Machine Learning for Molecular Engineering

Problem Set 5

Date: April 14, 2023
Due: 11:59 pm ET on Thursday, April 27, 2023

Instructions Please submit your solution on Canvas as an IPython (Jupyter) Notebook file, `*.ipynb`. We highly suggest using Google Colab for this course and using this template for this problem set. 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.

To get started, open your Google Colab or Jupyter notebook starting from the problem set template file pset4.ipynb (direct Colab link). You will need to use the data files here.

Save Trained Models In this problem set, we recommend that you save your models periodically during training because it can be time-consuming to retrain your models when you start a new Colab session. You can save your model to your Google Drive following the example here. The template file also has an example of how you can mount your Google Drive and save a PyTorch model.

Background: Representations of molecules

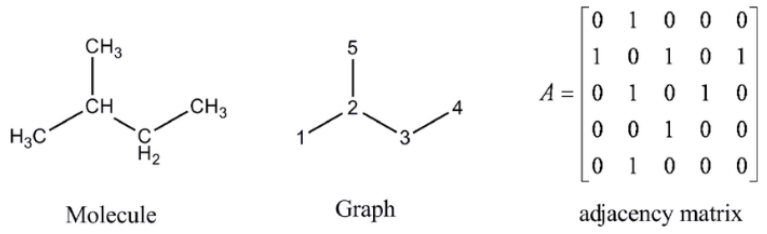
Molecular Graphs

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 1). 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.

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



$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Figure 1: The chemical graph and adjacency matrix of isopentane [1]

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 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.

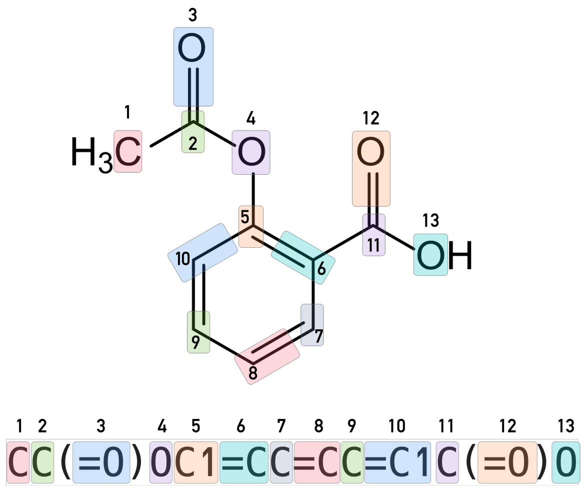


Figure 2: A molecular graph can be represented by SMILES string (image source)

Part 1: Predicting molecular properties with Graph Neural Networks

Part 1.1 (5 points) Install and try out RDKit

RDKit is a open-source cheminformatics package that can process and manipulate molecular structures. First, request a GPU as you've done in previous problem sets. Then, download and install RDKit with the code provided in the template file. 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 1.2 (10 points) Construct molecular graph Datasets and DataLoader

Figure 3 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.

