubiera sido un bloque residual:

$$\eta_{3j} = \eta_{2j} + \vec{m}_{ji}^{(l)} \tag{29}$$

Para concluir con este bloque, se generará el mensaje final actualizado, $\vec{m}_{ji}^{(l+1)}$, con dos bloques residuales finales:

$$\vec{m}_{ji}^{(l+1)} = \eta_{3j} \tag{30}$$

Este mensaje final se utiliza como entrada para una nueva capa de interacción y también para una capa de salida, que se va a explicar a continuación.

Bloque de salida: Una vez que se generaron los mensajes del j-esimo átomo hacia el i-ésimo átomo de una vecindad establecida de radio c centrada en el i-esimo átomo, falta recabar la información de todos los mensajes de todos los j-ésimos átomos, siguiendo los pasos de interacción que se describieron en la sección anterior. Teniendo estos mensajes, se puede generar la predicción final de la observable de interés. A continuación, se describe la formulación matemática de este bloque:

Una vez generado el mensaje de la capa $l+1$, se aplica un producto de Hadamard con una matriz de pesos, que a su vez realiza el producto matricial con la representación de la distancia $e(d_{ji})$, que es la misma que se estaba utilizando en el bloque de interacción:

$$\tau_{1ij} = \mathbf{W}e(d_{ij}) \odot \vec{m}_{ji}^{(l+1)} \tag{31}$$

Teniendo este producto, hay que sumar las contribuciones de todos los vecinos de la vecindad:

$$\tau_{2i} = \sum_{j \in \mathcal{N}_i} \tau_{1ij} \tag{32}$$

El siguiente tramo del bloque es aplicar tres capas lineales, las cuales son de la forma

$$\text{Linear}(z) = \sigma\left(\mathbf{W}z + \mathbf{b}\right) \tag{33}$$

Entonces, la tercera representación se genera con la siguiente expresión:

$$\tau_{3i} = \text{Linear}\left(\text{Linear}\left(\text{Linear}\left(\tau_{2i}\right)\right)\right) \tag{34}$$

La predicción final de la red DimeNet++ se genera con la ecuación

$$t_i^{(l+1)} = \mathbf{W}\tau_{3i} \tag{35}$$

3.3. Teoría Efectiva del Aprendizaje Profundo

Siguiendo la discusión expuesta en el apartado anterior, dadas las diferentes arquitecturas disponibles en la actualidad para resolver problemas usando redes neuronales, existe actualmente una corriente de interés alrededor del comportamiento de las redes, tanto a nivel elemental como global, como se expone en [14], en donde a través de conceptos físicos, así como su debida justificación matemática, se busca explicar el comportamiento de las redes neuronales bajo cierto régimen. A continuación, se expondrán ideas globales que tratan en esta Teoría Efectiva de las Redes Neuronales, y que serán utilizados en este trabajo.

La idea central de la teoría efectiva desarrollada por Daniel A. Roberts and Sho Yaida [14] es en estudiar la distribución de probabilidad que caracteriza a un conjunto de redes neuronales, dado un conjunto de datos como entrada a dichos algoritmos. La primera propuesta de solución es una distribución Gaussiana, ya que es una distribución muy bien estudiada por el mundo científico, y ofrecería resultados inmediatos sobre el comportamiento de las redes neuronales a las distintas escalas de interés. Sin embargo, no es posible dicha descripción de la redes neuronales por medio de gaussianas. Luego, siguiendo el método de teoría de perturbaciones que se usa frecuentemente en Mecánica Cuántica [43], los autores proponen un desarrollo en donde las interacciones no Gaussianas se describen a través de las funciones de correlación de M elementos del conjunto de datos; dado que dichas funciones, también llamadas momentos, describen por completo a una distribución [44], entonces serían suficientes para describir el problema y presentar una solución.

A través del método perturbativo se encuentra que dicha distribución contiene una descripción basada en distribuciones Gaussianas, y que los elementos no Gaussianos están escalados por un factor, ϵ, llamado cociente de aspecto, con el cual se puede hacer el análisis de distintos regímenes dependiendo del valor de dicho factor.

El cociente de aspecto incluye los siguientes elementos:

- El ancho de la red, n, el cual se define como el número máximo de neuronas contenidas en una capa de la red neuronal
- La profundidad, L, que es el número total de capas de una red.

Así, se define este cociente como

$$\epsilon \equiv \frac{L}{n} \tag{36}$$

Este cociente toma relevancia cuando se estudian diferentes casos de los límites que puede adquirir cada elemento involucrado:

1. Haciendo $n \longrightarrow \infty$, dejando fijo L, entonces $\epsilon \approx 0$ y se pierden los términos no Gaussianos, los cuales son clave para que las redes neuronales puedan realizar mapeos complejos, según el problema planteado. Otra interpretación de este límite es que dado que coexisten muchas neuronas en una determinada capa, entonces se pierde la interacción entre ellas, y no se puede obtener información de los datos de entrada que permitan una salida Y acorde a las necesidades de cada planteamiento. Luego, si las redes neuronales estuvieran determinadas únicamente bajo una distribución gaussiana, entonces las redes no serían algoritmos de tanta utilidad, pues serían mapeos sencillos
2. Tomando ahora $L \longrightarrow \infty$, fijando n, entonces $\epsilon \gg 1$ y se tendrían muchas fluctuaciones cada vez que se pase información de una capa a la otra, por lo que dichos modelos no tienen utilidad alguna en el esquema de aprendizaje automático.

Dado que esta teoría surge de la necesidad de describir redes neuronales reales (finitas) [45], la intención del cociente de aspecto es describir un escenario real, en donde se eviten escenarios de pérdida de información o de inestabilidad entre capas, a través de valores reales para el ancho y profundidad de una red neuronal. En este trabajo se utilizará este cociente para justificar la arquitectura final para describir el potencial de interacción del sistema estudiado

3.4. Física

3.4.1. Dinámica Molecular

La dinámica molecular estudia cómo se mueven las moléculas, cómo se deforman y cómo interactúan en el tiempo [46]. Diferentes trabajos alrededor de esta técnica de simulación se han presentado a lo largo del tiempo [47]. Un ingrediente importante en las simulaciones de dinámica molecular son las fuerzas moleculares, las cuales se buscan obtener preferentemente a través de cálculos con mecánica cuántica de la estructura electrónica del objeto a estudiar. Desafortunadamente, estos cálculos son computacionalmente costosos y solo pueden describir sistemas pequeños o sistemas con escalas de tiempo pequeñas. Para subsanar este impedimento, se trabajan, para escalas mayores, campos clásicos que suelen tener buenos resultados a costa de no obtener la precisión de los cálculos cuánticos [38].

Como se ha descrito en secciones anteriores, para resolver este problema se han empleado técnicas de aprendizaje automático para obtener los potenciales y las fuerzas del sistema. De los primeros trabajos que describen el empleo de redes neuronales para calcular las superficies de energía potencial se encuentra el realizado por Thomas Blank y sus colaboradores en 1995 [48], en el cual muestran la aplicación de redes neuronales a diferentes configuraciones, así como la comparación en rapidez y eficacia con respecto a otros métodos, como el método Monte Carlo. Actualmente, los métodos de aprendizaje supervisado son los más utilizados para determinar la superficie de potencial y así determinar con precisión las propiedades y aplicaciones de diversos elementos en ciertos compuestos, como en el caso del titanio [49]; cabe resaltar que en el trabajo citado del titanio, se hace explícita la dependencia de los cálculos con funcionales para obtener los valores a conseguir en el aprendizaje del modelo.

Los potenciales obtenidos por las redes neuronales modelan las propiedades de un sistema basadas en las contribuciones individuales de cada átomo dentro de un entorno local definido [38]. Estos entornos locales se representan mediante las funciones de simetría de Behler dadas por [50]

$$G_i^{rad} = \sum_{j \neq i}^{N} e^{-\eta\left(r_{ij} - r_0\right)^2} f_{\text{corte}}\left(r_i j\right) \tag{37}$$

las cuales, son distribuciones radial y angular, que toman en cuenta las invariancias rotacionales y traslacionales del sistema. En la ecuación 37, η y r_0 son parámetros de la función gaussiana y f_{corte} se encarga de tomar únicamente las contribuciones del entorno local [51].

Teniendo dichas representaciones para cada átomo, el siguiente paso es sumar todas las contribuciones de cada entorno local para obtener el resultado global del sistema. En el caso de los potenciales, para la molécula m, la energía total está dada por

$$E^m = \sum_{i}^{N_m} E_i^m \tag{38}$$

Recientes desarrollos empleando técnicas de inteligencia artificial utilizan redes neuronales de grafos como modelo para obtener los potenciales de las interacciones. Estas redes utilizan el mecanismo desarrollado por el equipo de Gilmer y colaboradores sobre el pase de mensajes en una red neuronal para describir propiedades químicas [52], así como la arquitectura diseñada por Batagglia et al, en donde la entrada de dichas redes son grafos, y la salida son igualmente grafos, pero con actualizaciones en la información relacionada con los nodos, vértices y contexto global [12].

La principal ventaja de utilizar redes de grafos es que no se tienen que desarrollar funciones que describan al entorno local, pues a través de las capas que combinan convoluciones con grafos se extraen las propiedades locales, e incluso se puede, en principio, romper esta localidad a medida que se toman en cuenta más vecinos en el entorno [53].

3.4.2. Sistemas bajo condiciones termodinámicas NPT

Cuando se estudia un sistema a presión y temperatura constante, se debe tomar en cuenta al contenedor, de modo que en el desarrollo de las ecuaciones de movimiento también se tome en cuenta la estructura atómica y el número de partículas que conforman a dicho contenedor. Si bien esta descripción es la más fidedigna de la

realidad, implica un costo computacional elevado [54]. Ante esta disyuntiva, se propone extender el espacio fase de soluciones de las ecuaciones de movimiento del sistema, añadiendo variables de control que representen las fluctuaciones en la temperatura y la presión. La principal desventaja alrededor de estas técnicas es que no son generales y no existe actualmente un formalismo a seguir para la derivación de estas nuevas ecuaciones [55]. En el Apéndice de este trabajo se desarrolla el formalismo presentado por el doctor Santamaría y el doctor Soullard, del Instituto de Física [54] para desarrollar lo que se conoce como dinámica No Hamiltoniana. El tema importante a resaltar es que al añadir los dos grados de libertad adicional al sistema original de estudio, se necesita ahora formular una manera de solución a dichas ecuaciones. A continuación se presentan la metodología de Nosé y Hoover [56] para conocer dichas soluciones.

3.4.3. Cadenas de termostatos de Nosé-Hoover

Al añadir los dos grados de libertad a las ecuaciones de movimiento, se obtiene lo que se conoce como dinámica de Nosé-Hoover (ver Apéndice para mayores detalles):

$$\dot{q}_i = \frac{p_i}{m_i} \tag{39}$$

$$\dot{p}_i = -\frac{\partial U(\vec{q})}{\partial q_i} - p_i \frac{p_s}{M_s} \tag{40}$$

$$\dot{p}_s = \sum_{i=1}^{N} \frac{p_i^2}{m} - N k_B T$$

$$\dot{s} = \frac{p_s}{M_s}$$

en donde s, M_s y p_s son la posición, la masa y el momento asociados a la partícula virtual añadida al sistema que controla las fluctuaciones de la temperatura.

Este sistema de ecuaciones induce a una cantidad conservada, que es

$$H'(\vec{p}, \vec{q}, s, p_s) = U(\vec{q}) + \sum_{i=1}^{N} \frac{p_i^2}{2m_i} + \frac{p_s^2}{2M_s} + N k_B T \tag{41}$$

Esta cantidad no es el Hamiltoniano del sistema; se conoce como Hamiltoniano extendido, debido a que cumple que

$$\frac{dH'}{dt} = \sum_{i=1}^{N} \left[\frac{\partial H'}{\partial p_i} \dot{p}_i + \frac{\partial H'}{\partial q_i} \dot{q}_i \right] + \frac{\partial H'}{\partial p_s} \dot{p}_s + \frac{\partial H'}{\partial s} \dot{s} = 0 \tag{42}$$

Para demostrar 42, se debe suponer que el sistema es ergódico. Es decir, que todos los microestados son igualmente probables de ocurrir en un tiempo suficientemente grande. Utilizando el trabajo del doctor José Alejandre, Roberto López Rendón y colaboradores [57], proponen una condición para analizar la ergodicidad del sistema:

Dado un sistema dinámico que cumpla las siguientes ecuaciones de movimiento:

$$\dot{\vec{x}} = \xi(\vec{x}) \tag{43}$$

donde $\vec{x}$ es un vector en el espacio fase, y ξ un campo vectorial de dicho espacio fase. Una condición suficiente para saber si el sistema no es ergódico, es que

$$\nabla \cdot \xi(\vec{x}) \neq 0 \tag{44}$$

El problema con 42 y 44 es que no son condiciones necesarias para establecer si el sistema ergódico. Más aún, en 42 se ve claramente que el termostato controla las fluctuaciones del momento p_i, pero no hay nada que controle las fluctuaciones del momento del termostato, p_s. Luego, hay que agregar otro termostato que controle p_s, y así sucesivamente se crean las denominadas cadenas de Nosé-Hoover [56].

Considérense M termostatos añadidos. Las ecuaciones de movimiento quedan marcadas como

$$
\begin{aligned}
\dot{q}_i &= \frac{p_i}{m_i} \\
\dot{p}_i &= -\frac{\partial U(\vec{q})}{\partial q_i} - p_i \frac{p_{\eta_1}}{Q_{\eta_1}} \\
\dot{q}_{\eta_1} &= \frac{p_{\eta_1}}{m_{\eta_1}} \\
\dot{p}_{\eta_1} &= \left[\sum_{i=1}^{N} \frac{p_1^2}{m_i} - N k_b T \right] - p_{\eta_1} \frac{p_{\eta_2}}{Q_{\eta_2}} \\
\dot{p}_{\eta_j} &= \left[\frac{p_{\eta_{j-1}}^2}{Q_{\eta_{j-1}}} - k_B T \right] - p_{\eta_j} \frac{p_{\eta_{j+1}}}{Q_{\eta_{j+1}}} \\
\dot{p}_{\eta_M} &= \left[\frac{p_{\eta_{M-1}}^2}{Q_{\eta_{M-1}}} - k_B T \right]
\end{aligned}
\tag{45}
$$

donde las Q_{η_j} son las masas asociadas a los termostatos. Cabe resaltar que las ecuaciones que rigen la dinámica de los termostatos tienen un significado físico: los términos que acompañan a los momentos p_i actúan como una especie de fuerza de fricción dinámica. Además, estas ecuaciones originan una función de distribución en el espacio fase de la forma

$$
f(\vec{v}, \vec{q}, \vec{p}_\eta, \vec{\eta}) \propto \exp\left\{ -\frac{1}{k_B T} \left[U(\vec{q}) + \sum_{i=1}^{N} \frac{p_i^2}{2m_i} + \sum_{j=1}^{M} \frac{p_{\eta_j}^2}{2Q_{\eta_j}} \right] \right\} \tag{46}
$$

Así, la cantidad conservada es

$$
H'(\vec{v}, \vec{q}, \vec{p}_\eta, \vec{\eta}) = U(\vec{q}) + \sum_{i=1}^{N} \frac{p_i^2}{2m_i} + \sum_{j=1}^{M} \frac{p_{\eta_j}^2}{2Q_{\eta_j}} + N k_B T \eta_1 + \sum_{j=2}^{M} k_B T \eta_j \tag{47}
$$

De la ecuación 47, se observa que añadir más termostatos se ve reflejado como una sola cadena. Únicamente el primer termostato es el que interactúa con las N partículas del sistema.

Como proponen adecuadamente Tuckerman y colaboradores [57], los valores de las masas de los termostatos más idóneos son los siguientes:

$$
Q_{\eta_1} = N k_B T \tau^2 \tag{48}
$$

$$
Q_{\eta_j} = k_B T \tau^2 \qquad\qquad \forall j \neq 1 \tag{49}
$$

donde

$$
\tau = \frac{1}{\omega} \tag{50}
$$

con ω la frecuencia fundamental del sistema (el trabajo citado en [57] llegan a términos de oscilador armónico en su argumentación de dichas masas).

3.4.4. Algoritmo de velocidad de Verlet

En 1967 Loup Verlet propone una manera de resolver las ecuaciones diferenciales de segundo orden asociadas a la dinámica Newtoniana de los sistemas moleculares [58], utilizando intervalos discretos del tiempo, para poder resolver dichas ecuaciones por cálculos computacionales. Las ecuaciones de Verlet son las siguientes:

$$
x_{t+\Delta t} = x_t + v_t \Delta t + F_t \frac{\Delta t^2}{2} \tag{51}
$$

$$
v_{t+\Delta t} = v_t + \frac{F_t + F_{t+\Delta t}}{m} \frac{\Delta t}{2} \tag{52}
$$

3.4.5. Potencial de Lennard-Jones

El potencial de Lennard-Jones [59]es uno de los potenciales más utilizados para describir la interacción entre dos cuerpos en Dinámica Molecular. La forma más habitual de trabajar con el mencionado potencial es con la forma 12-6, la cual toma la siguiente forma:

$$V(r) = 4\epsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right] \tag{53}$$

donde ϵ representa la profundidad del pozo de potencial, mientras que σ denota la distancia para la cual la fuerza neta aplicada sobre las partículas se vuelve nula. Dicha forma fue propuesta por J.E. Lennard-Jones en 1931, luego de que se conociera que la interacción de dispersión entre los átomos obedece una regla de r^{-6} [60], que se atribuye a la fuerza de Van der Waals.

Este potencial posee la característica de separarse en dos partes: una parte repulsiva, que corresponde al término con el exponente 12, que se hace presente en distancias pequeñas; el término restante, denominado término atractivo y que se identifica con el exponente 6, cobra relevancia para representar la interacción a grandes distancias. Esta distinción sirve para ver la descripción del potencial de Lennard-Jones como un desarrollo perturbativo, en donde el término central es el término repulsivo, mientras que la perturbación viene dada por el término de atracción [9].

Con esta idea presente, en la presente tesis se propone como potencial de interacción un modelo con desarrollo perturbativo, en donde el papel de término central será descrito por la repulsión del potencial de Lennard Jones. Por otro lado, el término perturbativo será modelado por medio de la red neuronal de grafos DimeNet++

3.4.6. Física del agua: Distribución radial y angular

Dado que en este trabajo se verá a la molécula de agua de manera granular, se puede obtener la función de distribución de pares radial, denotada en algunos textos como $g(r)$ [9]. En el ensamble canónico, dicha distribución tiene la forma siguiente:

$$g(r_1, r_2) = \frac{N(N-1)}{\rho^2 Z_{NVT}} \int dr_3 dr_4 \cdots dr_N e^{-\frac{1}{k_B T} V(r_1 r_2 \cdot\cdot)} \tag{54}$$

donde la elección de los átomos 1 y 2 es totalmente arbitraria, y donde la función de partición Z_{NVT} es

$$Z_{NVT} = \int dr_1 dr_2 dr_3 \cdot\cdot e^{-\frac{1}{k_B T} V(r_1 r_2 \cdot\cdot)} \tag{55}$$

(nótese que en 54 el numerador contiene la integral que define a Z_{NVT}, pero no se integra sobre r_1 y r_2).

Físicamente, esta función tiene la siguiente interpretación: dada la densidad del sistema, ρ, se puede definir una densidad local de la siguiente manera: se fija un determinado elemento de interés, y a la densidad total se le multiplica un cierto factor, $g(r)$, con $r = |r_2 - r_1|$, siendo r_1 el punto fijo predeterminado anteriormente y r_2 cualquier otro punto.

Esto da luz a los siguientes puntos detrás de dicha distribución:

1. Si se realiza la integral de volumen de la distribución radial sobre todo el volumen, se obtendrá de resultado N-1, con N el número de elementos en el sistema.

2. Tomando fijo un elemento y midiendo la distancia hacia cualquier punto, entonces la distribución radial debe tender a cero cuando la distancia también tienda a cero, debido a que los elementos en esta región se comportan como esferas rígidas [61].

3. Considerando ahora el límite cuando $r \to \infty$, entonces $g(r) = 1$, pues la influencia del elemento comienza a desvanecerse a medida que se incrementa la distancia, y, por lo tanto, solo se encuentra ese mismo elemento fijo.

En cuanto al interés de predecir la función de distribución entre tripletes de oxígenos, se puede encontrar en el trabajo realizado por DiStasio Jr y colaborares [62], en donde a partir de un experimento por difracción de rayos X los investigadores utilizan metodologías ab initio para predecir dicha observable y sus cálculos no son suficientes

para reproducir los resultados experimentales. Tanto en este trabajo de experimentación como en el citado previamente, los elementos a considerar para la distribución angular son de los primeros vecinos que se tienen entre pares de elementos. Es decir, dado un cierto radio de corte, si al trazar la vecindad alrededor de un elemento fijo se encuentran vecinos, entonces con ese par definido se tomará un tercer elemento perteneciente a alguna de las vecindades de los dos anteriormente definidos para conformar el triplete.

4. Metodología: Planteamiento de la forma de trabajo mediante el algoritmo DiffTRe

En esta sección se expondrá detalladamente el procedimiento a seguir para llevar a cabo el proceso de entrenamiento de la red de grafos DimeNet++, siguiendo el algoritmo DiffTRe, desarrollado por Stephan Taler, de la Universidad Tecnológica de Múnich [15].

Dicho algoritmo responde a la siguiente cuestión: para poder realizar un proyecto con Aprendizaje Automático, es necesario definir puntualmente las variables de entrada, así como las variables dependientes que busca el modelo predecir. Como se discutió en la sección teórica, existen metodologías supervisadas y no supervisadas para plantear el problema bajo estos términos. No obstante, en este proyecto de investigación no se tiene directamente la información de los datos de entrada, y no se van a utilizar cálculos cuánticos como forma de supervisar el entrenamiento del modelo. Esto motiva a buscar otra metodología para poder predecir el potencial de un sistema molecular bajo estas peculiaridades.

DiffTRe es un algoritmo basado en Aprendizaje por Refuerzo, en donde el agente propondrá una serie de pasos, que se nombran en física como trayectoria del sistema, y con dicha propuesta el agente describirá la energía potencial del sistema, con la cual podrá determinar las fuerzas y a su vez explicar, con un resultado a nivel cuántico, propiedades macroscópicas del sistema a estudiar. Comparando las predicciones hechas por el agente con las del resultado experimental, se toma la decisión si la trayectoria actual es suficiente para seguir mejorando las predicciones o si es necesario generar una nueva, utilizando el último potencial obtenido. A todo este flujo iterativo se le denomina en este trabajo algoritmo DiffTRe .

Ahora que se tiene la idea general detrás de DiffTRe, se desglosarán a continuación los puntos clave que se deben identificar detalladamente para entender por completo el algoritmo:

- Adquisición de datos
- Definición de observables de referencia
- Inicialización de los parámetros del potencial
- Inicialización de DiffTRe
- Entrenamiento e inferencia

En esta sección, se expondrá a detalle los pasos a seguir para poder realizar el entrenamiento de la red DimeNet++ con el algoritmo DiffTRe. La descripción de los puntos detallados anteriormente se conformará por la física y el código realizado y estudiado alrededor del mencionado algoritmo. Se añadirán elementos gráficos que permitan al lector comprender y visualizar los conceptos inmersos en cada apartado.

4.1. Adquisición de datos

El sistema a estudiar consta de un cúmulo de moléculas de agua, vistas de manera granular como un solo átomo y cuyos centros se localizan en los oxígenos del agua. Los datos experimentales se obtienen del artículo [63], con formato GROMACS [64], el cual presenta la información de las tres coordenadas espaciales, en nanómetros, con precisión de tres decimales; así mismo, se presenta la información de las componentes de la velocidad, que tienen unidades de nanómetros sobre picosegundos, con 4 decimales. En el último renglón del archivo se presentan las dimensiones de la caja en las que se lleva a cabo el experimento, con unidades de nanómetros. Es importante resaltar que no se tiene registro alguno de la energía.

En el código, la adquisición del archivo para leerlo en el lenguaje Python se basa en la librería MDAnalysis [65]. Esta librería posee la cualidad de poder detectar automáticamente diversos formatos empleados en simulaciones de dinámica molecular para reportar resultados; entre ellos, se encuentra el formato GROMACS. Después, se crea un objeto llamado Universo, con el cual se tiene acceso directo a la información principal del experimento, como las coordenadas, la velocidad y las dimensiones de la caja. Dado que la descripción espacial que se realiza en este proyecto usa los nanómetros como unidades, como paso final se verifica que se respeten dichas dimensiones.

