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

Problem Set #4 - FASTA

Date: March 21, 2024
Due: April 11, 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 two 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 "pset4_bio_template_2024.ipynb" (direct Colab link). You will need to use the data files here.

Background

SMILES for representing molecules

Whether it's designing specific ligands for enzyme inhibition, predicting the biological activity of small molecules, or exploring chemical space for drug discovery, understanding how to encode chem-

ical structures as strings, called SMILES, is essential for computational biology (as well as computational chemistry, of course). 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.

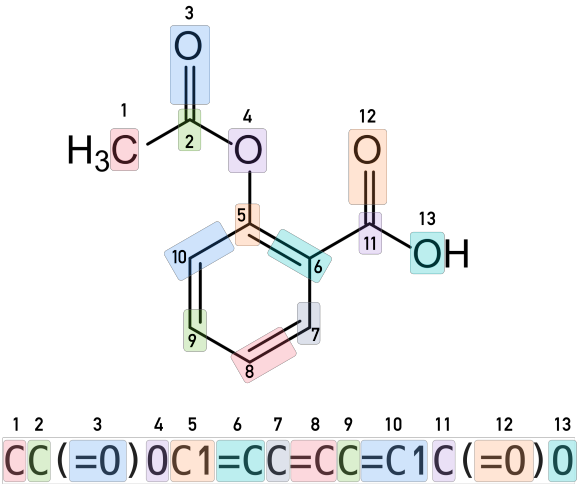

Figure 1: A molecular structure can be represented by SMILES string (image source)

FASTA for representing proteins

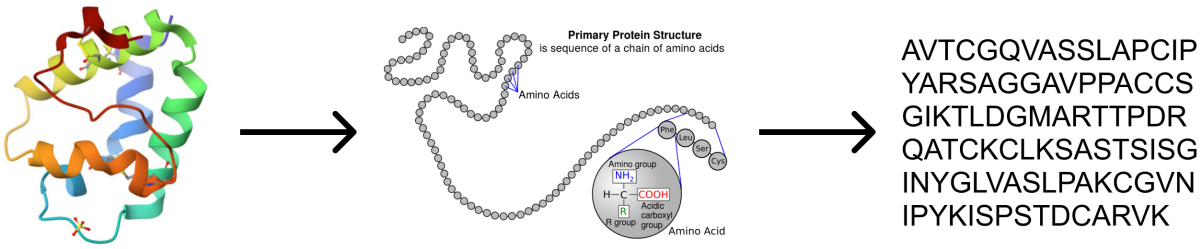

Figure 2: A protein (left; image source) is a sequence of amino acids (center; image source) that can be represented as a sequence of letters in the FASTA format (right).

We just discussed how the structures of molecules are commonly represented with SMILES strings. What about nucleotide (DNA, RNA) sequences or amino acid (protein) sequences? We, of course, could also represent them with SMILES strings but these would get very long! Fortunately, biology has given us a neat higher-level organization for nucleic acids and proteins. Namely, diversity

in nucleotides can largely be captured by sequences of adenosine (A), cytosine (C), guanine (G), thymine (T), or uracil (U). Similarly, diversity in most proteins can be captured by sequences of 20 amino acids. Thus, as you likely can imagine after considering SMILES, the bioinformatics community has developed the FASTA format for the representation of nucleic acids and proteins. Like SMILES, the FASTA format is a text-based notation describing structure using short ASCII strings. In the FASTA format, one letter represents one amino acid or one nucleotide. In this PSet, we will use the FASTA format for proteins. A table to translate each one-letter abbreviation to the full amino acid name can be viewed here.

Part 1: Dimensionality Reduction for Molecular Representations

In this part, you will be performing dimensionality reduction analysis on a set of candidate molecules assessed for inhibition of the protein Janus kinase 2, also called JAK2. The data is from Popova et al. (2018) [1], which used deep reinforcement learning to generate a dataset of 1,911 small molecules with pIC50 values for JAK2.