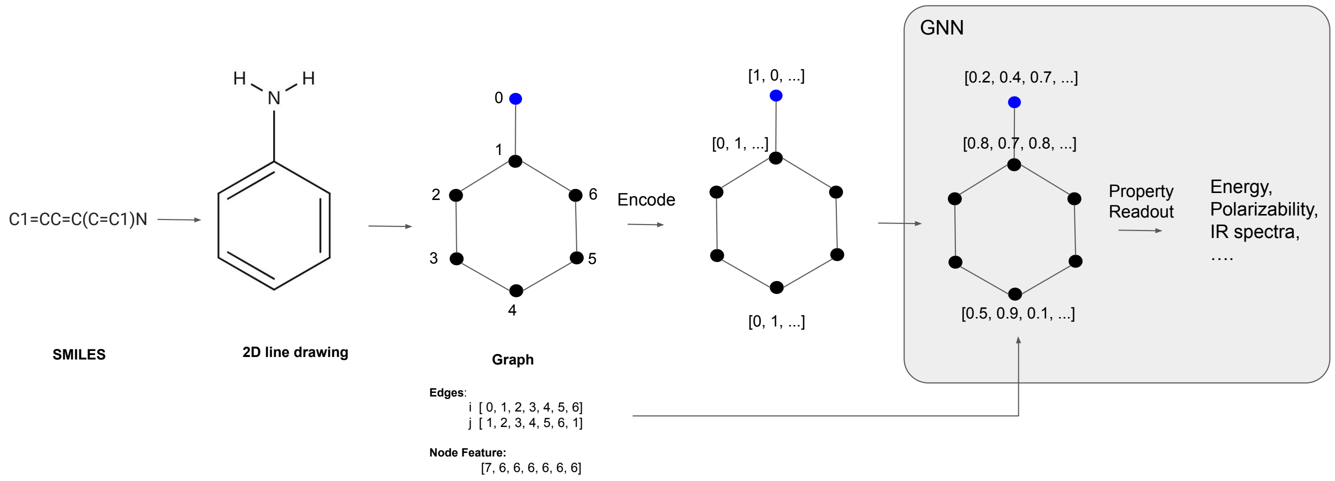


Figure 3: Data structure and model inputs for Graph Neural Networks

You will work on a subset of the QM9 dataset [2] to predict the isotropic polarizabilities (unit: Bohr^3) of molecules calculated with density functional theory (quantum chemistry).

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 4 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.

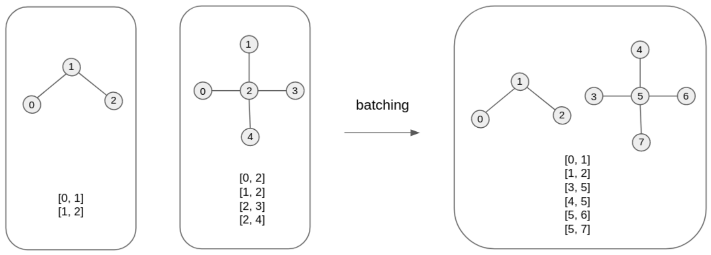


Figure 4: Batching operations for graphs

Part 1.3 (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:

$$h_i^{t=0} = \mathbf{W}\texttt{onehot}(z_i) \tag{1}$$

Let the size of the feature dimension be M and N_{types} be the number of atomic types. Equation 1 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:

$$\begin{aligned} m_i^t &= \sum_{j \in N(i)} m_{ij}^t = \sum_{j \in N(i)} \mathbf{MessageMLP^t}(h_i^{t-1} \cdot h_j^{t-1}) \\ h_i^t &= h_i^{t-1} + \mathbf{UpdateMLP^t}(m_i^t) \end{aligned} \tag{2}$$

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:

$$y = \sum_{i \in \{1,2,\dots|\mathcal{V}|\}} \mathbf{ReadoutMLP}(h_i^T) \tag{3}$$

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 2, 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 3. 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 $\mathbf{ReadoutMLP}(h_i^T)$ 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 1.4 (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 1.5 (10 points) Train and test your GNN

Train your GNN for 500 epochs with the train and validation loop we provided to you. Use the hyperparameters and model defined in Part 1.4. This will probably take at least a couple of hours; if your implementation is inefficient, it may take up to 6 hours. If your first epoch takes longer than a minute to run, consult with your classmates or the TAs. Render a scatter plot of predicted vs. true polarizabilities for both the training and test set. Include the MSE loss in the legend. If you need to restart training, use the `torch.load` function to retrieve your saved model, instead of retraining from scratch. Leave a note in your submission explaining that you restarted training.

Part 2: Variational auto-encoders (VAEs) for SMILES strings

This part is completely independent of Part 1 and can be done in a separate Colab session.