El siguiente paso es convertir la información de las coordenadas, velocidades y dimensiones de la caja en arreglos de JAX Numpy [10], la cual es una librería de Python de nivel medio, en términos de lenguajes de programación, con el cual se pueden realizar optimizaciones de cálculos en tarjetas gráficas si se disponen de ellas, con modificaciones mínimas al código. A lo largo de la sección se harán hincapié en ciertos detalles que conciernen a la metodología de trabajo de JAX. Adicionalmente, el presente trabajo se llevará a cabo con una tarjeta gráfica NVIDIA V100, por lo que todos los resultados y tiempos de ejecución se verán favorecidos, gracias al nivel de cálculo que dispone dicha tarjeta gráfica.

Se muestra a continuación el código para llevar a cabo los pasos mencionados previamente [15]:

```
import MDAnalysis
import jax.numpy as jnp


def load_configuration(file):

    universe = MDAnalysis.Universe(file, file)  # hacer conversion a nanometros
    coordinates = universe.atoms.positions * 0.1
    if hasattr(universe.atoms, 'velocities'):
        velocities = universe.atoms.velocities * 0.1
    else:
        velocities = jnp.zeros_like(coordinates)
    box = universe.dimensions[:3] * 0.1
    return jnp.array(coordinates), jnp.array(velocities), jnp.array(box)
```

Código 1: Importar datos usando MDAnalysis y JAX

En este apartado se describió la forma de adquirir los datos asociados a cada molécula de agua del sistema. En este punto, se tienen 901 arreglos con seis componentes: las tres primeras con las coordenadas espaciales y las tres restantes con las velocidades en las direcciones cartesianas en tres dimensiones. Dicho conjunto conformará el marco inicial, con el cual más adelante se generará una colección de marcos que se denominará trayectoria. En la siguiente sección se describirá la adquisición de los resultados experimentales asociados al sistema molecular del agua, cuya función será retroalimentar al agente en la manera de cómo definir los parámetros de DimeNet++.

4.2. Definición de observables de referencia

A continuación, se definen las observables de referencia, las cuales funcionarán para comparar las predicciones hechas por el algoritmo, y a su vez realizar el método de optimización de parámetros de la red neuronal. En el caso de estudio de esta investigación, se considerarán tres observables de referencia para el entrenamiento del algoritmo: la distribución radial y angular de los oxígenos de la molécula del agua, así como la presión del sistema.

Para poder realizar la optimización de parámetros de la red neuronal a trabajar, se necesita tener los valores de referencia para comparar las predicciones que se realizan en cada iteración del aprendizaje. Dichas referencias corresponden a observables macroscópicas asociadas al sistema, obtenidas directamente del experimento realizado en [63]. En este apartado se describen los pasos para obtener la información de estos resultados y configurarla en el formato necesario para alimentar a la red neuronal.

Las observables de referencia a utilizar en esta tesis serán la distribución radial, la distribución angular y la presión del sistema. Respecto a la presión, dicho dato se obtiene directamente del reporte de resultados del experimento [63], con lo cual no se requiere ningún procedimiento extra para indicarlo en el algoritmo. Sin embargo, no sucede lo mismo con las demás variables de referencia.

Para el caso de la distribución radial, hay que recordar que dado un elemento del sistema, se toman vecindades con radios definidos en un intervalo continuo de valores, y se contabilizan cuántos elementos se encuentran en dichas vecindades. Luego, la distribución radial es una función continua, en donde a cada valor de distancia se asocia una densidad asociada a la frecuencia de elementos encontrados en dicha vecindad. No obstante, los experimentos en la vida real no toman mediciones continuas. La discretización de la función es parte del planteamiento experimental a considerar. Análogamente, esta situación se presenta igualmente para la distribución angular, en

donde se mide la frecuencia de ángulos que pueden conformar tripletas de elementos del sistema.

Los resultados experimentales reportados en [63] para la distribución radial se encuentra en intervalos de 0.004 nanómetros de distancia, partiendo del punto 0.002. Para la angular, los intervalos son de $1.745 \text{x} 10^{-1}$ radianes de tamaño.

Una vez que se tienen los resultados discretos, es posible recuperar la función continua por medio de alguna técnica de interpolación. Con esto en mente, lo que se debe hacer para el experimento computacional de esta tesis es proponer puntos discretos en intervalos fijos, y usando la función continua recuperada a partir de los resultados experimentales, llevar a cabo evaluaciones de dicha función en estos puntos discretos. La colección de puntos evaluados constituirán los valores de referencia que representen a las dos diferentes funciones de distribución.

Los pasos a seguir para resolver ambos casos se pueden describir en los siguientes puntos

1. Definición de la función de distribución continua por medio de los resultados experimentales.
2. Evaluación de dicha función en un conjunto discreto de valores propuestos.

Teniendo presentes estos puntos, se presenta a continuación los pasos a seguir, tanto teóricamente como en código, para generar la distribución radial y angular experimental de referencia del experimento.

4.2.1. Distribución radial

Una vez que se adquieren los datos experimentales de la distribución radial, se propone una interpolación cúbica para poder construir la función continua de distribución. La interpolación se realiza con la librería SciPy [66]. Se anexa la gráfica con el resultado de la interpolación sobre los puntos definidos por el experimento

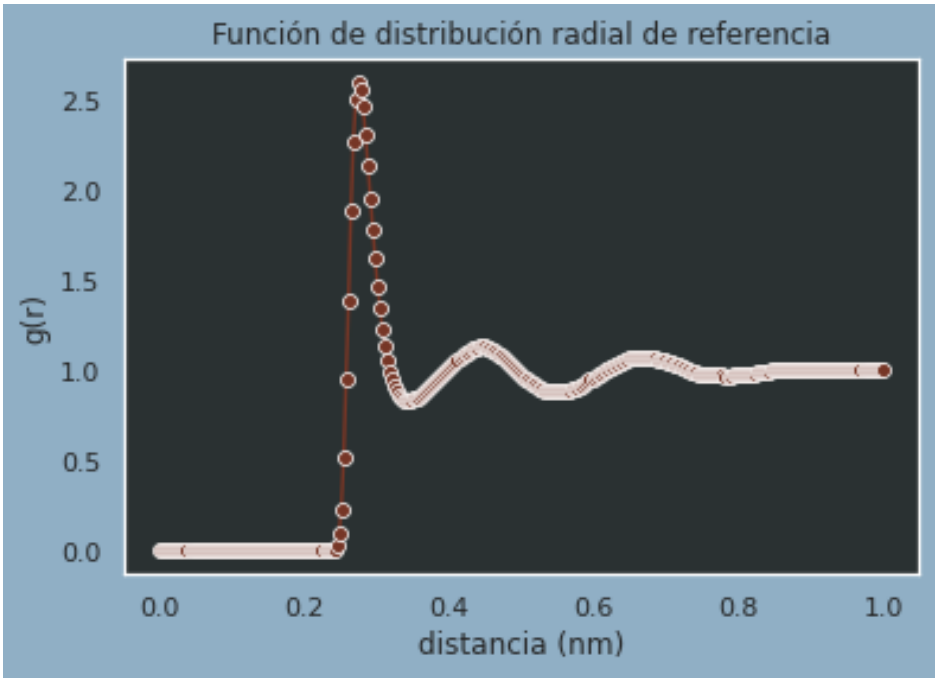

Figura 1: Distribución radial del sistema molecular de agua, usando una interpolación cúbica sobre los datos reportados del experimento. Los puntos representan los datos experimentales, mientras que la línea continua representa la interpolación.

Después, se define un intervalo de 0 a 1 nanómetro, y se define una equipartición de 300 sub intervalos y cuyos centros servirán como los valores discretos del experimento computacional. El arreglo de datos de referencia serán las evaluaciones de la distribución continua obtenida en la Figura 1 sobre estos puntos.

Se anexan a continuación los códigos para definir los intervalos discretos, así como la construcción y evaluación de la función de distribución continua sobre dichos puntos

```

def rdf_discretization(RDF_cut, nbins=300, RDF_start=0.):
    """
    Se realiza el calculo de los parametros de inicializacion de la funcion radial discreta

    Args:
        RDF_cut: distancia maxima de separacion entre moleculas
        nbins: total de intervalos
        RDF_start: distancia minima entre moleculas

    Returns:
        tres arreglos con los centros de los intervalos, el inicio y fin de cada intervalo
    generado y la longitud de dichos intervalos

    """
    dx_bin = (RDF_cut - RDF_start) / float(nbins)
    rdf_bin_centers = jnp.linspace(RDF_start + dx_bin / 2., RDF_cut - dx_bin / 2., nbins)
    rdf_bin_boundaries = jnp.linspace(RDF_start, RDF_cut, nbins + 1)
    sigma_RDF = jnp.array(dx_bin)
    return rdf_bin_centers, rdf_bin_boundaries, sigma_RDF
```

Código 2: Se inicializan las variables necesarias para crear la propuesta discreta de la función radial

```

from jax_md import dataclasses
from scipy import interpolate as sci_interpolate


class ADFParams:

    reference_adf: Array
    adf_bin_centers: Array
    sigma_ADF: Array
    r_outer: Array
    r_inner: Array


rdf_bin_centers, rdf_bin_boundaries, sigma_RDF = rdf_discretization(RDF_cut=1.0)  # cut RDF
    at 1nm
reference_rdf = np.loadtxt('data/experimental/O_O_RDF.csv')
rdf_spline = sci_interpolate.interp1d(reference_rdf[:, 0], reference_rdf[:, 1],
    kind='cubic')
reference_rdf = rdf_spline(rdf_bin_centers)
rdf_struct = RDFParams(reference_rdf, rdf_bin_centers, rdf_bin_boundaries, sigma_RDF)
```

Código 3: Paso 2 de ejecución de la interpolación de la función de distribución radial de pares de oxígenos

4.2.2. Distribución angular entre tripletes de oxígenos

Haciendo el procedimiento análogo a la distribución radial, para la distribución angular de tres moléculas de agua se dividirán π radianes en 200 partes iguales, similar a lo que se realizó con el intervalo de la distribución radial. Una vez adquiridos los resultados experimentales, se propone la interpolación para obtener la función continua, y finalmente se evalúa sobre los doscientos centros de los intervalos anteriores. Se incluye la gráfica de la distribución angular continua:

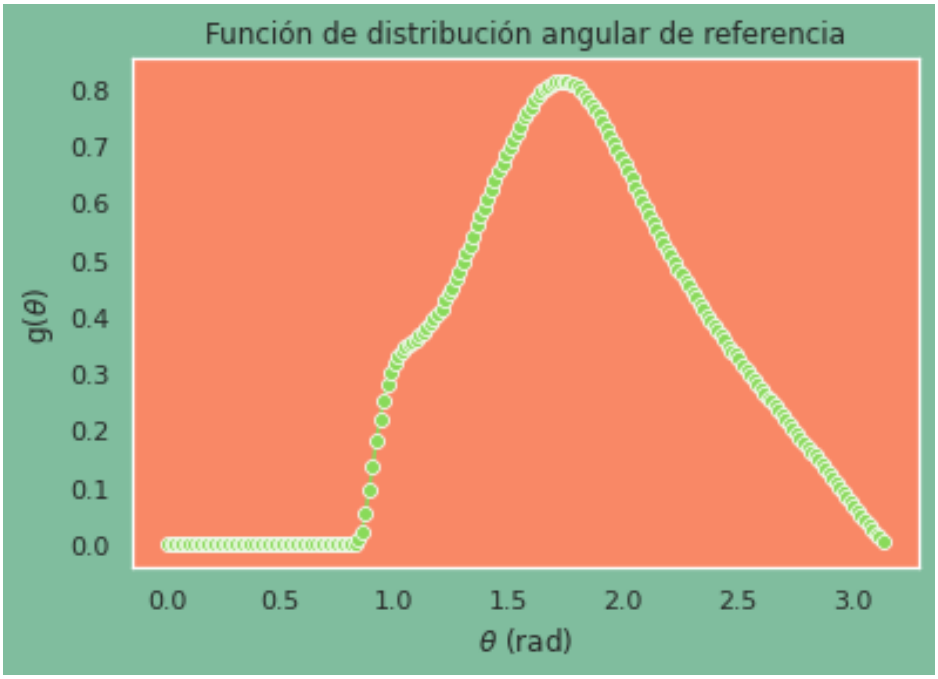

Figura 2: Distribución angular del sistema molecular de agua, usando una interpolación cúbica sobre los datos reportados del experimento

Los códigos que realizan los pasos descritos anteriormente se presentan a continuación:

```

def adf_discretization(nbins=200):
    """
    Se inicializan los parametros de la funci n angular discreta

    Args:
        nbins_theta: numero de intervalos a definir en pi radianes

    Returns:
        Arrays containing bin centers in theta direction and the standard
        deviation of the Gaussian smoothing kernel.
    """
    dtheta_bin = jnp.pi / float(nbins)
    adf_bin_centers = jnp.linspace(dtheta_bin / 2., jnp.pi - dtheta_bin / 2., nbins)
    sigma_ADF = util.f32(dtheta_bin)
    return adf_bin_centers, sigma_ADF
```

Código 4: Paso 1 del flujo de trabajo para obtener la distribución angular de tripletes de oxígenos

```

class ADFParams:

    reference_adf: Array
    adf_bin_centers: Array
    sigma_ADF: Array
    r_outer: Array
    r_inner: Array

adf_bin_centers, sigma_ADF = custom_quantity.adf_discretization(nbins=200)
reference_adf = np.loadtxt('data/experimental/O_O_O_ADF.csv')
adf_spline = sci_interpolate.interp1d(reference_adf[:, 0], reference_adf[:, 1],
    kind='cubic')
reference_adf = adf_spline(adf_bin_centers)
```

```
adf_struct = custom_quantity.ADFParams(reference_adf, adf_bin_centers, sigma_ADF,
    r_outer=0.318, r_inner=0.)
```

Código 5: Ejecución completa de la interpolación de la distribución angular a partir de los datos experimentales

4.2.3. Presión del sistema

Finalmente, se tomará como presión de referencia 1 bar. Se presenta el código para convertirlo a unidades del Sistema Internacional. La conversión se realiza directamente en código de la siguiente manera:

```
pressure_conversion = 16.6054  # from kJ/mol nm^-3 to bar
pressure_target = 1. / pressure_conversion  # 1 bar in kJ / mol nm^3
```

Código 6: Conversión de la presión a unidades del Sistema Internacional

A continuación, dado que se han declarado todas las observables a las cuales se desea llegar a través del algoritmo con redes neuronales, se define un diccionario en donde se guardarán estos valores:

```
target_dict = {'rdf': rdf_struct, 'adf': adf_struct, 'pressure': pressure_target}
```

Código 7: Diccionario donde se guardan las observables de referencia para el entrenamiento de las redes neuronales

Al concluir este apartado, se cuenta con el marco inicial de posiciones y velocidades del sistema, así como las variables objetivo del entrenamiento que se hará más adelante. Corresponde ahora detallar la generación de la trayectoria que se utilizará como variable independiente de entrada al modelo de grafos DimeNet++. Para generar dicha trayectoria, es necesario construir la propuesta de potencial, la cual se irá refinando por medio del entrenamiento del algoritmo de red neuronal. En la siguiente sección se detalla la manera de construir dicho potencial.

4.3. Inicialización de los parámetros del potencial

Como se mencionó en la parte teórica del proyecto de investigación, la propuesta de potencial que describa al sistema de moléculas de agua será de la forma

$$U = U_{\text{modelo clásico}} + U_{\text{red neuronal de grafos}} \tag{56}$$

donde

- $U_{\text{modelo clásico}}$ es un potencial que contiene pocos parámetros de ajuste, y dichos ajustes se realizan con base en métodos experimentales. En el caso del cúmulo del agua, el potencial será la parte repulsiva del potencial de Lennard-Jones [67].
- $U_{\text{red neuronal de grafos}}$ es la red neuronal de grafos DimeNet++ [39], con ciertas modificaciones que se expondrán con más detalle en esta sección.

Para llegar a este proceso, es necesario realizar una serie de pasos, que se enlistan a continuación:

1. Definir las variables de la simulación
2. Identificar tiempos de la simulación
3. Inicializar el potencial de Lennard-Jones
4. Inicializar la red neuronal de grafos

4.3.1. Definir las variables de la simulación

Siguiendo las indicaciones realizadas en el experimento, se fija la temperatura a 23 grados Celsius, y se hace su conversión a Kelvin, así como a las unidades de energía kbT, como se muestra a continuación:

```
system_temperature = 296.15  # Kelvin = 23 deg. Celsius
Boltzmann_constant = 0.0083145107  # in kJ / mol K
kbT = system_temperature * Boltzmann_constant
```

Código 8: Definición de la temperatura

Después, siguiendo los pasos de adquisición de datos que se mencionaron en el primer punto de la metodología, se obtiene el número de moléculas de agua presentes en el cúmulo, de modo que se pueda definir la densidad del sistema. Se muestra a continuación el código para realizar este cálculo:

```
file = 'data/confs/Water_experimental.gro'  # 901 particles
R_init, v, box = load_configuration(file)  # initial configuration
N = R_init.shape[0]

mass = 18.0154  # in u: O + 2 * H
density = mass * N * 1.66054 / jnp.prod(box)
print('Model Density:', density, 'g/l. Experimental density: 997.87 g/l')
```

Código 9: Densidad teórica y experimental del modelo

4.3.2. Identificar tiempos de la simulación

Partiendo de la distribución atómica inicial obtenida en la sección anterior, se hará evolucionar el sistema en un cierto periodo de tiempo. Y para cada cierto intervalo de tiempo transcurrido, la nueva distribución atómica resultante será guardado en la trayectoria.

Puntualizando los diferentes tiempos inmersos en la simulación, así como su respectiva duración, se tiene lo siguiente:

1. El tiempo total de la simulación será de 70 picosegundos.
2. Los primeros 10 picosegundos se utilizarán para que el sistema entre en equilibrio térmico.
3. La distribución atómica será actualizada en tiempos de 0.002 picosegundos.
4. Las distribuciones atómicas generadas en múltiplos de 0.1 picosegundos serán guardadas en el arreglo que define la trayectoria.

Cada elemento generado en los intervalos mencionados anteriormente contiene su propia explicación física y nomenclatura. Se especifica a continuación cada uno de ellos:

- Dada la energía potencial del sistema, es posible describir la dinámica del sistema, a partir de termostatos deterministas o estocásticos. Partiendo de la distribución atómica inicial, cada 0.002 se generarán nuevas coordenadas y velocidades de dicha distribución.

- Dado que el tamaño de paso que se utilizará en la simulación es una fracción de picosegundos, las nuevas distribuciones generadas por el termostato se encuentran muy correlacionadas, lo cual no ofrece información relevante del sistema y pueden producir inferencias sobreajustadas del potencial de interacción. Para evitar esta situación, se descartarán ciertas distribuciones para obtener una que sí contribuya significativamente en la descripción de la energía potencial del sistema. Las distribuciones que serán tomadas en cuenta para el entrenamiento y las que se descartarán se tomarán mediante la siguiente metodología: cada 0.1 picosegundos se obtendrán distribuciones significativas, que se denominarán, en este proyecto, estados de trayectoria. Para obtener dichas distribuciones significativas, si cada 0.002 segundos se obtiene una distribución cualesquiera, entonces tienen que transcurrir 50 pasos para obtener una distribución que sea considerada un estado de trayectoria.

- No obstante, en los primeros 10 picosegundos el sistema estará en proceso de equilibrio térmico, por lo que los estados generados en este periodo no se considerarán parte de la trayectoria final.

- Los 60 picosegundos restantes sí servirán para generar estados de trayectoria.

Se muestra a continuación una imagen ilustrativa del proceso descrito anteriormente:

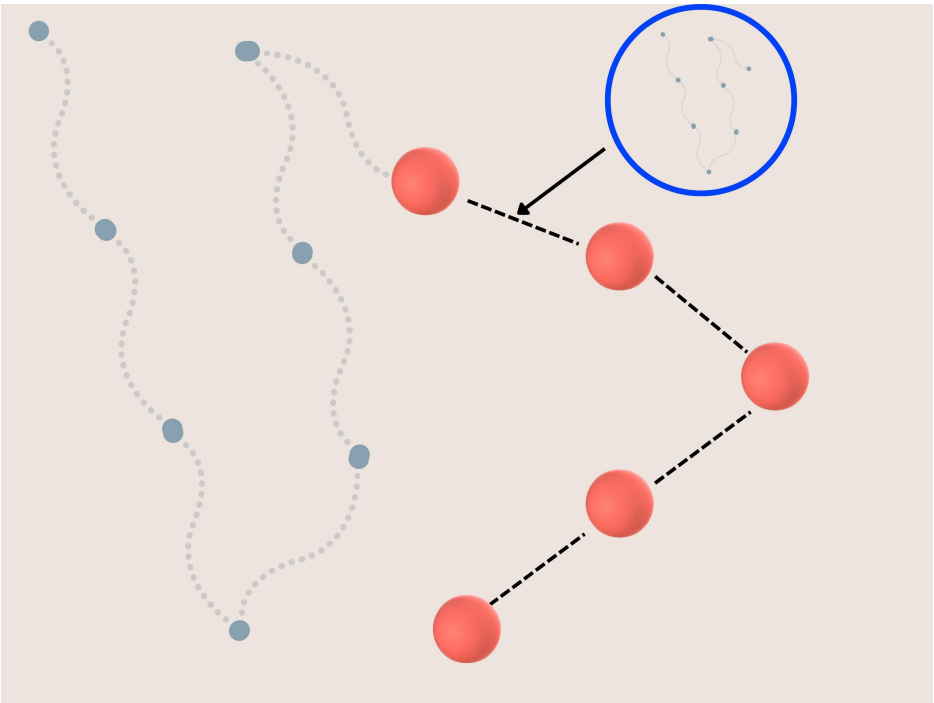

Figura 3: Representación gráfica de la generación de la trayectoria. Los puntos grises representan las salidas obtenidas cada 0.002 picosegundos. Las esferas grises representan estados generados en los primeros 10 picosegundos de la simulación, los cuales no son considerados para ser miembros de la trayectoria. Las esferas rojas representan estados generados en los 60 picosegundos efectivos de la simulación. La línea punteada negra representa salidas generadas en este lapso efectivo de generación de estados de trayectoria.

