Machine Learning for Molecular Engineering
3/7/10/20.C01 (U) 3/7/10/20.C51 (G)
Spring 2024

Problem Set #3

Date: March 7, 2024
Due: March 21, 2024 @ 11:59 pm

Instructions

- **Important:** This problem set requires a GPU. Before you start in Google Colab, find `Notebook Settings` under the `Edit` menu. In the `Hardware accelerator` drop-down, select a GPU as your hardware accelerator.
- This problem set contains three questions, each containing sub-questions. Some questions have sub-questions marked as grad version. These questions are mandatory for those who have registered for the graduate version of this class (X.C51). They are optional for others. We've listed the points associated with each question in blue for the undergraduate and graduate version and orange for the graduate-only version. The undergraduate version is out of 75 and the graduate version is out of 100 points.
- Please submit your solution on Canvas as an IPython (Jupyter) Notebook file, `*.ipynb`. Your submission should contain *all* of the code that is needed to fully answer the questions in this problem set when executed from top to bottom and should run without errors (unless intentional). Please ensure that your plots are visible in the output file and that you are printing any additional comments or variables as needed. Commenting your code in-line and adding any additional comments in markdown (as "Text cells") will help the grader understand your approach and award partial credit.
- Note that collaboration between students is encouraged, however every student must be responsible for all the work in their individual submission. I.e., discussing solutions and debugging together should be considered fruitful teamwork, however physically taking the keyboard of another student and directly entering code for their submission would be intellectually dishonest and as such is prohibited. You should first read through and attempt every problem before discussing with others. Please list the names of all collaborators at the bottom of your .ipynb submission file.
- To get started, open your Google Colab or Jupyter notebook starting from the problem set template file pset3.ipynb (direct Colab link). You will need to use the data files here.

Background

In part 1, you will build a sequence-based deep learning model to classify DNA binding sites based on a DNA sequence. DNA binding sites are fragments in DNA sequence where other molecules/proteins may bind for biological functions. Binding activities in DNA are also associated with important biological process like transcription, a process to turn your DNA into mRNA which is later used to

synthesize proteins in your cells. The dataset you will work with is from ChIP sequencing which is an experimental method to probe protein interactions with DNA.

In part 2, you will develop a graph neural network to predict the aqueous solubility of molecules. Aqueous solubility, the ability of a substance to dissolve in a water, is a key property in drug development as it directly influences the bioavailability of a drug. Poor solubility can lead to formulation challenges, impacting manufacturing, stability, and ultimately requiring specialized approaches to enhance solubility and optimize drug delivery.

Part 1: Predicting DNA Binding Sites

In this part, you will write a pipeline that builds and trains a model to predict if a section of DNA sequence is a binding site for a protein. You will be provided a dataset of DNA sequences and a binary label 0/1 that indicates if the sequence binds to a protein or not. The sample data format is presented in Table 1.

DNA sequence	binder or not
'ATCGGGAA...'	1
'TGCAGTAT...'	0
...	...

Table 1: Dataframe snapshot from DNA binding data

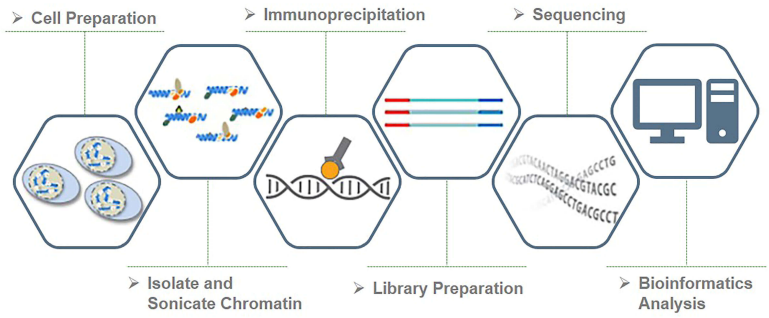

Figure 1: A typical ChIP workflow (source)

This part will require you to implement a deep neural network in PyTorch, just as you did in Problem Set 2. If you need to, please review Part 3 from Problem Set 2 before beginning this problem. If you did not do this previously, also go through the PyTorch tutorial here and the quickstart guide here.

