

Réseaux de neurones graphiques

Contenu

- [8.1. Représenter un graphique](#)
- [8.2. Exécution de ce bloc-notes](#)
- [8.3. Un réseau de neurones graphiques](#)
- [8.4. GCN Kipf & Welling](#)
- [8.5. Exemple de solubilité](#)
- [8.6. Point de vue de passage de message](#)
- [8.7. Réseau de neurones à graphes fermés](#)
- [8.8. mise en commun](#)
- [8.9. Fonction de lecture](#)
- [8.10. Équations générales de Battaglia](#)
- [8.11. L'architecture SchNet](#)
- [8.12. Exemple SchNet : Prédire les groupes d'espace](#)
- [8.13. Orientations de recherche actuelles](#)
- [8.14. Résumé du chapitre](#)
- [8.15. Des exercices](#)
- [8.16. Vidéos pertinentes](#)
- [8.17. Références citées](#)

Historiquement, la plus grande difficulté pour l'apprentissage automatique avec des molécules était le choix et le calcul des « descripteurs ». Les réseaux de neurones graphes (GNN) sont une catégorie de réseaux de neurones profonds dont les entrées sont des graphes et permettent de contourner le choix des descripteurs. Un GNN peut prendre une molécule directement en entrée.

i Public et objectifs

Ce chapitre s'appuie sur [les couches standard](#) et [la régression et l'évaluation du modèle](#). Bien qu'il soit défini ici, il serait bon de se familiariser avec les graphes/réseaux. Après avoir terminé ce chapitre, vous devriez être en mesure de

- Représenter une molécule dans un graphique
- Discuter et catégoriser les architectures courantes de réseaux de neurones à graphes
- Construisez un GNN et choisissez une fonction de lecture pour le type d'étiquettes
- Distinguer les fonctionnalités de graphe, d'arête et de nœud
- Formuler un GNN en mises à jour de périphérie, mises à jour de nœud et étapes d'agrégation

Les GNN sont des couches spécifiques qui entrent un graphique et génèrent un graphique. Vous pouvez trouver des critiques de GNN dans Dwivedi *et al.* [[DJL+20](#)], Bronstein *et al.* [[BBL+17](#)], et Wu *et al.* [[WPC+20](#)]. Les GNN peuvent être utilisés pour tout, de la dynamique moléculaire à gros grains [[LWC+20](#)] à la prédiction des déplacements chimiques RMN [[YCW20](#)] à la modélisation de la dynamique des solides [[XFLW+19](#)]. Avant de nous y plonger trop profondément, nous devons d'abord comprendre comment un graphe est représenté dans un ordinateur et comment les molécules sont converties en graphes.

Vous pouvez trouver un article d'introduction interactif sur les graphes et les réseaux neuronaux de graphes sur [distill.pub \[SLRPW21 \]](#). La plupart des recherches actuelles sur les GNN sont effectuées avec des bibliothèques d'apprentissage en profondeur spécialisées pour les graphes. Depuis 2022, les plus courants sont [PyTorch Geometric](#), [Deep Graph library](#), [DIG](#), [Spektral](#) et [TensorFlow GNNs](#).

8.1. Représenter un graphique

Un graphique G est un ensemble de nœuds V et bords E . Dans notre contexte, nœud i est défini par un vecteur \vec{v}_i , de sorte que l'ensemble des nœuds peut être écrit comme un tenseur de rang 2. Les arêtes peuvent être représentées comme une matrice d'adjacence E , où $e_{ij} = 1$ si nœuds i et j sont reliés par une arête. Dans de nombreux domaines, les graphes sont souvent immédiatement simplifiés pour être orientés et acycliques, ce qui simplifie les choses. Les molécules sont plutôt non dirigées et ont des cycles (anneaux). Ainsi, nos matrices d'adjacence sont toujours symétriques $e_{ij} = e_{ji}$ car il n'y a pas de notion de direction dans les liaisons chimiques. Souvent, nos bords eux-mêmes ont des caractéristiques, de sorte que e_{ij} est lui-même un vecteur. La matrice d'adjacence devient alors un tenseur de rang 3. Des exemples de caractéristiques de bord peuvent être l'ordre des liaisons covalentes ou la distance entre deux nœuds.

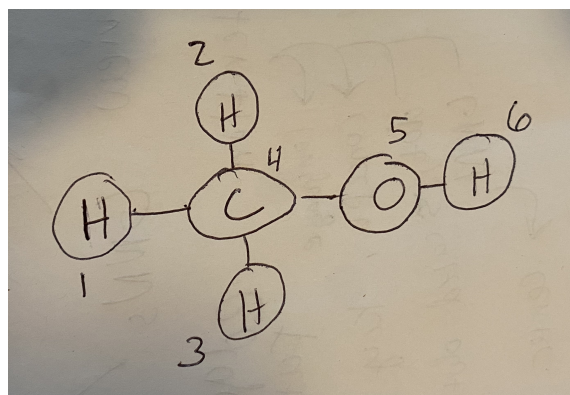


Fig. 8.1 Méthanol avec des atomes numérotés afin que nous puissions le convertir en graphique.

Voyons comment un graphe peut être construit à partir d'une molécule. Considérez le méthanol, illustré à [la Fig. 8.1](#). J'ai numéroté les atomes afin que nous ayons un ordre pour définir les nœuds/arêtes. Tout d'abord, les caractéristiques du nœud. Vous pouvez utiliser n'importe quoi pour les fonctionnalités de nœud, mais nous commencerons souvent par des vecteurs de fonctionnalités codés à chaud :

| Nœud | C | H | O |
|------|---|---|---|
| 1 | 0 | 1 | 0 |
| 2 | 0 | 1 | 0 |
| 3 | 0 | 1 | 0 |
| 4 | 1 | 0 | 0 |
| 5 | 0 | 0 | 1 |
| 6 | 0 | 1 | 0 |

V seront les vecteurs de caractéristiques combinés de ces nœuds. La matrice d'adjacence E ressemblera:

one-hot


Rappelez-vous qu'un one-hot est un vecteur de tous les 0 et d'un seul 1. L'indice de l'élément non nul indique la classe. Dans ce cas, la classe est un élément `[0, 1, 0, 0]`

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 0 | 1 | 0 | 0 |
| 3 | 0 | 0 | 0 | 1 | 0 | 0 |
| 4 | 1 | 1 | 1 | 0 | 1 | 0 |
| 5 | 0 | 0 | 0 | 1 | 0 | 1 |
| 6 | 0 | 0 | 0 | 0 | 1 | 0 |

Prenez un moment pour comprendre ces deux. Par exemple, notez que les lignes 1, 2 et 3 n'ont que la 4^{ème} colonne non nulle. C'est parce que les atomes 1 à 3 ne sont liés qu'au carbone (atome 4). De plus, la diagonale est toujours 0 car les atomes ne peuvent pas être liés entre eux.

You can find a similar process for converting crystals into graphs in Xie et al. [XG18]. We'll now begin with a function which can convert a smiles string into this representation.

8.2. Running This Notebook

Click the  above to launch this page as an interactive Google Colab. See details below on installing packages.

💡 Tip

Pour installer des packages, exécutez ce code dans une nouvelle cellule.

```
!pip install dmol-book
```

Si vous rencontrez des problèmes d'installation, vous pouvez obtenir les dernières versions de travail des packages utilisés dans [ce livre ici](#)

```
import matplotlib.pyplot as plt
import matplotlib as mpl
import numpy as np
import tensorflow as tf
import pandas as pd
import rdkit, rdkit.Chem, rdkit.Chem.rdDepictor, rdkit.Chem.Draw
import networkx as nx
import dmol
```

```
soldata = pd.read_csv(
    "https://github.com/whitead/dmol-book/raw/main/data/curated-solubility-
    dataset.csv"
)
np.random.seed(0)
my_elements = {6: "C", 8: "O", 1: "H"}
```

The hidden cell below defines our function `smiles2graph`. This creates one-hot node feature vectors for the element C, H, and O. It also creates an adjacency tensor with one-hot bond order being the feature vector.

```
def smiles2graph(sml):
    """Argument for the RD2NX function should be a valid SMILES sequence
    returns: the graph
    """
    m = rdkit.Chem.MolFromSmiles(sml)
    m = rdkit.Chem.AddHs(m)
    order_string = {
        rdkit.Chem.rdchem.BondType.SINGLE: 1,
        rdkit.Chem.rdchem.BondType.DOUBLE: 2,
        rdkit.Chem.rdchem.BondType.TRIPLE: 3,
        rdkit.Chem.rdchem.BondType.AROMATIC: 4,
    }
    N = len(list(m.GetAtoms()))
    nodes = np.zeros((N, len(my_elements)))
    lookup = list(my_elements.keys())
    for i in m.GetAtoms():
        nodes[i.GetIdx(), lookup.index(i.GetAtomicNum())] = 1

    adj = np.zeros((N, N, 5))
    for j in m.GetBonds():
        u = min(j.GetBeginAtomIdx(), j.GetEndAtomIdx())
        v = max(j.GetBeginAtomIdx(), j.GetEndAtomIdx())
        order = j.GetBondType()
        if order in order_string:
            order = order_string[order]
        else:
            raise Warning("Ignoring bond order" + order)
        adj[u, v, order] = 1
        adj[v, u, order] = 1
    return nodes, adj
```

```
nodes, adj = smiles2graph("CO")
nodes
```