El siguiente código muestra la manera de calcular los estados de trayectoria explicados anteriormente, a partir de indicar los tiempos de la simulación.

```
@dataclasses.dataclass
class TimingClass:
    """
    Esta clase est  dise ada para recibir el n mero de estados a generar en la
    simulaci n, el n mero de estados a descartar por el tiempo de equilibrio y el n mero
    de salidas que se descartan previo a guardar una salida como estado de la trayectoria

    Attributes:
    num_printouts_production: N mero de estados generados
    num_dumped: N mero de estados descartado por el equilibrio
    timesteps_per_printout: N mero de pasos que debe dar la simulaci n para generar un
    estado

    """
    num_printouts_production: int
    num_dumped: int
    timesteps_per_printout: int

def process_printouts(time_step, total_time, t_equilib, print_every):
    """
    Funci n que calcula los elementos de la clase TimingClass a partir de los tiempos de
    la simulacion

    Args:
```

```
        time_step: Paso de tiempo entre cada salida
        total_time: Tiempo total de simulacion
        t_equilib: Tiempo de equilibrio termico
        print_every: Intervalo de tiempo en que una salida ser  considerada como estado de
    trayectoria

    Returns:
        La clase TimingClass

    """
    timesteps_per_printout = int(print_every / time_step)
    num_printouts_production = int((total_time - t_equilib) / print_every)
    num_dumped = int(t_equilib / print_every)
    timings_struct = TimingClass(num_printouts_production, num_dumped,
                                 timesteps_per_printout)
    return timings_struct
```

Código 10: Tiempos de la simulación almacenados en un arreglo optimizado de JAX

Una vez que se tienen los tiempos para la simulación, se prosigue a convertir las coordenadas iniciales en coordenadas fraccionales. Esto se hace mediante una normalización de las posiciones moleculares con respecto a los ejes de la celda unitaria. Para llevar a cabo este proceso, se presenta a continuación el código para realizar este cambio, teniendo en cuenta el siguiente aspecto: se ha realizado una modificación con respecto al código original de DiffTRe, ya que el código que se presentaba estaba sujeto a una versión antigua de JAX. Se presenta la nueva modificación a la función de conversión a coordenadas fraccionales:

```

from jax_md import space
def rectangular_boxtensor(box, spacial_dim):
    """
    La funci n contenida originalmente en esta secci n trabaja los  ndices  con un m todo
    llamado ops. Dicho m todo es obsoleto para versiones recientes de JAX, por lo que hay
    que redefinir bien el c digo.

    La intenci n de este c digo es crear una matriz bidimensional, la cual sobre su
    diagonal
    principal colocaremos los valores de la caja en las direcciones X,Y y Z.

    Primero, se crea una matriz identidad de 3x3, y despu s se prosigue a llenar la
    diagonal
    con los valores de las dimensiones.

    Inputs:
    * box = arreglo de JAX Numpy que contiene los valores de la dimensi n en cada
    direcci n
    * spatial_dim = es un valor num rico que indicar  el n mero de dimensiones a trabajar

    Outputs:
    * box_tensor = matriz de (spatial_dim x spatial_dim), en la cual sus valores ser n 0
                   excepto en la diagonal
    """

    box_tensor=jnp.eye(spacial_dim)
    box_tensor=box_tensor.at[jnp.diag_indices(spacial_dim)].set(box)
    return box_tensor


def scale_to_fractional_coordinates(R_init, box):
    spacial_dim = R_init.shape[1]
    box_tensor = rectangular_boxtensor(box, spacial_dim)
    inv_box_tensor = space.inverse(box_tensor)
    R_init = jnp.dot(R_init, inv_box_tensor)  # scale down to hypercube
```

```
    return R_init, box_tensor

R_init, box_tensor = custom_space.scale_to_fractional_coordinates(R_init, box)
```

Código 11: Función para convertir a coordenadas fraccionales las posiciones moleculares

Con esta conversión, se procede a imponer condiciones periódicas de frontera, utilizando como celda unitaria la caja definida en el código anterior. En este punto cabe recalcar que JAX trabaja principalmente con funciones, las cuales cumplen principalmente dos propósitos: descripción actual del sistema y manera de evolucionar a un nuevo estado del mismo. Dicho esto, para el caso concreto de la función creada en JAX MD sobre las condiciones de frontera periódicas, se obtienen dos funciones: una devuelve la manera de calcular la distancia entre partículas, mientras que otra describe la manera de mover una partícula en la posición R un desplazamiento dR. Así, al llamar a esta función de condiciones periódicas, obtenemos lo siguiente:

```
displacement, shift = space.periodic_general(box_tensor)
```

Se anexa a continuación una imagen que ilustra la peculiaridad de trabajar con condiciones periódicas de frontera:

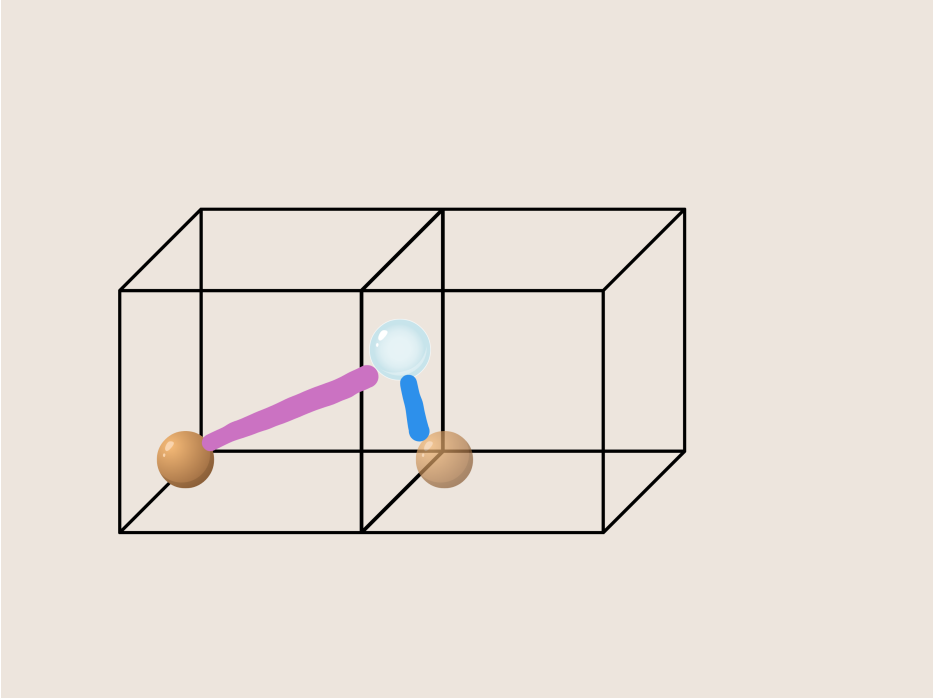

Figura 4: Ilustración del cálculo del vector desplazamiento hecho por JAX. Imponer condiciones de frontera hace que existan dos distancias asociadas a un mismo par de moléculas. La distancia representada con color rosa es aquella que se encuentra en la primera celda unitaria. Pero al repetir esa misma celda, debido a las condiciones periódicas de frontera, se aprecia que la distancia representada con el color azul es la menor distancia y, por lo tanto, describe la magnitud del vector desplazamiento.

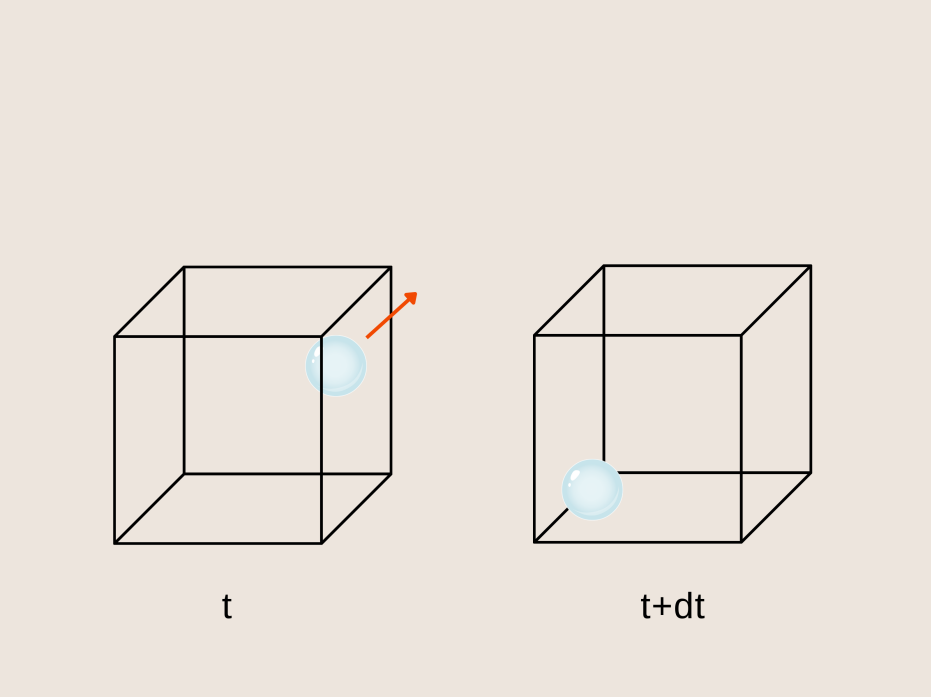

Figura 5: Representación del cambio de posición de una molécula en la celda unitaria. Si el cambio en la posición es tal que se sale de la celda unitaria, se debe reubicar correctamente dentro de la misma celda unitaria, según corresponda.

Para finalizar todos los prerrequisitos para comenzar a trabajar con los potenciales, es necesario definir reproducibilidad en el desarrollo del código. Para ello, JAX cuenta con su propio mecanismo de generación pseudoaleatoria de números. Consiste en llamar una llave, que se conoce mejor, en otros contextos de programación, como la semilla de aleatoriedad, la cual puede dividirse en otras llaves, para llevar a cabo diferentes inicializaciones que se necesiten en los diferentes procesos que requieran aleatoriedad. A diferencia de la librería Numpy, la cual usa el generador Mersenne-Twister [68] para obtener su aleatoriedad, el generador de JAX permite reproducibilidad en los diferentes procesos en los que se necesite rastrear algún proceso de cálculo. Esto significa que cada vez que llamamos a una función que requiera obtener datos aleatoriamente, si usamos la misma llave, obtendremos siempre el mismo resultado, caso contrario a lo que sucede con el generador de NumPy. Para lograr obtener mayor diversidad en los números aleatorios de una llave, es conveniente definir las llaves secundarias que se mencionaron previamente, de modo que por cada nuevo proceso aleatorio a llamar, se declare una nueva llave secundaria.

A continuación se muestra el código en donde se toma la llave cero, y con esa misma se definen dos llaves: una para inicializar los parámetros de la red DimeNet++ y la segunda para inicializar el resto de elementos asociados al experimento

```
key = random.PRNGKey(0)  # define random seed for initialization of model and simulation
model_init_key, simuation_init_key = random.split(key, 2)
```

Código 12: Definición de las llaves que iniciarán los parámetros de la red y de la simulación

Teniendo declarados todos los elementos de la simulación, se prosigue formalmente a la construcción del potencial.

4.3.3. Inicializar el potencial de Lennard-Jones

Como se mencionó al inicio de esta sección, se propone una función de la energía potencial, donde se incluye un potencial con pocos parámetros, que en el caso de este trabajo será el término repulsivo de Lennard-Jones. Previo a su debida inicialización, se tiene que discutir una característica que hace distinta a la librería de JAX de las demás librerías de alto nivel que tiene Python para Aprendizaje Automático:

JAX está diseñado para poder compilar las funciones, y con ello poder realizar ejecuciones más rápidas, tanto en CPU como en tarjetas gráficas. Esencialmente, la compilación se logra cuando se cumplen las siguientes características:

- Las entradas se definen en los argumentos de la función y las salidas únicamente se definen al terminar la definición de la función. Esto quiere decir que no se pueden usar variables globales dentro de la función, teniendo modificaciones en su valor fuera de ella, así como no se pueden imprimir textos o valores dentro de la misma, ya que se considera una salida extra que no contempla originalmente el cuerpo de la función.
- Las dimensiones de los argumentos de entrada deben ser estáticos. Si cambian las dimensiones, entonces JAX tiene que generar una nueva función compilada, lo cual toma tiempo para lograrlo.

Cuidando los dos puntos anteriores para trabajar y aprovechar al máximo la funcionalidad de JAX, tenemos que definir dimensiones estáticas para los potenciales. La estrategia para lograr dichas dimensiones es la siguiente: hay que definir un arreglo de prueba, que contenga el máximo de elementos que serán considerados para realizar los cálculos de la función. Existen módulos de la librería JAX MD que permiten lograr este cometido.

El primer arreglo estático a definir es aquel que define los vecinos de cada molécula. El radio de corte a utilizar es de 0.3 nanómetros. Para este paso de inicialización se obtendrán un cierto número de vecinos por cada molécula, pero se quiere compilar en memoria las dimensiones de dicho arreglo, para todas las subsecuentes salidas que se den en la evolución temporal del sistema. Entonces, se agrega una tolerancia de la siguiente manera:

- La vecindad será de 0.35 nanómetros, de modo que se agreguen vecinos extras a la vecindad original considerada.
- De todas las vecindades generadas en este paso, se encuentra el número máximo de vecinos en una misma vecindad, considerando el extra detallado en el punto anterior, y dicho valor se multiplica por un factor de 1.5, de modos que se incluyan aún más elementos. En el caso concreto de esta tesis, se tiene como máximo 9 vecinos originalmente, y los arreglos finales contienen 13 elementos. Los restantes elementos se identifican con el índice 901, el cual no hace referencia a ningún átomo del arreglo, ya que los 901 elementos del sistema molecular contienen etiquetas que van del 0 al 900.

Se anexa a continuación una ilustración en donde se muestran los pasos explicados anteriormente:

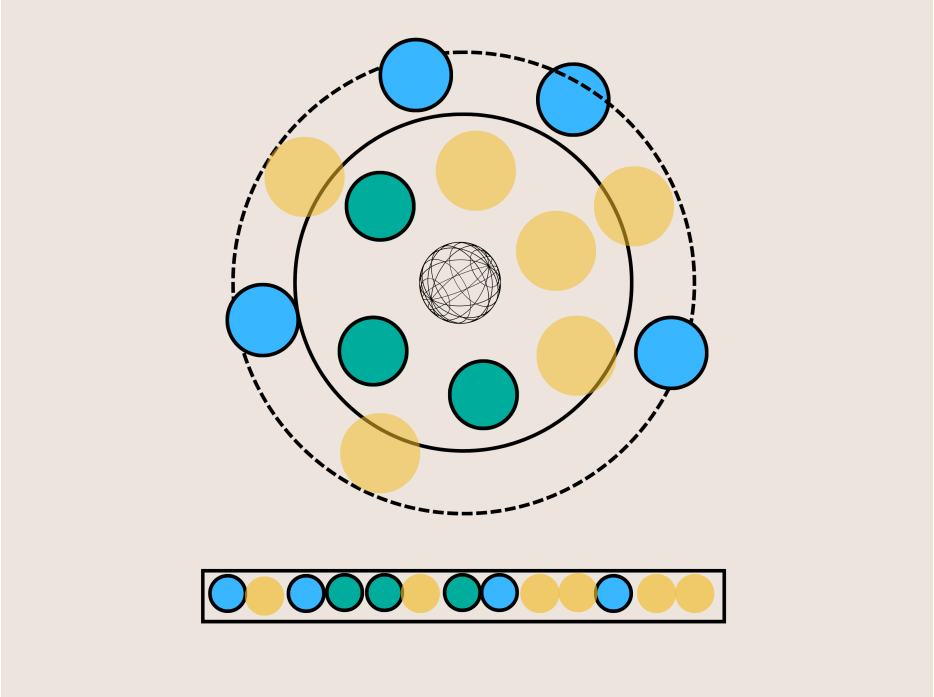

Figura 6: Representación de la creación de la vecindad estática de los elementos del sistema molecular. Los elementos de color verde representan a los vecinos originales. Luego, al extender la vecindad 0.05 nanómetros, representada en el dibujo con la circunferencia punteada, se añaden los vecinos azules. Finalmente, buscando el número máximo de vecinos en todo el sistema, se multiplica dicho máximo por 1.5 y la diferencia se añade como si pintáramos en la vecindad vecinos extras, representados de color amarillo. Así, el sistema compila en memoria que las vecindades que se vayan a crear más adelante van a tener un máximo de elementos y dicha dimensión no cambiará a lo largo del código. El arreglo de la vecindad del átomo representado anteriormente se dibuja con el rectángulo en la parte inferior. Nótese que no hay un orden específico de cómo se guardan los elementos en el arreglo final.

Los pasos para inicializar en código el arreglo de vecinos se presenta a continuación:

```
box_nbrs = jnp.ones(3)
    neighbor_fn = partition.neighbor_list(displacement, box_nbrs, run.rcut,
    dr_threshold=0.05, capacity_multiplier=1.5,
                                    disable_cell_list=True)
    nbrs_init = neighbor_fn(R_init)
```

Código 13: Generación del arreglo de vecinos para dimensionar adecuadamente las entradas de los potenciales

Este fue importante, ya que todos los cálculos que se realicen a partir de ahora se llevarán a cabo para todos los vecinos de las diferentes vecindades constituidas.

Aterrizando concretamente al potencial de Lennard-Jones, únicamente se va a utilizar el término repulsivo de dicho potencial, el cual es

$$U_{LJR} = \epsilon \left(\frac{\sigma}{r}\right)^{12} \tag{57}$$

donde

- ϵ =1
- σ=0.3165

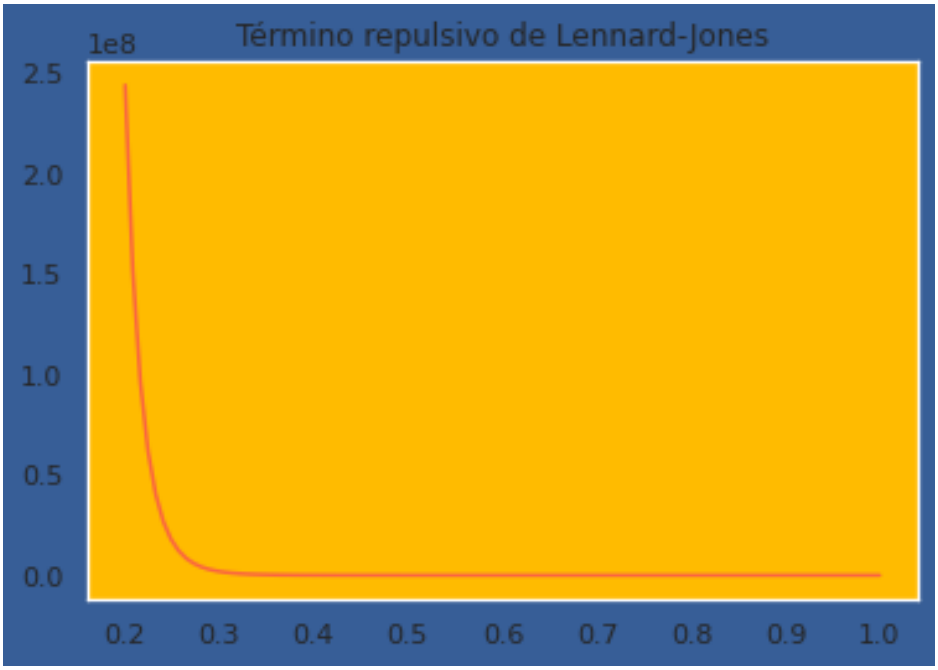

Figura 7: Gráfica del término repulsivo del potencial de Lennard-Jones

Hay que notar que esta función está orientada a un par de partículas. Para poder aplicarlas a un conjunto de elementos dentro de una cierta vecindad, la librería de JAX MD ofrece herramientas para poder extender el cálculo a todos los vecinos de una cierta molécula de interés. Para ello, primero se tiene que definir la función para un par, y luego dicha función se tiene que integrar a otra función, para que pueda hacer el mismo cálculo a todos los elementos de interés, de manera eficiente, según las ventajas que presenta JAX.

Una vez que se construye la función, se debe inicializar esta misma, indicando la función de desplazamiento, así como los parámetros que se mencionaron anteriormente para el caso del agua.

El código en JAX para lograr dicho cometido es el siguiente:

```
def generic_repulsion(dr: Array,
                      sigma: Array=1.,
                      epsilon: Array=1.,
                      exp: Array=12.,
                      **dynamic_kwargs) -> Array:
    """
    Interacci n repulsiva entre esferas s lidas, con expresi n: U = epsilon * (sigma /
    r)**exp.

    Args:
      dr: Arreglo con las distancias entre pares de atomos.
      sigma: Rescala de repulsion
      epsilon: escala de energia
      exp: exponente

    Returns:
      Arreglo de energias
    """

    dr = jnp.where(dr > 1.e-7, dr, 1.e7)  # save masks dividing by 0
    idr = (sigma / dr)
    U = epsilon * idr ** exp
    return U

def generic_repulsion_neighborlist(displacement_or_metric: DisplacementOrMetricFn,
```

```
                    box_size: Box=None,
                    species: Array=None,
                    sigma: Array=1.0,
                    epsilon: Array=1.0,
                    exp: Array=12.,
                    r_onset: Array = 0.9,
                    r_cutoff: Array = 1.,
                    dr_threshold: float=0.2,
                    per_particle: bool=False,
                    capacity_multiplier: float=1.25,
                    initialize_neighbor_list: bool=True):
    """
    Funcion que permite calcular optimamente el termino de repulsi n de Lennard-Jones en
    una misma vecindad

    """

    sigma = jnp.array(sigma, dtype=f32)
    epsilon = jnp.array(epsilon, dtype=f32)
    exp = jnp.array(exp, dtype=f32)
    r_onset = jnp.array(r_onset, dtype=f32)
    r_cutoff = jnp.array(r_cutoff, dtype=f32)

    energy_fn = smap.pair_neighbor_list(
      multiplicative_isotropic_cutoff(generic_repulsion, r_onset, r_cutoff),
      space.canonicalize_displacement_or_metric(displacement_or_metric),
      species=species,
      sigma=sigma,
      epsilon=epsilon,
      exp=exp,
      reduce_axis=(1,) if per_particle else None)

    if initialize_neighbor_list:
        assert box_size is not None
        neighbor_fn = partition.neighbor_list(displacement_or_metric, box_size, r_cutoff,
    dr_threshold,
                                              capacity_multiplier=capacity_multiplier)
        return neighbor_fn, energy_fn

    return energy_fn

prior_fn = custom_energy.generic_repulsion_neighborlist(displacement, sigma=0.3165,
    epsilon=1., exp=12,
                                                        initialize_neighbor_list=False)
```

Código 14: Generalización de la parte repulsiva del potencial de Lennard-Jones para una vecindad de moléculas.

Con este apartado se describe el primer término del potencial del sistema. Dicho potencial no se incluirá en el proceso de entrenamiento de DimeNet++ y optimización de parámetros de dicha red. La física contenida en este término es suficiente para poder utilizar la red neuronal en virtud de mejorar la descripción de la energía del sistema.

4.3.4. Inicializar el potencial DimeNet++

Para poder trabajar con la red de grafos DimeNet++, se necesitan identificar las aristas que se forman entre los elementos del sistema. A su vez, se debe tener a la mano la información respectos a los elementos que conforman a una cierta vecindad. Es decir, dada una vecindad arbitraria de un i-ésimo átomo, existirán j-ésimos vecinos, los cuales a su vez son el centro de otras vecindades, con sus respectivos k-ésimos vecinos. Teniendo esta información, se aplican las transformaciones de las distancias entre pares y de los ángulos entre tripletes por medio de las funciones radiales de Bessel y de los armónicos esféricos para los ángulos. Con esos datos, se realizan los procesos

de los diferentes bloques que conforman la red. En este apartado se explica a detalle cada paso, con sus respectivas ilustraciones y código de procedimiento.

Recordando que se tiene una lista de vecinos con elementos de tolerancia adicionales, se deben extraer los elementos que realmente conforman la vecindad de las moléculas. Una vez identificado los elementos reales, se procede a definir los tripletes de moléculas para los ángulos. Dicha definición parte de hacer todas las posibles combinaciones de tripletes posibles, según se pueda realizar en la consulta de los vecinos de los elementos pertenecientes en una misma vecindad. Se anexa una imagen para visualizar dichas combinaciones:

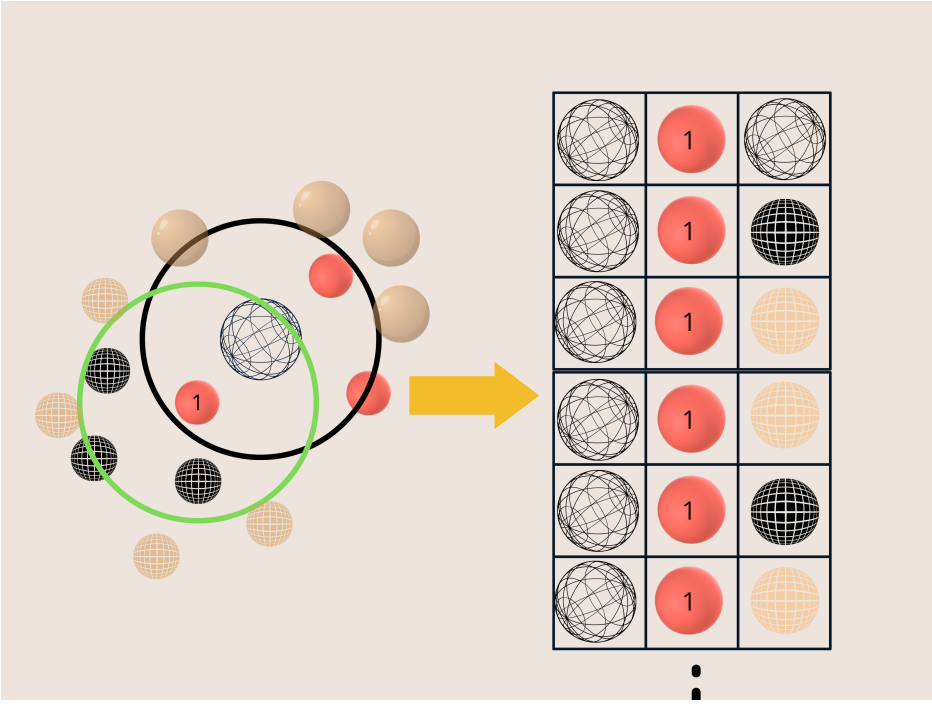

Figura 8: Dada un i-ésimo átomo, representado por la esfera transparente, tiene una vecindad con 3 elementos en ella, identificados con el color rojo, mientras que el arreglo original proporcionado, por la dimensión estática de los arreglos, podría considerarse más elementos que no son miembros reales de la vecindad. Luego, tomando el primer vecino rojo, también se define su propia vecindad, en donde el i-ésimo átomo es ahora el vecino, junto con otros k-ésimos vecinos, localizados con color negro, así como los otros elementos extras considerados inicialmente.

Utilizando la visualización de la Figura 8, se busca quedar con las combinaciones reales de tripletes de elementos. Las técnicas para realizar este filtrado se conocen como máscaras. Se presentan a continuación las 3 máscaras a aplicar al conjunto completo de combinaciones posibles:

1. El primer y el último elemento de la tripleta deben ser diferentes
2. El tercer elemento debe ser vecino del segundo elemento de la tripleta.
3. El tercer elemento debe ser un índice diferente al 901 (valor por default para llenar los espacios sobrantes del arreglo de vecinos posibles).

