ANALISIS “STATE OF THE ART”:

INTRODUCCIÓ:

Abans de profunditzar en els diversos treballs seleccionats, és essencial establir alguns conceptes fonamentals. Tot i que en els treballs estudiats s’hi tracten molts tipus de simulacions de dinàmica molecular (DM a partir d’ara), en aquest apartat es contemplaran majoritàriament les simulacions de DM que busquen calcular l'energia de l'estat fonamental **(U0)** d'una molècula. L'atribut **U0** indica l'energia total mínima d'una molècula en l'estat fonamental, es a dir l'estat d'energia més baixa que pot tenir el sistema. Aquesta energia es dona en electrons-volts **(eV)** o be amb **kcal/mol**.

Per a realitzar una simulació de DM d’aquest tipus, és necessari definir un conjunt de condicions inicials, que inclouen les posicions **(R)** i el numero atòmic **(Z)** dels àtoms del sistema. Aquestes condicions inicials es solen obtenir a partir d'una configuració estructural coneguda del sistema, com ara una estructura cristal·lina, o bé a partir d'una configuració aleatòria que segueixi les restriccions imposades per les interaccions del sistema.

Les posicions **(R)** son comunament representades per coordenades cartesianes, que són les més senzilles, consisteixen en la definició de la posició de cada àtom en un sistema de coordenades tridimensional, que es representa generalment en unitats de longitud com àngstroms **(Å)** o nanòmetres (nm). Així, una molècula es pot representar per la seva posició en l'espai, definida per les coordenades x, y i z de cada àtom que la conforma.

Tradicionalment el càlcul de l’U0 es fa resolent les equacions de Schrödinger per obtenir la funció d'ona electrònica i la seva energia associada[9]. Aquests mètodes són computacionalment costosos ja que han de tenir en compte la naturalesa quàntica dels electrons i el moviment dels nuclis atòmics. Això implica el càlcul de les funcions d'ona de tots els electrons de la molècula, i per tant, requereixen un gran nombre de càlculs matemàtics i computacionals. Com a resultat, aquests mètodes són limitats per la seva capacitat de resoldre problemes en molècules grans i complexes. Petant no es realista dur a terme simulacions a gran escala i queden limitats a molècules petites. Dit això, els mètodes que utilitzen Machine Learning s'han convertit en una alternativa prometedora per al càlcul de l'energia de l'estat fonamental de molècules més grans i complexes.

Recentment han aparegut nous paquets que fan servir tècniques de Machine Learning per a accelerar les simulacions, com SchNetPack i TorchMD. Aquests paquets permeten reduir dràsticament el temps de càlcul, cosa que els fa més atractius per als investigadors que realitzen simulacions a gran escala. En la següent secció es presentaran diversos treballs que han fet servir aquests paquets de programari per a realitzar les seves simulacions de dinàmica molecular.

(Afegir apartat de data sets)

SCHNETPACK2.0:

SchNetPack 2.0 és una "toolbox" dissenyada per al desenvolupament i desplegament de xarxes neuronals (XN) per a simulacions de DM. Proporciona un "framework" flexible i modular per a la construcció de models complexos que poden predir diverses propietats de molècules, com ara forces d'interacció intermoleculars, energies fonamentals, entre d'altres. Les principals aportacions de SchNetPack 2.0 son: una "data pipeline" flexible, modularitat a l'hora de construir els models de XN, implementació de PyTorch per a DM, una interfície de comandos basada en Hydra per a simplificar-ne l'ús i integració de PyTorch Lightning que permet gestionar i realitzar entrenaments fàcilment.

Data Pipeline:

La "data pipeline" es un component crucial per al funcionament de SchNetPack 2.0, forma el framework que permet processar i preparat les dades per els models de XN. Esta compost per 2 components principals en forma de classe: **`ASEAtomsData, AtomsLoader`**.