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.],
       [0., 0., 1.]])
```

8.3. A Graph Neural Network

A graph neural network (GNN) is a neural network with two defining attributes:

1. Its input is a graph
2. Its output is permutation equivariant

We can understand clearly the first point. Here, a graph permutation means re-ordering our nodes. In our methanol example above, we could have easily made the carbon be atom 1 instead of atom 4. Our new adjacency matrix would then be:

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 1 | 0 |
| 2 | 1 | 0 | 0 | 0 | 0 | 0 |
| 3 | 1 | 0 | 0 | 0 | 0 | 0 |
| 4 | 1 | 0 | 0 | 0 | 1 | 0 |
| 5 | 1 | 0 | 0 | 0 | 0 | 1 |
| 6 | 0 | 0 | 0 | 0 | 1 | 0 |

A GNN is permutation equivariant if the output change the same way as these exchanges. If you are trying to model a per-atom quantity like partial charge or chemical shift, this is obviously essential. If you change the order of atoms input, you would expect the order of their partial charges to similarly change.

Souvent, nous voulons modéliser une propriété de la molécule entière, comme la solubilité ou l'énergie. Cela devrait être **invariant** au changement de l'ordre des atomes. Pour rendre invariant un modèle équivariant, on utilise des lectures (définies ci-dessous). Voir [Données d'entrée et équivariances](#) pour une discussion plus détaillée de l'équivariance.

8.3.1. Un simple

Nous mentionnerons souvent un GNN alors que nous entendons vraiment une couche à partir d'un GNN. La plupart des GNN implémentent une couche spécifique qui peut traiter des graphiques, et donc généralement nous ne nous intéressons qu'à cette couche. Voyons un exemple de couche simple pour un GNN :

$$f_k = \sigma \left(\sum_i \sum_j v_{ij} w_{jk} \right) \quad (8.1)$$

Cette équation montre que nous multiplions d'abord chaque nœud (v_{ij}) caractéristique par poids entraînaables w_{jk} , additionnez toutes les fonctionnalités de nœud, puis appliquez une activation. Cela donnera un seul vecteur de caractéristiques pour le graphique. Cette permutation d'équation est-elle invariante ? Oui, car l'index de nœud dans notre expression est indexé qui peut être réorganisé sans affecter la sortie.

Voyons un exemple similaire, mais pas invariant de permutation :

$$f_k = \sigma \left(\sum_i v_{ij} w_{ik} \right) \quad (8.2)$$

C'est un petit changement. Nous avons maintenant un vecteur de poids par nœud. Cela fait que les poids entraînaables dépendent de l'ordre des nœuds. Ensuite, si nous échangeons l'ordre des nœuds, nos poids ne s'aligneront plus. Donc, si nous devions entrer deux molécules de méthanol, qui devraient avoir la même sortie, mais que nous intervertissions deux nombres d'atomes, nous obtiendrions des réponses différentes. Ces exemples simples diffèrent des vrais GNN de deux manières importantes : (i) ils donnent une seule sortie de vecteur de caractéristiques, qui rejette les informations par nœud, et (ii) ils n'utilisent pas la matrice de contiguïté. Voyons un vrai GNN qui a ces propriétés tout en maintenant l'invariance de permutation - ou l'équivariance (la permutation des entrées permute les sorties de la même manière).

8.4. Kipf & Welling NGC

L'un des premiers GNN populaires a été le réseau convolutif de graphes de Kipf & Welling (GCN) [[KW16](#)]. Bien que certaines personnes considèrent les GCN comme une large classe de GNN, nous utiliserons les GCN pour désigner spécifiquement le GCN Kipf & Welling. Thomas Kipf a écrit un [excellent article présentant le GCN](#).

L'entrée d'une couche GCN est \mathbf{V} , et il produit une mise à jour \mathbf{V}' . Chaque vecteur de caractéristique de nœud est mis à jour. La façon dont il met à jour un vecteur de caractéristiques de nœud consiste à faire la moyenne des vecteurs de caractéristiques de ses voisins, comme déterminé par \mathbf{E} . Le choix de la moyenne sur les voisins est ce qui rend une permutation de couche GCN équivariante. La moyenne sur les voisins n'est pas formable, nous devons donc ajouter des paramètres formables. Nous multiplions les caractéristiques voisines par une matrice entraînable avant la moyenne, ce qui donne au GCN la capacité d'apprendre. En notation Einstein, ce processus est :

$$v_{il} = \sigma \left(\frac{1}{d_i} e_{ij} v_{jk} w_{kl} \right) \quad (8.3)$$

où i est le nœud que nous considérons, j est l'indice du voisin, k est la caractéristique d'entrée du nœud, l est la caractéristique du nœud de sortie, d_i est le degré du nœud i (ce qui en fait une moyenne au lieu d'une somme), e_{ij} isole les voisins afin que tous les non-voisins v_{jk} s sont nuls, σ est notre activation, et w_{lk} est le poids entraînable. Cette équation est une bouchée, mais c'est vraiment juste la moyenne sur les voisins avec une matrice entraînable. Une modification courante consiste à rendre tous les nœuds voisins d'eux-mêmes. C'est ainsi que les caractéristiques du nœud de sortie v_{il} dépend des caractéristiques d'entrée v_{ik} . Nous n'avons pas besoin de changer notre équation, il suffit de faire en sorte que la matrice d'adjacence ait 1s sur la diagonale au lieu de 0 en ajoutant la matrice d'identité lors du pré-traitement.

Il est important de mieux comprendre le GCN pour comprendre les autres GNN. Vous pouvez voir la couche GCN comme un moyen de « communiquer » entre un nœud et ses voisins. La sortie pour le nœud i dépendra que de ses voisins immédiats. Pour la chimie, ce n'est pas satisfaisant. Vous pouvez cependant empiler plusieurs couches. Si vous avez deux couches, la sortie pour le nœud i inclura des informations sur le nœud j les voisins des voisins. Un autre détail important à comprendre dans les GCN est que la procédure de calcul de la moyenne atteint deux objectifs : (i) elle donne une équivariance de permutation en supprimant l'effet de l'ordre des voisins et (ii) elle empêche un changement d'amplitude dans les caractéristiques des nœuds. Une somme accomplirait (i) mais entraînerait une augmentation de la magnitude des caractéristiques des nœuds après chaque couche. Bien sûr, vous pouvez placer ad hoc une couche de normalisation par lots après chaque couche GCN pour maintenir les amplitudes de sortie stables, mais la moyenne est facile.

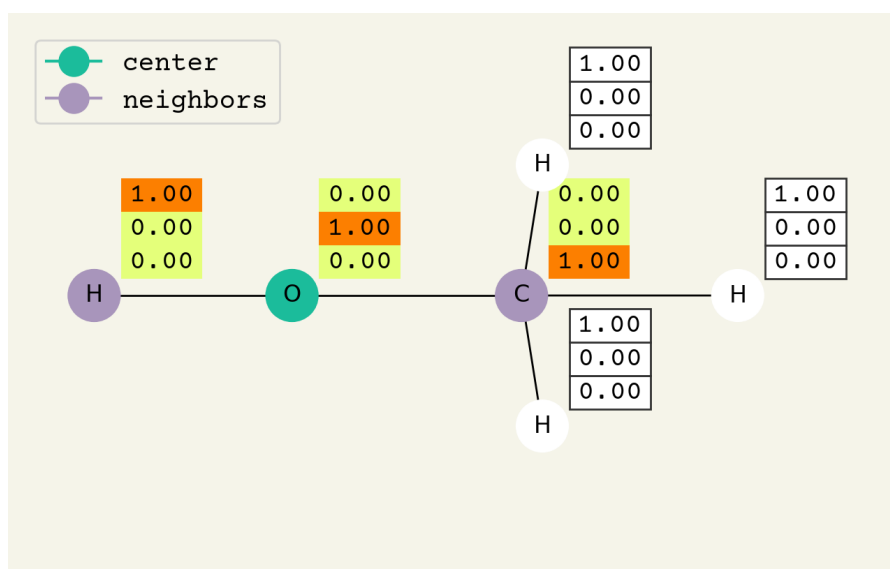


Fig. 8.2 Étape intermédiaire de la couche de convolution du graphe. Les vecteurs 3D sont les caractéristiques des nœuds et commencent comme un point chaud, donc un signifie hydrogène. Le nœud central sera mis à jour en faisant la moyenne de ses fonctionnalités voisines. [1.00, 0.00, 0.00]

Pour vous aider à comprendre la couche GCN, regardez la [Fig. 8.2](#) . Il montre une étape intermédiaire de la couche GCN. Chaque caractéristique de nœud est représentée ici sous la forme d'un vecteur codé à chaud en entrée. L'animation de [la Fig. 8.3](#) montre le processus de moyennage sur les entités voisines. Pour rendre cette animation facile à suivre, les poids entraîlables et les fonctions d'activation ne sont pas pris en compte. Notez que l'animation se répète pour un second calque. Regardez comment l'"information" sur la présence d'un atome d'oxygène dans la molécule ne se propage qu'après deux couches à chaque atome. Tous les GNN fonctionnent avec des approches similaires, alors essayez de comprendre comment cette animation fonctionne.

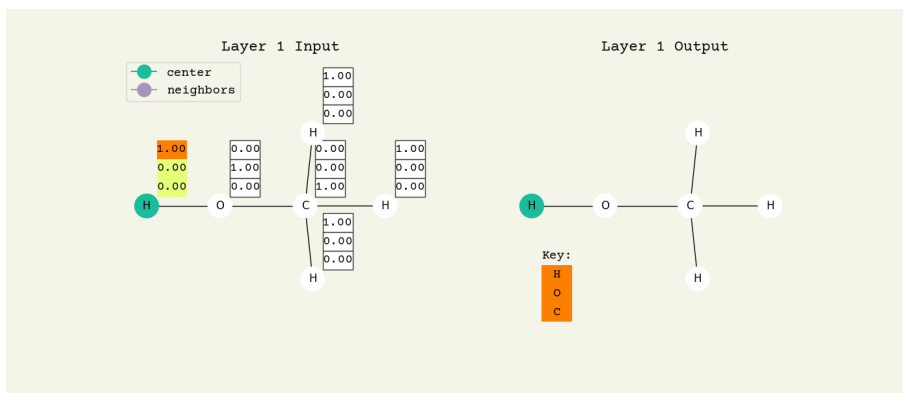


Fig. 8.3 Animation de l'opération de la couche de convolution du graphe. La gauche est l'entrée, la droite les caractéristiques du nœud de sortie. Notez que deux calques sont affichés (voir changement de titre). Au fur et à mesure que l'animation se déroule, vous pouvez voir comment les informations sur les atomes se propagent à travers la molécule via la moyenne sur les voisins. Ainsi, l'oxygène passe d'un simple oxygène à un oxygène lié à C et H, à un oxygène lié à un H et CH₃. Les couleurs reflètent simplement les mêmes informations dans les valeurs numériques.

8.4.1. de mise en œuvre du GCN

Créons maintenant une implémentation tensorielle du GCN. Nous allons ignorer l'activation et les poids entraînaables pour l'instant. Nous devons d'abord calculer notre matrice d'adjacence de rang 2. Le `smiles2graphcode` ci-dessus calcule un tenseur d'adjacence avec des vecteurs de caractéristiques. Nous pouvons résoudre ce problème avec une simple réduction et ajouter l'identité en même temps

```
nodes, adj = smiles2graph("CO")
adj_mat = np.sum(adj, axis=-1) + np.eye(adj.shape[0])
adj_mat
```

```
array([[1., 1., 1., 1., 1., 0.],
       [1., 1., 0., 0., 0., 1.],
       [1., 0., 1., 0., 0., 0.],
       [1., 0., 0., 1., 0., 0.],
       [1., 0., 0., 0., 1., 0.],
       [0., 1., 0., 0., 0., 1.]])
```

Pour calculer le degré de chaque nœud, nous pouvons faire une autre réduction :

```
degree = np.sum(adj_mat, axis=-1)
degree
```

```
array([5., 3., 2., 2., 2., 2.])
```

Maintenant, nous pouvons mettre toutes ces pièces ensemble dans l'équation d'Einstein

```
print(nodes[0])
# note to divide by degree, make the input 1 / degree
new_nodes = np.einsum("i,ij,jk->ik", 1 / degree, adj_mat, nodes)
print(new_nodes[0])
```

```
[1. 0. 0.]
[0.2 0.2 0.6]
```

To now implement this as a layer in Keras, we must put this code above into a new Layer subclass. The code is relatively straightforward, but you can read-up on the function names and Layer class in [this tutorial](#). The three main changes are that we create trainable parameters `self.w` and use them in the `tf.einsum`, we use an activation

`self.activation`, and we output both our new node features and the adjacency matrix. The reason to output the adjacency matrix is so that we can stack multiple GCN layers without having to pass the adjacency matrix each time.

```
class GCNLayer(tf.keras.layers.Layer):
    """Implementation of GCN as layer"""

    def __init__(self, activation=None, **kwargs):
        # constructor, which just calls super constructor
        # and turns requested activation into a callable function
        super(GCNLayer, self).__init__(**kwargs)
        self.activation = tf.keras.activations.get(activation)

    def build(self, input_shape):
        # create trainable weights
        node_shape, adj_shape = input_shape
        self.w = self.add_weight(shape=(node_shape[2], node_shape[2]),
                                name="w")

    def call(self, inputs):
        # split input into nodes, adj
        nodes, adj = inputs
        # compute degree
        degree = tf.reduce_sum(adj, axis=-1)
        # GCN equation
        new_nodes = tf.einsum("bi,bij,bjk,kl->bil", 1 / degree, adj, nodes,
                                self.w)
        out = self.activation(new_nodes)
        return out, adj
```

A lot of the code above is Keras/TF specific and getting the variables to the right place. There are really only two key lines here. The first is to compute the degree by summing over the columns of the adjacency matrix:

```
degree = tf.reduce_sum(adj, axis=-1)
```

The second key line is to do the GCN equation [\(8.3\)](#) (without the activation)

```
new_nodes = tf.einsum("bi,bij,bjk,kl->bil", 1 / degree, adj, nodes, self.w)
```

We can now try our layer:

```
gcnlayer = GCNLayer("relu")
# we insert a batch axis here
gcnlayer((nodes[np.newaxis, ...], adj_mat[np.newaxis, ...]))
```

```
(<tf.Tensor: shape=(1, 6, 3), dtype=float32, numpy=
array([[0.          , 0.46567526, 0.07535715],
       [0.          , 0.12714943, 0.05325063],
       [0.01475453, 0.295794  , 0.39316285],
       [0.01475453, 0.295794  , 0.39316285],
       [0.01475453, 0.295794  , 0.39316285],
       [0.          , 0.38166213, 0.          ]]), dtype=float32)>,
<tf.Tensor: shape=(1, 6, 6), dtype=float32, numpy=
array([[1., 1., 1., 1., 1., 0.],
       [1., 1., 0., 0., 0., 1.],
       [1., 0., 1., 0., 0., 0.],
       [1., 0., 0., 1., 0., 0.],
       [1., 0., 0., 0., 1., 0.],
       [0., 1., 0., 0., 0., 1.]]), dtype=float32)>)
```

It outputs (1) the new node features and (2) the adjacency matrix. Let's make sure we can stack these and apply the GCN multiple times

```
x = (nodes[np.newaxis, ...], adj_mat[np.newaxis, ...])
for i in range(2):
    x = gcnlayer(x)
print(x)
```