Una vez aplicadas estas máscaras, se obtienen las tripletas de interés, como se muestra en la siguiente imagen:

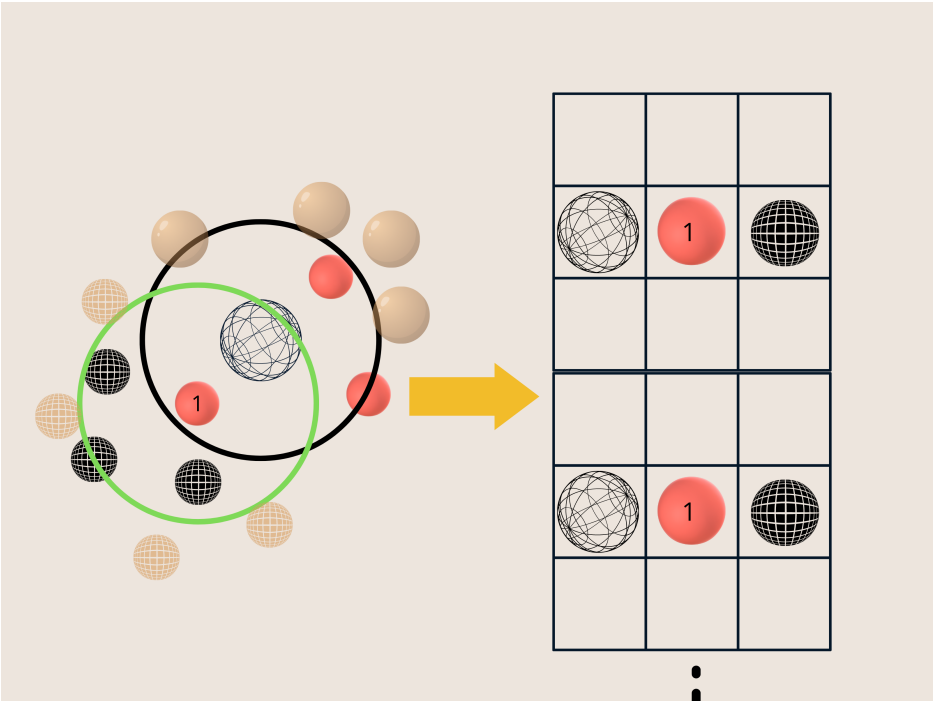

Figura 9: Ilustración de la aplicación de las máscaras a todos los posibles resultados ejemplificados en la Figura 8

El último elemento a crear es el polinomio que permitirá convertir las funciones de Bessel y los armónicos esféricos en funciones de clase $\mathbf{C}^2$, condición necesaria para trabajar con redes neuronales. Se anexa la gráfica de dicho polinomio:

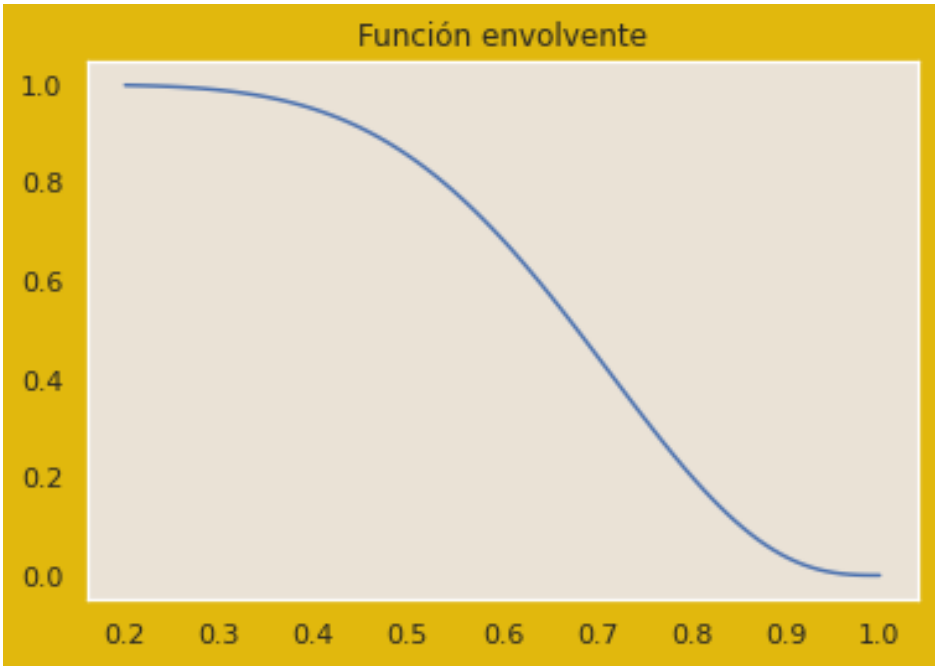

Figura 10: Polinomio envolvente definido para recuperar diferenciabilidad de las funciones de Bessel y los armónicos esféricos, necesarios para describir las distancia y los ángulos.

Comenzando formalmente con la red DimeNet++, se deben crear los primeros mensajes o incrustes de los pares del sistema que comparten una arista. Tomando la función de Bessel para representar la distancia entre dichos elementos, así como el tipo de elemento que representan, se usan capas lineales para crear una representación

inicial de 32 entradas. Se anexa la imagen que ilustra la capa de incrustación descrita anteriormente:

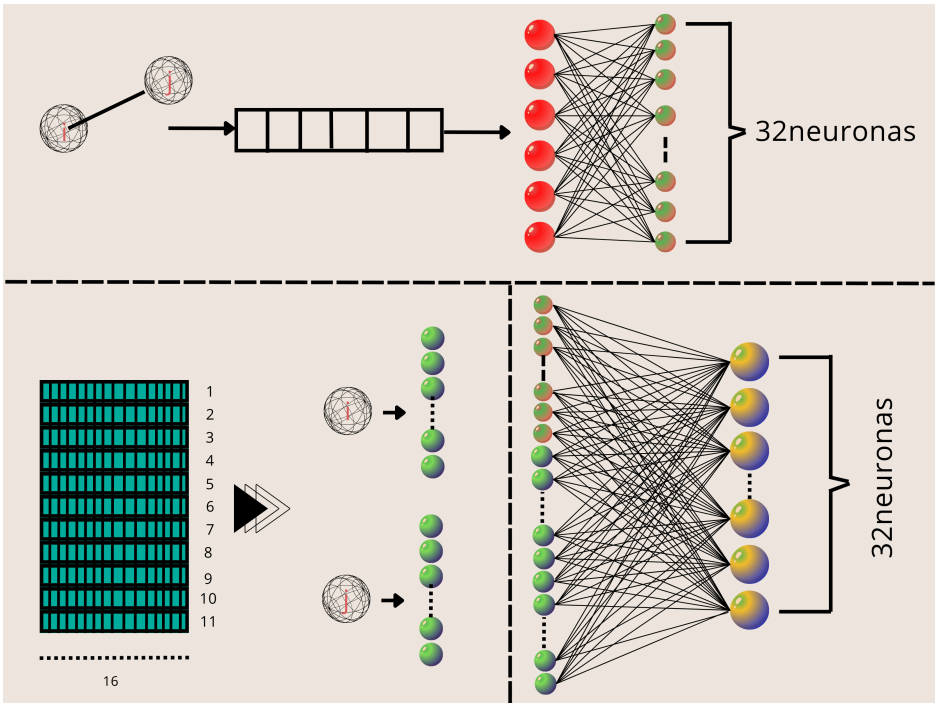

Figura 11: Capa de incrustación de la red de grafos DimeNet++. Utilizando seis funciones de Bessel se describe la distancia, mientras que con una matriz de identificación de especies se manda cada elemento a su respectiva especie. Dado que la molécula de agua se estudia a nivel granular, solo se usa una fila de esa matriz de identificación de especies

Después de inicializar los mensajes, se mide la interacción entre los elementos de la vecindad por medio del paso de mensajes. El bloque de interacción consiste en actualizar el nivel de interacción entre pares, pero considerando a los vecinos de la vecindad a la que pertenecen. Esto es, para medir la interacción entre el átomo i y el átomo j, se mide a su vez la interacción entre el átomo j y todos sus vecinos que conforman su vecindad, para después intercambiar dicha información con el átomo i.

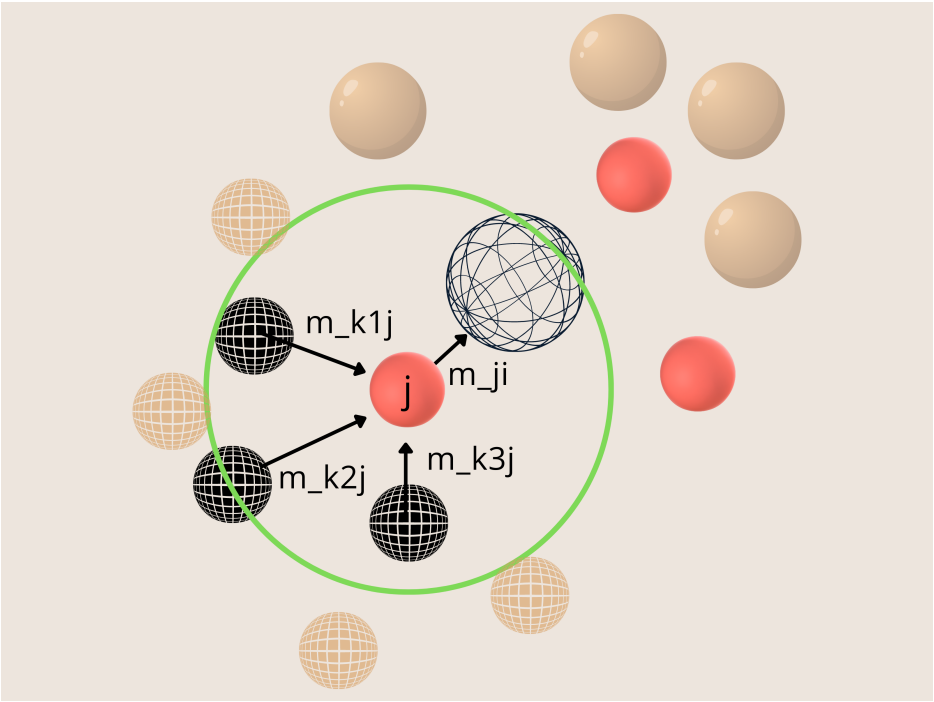

Figura 12: Interacción entre el elemento i y el elemento j, considerando los elementos de la vecindad j

El último paso consiste en considerar el mensaje de la capa anterior asociado a dicho par, de modo que se obtenga un nuevo mensaje, que se comparte a la siguiente capa de interacción, así como a la capa de salida. Se anexan los pasos de este bloque:

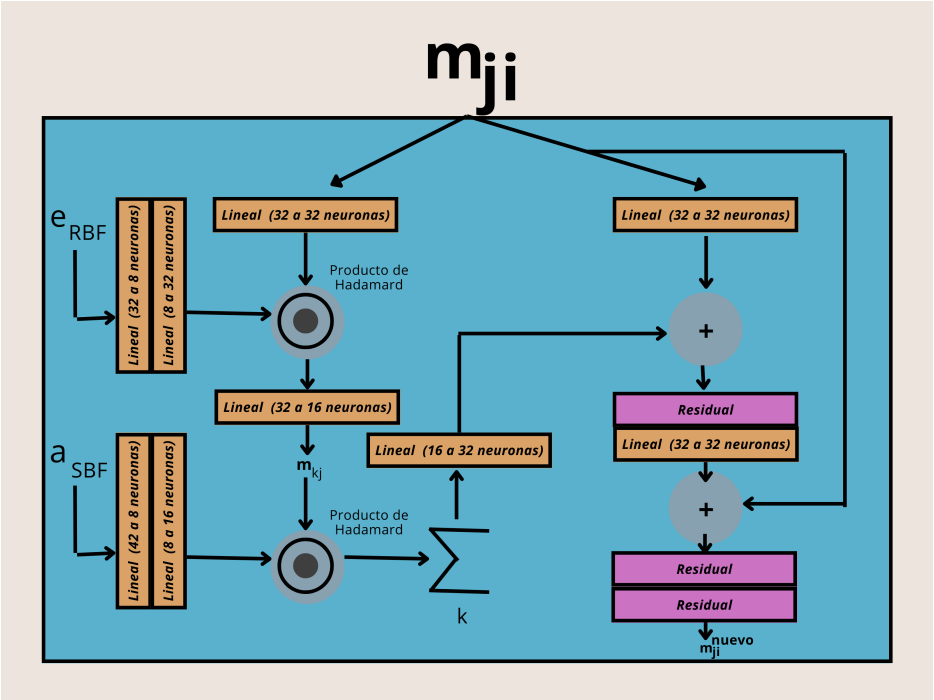

Figura 13: Capa de interacción de la red DimeNet++. En la imagen se señalan las dimensiones en las que se transforman los vectores de información en cada paso dado.

La última capa a estudiar es la de salida, la cual corresponde a la predicción que se busca hacer con la red neuronal y que corresponde a la energía potencial del sistema. Se anexa una imagen con los elementos de dicha

capa:

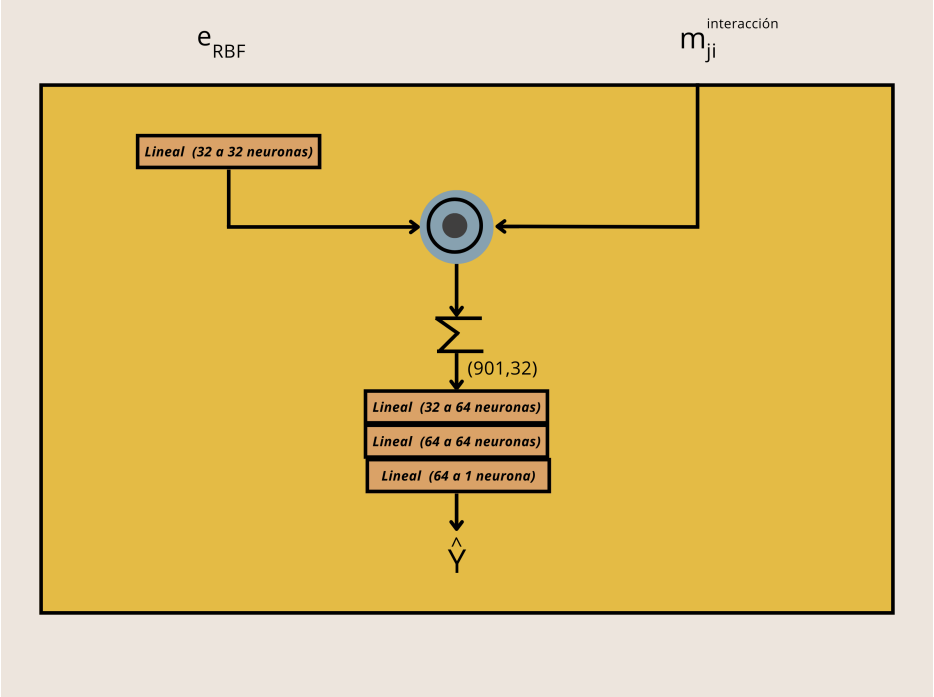

Figura 14: Capa de salida de la red DimeNet++. En la imagen se señalan las dimensiones en las que se transforman los vectores de información en cada paso dado.

En cuanto a la parte del código, se incluye a continuación los pasos para construir cada uno de los bloques explicados e ilustrados en los pasos anteriores:

```
    class DimeNetPPEnergy(hk.Module):


    def __init__(self,
                 r_cutoff: float,
                 n_species: int,
                 n_particles: int,
                 embed_size: int = 32,
                 n_interaction_blocks: int = 4,
                 num_residual_before_skip: int = 1,
                 num_residual_after_skip: int = 3,
                 out_embed_size=None,
                 type_embed_size=None,
                 angle_int_embed_size=None,
                 basis_int_embed_size = 8,
                 num_dense_out: int = 3,
                 num_RBF: int = 6,
                 num_SBF: int = 7,
                 activation=jax.nn.swish,
                 envelope_p: int = 6,
                 init_kwargs: Dict[str, Any] = None,
                 name: str = 'Energy'):
        super(DimeNetPPEnergy, self).__init__(name=name)

        # input representation:
        self.rbf_layer = RadialBesselLayer(r_cutoff, num_radial=num_RBF,
    envelope_p=envelope_p)
        self.sbf_layer = SphericalBesselLayer(r_cutoff, num_radial=num_RBF,
    num_spherical=num_SBF, envelope_p=envelope_p)
```

```
28
29          # build GNN structure
30          self.n_interactions = n_interaction_blocks
31          self.output_blocks = []
32          self.int_blocks = []
33          self.embedding_layer = EmbeddingBlock(embed_size, n_species,
    type_embed_size=type_embed_size,
34                                                activation=activation,
    init_kwargs=init_kwargs)
35          self.output_blocks.append(OutputBlock(embed_size, n_particles, out_embed_size,
    num_dense=num_dense_out,
36                                                num_targets=1, activation=activation,
    init_kwargs=init_kwargs))
37
38          for _ in range(n_interaction_blocks):
39              self.int_blocks.append(InteractionBlock(embed_size, num_residual_before_skip,
    num_residual_after_skip,
40                                                      activation=activation,
    angle_int_embed_size=angle_int_embed_size,
41
    basis_int_embed_size=basis_int_embed_size, init_kwargs=init_kwargs))
42              self.output_blocks.append(OutputBlock(embed_size, n_particles, out_embed_size,
    num_dense=num_dense_out,
43                                                    num_targets=1, activation=activation,
    init_kwargs=init_kwargs))
44
45      def __call__(self, distances, angles, species, pair_connections, angular_connections)
    -> jnp.ndarray:
46          rbf = self.rbf_layer(distances)  # correctly masked by construction: masked
    distances are 2 * cut-off --> rbf=0
47          sbf = self.sbf_layer(distances, angles, angular_connections)  # is masked too
48
49          messages = self.embedding_layer(rbf, species, pair_connections)
50          per_atom_quantities = self.output_blocks[0](messages, rbf, pair_connections)
51
52          for i in range(self.n_interactions):
53              messages = self.int_blocks[i](messages, rbf, sbf, angular_connections)
54              per_atom_quantities += self.output_blocks[i + 1](messages, rbf,
    pair_connections)
55
56          predicted_quantities = util.high_precision_sum(per_atom_quantities, axis=0)  # sum
    over all atoms
57          return predicted_quantities
```

El código mostrado anteriormente sobre la arquitectura de la red es llamada de la siguiente manera, en donde se obtendrán la función que inicializa los parámetros, así como la función que los actualiza:

```
1   init_fn, GNN_energy = custom_energy.DimeNetPP_neighborlist(displacement, R_init,
    nbrs_init, run.rcut, embed_size=run.embed, n_interaction_blocks=run.interactions)
2   init_params = init_fn(model_init_key, R_init, neighbor=nbrs_init)
```

Para finalizar formalmente con esta sección, se creará una función, con la cual se identifiquen aquellos parámetros de entrada estáticos que permitan una compilación más efectiva y rápida, tanto de la energía predicha por el potencial de Lennard-Jones, así como el de DimeNet++:

```
1   def energy_fn_template(energy_params):
2       gnn_energy = partial(GNN_energy, energy_params)
3       def energy(R, neighbor, **dynamic_kwargs):
4           return gnn_energy(R, neighbor=neighbor, **dynamic_kwargs) + prior_fn(R,
    neighbor=neighbor, **dynamic_kwargs)
```

```
        return jit(energy)
```

4.4. Dinámica Molecular

En este apartado se desarrolla el código relacionado con la dinámica molecular del sistema de moléculas de agua, utilizando como sustento teórico los puntos señalados en el capítulo 2.

Lo primero que hay que definir es el termostato a trabajar. Como se desarrolló en el marco teórico, se usará el de Nosé-Hoover. Para ello, siguiendo la filosofía de trabajo de JAX, se van a definir tres funciones necesarias para trabajar con este tipo de dinámica:

- Una función de inicialización.
- Otra función para actualizar la dinámica en medio paso.
- Finalmente, una función que actualice las masas de los termostatos

Comenzando con la inicialización, se definen los arreglos de posiciones y velocidades en ceros, con una longitud igual al número de termostatos que se hayan elegido añadir a la cadena de Nosé-Hoover. Luego, tomando a la cantidad $k_B T$ como una cantidad que el usuario añadirá como parámetro de la función, así como la τ que tiene que ver con la oscilación fundamental del sistema, descrita en la ecuación 50, se define otro arreglo para las masas de los termostatos, dada por la ecuación 49. Como el primer termostato tiene interacción con todas las partículas del sistema, tal como se señaló en 48, entonces utilizando la sintaxis adecuada para actualizar arreglos, la primera entrada del arreglo de las masas debe contener el término adecuado, $Q_{\eta 1}$, de la ecuación 48.

```
    def init_fn(degrees_of_freedom, KE, kT):
    xi = jnp.zeros(chain_length, KE.dtype)
    v_xi = jnp.zeros(chain_length, KE.dtype)

    Q = kT * tau ** f32(2) * jnp.ones(chain_length, dtype=f32)
    Q = ops.index_update(Q, 0, Q[0] * degrees_of_freedom)
    return NoseHooverChain(xi, v_xi, Q, tau, KE, degrees_of_freedom)
```

Con respecto a la función que actualizará los respectivos arreglos mencionados previamente, hay que crear una subrutina para realizar todos los cálculos necesarios. Se describe a continuación cómo se definirá dicha subrutina, la cual se basa en los pasos descritos en [69]:

1. Lo primero es dejar marcados los diferentes pasos de tiempo que se utilizarán, según marcan las ecuaciones de la referencia mencionada. También, hay que tener cuidado con la manera en que se hará el recorrido de los índices. Como señala el algoritmo señalado por Martyna y colaboradores [69], se debe realizar un primer cálculo, comenzando a partir del último termostato, hacia el primero. Después, realizar otro recorrido de cálculos, pero ahora comenzando desde el primero hacia el último.

2. Con lo señalado previamente en el primer paso, se debe definir una función que sea la que realice los cálculos de la primera vuelta mencionada. Para ello, se debe señalar puntualmente el valor inicial de la fuerza G_i, asociada a los termostatos. Esta primera fuerza permite redefinir la velocidad concerniente al M-ésimo termostato. A partir de este punto, comienza la iteración hacia atrás, hacia el primer termostato. Al comenzar el ciclo, dada la velocidad a actualizar, se debe actualizar también el valor de la fuerza del termostato, como se indica en el marco teórico. También, se debe calcular el término de re escalamiento que se indica en el algoritmo de Martyna y colaboradores señalado previamente. Finalmente, con estos elementos, ya se puede redefinir la velocidad, acorde a lo señalado en el desarrollo teórico. Para hacer eficiente el código, utilizando la función scan de JAX, se definen los índices de los termostatos faltantes por recorrer, y se comienza el ciclo. La salida del ciclo es un arreglo con las nuevas velocidades calculadas, por lo que hay que actualizar el arreglo donde se iban guardando las velocidades con esta nueva lista de cálculos.