`ASEAtomsData` proporciona una interfície per carregar i manipular les dades. Es un afegit a la interfície de la llibreria Atomic Simulation Environment (ASE)[10], proveïda per PyTorch això permet a l'usuari establir una sèrie de "preprocessing transforms" que s'apliquen a les dades individualment previ a que siguin agrupades i enviades als models. Aquesta funcionalitat és particularment útil per calcular llistes de veïns, eliminar offsets o assignar propietats.

Category	Transform	Usage	Description
Neighbor lists	`MatScipyNeighborList`	Pre	Neighbor list implementation based on Matscipy.[59] This should be preferred
	`ASENeighborList`	Pre	Neighbor list based on atomic simulation environment
	`TorchNeighborList`	Pre	Neighbor list implemented in PyTorch
	`CachedNeighorList`	Pre	Wrapper for other neighbor list transforms that caches the results
	`SkinNeighborList`	Pre	Wrapper around neighbor list transform that only recalculates neighbor indices after atom positions change more than a given threshold. This can be useful for structure relaxation
	`FilterNeighbors`	Pre	Filter previously calculated neighbor indices
	`CountNeighbors`	Pre	Count and store number of neighbors for each atom
	`WrapPositions`	Pre	Wrap atom position into periodic cell
Casting	`CastMap`	Pre/Post	Cast all properties according to supplied type map
	`CastTo32`	Pre/Post	Cast all double precision inputs to single precision
	`CastTo64`	Pre/Post	Cast all single precision inputs to double precision
Scale and offset	`ScaleProperty`	Pre/Post	Scale an input or result by data mean, standard deviation or given factor
	`RemoveOffsets`	Pre/Post	Remove single-atom reference and/or mean from an input or result
	`AddOffsets`	Pre/Post	Add single-atom reference and/or mean to an input or result

Taula 1: Transforms/Operacions de pre- i postprocesament disponibles.

Aquestes operacions són importants per garantir que les dades estiguin en el format correcte i que puguin ser processades eficientment pel model. Algunes de les operacions més comuns inclouen el càlcul de llistes de veïns, la eliminació d'offsets i el casting de propietats. Per exemple, les llistes de veïns s'utilitzen per identificar quins àtoms estan prou propers per interactuar entre ells. Les operacions no nomes s'apliquen abans d'enviar al model sinó també en la sortida. És important destacar que aquestes operacions només s'habiliten per a la predicció i no durant el entrenament o l'avaluació del model, donat que la funció utilitzada per avaluar el model es independent de les operacions de postprocesament.

`AtomsLoader` agrupa grans quantitats de dades per processar-les amb models. Concretament, AtomsLoader és una funcionalitat Pytorch per carrega de dades amb una funció collate personalitzada que permet agrupar les dades de manera personalitzada per adaptar-se a les necessitats i aplicacions específiques. Permet carregar aquests lots en paral·lel així assolint un processament ràpid i eficient de grans conjunts de dades.