```
(<tf.Tensor: shape=(1, 6, 3), dtype=float32, numpy=
array([[0.          , 0.18908624, 0.          ],
       [0.          , 0.          , 0.          ],
       [0.          , 0.145219  , 0.          ],
       [0.          , 0.145219  , 0.          ],
       [0.          , 0.145219  , 0.          ],
       [0.          , 0.          , 0.          ]]), dtype=float32)>, <tf.Tensor:
shape=(1, 6, 6), dtype=float32, numpy=
array([[1., 1., 1., 1., 1., 0.],
       [1., 1., 0., 0., 0., 1.],
       [1., 0., 1., 0., 0., 0.],
       [1., 0., 0., 1., 0., 0.],
       [1., 0., 0., 0., 1., 0.],
       [0., 1., 0., 0., 0., 1.]])], dtype=float32)>)
```

It works! Why do we see zeros though? Probably because we had negative numbers that were removed by our ReLU activation. This will be solved by training and increasing our dimension number.

8.5. Solubility Example

Nous allons maintenant revoir la prédiction de la solubilité avec les GCN. N'oubliez pas qu'auparavant, nous avons utilisé les fonctionnalités incluses dans l'ensemble de données. Maintenant, nous pouvons utiliser directement les structures moléculaires. Notre couche GCN génère des fonctionnalités au niveau du nœud. Pour prédire la solubilité, nous devons obtenir une fonctionnalité au niveau du graphique. Nous verrons plus tard comment être plus sophistiqué dans ce processus, mais pour l'instant prenons simplement la moyenne sur toutes les fonctionnalités de nœud après nos couches GCN. C'est simple, invariant par permutation, et nous fait passer du niveau nœud au niveau graphe. Voici une implémentation de ceci

```
class GRLayer(tf.keras.layers.Layer):
    """A GNN layer that computes average over all node features"""
    def __init__(self, name="GRLayer", **kwargs):
        super(GRLayer, self).__init__(name=name, **kwargs)
    def call(self, inputs):
        nodes, adj = inputs
        reduction = tf.reduce_mean(nodes, axis=1)
        return reduction
```

La ligne clé dans ce code consiste simplement à calculer la moyenne sur les nœuds (`axis=1`):

```
reduction = tf.reduce_mean(nodes, axis=1)
```

Pour compléter notre prédicteur de solubilité profonde, nous pouvons ajouter des couches denses et nous assurer que nous avons une sortie unique sans activation puisque nous faisons une régression. Notez que ce modèle est défini à l'aide de l' [API fonctionnelle Keras](#) qui est nécessaire lorsque vous avez plusieurs entrées.

```

ninput = tf.keras.Input(
    (
        None,
        100,
    )
)
ainput = tf.keras.Input(
    (
        None,
        None,
    )
)
# GCN block
x = GCNLayer("relu")([ninput, ainput])
x = GCNLayer("relu")(x)
x = GCNLayer("relu")(x)
x = GCNLayer("relu")(x)
# reduce to graph features
x = GRLayer()(x)
# standard layers (the readout)
x = tf.keras.layers.Dense(16, "tanh")(x)
x = tf.keras.layers.Dense(1)(x)
model = tf.keras.Model(inputs=(ninput, ainput), outputs=x)

```

d'où vient le 100 ? Eh bien, cet ensemble de données contient de nombreux éléments, nous ne pouvons donc pas utiliser nos encodages uniques de taille 3, car nous aurons plus de 3 éléments uniques. Nous n'avions auparavant que C, H et O. C'est le bon moment pour mettre à jour notre `smiles2graph` fonction pour y faire face.

```

def gen_smiles2graph(sml):
    """Argument for the RD2NX function should be a valid SMILES sequence
    returns: the graph
    """
    m = rdkit.Chem.MolFromSmiles(sml)
    m = rdkit.Chem.AddHs(m)
    order_string = {
        rdkit.Chem.rdchem.BondType.SINGLE: 1,
        rdkit.Chem.rdchem.BondType.DOUBLE: 2,
        rdkit.Chem.rdchem.BondType.TRIPLE: 3,
        rdkit.Chem.rdchem.BondType.AROMATIC: 4,
    }
    N = len(list(m.GetAtoms()))
    nodes = np.zeros((N, 100))
    for i in m.GetAtoms():
        nodes[i.GetIdx(), i.GetAtomicNum()] = 1

    adj = np.zeros((N, N))
    for j in m.GetBonds():
        u = min(j.GetBeginAtomIdx(), j.GetEndAtomIdx())
        v = max(j.GetBeginAtomIdx(), j.GetEndAtomIdx())
        order = j.GetBondType()
        if order in order_string:
            order = order_string[order]
        else:
            raise Warning("Ignoring bond order" + order)
        adj[u, v] = 1
        adj[v, u] = 1
    adj += np.eye(N)
    return nodes, adj

```

```

nodes, adj = gen_smiles2graph("CO")
model((nodes[np.newaxis], adj_mat[np.newaxis]))

```

```

<tf.Tensor: shape=(1, 1), dtype=float32, numpy=array([[0.0107595]]),
dtype=float32>

```

Il sort un numéro! C'est toujours agréable d'avoir. Maintenant, nous devons faire du travail pour obtenir un ensemble de données entraînable. Notre jeu de données est un peu complexe car nos caractéristiques sont des tuples de tenseurs (**V**, **E**) pour que notre jeu de données soit un tuple de tuples : ((**V**, **E**), *y*). Nous utilisons un **générateur**, qui est juste une fonction python qui peut revenir plusieurs fois. Notre fonction revient une fois pour chaque exemple d'apprentissage. Ensuite, nous devons le passer au `from_generator` `tf.data.Dataset` constructeur qui nécessite une déclaration explicite des formes de ces exemples.

Nous sommes passés du tenseur d'adjacence à la matrice uniquement parce qu'un GCN ne peut pas utiliser les entités de bord. D'autres architectures le peuvent cependant.

```
def example():
    for i in range(len(soldata)):
        graph = gen_smiles2graph(soldata.SMILES[i])
        sol = soldata.Solubility[i]
        yield graph, sol

data = tf.data.Dataset.from_generator(
    example,
    output_types=((tf.float32, tf.float32), tf.float32),
    output_shapes=(
        (tf.TensorShape([None, 100]), tf.TensorShape([None, None])),
        tf.TensorShape([]),
    ),
)
```

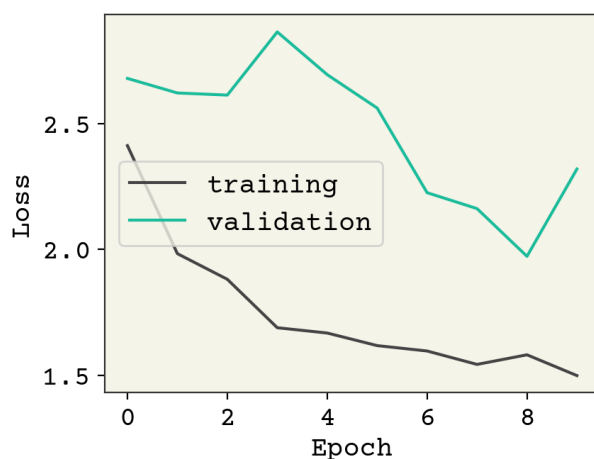
Ouf, c'est beaucoup. Nous pouvons maintenant procéder à notre division habituelle de l'ensemble de données.

```
test_data = data.take(200)
val_data = data.skip(200).take(200)
train_data = data.skip(400)
```

Et enfin, le temps de s'entraîner.

```
model.compile("adam", loss="mean_squared_error")
result = model.fit(train_data.batch(1), validation_data=val_data.batch(1),
    epochs=10)
```

```
plt.plot(result.history["loss"], label="training")
plt.plot(result.history["val_loss"], label="validation")
plt.legend()
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.show()
```



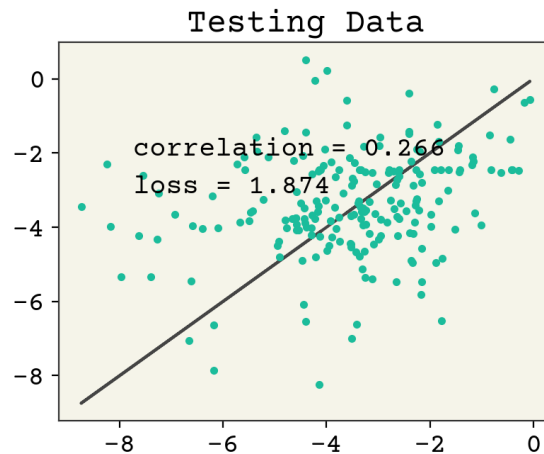
Ce modèle est définitivement sous-équipé. L'une des raisons est que notre taille de lot est de 1. C'est un effet secondaire de rendre le nombre d'atomes variable, puis Keras/tensorflow a du mal à regrouper nos données s'il y a deux dimensions inconnues. Une astuce standard consiste à regrouper plusieurs molécules dans un seul graphique, mais en s'assurant qu'elles sont déconnectées (pas de liaisons entre les molécules). Cela vous permet de grouper des molécules sans augmenter le rang de votre modèle/données.

Vérifions maintenant le diagramme de parité.

