НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Факультет прикладної математики Кафедра прикладної математики

Пояснювальна записка до курсового проекту

із дисципліни «Алгоритми і системи комп’ютерної математики»

Виконала: Керівник:

студентка групи КМ-81 *доцент Олефір О. Верзун Поліна*

Київ — 2021

Визначення теми

Назва курсового проекту

Моделювання та прогнозування впливу мутацій на протеїнові функції

Замовник

Компанія Facebook.

Програмне середовище

Мова програмування - Python.

Список вивченої літератури

1. Ahmed Elnaggar, Michael Heinzinger, Christian Dallago, Ghalia Rihawi, Yu Wang, Llion Jones, Tom Gibbs, Tamas Feher, Christoph Angerer, Martin Steinegger, Debsindhu Bhowmik, and Burkhard Rost. ProtTrans: Towards Cracking the Language of Life’s Code Through Self-Supervised Deep Learning and High Performance Computing. bioRxiv, 7 2020. URL <http://arxiv.org/abs/2007.06225.>
2. Chloe Hsu, Hunter Nisonoff, Clara Fannjiang, and Jennifer Listgarten. Combining evolutionary and assay-labelled data for protein fitness prediction. bioRxiv, page 2021.03.28.437402, березень 2021. doi: 10.1101/2021.03.28.437402.
3. Surojit Biswas, Grigory Khimulya, Ethan C Alley, Kevin M Esvelt, and George M Church. Low-N protein engineering with data-efficient deep learning. bioRxiv, сторінка 2020.01.23.917682, 2020. doi: 10.1101/2020.01.23.917682.
4. Christoph H Lampert, Hannes Nickisch, and Stefan Harmeling. Learning to detect unseen object classes by between-class attribute transfer. In 2009 IEEE Conference on Computer Vision and Pattern Recognition, сторінки 951–958. IEEE, 2009.
5. Hugo Larochelle, Dumitru Erhan, and Yoshua Bengio. Zero-data learning of new tasks. In AAAI, том 1, сторінка 3, 2008.
6. KM Reinish, L Chen, GL Verdine, and WN Lipscomb. The crystal structure of haeiii methyltransferase covalently complexed to dna: an extrahelical cytosine and rearranged base pairing. Cell, 82(1):143–153, 1995.
7. Michael Hennig, Beatrice Darimont, Reinhard Sterner, Kasper Kirschner, and Johan N Jansonius.2.0 å structure of indole-3-glycerol phosphate synthase from the hyperthermophile sulfolobus solfataricus: possible determinants of protein stability. Structure, 3(12):1295–1306, 1995.
8. Doris Forst, Wolfram Welte, Thomas Wacker, and Kay Diederichs. Structure of the sucrosespecific porin scry from salmonella typhimurium and its complex with sucrose. Nature structural biology, 5(1):37–46, 1998.
9. Tristan Bepler and Bonnie Berger. Learning protein sequence embeddings using information from structure, 2019. URL http://arxiv.org/abs/1902.08661https://arxiv.org/abs/1902.08661.
10. Amy X Lu, Haoran Zhang, Marzyeh Ghassemi, and Alan Moses. Self-Supervised Contrastive Learning of Protein Representations By Mutual Information Maximization. bioRxiv, page 2020.09.04.283929, 2020. doi: 10.1101/2020.09.04.283929. URL https://doi.org/10.1101/2020.09.04.283929.
11. Ali Madani, Bryan McCann, Nikhil Naik, Nitish Shirish Keskar, Namrata Anand, Raphael R. Eguchi, Po-Ssu Huang, and Richard Socher. ProGen: Language Modeling for Protein Generation. bioRxiv, 2020. URL http://arxiv.org/abs/2004.03497.
12. Jesse Vig, Ali Madani, Lav R. Varshney, Caiming Xiong, Richard Socher, and Nazneen Fatema Rajani. BERTology Meets Biology: Interpreting Attention in Protein Language Models. bioRxiv, сторінка 2020.06.26.174417, 6 2020. doi: 10.1101/2020.06.26.174417. URL http://arxiv.org/abs/2006.15222.
13. Kevin K. Yang, Zachary Wu, and Frances H. Arnold. Machine-learning-guided directed evolution for protein engineering, aug 2019. ISSN 15487105.
14. Haicang Zhang, Michelle S Xu, Wendy K Chung, and Yufeng Shen. Predicting functional effect of missense variants using graph attention neural networks. bioRxiv, 2021.

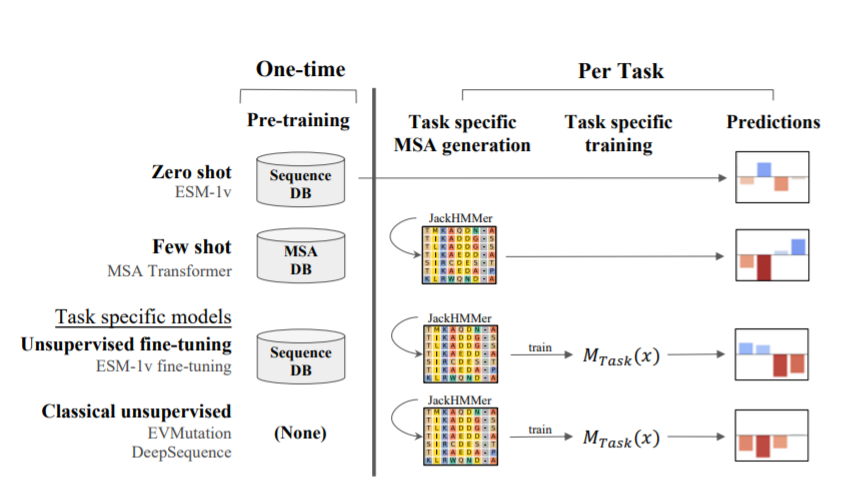
1. Adam Riesselman, Jung-Eun Shin, Aaron Kollasch, Conor McMahon, Elana Simon, Chris Sander, Aashish Manglik, Andrew Kruse, and Debora Marks. Accelerating Protein Design Using Autoregressive Generative Models. bioRxiv, сторінка 757252, 2019. doi: 10.1101/757252. URL https://www.biorxiv.org/content/10.1101/757252v1.