3. Finalmente, para el ciclo faltante, se implementa una metodología similar, en el sentido de que manualmente se debe hacer la actualización para el primer termostato, y a partir de ahí comenzar a iterar hacia adelante, hasta llegar al último termostato. Se calculan las fuerzas de los termostatos, así como el término de escalamiento, correspondientes a cada termostato. Una vez realizado el ciclo hacia atrás y este nuevo ciclo hacia delante, la función devolverá el arreglo de posiciones y velocidades.

```
    def substep_fn(delta, V, state, kT):
    """Apply a single update to the chain parameters and rescales velocity."""

    xi, v_xi, Q, _tau, KE, DOF = dataclasses.astuple(state)

    delta_2 = delta   / f32(2.0)
    delta_4 = delta_2 / f32(2.0)
    delta_8 = delta_4 / f32(2.0)

    M = chain_length - 1

    G = (v_xi[M - 1] ** f32(2) * Q[M - 1] - kT) / Q[M]
    v_xi = ops.index_add(v_xi, M, delta_4 * G)

    def backward_loop_fn(v_xi_new, m):
      G = (v_xi[m - 1] ** 2 * Q[m - 1] - kT) / Q[m]
      scale = jnp.exp(-delta_8 * v_xi_new)
      v_xi_new = scale * (scale * v_xi[m] + delta_4 * G)
      return v_xi_new, v_xi_new
    idx = jnp.arange(M - 1, 0, -1)
    _, v_xi_update = lax.scan(backward_loop_fn, v_xi[M], idx, unroll=2)
    v_xi = ops.index_update(v_xi, idx, v_xi_update)

    G = (f32(2.0) * KE - DOF * kT) / Q[0]
    scale = jnp.exp(-delta_8 * v_xi[1])
    v_xi = ops.index_update(v_xi, 0, scale * (scale * v_xi[0] + delta_4 * G))

    scale = jnp.exp(-delta_2 * v_xi[0])
    KE = KE * scale ** f32(2)
    V = V * scale

    xi = xi + delta_2 * v_xi

    G = (f32(2) * KE - DOF * kT) / Q[0]
    def forward_loop_fn(G, m):
      scale = jnp.exp(-delta_8 * v_xi[m + 1])
      v_xi_update = scale * (scale * v_xi[m] + delta_4 * G)
      G = (v_xi_update ** 2 * Q[m] - kT) / Q[m + 1]
      return G, v_xi_update
    idx = jnp.arange(M)
    G, v_xi_update = lax.scan(forward_loop_fn, G, idx, unroll=2)
    v_xi = ops.index_update(v_xi, idx, v_xi_update)
    v_xi = ops.index_add(v_xi, M, delta_4 * G)

    return V, NoseHooverChain(xi, v_xi, Q, _tau, KE, DOF), kT
```

Teniendo la subrutina descrita anteriormente, se procede a describir la función que realiza medio paso de la simulación. Aquí se deben considerar los pesos de Suzuki-Yoshida, cuyos valores se obtuvieron de la aplicación OpenMM, la cual está enfocada también a dinámica molecular [70], y con los cuales se permite definir los intervalos de tiempo apropiados para llevar a cabo la propagación del baño de calor, representada por los termostatos. Habiendo definido adecuadamente los intervalos de tiempo, tomando en cuenta los pesos mencionados, se procede a realizar la subrutina mencionada anteriormente, para el número de pasos acordes al número de cadenas y de pesos de Suzuki-Yoshida a tomar en cuenta.

```
    def half_step_chain_fn(V, state, kT):
    if chain_steps == 1 and sy_steps == 1:
      V, state, _ = substep_fn(dt, V, state, kT)
      return V, state

    delta = dt / chain_steps
    ws = jnp.array(SUZUKI_YOSHIDA_WEIGHTS[sy_steps], dtype=V.dtype)
    def body_fn(cs, i):
      d = f32(delta * ws[i % sy_steps])
      return substep_fn(d, *cs), 0
    V, state, _ = lax.scan(body_fn,
                           (V, state, kT),
                           jnp.arange(chain_steps * sy_steps))[0]
    return V, state
```

Finalmente, para el caso de la actualización de las masas, se lleva a cabo el mismo procedimiento mencionado en la función de inicialización, donde se debe tener cuidado en definir apropiadamente la masa del primer termostato.

```
    def update_chain_mass_fn(state, kT):
    xi, v_xi, Q, _tau, KE, DOF = dataclasses.astuple(state)

    Q = kT * _tau ** f32(2) * jnp.ones(chain_length, dtype=f32)
    Q = ops.index_update(Q, 0, Q[0] * DOF)

    return NoseHooverChain(xi, v_xi, Q, _tau, KE, DOF)
```

Un último paso por aclarar para poder describir por completo el flujo de pasos para inicializar la cadena de termostatos, es incluir el algoritmo de la velocidad de Verlet. Entonces, se define una función aparte, que realice los pasos marcados en las ecuaciones 51 y 52

```
    dt = f32(dt)
  dt_2 = f32(dt / 2)
  dt2_2 = f32(dt ** 2 / 2)

  R, V, F, M = state.position, state.velocity, state.force, state.mass

  Minv = 1 / M

  R = shift_fn(R, V * dt + F * dt2_2 * Minv, **kwargs)
  F_new = force_fn(R, **kwargs)
  V += (F + F_new) * dt_2 * Minv
  return dataclasses.replace(state,
                             position=R,
                             velocity=V,
                             force=F_new)
```

Con este último algoritmo aclarado, se procede a explicar el flujo del algoritmo que realiza los cálculos alrededor de la dinámica de la cadena de termostatos de Nosé-Hoover

1. La primera parte consiste en inicializar la función encargada de realizar los pasos de la dinámica de la cadena de termostatos. Para ello, esta función debe recibir como parámetros de entrada la fuerza; la función que realizará el cambio de posición de las moléculas; una delta de cambio en el tiempo; la longitud de la cadena; el número de pasos intermedios para cubrir todo un paso completo, definido por la delta de tiempo dada; un entero asociado a los pesos a utilizar de Yoshida-Suzuki; finalmente, la τ asociada a la definición de las masas de los termostatos.

2. Con estos elementos identificados, se realiza la función de inicio, con la cual se inicializa primero las masas de los termostatos, así como las velocidades. Las velocidades se inicializan a partir de la una distribución normal, escalada por el factor

$$\sqrt{\frac{k_B T}{M}}$$

de tal modo que el análisis dimensional permita tener las unidades de velocidad. Esta velocidad es normalizada, utilizando la siguiente relación

$$V_{\text{norm}} = \frac{V - \mathbf{E}(VM)}{M} \tag{58}$$

Finalmente, se calcula la energía cinética del sistema. Toda la información calculada en esta primera función es guardada como un estado inicial, y es lo que devuelve la función mencionada.

```
      dt = f32(dt)
  if tau is None:
    tau = dt * 100
  tau = f32(tau)

  force_fn = quantity.canonicalize_force(energy_or_force_fn)
  chain_fns = nose_hoover_chain(dt, chain_length, chain_steps, sy_steps, tau)

  def init_fn(key, R, mass=f32(1.0), **kwargs):
    _kT = kT if 'kT' not in kwargs else kwargs['kT']

    mass = quantity.canonicalize_mass(mass)
    V = jnp.sqrt(_kT / mass) * random.normal(key, R.shape, dtype=R.dtype)
    V = V - jnp.mean(V * mass, axis=0, keepdims=True) / mass
    KE = quantity.kinetic_energy(V, mass)

    return NVTNoseHooverState(R, V, force_fn(R, **kwargs), mass,
                              chain_fns.initialize(R.size, KE, _kT))

```

3. La siguiente función aplica la función de medio paso que se definió previamente cuando se inicializaron las funciones de la cadena. Para ello, se aplica una vez la mencionada función, se actualiza el valor de la energía cinética, para después aplicar otra vez la función de medio paso, actualizando para este momento la velocidad con el algoritmo de Verlet. Con estos pasos, se tiene un nuevo estado del sistema con su respectiva velocidad calculada. Se muestra a continuación el código de esta sección

```
      def apply_fn(state, **kwargs):
    _kT = kT if 'kT' not in kwargs else kwargs['kT']

    chain = state.chain

    chain = chain_fns.update_mass(chain, _kT)

    v, chain = chain_fns.half_step(state.velocity, chain, _kT)
    state = dataclasses.replace(state, velocity=v)

    state = velocity_verlet(force_fn, shift_fn, dt, state, **kwargs)

    KE = quantity.kinetic_energy(state.velocity, state.mass)
    chain = dataclasses.replace(chain, kinetic_energy=KE)

    v, chain = chain_fns.half_step(state.velocity, chain, _kT)
    state = dataclasses.replace(state, velocity=v, chain=chain)
```

```
18
19      return state
20
```

Para finalizar con esta sección de Dinámica Molecular, se llama puntualmente estas funciones explicadas anteriormente, utilizando los parámetros de entrada que se muestran a continuación en el código. Se compilan dichos argumentos en una nueva función, dejando libre la energía que se calculará dinámicamente por medio del potencial planteado en esta tesis.

```
1     simulator_template = partial(simulate.nvt_nose_hoover, shift_fn=shift, dt=time_step,
      kT=kbT,
2                                  chain_length=3, chain_steps=1)
3     init, _ = simulator_template(energy_fn_init)
4     state = init(simuation_init_key, R_init, mass=mass, neighbor=nbrs_init)
5     init_sim_state = (state, nbrs_init)
```

4.5. Generación de Trayectorias por medio del algoritmo DiffTRe

La siguiente parte del trabajo consiste en desarrollar el algoritmo central del trabajo, el cual fue desarrollado por Stephan Taler y la profesora Julija Zavadla [15]. Se explica a continuación el código concerniente a DiffTRe.

Lo primero que hay que hacer es identificar las entradas de la función que dará inicio a este procedimiento. Los argumentos de entrada son:

- **simulation funs**: Este argumento debe ser una tupla, que contiene tres elementos: la función molde que ejecuta la dinámica molecular de las cadenas de termostatos; la función molde de la energía, la cual incluye el potencial de Lennard-Jones y la red de grafos; finalmente, la función que define los vecinos de un elemento del sistema.

- **timing struct**: Esta estructura contiene los tiempos que se considerarán para llevar a cabo la simulación. En el caso concreto de la simulación, se tomaron 10 picosegundos de tiempo de equilibrio, y 60 picosegundos de producción de estados después de este tiempo de equilibrio. Luego, cada 0.1 picosegundos se van guardando los estados, de modo que se tengan estados no correlacionados. Finalmente, se tomó un paso de tiempo de 0.002 picosegundos. Luego, para crear este arreglo, quedaría de la forma

$$\text{timing struc} = (600, 100, 50)$$

- k_BT= la temperatura en unidades de k_BT.

- init params=Son los parámetros de inicio de la función molde de la energía.

- reference state= Es una tupla que contiene dos elementos: el estado inicial que contiene la información de las posiciones iniciales de los elementos del sistema a estudiar, mientras que el segundo elemento contiene la información de los vecinos de cada uno de los elementos mencionados previamente.

- **optimizer**= El optimizador a utilizar para mejorar los pesos de la red neuronal conforme se hagan las épocas.
- loss fn= Es la función de costo a introducir en el algoritmo. Si no se define, hay una función que la calcula.
- reweight ratio= Es la tasa de referencia con la cual se decide si se va a volver a calcular una trayectoria o no. Se toma para este valor 0.90.

Teniendo especificadas las entradas, se procede a explicar el desarrollo del código. Primero, se verifica que haya una función de costo definida para realizar el entrenamiento de la red; si no se definió, se procede a crear una, basada en las cantidades de interés a predecir, cuya definición sigue la forma general presentada en la ecuación 5.

A continuación sigue identificar los diferentes moldes que se empaquetan en la tupla **simulations fns**, con la finalidad de manejar más fácilmente estas funciones a lo largo del código. Luego, viene un procedimiento extenso, que consiste en crear una función que genere la trayectoria del sistema para un cierto tiempo dado. Se explica a continuación cómo obtener dicha función:

1. Antes de comenzar con el desarrollo, se deben identificar los parámetros de entrada de esta función. Se deben tomar los tres moldes de la simulación, así como la tupla con los tiempos de la simulación.

2. El siguiente paso en definir una función que evolucione el sistema y se rescate el estado final de dicha evolución, para un cierto periodo que se escoge desde la definición de la tupla de estructura de tiempo. En el caso de este trabajo, se realizarán 50 pasos de tiempo, para obtener un estado y guardarlo para la definición de la trayectoria.

3. Con esa función, se procede a aplicar dicha función de evolución 100 veces, que corresponde al tiempo de equilibrio que se definió previo a considerar generar la trayectoria. En este paso, solo se guarda el último paso generado después de los 10 picosegundos de equilibrio. Aquí cabe recalcar que no se evoluciona el sistema únicamente 100 veces, sino que dicho sistema evoluciona 100·50 veces, donde recordemos que la función de evolución realiza 50 pasos de tiempo, correspondientes a cubrir 0.1 nanosegundos en pasos de 0.002 ps; con ello, los 100 restantes marcados en el producto anteriormente señalado son los 10 picosegundos de equilibrio previo a considerar el primer estado del sistema, tomando estados cada 0.1 picosegundos un estado del sistema. Tomar 10 picosegundos de equilibrio se sigue de la sugerencia hecha en [71].

4. Con este último estado generado después de los 10 picosegundos de equilibrio, se procede a tomar dicho estado como el inicial, para volver a aplicar la función de evolución del sistema 600 veces, que corresponde a los 60 nanosegundos marcados en la definición de la estructura de tiempo. En este punto se guardan como resultado de este procedimiento el estado final y la trayectoria generada, la cual se obtiene fácilmente gracias a la función scan de JAX.

5. Con esta trayectoria obtenida, se calcula para todos los estados de la trayectoria su respectiva energía potencial. Para este caso, también se usa la función scan, pero no se itera sobre un índice mudo, sino que se opera sobre los distintos estados de la trayectoria para calcular a cada uno de ellos su energía potencial y se guarda el resultado acumulado en otro arreglo.

6. Las salidas de esta función serán tres elementos: El último estado de la simulación, la trayectoria nueva calculada y su respectivo arreglo con las energías calculadas.

```
    def trajectory_generator_init(sim_funs, timings_struct):

    num_printouts_production, num_dumped, timesteps_per_printout =
    dataclasses.astuple(timings_struct)
    simulator_template, energy_fn_template, neighbor_fn = sim_funs

    def generate_reference_trajectory(params, sim_state):
        energy_fn = energy_fn_template(params)
        _, apply_fn = simulator_template(energy_fn)
        run_small_simulation = custom_simulator.run_to_next_printout_fn_neighbors(apply_fn,
    neighbor_fn,

    timesteps_per_printout)

```

```
        sim_state, _ = lax.scan(run_small_simulation, sim_state, xs=jnp.arange(num_dumped))  # equilibrate
        new_sim_state, traj = lax.scan(run_small_simulation, sim_state, xs=jnp.arange(num_printouts_production))
        final_state, nbrs = new_sim_state


        U_traj = compute_energy_trajectory(traj, nbrs, neighbor_fn, energy_fn)
        return new_sim_state, traj, U_traj

    return generate_reference_trajectory
```

Código 15: Función que calcula una trayectoria a partir de un estado inicial y su correspondiente arreglo de energías de cada estado de la misma

El siguiente paso es obtener una función que defina los pesos señalados en el marco teórico, los cuales defines si es necesario recalcular la trayectoria o no. Para ello, se procede a explicar por pasos la manera de programas esta función:

- Dada la función definida anteriormente, se explicó que dicha función tiene tres salidas: el estado final de la trayectoria, la trayectoria completa y la energía potencial en cada estado. Se identifican cada uno de estos elementos, así como la lista de vecinos, ubicada en la variable que describe el estado final.

- A continuación, se inicializa la función de cálculo de la energía, usando los parámetros iniciales como entrada, así como utilizando la función checkpoint de JAX, la cual permite utilizar eficientemente la memoria para poder realizar el cálculo de la regla de la cadena cuando se necesiten tomar derivadas. Este es un paso muy importante que hace característico a este framework de Inteligencia Artificial, pues calcular las derivadas y almacenarlas en memoria es un proceso que consume mucha memoria de la computadora y puede hacer que el código aborte la ejecución del programa por este punto.

- Luego, se procede a realizar el cálculo de una nueva trayectoria de energías potenciales, dado los parámetros que inicializaron la función de energía potencial anteriormente. Para el caso de la inicialización, se vuelve a obtener el mismo arreglo de energías, pero esto no sucederá cuando se tengan nuevos parámetros optimizados en el entrenamiento del modelo.

- Con estos arreglos de energías, se procede a calcular los pesos que se mencionan en el Marco Teórico y que siguen una distribución de Boltzmann.

- Finalmente, con estos pesos se procede a calcular la tasa efectiva de muestreo de esta trayectoria generada. La función final devuelve tanto los pesos como la tasa mencionada.

```
    def estimate_effective_samples(weights):
        weights = jnp.where(weights > 1.e-10, weights, 1.e-10)  # mask to avoid NaN from log(0) if a few weights are 0.
        exponent = - jnp.sum(weights * jnp.log(weights))
        return jnp.exp(exponent)

    def compute_weights(params, traj_state):
        sim_state, trajectory, U_traj = traj_state
        _, nbrs = sim_state

        energy_fn = checkpoint(energy_fn_template(params))  # whole backward pass too memory consuming
        U_traj_new = compute_energy_trajectory(trajectory, nbrs, neighbor_fn, energy_fn)

        # Difference in pot. Energy is difference in total energy as kinetic energy is the same and cancels
```

```
        exponent = -(1. / kbT) * (U_traj_new - U_traj)
        # Note: The reweighting scheme is a softmax, where the exponent above represents the logits.
        #       To improve numerical stability and guard against overflow it is good practice to subtract
        #       the max of the exponent using the identity softmax(x + c) = softmax(x). With all values
        #       in the exponent <=0, this rules out overflow and the 0 value guarantees a denominator >=1.
        exponent -= jnp.max(exponent)
        prob_ratios = jnp.exp(exponent)
        weights = prob_ratios / util.high_precision_sum(prob_ratios)
        n_eff = estimate_effective_samples(weights)
        return weights, n_eff

    return compute_weights
```

La tercera función importante a definir consiste en definir la manera de realizar la optimización de parámetros y calcular el gradiente del costo. Para este proceso se toman en cuenta las dos funciones trabajadas en los pasos descritos previamente. Ahora, se muestran los pasos a considerar para desarrollar esta función de optimización:

1. Esta función estará compuesta de tres subfunciones, con las cuales se busca describir mejor los detalles de este paso. Comenzado con la primera función, se enfoca en calcular el costo a partir de los pesos que devuelve la función descrita en la lista de pasos anteriores. Dados los parámetros y la trayectoria, se llama a la función de cálculo de pesos, con lo que se obtienen dichos pesos y la tasa efectiva, que para esta subrutina no será necesario conservarla en memoria. Luego, se calculan las observables de interés para cada estado de la trayectoria, y finalmente se calcula el costo, siendo el valor de toda la trayectoria la suma pesada de los diferentes valores obtenidos, utilizando para ello los pesos mencionados previamente. Se anexan los códigos que realizan estos pasos.

```
    def compute_quantity_traj(traj_state, quantities, neighbor_fn, energy_params=None):
        sim_state, trajectory, _ = traj_state
    _, fixed_reference_nbrs = sim_state

        @jit
        def quantity_trajectory(dummy_carry, state):
            R = state.position
            nbrs = neighbor_fn(R, fixed_reference_nbrs)
            computed_quantities = {quantity_fn_key:
    quantities[quantity_fn_key]['compute_fn'](
                state, neighbor=nbrs, energy_params=energy_params) for quantity_fn_key
    in quantities}
            return dummy_carry, computed_quantities

    _, quantity_trajs = lax.scan(quantity_trajectory, 0., trajectory)
    return quantity_trajs

```

Código 16: función para calcular las observables de interés para cada estado de la trayectoria

```
        def loss_fn(quantity_trajs, weights):
        loss = 0.
        predictions = {}
        for quantity_key in quantities:
            quantity_snapshots = quantity_trajs[quantity_key]
            weighted_snapshots = (quantity_snapshots.T * weights).T
            ensemble_average = util.high_precision_sum(weighted_snapshots, axis=0)  #
    weights account for "averaging"
            predictions[quantity_key] = ensemble_average
```

```
            loss += quantities[quantity_key]['gamma'] * mse_loss(ensemble_average,
    quantities[quantity_key]['target'])
        return loss, predictions
    return loss_fn

```

Código 17: Función de costo donde se observa la obtención de la onservable promedio a través de los pesos generados con la distribución de Boltzmann

```
        def reweighting_loss(params, traj_state):

        weights, _ = compute_weights(params, traj_state)
        quantity_trajs = compute_quantity_traj(traj_state, quantities, neighbor_fn,
    params)
        loss, predictions = loss_fn(quantity_trajs, weights)
        return loss, predictions

```

Código 18: Definición de la subrutina descrita en los pasos anteriores

2. La siguiente subrutina es bastante sencilla y consiste en definir la función identidad, con la cual se busca seguir la ideología de JAX de trabajar siempre con funciones, y dicha función se utilizará cuando no sea necesario recalcular la trayectoria.

```
        def trajectory_identity_mapping(input):

        _, traj_state = input
        error_code = 0
        return traj_state, error_code

```

Código 19: Función identidad para mantener la trayectoria actual

3. La siguiente subrutina sí recalculará la trayectoria en caso de ser necesario. Para esta situación, se toma el último estado de la trayectoria anterior, y a partir de él se desarrolla una nueva trayectoria.

```
        def recompute_trajectory(input):

        params, traj_state = input
        sim_state = traj_state[0]
        updated_traj_state = trajectory_generation_fn(params, sim_state)
        _, nbrs = updated_traj_state[0]
        error_code = lax.cond(nbrs.did_buffer_overflow, lambda _: 1, lambda _: 0, None)
        return updated_traj_state, error_code

```

4. Con estas subrutinas definidas, se puede conjuntar para tener la función de propagación deseada. Se calculan los pesos y la tasa efectiva de muestreo. Luego, se pregunta la condición si dicha tasa efectiva es mayor al 90 por ciento de los estados totales del sistema. Si no se cumple, se recalcula la trayectoria. Con esta nueva trayectoria y los parámetros de la red, se calcula el costo y las predicciones para las observables usando la primera subrutina explicada en el paso 1. Las salidas de esta función serán la nueva o la misma trayectoria, el gradiente, el costo, un error que indica si en el cálculo de una nueva trayectoria se necesitó recalcular la lista de vecinos y las predicciones de las observables.

```
        @jit
    def propagation_fn(params, traj_state):
        # check if trajectory re-use is possible; otherwise recompute trajectory
        weights, n_eff = compute_weights(params, traj_state)
        n_snapshots = traj_state[2].size
```

```
        recompute = n_eff < reweight_ratio * n_snapshots
        traj_state, error_code = lax.cond(recompute, recompute_trajectory,
    trajectory_identity_mapping,
                                          (params, traj_state))

        outputs, curr_grad = value_and_grad(reweighting_loss, has_aux=True)(params,
    traj_state)
        loss_val, predictions = outputs
        return traj_state, curr_grad, loss_val, error_code, predictions

    return propagation_fn

```

Usando estas tres funciones principales, se procede a actualizar los parámetros de la red neuronal de grafos de la siguiente manera: se identifican todas las salidas de la función anterior de propagación y cálculo de gradientes. Luego, si hubo una saturación en memoria y se tuvieron que calcular nuevos vecinos, entonces no se puede utilizar el gradiente para optimizar los parámetros; se deben quedar los mismos y se debe crear la nueva lista de vecinos. Esto es importante, porque la red neuronal de grafos está creada con base en los vecinos que se consideran en sus vecindades. Incluso, en pasos anteriores, se habló de dejar una tolerancia para poder incluir dinámicamente una cierta cantidad extra de posibles vecinos. Si no sucede nada de este estilo, entonces se utiliza un optimizador para actualizar los parámetros de la red.

```
    def update(step, params, opt_state, traj_state):
        new_traj_state, curr_grad, loss_val, error_code, predictions =
    propagation_fn(params, traj_state)

        if error_code == 1:  # neighbor list buffer overflowed: Enlarge neighbor list and
    rerun with old traj_state
            warnings.warn('Neighborlist buffer overflowed at step ' + str(step) + '.
    Initializing larger neighborlist.')
            new_state = new_traj_state[0][0]
            enlarged_nbrs = neighbor_fn(new_state.position)
            reset_traj_state = ((traj_state[0][0], enlarged_nbrs), traj_state[1],
    traj_state[2])
            new_traj_state = reset_traj_state
            new_params = params
        else:  # only use gradient if neighbor list did not overflowed
            scaled_grad, opt_state = optimizer.update(curr_grad, opt_state, params)
            new_params = optax.apply_updates(params, scaled_grad)
        return new_params, opt_state, new_traj_state, loss_val, predictions
```

Para finalizar con la inicialización de la trayectoria, se muestra todo el código compactado de la inicialización del algoritmo DiffTRe. Las funciones que se definieron anteriormente permiten actualizar para cada época los parámetros de la energía, pero hace faltar generar primero una trayectoria inicial, y a partir de ella considerar las posibilidades que se discutieron con detenimiento en el procedimiento anterior. Esta inicialización debe devolver la función de actualización y la trayectoria inicial.

```
    def DiffTRe_init(simulation_fns, timings_struct, quantities, kbT, init_params,
    reference_state, optimizer,
                 loss_fn=None, reweight_ratio=0.9):
    if loss_fn is None:
        loss_fn = init_independent_mse_loss_fn(quantities)
    else:
        print('Using custom loss function. Ignoring \'gamma\' and \'target\' in
    \"quantities\".')

    _, energy_fn_template, neighbor_fn = simulation_fns
    trajectory_generation_fn = trajectory_generator_init(simulation_fns, timings_struct)
```

```
    compute_weights = weight_computation_init(energy_fn_template, neighbor_fn, kbT)
    propagation_fn = gradient_and_propagation_init(compute_weights,
    trajectory_generation_fn, loss_fn, quantities,
                                                   neighbor_fn,
    reweight_ratio=reweight_ratio)
    update_fn = update_init(propagation_fn, optimizer, neighbor_fn)

    t_start = time.time()
    traj_initstate = trajectory_generation_fn(init_params, reference_state)
    runtime = (time.time() - t_start) / 60.
    print('Time for a single trajectory generation:', runtime, 'mins')

    return update_fn, traj_initstate
```