```

yhat = model.predict(test_data.batch(1), verbose=0)[: , 0]
test_y = [y for x, y in test_data]
plt.figure()
plt.plot(test_y, test_y, "-")
plt.plot(test_y, yhat, ".")
plt.text(
    min(test_y) + 1,
    max(test_y) - 2,
    f"correlation = {np.corrcoef(test_y, yhat)[0,1]:.3f}",
)
plt.text(
    min(test_y) + 1,
    max(test_y) - 3,
    f"loss = {np.sqrt(np.mean((test_y - yhat)**2)):.3f}",
)
plt.title("Testing Data")
plt.show()

```



8.6. Message passant le point de vue

Une façon de voir plus largement une couche GCN est qu'il s'agit d'une sorte de couche de « transmission de messages ». Vous calculez d'abord un message provenant de chaque nœud voisin :

$$\vec{e}_{s_i,j} = \vec{v}_{s_i,j} \mathbf{W} \quad (8.4)$$

où $\vec{v}_{s_i,j}$ signifie le j ème voisin du nœud i . Les s_i désigne les expéditeurs vers i . C'est ainsi qu'un GCN calcule les messages, c'est juste une matrice de poids multipliée par chaque fonctionnalité de nœud voisin. Après avoir reçu les messages qui iront au nœud i , $\vec{e}_{s_i,j}$, on les agrège à l'aide d'une fonction invariante par permutation à l'ordre des voisins :

$$\vec{e}_i = \frac{1}{|\vec{e}_{s_i,j}|} \sum \vec{e}_{s_i,j} \quad (8.5)$$

Dans le GCN, cette agrégation n'est qu'une moyenne, mais il peut s'agir de n'importe quelle fonction invariante de permutation (éventuellement entraînable). Enfin, nous mettons à jour notre nœud en utilisant le message agrégé dans le GCN :

$$\vec{v}'_i = \sigma(\vec{e}_i) \quad (8.6)$$

où \vec{v}'_i indique les nouvelles fonctionnalités du nœud. Il s'agit simplement du message agrégé activé. En l'écrivant de cette façon, vous pouvez voir comment il est possible d'apporter de petits changements. Un article important de Gilmer et al. a exploré certains de ces choix et décrit comment cette idée générale de couches de transmission de messages réussit bien à apprendre à prédire les énergies moléculaires à partir de la mécanique quantique [[GSR+17](#)]. Des exemples de modifications apportées aux équations GCN ci-dessus consistent à inclure des informations de bord lors du calcul des messages voisins ou à utiliser une couche de réseau neuronal dense à la place de σ . Vous pouvez considérer le GCN comme un type d'une classe plus large de réseaux neuronaux de graphes de transmission de messages, parfois abrégés en MPNN.

8.7. Réseau de neurones à graphes fermés

Une variante courante de la couche de transmission de messages est le **réseau de neurones à graphes fermés** (GGN) [[LTBZ15](#)]. Il remplace la dernière équation, la mise à jour du nœud, par

$$\vec{v}'_i = \text{GRU}(\vec{v}_i, \vec{e}_i) \quad (8.7)$$

où le $\text{GRU}(\cdot, \cdot)$ est une unité récurrente fermée [[CGCB14](#)]. Un GRU est un réseau neuronal binaire (deux arguments d'entrée) généralement utilisé dans la modélisation de séquences. La propriété intéressante d'un GGN par rapport à un GCN est qu'il a des paramètres entraînables dans la mise à jour du nœud (à partir du GRU), donnant au modèle un peu plus de flexibilité. Dans un GGN, les paramètres GRU sont conservés les mêmes à chaque couche, comme la façon dont un GRU est utilisé pour modéliser des séquences. Ce qui est bien, c'est que vous pouvez empiler des couches GGN infinies sans augmenter le nombre de paramètres pouvant être entraînés (en supposant que vous fassiez **W** le même à chaque couche). Ainsi, les GGN conviennent aux grands graphiques, comme une grande protéine ou une grande cellule unitaire.

8.8. Mise en commun

Du point de vue de la transmission des messages, et en général pour le GNNS, la façon dont les messages des voisins sont combinés est une étape clé. Ceci est parfois appelé **pooling**, car il est similaire à la couche de pooling utilisée dans les réseaux de neurones convolutifs. Tout comme dans la mise en commun des réseaux de neurones convolutifs, vous pouvez utiliser plusieurs opérations de réduction. Généralement, vous voyez une somme ou une réduction moyenne des GNN, mais vous pouvez être assez sophistiqué comme dans les réseaux d'isomorphisme de graphes [[XHLJ18](#)]. Nous verrons un exemple dans notre chapitre sur l'attention sur l'utilisation de l'auto-attention, qui peut également être utilisée pour la mise en commun. Il peut être tentant de se concentrer sur cette étape, mais il a été constaté empiriquement que le choix de la mutualisation n'est pas si important [[LDLio19](#), [MSK20](#)]. La propriété clé de la mise en commun est *l'invariance* de permutation - nous voulons que l'opération d'agrégation ne dépende pas de l'ordre des nœuds (ou des arêtes si elles sont regroupées). Vous pouvez trouver une revue récente des méthodes de mise en commun dans Grattarola et al. [[GZBA21](#)].

Vous pouvez voir une comparaison plus visuelle et un aperçu des différentes stratégies de mise en commun dans cet article distillé de Daigavane et al. [[DRA21](#)].

Vous verrez souvent le préfixe "gated" sur les GNN et cela signifie que les nœuds sont mis à jour en fonction d'un GRU.

8.9. Fonction de lecture

GNNs output a graph by design. It is rare that our labels are graphs - typically we have node labels or a single graph label. An example of a node label is partial charge of atoms. An example of a graph label would be the energy of the molecule. The process of converting the graph output from the GNN into our predicted node labels or graph label is called the **readout**. If we have node labels, we can simply discard the edges and use our output node feature vectors from the GNN as the prediction, perhaps with a few dense layers before our predicted output label.

Si nous essayons de prédire une étiquette au niveau du graphique comme l'énergie de la molécule ou la charge nette, nous devons être prudents lors de la conversion des caractéristiques nœud/arête en une étiquette de graphique. Si nous plaçons simplement les entités de nœud dans une couche dense pour obtenir l'étiquette de graphique de forme souhaitée, nous perdrons l'équivariance de permutation (techniquement, c'est maintenant l'invariance de permutation puisque notre sortie est une étiquette de graphique, pas des étiquettes de nœud). La lecture que nous avons faite ci-dessus dans l'exemple de solubilité était une réduction sur les caractéristiques des nœuds pour obtenir

une caractéristique graphique. Ensuite, nous avons utilisé cette fonctionnalité graphique dans des couches denses. Il s'avère que c'est le seul moyen [ZKR+17] pour effectuer une lecture de caractéristiques graphiques : une réduction sur les nœuds pour obtenir une caractéristique graphique, puis des couches denses pour obtenir une étiquette de graphe prédite à partir de ces caractéristiques graphiques. Vous pouvez également créer des couches denses sur les entités de nœud individuellement, mais cela se produit déjà dans GNN, donc je ne le recommande pas. Cette lecture est parfois appelée DeepSets car elle a la même forme que l'architecture DeepSets, qui est une architecture invariante de permutation pour les fonctionnalités qui sont des ensembles [ZKR+17].

Vous remarquerez peut-être que le regroupement et les lectures utilisent tous deux des fonctions invariantes de permutation. Ainsi, les DeepSets peuvent être utilisés pour la mise en commun et l'attention peut être utilisée pour les lectures.

8.9.1. Intensif vs Extensif

Une considération importante d'une lecture en régression est de savoir si vos étiquettes sont **intensives** ou **extensives**. Une étiquette intensive est une étiquette dont la valeur est indépendante du nombre de nœuds (ou d'atomes). Par exemple, l'indice de réfraction ou la solubilité sont intensifs. La lecture d'une étiquette intensive devrait (généralement) être indépendante du nombre de nœuds/atomes. Ainsi, la réduction de la lecture pourrait être une moyenne ou un maximum, mais pas une somme. En revanche, une étiquette extensive devrait (généralement) utiliser une somme pour la réduction de la lecture. Un exemple d'une propriété moléculaire étendue est l'enthalpie de formation.

8.10. Équations générales de Battaglia

Comme vous pouvez le constater, les couches de transmission de messages sont un moyen général de visualiser les couches GNN. Battaglia *et al.* [BHB+18] est allé plus loin et a créé un ensemble général d'équations qui capture presque tous les GNN. Ils ont divisé les équations de la couche GNN en 3 équations de mise à jour, comme l'équation de mise à jour des nœuds que nous avons vue dans les équations de la couche de transmission de messages, et 3 équations d'agrégation (6 équations au total). Il y a un nouveau concept dans ces équations : les vecteurs de caractéristiques graphiques. Au lieu d'avoir deux parties sur votre réseau (GNN puis lecture), une fonctionnalité de niveau graphique est mise à jour à chaque couche GNN. Le vecteur de caractéristiques du graphe est un ensemble de caractéristiques qui représentent l'ensemble du graphe ou de la molécule. Par exemple, lors du calcul de la solubilité, il peut avoir été utile de créer un vecteur de caractéristiques par molécule qui est finalement utilisé pour calculer la solubilité au lieu d'avoir la lecture. Tout type de quantité par molécule, comme l'énergie, doit être prédit avec le vecteur de caractéristiques au niveau du graphique.

La première étape de ces équations consiste à mettre à jour les vecteurs de caractéristiques de bord, écrits sous la forme \vec{e}_k , que nous n'avons pas encore vu :

$$\vec{e}'_k = \phi^e(\vec{e}_k, \vec{v}_{rk}, \vec{v}_{sk}, \vec{u}) \quad (8.8)$$

où \vec{e}_k est le vecteur caractéristique du bord k , \vec{v}_{rk} est le vecteur de caractéristiques du nœud récepteur pour le bord k , \vec{v}_{sk} est le vecteur de caractéristiques du nœud d'envoi pour le bord k , \vec{u} est le vecteur caractéristique du graphe, et ϕ^e est l'une des trois fonctions de mise à jour qui définissent la couche GNN. Notez qu'il s'agit d'expressions générales et que vous définissez ϕ^e pour votre couche GNN spécifique.

Nos graphes moléculaires ne sont pas orientés, alors comment décidons-nous quel nœud reçoit \vec{v}_{rk} et quel nœud envoie \vec{v}_{sk} ? L'individu \vec{e}'_k sont agrégées à l'étape suivante comme toutes les entrées dans le nœud \vec{v}_{rk} . Dans notre graphe moléculaire, toutes les liaisons sont à la fois des "entrées" et des "sorties" d'un atome (comment pourrait-il en être autrement ?), il est donc logique de simplement considérer chaque liaison comme deux

arêtes dirigées : une liaison CH a une arête de C à H et une arête de H à C. En fait, nos matrices d'adjacence reflètent déjà cela. Il y a deux éléments non nuls pour chaque liaison : un pour C à H et un pour H à C. Revenons à la question initiale, qu'est-ce que \vec{v}_{rk} et \vec{v}_{sk} ? Nous considérons chaque élément de la matrice de contiguïté (chaque k) et quand nous sommes sur l'élément $k = \{ij\}$, lequel est A_{ij} , alors le nœud récepteur est j et le nœud émetteur est i . Lorsque nous considérons le bord compagnon A_{ji} , le nœud récepteur est i et le nœud émetteur est j .

\vec{e}'_k est comme le message du GCN. Sauf que c'est plus général : cela peut dépendre du nœud de réception et du vecteur de caractéristiques du graphe \vec{u} . La métaphore d'un « message » ne s'applique pas tout à fait, puisqu'un message ne peut pas être affecté par le récepteur. Quoi qu'il en soit, les nouvelles mises à jour de bord sont ensuite agrégées avec la première fonction d'agrégation :

$$\vec{e}'_i = \rho^{e \rightarrow v} \left(E'_i \right) \quad (8.9)$$

où $\rho^{e \rightarrow v}$ est notre fonction définie et E'_i représente l'empilement de tous \vec{e}'_k des bords au nœud i . Ayant nos arêtes agrégées, nous pouvons calculer la mise à jour du nœud :

$$\vec{v}'_i = \phi^v \left(\vec{e}'_i, \vec{v}_i, \vec{u} \right) \quad (8.10)$$

Ceci conclut les étapes habituelles d'une couche GNN car nous avons de nouveaux nœuds et de nouvelles arêtes. Si vous mettez à jour les fonctionnalités du graphique (\vec{u}), les étapes supplémentaires suivantes peuvent être définies :

$$\vec{e}' = \rho^{e \rightarrow u} \left(E' \right) \quad (8.11)$$

Cette équation agrège tous les messages/arêtes agrégées sur l'ensemble du graphique. Ensuite, nous pouvons agréger les nouveaux nœuds sur l'ensemble du graphique :

$$\vec{v}' = \rho^{v \rightarrow u} \left(V' \right) \quad (8.12)$$

Enfin, nous pouvons calculer la mise à jour du vecteur de caractéristiques du graphe comme :

$$\vec{u}' = \phi^u \left(\vec{e}', \vec{v}', \vec{u} \right) \quad (8.13)$$

8.10.1. Reformuler GCN en équations de Battaglia

Voyons comment le GCN est présenté sous cette forme. Nous calculons d'abord nos messages de voisinage pour tous les voisins possibles en utilisant (8.8). N'oubliez pas que dans le GCN, les messages ne dépendent que des expéditeurs.

$$\vec{e}'_k = \phi^e \left(\vec{e}_k, \vec{v}_{rk}, \vec{v}_{sk}, \vec{u} \right) = \vec{v}_{sk} \mathbf{W}$$

Pour agréger nos messages entrant dans i dans (8.9), on les moyenne.

$$\vec{e}'_i = \rho^{e \rightarrow v} \left(E'_i \right) = \frac{1}{|E'_i|} \sum E'_i$$

Notre node update est alors l'activation (8.10)

$$\vec{v}'_i = \phi^v \left(\vec{e}'_i, \vec{v}_i, \vec{u} \right) = \sigma(\vec{e}'_i)$$

nous pourrions inclure l'auto-boucle ci-dessus en utilisant $\sigma(\vec{e}'_i + \vec{v}_i)$. Les autres fonctions ne sont pas utilisées dans un GCN, donc ces trois-là définissent complètement le GCN.

Même si nous utilisons la fonction "mise à jour des bords", rappelez-vous que dans un GCN, nous ignorons les caractéristiques des bords. Nous ne nous soucions que des arêtes pour définir la connectivité du graphe.

8.11. L'architecture SchNet

L'un des GNN les plus anciens et les plus populaires est le réseau SchNet [[SchüttSK+18](#)]. Il n'était pas vraiment reconnu au moment de la publication comme un GNN, mais il est maintenant reconnu comme tel et vous le verrez souvent utilisé comme modèle de base. Un modèle **de base** est un modèle bien accepté et précis auquel on compare.

SchNet est pour les atomes représentés sous forme de coordonnées xyz (points) - et non sous forme de graphique moléculaire. Tous nos exemples précédents utilisaient le graphe moléculaire sous-jacent comme entrée. Dans SchNet, nous convertirons nos coordonnées xyz en un graphique, afin que nous puissions appliquer un GNN. SchNet a été développé pour prédire les énergies et les forces à partir de configurations d'atomes sans information sur les liaisons. Ainsi, nous devons d'abord voir comment un ensemble d'atomes et leurs positions sont convertis en un graphe. Pour obtenir les nœuds, nous effectuons un processus similaire à celui ci-dessus et le numéro atomique est transmis à travers une couche d'intégration, ce qui signifie simplement que nous attribuons un vecteur entraînable à chaque numéro atomique (voir Couches standard [pour](#) un examen des intégrations).

Obtenir la matrice d'adjacence est également simple : nous faisons simplement en sorte que chaque atome soit connecté à chaque atome. L'intérêt d'utiliser un GNN peut sembler déroutant, si nous ne faisons que tout connecter. *C'est parce que les GNN sont équivariants de permutation*. Si nous essayions d'apprendre sur les atomes sous forme de coordonnées xyz, nous aurions des poids dépendant de l'ordre des atomes et ne parviendrions probablement pas à gérer différents nombres d'atomes.

Il manque encore un détail : où vont les coordonnées xyz ? Nous faisons dépendre le modèle des coordonnées xyz en construisant les entités de bord à partir des coordonnées xyz. Le bord entre les atomes i et j est calculé uniquement à partir de leur distance r :

$$e_k = \exp\left(-\gamma(r - \mu_k)^2\right) \quad (8.14)$$

où γ est un hyperparamètre (par exemple, 10\AA) et μ_k est une grille de scalaires à espace égal - comme . Le but de (8.14) est similaire à la transformation d'une caractéristique de catégorie comme le numéro atomique ou le type de liaison covalente en un vecteur one-hot. Cependant, nous ne pouvons pas créer de vecteur à chaud, car il existe un nombre infini de distances possibles. Ainsi, on a une sorte de « lissage » qui nous donne un pseudo one-hot pour la distance. Voyons un exemple pour en avoir une idée : [0, 5, 10, 15, 20]