Методи

Мовні моделі для білків, навчені з функцією мети такою, як в класичній концепції “masked language modeling”, контролюються, щоб вивести ймовірність того, що амінокислота зустрічається на певній позиції в білку з огляду на навколишній контекст. Ця можливість використовується для оцінки варіацій послідовності амінокислот. Для даної мутації ми можемо розглядати амінокислоту в білку дикого типу як еталонний стан, порівнюючи ймовірність, присвоєну мутованій амінокислоті, з ймовірністю, призначеною для амінокислоти дикого типу.

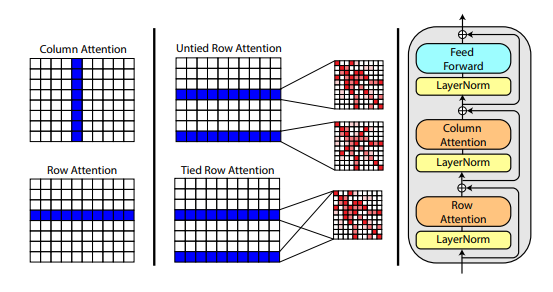
1. Zero-shot і few-shot трансформер

В умовах Zero-shot, передбачення виконується безпосередньо для оцінюваної послідовності. Оскільки MSA Transformer (multiple sequence alignment) може приймати декілька послідовностей як вхідні дані під час оцінки ймовірностей, ми використовуємо цю модель у режимі кількох кадрів, де надаються додаткові послідовності з сімейства білків разом із послідовністю, яку потрібно оцінити. Як у налаштуваннях zero-shot, так і few-shot під час висновку виконуються лише прямий прохід даних через моделі; жодних оновлень градієнта не робиться.



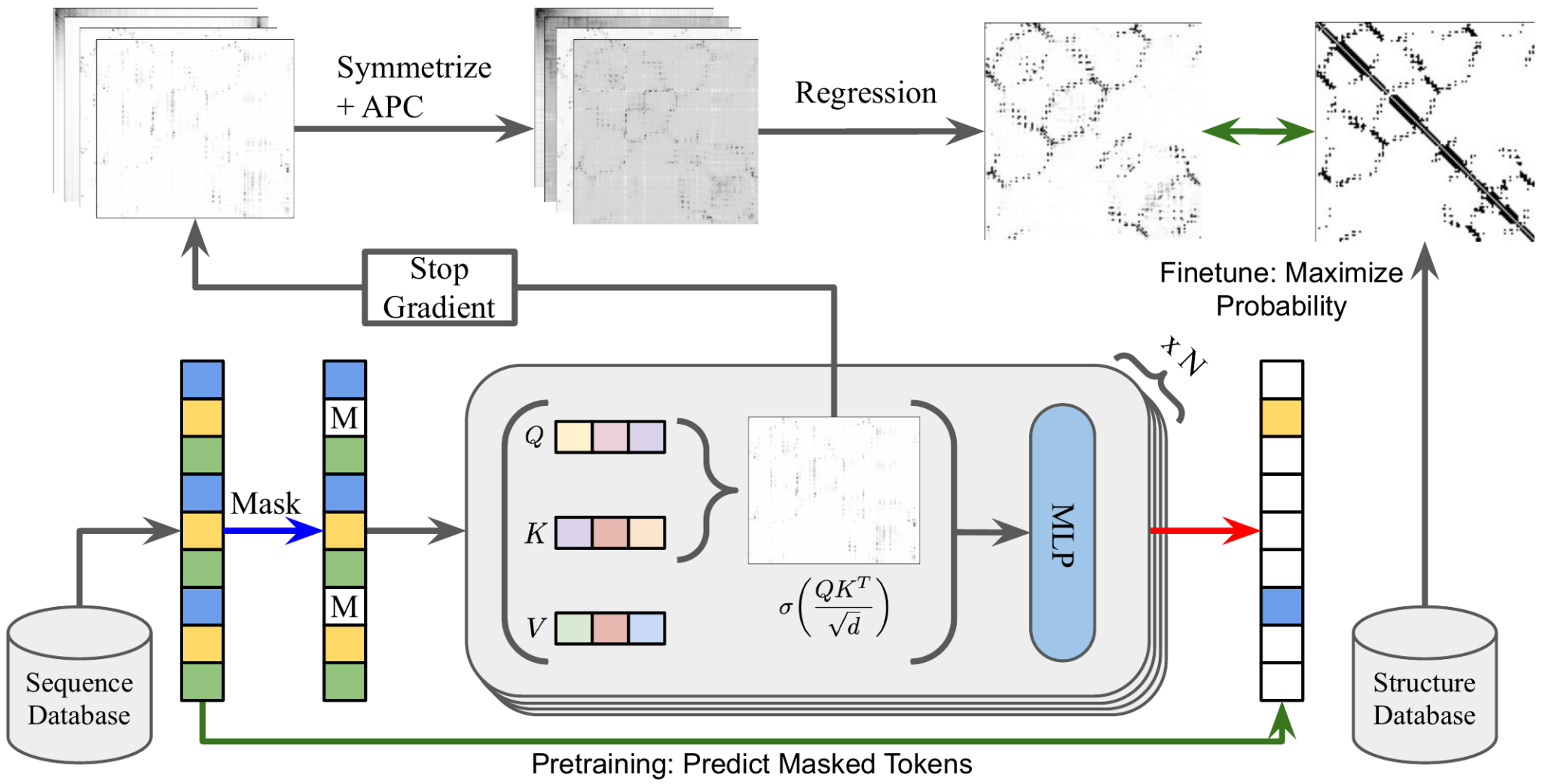
2. MSA Transformer

Така модель складається з декількох блоків наступної структури:



3. ESM Transformer

Трансформер з наступною архітектурою та схемою навчання:



Вибір, обґрунтування методу розв’язання задачі.

MSA Transformer 0.524

ESM-1v (zero shot) 0.482

ESM-1v (+further training) 0.519

Таблиця 1: Порівняння мовних моделей структури протеїну. Середній коефіцієнт Спірмана |Spearman ρ| на тестових данних.

Для отримання найкращих результатів було вирішено використати модель ESM тому що вона працює швидше і підтримує zero-shot метод обчислення ймовірностей, при цьому не сильно поступаючись у якості, як видно з таблиці

**Архітектура розроблюваних програмних засобів.**

Модель ESM була розширена до 650 мільйонів параметрів та натренована на 98 мільйонах різних протеїнових послідовностях разом з їх мутаціями та еволюціями.