As in Problem Set 2, you'll want a GPU for this part. Request one now.

Part 1.1 (10 points) Build Datasets and DataLoaders in PyTorch

Now you will construct your data processing pipeline. First load the DNA sequence data with the code provided. Because the data is loaded as strings like "ATGTCA...", you will need to one-hot encode the DNA sequences into bit vectors of size 4 (corresponding to the 4 possible DNA bases A, T, G, and C) and split the data set into 80% train and 20% test. Further retrieve 10% of your training data as the validation set which will be used to check for training convergence. Because training a deep neural network takes time, so we won't perform cross validation or hyperparameter

optimization as you did in previous psets. However, you might still want to do it in your final project if you decide to use a neural network based model.

Next, you need to construct your dataset and loader, with `torch.utils.data.Dataset` and `torch.utils.data.DataLoader`. Mimic our code from Problem Set 2 and the Quickstart guide to write a `dataset` class. Essentially, you need to implement the `__getitem__()` method that return a sequence `x` and its corresponding label `y` given an `index`. Construct train, validation and test data loader with `batch_size=256`, using your train, validation and test datasets.

The concept of batching data might still be new to you, so take a moment to think about what it's doing and why (and optionally, do some brief research on your own about batching in deep learning). What is the benefit of batching your data into mini-batches versus using the entire dataset to optimize the model all at once?

Part 1.2 (20 points) Write an LSTM-based classifier

In this section, you will build a DNA sequence classifier with recurrent neural networks. In contrast to a standard feed forward neural network (i.e. a multi-layer perceptron), an RNN is a specialized network architecture that operates on sequential data such as text, DNA sequences, and stock market prices. When RNNs operate on an input at a current time point, it considers not just the value of the current input, but also what it has learned from the inputs it received previously through a hidden context vector. It is through this persistent context vector that the model architectures captures the sequential structure of our data.

Now, let us try to understand the math behind a RNN. First, recall that a single transformation of a neural network requires the following computation:

$$\sigma(Wx + b) \tag{1}$$

where x is the data, W is a learned matrix, σ is a element-wise nonlinear transformation, and b is a learned bias. If we applied this calculation to our sequential data as one vector, we will have ignored the sequential (e.g., time-varying) nature of our data. RNNs are designed to operate on sequences one element at a time. Suppose our data has the form $\{x_0, x_1, x_2, ...x_t, ...x_T\}$; an RNN will sequentially construct a hidden embedding h_t for each x_t:

$$h_t = \sigma(W_{ih}x_t + b_{ih} + W_{hh}h_{t-1} + b_{hh}) \tag{2}$$

In this equation t is the index for the element in your sequence. x_t is your sequence data, and the embedding h_t is the representation learned up to time t. At step t, $W_{ih}x_t + b_{ih}$ transforms the data x_t to a hidden state and $W_{hh}h_{(t-1)} + b_{hh}$ linearly transforms the hidden embedding of the *previous* state h_{t-1} to produce the next hidden state h_t. The two contributions (from h_{-1} and x_t) are added together and passed through a nonlinear activation function, σ. This procedure is applied to the sequence from x_0 all the way to x_T with the same set of parameters $W_{ih}, W_{hh}, b_{ih}, b_{hh}$ for $T + 1$ times.

For this pset, we will use a long short-term memory (LSTM) model, which is a variant of the vanilla RNN described above. LSTMs have been very successful in deep learning applications for natural language processing. In 2017, Facebook switched to using LSTMs to perform billions of language translations on user comments. In contrast to regular RNNs described above, LSTMs contain a gated 'memory cell' to store long-range correlation along the sequence. You are not required to understand the inner-workings of an LSTM for this assignment (although you are encouraged to read about it on your own here).