```
gamma = 1
mu = np.linspace(0, 10, 5)

def rbf(r):
    return np.exp(-gamma * (r - mu) ** 2)

print("input", 2)
print("output", np.round(rbf(2), 2))
```

```
input 2
output [0.02 0.78 0.    0.    0. ]
```

Vous pouvez voir qu'une distance de $r = 2$ donne un vecteur avec la plupart de l'activation pour le $k = 1$ position - qui correspond à $\mu_1 = 2$.

Nous avons nos nœuds et nos arêtes et sommes sur le point de définir les équations de mise à jour GNN. Nous avons besoin d'un peu plus de notation. Je vais utiliser $h(\vec{x})$ pour indiquer un perceptron multicouche (MLP) - essentiellement un réseau de neurones à 1 à 2 couches denses. Le nombre exact de couches denses et quand/où l'activation est utilisée dans ces MLP sera défini dans la mise en œuvre, car ce n'est pas si important pour la compréhension. Rappelons que la définition d'une couche dense est

Modèles de base

Une sagesse commune est que si vous voulez résoudre un vrai problème avec l'apprentissage en profondeur, vous devriez lire l'article populaire le plus récent dans un domaine et utiliser la référence à laquelle ils se comparent au lieu de leur modèle proposé. La raison en est qu'un modèle de référence doit généralement être simple, rapide et bien testé, ce qui est généralement plus important que d'être le plus précis.

$$h(\vec{x}) = \sigma(Wx + b)$$

Nous utiliserons également une fonction d'activation différente appelée "softplus décalé" dans SchNet : $\ln(0.5e^x + 0.5)$. Tu peux voir $\sigma(x)$ par rapport à l'activation ReLU habituelle de la Fig. 8.4. La justification de l'utilisation de softplus décalé est qu'il est lisse par rapport à son entrée, il pourrait donc être utilisé pour calculer des forces dans une simulation de dynamique moléculaire qui nécessite de prendre des dérivées lisses par rapport aux distances par paires.

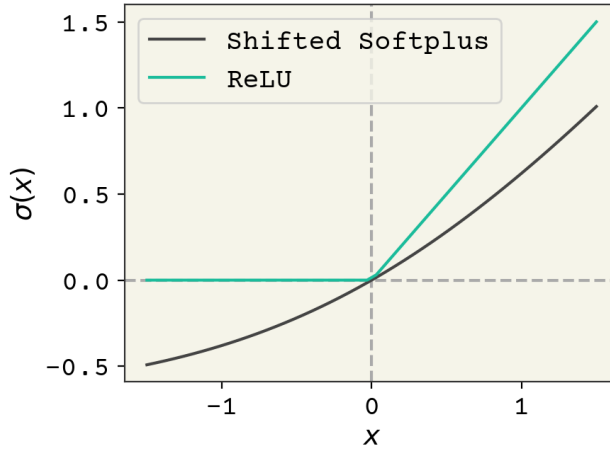


Fig. 8.4 Comparaison de la fonction d'activation ReLU usuelle et du softplus décalé utilisé dans le modèle SchNet.

Maintenant, les équations GNN ! L'équation de mise à jour des bords (8.8) est composée de deux parties. Tout d'abord, nous exécutons la fonction de bord entrant via un MLP et les atomes via un MLP. Ensuite, le résultat est exécuté via un MLP :

$$\vec{e}'_k = \phi^e(\vec{e}_k, \vec{v}_{rk}, \vec{v}_{sk}, \vec{u}) = h_1(\vec{v}_{sk}) \cdot h_2(\vec{e}_k)$$

L'équation suivante est l'équation d'agrégation des arêtes, (8.9). Pour SchNet, l'agrégation des arêtes est une somme sur les caractéristiques des atomes voisins.

$$\vec{e}'_i = \sum E'_i$$

Enfin, l'équation de mise à jour du nœud pour SchNet est :

$$\vec{v}'_i = \phi^v(\vec{e}'_i, \vec{v}_i, \vec{u}) = \vec{v}_i + h_3(\vec{e}'_i)$$

Les mises à jour GNN sont généralement appliquées 3 à 6 fois. Bien que nous ayons une équation de mise à jour des bords, comme dans GCN, nous ne remplaçons pas les bords et les gardons les mêmes à chaque couche. Le SchNet original était destiné à prédire les énergies et les forces, de sorte qu'une lecture peut être effectuée en utilisant la mise en commun des sommes ou toute autre stratégie décrite ci-dessus.

Celles-ci sont parfois modifiées, mais dans l'article original de SchNet h_1 est une couche dense sans activation, h_2 est deux couches denses avec activation, et h_3 c'est 2 couches denses avec activation sur la première et non sur la seconde.

i Qu'est-ce que SchNet ?

La fonctionnalité GNN clé d'un GNN de type SchNet est (1) utiliser les fonctionnalités de bord et de nœud dans la mise à jour du bord (construction du message) :

$$\vec{e}'_k = h_1(\vec{v}_{sk}) \cdot h_2(\vec{e}_k)$$

où $h_i()$ s sont des fonctions pouvant être entraînées et (2) utilisent un résidu dans la mise à jour du nœud :

$$\vec{v}'_i = \vec{v}_i + h_3(\vec{e}'_i)$$

Tous les autres détails sur la façon de caractériser les bords, la profondeur h_i , c'est-à-dire quelle activation choisir, comment lire et comment convertir les nuages de points en graphiques concernent le modèle SchNet spécifique dans [[SchuttSK+18](#)] .

8.12. Exemple SchNet : Prédire les groupes d'espace

Notre prochain exemple sera un modèle SchNet qui prédit des groupes spatiaux de points. L'identification du groupe spatial d'atomes est une partie importante de l'identification de la structure cristalline et des simulations de cristallisation. Notre modèle SchNet prendra comme points d'entrée et produira le groupe d'espace prédit. Il s'agit d'un problème de classification ; en particulier, il est multi-classe car un ensemble de points ne doit appartenir qu'à un seul groupe d'espace. Pour simplifier nos tracés et notre analyse, nous allons travailler en 2D où il y a 17 groupes d'espaces possibles.

Nos données pour cela sont un ensemble de points de divers groupes de points. Les entités sont des coordonnées xyz et l'étiquette est le groupe d'espace. Nous n'aurons pas plusieurs types d'atomes pour ce problème. La cellule masquée ci-dessous charge les données et les remodèle pour l'exemple.

```

import gzip
import pickle
import urllib

urllib.request.urlretrieve(
    "https://github.com/whitead/dmol-book/raw/main/data/sym_trajs.pb.gz",
    "sym_trajs.pb.gz",
)
with gzip.open("sym_trajs.pb.gz", "rb") as f:
    trajs = pickle.load(f)

label_str = list(set([k.split("-")[0] for k in trajs]))

# now build dataset
def generator():
    for k, v in trajs.items():
        ls = k.split("-")[0]
        label = label_str.index(ls)
        traj = v
        for i in range(traj.shape[0]):
            yield traj[i], label

data = tf.data.Dataset.from_generator(
    generator,
    output_signature=(
        tf.TensorSpec(shape=(None, 2), dtype=tf.float32),
        tf.TensorSpec(shape=(), dtype=tf.int32),
    ),
).shuffle(
    1000,
    reshuffle_each_iteration=False, # do not change order each time (!)
    otherwise will contaminate
)

# The shuffling above is really important because this dataset is in order
# of labels!

val_data = data.take(100)
test_data = data.skip(100).take(100)
train_data = data.skip(200)

```

Jetons un coup d'œil à quelques exemples de l'ensemble de données

i Les données

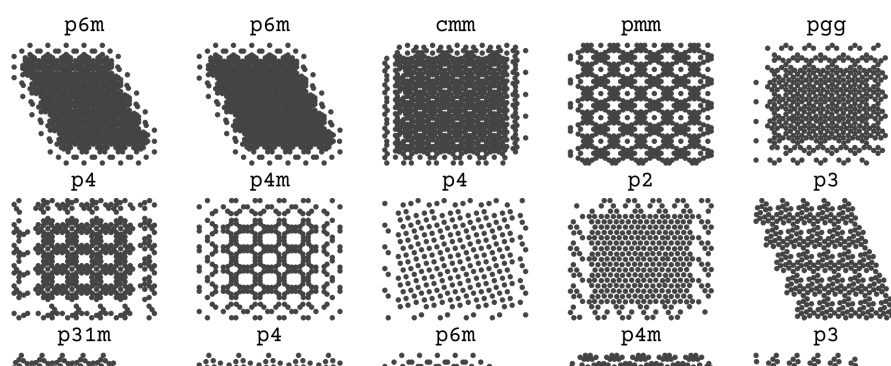
Ces données ont été générées à partir de [[CW22](#)] et tous les points sont contraints de correspondre exactement au groupe spatial lors d'une simulation de dynamique moléculaire. Les trajectoires étaient NPT avec une pression positive et suivaient la procédure de cet article pour la figure 2. Le champ de force est Lennard-Jones avec $\sigma = 1$ et $\epsilon = 1$

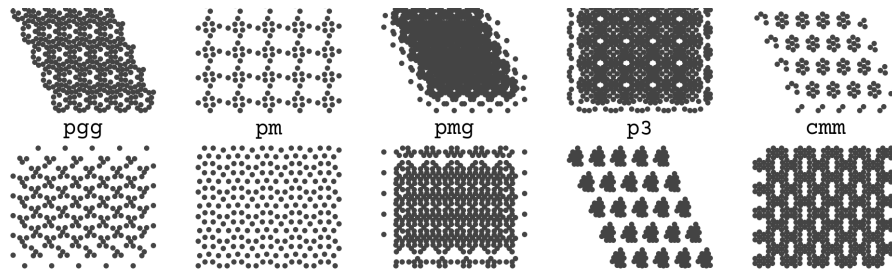
```

fig, axs = plt.subplots(4, 5, figsize=(12, 8))
axs = axs.flatten()

# get a few example and plot them
for i, (x, y) in enumerate(data):
    if i == 20:
        break
    axs[i].plot(x[:, 0], x[:, 1], ".")
    axs[i].set_title(label_str[y.numpy()])
    axs[i].axis("off")

```





Vous pouvez voir qu'il y a un nombre variable de points et quelques exemples pour chaque groupe d'espace. Le but est de déduire ces titres sur l'intrigue à partir des seuls points.

8.12.1. Construire les graphiques

Nous devons maintenant construire les graphiques pour les points. Les nœuds sont tous identiques - ils ne peuvent donc être que des 1 (nous réserverons 0 au cas où nous voudrions masquer ou remplir à un moment donné dans le futur). Comme décrit dans la section SchNet ci-dessus, les arêtes doivent être distantes de tous les autres atomes. Dans la plupart des implémentations de SchNet, nous ajoutons pratiquement une coupure sur la distance ou le degré maximum (arêtes par nœud). Nous ferons un degré maximum pour ce travail de 16.

J'ai une fonction ci-dessous qui est un peu sophistiquée. Il prend une matrice de positions de points dans une dimension arbitraire et renvoie les distances et les indices aux k voisins les plus proches - exactement ce dont nous avons besoin. Il utilise quelques astuces de [Tensors and Shapes](#). Cependant, il n'est pas si important que vous compreniez cette fonction. Sachez simplement que cela prend des points et nous donne les fonctionnalités de bord et les nœuds de bord.