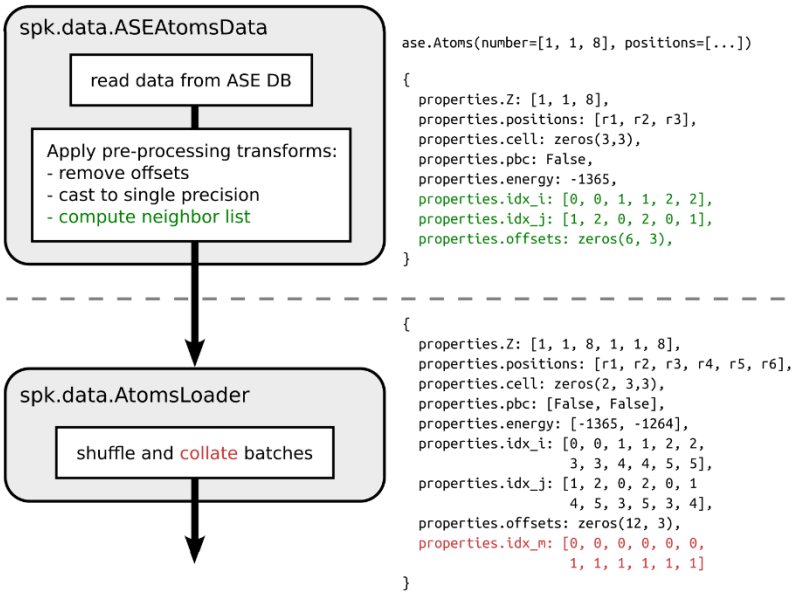


Figura 1: Diagrama del funcionament de la carrega, preprocesament i agrupament de les dades amb l'ús dels mòduls ASEAtomsData i AtomsLoader. A la dreta exemple dels diccionaris generats durant el procés, (sobre) diccionaris després del preprocesament, (sota) després de l'agrupament. Tots els valors son Tensors.

Figura extreta del paper original[10]

La "data pipeline" representada en la Figura 1 (esquerra), explica els passos per els quals passen les dades per a poder ser processades per els models. S'inicia el procés llegint les dades guardades en la base de dades ASE mitjançant la interfície proporcionada per ASEAtomsData, seguidament s'apliquen les operacions de preprocesament necessàries (llistat de totes les operacions disponibles en la Taula 1). Una vegada les dades han estat pre-processades, es passen a AtomsLoader, que carrega les dades que les agrupa per passar-les al model. La funció collate en AtomsLoader està personalitzada per al agrupament correcte de dades i garanteix que els tensors índex com `idx_i`, `idx_j` i `idx_m` estiguin correctament desplaçats per referir-se a la posició correcta en l'agrupació.

En el segment de codi de la Figura 1 (dreta), s'exposa un exemple de la representació d'una col·lecció de 3 àtoms en la base de dades ASE[11], un diccionari de propietats on la clau es una llista del numero atòmic **Z** i les posicions **r** dels àtoms, i les dades son les propietats com ara l'**energia** en la configuració actual o **cell** utilitzada per representar les dimensions i forma de l'espai on es troben els àtoms. En verd el resultat del càlcul de les llistes veïns `idx_i`, `idx_j` i de les nullificacions dels offsets. A sota el resultat de la agrupació amb una altre col·lecció d'àtoms i en vermell la generació del tensor "idx_m" que indexa les entrades amb el seu la seva col·lecció original.

Models:

SchNetPack te com a objectiu tant subministrar models amb estructures ja definides com permetre el desenvolupament i entrenament de models nous, la classe **`AtomisticModel`**[13] es la base d'aquesta idea.

La classe `AtomisticModel` és una classe que hereta de la classe `nn.Module`[12] de PyTorch. La funcionalitat principal de la classe `nn.Module` es la definició de mòduls per XN, aquests estan formats per capes com ara `nn.Conv2d()` (una capa de convolucio2d), exemples de la definició d'un mòdul en la documentació[12], un mòdul pot contenir altres mòduls dintre de si mateix, fet que simplifica significativament la creació de models personalitzats a partir de mòduls ja definits. En la pràctica, això significa que els usuaris poden utilitzar `AtomisticModel` per definir les seves pròpies arquitectures de xarxa neuronal utilitzant una combinació de mòduls predefinits (com ara capes convolucionals o capes denses) i mòduls personalitzats (com ara Atomwise).

La classe AtomisticModel pren per input 5 camps:

- `postprocessors`: Llista opcional del les operacions per postprocessing (transforms), no s'utilitzen durant l'entrenament (com ja es va mencionar en l'apartat de data pipeline).
- `representation`: instancia de nn.Module, el mòdul que s'aplica al diccionari input per generar la representació que utilitzarà el mòdul output.
- `output_module`: instancia de nn.Module, el mòdul que s'aplica a la representació generada per el mòdul representation per produir la output.
- `input_dtype_str`: string que indica el tipus de data de la output (el valor predeterminat es "float32"). La data de input abans de ser processada per el model serà transformada al tipus especificat en aquest camp.
- `do_postprocessing`: Booleà que indica si s'apliquen o no les operacions de postprocessing (el valor predeterminat es True).