Now, let us talk about how to construct a classifier using LSTMs in PyTorch. You will first apply an LSTM on the one-hot encoded sequential data you prepared in the previous part using the

`torch.nn.LSTM` class to construct a learned representation. Then, you will extract the parameterized hidden state after the final input, h_T, as an input to an MLP. The input dimension of the MLP should have the same dimension as the h_T and should output a scalar for each sample. Finally, a sigmoid activation function on the scalar output will provide a probability between 0 and 1 corresponding to the probability that this sequence is a DNA binding site. To summarize, there are three elements you need to construct:

- `lstm`: a LSTM module (`torch.nn.LSTM()`)
- `mlp`: a neural net (`torch.nn.Sequential()`, as in Problem Set 2). Use 2 hidden layers, each with tanh activation, and a final output layer with 1 output dimension.
- `sigmoid`: an element-wise sigmoid transformation (`torch.nn.Sigmoid()`)

In terms of mathematical equations:

$$\begin{aligned} \{h_t\}, h_T, c_T &= \texttt{lstm}(\{x_t\}) \\ \texttt{prob} &= \texttt{sigmoid}(\texttt{mlp}(h_T)) \end{aligned} \tag{3}$$

In this equation, x_t is the one-hot encoded DNA sequence, h_t are the hidden embeddings of the sequence, h_T is the embedding of the last element of the sequence. c_T is the cell state of the last element in the sequence. h_T is parameterized to encode the information of the entire sequence, and is the representation you need to use to feed into an MLP regressor.

For `torch.nn.LSTM`, please set `batch_first=True`, `num_layers=1`, `bidirectional=True`, and specify `hidden_size=hidden_dim` (the input to the `LSTMSeq` class). Since we're using a bidirectional LSTM, you'll have two hidden states produced (one for each direction); use the one corresponding to the backwards direction as input for the MLP, which will be $h_T[1]$ or $h_T[-1]$. For the `mlp`, use hidden dimensionality of `hidden_dim` in both layers. We have provided an example computation of LSTM-based classifier in the template file to help guide you, as well as some skeleton code.

Part 1.3 (20 points) Implement functions for training and testing

As in Problem Set 2, set up the Adam optimizer to train your model, with a learning rate `lr` of `1e-2`. In this problem, we will also be using a learning rate scheduler, specifically the scheduler `torch.optim.lr_scheduler.ReduceLROnPlateau`. This is a common strategy to prevent overfitting or overshooting a loss minimum by gradually reducing the learning rate during training. We have provided the scheduler for you in the skeleton code.

Next, write your training loop, following the Quickstart guide or the example we provided in Problem Set 2. Unlike before, this is a classification problem, so you will want to compute the binary cross entropy loss (`nn.functional.binary_cross_entropy()`) which requires you to input classification probabilities and the ground truth labels from your data. This should be implemented in the provided `train()` function, which already loops over minibatches for you. Record the loss of each batch and compute the average loss for all batches. Your `train()` function should return the average training loss.

Similarly, complete the `validate()` function to iterate over validation data and return the mean validation loss; remember to ensure that the validation function does not call the optimizer because the validation dataset is not used for training. You can make sure this is the case by calling `model.eval()` and turn on the `torch.no_grad()` context manager at the beginning of your validation function (we have implemented that for you). You will need to call `model.train()` to turn the gradients and optimizer updates back on in the `train()` function.

Finally, complete the `evaluate()` function. This function takes in data and produces the model's predictions, along with the true values for that dataset. You'll use it to evaluate your model's predictions on the test data and measure model performance.

Looping over the entire training data set once is called an *epoch*. The `train()` and `test()` function you will implement is only for one epoch, and should return a train loss and validation loss computed as averages over all the batches. Train and validate your model for 500 epochs; we have provided you with the code for this. This will take a few minutes.

Record the average train and validation loss for each epoch and plot these on a single graph, using our provided code. Finally report your test AUC score of your trained model on the test data. You would need to transform your prediction from `torch.Tensor` to `np.array` to compute ROC/AUC scores. As usual, you should use `sklearn.metrics.roc_auc_score`.