Було вирішено тренувати модель лише на наявних протеїнових послідовностях, без використання методів навчання з учителем, натомість був використаний метод Masked LM. Суть метода полягає в тому, що маючи деяку послідовність, наприклад

VVFYFPDWTNHSDPNNNCRYCPFVDWAMGNMMTINDFHCDIIEMLICM

Під час навчання у цій послідовності “маскується” деяка її частка, наприклад:

VVFYFPDWTNHSDPNNNCRY[ ]VDWAMGNMMTINDFHCDIIEMLICM

Задача моделі полягає у тому, щоб вгадати замасковану частку, тобто CPF

Такий метод називається Masked Language Modelling тому що вперше був використаний саме в трансформерах, що моделювали живу мову. В такому варіанті маскувалося одне слово у реченні та модель повинна була його вгадали виходячи з контексту. Такий самий метод навчання використаний на послідовностях протеїнів теж дав несподівано хороший результат (наведено в попередньому етапі).

Для навчання було використано датасет послідовностей протеїнів UniRef90, він містить в собі близько 100 мільйонів послідовностей довжиною більше 11.

Частина датасету не була використана для навчання, замість того її було використано для обчислення перплексії моделі, яка мала значення 7.29.

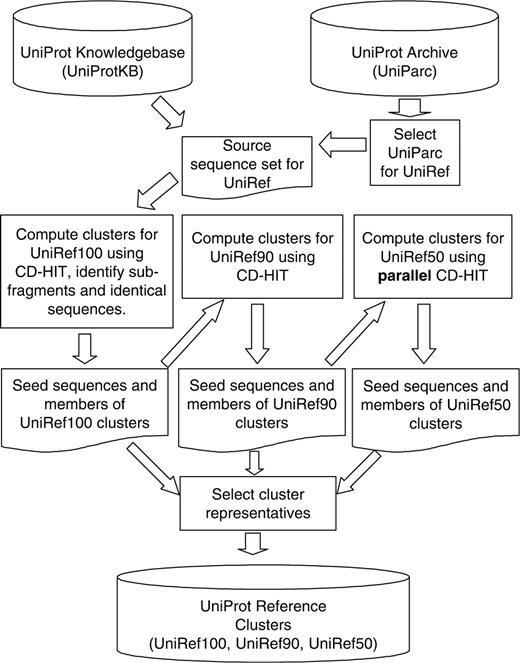
Для написання основної частини коду необхідна була робота з відеокартою серверу, на якому відбулося навчання, також була необхідність у реалізації тензорних операцій.

Для вирішення цих задач було використано Python фреймворк для моделювання, роботи з тензорами та з відеокартою pytorch версії 1.10

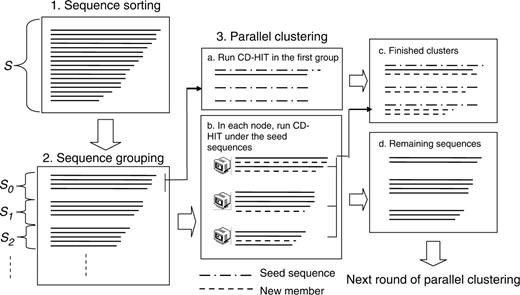
Було також використано pandas, numpy для роботи з даними і частина коду була оптимізована на мові програмування CUDA

**Склад та формат вхідних даних**

В цій роботі було використано датасет UniRef-90, що складається з декількох кластерів даних з мутаціями протеїнів. Цей датасет в свою чергу був отриманий в результаті процесу обробки даних з бази UniProtKB по схемі, зображеній на графіку:

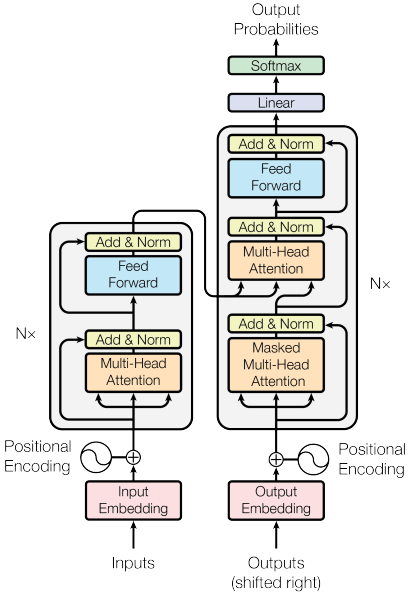


Більш детально процедура розділення даних на кластери зображена на наступному графіку:

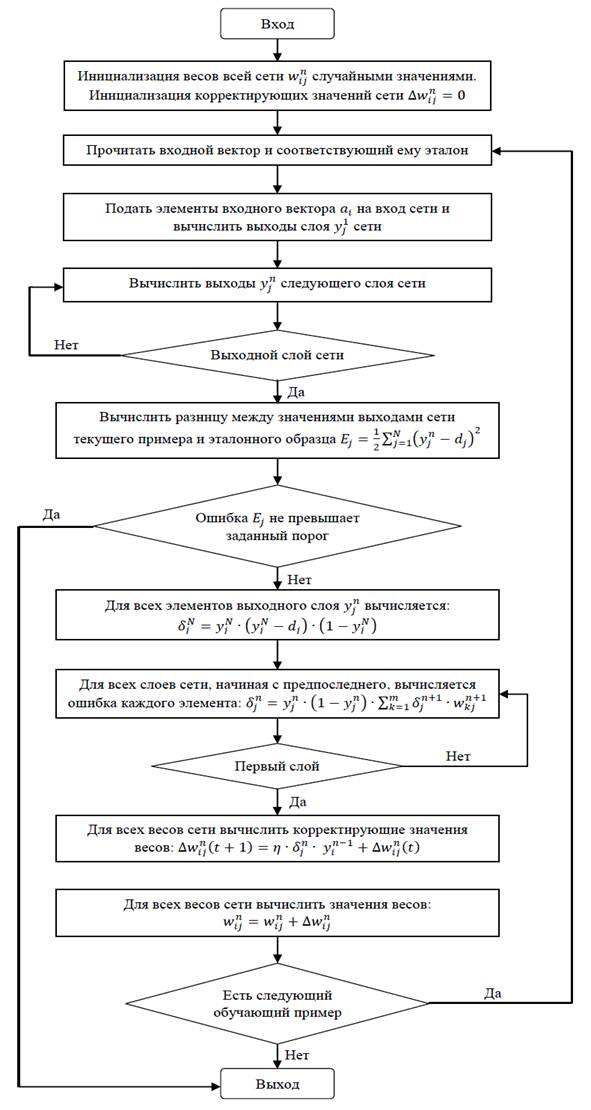


Розробка алгоритмів

Схема моделі

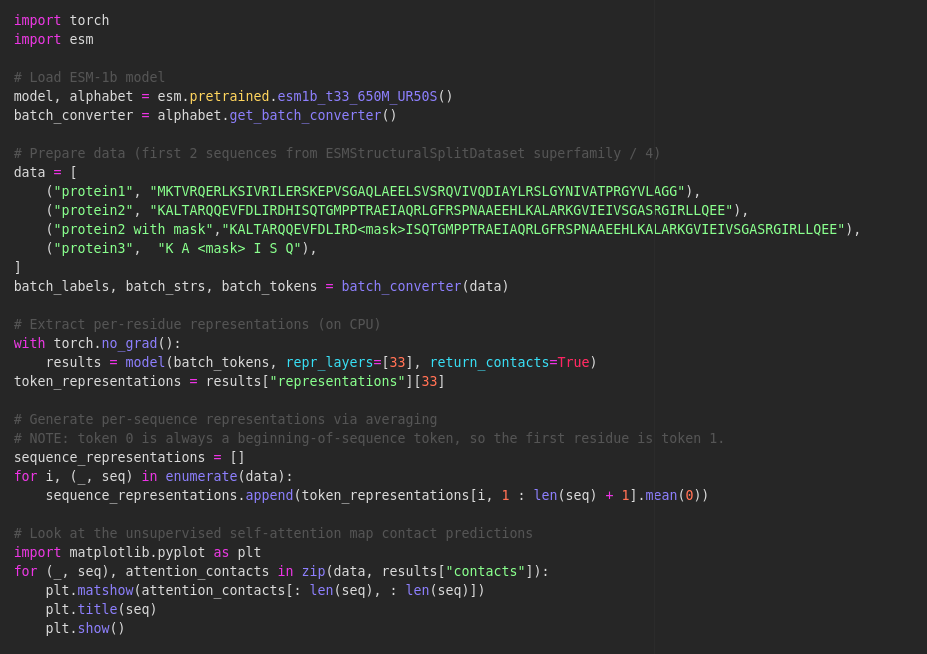


Блок-Схема навчання моделі



Розробка мови управління програмою

Для запуску навчених моделей і виконання передбачення мутації протеїнів можна застосувати класи та функції реалізовані для зручності користувача:



import torch

import esm

# Load ESM-1b model

model, alphabet = esm.pretrained.esm1b\_t33\_650M\_UR50S()

batch\_converter = alphabet.get\_batch\_converter()

# Prepare data (first 2 sequences from ESMStructuralSplitDataset superfamily / 4)

data = [

("protein1", "MKTVRQERLKSIVRILERSKEPVSGAQLAEELSVSRQVIVQDIAYLRSLGYNIVATPRGYVLAGG"),

("protein2", "KALTARQQEVFDLIRDHISQTGMPPTRAEIAQRLGFRSPNAAEEHLKALARKGVIEIVSGASRGIRLLQEE"),

("protein2 with mask","KALTARQQEVFDLIRD<mask>ISQTGMPPTRAEIAQRLGFRSPNAAEEHLKALARKGVIEIVSGASRGIRLLQEE"),

("protein3", "K A <mask> I S Q"),

]

batch\_labels, batch\_strs, batch\_tokens = batch\_converter(data)

# Extract per-residue representations (on CPU)

with torch.no\_grad():

results = model(batch\_tokens, repr\_layers=[33], return\_contacts=True)

token\_representations = results["representations"][33]

# Generate per-sequence representations via averaging

# NOTE: token 0 is always a beginning-of-sequence token, so the first residue is token 1.

sequence\_representations = []

for i, (\_, seq) in enumerate(data):

sequence\_representations.append(token\_representations[i, 1 : len(seq) + 1].mean(0))

# Look at the unsupervised self-attention map contact predictions

import matplotlib.pyplot as plt

for (\_, seq), attention\_contacts in zip(data, results["contacts"]):

plt.matshow(attention\_contacts[: len(seq), : len(seq)])

plt.title(seq)

plt.show()

Код Програми

import itertools

import os

from typing import Sequence, Tuple, List, Union

import pickle

import re

import shutil

import torch

from pathlib import Path

from .constants import proteinseq\_toks

RawMSA = Sequence[Tuple[str, str]]

class FastaBatchedDataset(object):

def \_\_init\_\_(self, sequence\_labels, sequence\_strs):

self.sequence\_labels = list(sequence\_labels)

self.sequence\_strs = list(sequence\_strs)

@classmethod

def from\_file(cls, fasta\_file):

sequence\_labels, sequence\_strs = [], []

cur\_seq\_label = None

buf = []

def \_flush\_current\_seq():

nonlocal cur\_seq\_label, buf

if cur\_seq\_label is None:

return

sequence\_labels.append(cur\_seq\_label)

sequence\_strs.append("".join(buf))

cur\_seq\_label = None

buf = []

with open(fasta\_file, "r") as infile:

for line\_idx, line in enumerate(infile):

if line.startswith(">"): # label line

\_flush\_current\_seq()

line = line[1:].strip()

if len(line) > 0:

cur\_seq\_label = line

else:

cur\_seq\_label = f"seqnum{line\_idx:09d}"

else: # sequence line

buf.append(line.strip())

\_flush\_current\_seq()

assert len(set(sequence\_labels)) == len(

sequence\_labels

), "Found duplicate sequence labels"

return cls(sequence\_labels, sequence\_strs)

def \_\_len\_\_(self):

return len(self.sequence\_labels)

def \_\_getitem\_\_(self, idx):

return self.sequence\_labels[idx], self.sequence\_strs[idx]

def get\_batch\_indices(self, toks\_per\_batch, extra\_toks\_per\_seq=0):

sizes = [(len(s), i) for i, s in enumerate(self.sequence\_strs)]

sizes.sort()

batches = []

buf = []

max\_len = 0

def \_flush\_current\_buf():

nonlocal max\_len, buf

if len(buf) == 0:

return

batches.append(buf)

buf = []

max\_len = 0

for sz, i in sizes:

sz += extra\_toks\_per\_seq

if max(sz, max\_len) \* (len(buf) + 1) > toks\_per\_batch:

\_flush\_current\_buf()

max\_len = max(max\_len, sz)

buf.append(i)

\_flush\_current\_buf()

return batches

class Alphabet(object):

def \_\_init\_\_(

self,

standard\_toks: Sequence[str],

prepend\_toks: Sequence[str] = ("<null\_0>", "<pad>", "<eos>", "<unk>"),

append\_toks: Sequence[str] = ("<cls>", "<mask>", "<sep>"),

prepend\_bos: bool = True,

append\_eos: bool = False,

use\_msa: bool = False,

):

self.standard\_toks = list(standard\_toks)

self.prepend\_toks = list(prepend\_toks)

self.append\_toks = list(append\_toks)

self.prepend\_bos = prepend\_bos

self.append\_eos = append\_eos

self.use\_msa = use\_msa

self.all\_toks = list(self.prepend\_toks)

self.all\_toks.extend(self.standard\_toks)

for i in range((8 - (len(self.all\_toks) % 8)) % 8):

self.all\_toks.append(f"<null\_{i + 1}>")

self.all\_toks.extend(self.append\_toks)

self.tok\_to\_idx = {tok: i for i, tok in enumerate(self.all\_toks)}

self.unk\_idx = self.tok\_to\_idx["<unk>"]

self.padding\_idx = self.get\_idx("<pad>")

self.cls\_idx = self.get\_idx("<cls>")

self.mask\_idx = self.get\_idx("<mask>")

self.eos\_idx = self.get\_idx("<eos>")

self.all\_special\_tokens = ['<eos>', '<unk>', '<pad>', '<cls>', '<mask>']

self.unique\_no\_split\_tokens = self.all\_toks

def \_\_len\_\_(self):

return len(self.all\_toks)

def get\_idx(self, tok):

return self.tok\_to\_idx.get(tok, self.unk\_idx)

def get\_tok(self, ind):

return self.all\_toks[ind]

def to\_dict(self):

return self.tok\_to\_idx.copy()

def get\_batch\_converter(self):

if self.use\_msa:

return MSABatchConverter(self)

else:

return BatchConverter(self)

@classmethod

def from\_architecture(cls, name: str) -> "Alphabet":

if name in ("ESM-1", "protein\_bert\_base"):

standard\_toks = proteinseq\_toks["toks"]

prepend\_toks: Tuple[str, ...] = ("<null\_0>", "<pad>", "<eos>", "<unk>")

append\_toks: Tuple[str, ...] = ("<cls>", "<mask>", "<sep>")

prepend\_bos = True

append\_eos = False

use\_msa = False

elif name in ("ESM-1b", "roberta\_large"):

standard\_toks = proteinseq\_toks["toks"]

prepend\_toks = ("<cls>", "<pad>", "<eos>", "<unk>")

append\_toks = ("<mask>",)

prepend\_bos = True

append\_eos = True

use\_msa = False

elif name in ("MSA Transformer", "msa\_transformer"):

standard\_toks = proteinseq\_toks["toks"]

prepend\_toks = ("<cls>", "<pad>", "<eos>", "<unk>")

append\_toks = ("<mask>",)

prepend\_bos = True

append\_eos = False

use\_msa = True

else:

raise ValueError("Unknown architecture selected")

return cls(standard\_toks, prepend\_toks, append\_toks, prepend\_bos, append\_eos, use\_msa)

def \_tokenize(self, text) -> str:

return text.split()

def tokenize(self, text, \*\*kwargs) -> List[str]:

"""

Inspired by https://github.com/huggingface/transformers/blob/master/src/transformers/tokenization\_utils.py

Converts a string in a sequence of tokens, using the tokenizer.

Args:

text (:obj:`str`):

The sequence to be encoded.

Returns:

:obj:`List[str]`: The list of tokens.

"""

def split\_on\_token(tok, text):

result = []

split\_text = text.split(tok)

for i, sub\_text in enumerate(split\_text):

# AddedToken can control whitespace stripping around them.

# We use them for GPT2 and Roberta to have different behavior depending on the special token

# Cf. https://github.com/huggingface/transformers/pull/2778

# and https://github.com/huggingface/transformers/issues/3788

# We strip left and right by default

if i < len(split\_text) - 1:

sub\_text = sub\_text.rstrip()

if i > 0:

sub\_text = sub\_text.lstrip()

if i == 0 and not sub\_text:

result.append(tok)

elif i == len(split\_text) - 1:

if sub\_text:

result.append(sub\_text)

else:

pass

else:

if sub\_text:

result.append(sub\_text)

result.append(tok)

return result

def split\_on\_tokens(tok\_list, text):

if not text.strip():

return []

tokenized\_text = []

text\_list = [text]

for tok in tok\_list:

tokenized\_text = []

for sub\_text in text\_list:

if sub\_text not in self.unique\_no\_split\_tokens:

tokenized\_text.extend(split\_on\_token(tok, sub\_text))

else:

tokenized\_text.append(sub\_text)

text\_list = tokenized\_text

return list(

itertools.chain.from\_iterable(

(

self.\_tokenize(token)

if token not in self.unique\_no\_split\_tokens

else [token]

for token in tokenized\_text

)

)

)

no\_split\_token = self.unique\_no\_split\_tokens

tokenized\_text = split\_on\_tokens(no\_split\_token, text)

return tokenized\_text

def encode(self, text):

return [self.tok\_to\_idx[tok] for tok in self.tokenize(text)]

class BatchConverter(object):

"""Callable to convert an unprocessed (labels + strings) batch to a

processed (labels + tensor) batch.