Implementa 6 mètodes:

- `collect_derivative`: amb l'ús de la funció modules() itera per cada mòdul en la classe (tant els de representació com els de output) i comprova si necessita el mòdul necessita realitzar "backpropagation", donat que no tots els mòduls disponibles en AtomisticModel ho necessiten, com ara `nn.ReLU`, una capa que actua com a funció de activació per a la input (no te cap paràmetre que requereixi d'aprenentatge). Una vegada identificats els guarda en la llista `required_derivatives`.
- `collect_outputs`: similar a `collect_derivative` itera cada mòdul per a comprovar si genera outputs, de ser el cas els emmagatzema a model_outputs, això significa que el model genera no nomes l'output final sinó tots els entremitjos, facilitant el depurament del model.
- `initialize_derivatives`: pren com a input la llista de inputs i itera la llista required_derivatives mitjançant la funció `requires_grad()` l'etiqueta les inputs dels mòduls de la llista com a que necessita càlcul de gradient per la backpropagation.
- `initialize_transforms`: pren com a input la itera la llista de operacions de postprocessing (transforms) i executa la funció `datamodule()` enviant una instancia de datamodule de la base de dades amb la que estiguem treballant en el moment. Això es fa per assegurar que el transform te a la seva disposició les dades que necessita de la base de dades actual.

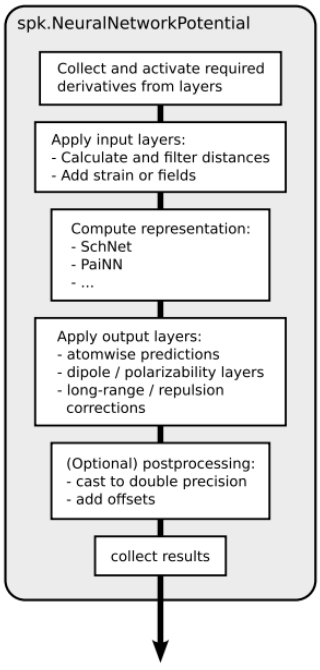
- `postprocess`: pren com a input la llista de inputs una vegada han sigut processats per el model, itera la llista de operacions de preprocessing i les aplica a la llista de inputs.
- `extract_outputs`: pren com a input la llista de inputs una vegada ha passat per el postprocessament, itera per la llista postprocessada i extreu les outputs especificades en model_outputs i construeix un diccionari amb les outputs extretes.

Dit això SchNetPack recomana utilitzar classe `NeuralNetworkPotential` una subclasse de `AtomisticModel`, amb l'objectiu de simplificar la creació de models MLP (Machine learning potencials). `NeuralNetworkPotential` aplica seqüencialment les funcions definides en la classe `AtomisticModel`, comparteix els mateixos paràmetres que `AtomisticModel` menys `model_outputs` i afegeix els paràmetre `input_modules` i `output_modules`.

- `input_modules`: llista de mòduls de `nn.Module`, s'apliquen a els diccionaris de input per tant es consideren operacions perprocessament.
- `output_modules`: llista de mòduls de `nn.Module`, tenen el rol del predictor del model, son els mòduls que s'ocupen de generar les outputs.

Implementa un nou mètode respecte `AtomisticModel`:

- `forward`: pren com a input un diccionari de inputs (durant les anteriors seccions "llistes de inputs"), es la funció responsable de passar el diccionari de inputs per totes les funcions necessàries. El diccionari de input te el següent format: `inputs: Dict[str, torch.Tensor]`, on les claus son el nom dels atributs i els valors son tensors de PyTorch.




```python
def forward(self, inputs: Dict[str, torch.Tensor])
    -> Dict[str, torch.Tensor]:

        # Inicialitzar les derivatives
        inputs = self.initialize_derivatives(inputs)

        # Aplicar les operacions de pre-processing
        for m in self.input_modules:
            inputs = m(inputs)

        # Aplicar les representacions
        inputs = self.representation(inputs)

        # Aplicar els moduls de prediccio
        for m in self.output_modules:
            inputs = m(inputs)

        # Aplicar les operacions de pre-processing
        inputs = self.postprocess(inputs)

        # Extreure els resultats
        results = self.extract_outputs(inputs)

        return results
```

Figura 2: Estructura de la classe NeuralNetworkPotential, (esquerra) diagrama de flux, (dreta) codi extret del github[13].