Part 2: Predicting molecular properties with Graph Neural Networks

In this part, you will write a pipeline that builds and trains a graph neural network (GNN) to predict the solubility of a molecule. You will work on a subset of the AqSolDB dataset [1] containing experimentally determined aqueous solubilities (unit: log mol/l). Solubility is crucial for drug development because it influences a drug's absorption and distribution within the body, affecting its bioavailability and therapeutic efficacy. Poorly soluble drugs may not reach therapeutic levels in the bloodstream, limiting their effectiveness.

Molecular Graphs

A graph $\mathcal{G}$ is a mathematical object that models connections (edges) between objects (nodes). Graph-structured data can be found in communication networks, ecological systems, and social networks. It is natural to consider using graphical representations for molecules because a molecule can be thought of as a set of atoms connected with chemical bonds.[1]

A graph $\mathcal{G}$ consists of a set of nodes $\mathcal{V}$ of size $n = |\mathcal{V}|$ and a set of edges $\mathcal{E}$ that represents the connections between pairs of nodes. The edges can be represented by a binary adjacency matrix $\mathbf{A} \in \{0,1\}^{n \times n}$ (Figure 2). For molecules, $\mathbf{A}$ is typically sparse since atoms are limited in terms of the number of connections they can make. Node $v_i \in \mathcal{V}$ and node $v_j \in \mathcal{V}$ are connected if $A_{ij} = 1$. $\mathbf{A}$ can be more compactly represented by an array of index pairs (i, j).

Nodes and edges of a graph $\mathcal{G}$ can be understood as data structure that stores chemical information for molecules just like how pixels in a image stores the information of colors. Based on the detailed chemistry of each molecule, the atoms (nodes) can have information about atomic numbers, formal charge, number of neighbors, etc., which can be stored in a vector of *node features*, $\mathbf{x}_i$. Edges can store information about bond orders, aromaticity, ring membership, etc., in vectors of *edge features*, $\mathbf{e}_{ij}$. *Edge features* and *node features* for a molecular graph can be constructed using various cheminformatics packages, and can capture structural information, properties computed through heuristics, or properties computed through quantum chemistry calculations.

SMILES

The simplified molecular-input line-entry system (SMILES) is a text based notation describing the structure of molecules using short ASCII strings. In terms of a graph-based computational

[1] This is a bit of a simplification, particularly for molecules that exhibit stereochemistry, but good enough for most applications.

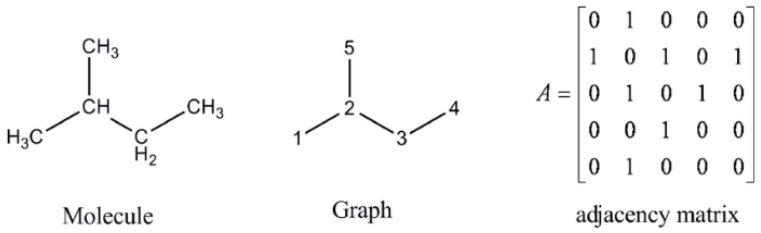

Figure 2: The chemical graph and adjacency matrix of isopentane [2]

procedure, SMILES strings are generated by printing the symbol nodes encountered in a breadth-first traversal of the graphs, typically excluding hydrogen atoms. Any cycles are broken so that the graph becomes an acyclic (tree-structured) graph. Numbers indicate connections between non-adjacent characters in the SMILES string; parentheses are used to indicate points of branching on the tree. Every molecule can be represented by multiple SMILES strings. For example, CCO, OCC, C(O)C, and [CH3][CH2][OH] all specify the structure of an ethanol molecule. There exist algorithms that can reproducibly generate a *canonical* SMILES string for a molecule. However, an arbitrary string of characters found in SMILES does not generate a SMILES string or a molecule; there is a fairly complex grammar that must be carefully respected.