Variational Auto-Encoders (VAEs) are a class of generative models. The encoder consists of Gated Recurrent Units to encode a SMILES string into a latent representation. The decoder is also a stacked GRU that takes the latent representation to reconstruct the input SMILES. The autoencoder is trained to use the encoder and decoder to reconstructs your input as closely as possible. In this exercise, you will be asked to implement the sampling and loss function for a SMILES VAE. Because training a VAE on a large dataset can be computationally demanding, we have provided a SMILES-VAE model that is pre-trained on 1 million molecules. You can load the model with the code we provide. You will train on a smaller dataset of of 50,000 molecules to fine-tune the model. It is still a lot of data, so please train your model on a GPU.

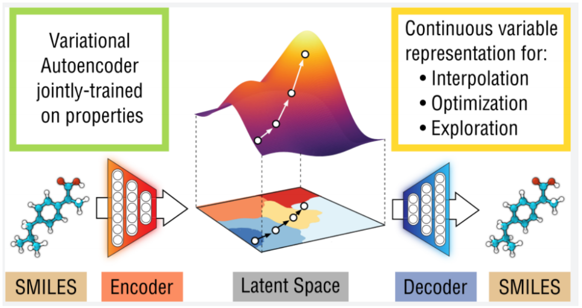

Variational Autoencoder jointly-trained on properties
Continuous variable representation for:
• Interpolation
• Optimization
• Exploration
SMILES
Encoder
Latent Space
Decoder
SMILES

Figure 5: Applying a VAE on SMILES strings for molecular design [3]

Part 2.1 (5 points) One-hot encode SMILES strings into padded numerical vectors

As introduced in the background section, molecules can be represented as 1D strings with SMILES representation. Encode each character in a SMILES string into categorical numbers (note that this is *not* the same as one-hot encoding) using `LabelEncoder()` from `sklearn`. We have provided a list of string characters in `moses_charset` which you can use as a dictionary to write a SMILES. This dataset has SMILES strings with different character length, this requires an additional preprocessing procedures called padding, *i.e.*, adding empty characters (`' '`) to the end of all strings to make sure all the strings has the same length. The length of strings is determined by the longest SMILES string which you need to find out.

Next, transform your encoded SMILES data into a `torch.LongTensor`. This is an unsupervised learning task, so there is no `y`. Split your data into a 60:20:20 train:validation:test split as before, using `torch.utils.data.TensorDataset` as your `Dataset` object. Like what you did in Problem Set 3, use a `DataLoader` with `batch=512` and `shuffle=True`.

Part 2.2 (14 points) Implement the reparametrization trick for VAE

We have provided the implementation for a SMILES-VAE in the `MoleVAE()` class. In VAEs, the encoding of an input x into an embedding z is not deterministic. The model is trying to compress the data into a latent distribution via a conditional distribution $Q_\phi(z|x) = \mathcal{N}(\mu_\phi, \sigma_\phi^2)$ which is parametrized by an encoder function, ϕ. The model needs to sample from the distribution: $z \sim Q_\phi(z|x)$. However, this sampling process requires some extra attention because we want to ensure the sampling procedure is differentiable for gradient optimization. Generally, we cannot do this for a sampling process:

$$z \sim \mathcal{N}(\mu_\phi, \sigma_\phi^2) \tag{4}$$

since the derivatives $\frac{dz}{d\mu_\phi}$ and $\frac{dz}{d\sigma_\phi}$ are not clearly defined for this sampling procedure. However, we can use the reparameterization trick which suggests that we randomly sample ϵ from a unit multivariate Gaussian, and then shift the randomly sampled ϵ by the latent distribution's mean μ and scale it by σ.

$$\begin{aligned} \epsilon &\sim \mathcal{N}(0, \mathbf{1}) \\ z &= \mu_\phi + \epsilon \cdot \sigma_\phi \end{aligned} \tag{5}$$

The reparametrization trick makes the sampling process from a multivariate Gaussian *differentiable*! The reason why it works is because the randomness in the sampling is 'reparameterized' into a leaf node which does not require gradient calculation in the backward computation.[2] To put it more concretely, during the backpropagation process when the gradient $\frac{dL}{dz}$ is computed (L is the scalar loss function), the gradients on μ_ϕ and σ_ϕ can be further computed with:

$$\begin{aligned} \frac{dL}{d\mu_\phi} &= \frac{dL}{dz}\frac{dz}{d\mu_\phi} = \frac{dL}{dz} \\ \frac{dL}{d\sigma_\phi} &= \frac{dL}{dz}\frac{dz}{d\sigma_\phi} = \frac{dL}{dz}\epsilon \end{aligned} \tag{6}$$

[2]The reparameterization trick is a useful technique for variational inference trained with a gradient-based method. Similar reparameterization tricks can be derived for other types of distributions like the Beta, Dirichlet and Von-Mises distributions [4].