"""

def \_\_init\_\_(self, alphabet):

self.alphabet = alphabet

def \_\_call\_\_(self, raw\_batch: Sequence[Tuple[str, str]]):

# RoBERTa uses an eos token, while ESM-1 does not.

batch\_size = len(raw\_batch)

batch\_labels, seq\_str\_list = zip(\*raw\_batch)

seq\_encoded\_list = [self.alphabet.encode(seq\_str) for seq\_str in seq\_str\_list]

max\_len = max(len(seq\_encoded) for seq\_encoded in seq\_encoded\_list)

tokens = torch.empty(

(

batch\_size,

max\_len + int(self.alphabet.prepend\_bos) + int(self.alphabet.append\_eos),

),

dtype=torch.int64,

)

tokens.fill\_(self.alphabet.padding\_idx)

labels = []

strs = []

for i, (label, seq\_str, seq\_encoded) in enumerate(

zip(batch\_labels, seq\_str\_list, seq\_encoded\_list)

):

labels.append(label)

strs.append(seq\_str)

if self.alphabet.prepend\_bos:

tokens[i, 0] = self.alphabet.cls\_idx

seq = torch.tensor(seq\_encoded, dtype=torch.int64)

tokens[

i,

int(self.alphabet.prepend\_bos) : len(seq\_encoded)

+ int(self.alphabet.prepend\_bos),

] = seq

if self.alphabet.append\_eos:

tokens[i, len(seq\_encoded) + int(self.alphabet.prepend\_bos)] = self.alphabet.eos\_idx

return labels, strs, tokens

class MSABatchConverter(BatchConverter):

def \_\_call\_\_(self, inputs: Union[Sequence[RawMSA], RawMSA]):

if isinstance(inputs[0][0], str):

# Input is a single MSA

raw\_batch: Sequence[RawMSA] = [inputs] # type: ignore

else:

raw\_batch = inputs # type: ignore

batch\_size = len(raw\_batch)

max\_alignments = max(len(msa) for msa in raw\_batch)

max\_seqlen = max(len(msa[0][1]) for msa in raw\_batch)

tokens = torch.empty(

(

batch\_size,

max\_alignments,

max\_seqlen + int(self.alphabet.prepend\_bos) + int(self.alphabet.append\_eos),

),

dtype=torch.int64,

)

tokens.fill\_(self.alphabet.padding\_idx)

labels = []

strs = []

for i, msa in enumerate(raw\_batch):

msa\_seqlens = set(len(seq) for \_, seq in msa)

if not len(msa\_seqlens) == 1:

raise RuntimeError(

"Received unaligned sequences for input to MSA, all sequence "

"lengths must be equal."

)

msa\_labels, msa\_strs, msa\_tokens = super().\_\_call\_\_(msa)

labels.append(msa\_labels)

strs.append(msa\_strs)

tokens[i, : msa\_tokens.size(0), : msa\_tokens.size(1)] = msa\_tokens

return labels, strs, tokens

def read\_fasta(

path,

keep\_gaps=True,

keep\_insertions=True,

to\_upper=False,

):

with open(path, "r") as f:

for result in read\_alignment\_lines(

f, keep\_gaps=keep\_gaps, keep\_insertions=keep\_insertions, to\_upper=to\_upper

):

yield result

def read\_alignment\_lines(

lines,

keep\_gaps=True,

keep\_insertions=True,

to\_upper=False,

):

seq = desc = None

def parse(s):

if not keep\_gaps:

s = re.sub("-", "", s)

if not keep\_insertions:

s = re.sub("[a-z]", "", s)

return s.upper() if to\_upper else s

for line in lines:

# Line may be empty if seq % file\_line\_width == 0

if len(line) > 0 and line[0] == ">":

if seq is not None:

yield desc, parse(seq)

desc = line.strip()

seq = ""

else:

assert isinstance(seq, str)

seq += line.strip()

assert isinstance(seq, str) and isinstance(desc, str)

yield desc, parse(seq)

class ESMStructuralSplitDataset(torch.utils.data.Dataset):

"""

Structural Split Dataset as described in section A.10 of the supplement of our paper.

https://doi.org/10.1101/622803

We use the full version of SCOPe 2.07, clustered at 90% sequence identity,

generated on January 23, 2020.

For each SCOPe domain:

- We extract the sequence from the corresponding PDB file

- We extract the 3D coordinates of the Carbon beta atoms, aligning them

to the sequence. We put NaN where Cb atoms are missing.

- From the 3D coordinates, we calculate a pairwise distance map, based

on L2 distance

- We use DSSP to generate secondary structure labels for the corresponding

PDB file. This is also aligned to the sequence. We put - where SSP

labels are missing.

For each SCOPe classification level of family/superfamily/fold (in order of difficulty),

we have split the data into 5 partitions for cross validation. These are provided

in a downloaded splits folder, in the format:

splits/{split\_level}/{cv\_partition}/{train|valid}.txt

where train is the partition and valid is the concatentation of the remaining 4.

For each SCOPe domain, we provide a pkl dump that contains:

- seq : The domain sequence, stored as an L-length string

- ssp : The secondary structure labels, stored as an L-length string

- dist : The distance map, stored as an LxL numpy array

- coords : The 3D coordinates, stored as an Lx3 numpy array

"""

base\_folder = "structural-data"

file\_list = [

# url tar filename filename MD5 Hash

(

"https://dl.fbaipublicfiles.com/fair-esm/structural-data/splits.tar.gz",

"splits.tar.gz",

"splits",

"456fe1c7f22c9d3d8dfe9735da52411d",

),

(

"https://dl.fbaipublicfiles.com/fair-esm/structural-data/pkl.tar.gz",

"pkl.tar.gz",

"pkl",

"644ea91e56066c750cd50101d390f5db",

),

]

def \_\_init\_\_(

self,

split\_level,

cv\_partition,

split,

root\_path=os.path.expanduser("~/.cache/torch/data/esm"),

download=False,

):

super().\_\_init\_\_()

assert split in [

"train",

"valid",

], "train\_valid must be 'train' or 'valid'"

self.root\_path = root\_path

self.base\_path = os.path.join(self.root\_path, self.base\_folder)

# check if root path has what you need or else download it

if download:

self.download()

self.split\_file = os.path.join(

self.base\_path, "splits", split\_level, cv\_partition, f"{split}.txt"

)

self.pkl\_dir = os.path.join(self.base\_path, "pkl")

self.names = []

with open(self.split\_file) as f:

self.names = f.read().splitlines()

def \_\_len\_\_(self):

return len(self.names)

def \_check\_exists(self) -> bool:

for (\_, \_, filename, \_) in self.file\_list:

fpath = os.path.join(self.base\_path, filename)

if not os.path.exists(fpath) or not os.path.isdir(fpath):

return False

return True

def download(self):

if self.\_check\_exists():

print("Files already downloaded and verified")

return

from torchvision.datasets.utils import download\_url

for url, tar\_filename, filename, md5\_hash in self.file\_list:

download\_path = os.path.join(self.base\_path, tar\_filename)

download\_url(url=url, root=self.base\_path, filename=tar\_filename, md5=md5\_hash)

shutil.unpack\_archive(download\_path, self.base\_path)

def \_\_getitem\_\_(self, idx):