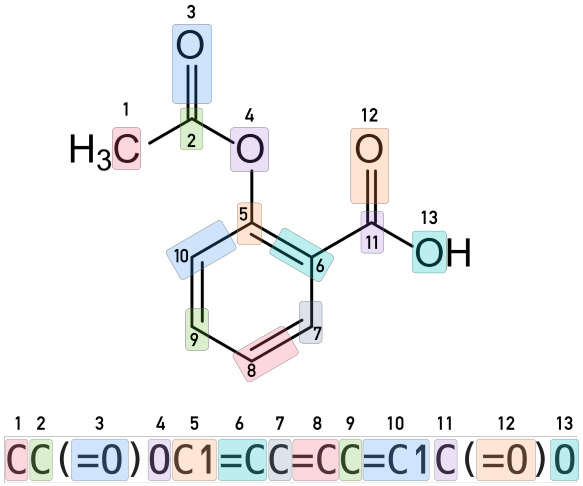

Figure 3: A molecular graph can be represented by SMILES string (image source)

Part 2.1 (5 points) Install and try out RDKit

RDKit is a open-source cheminformatics package that can process and manipulate molecular structures. Follow the example code to create a `rdkit.Chem.rdchem.Mol` object and visualize it as a 2D line drawing. Choose 4 of your favorite molecules (or any molecules) and visualize their molecular graphs arranged in an image grid with `rdkit.Chem.Draw.MolsToGridImage`.

Part 2.2 (10 points) Construct molecular graph Datasets and DataLoader

Figure 4 shows the molecular graph data structure for a GNN model. The GNN requires a feature set on the nodes and a list of edges as inputs. A common choice for the node feature is atomic

numbers which can be further encoded as bit vectors for each atom. A Graph Neural Network uses these node features and the edges to perform convolution operations and generate node-wise vector embeddings. A final readout layer takes the node-wise embedding to parameterize a pooled scalar (or vector) outputs. In many cases, an edge feature set is also included to encode chemical bond information like bond orders and aromaticity. For simplicity, the model you are going to work with only deals with node features.