Especially relevant in pharmacology and biochemistry, pIC50 refers to the negative logarithm of the IC50 value, where IC50 stands for the half-maximal inhibitory concentration and measures the potency of a substance in inhibiting a specific biochemical function, often the activity of a protein. A higher pIC50 value indicates a higher potency of inhibition. Inhibitors of JAK2 have applications for the treatment of cancers and inflammatory diseases; thus, there is interest in identifying molecules with high pIC50 for JAK2. The goal of this part is to use dimensionalty reduction to assess if the representative dimensions of candidate molecules' chemical structures fundamentally organize them by their predicted pIC50 for JAK2.

As discussed in Background, the chemical structure of molecules like our potential inhibitors of JAK2 can be represented by SMILES strings. Howeover, in order to convert these SMILES strings into a vector representation, the cheminformatics community has developed a variety of algorithms to numerically encode the structural information of molecules. Morgan fingerprints, also known as circular fingerprints, represent the molecular graph as a unique bit string that encodes the molecular topology. You can understand it as a bit-vector categorical encoding with 1 for substructures present in the molecule and 0 for those not present. Some implementations of fingerprints use counts of how many times a substructure is present, instead of just 0s and 1s for absence/presence. Morgan fingerprints are typically high dimensional to maximize their descriptive power and sizes of 512, 1024 or 2048 are common (you can choose the number of bits when generating these fingerprints with the Morgan Algorithm). Morgan fingerprints provide a vector representation of molecules that is nearly unique or each molecule (although fingerprint collisions for non-identical molecules are possible), on which we can apply machine learning algorithms. Note that we provide the code to convert the SMILES strings to 2048-dimensional molecular fingerprints. The fingerprints are computed with the Morgan algorithm implemented in the RDkit package.

In this question, you will apply dimensionality reduction algorithms on the high-dimensional molecular fingerprint data.

First, run the code we provide to load in `jak2.csv`, a dataframe with two columns for SMILES strings and pIC50 values for JAK2 for 1,911 molecules. Next, run the code we provide to convert these SMILES strings into 2048-bit Morgan fingerprints. The row in the fingerprint dataframe (i.e., `jak2_fp`) is mapped one-to-one to the row of the molecule dataframe.

Part 1.1 (5 points, Grad only) Choosing radius and number of bits for Morgan fingerprints

You will notice that we set the radius and number of bits to 3 and 2048, respectively, in the code we provided to convert SMILES to Morgan fingerprints. Do some quick researching on the radius and number of bits parameters in in RDKit's Morgan fingerprinting. Provide a one-sentence description of what the radius represents and another of what the number of bits represents. How does adjusting the radius parameter affect the granularity of the motifs captured by the fingerprints, and how does this relate to the choice of the number of bits? (No code, just your thoughts for this.)

Part 1.2 (10 points) Principal Component Analysis on Molecular Fingerprints

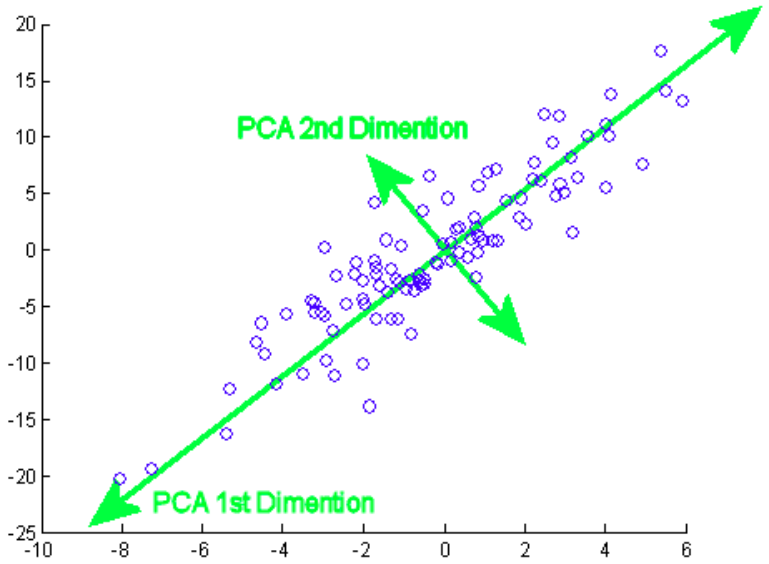

Figure 3: An example PCA analysis on a 2D dataset. The principal components are marked in green arrows. (source)

Principal Component Analysis (PCA) is a dimensionality-reduction method that is often used to tackle high-dimensional, large data sets. In PCA, a large set of variables is transformed into a smaller one that contains as much of the information in the larger set as possible. PCA performs a linear transformation on your data by re-projecting in onto a new orthogonal basis. These new basis is chosen such that the fewest basis vectors best explain the variance of your data. Because of ease of visualization, it is common to just interrogate the first two dimensions (also called eigenvectors or components) of the PCA, but its possible to select additional dimensions.