"""

Returns a dict with the following entires

- seq : Str (domain sequence)

- ssp : Str (SSP labels)

- dist : np.array (distance map)

- coords : np.array (3D coordinates)

"""

name = self.names[idx]

pkl\_fname = os.path.join(self.pkl\_dir, name[1:3], f"{name}.pkl")

with open(pkl\_fname, "rb") as f:

obj = pickle.load(f)

return obj

import math

import torch

import torch.nn as nn

import torch.nn.functional as F

from .modules import (

TransformerLayer,

AxialTransformerLayer,

LearnedPositionalEmbedding,

SinusoidalPositionalEmbedding,

RobertaLMHead,

ESM1bLayerNorm,

ContactPredictionHead,

)

from .axial\_attention import RowSelfAttention, ColumnSelfAttention

class ProteinBertModel(nn.Module):

@classmethod

def add\_args(cls, parser):

parser.add\_argument(

"--num\_layers", default=36, type=int, metavar="N", help="number of layers"

)

parser.add\_argument(

"--embed\_dim", default=1280, type=int, metavar="N", help="embedding dimension"

)

parser.add\_argument(

"--logit\_bias", action="store\_true", help="whether to apply bias to logits"

)

parser.add\_argument(

"--ffn\_embed\_dim",

default=5120,

type=int,

metavar="N",

help="embedding dimension for FFN",

)

parser.add\_argument(

"--attention\_heads",

default=20,

type=int,

metavar="N",

help="number of attention heads",

)

def \_\_init\_\_(self, args, alphabet):

super().\_\_init\_\_()

self.args = args

self.alphabet\_size = len(alphabet)

self.padding\_idx = alphabet.padding\_idx

self.mask\_idx = alphabet.mask\_idx

self.cls\_idx = alphabet.cls\_idx

self.eos\_idx = alphabet.eos\_idx

self.prepend\_bos = alphabet.prepend\_bos

self.append\_eos = alphabet.append\_eos

self.emb\_layer\_norm\_before = getattr(self.args, "emb\_layer\_norm\_before", False)

if self.args.arch == "roberta\_large":

self.model\_version = "ESM-1b"

self.\_init\_submodules\_esm1b()

else:

self.model\_version = "ESM-1"

self.\_init\_submodules\_esm1()

def \_init\_submodules\_common(self):

self.embed\_tokens = nn.Embedding(

self.alphabet\_size, self.args.embed\_dim, padding\_idx=self.padding\_idx

)

self.layers = nn.ModuleList(

[

TransformerLayer(

self.args.embed\_dim,

self.args.ffn\_embed\_dim,

self.args.attention\_heads,

add\_bias\_kv=(self.model\_version != "ESM-1b"),

use\_esm1b\_layer\_norm=(self.model\_version == "ESM-1b"),

)

for \_ in range(self.args.layers)

]

)

self.contact\_head = ContactPredictionHead(

self.args.layers \* self.args.attention\_heads,

self.prepend\_bos,

self.append\_eos,

eos\_idx=self.eos\_idx,

)

def \_init\_submodules\_esm1b(self):

self.\_init\_submodules\_common()

self.embed\_scale = 1

self.embed\_positions = LearnedPositionalEmbedding(

self.args.max\_positions, self.args.embed\_dim, self.padding\_idx

)

self.emb\_layer\_norm\_before = (

ESM1bLayerNorm(self.args.embed\_dim) if self.emb\_layer\_norm\_before else None

)

self.emb\_layer\_norm\_after = ESM1bLayerNorm(self.args.embed\_dim)

self.lm\_head = RobertaLMHead(

embed\_dim=self.args.embed\_dim,

output\_dim=self.alphabet\_size,

weight=self.embed\_tokens.weight,

)

def \_init\_submodules\_esm1(self):

self.\_init\_submodules\_common()

self.embed\_scale = math.sqrt(self.args.embed\_dim)

self.embed\_positions = SinusoidalPositionalEmbedding(self.args.embed\_dim, self.padding\_idx)

self.embed\_out = nn.Parameter(torch.zeros((self.alphabet\_size, self.args.embed\_dim)))

self.embed\_out\_bias = None

if self.args.final\_bias:

self.embed\_out\_bias = nn.Parameter(torch.zeros(self.alphabet\_size))

def forward(self, tokens, repr\_layers=[], need\_head\_weights=False, return\_contacts=False):

if return\_contacts:

need\_head\_weights = True

assert tokens.ndim == 2

padding\_mask = tokens.eq(self.padding\_idx) # B, T

x = self.embed\_scale \* self.embed\_tokens(tokens)

if getattr(self.args, "token\_dropout", False):

x.masked\_fill\_((tokens == self.mask\_idx).unsqueeze(-1), 0.0)

# x: B x T x C

mask\_ratio\_train = 0.15 \* 0.8

src\_lengths = (~padding\_mask).sum(-1)

mask\_ratio\_observed = (tokens == self.mask\_idx).sum(-1).float() / src\_lengths

x = x \* (1 - mask\_ratio\_train) / (1 - mask\_ratio\_observed)[:, None, None]

x = x + self.embed\_positions(tokens)

if self.model\_version == "ESM-1b":

if self.emb\_layer\_norm\_before:

x = self.emb\_layer\_norm\_before(x)

if padding\_mask is not None:

x = x \* (1 - padding\_mask.unsqueeze(-1).type\_as(x))

repr\_layers = set(repr\_layers)

hidden\_representations = {}

if 0 in repr\_layers:

hidden\_representations[0] = x

if need\_head\_weights:

attn\_weights = []

# (B, T, E) => (T, B, E)

x = x.transpose(0, 1)

if not padding\_mask.any():

padding\_mask = None

for layer\_idx, layer in enumerate(self.layers):

x, attn = layer(

x, self\_attn\_padding\_mask=padding\_mask, need\_head\_weights=need\_head\_weights

)

if (layer\_idx + 1) in repr\_layers:

hidden\_representations[layer\_idx + 1] = x.transpose(0, 1)

if need\_head\_weights:

# (H, B, T, T) => (B, H, T, T)

attn\_weights.append(attn.transpose(1, 0))

if self.model\_version == "ESM-1b":

x = self.emb\_layer\_norm\_after(x)

x = x.transpose(0, 1) # (T, B, E) => (B, T, E)