```
# this decorator speeds up the function by "compiling" it (tracing it)
# to run efficiently
@tf.function(
    reduce_retracing=True,
)
def get_edges(positions, NN, sorted=True):
    M = tf.shape(input=positions)[0]
    # adjust NN
    NN = tf.minimum(NN, M)
    qexpand = tf.expand_dims(positions, 1) # one column
    qTexpand = tf.expand_dims(positions, 0) # one row
    # repeat it to make matrix of all positions
    qtile = tf.tile(qexpand, [1, M, 1])
    qTtile = tf.tile(qTexpand, [M, 1, 1])
    # subtract them to get distance matrix
    dist_mat = qTtile - qtile
    # mask distance matrix to remove zros (self-interactions)
    dist = tf.norm(tensor=dist_mat, axis=2)
    mask = dist >= 5e-4
    mask_cast = tf.cast(mask, dtype=dist.dtype)
    # make masked things be really far
    dist_mat_r = dist * mask_cast + (1 - mask_cast) * 1000
    topk = tf.math.top_k(-dist_mat_r, k=NN, sorted=sorted)
    return -topk.values, topk.indices
```

Voyons comment cette fonction fonctionne en montrant les connexions entre les points dans l'un de nos exemples. J'ai caché le code ci-dessous. Il montre les voisins de certains points et les relie afin que vous puissiez avoir une idée de la façon dont un ensemble de points est converti en graphique. Le graphique complet contiendra les voisinages de tous les points.

```

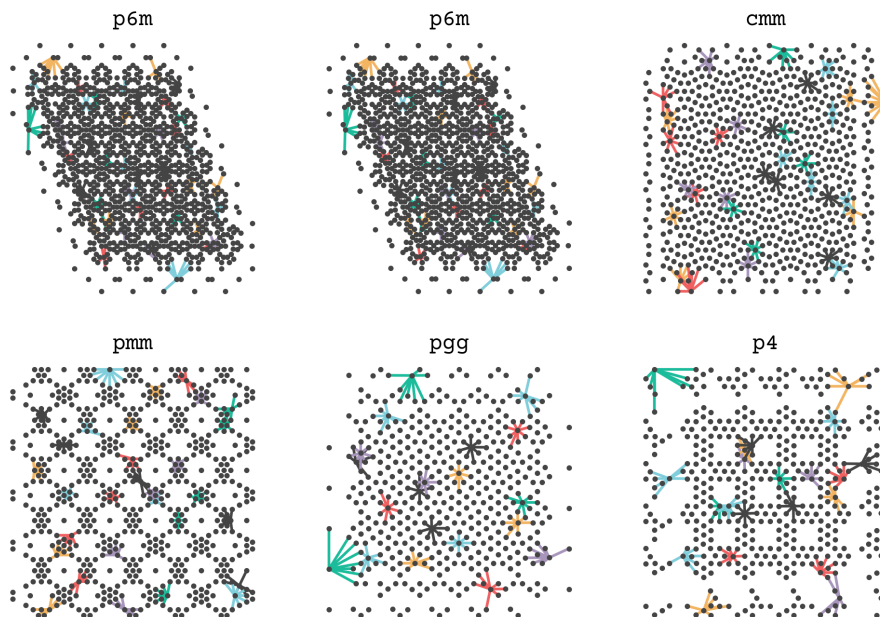
from matplotlib import collections

fig, axs = plt.subplots(2, 3, figsize=(12, 8))
axs = axs.flatten()
for i, (x, y) in enumerate(data):
    if i == 6:
        break
    e_f, e_i = get_edges(x, 8)

    # make things easier for plotting
    e_i = e_i.numpy()
    x = x.numpy()
    y = y.numpy()

    # make lines from origin to its neighbors
    lines = []
    colors = []
    for j in range(0, x.shape[0], 23):
        # lines are [(xstart, ystart), (xend, yend)]
        lines.extend([(x[j, 0], x[j, 1]), (x[k, 0], x[k, 1])] for k in
e_i[j]))
        colors.extend([f"C{j}"] * len(e_i[j]))
    lc = collections.LineCollection(lines, linewidths=2, colors=colors)
    axs[i].add_collection(lc)
    axs[i].plot(x[:, 0], x[:, 1], ".")
    axs[i].axis("off")
    axs[i].set_title(label_str[y])
plt.show()

```



Nous allons maintenant ajouter cette fonction et la caractérisation des arêtes de SchNet [\(8.14\)](#) pour obtenir les graphiques des étapes GNN.

```

MAX_DEGREE = 16
EDGE_FEATURES = 8
MAX_R = 20

gamma = 1
mu = np.linspace(0, MAX_R, EDGE_FEATURES)

def rbf(r):
    return tf.exp(-gamma * (r[..., tf.newaxis] - mu) ** 2)

def make_graph(x, y):
    edge_r, edge_i = get_edges(x, MAX_DEGREE)
    edge_features = rbf(edge_r)
    return (tf.ones(tf.shape(x)[0], dtype=tf.int32), edge_features, edge_i),
y[None]

graph_train_data = train_data.map(make_graph)
graph_val_data = val_data.map(make_graph)
graph_test_data = test_data.map(make_graph)

```

Examinons un graphique pour voir à quoi il ressemble. Nous ne découperons que les premiers nœuds.

```
for (n, e, nn), y in graph_train_data:
    print("first node:", n[1].numpy())
    print("first node, first edge features:", e[1, 1].numpy())
    print("first node, all neighbors", nn[1].numpy())
    print("label", y.numpy())
    break
```

```
first node: 1
first node, first edge features: [4.87925738e-01 1.75932534e-02 5.15009597e-11
1.22395275e-26
 0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
first node, all neighbors [ 3  2 12 16  4  8  0  9 23 15 353 20 361
223 21 10]
label [1]
```

8.12.2. Mise en œuvre des MLP

Nous pouvons maintenant implémenter le modèle SchNet ! Commençons par le h_1, h_2, h_3 MLP utilisés dans les équations de mise à jour GNN. Dans l'article de SchNet, chacun avait un nombre différent de couches et des décisions différentes sur les couches activées. Créons-les maintenant.

```
def ssp(x):
    # shifted softplus activation
    return tf.math.log(0.5 * tf.math.exp(x) + 0.5)

def make_h1(units):
    return tf.keras.Sequential([tf.keras.layers.Dense(units)])

def make_h2(units):
    return tf.keras.Sequential(
        [
            tf.keras.layers.Dense(units, activation=ssp),
            tf.keras.layers.Dense(units, activation=ssp),
        ]
    )

def make_h3(units):
    return tf.keras.Sequential(
        [tf.keras.layers.Dense(units, activation=ssp),
         tf.keras.layers.Dense(units)]
    )
```

Un détail qui peut être manqué est que les poids dans chaque MLP devraient changer dans chaque couche de SchNet. Ainsi, nous avons écrit les fonctions ci-dessus pour toujours renvoyer un nouveau MLP. Cela signifie qu'un nouvel ensemble de poids entraînaibles est généré à chaque appel, ce qui signifie qu'il est impossible que nous ayons par erreur les mêmes poids dans plusieurs couches.

8.12.3. Mise en œuvre du GNN

Nous avons maintenant toutes les pièces pour faire le GNN. Ce code sera très similaire à l'exemple GCN ci-dessus, sauf que nous avons maintenant des fonctionnalités de bord. Un autre détail est que notre lecture sera également un MLP, à la suite de l'article SchNet. Le seul changement que nous apporterons est que nous voulons que notre propriété de sortie soit (1) une classification multi-classes et (2) intensive (indépendante du nombre d'atomes). Nous terminerons donc par une moyenne (intensive) et terminerons par un vecteur de sortie de logits de la taille de nos étiquettes.

```

class SchNetModel(tf.keras.Model):
    """Implementation of SchNet Model"""

    def __init__(self, gnn_blocks, channels, label_dim, **kwargs):
        super(SchNetModel, self).__init__(**kwargs)
        self.gnn_blocks = gnn_blocks

        # build our layers
        self.embedding = tf.keras.layers.Embedding(2, channels)
        self.h1s = [make_h1(channels) for _ in range(self.gnn_blocks)]
        self.h2s = [make_h2(channels) for _ in range(self.gnn_blocks)]
        self.h3s = [make_h3(channels) for _ in range(self.gnn_blocks)]
        self.readout_l1 = tf.keras.layers.Dense(channels // 2, activation=ssp)
        self.readout_l2 = tf.keras.layers.Dense(label_dim)

    def call(self, inputs):
        nodes, edge_features, edge_i = inputs
        # turn node types as index to features
        nodes = self.embedding(nodes)
        for i in range(self.gnn_blocks):
            # get the node features per edge
            v_sk = tf.gather(nodes, edge_i)
            e_k = self.h1s[i](v_sk) * self.h2s[i](edge_features)
            e_i = tf.reduce_sum(e_k, axis=1)
            nodes += self.h3s[i](e_i)
        # readout now
        nodes = self.readout_l1(nodes)
        nodes = self.readout_l2(nodes)
        return tf.reduce_mean(nodes, axis=0)

```

N'oubliez pas que les attributs clés d'un SchNet GNN sont la façon dont nous utilisons les fonctionnalités de bord et de nœud. Nous pouvons voir le mélange de ces deux dans la ligne clé pour le calcul de la mise à jour des bords (calcul des valeurs de message) :

```
e_k = self.h1s[i](v_sk) * self.h2s[i](edge_features)
```

suivi de l'agrégation des mises à jour des arêtes (messages de pooling) :

```
e_i = tf.reduce_sum(e_k, axis=1)
```

et la mise à jour du nœud

```
nodes += self.h3s[i](e_i)
```

Il convient également de noter comment nous passons des fonctionnalités de nœud aux multi-classes. Nous utilisons des couches denses qui transforment la forme par nœud en nombre de classes

```
self.readout_l1 = tf.keras.layers.Dense(channels // 2, activation=ssp)
self.readout_l2 = tf.keras.layers.Dense(label_dim)
```

puis on prend la moyenne sur tous les nœuds

```
return tf.reduce_mean(nodes, axis=0)
```

Utilisons maintenant le modèle sur certaines données.

```
small_schnet = SchNetModel(3, 32, len(label_str))
```

```
for x, y in graph_train_data:
    yhat = small_schnet(x)
    break
print(yhat.numpy())
```

```

[ 1.7014749e-02  1.2131847e-02 -4.6764631e-03 -1.0873002e-02
  7.4983016e-03  1.2150299e-02  3.1273261e-02  1.7637234e-02
 -2.1602984e-03  2.6902608e-03 -1.9115655e-02 -1.0214287e-02
  1.0792590e-03  1.4398353e-05 -9.2129409e-03  2.0333942e-02
  9.9622970e-03]

```

La sortie est la forme correcte et rappelez-vous qu'il s'agit de logits. Pour obtenir une prédiction de classe qui correspond à la probabilité 1, nous devons utiliser un softmax :

```
print("predicted class", tf.nn.softmax(yhat).numpy())
```

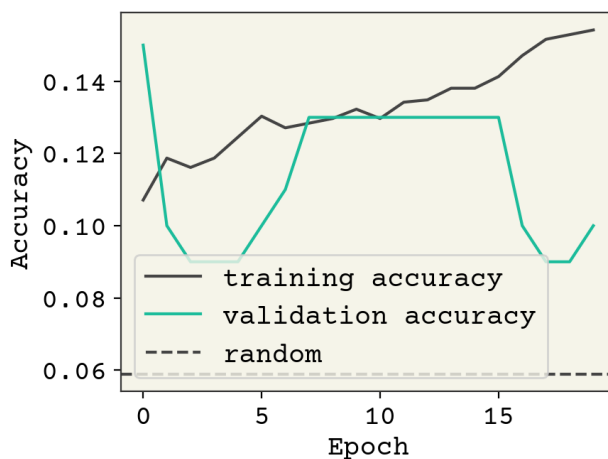
```
predicted class [0.05956277 0.05927264 0.0582847  0.05792465 0.05899863
0.05927373
 0.06041813 0.05959986 0.05843153 0.05871565 0.05744916 0.05796282
 0.05862113 0.05855874 0.05802089 0.0597608  0.05914419]
```

8.12.4. Formation

Super! Il n'est cependant pas formé. Maintenant, nous pouvons mettre en place une formation. Notre perte sera une entropie croisée des logits, mais nous devons faire attention au formulaire. Nos étiquettes sont des nombres entiers - appelés étiquettes "éparses" car elles ne sont pas complètes. La classification multi-classes est également connue sous le nom de classification catégorielle. Ainsi, la perte que nous voulons est une entropie croisée catégorique clairsemée à partir des logits.