You should perform PCA on the fingerprint data with `sklearn.decomposition.PCA()`. Use 100 components by setting `n_components=100`. This will reduce your original 2048-dimensional space onto a space of just 100 dimensions. Visualize the first two components of your data in a 2D scatter plot and color each molecule by its pIC50. What is the total explained variance ratio of the first 100 principal components? (This is an estimate of how much information is preserved after this dimensionality reduction procedure). You can retrieve the ratios of explained variances by checking the `.explained_variance_ratio_` attribute of a `PCA()` object.

Inspect the scatter plot. Do you observe any patterns in the distribution of active drugs in this 2D plot?

Part 1.3 (10 points) t-SNE analysis on Molecular Fingerprints

PCA applies linear transformations on your data, but may struggle to tease out subtle underlying patterns in your data. Non-linear transformations can distort or compress the dimensions much more effectively and can uncover much more complex structure in the dataset. There are two popular choices for non-linear dimension reduction: t-SNE and UMAP. t-Distributed Stochastic Neighbor Embedding (t-SNE) performs a stochastic non-linear transformation on your data and it is implemented in sklearn ((`sklearn.manifold.TSNE()`)). In general, t-SNE is a random algorithm so you will have different looking embeddings every time; it is also sensitive to *perplexities* which balance attention between local and global aspects of your data during the optimization. Read the documentation for `TSNE()` to understand these options. Moreover, if you're curious, check out this great interactive website for comparing different dimensionality reduction techniques and their respective parameters.

Apply t-SNE on your the 100-dimensional embedding you obtained from previous part with `n_components=2` with a perplexity value of 2, 30 and 500 separately and produce three labeled t-SNE plots with molecules colored by pIC50. The t-SNE computation might take a couple minutes. Do you notice any differences in the clustering? Focus on the plots obtained with `perplexity=30`. In your 2D scatter plot, do you observe any patterns among the molecules with respect to pIC50?

Part 1.4 (20 points, Grad only) Are the low dimensional embeddings meaningful?

In this question, you will be asked to make sense of the extracted low dimensional embeddings and how the may relate to prediction performance in a supervised learning task. You will need to train a Random Forest classifier and then relate the performance of this supervised to the embeddings you learned in the previous part.

Let's imagine a biopharmaceutical company is only willing to invest resources into making molecules that have $\geq$ 9.5 pIC50, as they consider this threshold important in discerning potent, effective inhibitors of JAK2. Therefore, you should discretize pIC50 data by classifying any molecule with a pIC50 value $\geq$ 9.5 as effective (i.e., 1) and any with $<$ 9.5 as ineffective (i.e., 0). Capture this information as a new column called `is_effective` in the `jak2` dataframe.

Next, split your data into 10 folds using `sklearn.model_selection.KFold()` and train a random forest classifier with 100 trees (`n_estimators=100`). Use the full fingerprint (not the low-dimensional embedding) as your featurization for the random forest. Just as you did in PSet 1, use `sklearn.ensemble.RandomForestClassifier`. Test each fold with models trained on the other 9 folds, and record your prediction in a dataframe or an array. Classify your predictions in 4 categories: True Positives (TP), True Negatives (TN), False Positives (FP) and False Negatives (FN). Make a scatter plot with the t-SNE embeddings, and color each of the four categories above with different colors. Inspect your plot; do you observe any patterns in the prediction categories (TP, TN, FP, FN) on your t-SNE plots?

Part 2: Variational auto-encoders (VAEs) for proteins

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

Save Trained Models In this part, 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.