Código 20: Algoritmo de DiffTRe compacto

4.5.1. Entrenamiento de la red de grafos con DiffTRe

Para el entrenamiento de la red, se utilizó la técnica de Búsqueda por Mallado [72], utilizando como control el producto cartesiano de todas las posibles combinaciones generadas por este método, y los resultados se guardan en un archivo CSV, donde se guardan los metadatos de cada entrenamiento realizado. Se utiliza como optimizador el algoritmo Adam [73], así como un escalador de tasas de aprendizajes con decaimiento exponencial [74]. Los hiperparámetros a considerar son

- El radio de corte.
- El tamaño de la incrustación que describe a los átomos.
- El número de bloques de interacción
- El número de épocas
- La tasa de aprendizaje

```
    hparams=OrderedDict(
    rcut=[0.5],
    embed=[32],
    interactions=[3],
    epochs=[300],
    lr=[0.01]
)

for run in RunBuilder().get_runs(hparams):
    m.begin_run()


    print('Entrando a la corrida ', str(m.run_count))
    box_nbrs = jnp.ones(3)
    neighbor_fn = partition.neighbor_list(displacement, box_nbrs, run.rcut,
    dr_threshold=0.05, capacity_multiplier=1.5,
                                          disable_cell_list=True)
    nbrs_init = neighbor_fn(R_init)

    prior_fn = custom_energy.generic_repulsion_neighborlist(displacement, sigma=0.3165,
    epsilon=1., exp=12,
                                                            initialize_neighbor_list=False)

    print('Inicializando energ a de DimeNet')

    init_fn, GNN_energy = custom_energy.DimeNetPP_neighborlist(displacement, R_init,
    nbrs_init, run.rcut, embed_size=run.embed, n_interaction_blocks=run.interactions)
```

```
    init_params = init_fn(model_init_key, R_init, neighbor=nbrs_init)

    def energy_fn_template(energy_params):
        gnn_energy = partial(GNN_energy, energy_params)
        def energy(R, neighbor, **dynamic_kwargs):
            return gnn_energy(R, neighbor=neighbor, **dynamic_kwargs) + prior_fn(R, neighbor=neighbor, **dynamic_kwargs)
        return jit(energy)

    # Loss funciton
    quantity_dict = {}
    if 'rdf' in target_dict:
        rdf_struct = target_dict['rdf']
        rdf_fn = custom_quantity.initialize_radial_distribution_fun(box_tensor, displacement, rdf_struct)
        rdf_dict = {'compute_fn': checkpoint(rdf_fn), 'target': rdf_struct.reference_rdf, 'gamma': 1.}
        quantity_dict['rdf'] = rdf_dict
    if 'adf' in target_dict:
        adf_struct = target_dict['adf']
        adf_fn = custom_quantity.initialize_angle_distribution_neighborlist(displacement, adf_struct,
                                                                            R_init=R_init, nbrs_init=nbrs_init)
        adf_target_dict = {'compute_fn': checkpoint(adf_fn), 'target': adf_struct.reference_adf, 'gamma': 1.}
        quantity_dict['adf'] = adf_target_dict
    if 'pressure' in target_dict:
        pressure_fn = custom_quantity.init_pressure(energy_fn_template, box_tensor)
        pressure_target_dict = {'compute_fn': checkpoint(pressure_fn), 'target': target_dict['pressure'], 'gamma': 1.e-7}
        quantity_dict['pressure'] = pressure_target_dict



    energy_fn_init = energy_fn_template(init_params)
    print('Inicializando simulaci n')
    simulator_template = partial(simulate.nvt_nose_hoover, shift_fn=shift, dt=time_step, kT=kbT,
                                 chain_length=3, chain_steps=1)
    init, _ = simulator_template(energy_fn_init)
    state = init(simuation_init_key, R_init, mass=mass, neighbor=nbrs_init)
    init_sim_state = (state, nbrs_init)

    num_updates = run.epochs
    initial_lr = run.lr
    lr_schedule = optax.exponential_decay(-initial_lr, 200, 0.01)
    optimizer = optax.chain(
        optax.scale_by_adam(0.1, 0.4),
        optax.scale_by_schedule(lr_schedule)

    )
    simulation_funs = (simulator_template, energy_fn_template, neighbor_fn)

    print('Generando primer trayectoria de referncia con DiffTRe')

    update_fn, trajectory_state = difftre.DiffTRe_init(simulation_funs, timings_struct, quantity_dict, kbT,
                                                       init_params, init_sim_state, optimizer)


    #training loop
```

```
78
79
80      loss_history, times_per_update, predicted_quantities = [], [], []
81
82      params = init_params
83      opt_state = optimizer.init(init_params)  # initialize optimizer state
84
85
86      print('Comenzando entrenamiento')
87
88      for step in range(num_updates):
89          m.begin_epoch()
90
91          m.track("rcut", run.rcut)
92          m.track("embedding_size", run.embed)
93          m.track("num_interaction_blocks", run.interactions)
94          m.track("learning_rate", run.lr)
95
96
97          m.add_result("r_cut",run.rcut)
98          m.add_result("embedding_size", run.embed)
99          m.add_result("num_interaction_blocks", run.interactions)
100         m.add_result("learning_rate", run.lr)
101         #start_time = time.time()
102         m.display_progress()
103         params, opt_state, trajectory_state, loss_val, predictions = update_fn(step,
    params, opt_state, trajectory_state)
104         loss_val.block_until_ready()
105         m.track('loss', loss_val)
106
107         #step_time = time.time() - start_time
108
109         #times_per_update.append(step_time)
110         loss_history.append(loss_val)
111         predicted_quantities.append(predictions)
112         #print("Step {} in {:0.2f} sec".format(step, step_time), 'Loss = ', loss_val, '\n')
113         m.add_result('loss', loss_val)
114
115         m.end_epoch()
116         m.display_run_results()
117
118         if jnp.isnan(loss_val):  # stop learning when optimization diverged
119             print('Loss is NaN. This was likely caused by divergence of the optimization or
    a bad model setup '
120                 'causing a NaN trajectory.')
121             break
122     m.end_run()
123     m.save('sprint3/'+str(m.run_count))
```

5. Resultados

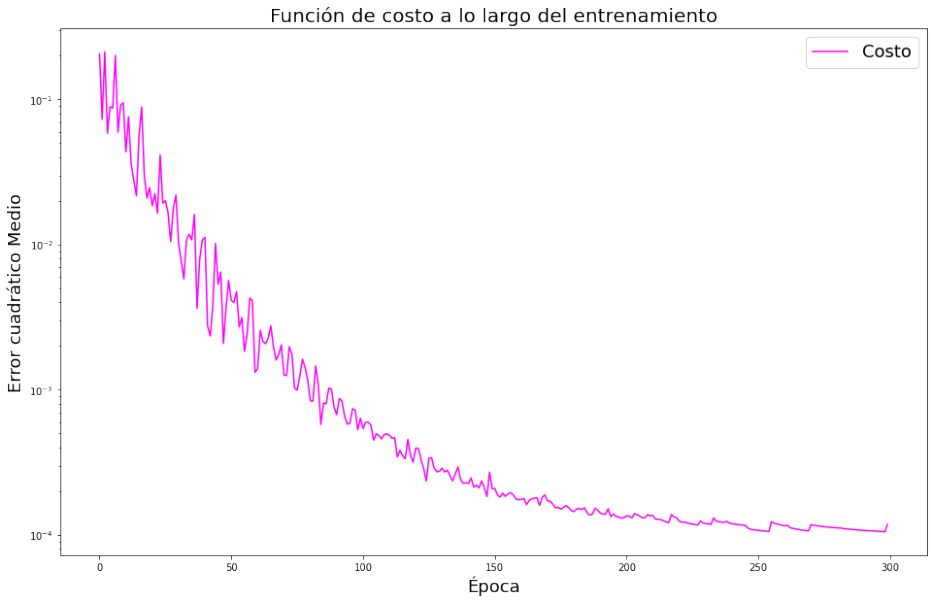

Figura 15: Error cuadrático medio a lo largo del entrenamiento de la red DimeNet++. Al término del entrenamiento, el error obtenido fue de 0.00011822

El primer punto a justificar es la elección de los hiperparámetros del modelo. En particular, del número de bloques de interacción. Para ello, se utilizará la teoría efectiva de redes neuronales para encontrar el cociente de aspecto de la red definida. De la ecuación 36, se tiene que especificar el ancho y la profundidad de la red:

- El ancho de la red es de 64, que corresponde al número máximo de neuronas en una capa. Este valor se obtiene en la capa densa al concatenar las representaciones iniciales de las moléculas con las 32 neuronas de la distancia codificada inicialmente con 6 funciones radiales de Bessel.
- La profundidad de la red es el número total de capas utilizadas, que corresponde a 94.

Introduciendo estos valores en el cociente de aspecto, se tiene

$$r = \frac{94}{64} = 1.468$$

Dicho valor no es lejano a la unidad y es mucho mayor a cero, por lo que no se pierde complejidad ni se agrega profundidad demás que impida explicar el comportamiento de la red. La Gráfica 15 muestra que la red está aprendiendo y se equivoca menos en sus predicciones en los tres rubros de observables macroscópicas a evaluar.

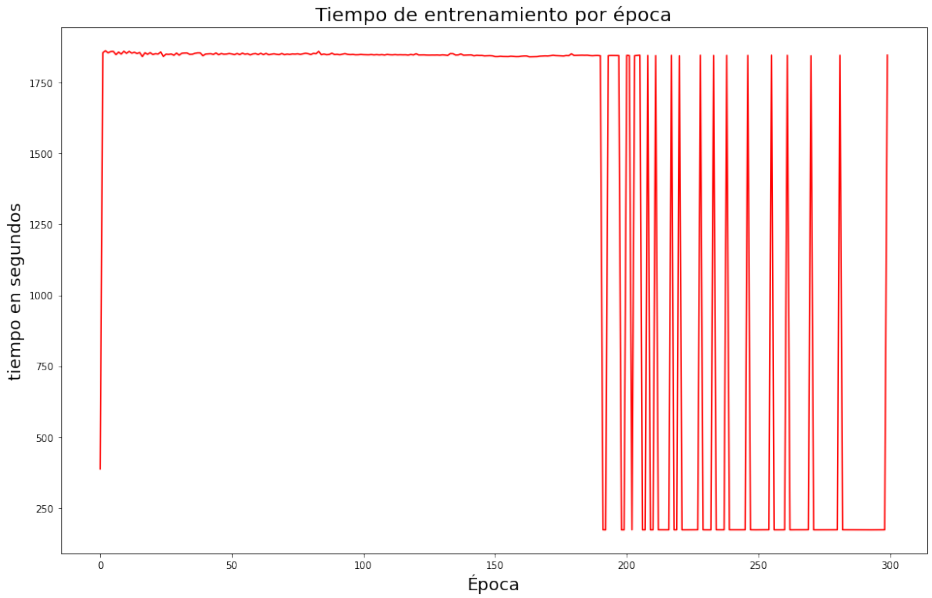

Figura 16: Tiempo transcurrido en cada una de las épocas del entrenamiento. La última época tuvo un tiempo de duración de 1846.1183 segundos, lo que indica que se volvió a calcular la trayectoria

Respecto al tema de los tiempos que se presentan en la Gráfica 16, para las primeras 200 épocas se recalculaba siempre la trayectoria. En términos del aprendizaje por refuerzo, el agente recibía del entorno una penalización por no cumplir la regla de correlación de estados generados y era necesario retroceder a calcular nuevos estados que conformarán la trayectoria. No obstante, pasando el umbral de las 200 épocas, el agente ya es capaz de generar una trayectoria que le permita seguir explorando el entorno adecuadamente para predecir el potencial del sistema por mayor tiempo. La creación o no de una trayectoria se nota por los tiempos de ejecución. En promedio, el entrenamiento con la generación de la trayectoria dura 30 minutos, mientras que el entrenamiento manteniendo una misma trayectoria dura 3 minutos aproximadamente.

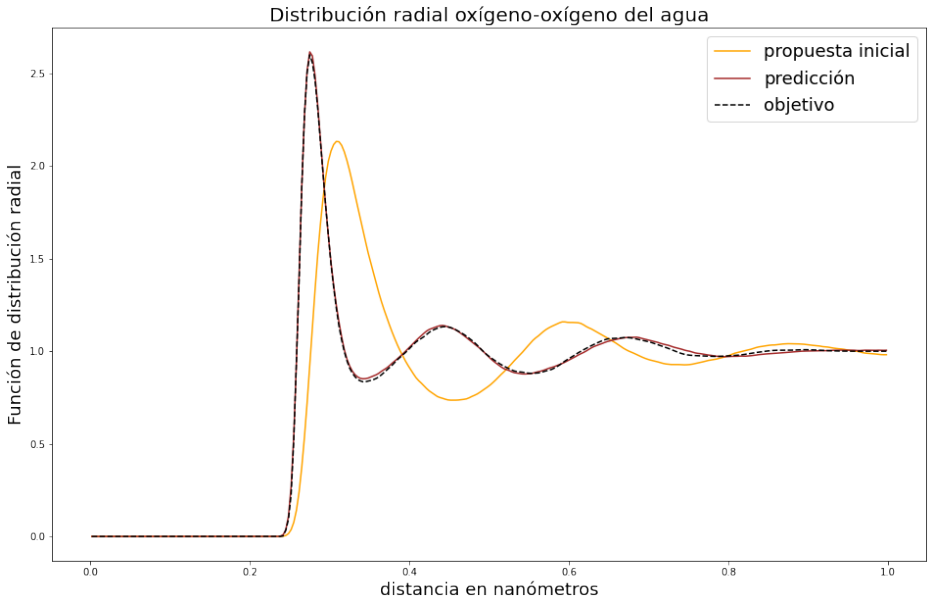

Figura 17: Distribución radial antes del entrenamiento y después del mismo, comparados con la distribución de referencia. La predicción final se alinea con la distribución objetivo

Analizando ahora las predicciones finales de la red, nótese en la Gráfica 17 que la física del potencial de

Lennard-Jones es tal que la forma de la distribución es correcta desde la primera iteración, sin ayuda de la red de grafos. El trabajo de DimeNet ++ consiste en ajustar dicha forma a la complejidad necesaria para describir al sistema de moléculas de agua. La predicción muestra que alrededor de los 3 Ángstroms se encuentra la primera vecindad de vecinos, siendo la más numerosa. La segunda y tercera vecindad de vecinos aparecen a los 5 y 7 Ángstroms aproximadamente. Los comportamientos asintóticos mencionados en la parte teórica se cumplen en la predicción final. Existen pequeñas variaciones en el primer y tercer valle de la distribución, respecto al valor de referencia ilustrado con la línea punteada.

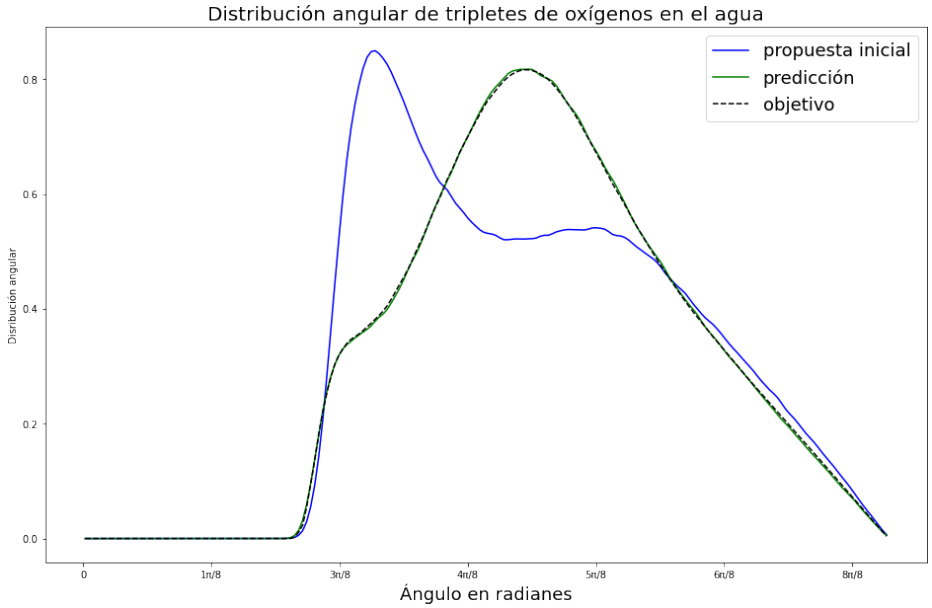

Figura 18: Distribución angular antes del entrenamiento y después del mismo, comparados con la distribución de referencia. La predicción inicial dista mucho del objetivo, lo que muestra la eficacia del entrenamiento.

Para la distribución angular de la Gráfica 18 se muestra a continuación la gráfica del trabajo citado en [62] que realizan un estudio similar pero con cálculos cuánticos:

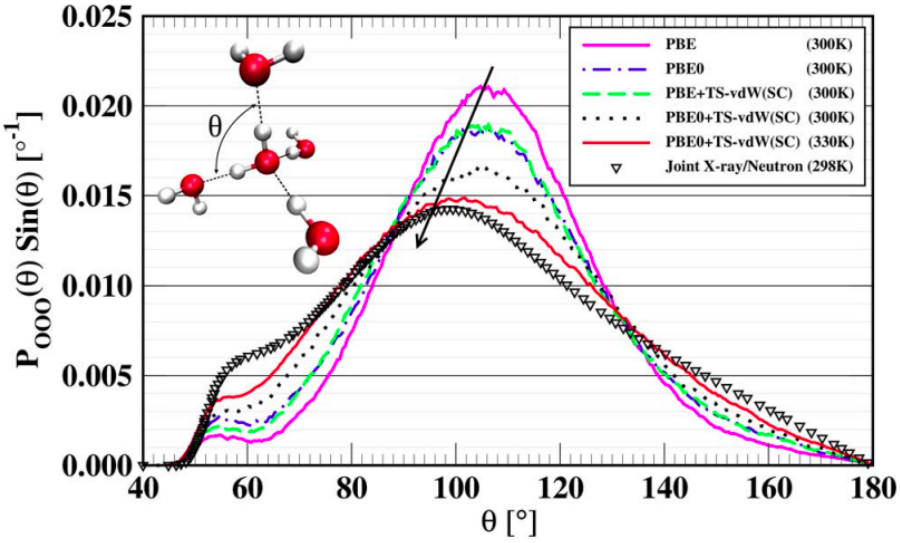

Figura 19: Gráfica tomada del trabajo de [62], en donde tomando diferentes funcionales para los cálculos cuánticos se comparan las predicciones con el experimento.

En la Gráfica 19 se puede ver que con la metodología por construcción no se puede obtener la forma correcta de las colas de la distribución angular, mientras que con la metodología por definición realizada en esta tesis se obtiene con buena precisión la distribución mencionada. En esta ocasión la física codificada en el término de Lennard-Jones no es suficiente para aproximar a la distribución. Fue la red de grafos quien corrigió por completo la forma inicial de la distribución, lo cual también refleja que el entrenamiento impulsó a añadir la suficiente complejidad al potencial para obtener la distribución correcta.

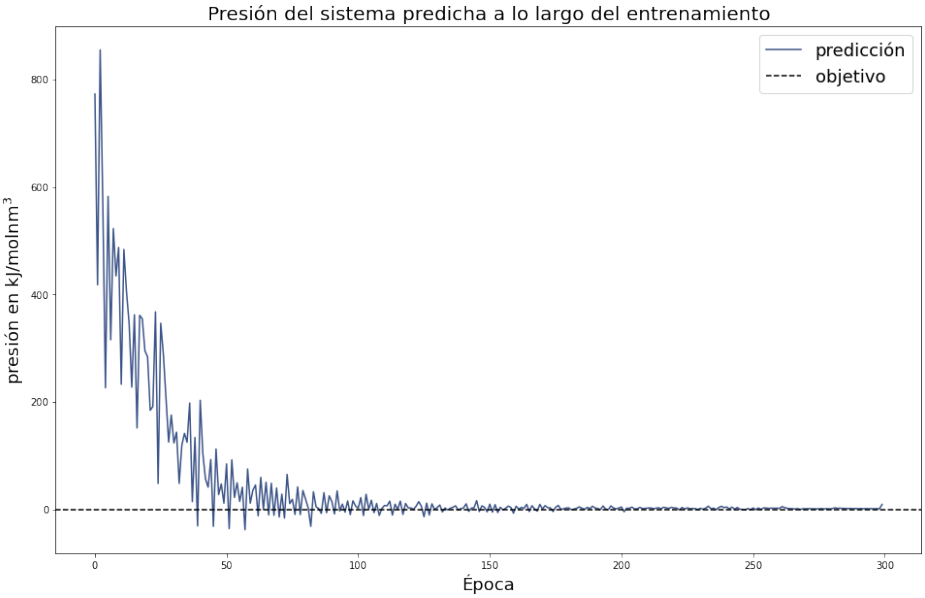

Figura 20: Predicción de la presión del sistema a lo largo del entrenamiento. El resultado final para la presión es de 9.259386 $\frac{\text{kJ}}{\text{mol nm}^3}$.

Finalmente, para analizar las presiones predichas en la Gráfica 21, se aprecia que la presión inicial se alejaba demasiado del valor de referencia de 1 bar, y también el entrenamiento del grafo fue determinante para lograr disminuir el error de las predicciones. No obstante, si se analiza con cuidado las últimas 20 épocas del entrenamiento, la presión predicha diverge del valor de referencia, como se puede apreciar en la siguiente Gráfica:

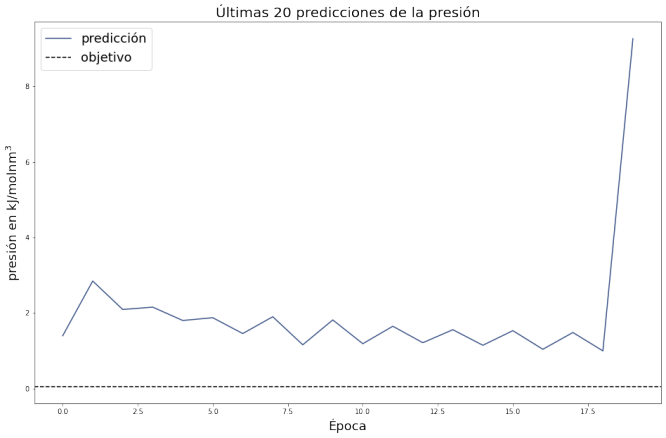

Figura 21: Últimas 20 predicciones hechas a partir del potencial inferido por el algoritmo DiffTRe. El resultado final se encuentra a dos órdenes de magnitud de distancia del valor real de la presión.

6. Conclusiones y trabajo futuro

Las metodologías por construcción, o mejor conocidas en inglés como bottom-top, utilizan cálculos a priori para realizar el ajuste de los parámetros de sus modelos y que las predicciones de los potenciales de interacción vayan acorde con los cálculos mencionados, lo cual implica una limitación en la precisión de los resultados. Por otro lado, la mayor parte de los trabajos que utilizan las metodologías por definición, o metodología top-down, utilizan los resultados experimentales como valores de referencia, de tal modo que la propuesta de potencial de interacción de un determinado sistema sea capaz de poder reproducir dicha propiedad del sistema, teniendo como principal desventaja que son pocos los parámetros a ajustar para obtener la complejidad necesaria que el experimento demanda.

El presente estudio ha buscado realizar una aportación a las metodologías por definición, usando como propuesta de potencial un término que contenga la física elemental del sistema a estudiar, así como una red neuronal de grafos que complemente para obtener la complejidad necesaria para reproducir las observables de los sistemas de estudio. Se trabajó el caso particular de un sistema con 901 moléculas de agua vistas granularmente.

Partiendo de una distribución atómica inicial de dicho sistema, se presentó el algoritmo DiffTRe con el cual, basándose en el Aprendizaje por Refuerzo, se generó iterativamente trayectorias que permitieran representar al sistema como un grafo y así incluir la información de la distancia y los ángulos en la propuesta final del potencial.

Analizando las predicciones finales con las observables de referencia, se puede concluir que las aportaciones hechas por pares de elementos, así como las aportaciones de los tripletes, al potencial final, permiten describir con mayor precisión la interacción global del sistema. Dichas aportaciones fueron posibles gracias a la red neuronal DiffTRe, cuya construcción se justificó con la teoría efectiva de redes neuronales realizada por físicos teóricos.

Existe todavía oportunidad de mejora para tener una mayor precisión en la predicción de la presión del sistema. Para ello, se puede utilizar el mismo código y realizar un entrenamiento con más épocas. De igual forma, se podría añadir mayor complejidad a la red DimeNet++, cuidando el cociente de aspecto y los escenarios en donde se pierde información o se añade complejidad demás.