```
small_schnet.compile(
    optimizer=tf.keras.optimizers.Adam(1e-4),
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
    metrics="sparse_categorical_accuracy",
)
result = small_schnet.fit(graph_train_data, validation_data=graph_val_data,
epochs=20)
```

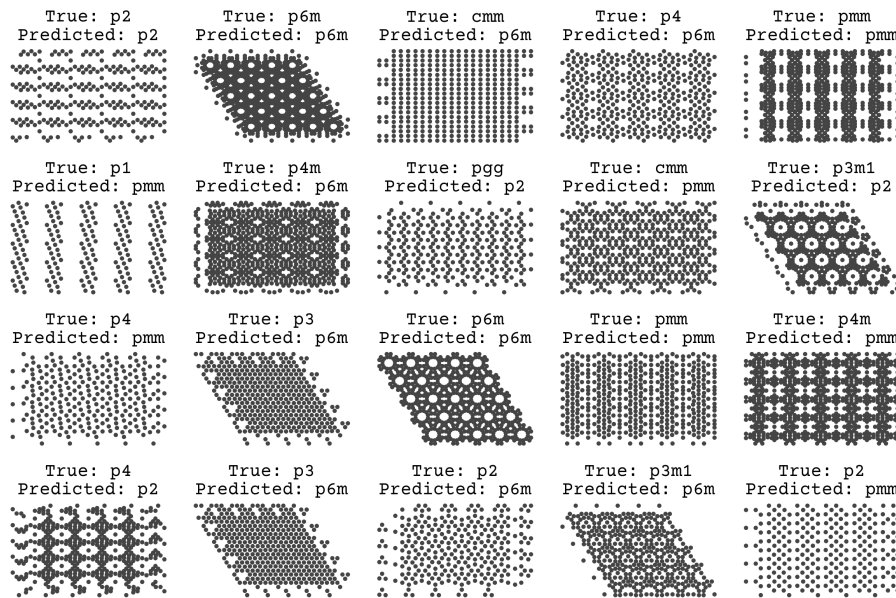
```
plt.plot(result.history["sparse_categorical_accuracy"], label="training
accuracy")
plt.plot(result.history["val_sparse_categorical_accuracy"], label="validation
accuracy")
plt.axhline(y=1 / 17, linestyle="--", label="random")
plt.legend()
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.show()
```



La précision n'est pas excellente, mais il semble que nous pourrions continuer à nous entraîner. Nous avons un très petit SchNet ici. Le SchNet standard décrit dans [SchuttSK+18] utilise 6 couches et 64 canaux et 300 fonctionnalités de bord. Nous avons 3 couches et 32 canaux. Néanmoins, nous sommes en mesure d'apprendre. Voyons visuellement ce qui se passe avec le modèle entraîné sur certaines données de test


```
fig, axs = plt.subplots(4, 5, figsize=(12, 8))
axs = axs.flatten()