The distributional output of the encoder input requires additional attention. Note that the encoder module parametrized by an MLP might output a negative σ_ϕ^2 which might annoy a statistician. The numerical trick to ensure that σ_ϕ has only positive values is to output the log of σ_ϕ^2 instead and then exponentiate:

$$\begin{aligned} \mu_\phi, \log \sigma_\phi^2 &= \texttt{encoder(SMILES)} \\ \sigma_\phi &= \exp\left(\frac{1}{2}\log \sigma_\phi^2\right) \end{aligned} \tag{7}$$

First implement the function to transform $\log \sigma_\phi^2$ to σ in `MolVAE().get_std()` (as in equation 7)). Then, implement the reparametrization trick in the `MolVAE().reparametrize()` function which takes two inputs: μ_ϕ and σ_ϕ and outputs a latent vector z as in equation 5 (you will need the `torch.randn` method to sample ε). Test your program `reparametrize()` by using our provided code to sample 1000 samples from a 1D distribution with $\mu = 0$ and $\sigma^2 = \mathbf{1}$ and compare the sampled distribution with $\mathcal{N}(0, \mathbf{1})$.

Part 2.3 (14 points) Implement the SMILES VAE loss function

The decoder model $P_\theta(x|z)$, parameterized by a set of parameters θ, takes the sampled latent code $z \sim Q_\phi(z|x)$ to reconstruct x which is your input SMILES string. The training objective of a VAE is to minimize the negative evidence lower bound, or ELBO, which can be understood as optimizing a reconstruction loss and regularization term. The regularization term is the KL divergence between the parameterized distribution and the prior distribution.

$$\begin{aligned} L &= L_{recon} + \beta L_{regularization} \\ &= \int dz\, Q_\phi(z|x) \log P_\theta(x|z) + \beta \int dz\, p(z) \log \frac{p(z)}{Q_\phi(z|x)} \end{aligned} \tag{8}$$

β is a hyperparameter that balances the two loss terms (see Ref.[5] for more information about the effect of β). The reconstruction term compares the original input and the decoded inputs. The input data here is a sequence of vectors with encoded categorical values and the output is a sequence with probabilities for each character categories, so we can use `nn.Functional.cross_entropy` as the training objective to minimize.

$$L_{recon} = -\frac{1}{N_{seq} N_{char}} \sum_i^{N_{seq}} \sum_k^{N_{char}} p_{data}(\hat{x}_{i,k}) \log(p(x_{i,k})) \tag{9}$$

$\log(p(x_{i,k}))$ is the logit for each possible character category reconstructed by the decoder, $\hat{x}_{i,k}$ is the original molecular SMILES; N_{seq} is the length of the sequence and N_{char} is the total number of possible characters in the sequence. `nn.Functional.cross_entropy` takes two inputs, the predicted logits for each character at each position in the SMILES string (dimension $= N_{batch} \times N_{seq} \times N_{char}$), and the original data as character category represented by integers at each position in the padded SMILES string (dimension $= N_{batch} \times N_{seq}$)). Please check the documentation for `nn.Functional.cross_entropy` for example usage.

Warning: Check your dimensions carefully in this step. In particular, the decoder output will need the batch size to be dimension 0, the number of characters to be dimension 1, and the sequence length to be dimension 2, and the original SMILES input will need the batch size to be dimension 0 and the sequence length to be dimension 1 for `nn.Functional.cross_entropy` to work as intended. Transpose your tensor with `transpose` if necessary to get this to line up.