Finalmente, una hipótesis importante para el éxito del modelo de potencial era que el primer término contuviera la física suficiente para estudiar al sistema de interés. Actualmente, existe una corriente de soluciones a ecuaciones que aparecen en problemas de Física llamada Redes Neuronales con Información Física", como se puede ver en el siguiente trabajo alrededor de los problemas de transferencia de calor [75], en donde las redes neuronales resuelven en su mayoría ecuaciones diferenciales parciales. El trabajo realizado en esta tesis se asemeja mucho a dicha corriente y con las nuevas teorías detrás para explicar el comportamiento de las redes neuronales, como es la teoría efectiva utilizada en este proyecto, el campo de investigación de aplicaciones física con redes neuronales seguirá en aumento en los próximos años.

7. Apéndice

7.1. Mecánica Estadística

Con la finalidad de que el trabajo presentado sea autocontenido, se presentan los conceptos básicos alrededor de la Mecánica Estadísticas, los cuales se utilizan a lo largo de las diferentes secciones desarrolladas en este proyecto de investigación.

7.1.1. Observables y promedios del ensamble

La manera de estudiar un sistema de partículas en Dinámica Molecular es a través de sus posiciones y velocidades. El espacio fase toma en cuenta estos rasgos, originando un espacio de 6N dimensiones, siendo N el número de partículas presentes en el sistema. Luego, se denotará por Γ un punto en este espacio fase descrito; así mismo, se denotará por $A(\Gamma)$ al valor instantáneo de alguna propiedad del sistema.

Cuando el sistema evoluciona, entonces Γ y $A(\Gamma)$ también. Entonces, se debe suponer que la variable macroscópica A_{obs} es el promedio de $A(\Gamma)$ en un intervalo de tiempo suficientemente grande. Esto se puede escribir como

$$A_{\text{obs}} = \langle A(\Gamma(t)) \rangle = \lim_{t\to\infty} \frac{1}{t} \int_0^t A(\Gamma(t))\, dt \tag{59}$$

En la práctica, se debe escoger un tiempo discreto τ que sea lo suficientemente grande, de tal manera que se pueda obtener la observable deseada en un tiempo discreto de tiempo

$$A_{\text{obs}} = \frac{1}{\tau} \sum_{t=1}^{\tau} A(\Gamma(t)) \tag{60}$$

Nótese que ahora t es una variable discreta, y el paso con el que recorreremos el tiempo para la simulación molecular dependerá de la decisión del usuario, así como de las condiciones que el problema requiera. La manera de escoger τ, sin embargo, no son claras. Dicho tiempo debe ser suficiente para poder recorrer gran parte del espacio fase en una sola trayectoria y así obtener un promedio que sea acorde a la observable que se desea obtener.

Para darle la vuelta al problema, Gibbs sugiere [76] que se reemplace el promedio temporal por el promedio del ensamble, donde el ensamble es una colección de puntos Γ en el espacio fase. Dichos puntos se distribuyen siguiendo una densidad de probabilidad ρ, la cual está determinada por los parámetros macroscópicos del sistema, como la presión, el volumen y la temperatura.

Hay que recordar que cada punto del espacio fase representa a un determinado elemento del sistema a un cierto instante de tiempo. Cada elemento evoluciona independiente de los demás elementos presentes, por lo que la densidad en el espacio fase $\rho(\Gamma)$ cambiará igualmente en el tiempo.

No obstante, no se crean ni se destruyen elementos durante la evolución del sistema completo. Luego, utilizando el teorema de Liouville, que es una ley de conservación para la densidad de probabilidad, estipula que

$$\frac{d\rho}{dt} = 0 \tag{61}$$

donde el operador d/dt está dado por

$$\frac{d}{dt} = \frac{\partial}{\partial t} + \dot{\vec{r}} \cdot \nabla_r + \dot{\vec{p}} \cdot \nabla_p \tag{62}$$

Definiendo el operador de Liouville como

$$iL = \dot{\vec{r}} \cdot \nabla_r + \dot{\vec{p}} \cdot \nabla_p \tag{63}$$

Usando el teorema de Liouville [57],

$$\frac{\partial \rho(\Gamma, t)}{\partial t} = iL\rho(\Gamma, t) \tag{64}$$

Lo que dice la ecuación 64 es que la tasa de cambio de la densidad, a un punto fijo en el espacio fase, está relacionada con el flujo que entra y sale en dicho punto. Esta ecuación tiene una solución formal, dada por

$$\rho(\Gamma, t) = \exp(-iLt)\rho(\Gamma, 0) \tag{65}$$

Por otro lado, la ecuación de movimiento de una función que no depende explícitamente del tiempo, como $A(\Gamma)$, tiene una forma conjugada, dada por

$$\dot{A}(\Gamma) = iLA(\Gamma(t)) \tag{66}$$

con su correspondiente solución

$$A(\Gamma(t)) = \exp(iLt)A(\Gamma(0)) \tag{67}$$

Si $\rho(\Gamma)$ representa un ensamble en equilibrio, entonces su dependencia en el tiempo desaparece

$$\frac{\partial \rho}{\partial t} = 0 \tag{68}$$

A medida que un elemento deja un cierto estado $\Gamma(\tau)$ y se mueve al nuevo estado $\Gamma(\tau + 1)$, otro elemento llega del estado $\tau - 1$ para reemplazarlo. Aquellos sistemas que cumplen esta propiedad se conocen como sistemas ergódicos.

7.1.2. Función de pesos

Es posible escribir la densidad $\rho(\Gamma)$ a través de una función $\omega(\Gamma)$, que representa pesos de un cierto estado. Esta función satisface lo siguiente:

$$\rho(\Gamma) = \frac{1}{Z}\omega(\Gamma) \tag{69}$$

donde

$$Z = \sum_{\Gamma} \omega(\Gamma) \tag{70}$$

es la función de partición, que actúa como un factor de normalización de la función de pesos $\omega(\Gamma)$. Esta función también describe las propiedades macroscópicas que definen al ensamble, cuya conexión con la termodinámica clásica está dada por el potencial termodinámico

$$\Psi_{\text{ensamble}} = -\ln Z \tag{71}$$

A partir de esta descripción de la función de partición con los pesos definidos previamente, nace la idea del esquema de DiffTRe [15] que se trabaja en esta tesis.

7.1.3. Ensamble canónico

La densidad de probabilidad en el ensamble canónico está dado por la expresión

$$\rho_{NVT} = \exp\left(-\frac{H(\Gamma)}{k_B T}\right) \tag{72}$$

y su función de partición es de la forma

$$Z_{NVT} = \frac{1}{N!\,h^{3N}} \int d\vec{r}\,d\vec{p} \exp\left(-\frac{H(\vec{r}, \vec{p})}{k_B T}\right) \tag{73}$$

La principal característica de este ensamble es que las fluctuaciones de energía no son cero, en contraste con el ensamble microcanónico. Esta característica es lo que orilla a introducir el esquema de termostatos que se expone en la parte teórica de este trabajo [9]

7.1.4. Derivación de las ecuaciones de movimiento no Hamiltonianas

Un sistema en el ensamble NPT implica que se mantiene la temperatura T constante gracias al contacto del mencionado sistema con un reservorio de calor; de igual manera, la presión P se mantiene constante, permitiendo al sistema cambiar su volumen V. El reservorio es construido de tal forma que tenga un número grande de partículas, con lo cual se evitan grandes fluctuaciones térmicas debido al contacto con el sistema de partículas de interés, el cual está conformado de n partículas, con sus respectivas posiciones $\vec{r}_\alpha$, masas m_α y velocidades $\frac{d\vec{r}_\alpha}{dt}$.

El Lagrangiano de este sistema de partículas está dado por

$$\mathcal{L} = \sum_{\alpha=1}^{n} \frac{m_\alpha}{2}\left(\frac{d\vec{r}_\alpha}{dt}\right)^2 - U\left(\{\vec{r}_\alpha\}\right) \tag{74}$$

donde el primer término de la ecuación 74 representa la energía cinética y el segundo la energía potencial, la cual es una función de las posiciones y describe la interacción entre las n partículas.

Si se desea construir un sistema con presión constante, entonces se debe agregar una fuerza externa; del mismo modo, se debe agregar el intercambio de calor con el entorno para mantener la temperatura constante. Por estas razones, es necesario incluir en el Lagrangiano del sistema a las partículas del reservorio de calor.

Al tener estas consideraciones, cuando se estudian las ecuaciones de movimiento newtonianas

$$\vec{F}_\alpha = -\nabla_U U\left(\{\vec{r}_1, \vec{r}_2, ..., \vec{r}_n\}\right) \tag{75}$$

se tiene que la fuerza en la α-ésima partícula es el resultado de la interacción con las otras partículas del sistema, incluyendo ahora las del reservorio de calor. Aquí es donde resalta el problema de los cálculos computacionales, pues se tienen que agregar suficientes términos para describir todas las interacciones de todas las partículas mencionadas. La motivación ahora es encontrar una simplificación en el ensamble NPT para esta situación.

Una simplificación consiste en el potencial de fuerza media, el cual requiere conocerse la función de distribución reducida, $\rho^{(n)}$ de las partículas en el volumen V, para obtener la mencionada fuerza media. La función de distribución reducida está dada por

$$\rho^{(n)} = V^n \frac{\int_V e^{-\frac{U(\{\vec{r}_\alpha\})}{k_B T}} d^3 r_{n+1} \cdots d^3 r_m}{\int_V e^{-\frac{U(\{\vec{r}_\alpha\})}{k_B T}} d^3 r_1 \cdots d^3 r_m} \tag{76}$$

donde se hace la distinción entre las partículas del sistema de interés, con etiquetas del 1 al n, mientras que las del entorno se etiquetan a partir del n+1 hasta m. Con esta aclaración, se resalta el hecho de que no se aplica el promedio canónico sobre las partículas 1 a n, ya que su posición en el espacio está fija en la expresión 76.

El potencial termodinámico de las partículas del medio actuando sobre el sistema de partículas de interés está dado por

$$w^{(n)}(\vec{r}_1, ..., \vec{r}_n) = -k_b T \ln \rho^{(n)}(\vec{r}_1, ..., \vec{r}_n) \tag{77}$$

Con ello, se puede describir la fuerza que actúa sobre una partícula β del sistema, $\beta \in$ [1,...,N], tomando el gradiente del potencial termodinámico:

$$\nabla_\beta w^{(n)} = -k_b T \left(\rho^{(n)}\right)/\rho^{(n)} \tag{78}$$

Desarrollando la expresión anterior

$$\nabla_\beta w^{(n)} = -k_b T \left[\frac{1}{V^n} \frac{\int_V e^{-\frac{U(\{\vec{r}_\alpha\})}{k_B T}} d^3 r_1 \cdots d^3 r_m}{\int_V e^{-\frac{U(\{\vec{r}_\alpha\})}{k_B T}} d^3 r_{n+1} \cdots d^3 r_m}\right]\left(V^n \frac{\int_V \frac{-1}{k_b T} \nabla_\beta U e^{-\frac{U(\{\vec{r}_\alpha\})}{k_B T}} d^3 r_{n+1} \cdots d^3 r_m}{\int_V e^{-\frac{U(\{\vec{r}_\alpha\})}{k_B T}} d^3 r_1 \cdots d^3 r_m}\right)$$

Simplificando la expresión anterior

$$\nabla_\beta w^{(n)} = \frac{\int_V \left(\nabla_\beta U\right) e^{-\frac{U(\{\vec{r}_\alpha\})}{k_B T}} d^3 r_{n+1} \cdots d^3 r_m}{\int_V e^{-\frac{U(\{\vec{r}_\alpha\})}{k_B T}} d^3 r_{n+1} \cdots d^3 r_m} \tag{79}$$

Para ejemplificar la complejidad detrás de la expresión 79, se considerará interacciones entre pares de partículas. Así, si se toma la partícula 1 del sistema, la fuerza que actúa sobre ella se simplifica como

$$\nabla_1 w^{(n)} = \sum_{i=2}^{n} \nabla_1 u(r_{1i}) + \frac{\sum_{i=n+1}^{m} \int \nabla_1 u(r_{1i}) e^{-\frac{U}{k_b T}} d^3 r_{n+1} \cdots d^3 r_m}{\int e^{-\frac{U}{k_b T}} d^3 r_{n+1} \cdots d^3 r_m} \tag{80}$$

De la ecuación 80 se puede describir al primer término como las fuerzas sistemáticas en la partícula 1 debido a la interacción con las demás partículas confinadas. El segundo término es una fuerza estadística sobre la partícula uno debido a la interacción con el ambiente. Es este término el que presenta la mayor dificultad de ser evaluada, pues es necesario conocer las posiciones exactas de cada partícula para cada instante. Luego, hay que buscar una manera de reducir esta complejidad.

Cualesquiera que sea la propuesta para simplificar la complejidad de la expresión anterior, se debe agregar un barostato y un termostato para imponer la presión y la temperatura del sistema. Luego, se deben añadir dos grados de libertad adicionales, los cuales deben realizar los siguientes efectos en el sistema:

1. El grado de libertad asociado al barostato debe ser capaz de aumentar o diminuir la distancia entre las partículas, de modo que se alcance un volumen tal que se adecúe a la presión deseada.

2. El otro grado asociado al termostato debe modificar el valor de la rapidez de las partículas, para emular la temperatura del sistema impuesta.

Aclarado estos puntos, se denota por s y V los grados de libertad asociados al termostato y el barostato respectivamente. Así, el Lagrangiano del sistema queda escrito como:

$$\mathcal{L} = \sum_\alpha \frac{m_\alpha}{2} s^2 V^{2/3} \dot{q}_\alpha^2 + \frac{M_s}{2}\dot{s}^2 + \frac{M_V}{2}\dot{V}^2 - U\left(\left\{V^{1/3}\vec{q}_\alpha\right\}\right) - g k_b T \ln s - P_{ext} V \tag{81}$$

donde g es el número total de grados de libertad, siendo en total 3n+2, con el primer término asociado a las 3n coordenadas generalizadas q_α, y los dos grados restantes asociados a s, que es una variable adimensional que escala el tiempo, y V, que es el volumen de la celda de simulación. Las masas asociadas a estos nuevos grados de libertad deben cumplir la siguiente dimensionalidad:

$$[M_V] = \text{energía x tiempo}^2/\text{volumen}^2$$

$$[M_s] = \text{energía x tiempo}^2$$

Continuando con el análisis del Lagrangiano 81, los términos indicados en la suma juegan el papel de un campo externo. No obstante, los términos s y V pueden verse como variables que describen las posiciones de un par de partículas virtuales, las cuales serían las partículas del barostato y la del termostato. A la ecuación 81 se le conoce como Lagrangiano extendido.

El potencial que rige a la partícula del barostato es proporcional a PV, en concordancia con la suma pesada asociada a la función de partición canóninca Z_{NVT}, la cual se usa para integrar a la función de partición del ensemble isobárico-isotérmico Z_{NPT}

$$\int_0^\infty Z_{NVT} e^{-\frac{PV}{k_B T}} dV = \frac{1}{N! h^{3N}} \int_0^\infty e^{-[\mathcal{H}(N,V)+PV]/k_B T} d\Gamma = Z_{NPT} \tag{82}$$

Por otro lado, el potencial de la partícula del termostato se indica en el Lagrangiano extendido 81 de tal suerte que se remita a la forma del potencial termodinámica de la ecuación 79, salvo que ahora no se necesita conocer la función de distribución $\rho^{(n)}$.

Lo que se debe realizar a continuación es escribir las ecuaciones de movimientos, siguiendo la formulación Lagrangiana

$$\frac{d}{dt}\pi_x = \frac{d}{dt}\left(\frac{\partial \mathcal{L}}{d\dot{x}}\right) = \frac{\partial \mathcal{L}}{\partial x} \tag{83}$$

donde $x = q_\alpha, s, V$ y con $\alpha \in [1,...,n]$. Realizando los cálculos de cada elemento de la ecuación 83, se tiene entonces para las coordenadas generalizadas:

$$\pi_\alpha = \frac{\partial \mathcal{L}}{\partial \dot{q}_\alpha} = \frac{m_\alpha}{2} s^2 V^{2/3} 2\dot{q}_\alpha = m_\alpha s^2 V^{2/3} \dot{q}_\alpha \tag{84}$$

$$\frac{\partial \mathcal{L}}{\partial q_\alpha} = -V^{1/3} \frac{\partial U\left(\left\{V^{1/3}\vec{q}_\gamma\right\}\right)}{\partial U\left(\left\{V^{1/3}\vec{q}_\alpha\right\}\right)} \tag{85}$$

Tomando la derivada con respecto al tiempo de 84

$$\frac{d}{dt}\pi_\alpha = 2m_\alpha s\dot{s}V^{2/3}\dot{q}_\alpha + \frac{2}{3}m_\alpha s^2 \frac{\dot{V}}{V^{1/3}}\dot{q}_\alpha + m_\alpha s^2 V^{2/3}\ddot{q}_\alpha \tag{86}$$

Finalmente, igualando 86 con 85, tenemos la ecuación de movimiento para las coordenadas generalizadas:

$$2m_\alpha s\dot{s}V^{2/3}\dot{q}_\alpha + \frac{2}{3}m_\alpha s^2 \frac{\dot{V}}{V^{1/3}}\dot{q}_\alpha + m_\alpha s^2 V^{2/3}\ddot{q}_\alpha = -V^{1/3} \frac{\partial U\left(\left\{V^{1/3}\vec{q}_\gamma\right\}\right)}{\partial U\left(\left\{V^{1/3}\vec{q}_\alpha\right\}\right)} \tag{87}$$

Repitiendo el mismo procedimiento para la partícula del termostato

$$\pi_s = M_s\dot{s} \tag{88}$$

$$\frac{\partial \mathcal{L}}{\partial s} = \sum_\alpha m_\alpha s V^{2/3}\dot{q}_\alpha - g k_b T_{eq}/s \tag{89}$$

$$\frac{d}{dt}\pi_s = M_s\ddot{s} \tag{90}$$

Igualando 90 con 89, obtenemos la ecuación de movimiento para el termostato

$$M_s\ddot{s} = \sum_\alpha m_\alpha s V^{2/3}\dot{q}_\alpha - g k_B T_{eq}/s \tag{91}$$

Finalmente, para describir el movimiento del barostato, desarrollamos las siguientes ecuaciones

$$\pi_V = M_V\dot{V} \tag{92}$$

$$\frac{\partial \mathcal{L}}{\partial V} = \sum_\alpha \frac{m_\alpha}{3}\frac{s^2}{V^{13}}\dot{q}_\alpha^2 - \frac{1}{3}\sum_\alpha \frac{\partial U\left(\left\{V^{1/3}\vec{q}_\gamma\right\}\right)}{\partial\left(U\left(\left\{V^{1/3}\vec{q}_\alpha\right\}\right)\right)}\frac{q_\alpha}{V^{2/3}} - P_{ext} \tag{93}$$

$$\frac{d}{dt}\pi_V = M_V\ddot{V} \tag{94}$$

Igualando 94 con 93, tenemos la ecuación de movimiento para la partícula del barostato

$$M_V\ddot{V} = \sum_\alpha \frac{m_\alpha}{3}\frac{s^2}{V^{13}}\dot{q}_\alpha^2 - \frac{1}{3}\sum_\alpha \frac{\partial U\left(\left\{V^{1/3}\vec{q}_\gamma\right\}\right)}{\partial\left(U\left(\left\{V^{1/3}\vec{q}_\alpha\right\}\right)\right)}\frac{q_\alpha}{V^{2/3}} - P_{ext} \tag{95}$$

El siguiente paso es formular el Hamiltoniano del sistema a partir del Lagrangiano extendido por medio de una transformación de Legendre:

$$\mathcal{H} = \sum_\alpha \dot{q}_\alpha\pi_\alpha + \dot{s}\pi_s + \dot{V}\pi_V - \mathcal{L}\left(\{q_\alpha,\dot{q}_\alpha\}, s, \dot{s}, V, \dot{V}\right) \tag{96}$$

Sustituyendo 81 en 96

$$\mathcal{H} = \sum_{\alpha} \dot{q}_{\alpha}\pi_{\alpha} + \dot{s}\pi_{s} + \dot{V}\pi_{V} - \left[\sum_{\alpha} \frac{m_{\alpha}}{2} s^2 V^{2/3} \dot{q}_{\alpha}^2 + \frac{M_s}{2}\dot{s}^2 + \frac{M_V}{2}\dot{V}^2 - U\left(\left\{V^{1/3}\vec{q}_{\alpha}\right\}\right) - gk_b T \ln s - P_{ext} V\right] \tag{97}$$

Ahora, lo que hay que hacer es dejar la ecuación 97 en términos de los momentos conjugados. Para ello, se utilizarán las expresiones 84, 88 y 92 para expresar $\dot{q}_{\alpha}$, $\dot{s}$ y $\dot{V}$ en términos de los momentos conjugados:

$$\mathcal{H} = \sum_{\alpha} \frac{\pi_{\alpha}^2}{m_{\alpha} s^2 V^{2/3}} + \frac{\pi_s^2}{M_S} + \frac{\pi_v^2}{M_V} - \left[\sum_{\alpha} \frac{m_{\alpha}}{2} s V^{2/3} \frac{\pi_{\alpha}^2}{m_{\alpha}^2 s^4 V^{4/3}} + \frac{\pi_s^2}{2M_s} + \frac{\pi_V^2}{2M_V} - U\left(\{\} V^{1/3}\vec{q}_{\alpha}\right) - gk_B T \ln s - PV\right]$$

Simplificando la expresión anterior

$$\mathcal{H} = \sum_{\alpha} \frac{\pi_{\alpha}^2}{2m_{\alpha} s^2 V^{2/3}} + \frac{\pi_s^2}{2M_s} + \frac{\pi_V}{2M_V} + U\left(\{\} V^{1/3}\vec{q}_{\alpha}\right) + gk_b T \ln s + PV \tag{98}$$

Las ecuaciones de evolución temporal se obtienen derivando al Hamiltoniano.

$$\frac{dq_{\alpha}}{dt} = \frac{\partial \mathcal{H}}{\partial \pi_{\alpha}} = \frac{\pi_{\alpha}}{m_{\alpha} s^2 V^{2/3}} \tag{99}$$

$$\frac{d\pi_{\alpha}}{dt} = -\frac{\partial \mathcal{H}}{\partial q_{\alpha}} = -V^{1/3}\frac{\partial U\left(\left\{\vec{r}_{\gamma}\right\}\right)}{\partial r_{\alpha}} \tag{100}$$

$$\frac{ds}{dt} = \frac{\partial \mathcal{H}}{\partial \pi_s} = \frac{\pi_s}{M_s} \tag{101}$$

$$\frac{d\pi_s}{dt} = -\frac{\partial \mathcal{H}}{\partial s} = \frac{1}{s}\left[\sum_{\alpha} \frac{\pi_{\alpha}^2}{m_{\alpha} s^2 V^{2/3}} - gk_B T\right] \tag{102}$$

$$\frac{dV}{dt} = \frac{\partial \mathcal{H}}{\partial \pi_V} = \frac{\pi_V}{M_V} \tag{103}$$

$$\frac{d\pi_V}{dt} = -\frac{\partial \mathcal{H}}{\partial V} = \frac{1}{3V}\sum_{\alpha}\left[\frac{\pi_{\alpha}^2}{m_{\alpha} s^2 V^{2/3}} - q_{\alpha} V^{1/3}\frac{\partial U\left(\left\{V^{1/3}\vec{q}_{\gamma}\right\}\right)}{\partial\left(V^{1/3} q_{\alpha}\right)}\right] - P \tag{104}$$

Estas ecuaciones de movimiento no están en términos de las variables reales, pues las coordenadas y momentos generalizados contienen tanto las variables r_{α} y p_{α}, así como los grados de libertad añadidos s y V. Se deben establecer conexiones entre ambos conjuntos de variables para hacer la transición al espacio real.

Primero, se tiene esta relación, la cual se obtiene directamente de comparar el argumento de la energía potencial U:

$$r_{\alpha} = q_{\alpha} V^{1/3} \tag{105}$$

Esta relación nos indica que q_{α} es adimensional y está normalizada. Luego, se puede concluir que efectivamente la introducción del barostato se traduce en un incremento o reducción de las posiciones de los átomos para simular los efectos de la presión en el sistema.