for i, ((x, y), (gx, _)) in enumerate(zip(test_data, graph_test_data)):
    if i == 20:
        break
    axs[i].plot(x[:, 0], x[:, 1], ".")
    yhat = small_schnet(gx)
    yhat_i = tf.math.argmax(tf.nn.softmax(yhat)).numpy()
    axs[i].set_title(f"True: {label_str[y.numpy()]} \n Predicted: {label_str[yhat_i]}")
    axs[i].axis("off")
plt.tight_layout()
plt.show()
```



Nous reviendrons sur cet exemple plus tard ! Un fait unique à propos de cet ensemble de données est qu'il est *synthétique*, ce qui signifie qu'il n'y a pas de bruit d'étiquette. Comme discuté dans [Régression et évaluation du modèle](#), cela supprime la possibilité de surajustement et nous conduit à privilégier les modèles à forte variance. Le but d'apprendre à un modèle à prédire des groupes spatiaux est de l'appliquer sur des simulations réelles ou des données de microscopie, qui auront certainement du bruit. Nous aurions pu imiter cela en ajoutant du bruit aux étiquettes dans les données et/ou en supprimant au hasard des atomes pour simuler des défauts. Cela aiderait mieux notre modèle à fonctionner dans un cadre réel.

8.13. Orientations de recherche actuelles

8.13.1. Motifs d'architecture courants et comparaisons

Nous avons maintenant vu les couches GNN, GCN, GGN et les équations de Battaglia généralisées. Vous trouverez des motifs communs dans les architectures, comme le déclenchement, [les couches d'attention](#) et les stratégies de mise en commun. Par exemple, les Gated GNNS (GGN) peuvent être combinés avec la mise en commun de l'attention pour créer des Gated Attention GNN (GAAN) [[ZSX+18](#)]. GraphSAGE est similaire à un GCN mais il échantillonne lors du regroupement, ce qui rend les mises à jour voisines de dimension fixe [[HYL17](#)]. Ainsi, vous verrez le suffixe « sage » lorsque vous échantillonnez sur des voisins lors de la mise en commun. Ceux-ci peuvent tous être représentés dans les équations de Battaglia, mais vous devez connaître ces noms.

L'énorme variété d'architectures a conduit à travailler sur l'identification de la « meilleure » architecture GNN ou la plus générale [[DJL+20](#), [EPBM19](#), [SMBGunnemann18](#)]. Malheureusement, la question de savoir quelle architecture GNN est la meilleure est aussi difficile que "quels problèmes de référence sont les meilleurs ?" Il n'y a donc pas de conclusions consensuelles sur la meilleure architecture. Cependant, ces articles sont

d'excellentes ressources sur la formation, les hyperparamètres et les suppositions de départ raisonnables et je recommande fortement de les lire avant de concevoir votre propre GNN. Il y a eu des travaux théoriques pour montrer que les architectures simples, comme les GCN, ne peuvent pas faire la distinction entre certains graphes simples [[XHLJ18](#)]. L'importance pratique de cela dépend de vos données. En fin de compte, il y a tellement de variété dans les hyperparamètres, les équivariances de données et les décisions de formation que vous devriez réfléchir attentivement à l'importance de l'architecture GNN avant de l'explorer avec trop de profondeur.

8.13.2. Nœuds, arêtes et fonctionnalités

Vous constaterez que la plupart des GNN utilisent l'équation de mise à jour des nœuds dans les équations de Battaglia mais ne mettent pas à jour les arêtes. Par exemple, le GCN mettra à jour les nœuds à chaque couche mais les bords sont constants. Certains travaux récents ont montré que la mise à jour des arêtes peut être importante pour apprendre quand les arêtes ont des informations géométriques, comme si le graphe d'entrée est une molécule et les arêtes sont la distance entre les atomes [[KGrossGunnemann20](#)]. Comme nous le verrons dans le chapitre sur les équivariances ([Input Data & Equivariances](#)), l'une des principales propriétés des réseaux de neurones avec des nuages de points (c'est-à-dire des coordonnées cartésiennes xyz) est d'avoir une équivariance de rotation. [[KGrossGunnemann20](#)] a montré que vous pouvez y parvenir si vous effectuez des mises à jour des bords et encodez les vecteurs de bord à l'aide d'un ensemble de bases équivariantes de rotation avec des harmoniques sphériques et des fonctions de Bessel. Ces types de GNN de mise à jour des bords peuvent être utilisés pour prédire la structure des protéines [[JES+20](#)].

Une autre variante courante des fonctionnalités de nœud consiste à intégrer davantage de fonctionnalités de nœud qu'une simple identité d'élément. Dans de nombreux exemples, vous verrez des gens insérer la valence, la masse élémentaire, l'électronégativité, un peu indiquant si l'atome est dans un anneau, un peu indiquant si l'atome est aromatique, etc. Généralement, ceux-ci sont inutiles, car un modèle devrait pouvoir apprendre l'une de ces caractéristiques qui sont calculées à partir des éléments de graphe et de nœud. Cependant, nous et d'autres avons découvert empiriquement que certains peuvent aider, en indiquant spécifiquement si un atome est dans un anneau [[LWC+20](#)]. Le choix de fonctionnalités supplémentaires à inclure devrait cependant figurer au bas de votre liste de choses à explorer lors de la conception et de l'utilisation des GNN.

8.13.3. Au-delà du passage de message

L'un des thèmes communs de la recherche GNN est d'aller «au-delà de la transmission de messages», où la transmission de messages est la construction, l'agrégation et la mise à jour des nœuds avec des messages. Certains considèrent cela comme impossible – affirmant que tous les GNN peuvent être réfondus en tant que transmission de messages [[Velivckovic22](#)]. Une autre direction consiste à déconnecter le graphique sous-jacent entré dans le GNN et le graphique utilisé pour calculer les mises à jour. Nous avons en quelque sorte vu cela ci-dessus avec SchNet, où nous avons limité le degré maximum de transmission du message. Plus utiles sont des idées comme « soulever » les graphes en objets plus structurés comme des complexes simpliciaux [[BFO+21](#)]. Enfin, vous pouvez également choisir où envoyer les messages au-delà des voisins [[TZK21](#)]. Par exemple, tous les nœuds d'un chemin peuvent communiquer des messages ou tous les nœuds d'une clique.

8.13.4. Avons-nous besoin de graphiques ?

Il est possible de convertir un graphique en chaîne si vous travaillez avec une matrice d'adjacence sans valeurs continues. Les molécules peuvent spécifiquement être converties en une chaîne. Cela signifie que vous pouvez utiliser des couches pour des

séquences/chaînes (par exemple, des réseaux neuronaux récurrents ou des convolutions 1D) et éviter les complexités d'un réseau neuronal graphique. SMILES est un moyen de convertir des graphes moléculaires en chaînes. Avec SMILES, vous ne pouvez pas prédire une quantité par atome et donc un réseau neuronal graphique est nécessaire pour les étiquettes d'atome/liaison. Cependant, le choix est moins clair pour les propriétés par molécule comme la toxicité ou la solubilité. Il n'y a pas de consensus quant à savoir si une représentation graphique ou chaîne / SMILES est meilleure. SMILES peut dépasser certains réseaux de neurones graphiques en précision sur certaines tâches. SMILES est généralement meilleur sur les tâches génératives. Les graphiques battent évidemment SMILES dans les représentations d'étiquettes, car ils ont une granularité de liaisons/arêtes. Nous verrons comment modéliser SOURIRES dans [Apprentissage profond sur les séquences](#), mais c'est une question ouverte de savoir lequel est le meilleur.

8.13.5. Stéréochimie/Molécules chirales

La stéréochimie est fondamentalement une propriété 3D des molécules et n'est donc pas présente dans la liaison covalente. Elle est mesurée expérimentalement en voyant si les molécules font tourner la lumière polarisée et une molécule est dite chirale ou "optiquement active" si on sait expérimentalement qu'elle possède cette propriété. La stéréochimie est la catégorisation de la façon dont les molécules peuvent faire pivoter préférentiellement la lumière polarisée à travers des asymétries par rapport à leurs images miroir. En chimie organique, la majorité de la stéréochimie est constituée d'énantiomères. Les énantiomères sont une « latéralité » autour d'atomes spécifiques appelés centres chiraux qui ont 4 atomes liés différents ou plus. Ceux-ci peuvent être traités dans un graphique en indiquant quels nœuds sont des centres chiraux (nœuds) et quel est leur état ou mélange d'états (racémique). Cela peut être considéré comme une étape de traitement supplémentaire. Les acides aminés et donc toutes les protéines sont des entaniomères avec une seule forme présente. Cette chiralité des protéines signifie que de nombreuses molécules médicamenteuses peuvent être plus ou moins puissantes en fonction de leur stéréochimie.



Fig. 8.5 Il s'agit d'une molécule à stéréochimie axiale. Sa petite hélice peut être à gauche ou à droite.

L'ajout d'étiquettes de nœud ne suffit généralement pas. Les molécules peuvent s'interconvertir entre les stéréoisomères au niveau des centres chiraux grâce à un processus appelé tautomérisation. Il existe également des types de stéréochimie qui ne concernent pas un atome spécifique, comme les rotamères autour d'une liaison. Ensuite, il y a la stéréochimie qui implique plusieurs atomes comme l'hélicène axial. Comme le

montre la [Fig. 8.5](#), la molécule n'a pas de centres chiraux mais est "optiquement active" (expérimentalement mesurée comme étant chirale) en raison de son hélice qui peut être gauche ou droite.

8.14. Résumé du chapitre

- Les molécules peuvent être représentées par des graphiques en utilisant des vecteurs de caractéristiques codés à chaud qui montrent l'identité élémentaire de chaque nœud (atome) et une matrice de contiguïté qui montre les voisins immédiats (atomes liés).
- Les réseaux de neurones graphes sont une catégorie de réseaux de neurones profonds qui ont des graphes comme entrées.
- L'un des premiers GNN est le GCN Kipf & Welling. L'entrée du GCN est le vecteur de caractéristiques de nœud et la matrice de contiguïté, et renvoie le vecteur de caractéristiques de nœud mis à jour. Le GCN est invariant de permutation car il fait la moyenne sur les voisins.
- Un GCN peut être considéré comme une couche de transmission de messages, dans laquelle nous avons des expéditeurs et des destinataires. Les messages sont calculés à partir des nœuds voisins, qui, une fois agrégés, mettent à jour ce nœud.
- Un réseau de neurones à graphes fermés est une variante de la couche de passage de messages, pour laquelle les nœuds sont mis à jour selon une fonction d'unité récurrente fermée.
- L'agrégation des messages est parfois appelée pooling, pour laquelle il existe de multiples opérations de réduction.
- Les GNN génèrent un graphique. Pour obtenir une propriété par atome ou par molécule, utilisez une fonction de lecture. La lecture dépend si votre propriété est intensive ou extensive
- Les équations de Battaglia englobent presque tous les GNN dans un ensemble de 6 équations de mise à jour et d'agrégation.
- Vous pouvez convertir les coordonnées xyz en un graphique et utiliser un GNN comme SchNet

8.15. Exercices

1. Écrivez l'éthanol sous forme de graphique avec des caractéristiques de nœud chaud et un tenseur d'adjacence.
2. Le GCN tel que présenté est supposé être invariant par permutation. Rappelons que l'équation clé GCN est :

$$v'_{il} = \sigma \left(\frac{1}{d_i} e_{ij} v_{jk} w_{lk} \right)$$

où i est le nœud récepteur, j est le nœud d'envoi, k est les entités en entrée, et l est les caractéristiques de sortie. v est les caractéristiques du nœud, v' est mise à jour des fonctionnalités du nœud, e est bord, d est degré, et w sont des paramètres entraîables. Déterminez et montrez si les variations suivantes de l'équation GCN sont invariantes par permutation :

Variante 1 : $v'_{ik} = \sigma \left(\frac{1}{d_i} e_{ij} v_{jk} w_{ik} \right)$

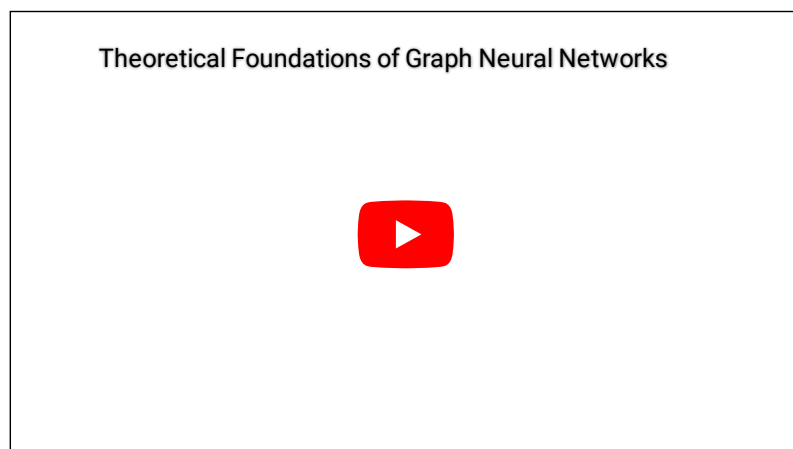
Variante 2 : $v'_{ik} = \sigma \left(\frac{1}{d_i} e_{ij} v_{jk} w_{jk} \right)$

Variante 3 : $v'_{il} = \sigma \left(\frac{1}{d_i} e_{ij} v_{jk} w_{ilk} \right)$

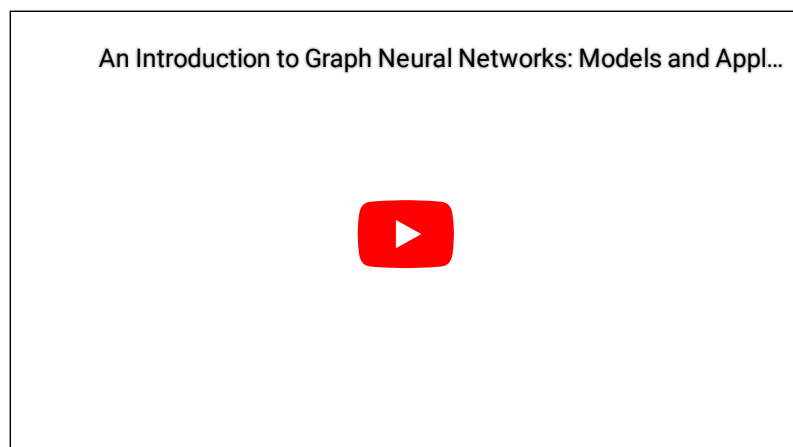
1. Avec les équations de Battaglia, pouvez-vous avoir un GCN "inverse" qui ne met à jour que les entités de bord mais pas les entités de nœud ? Pourrait-il apprendre quelque chose ?
2. Nous voudrions identifier un atome spécifique dans une molécule, comme un site potentiel d'une réaction. Nous avons des données d'entraînement composées de graphiques et de nœuds spécifiques. S'agit-il d'une régression ou d'une classification ? Quelle serait une lecture appropriée et quelle serait la perte correcte ?
3. Proposer une modification au GCN qui prédit les propriétés des bords. Comparez votre réseau avec SchNet - en quoi est-il différent ?
4. Quelle est la relation entre le nombre de couches GCN et le nombre maximum de liaisons entre lesquelles les informations peuvent passer ?
5. Combien de couches faudrait-il pour qu'un GNN détecte si une molécule a un cycle à 6 chaînons ?

8.16. Vidéos pertinentes

8.16.1. Introduction aux GNN



8.16.2. Vue d'ensemble de GNN avec Molecule, Exemples de compilateur



8.17. Références citées

[DJL+20] (1, 2) Vijay Prakash Dwivedi, Chaitanya K Joshi, Thomas Laurent, Yoshua Bengio et Xavier Bresson. Analyse comparative des réseaux de neurones de graphes. préprint arXiv arXiv:2003.00982, 2020.

[BBL+17]

Michael M Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam et Pierre Vandergheynst. Apprentissage profond géométrique : aller au-delà des données euclidiennes. *Magazine de traitement du signal IEEE*, 34(4):18–42, 2017.

[CMP+20] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang et S Yu Philip. Une enquête complète sur les réseaux de neurones graphiques. *Transactions IEEE sur les réseaux de neurones et les systèmes d'apprentissage*, 2020.

[LWC+20] (1, 2) Zhiheng Li, Geemi P Wellawatte, Maghesree Chakraborty, Heta A Gandhi, Chenliang Xu et Andrew D White. Prédiction de cartographie à gros grains basée sur un réseau neuronal graphique. *Science chimique*, 11(35):9524–9531, 2020.

[YCW20] Ziyue Yang, Maghesree Chakraborty et Andrew D White. Prédire les déplacements chimiques avec les réseaux de neurones de graphes. *bioRxiv*, 2020.

[XFLW+19] Tian Xie, Arthur France-Lanord, Yanming Wang, Yang Shao-Horn et Jeffrey C Grossman. Réseaux dynamiques de graphes pour l'apprentissage non supervisé de la dynamique à l'échelle atomique dans les matériaux. *Communications sur la nature*, 10(1):1–9, 2019.

[SLRPW21] Benjamin Sanchez-Lengeling, Emily Reif, Adam Pearce et Alex Wiltchko. Une introduction en douceur aux réseaux de neurones graphiques. *Distill*, 2021. <https://distill.pub/2021/gnn-intro>. doi:10.23915/distill.00033.

[XG18] Tian Xie et Jeffrey C. Grossman. Réseaux de neurones convolutionnels à graphe cristallin pour une prédiction précise et interprétable des propriétés des matériaux. *Phys. Rev. Lett.*, 120:145301, avril 2018. URL : <https://linkaps.org/doi/10.1103/PhysRevLett.120.145301>, doi:10.1103/PhysRevLett.120.145301.

[KW16] Thomas N Kipf et Max Welling. Classification semi-supervisée avec réseaux convolutifs de graphes. *arXiv preprint arXiv:1609.02907*, 2016.

[RSG+17] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals et George E Dahl. Message neuronal passant pour la chimie quantique. *arXiv preprint arXiv:1704.01212*, 2017.

[LTBZ15] Yujia Li, Daniel Tarlow, Marc Brockschmidt et Richard Zemel. Réseaux de neurones à séquence de graphes contrôlés. *arXiv preprint arXiv:1511.05493*, 2015.

[CGCB14] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho et Yoshua Bengio. Évaluation empirique des réseaux de neurones récurrents contrôlés sur la modélisation de séquences. *arXiv preprint arXiv:1412.3555*, 2014.

[XHLJ18] (1, 2) Keyulu Xu, Weihua Hu, Jure Leskovec et Stefanie Jegelka. Quelle est la puissance des réseaux de neurones graphiques ? Dans *Conférence internationale sur les représentations de l'apprentissage*. 2018.

[LDLio19] Enxhell Luzhnica, Ben Day et Pietro Liò. Sur les réseaux de classification de graphes, les jeux de données et les lignes de base. *arXiv preprint arXiv:1905.04682*, 2019.

[MSK20] Diego Mesquita, Amauri Souza et Samuel Kaski. Repenser la mutualisation dans les réseaux de neurones de graphes. *Avancées dans les systèmes de traitement de l'information neuronale*, 2020.

[GZBA21] Daniele Grattarola, Daniele Zambon, Filippo Maria Bianchi et Cesare Alippi. Comprendre la mise en commun dans les réseaux de neurones de graphes. *arXiv preprint arXiv:2110.05292*, 2021.

- [DRA21] Ameya Daigavane, Balaraman Ravindran et Gaurav Aggarwal. Comprendre les convolutions sur les graphes. *Distill* , 2021. <https://distill.pub/2021/understanding-gnns>. [doi:10.23915/distill.00032](https://doi.org/10.23915/distill.00032).
- [ZKR+17] (1, 2) Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Russ R Salakhutdinov et Alexander J Smola. Séries profondes. Dans *les progrès des systèmes de traitement de l'information neuronale* , 3391-3401. 2017.
- [BHB+18] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner et d'autres. Biais inductifs relationnels, apprentissage profond et réseaux de graphes. *arXiv preprint arXiv:1806.01261* , 2018.
- [SchuttSK+18] (1, 2, 3) Kristof T Schütt, Huziel E Sauceda, PJ Kindermans, Alexandre Tkatchenko et KR Müller. Schnet - une architecture d'apprentissage en profondeur pour les molécules et les matériaux. *Le Journal of Chemical Physics* , 148(24):241722, 2018.
- [CW22] Sam Cox et Andrew D White. Dynamique moléculaire symétrique. *arXiv preprint arXiv:2204.01114* , 2022.
- [ZSX+18] Jiani Zhang, Xingjian Shi, Junyuan Xie, Hao Ma, Irwin King et Dit-Yan Yeung. Gaan : réseaux d'attention fermée pour l'apprentissage sur de grands graphes spatio-temporels. *arXiv preprint arXiv:1803.07294* , 2018.
- [HYL17] Will Hamilton, Zhitao Ying et Jure Leskovec. Apprentissage de la représentation inductive sur de grands graphes. Dans *les progrès des systèmes de traitement de l'information neuronale* , 1024-1034. 2017.
- [EPBM19] Federico Errica, Marco Podda, Davide Bacciu et Alessio Micheli. Une comparaison équitable des réseaux de neurones de graphes pour la classification des graphes. Dans *Conférence internationale sur les représentations de l'apprentissage* . 2019.
- [SMBGunnemann18] Oleksandr Shchur, Maximilian Mumme, Aleksandar Bojchevski et Stephan Günnemann. Pièges de l'évaluation des réseaux neuronaux de graphes. *arXiv preprint arXiv:1811.05868* , 2018.
- [KGrossGunnemann20] (1, 2) Johannes Klicpera, Janek Groß et Stephan Günnemann. Message directionnel passant pour les graphes moléculaires. Dans *Conférence internationale sur les représentations de l'apprentissage* . 2020.
- [JES+20] Bowen Jing, Stephan Eismann, Patricia Suriana, Raphael JL Townshend et Ron Dror. Apprentissage de la structure des protéines avec des perceptrons vectoriels géométriques. *préirage arXiv arXiv:2009.01411* , 2020.
- [Velivckovic22] Petar Velicković. Message passant tout en haut. *arXiv preprint arXiv:2202.11097* , 2022.
- [BFO+21] Cristian Bodnar, Fabrizio Frasca, Nina Otter, Yuguang Wang, Pietro Lio, Guido F Montufar et Michael Bronstein. Weisfeiler et Lehman passent au cellulaire : réseaux cw. *Advances in Neural Information Processing Systems* , 34:2625–2640, 2021.
- [TZK21] Erik Thiede, Wenda Zhou et Risi Kondor. Autobahn : réseaux neuronaux de graphes basés sur l'automorphisme. *Avancées dans les systèmes de traitement de l'information neuronale* , 2021.