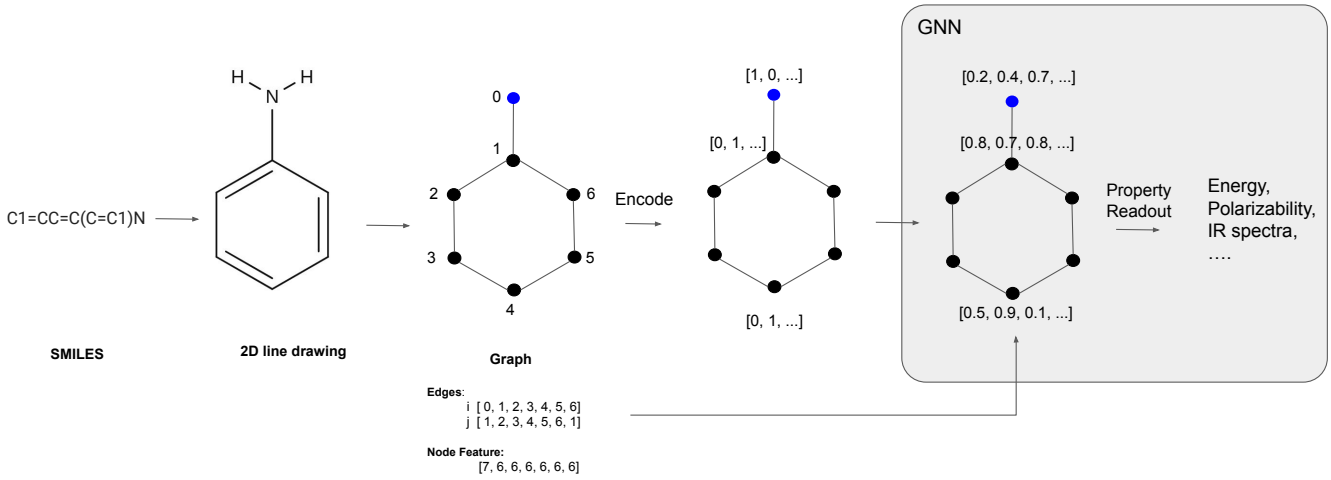

Figure 4: Data structure and model inputs for Graph Neural Networks

First, load the data as a `pandas.DataFrame` and shuffle its rows using `sklearn.utils.shuffle`. We have provided a `smiles2graph()` function that return the atomic numbers and edge array given a SMILES string. Loop over all the molecular SMILES and store the atomic numbers (`AtomicNum`), edge array (`Edge`), and the number of atoms ((`Natom`)) for each molecule and store them into three separate lists `AtomicNum_list`, `Edge_list`, and `Natom_list`. You also need to retrieve the polarizability values (as a `torch.FloatTensor`) which are under the column name 'α' in your dataframe; this is the target property to predict. Wrap the polarizability `y` values in 1-D array and store them all in a `y_list`. The data type required for `AtomicNum`, `Edge`, `Natom`, `y`, are `torch.LongTensor`, `torch.LongTensor` , `int`, and `torch.FloatTensor` (wrapped around a 1-D array) respectively.

Finally, as usual, split your data into 70% training, 10% validation, and 20% testing. You need to do this for `AtomicNum`, `Edge`, `Natom`, and `y`. Convert these into a `GraphDataset` object which we have implemented for you.

With your `Dataset` object defined, you can feed them into a `DataLoader`. This part is rather annoying, so we've done it for you. The challenge here is that batching a set of graphs is complicated because each graph has different number of node and edges: you cannot simply batch them by stacking arrays of uneven lengths together. There are additional book-keeping procedures you need to do: re-index node index in `Edge_batch` and record the sizes of graphs for the batch in `Natom_batch`. Figure 5 describes the batching operation.

In PyTorch, you can define your own customized collating function to make data batches with a user-defined function which takes a `list` of graph data tuple `(AtomicNum, Edge, Natom, y)` obtained from `GraphDataset.__getitem__()` method and combine them into one graph-structured data batch in a tuple `(AtomicNum_batch, Edge_batch, Natom_batch, y_batch)`. The function

takes a list of tuples that contain the data, and return the single joined graph with node order re-indexed. We have collated the graphs together in `collate_graphs()` and supplied that to the `DataLoader` to define your DataLoaders for you.

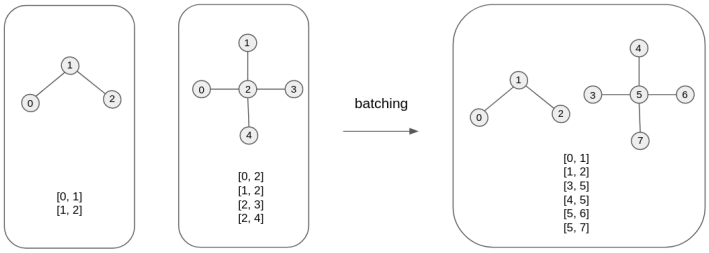

Figure 5: Batching operations for graphs

Part 2.3 (grad 20 points) Complete the definition of a GNN

There are many possible designs for a GNN which performs convolution operations with *node features* h_i and *edge features* e_{ij}. In this part, you will implement a simple GNN which only takes in atomic numbers z as initial node features h_i.

We start with an embedding layer:

$$h_i^{t=0} = \mathbf{W}\texttt{onehot}(z_i) \tag{4}$$

Let the size of the feature dimension be M and N_{types} be the number of atomic types. Equation 4 parameterizes the one-hot-encoded atomic numbers $\texttt{onehot}(z_i) \in \mathrm{R}^{\mathrm{N_{types}}}$ to an initial embedding $h_i^{t=0} \in \mathrm{R}^{\mathrm{M}}$. $\mathbf{W}$ is a $M \times N_{types}$ matrix. The embedding parametrization procedure can be done with the `torch.nn.Embeddings` module.