Para la variable s, se sabe que su finalidad era escalar el tiempo. Luego, se tiene que

$$dt = s dt' \tag{106}$$

donde t' es el tiempo real. Entonces, si se quiere dejar todo en términos del espaico real, se deben cambiar las derivadas temporales respecto a este tiempo real:

$$\frac{dq_{\alpha}}{dt'} = \frac{\pi_{\alpha}}{m_{\alpha} s V^{2/3}} \tag{107}$$

$$\frac{d\pi_\alpha}{dt'} = -sV^{1/3}\frac{\partial U\left(\left\{\vec{r}_\gamma\right\}\right)}{\partial r_\alpha} \tag{108}$$

$$\frac{ds}{dt'} = \frac{s\pi_s}{M_s} \tag{109}$$

$$\frac{d\pi_s}{dt'} = \sum_\alpha m_\alpha s^2 V^{2/3}\left(\frac{dq_\alpha}{dt'}\right)^2 - gk_BT \tag{110}$$

$$\frac{dV}{dt'} = \frac{s\pi_V}{M_V} \tag{111}$$

$$\frac{d\pi_V}{dt'}\frac{1}{s} = \frac{1}{3V}\sum_\alpha\left[m_\alpha V^{2/3}\left(\frac{dq_\alpha}{dt'}\right)^2 - q_\alpha V^{1/3}\frac{\partial U\left(\left\{V^{1/3}\vec{q}_\gamma\right\}\right)}{\partial\left(V^{1/3}q_\alpha\right)}\right] - P \tag{112}$$

Todas las nuevas ecuaciones en el tiempo real siguen conteniendo al factor s. Para eliminar dicho factor, se toma la ecuación 107 y se deriva nuevamente respecto al tiempo real:

$$\frac{d^2q_\alpha}{dt'^2} = \frac{1}{m_\alpha V^{2/3}s}\frac{d\pi_\alpha}{dt'} - \frac{\pi_\alpha}{m_\alpha V^{2/3}s}\left(\frac{1}{s}\frac{ds}{dt'}\right) - \frac{\pi_\alpha}{m_\alpha V^{2/3}s}\left(\frac{2}{3V}\frac{dV}{dt'}\right) \tag{113}$$

Haciendo

$$f_\alpha = \frac{1}{V^{1/3}s}\frac{d\pi_\alpha}{dt'} \tag{114}$$

Reescribiendo la ecuación 113 en términos de la primera derivada de las coordenadas generalizadas, del momento conjugado de s y de la ecuación 114, se tiene

$$\frac{d^2q_\alpha}{dt'^2} = \frac{f_\alpha}{m_\alpha V^{1/3}} - \frac{dq_\alpha}{dt'}\left(\frac{\pi_s}{M_s}\right) - \frac{dq_\alpha}{dt'}\left(\frac{2}{3V}\frac{dV}{dt'}\right) \tag{115}$$

Para establecer la conexión entre q_α y p_α, se propone reescribir la coordenada generalizada como

$$V^{1/3}\frac{d}{dt}q_\alpha = \frac{p_\alpha}{m_\alpha} \tag{116}$$

Utilizando ahora la ecuación 107 y la relación 116, se tiene que

$$p_\alpha = \frac{\pi_\alpha}{sV^{1/3}} \tag{117}$$

Finalmente, la última conexión entre r_α y p_α se consigue usando que

$$\frac{dr_\alpha}{dt'} = \frac{d\left(q_\alpha V^{1/3}\right)}{dt'}$$

y sustituyendo la ecuación 116 en la relación anterior,

$$\frac{d}{dt}\vec{r}_\alpha = \frac{d\left(\vec{q}_\alpha V^{1/3}\right)}{dt'} = V^{1(3}\frac{d\vec{q}_\alpha}{dt'} + \vec{q}_\alpha\frac{1}{3}\frac{1}{V^{2/3}}\frac{dV}{dt'} = V^{1/3}\frac{d\vec{q}_\alpha}{dt'} + \vec{q}_\alpha\frac{V^{1/3}}{3V}\frac{dV}{dt'} = \frac{\vec{p}_\alpha}{m_\alpha} + \vec{r}_\alpha\frac{d\epsilon}{dt'} \tag{118}$$

Para finalizar con las conexiones, se va a trabajar primero con el lado izquierdo de la ecuación 113

$$\frac{d}{dt}\left(\frac{dq_\alpha}{dt'}\right) = \frac{d}{dt}\left(\frac{p_\alpha}{m_\alpha V^{1/3}}\right)$$

$$= \frac{1}{m_\alpha V^{1/3}}\frac{d}{dt}p_\alpha + \frac{p_\alpha}{m_\alpha}\frac{d}{dt}V^{-1/3}$$

Igualando con el lado derecho de la mencionada ecuación 113

$$\Rightarrow \frac{d}{dt}p_\alpha = m_\alpha V^{1/3}\left[\frac{f_\alpha}{m_\alpha V^{1/3}} - \left(\frac{p_\alpha}{m_\alpha}\frac{d}{dt}V^{-1/3} + \frac{dq_\alpha}{dt}\left(\frac{\pi_s}{M_s} + \frac{2}{3V}\frac{dV}{dt}\right)\right)\right]$$

$$= m_\alpha V^{1/3}\left[\frac{f_\alpha}{m_\alpha V^{1/3}} - \left(-\frac{1}{3}\frac{p_\alpha}{m_\alpha}V^{-4/3} + \frac{p_\alpha}{m_\alpha V^{1/3}}\left(\frac{\pi_s}{M_s} + \frac{2}{3V}\frac{dV}{dt}\right)\right)\right]$$

$$= f_\alpha - \left(-\frac{p_\alpha}{3V}\frac{dV}{dt} + p_\alpha\frac{\pi_s}{M_s} + p_\alpha\frac{2}{3V}\frac{dV}{dt}\right)$$

$$\therefore \frac{d}{dt}p_\alpha = f_\alpha - p_\alpha\left(\frac{\pi_s}{M_s} + \frac{1}{3V}\frac{dV}{dt}\right) \tag{119}$$

La ecuación 119 es la segunda ley de Newton, con un término extra, el cual incluye unos términos, que se pueden identificar como coeficientes de fricción termodinámicos del termostato y del barostato, siguiendo el resultado que se obtuvo en 118:

$$\frac{d}{dt}\vec{p}_\alpha = \vec{f}_\alpha - \vec{p}_\alpha\left[\zeta + \dot{\epsilon}\right] \tag{120}$$

donde

$$\zeta = \frac{\pi_s}{M_s} \qquad\qquad \frac{d\epsilon}{dt} = \frac{1}{3V} \tag{121}$$

Nótese que se simplificó la notación del tiempo para representar el tiempo real sin la notación primada.

La ecuación de movimiento de ζ se puede obtener partiendo de la ecuación 110 y sustituyendo el momento:

$$\frac{d}{dt}\pi_s = \sum_\alpha m_\alpha V^{2/3}\left(\frac{p_\alpha}{m_\alpha V^{1/3}}\right)^2 - gk_BT$$

$$= \sum_\alpha m_\alpha V^{2/3}\frac{p_\alpha^2}{m_\alpha^2 V^{2/3}} - gk_Bt = \sum_\alpha \frac{p_\alpha^2}{m_\alpha} - gk_BT$$

$$\therefore M_s\frac{d}{dt}\zeta = \sum_\alpha \frac{p_\alpha^2}{m_\alpha} - gk_BT \tag{122}$$

Usando el trabajo de Hoover [56] para describir la energía del sistema, se presenta dicha energía a partir de los términos deducidos en el Hamiltoniano

$$E = \sum_\alpha \frac{\vec{p}_\alpha}{2m_\alpha} + \frac{\pi_s^2}{2M_s} + \frac{\pi_V^2}{2M_V} + U(\{\vec{r}_\alpha\}) + gk_BT_{\text{equilibrio}}\ln s + P_{\text{externa}}V \tag{123}$$

La expresión 123 corresponde a un sistema de n+2 partículas, libres de alguna fuerza externa pero interactuando entre sí. Debido al mencionado aislamiento del sistema, la energía total, el momento lineal total y momento angular total se conservan.

Desde la perspectiva de la física estadística, el sistema de n+2 partículas se considera como ensamble microcanónico. Por otro lado, el cambio a las coordenadas reales representa un sistema de n partículas, el cual no conserva su energía a lo largo del tiempo, ya que hay una interacción con las partículas virtuales. No obstante, las partículas reales se encuentran en condiciones de presión y temperatura constantes, que era el propósito final de introducir el barostato y el termostato. Con estos argumentos expuestos anteriormente, se ve entonces que un sistema de n partículas con condiciones NPT se simula usando un sistema de n+2 partículas bajo condiciones NVE.

Referencias

[1] Cornell, Wendy D, Piotr Cieplak, Christopher I Bayly, Ian R Gould, Kenneth M Merz, David M Ferguson, David C Spellmeyer, Thomas Fox, James W Caldwell y Peter A Kollman: *A second generation force field for the simulation of proteins, nucleic acids, and organic molecules*. Journal of the American Chemical Society, 117:5179–5197, 1995.

[2] MacKerell Jr, Alexander D, Joanna Wiorkiewicz-Kuczera y Martin Karplus: *An all-atom empirical energy function for the simulation of nucleic acids*. Journal of the American Chemical society, 117:11946–11975, 1995.

[3] Fröhlking, Thorben, Mattia Bernetti, Nicola Calonaci y Giovanni Bussi: *Toward empirical force fields that match experimental observables*. The Journal of chemical physics, 152:230902, 2020.

[4] Sun, Tiedong, Vishal Minhas, Nikolay Korolev, Alexander Mirzoev, Alexander P Lyubartsev y Lars Nordenskiöld: *Bottom-up coarse-grained modeling of DNA*. Frontiers in Molecular Biosciences, 8:645527, 2021.

[5] Tanaka, A., Tomiya A. Hashimoto K.: *Deep Learning and Physics*. Springer, Singapúr, 1ª edición, 2021.

[6] Hohenberg, Pierre y Walter Kohn: *Inhomogeneous electron gas*. Physical review, 136:B864, 1964.

[7] Baydin, Atilim Gunes, Barak A. Pearlmutter y Alexey Andreyevich Radul: *Automatic differentiation in machine learning: a survey*. CoRR, abs/1502.05767, 2015. http://arxiv.org/abs/1502.05767.

[8] Baskin, Igor I, Vladimir A Palyulin y Nikolai S Zefirov: *A neural device for searching direct correlations between structures and properties of chemical compounds*. Journal of chemical information and computer sciences, 37:715–721, 1997.

[9] Allen, Michael P y Dominic J Tildesley: *Computer simulation of liquids*. Oxford university press, 2017.

[10] Bradbury, James, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne y Qiao Zhang: *JAX: composable transformations of Python+NumPy programs*, 2018. http://github.com/google/jax.

[11] Frostig, Roy, Matthew James Johnson y Chris Leary: *Compiling machine learning programs via high-level tracing*. Systems for Machine Learning, 4, 2018.

[12] Battaglia, Peter W, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner *y cols.*: *Relational inductive biases, deep learning, and graph networks*. arXiv preprint arXiv:1806.01261, 2018.

[13] Schoenholz, Samuel y Ekin Dogus Cubuk: *Jax md: a framework for differentiable physics*. Advances in Neural Information Processing Systems, 33:11428–11441, 2020.

[14] Roberts, Daniel A, Sho Yaida y Boris Hanin: *The Principles of Deep Learning Theory: An Effective Theory Approach to Understanding Neural Networks*. Cambridge University Press, 2022.

[15] Thaler, Stephan y Julija Zavadlav: *Learning neural network potentials from experimental data via Differentiable Trajectory Reweighting*. Nature Communications, 12:6884–6884.

[16] McCarthy, John, Marvin L Minsky, Nathaniel Rochester y Claude E Shannon: *A proposal for the dartmouth summer research project on artificial intelligence, august 31, 1955*. AI magazine, 27:12–12, 2006.

[17] Livingstone, David J: *Artificial neural networks: methods and applications*. Springer, 2008.

[18] Aggarwal, Charu C: *Artificial Intelligence-A Textbook*, 2021.

[19] Lü, Linyuan, Matúš Medo, Chi Ho Yeung, Yi Cheng Zhang, Zi Ke Zhang y Tao Zhou: *Recommender systems*. Physics reports, 519:1–49, 2012.

[20] Clark, WESLEY A y Belmont G Farley: *Generalization of pattern recognition in a self-organizing system*. En *Proceedings of the March 1-3, 1955, western joint computer conference*, páginas 86–91, 1955.

[21] Rosenblatt, Frank: *The perceptron: a probabilistic model for information storage and organization in the brain.* Psychological review, 65:386, 1958.

[22] Behler, Jörg y Michele Parrinello: *Generalized neural-network representation of high-dimensional potential-energy surfaces*. Physical review letters, 98:146401, 2007.

[23] Haykin, Simon: *Neural networks and learning machines, 3/E*. Pearson Education India, 2009.

[24] Dan, Yabo, Yong Zhao, Xiang Li, Shaobo Li, Ming Hu y Jianjun Hu: *Generative adversarial networks (GAN) based efficient sampling of chemical composition space for inverse design of inorganic materials*. npj Computational Materials, 6:1–7, 2020.

[25] Goodfellow, Ian, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville y Yoshua Bengio: *Generative adversarial nets*. Advances in neural information processing systems, 27, 2014.

[26] Samuel, Arthur L: *Some studies in machine learning using the game of checkers. II—Recent progress*. IBM Journal of research and development, 11:601–617, 1967.

[27] Thorndike, Edward L: *Animal intelligence: An experimental study of the associative processes in animals*. The Psychological Review: Monograph Supplements, 2:i, 1898.

[28] Sutton, Richard S. y Andrew G. Barto: *Reinforcement Learning: An Introduction*. The MIT Press, second edición, 2018.

[29] Rios, Jorge D, Alma Y Alanis, Nancy Arana-Daniel y Carlos Lopez-Franco: *Neural networks modeling and control: applications for unknown nonlinear delayed systems in discrete time*. 2020.

[30] Lapan, Maxim: *Deep Reinforcement Learning Hands-On: Apply modern RL methods to practical problems of chatbots, robotics, discrete optimization, web automation, and more*. Packt Publishing Ltd, 2020.

[31] Camperos, Edgar Nelson Sánchez y Alma Yolanda Alanís García: *Redes neuronales: conceptos fundamentales y aplicaciones a control automático*. 2006.

[32] Hagan, Martin T, Howard B Demuth, Mark Hudson Beale y Orlando De Jesús: *Neural network design. 2nd edtion*. Oklahoma: Martin Hagan, 2014.

[33] Scarselli, Franco, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner y Gabriele Monfardini: *The graph neural network model*. IEEE transactions on neural networks, 20:61–80, 2008.

[34] Srinivasan, Ashwin, Stephen Muggleton, Ross D King y Micheal JE Sternberg: *Mutagenesis: ILP experiments in a non-determinate biological domain*. En *Proceedings of the 4th international workshop on inductive logic programming*, volumen 237, páginas 217–232. Citeseer, 1994.

[35] Kipf, Thomas N y Max Welling: *Semi-supervised classification with graph convolutional networks*. arXiv preprint arXiv:1609.02907, 2016.

[36] Haykin, Simon: *Neural networks*. A comprehensive foundation, 1994.

[37] Leach, Andrew R y Andrew R Leach: *Molecular modelling: principles and applications*. Pearson education, 2001.

[38] Gastegger, Michael y Philipp Marquetand: *Molecular dynamics with neural network potentials*. En *Machine learning meets quantum physics*, páginas 233–252. Springer, 2020.

[39] Klicpera, Johannes, Shankari Giri, Johannes T Margraf y Stephan Günnemann: *Fast and uncertainty-aware directional message passing for non-equilibrium molecules*. arXiv preprint arXiv:2011.14115, 2020.

[40] Yedidia, Jonathan S, William T Freeman, Yair Weiss *y cols.*: *Understanding belief propagation and its generalizations*. Exploring artificial intelligence in the new millennium, 8:0018–9448, 2003.

[41] Quoc, V: *Le Prajit Ramachandran, Barret Zoph*. Searching for activation functions, 2018.

[42] He, Kaiming, Xiangyu Zhang, Shaoqing Ren y Jian Sun: *Deep residual learning for image recognition. CVPR. 2016*. arXiv preprint arXiv:1512.03385, 2016.

[43] Auletta, Gennaro, Mauro Fortunato y Giorgio Parisi: *Quantum mechanics*. Cambridge University Press, 2009.

[44] Rincón, Luis: *Introducción a la probabilidad*. 2014.

[45] Yaida, Sho: *Non-Gaussian processes and neural networks at finite widths*. En *Mathematical and Scientific Machine Learning*, páginas 165–192. PMLR, 2020.

[46] Hartmann, Carsten: *Molecular Dynamics. With Deterministic and Stochastic Numerical Methods*, 2016.

[47] Kapral, Raymond y Giovanni Ciccotti: *Molecular dynamics: an account of its evolution*. En *Theory and Applications of Computational Chemistry*, páginas 425–441. Elsevier, 2005.

[48] Blank, Thomas B, Steven D Brown, August W Calhoun y Douglas J Doren: *Neural network models of potential energy surfaces*. The Journal of chemical physics, 103:4129–4137, 1995.

[49] Wen, Tongqi, Rui Wang, Lingyu Zhu, Linfeng Zhang, Han Wang, David J Srolovitz y Zhaoxuan Wu: *Specialising neural network potentials for accurate properties and application to the mechanical response of titanium*. npj Computational Materials, 7:1–11, 2021.

[50] Behler, Jörg: *Atom-centered symmetry functions for constructing high-dimensional neural network potentials*. The Journal of chemical physics, 134:074106, 2011.

[51] Behler, Jörg: *Constructing high-dimensional neural network potentials: a tutorial review*. International Journal of Quantum Chemistry, 115:1032–1050, 2015.

[52] Gilmer, Justin, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals y George E Dahl: *Neural message passing for quantum chemistry*. En *International conference on machine learning*, páginas 1263–1272. PMLR, 2017.

[53] Hofstetter, Albert, Lennard Böselt y Sereina Riniker: *Graph Convolutional Neural Networks for (QM) ML/MM Molecular Dynamics Simulations*. arXiv preprint arXiv:2206.14422, 2022.

[54] Santamaría, Rubén y Jacques Soullard: *Particle dynamics in the NPT ensemble from extended Lagrangians*. 2022.

[55] Tuckerman, Mark E, Yi Liu, Giovanni Ciccotti y Glenn J Martyna: *Non-Hamiltonian molecular dynamics: Generalizing Hamiltonian phase space principles to non-Hamiltonian systems*. The Journal of Chemical Physics, 115:1678–1702, 2001.

[56] Martyna, Glenn J, Michael L Klein y Mark Tuckerman: *Nosé–Hoover chains: The canonical ensemble via continuous dynamics*. The Journal of chemical physics, 97:2635–2643, 1992.

[57] Tuckerman, Mark E, José Alejandre, Roberto López-Rendón, Andrea L Jochim y Glenn J Martyna: *A Liouville-operator derived measure-preserving integrator for molecular dynamics simulations in the isothermal–isobaric ensemble*. Journal of Physics A: Mathematical and General, 39:5629, 2006.

[58] Verlet, Loup: *Computer.^{ex}periments.^{o}n classical fluids. I. Thermodynamical properties of Lennard-Jones molecules*. Physical review, 159:98, 1967.

[59] Jones, John Edward: *On the determination of molecular fields.—I. From the variation of the viscosity of a gas with temperature*. Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character, 106:441–462, 1924.

[60] Lennard-Jones, John E: *Cohesion*. Proceedings of the Physical Society, 43:461, 1931.

[61] McQuarrie, DA: *Statistical mechanics/Donald A. McQuarrie*. Harper's chemistry series (Harper & Row, New York, 1975)"Portions of this work were originally published under the title: Statistical thermodynamics, 1975.

[62] DiStasio, Robert, Zhaofeng Li, Biswajit Santra, Xifan Wu y Roberto Car: *Liquid water from first principles: The importance of exact exchange, dispersion interactions, and nuclear quantum effects*. En *APS March Meeting Abstracts*, volumen 2013, páginas A5–005, 2013.

[63] Soper, AK y CJ Benmore: *Quantum differences between heavy and light water*. Physical review letters, 101:065502, 2008.

[64] Van Der Spoel, David, Erik Lindahl, Berk Hess, Gerrit Groenhof, Alan E Mark y Herman JC Berendsen: *GROMACS: fast, flexible, and free*. Journal of computational chemistry, 26:1701–1718, 2005.

[65] Michaud-Agrawal, Naveen, Elizabeth J Denning, Thomas B Woolf y Oliver Beckstein: *MDAnalysis: a toolkit for the analysis of molecular dynamics simulations*. Journal of computational chemistry, 32:2319–2327, 2011.

[66] Bressert, Eli: *SciPy and NumPy: an overview for developers*. 2012.

[67] Oobatake, Motohisa y Tatsuo Ooi: *Determination of Energy Parameters in Lennard-Jones Potentials from Second Virial Coefficients*. Progress of Theoretical Physics, 48:2132–2143, Diciembre 1972, ISSN 0033-068X. https://doi.org/10.1143/PTP.48.2132.

[68] Matsumoto, Makoto y Takuji Nishimura: *Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator*. ACM Transactions on Modeling and Computer Simulation (TOMACS), 8:3–30, 1998.

[69] Martyna, Glenn J, Mark E Tuckerman, Douglas J Tobias y Michael L Klein: *Explicit reversible integrators for extended systems dynamics*. Molecular Physics, 87:1117–1157, 1996.

[70] Eastman, Peter, Jason Swails, John D Chodera, Robert T McGibbon, Yutong Zhao, Kyle A Beauchamp, Lee Ping Wang, Andrew C Simmonett, Matthew P Harrigan, Chaya D Stern *y cols.*: *OpenMM 7: Rapid development of high performance algorithms for molecular dynamics*. PLoS computational biology, 13:e1005659, 2017.

[71] Scherer, Christoph y Denis Andrienko: *Understanding three-body contributions to coarse-grained force fields*. Physical Chemistry Chemical Physics, 20:22387–22394, 2018.

[72] Lerman, PM: *Fitting segmented regression models by grid search*. Journal of the Royal Statistical Society Series C: Applied Statistics, 29:77–84, 1980.

[73] Kingma, Diederik P y Jimmy Ba: *Adam: A method for stochastic optimization*. arXiv preprint arXiv:1412.6980, 2014.

[74] Darken, Christian, Joseph Chang, John Moody *y cols.*: *Learning rate schedules for faster stochastic gradient search*. En *Neural networks for signal processing*, volumen 2, páginas 3–12. Citeseer, 1992.

[75] Cai, Shengze, Zhicheng Wang, Sifan Wang, Paris Perdikaris y George Em Karniadakis: *Physics-informed neural networks for heat transfer problems*. Journal of Heat Transfer, 143, 2021.

[76] Kubo, R, H Ichimura, T Usui y N Hashitsume: *Statistical Mechanics*, 1990.

[77] Géron, Aurélien: *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*. .ºReilly Media, Inc.

, 2019.

[78] Rojas, Raúl: *Neural networks: a systematic introduction*. Springer Science & Business Media, 2013.

[79] McCulloch, Warren S y Walter Pitts: *A logical calculus of the ideas immanent in nervous activity*. The bulletin of mathematical biophysics, 5:115–133, 1943.

[80] Krizhevsky, Alex, Ilya Sutskever y Geoffrey E Hinton: *Imagenet classification with deep convolutional neural networks*. Advances in neural information processing systems, 25, 2012.

[81] Rosenblatt, Frank: *Principles of neurodynamics. perceptrons and the theory of brain mechanisms*. Informe técnico, Cornell Aeronautical Lab Inc Buffalo NY, 1961.

[82] Ramakrishnan, Raghunathan, Pavlo O Dral, Matthias Rupp y O Anatole Von Lilienfeld: *Big data meets quantum chemistry approximations: the Δ-machine learning approach*. Journal of chemical theory and computation, 11:2087–2096, 2015.

[83] Oliphant, Travis E: *A guide to NumPy*, volumen 1. Trelgol Publishing USA, 2006.

[84] Klicpera, Johannes, Aleksandar Bojchevski y Stephan Günnemann: *Predict then propagate: Graph neural networks meet personalized pagerank*. arXiv preprint arXiv:1810.05997, 2018.

[85] Horn, Roger A: *The hadamard product*. En *Proc. Symp. Appl. Math*, volumen 40, páginas 87–169, 1990.

[86] Hansen, Jean Pierre y Ian Ranald McDonald: *Theory of simple liquids: with applications to soft matter*. Academic press, 2013.

[87] Beale, Paul D y RK Pathria: *Statistical Mechanics Third Edition*. 2021.

[88] Wang, Xipeng, Simón Ramírez-Hinestrosa, Jure Dobnikar y Daan Frenkel: *The Lennard-Jones potential: when (not) to use it*. Physical Chemistry Chemical Physics, 22:10624–10633, 2020.

[89] Schroeder, Daniel V: *An introduction to thermal physics*, 1999.