Technical University of Munich

ROSTLAB.

Chair for Bioinformatics and Computational Biology
Prof. B. Rost
L. Richter, D. Le

SoSe 2022
Exercise sheet 1
03.05.2022 – 10.05.2022

Advanced Bioinformatics SoSe 2022

From Sequence to Embedding

(E) Embeddings and Dataloader

1.1 (E) Dataloader

In this in-class exercise we will have a look at how to write a simple dataloader and discuss what some of the parameters do.

1.2 (E) Embeddings

In this in-class exercise we will grasp the idea behind protein sequence embeddings and see an example on how to create an embedding from a sequence.

(H) On-the-fly Embedder $(\sum = 15)$

In this homework you are asked to complete the program `exe1_dataloader.py`.
Remark, you might want to have a look at the types of the input and output of the function to get an idea how the function should behave. Furthermore, you are allowed to add helper functions if necessary but you may **not change** the **signature** of the given functions.

You might want to have a look at how to create ProtBert embedding. Additionally this resource on Dataloaders is also helpful.

1.3 (H) Loading Data ... (3)

Complete the function `get_dataloader` that should yield a dataloader that was created from the dataset *TokenSeqDataset*. For that, initialize the *TokenSeqDataset* accordingly. Set the *batch_size*

to the given value you receive as input to the function. Choose *shuffle* to be *True* and set the *collate_function* of the dataloader to be the function `collate_paired_sequences`.
Additionally use the *seed* to get a reproducible output from your dataloader.
→ Input:

- *fasta_path*, a path to a file containing protein sequences in FASTA format.
- *labels_dict*, a dictionary containing the identifiers of the protein sequences and their corresponding label for the classification task later on.
- *batch_size*, an integer indicating how many sequences should be grouped in one batch.
- *max_num_residues*, an integer stating the cutoff value for a sequence.
- *protbert_cache*, a path to a directory where the weights of ProtBert can be stored or reused.
- *device*, a value yielding the device your training will be executed on
- *seed*, an integer to fix the starting point of a random number generator.

1.4 (H) Collate Collaterals (2)

Complete the function `collate_paired_sequences` that has the responsibility to group your preliminary "raw" data into a group of embeddings and a group of their corresponding labels. Return a `Tuple[torch.Tensor, torch.Tensor]` with shapes $(batch\_size,\ 1024)$ and $(batch\_size,\ 1)$ respectively.
→ Input:

- *data*, a list containing tuples. Each tuple consists of a protein sequence embedding of shape $(1024,)$ and its label as scalar at position 0 and 1 respectively. The list will contain $batch\_size$ many tuples. `List[Tuple[torch.Tensor, int]]`

1.5 (H) Tokenizer and Model (2)

Complete the functions `load_tokenizer` and `load_model` that load the tokenizer of ProtBert and ProtBert itself. Use the `Rostlab/prot_bert_bfd` version. Send the model to the device that is given in `__init__`.
Remark, make sure you load ProtBert and its tokenizer during the creation of an instance of TokenSeqDataset and save it to the `self.protbert` and `self.tokenizer` respectively.

1.6 (H) Parsing Fast(a) (2)

Complete the function `parse_fasta_input` that receives a path to a file containing protein sequences in FASTA format and returns a dictionary where the sequence identifiers are the keys and the corresponding protein sequences are the values. You might want to have a look at `Fasta` class imported from pyfaidx.
Additionally, return the total number of sequences that were parsed in the function `__len__`. You might also want to save your sequence dictionary into `self.data`.

1.7 (H) An item please (2)

Complete the function ***__getitem__*** where you receive an integer indicating the position that

was selected. Match this index with your sequence id in your "sequence_id x actual_sequence" dictionary created from `parse_fasta_input` at its respective position.
Remark, in your FASTA-parsing process, make sure you insert the sequences in the same order as they are contained in the input FASTA file.
Truncate your found protein sequence string to the *max*imal *num*ber of *res*idues and tokenize your sequence with `tokenize`. Finally return a tuple containing the embedding of the sequence at position 0 and its corresponding label at position 1. For the creation of the embedding, you might want to use the `embedd` function with the *tokens* and the *attention mask* given by `tokenize`.

1.8 (H) Token after token (2)

Complete the function *tokenize* that receives a sequence string and returns a tuple of tensors consisting of first the token ids and second an attention mask. Remember to add spaces between the residues. Furthermore, if your given sequence is shorter than the value *max_num_res* pad your tokens and attention mask tensor to a tensor of length *max_num_res* with the *pad_token_id* of the tokenizer.
Remark, your output of this function should be `Tuple[torch.Tensor, torch.Tensor]` with the tensors having a shape of $(1,\ max\_num\_residues + 2)$. If you do not like the *pad_token_id* of the tokenizer, in this case it is just the number 0.

1.9 (H) Em.bed-time (2)

Complete the function `embedd` given the token tensor and the attention mask. Do a forward pass through the model you have loaded via `load_model` and create a per-protein/per-sequence embedding by averaging over the sequence length.
In the end, you should return a tensor of shape $(1024,)$.