A simple prior distribution one can choose is a multivariate Gaussian distributions with all the means as zeros, and all the standard deviations as ones. The parameterized distribution from the encoder is a distribution of the same dimension with parametrized means/standard deviation. Minimizing the Kullback-Leibler (KL) divergence between the parametrized distribution $Q_\phi(z|x)$ and $\mathcal{N}(0, \mathbf{1})$ encourages the $Q_\phi(z|x)$ to be statistically closer to the Gaussian distribution prior $p(z) = \mathcal{N}(0, \mathbf{1})$. The KL divergence between the encoded latent distribution (approximated posterior) and the prior has a nice analytical form for Gaussian distributions with a diagonal covariance matrix (you can find the derivation here):

$$\begin{aligned} L_{regularization} &= KL(Q_\phi(z|x)|p(z)) \\ &= KL\left(\mathcal{N}(\mu_\phi(x_i), \sigma^2_\phi(x_i))|\mathcal{N}(0,1)\right) \\ &= \frac{1}{N_{batch}} \sum_i^{N_{batch}} \frac{1}{2} \left(\sum_d^{N_z} \sigma_{d,\phi}(x_i)^2 + \mu_{d,\phi}(x_i)^2 - \log \sigma_{d,\phi}(x_i)^2 - 1 \right) \end{aligned} \tag{10}$$

$d \in \{1, ..., N_z\}$ is the index for the latent dimension.

For this problem, you need to implement the reconstruction (equation 9) loss and the KL divergence (equation 10) in the `loss_function()` function.

Part 2.4 (2 points) Train your model

After implementing the reparameterization trick and loss function, you should be able to train a model with the train and test loop we provided to you. Load the pre-trained model with the code we provided to you. Train the VAE for 50 epochs. Make sure you obtain a training and test loss below 0.15 before proceeding to the next part. We recommend you save the trained model periodically in your Google Drive. Choose $\beta = 0.001$. This will take around 15 minutes, but shouldn't take too long.

Part 2.5 (10 points) Sample new molecules

The latent space learned by the model is an learned continuous space which you can navigate. The space encodes the complicated molecular 'grammar' of the data it trained on. By sampling vectors z, you can then use a decoder to reconstruct the continuous representation back to a SMILES string. Now use your trained model to sample novel molecules from the learned distribution. Randomly select two SMILES in your test data, and encode into latent vectors with the encoder and then linearly interpolate between the two molecules in the latent space to obtain 10 points in the z space. For each sampled latent code, decode z back to SMILES strings. You can use the `index2SMILES()` function we provided to convert categorical values to SMILES strings. We provide a function `check_SMILES()` to check the validity of generated SMILES strings.

Finally, produce a scatter plot with the first two dimensions of z of your test molecules and newly sampled molecules in the same figure, label your generated SMILES with the decoded SMILES in the legend. Among the 10 molecules you sampled, how many were decoded into a valid SMILES string? (Don't worry if most of the molecules don't decode into valid SMILES strings.) Use RDKit to show 2D line drawings of valid SMILES you generated. Can you propose a reason for why your VAE sometimes fails to generate valid SMILES strings?

Part 3: Feedback Survey

Part 3.1 (5 points) Please finish the feedback survey on our Canvas site

References

[1] 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).

[2] Ramakrishnan, R., Dral, P. O., Rupp, M. & Von Lilienfeld, O. A. Quantum chemistry structures and properties of 134 kilo molecules. *Scientific data* **1**, 1–7 (2014).

[3] Gómez-Bombarelli, R. *et al.* Automatic chemical design using a data-driven continuous representation of molecules. *ACS central science* **4**, 268–276 (2018).

[4] Figurnov, M., Mohamed, S. & Mnih, A. Implicit reparameterization gradients. *arXiv preprint arXiv:1805.08498* (2018).

[5] Higgins, I. *et al.* β-VAE: Learning basic visual concepts with a constrained variational framework (2016).