Each GNN layer operates on graph-structured data in two steps: a message step and an update step. The message step takes information from connected nodes, and the update function transforms these parameterized messages to update the features (embeddings) of each node:

$$\begin{aligned} m_i^t &= \sum_{j \in N(i)} m_{ij}^t = \sum_{j \in N(i)} \mathbf{MessageMLP^t}(h_i^{t-1} \cdot h_j^{t-1}) \\ h_i^t &= h_i^{t-1} + \mathbf{UpdateMLP^t}(m_i^t) \end{aligned} \tag{5}$$

Here, $\cdot$ is the element-wise multiplication and $j \in N(i)$ indicates the set of nodes that is connected to atom i. The superscript $t \in [0 \dots T]$ is the index of the layer; the subscript is index of the atom (node) in the graph. For each layer t, m_{ij}^t is the message from the edge set $(i, j) \in \mathcal{E}$ that affects the node embeddings in the next layer and h_i^t is the parametrized node embedding used to determine the message in the next layer. $\mathbf{MessageMLP^t} : \mathbb{R}^M \to \mathbb{R}^M$ and $\mathbf{UpdateMLP^t} : \mathbb{R}^M \to \mathbb{R}^M$ are MLPs that parameterize the messages and update atom (node) embeddings.

After T convolutional layers, each node receives a parameterized embedding $h_i^T \in \mathbb{R}^M$. To map all the node embeddings h_i^T onto a predicted scalar value, you need to construct a $\mathbf{ReadoutMLP} : \mathbb{R}^M \to \mathbb{R}$. The final property prediction layer has the form:

$$y = \sum_{i \in \{1,2,\dots|\mathcal{V}|\}} \mathbf{ReadoutMLP}(h_i^T) \tag{6}$$

In the `GNN()` class, we have already implemented all the MLPs you need for each convolution stored in a `nn.ModuleList` which you can loop over for each convolution step to retrieve each model, as well as the initial embedding and readout MLPs. You will need to implement the `forward()` function, which does the actual computation given all these MLPs. For this, you'll need a few additional tools that we've provided. In order to sum all the messages from $j \in N(i)$ onto each node i in Equation 5, use the `scatter_add()` function we've provided alongside clever PyTorch indexing into `h`. Do not use a for loop over either the atoms or the number of edges; that will be too slow for your GNN to run.

Once you finish running the convolutions, you'll want to implement Equation 6. The challenge here is that the graph convolutional procedure operates on batched graph data, so you will need to split the final property prediction output $\textbf{ReadoutMLP}(h_i^T)$ back into the original graphs to perform the summation; use `Natom` to determine the original size of each graph. To do this, use the `torch.split()` function to split the tensor based on the list of the number of atoms (indices) in each and then sum the embeddings of nodes within the same graph.

To aid you with these steps, we've provided examples of the use of both `scatter_add()` and `torch.split()` for you. You should not need any fancy PyTorch functions beyond the two that we've provided you. Remember to double-check the shape of your output from the GNN and make sure it's what you expect; you've already seen from previous problem sets that this can cause issues.

Implement the graph convolution model described above. Choose a hidden dimension of 64 ($M = 64$) and a depth of 3 ($T = 3$). You can specify M and T in the model as `n_embed` and `nconvs` in the `GNN()` class we defined for you.

Part 2.4 (grad 5 points) Verify that your GNN preserves permutational invariance

When a GNN operates on a set of graphs, the nodes of a graph are arranged in a particular order. An important property that GNNs need to ensure is that the output should be the same given arbitrary node ordering. In other words, the final molecule-level output of GNNs should preserve *permutational invariance*.

Verify that the GNN outputs respect permutation invariance by running the cell we have provided you. It operates on a small 4-node graph, generates a list of all possible permutation of node orders using `itertools.permutations`, and permutes the graph with the function `permute_graphs()` to get a re-ordered set of node and edge inputs. If you have implemented your GNN correctly, it should produce the same output for all the permuted graphs.

Part 2.5 (10 points) Train and test your GNN

Train your GNN for 500 epochs with the train and validation loop we provided to you. Calculate the R^2 and render a scatter plot of predicted vs. true solubilities for both the training and test set. Undergrads can use the provided GNN implementation.

References

[1] Sorkun, M. C., Khetan, A. & Er, S. AqSolDB, a curated reference set of aqueous solubility and 2D descriptors for a diverse set of compounds. *Scientific Data* (2019).

[2] Galvez, J., Garcia-Domenech, R. & Castro, E. Molecular topology in QSAR and drug design studies. QSPR-QSAR Studies on Desired Properties for Drug Design. *Research Signpost* 63–94 (2010).