Variational Auto-Encoders (VAEs) are a class of generative models. Adapted from the VAE architecture for SMILES detailed in Gómez-Bombarelli et al. (2018) [2], this part of the PSet implements a VAE for protein sequences represented in FASTA format. The encoder consists of Gated Recurrent Units to encode a FASTA sequence into a latent representation. The decoder is also a stacked GRU that takes the latent representation to reconstruct the input FASTA. The autoencoder is trained to use the encoder and decoder to reconstruct your input as closely as possible. In this exercise, you will be asked to implement the sampling and loss function for a FASTA VAE.

The dataset used for this exercise is subsetted from Hawkins-Hooker et al. (2018) [3]. We provide a dataset of 17,403 protein sequence homologues of the *luxA* bacterial luciferase protein. This protein makes bacteria "light up" (i.e., bioluminesce); thus, the application of generative ML here could enable us to discover and formulate novel luminescence proteins.

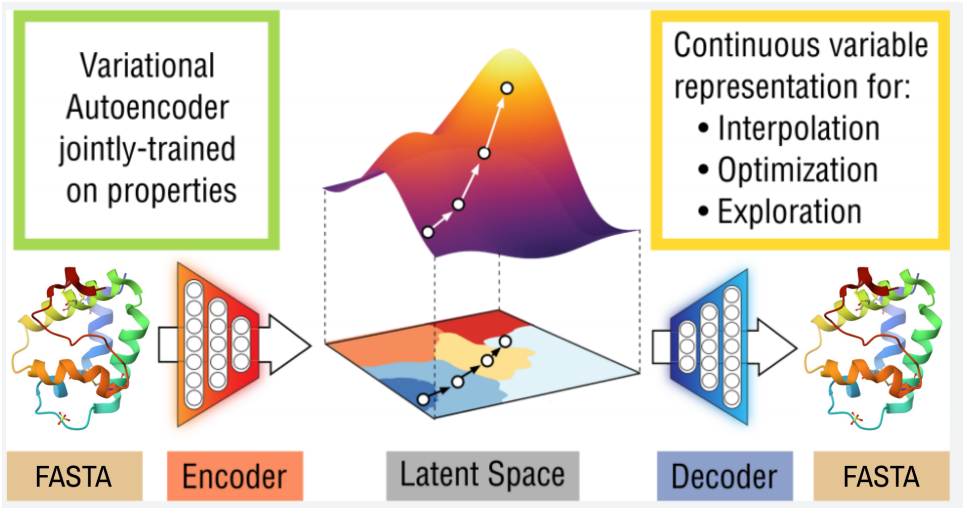

Figure 4: Applying a VAE on FASTA sequences for protein design, adapted from Ref.[2] which developed a SMILES VAE for molecular design.

The dataset we have provided consists of a total of 17,403 protein sequences. The sequences were aligned via multiple sequence alignment (MSA). All columns of the MSA that were unoccupied in the target *luxA* protein were not considered (i.e., marked with "-" in the FASTA representation). All sequences were padded to have equivalent length (i.e,. 360 amino acid positions).

Part 2.1 (5 points) Encode protein sequences into numerical vectors

You'll notice that we provided the code chunk to load in the train (70%), validation (10%), and test (20%) data. Rather than implementing random splitting, to avoid highly similar sequences appearing in the train and validation sets (and hence avoid overfitting), the data was split by first clustering all sequences using MMseq2 and adding discrete clusters to each split [3]. We have provided the split train/validation/test data for you (`luxA_train`, `luxA_val`, `luxA_test`), so you should simply run the code chunks to load the data. Each of these `pandas` dataframes has one column containing the amino acid sequences (`aa_sequence`).

As introduced in the background section, proteins can be represented as 1D strings with FASTA representation. Encode each character in a FASTA sequence 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 `aa_charset` which you can use as a dictionary for FASTA.

For this subquestion, you first need to transform your *encoded* train, validation, and test FASTA data into `torch.LongTensor` objects. This is an unsupervised learning task, so there is no `y`. Next, like what you did in prior PSets, use a `DataLoader` with `batch=512` and `shuffle=True`.