Figures opcionals:

```python
class ASEAtomsData(BaseAtomsData):

    def __init__(
        self,
        datapath: str,
        load_properties: Optional[List[str]] = None,
        load_structure: bool = True,
        transforms: Optional[List[torch.nn.Module]] = None,
        subset_idx: Optional[List[int]] = None,
        property_units: Optional[Dict[str, str]] = None,
        distance_unit: Optional[str] = None,
    ):
        """
        Args:
            datapath: Path to ASE DB.
            load_properties: Set of properties to be loaded and returned.
                If None, all properties in the ASE dB will be returned.
            load_structure: If True, load structure properties.
            transforms: preprocessing torch.nn.Module (see schnetpack.data.transforms)
            subset_idx: List of data indices.
            units: property-> unit string dictionary that overwrites the native units
                of the dataset. Units are converted automatically during loading.
        """
```

Figura 2: Fragment extret del codi de la classe ASEAtomsData que pren com a base la classe BaseAtomsData () original en el Github del Projecte. Es poden veure els atributs de propietats i transforms mencionats anteriorment. En l'apartat d'Arguments del codi s'explica aquests i altres no mencionats com ara les unitats.

1. Bibliografia:

[1] - Frenkel, Daan, and Berend Smit. **Understanding Molecular Simulation from Algorithms to Applications.** *Academic Press*, 2002.

[2] - Braun, Efrem, et al. **Best Practices for Foundations in Molecular Simulations [Article v1.0]. Living** *Journal of Computational Molecular Science*, vol. 1, no. 1, 2019

[3] - M. P. Allen and D. J. Tildesley. **Computer Simulations of Liquids.** *Oxford University Press*, 2002

[4] - J. Chem. Theory Comput. 2022, 18, 4, 2479–2493, Publication Date:March 8, 2022

[5] - K. T. Schütt, P. Kessel, M. Gastegger, K. A. Nicoli, A. Tkatchenko, and K.-R. Müller. **SchNetPack: A Deep Learning Toolbox For Atomistic Systems**. *Journal of chemical theory and computation*, 2019, 15, 448-455.

[6] - Stefan Doerr, Maciej Majewski, AdriàPérez, Andreas Krämer, Cecilia Clementi, Frank Noe, Toni Giorgino, and Gianni De Fabritiis. **TorchMD: A Deep Learning Framework for Molecular Simulations**. *Journal of chemical theory and computation*, 2021, 17, 2355−2363.

[7] - Takeru Miyagawa, Kazuki Mori, Nobuhiko Kato , Akio Yonezu. **Development of neural network potential for MD simulation and its application to TiN.** *Computational Material Sciencie*, 15 April 2022, 111303.

[8] - Larsen, A. H.; Mortensen, J. J.; Blomqvist, et. al, **The atomic simulation environmenta Python library for working with atoms.** *Journal Phys: Condens. Matter*, 2017, 29, 273002.

[9] - **The Schrödinger Equation for a Molecule**, (consultat: 10/5)

[10] - Kristof T. Schütt , Stefaan S. P. Hessmann, Niklas W. A. Gebauer, et. al, **SchNetPack 2.0: A neural network toolbox for atomistic machine learning** , *J. Chem. Phys.* 158, 144801 (2023)

[11] - **ASE representation for Atoms**, (consultat: 16/5)

[12] - **Documentacio nn.Module,** (consultat: 16/5)

[13] – **GitHub de SchNetPack**, (consultat: 17/5)