Part 2.2 (15 points) Implement the reparametrization trick for VAE

We have provided the implementation for a Protein VAE in the `ProteinVAE()` 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 $Q_\phi(z|x) = \mathcal{N}(\mu_\phi, \sigma^2_\phi)$ 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:

$$z \sim \mathcal{N}(\mu_\phi, \sigma^2_\phi) \tag{1}$$

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.

$$\begin{aligned} \epsilon &\sim \mathcal{N}(0, \mathbf{1}) \\ z &= \mu_\phi + \epsilon \cdot \sigma_\phi \end{aligned} \tag{2}$$

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.[1] 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:

$$\begin{aligned} \frac{dL}{d\mu_\phi} &= \frac{dL}{dz}\frac{dz}{d\mu_\phi} = \frac{dL}{dz} \\ \frac{dL}{d\sigma_\phi} &= \frac{dL}{dz}\frac{dz}{d\sigma_\phi} = \frac{dL}{dz}\epsilon \end{aligned} \tag{3}$$

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

$$\begin{aligned} \mu_\phi, \log \sigma^2_\phi &= \texttt{encoder(FASTA)} \\ \sigma_\phi &= \exp\left(\frac{1}{2}\log \sigma^2_\phi\right) \end{aligned} \tag{4}$$

First, implement the function to transform $\log \sigma^2_\phi$ to σ in `ProteinVAE().get_std()` (as in equation 4)). Then, implement the reparametrization trick in the `ProteinVAE().reparametrize()`

[1] 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].

function which takes two inputs: μ_ϕ and σ_ϕ and outputs a latent vector z as in equation 2 (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 (10 points) Implement the protein 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 FASTA sequence. 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.

$$\begin{aligned} L &= L_{recon} + \beta L_{regularization} \\ &= \int dz\, Q_\phi(z|x) \log P_\theta(x|z) + \beta \int dz\, p(z) \log \frac{p(z)}{Q_\phi(z|x)} \end{aligned} \tag{5}$$

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.

$$L_{recon} = -\frac{1}{N_{seq} N_{char}} \sum_{i}^{N_{seq}} \sum_{k}^{N_{char}} p_{data}(\hat{x}_{i,k}) \log(p(x_{i,k})) \tag{6}$$

$\log(p(x_{i,k}))$ is the logit for each possible character category reconstructed by the decoder, $\hat{x}_{i,k}$ is the original protein FASTA; 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 FASTA sequence (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 FASTA sequence (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 FASTA 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):

$$
\begin{aligned}
L_{regularization} &= KL(Q_\phi(z|x)|p(z)) \\
&= KL\left(\mathcal{N}(\mu_\phi(x_i), \sigma^2_\phi(x_i))|\mathcal{N}(0,1)\right) \\
&= \frac{1}{N_{batch}} \sum_i^{N_{batch}} \frac{1}{2} \left(\sum_d^{N_z} \sigma_{d,\phi}(x_i)^2 + \mu_{d,\phi}(x_i)^2 - \log \sigma_{d,\phi}(x_i)^2 - 1 \right)
\end{aligned} \tag{7}
$$

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

For this problem, you need to implement the reconstruction (equation 6) loss and the KL divergence (equation 7) in the `loss_function()` function.

Part 2.4 (5 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. Simply run the provided code chunks to train the VAE for 50 epochs. Make sure you obtain a training and test loss below about 1.6 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 35 minutes when run on the T4 GPU in Colab.

Part 2.5 (20 points) Sample new protein sequences

The latent space learned by the model is a 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 FASTA protein sequence. Now use your trained model to sample novel protein sequences from the learned distribution. Randomly select two protein sequences in your test data, and encode into latent vectors with the encoder and then linearly interpolate between the two protein sequences in the latent space to obtain 10 points in the z space. For each sampled latent code, decode z back to FASTA sequences. You can use the `index2fasta()` function we provided to convert categorical values to FASTA sequences.

Finally, produce a scatter plot with the first two dimensions of z of your test sequences and newly sampled sequences in the same figure. Color differently these test points and generated points. In a separate table, print the 10 generated FASTA sequences. Briefly comment on the sequence diversity you observe among these 10 generated sequences. How many are essentially identical? What do you think can be done to improve the model in its ability to finely interpolate sequences in this learned latent space?

References

[1] Popova, M., Isayev, O. & Tropsha, A. Deep reinforcement learning for de novo drug design. *Science Advances* **4** (2018).

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

[3] Hawkins-Hooker, A. *et al.* Generating functional protein variants with variational autoencoders. *PLOS Computational Biology* **17** (2021).

[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).