# last hidden representation should have layer norm applied

if (layer\_idx + 1) in repr\_layers:

hidden\_representations[layer\_idx + 1] = x

x = self.lm\_head(x)

else:

x = F.linear(x, self.embed\_out, bias=self.embed\_out\_bias)

x = x.transpose(0, 1) # (T, B, E) => (B, T, E)

result = {"logits": x, "representations": hidden\_representations}

if need\_head\_weights:

# attentions: B x L x H x T x T

attentions = torch.stack(attn\_weights, 1)

if self.model\_version == "ESM-1":

# ESM-1 models have an additional null-token for attention, which we remove

attentions = attentions[..., :-1]

if padding\_mask is not None:

attention\_mask = 1 - padding\_mask.type\_as(attentions)

attention\_mask = attention\_mask.unsqueeze(1) \* attention\_mask.unsqueeze(2)

attentions = attentions \* attention\_mask[:, None, None, :, :]

result["attentions"] = attentions

if return\_contacts:

contacts = self.contact\_head(tokens, attentions)

result["contacts"] = contacts

return result

def predict\_contacts(self, tokens):

return self(tokens, return\_contacts=True)["contacts"]

@property

def num\_layers(self):

return self.args.layers

class MSATransformer(nn.Module):

@classmethod

def add\_args(cls, parser):

# fmt: off

parser.add\_argument(

"--num\_layers",

default=12,

type=int,

metavar="N",

help="number of layers"

)

parser.add\_argument(

"--embed\_dim",

default=768,

type=int,

metavar="N",

help="embedding dimension"

)

parser.add\_argument(

"--logit\_bias",

action="store\_true",

help="whether to apply bias to logits"

)

parser.add\_argument(

"--ffn\_embed\_dim",

default=3072,

type=int,

metavar="N",

help="embedding dimension for FFN",

)

parser.add\_argument(

"--attention\_heads",

default=12,

type=int,

metavar="N",

help="number of attention heads",

)

parser.add\_argument(

"--dropout",

default=0.1,

type=float,

help="Dropout to apply."

)

parser.add\_argument(

"--attention\_dropout",

default=0.1,

type=float,

help="Dropout to apply."

)

parser.add\_argument(

"--activation\_dropout",

default=0.1,

type=float,

help="Dropout to apply."

)

parser.add\_argument(

"--max\_tokens\_per\_msa",

default=2 \*\* 14,

type=int,

help=(

"Used during inference to batch attention computations in a single "

"forward pass. This allows increased input sizes with less memory."

),

)

# fmt: on

def \_\_init\_\_(self, args, alphabet):

super().\_\_init\_\_()

self.args = args

self.alphabet\_size = len(alphabet)

self.padding\_idx = alphabet.padding\_idx

self.mask\_idx = alphabet.mask\_idx

self.cls\_idx = alphabet.cls\_idx

self.eos\_idx = alphabet.eos\_idx

self.prepend\_bos = alphabet.prepend\_bos

self.append\_eos = alphabet.append\_eos

self.embed\_tokens = nn.Embedding(

self.alphabet\_size, self.args.embed\_dim, padding\_idx=self.padding\_idx

)

if getattr(self.args, "embed\_positions\_msa", False):

emb\_dim = getattr(self.args, "embed\_positions\_msa\_dim", self.args.embed\_dim)

self.msa\_position\_embedding = nn.Parameter(

0.01 \* torch.randn(1, 1024, 1, emb\_dim),

requires\_grad=True,

)

else:

self.register\_parameter("msa\_position\_embedding", None)

self.dropout\_module = nn.Dropout(self.args.dropout)

self.layers = nn.ModuleList(

[

AxialTransformerLayer(

self.args.embed\_dim,

self.args.ffn\_embed\_dim,

self.args.attention\_heads,

self.args.dropout,

self.args.attention\_dropout,

self.args.activation\_dropout,

getattr(self.args, "max\_tokens\_per\_msa", self.args.max\_tokens),

)

for \_ in range(self.args.layers)

]

)

self.contact\_head = ContactPredictionHead(

self.args.layers \* self.args.attention\_heads,

self.prepend\_bos,

self.append\_eos,

eos\_idx=self.eos\_idx,

)

self.embed\_positions = LearnedPositionalEmbedding(

self.args.max\_positions,

self.args.embed\_dim,

self.padding\_idx,

)

self.emb\_layer\_norm\_before = ESM1bLayerNorm(self.args.embed\_dim)

self.emb\_layer\_norm\_after = ESM1bLayerNorm(self.args.embed\_dim)

self.lm\_head = RobertaLMHead(

embed\_dim=self.args.embed\_dim,

output\_dim=self.alphabet\_size,

weight=self.embed\_tokens.weight,

)

def forward(self, tokens, repr\_layers=[], need\_head\_weights=False, return\_contacts=False):

if return\_contacts:

need\_head\_weights = True

assert tokens.ndim == 3

batch\_size, num\_alignments, seqlen = tokens.size()

padding\_mask = tokens.eq(self.padding\_idx) # B, R, C

if not padding\_mask.any():

padding\_mask = None

x = self.embed\_tokens(tokens)

x += self.embed\_positions(tokens.view(batch\_size \* num\_alignments, seqlen)).view(x.size())

if self.msa\_position\_embedding is not None:

if x.size(1) > 1024:

raise RuntimeError(

"Using model with MSA position embedding trained on maximum MSA "

f"depth of 1024, but received {x.size(1)} alignments."

)

x += self.msa\_position\_embedding[:, :num\_alignments]

x = self.emb\_layer\_norm\_before(x)

x = self.dropout\_module(x)

if padding\_mask is not None:

x = x \* (1 - padding\_mask.unsqueeze(-1).type\_as(x))

repr\_layers = set(repr\_layers)

hidden\_representations = {}

if 0 in repr\_layers:

hidden\_representations[0] = x

if need\_head\_weights:

row\_attn\_weights = []

col\_attn\_weights = []

# B x R x C x D -> R x C x B x D

x = x.permute(1, 2, 0, 3)

for layer\_idx, layer in enumerate(self.layers):

x = layer(

x,

self\_attn\_padding\_mask=padding\_mask,

need\_head\_weights=need\_head\_weights,

)

if need\_head\_weights:

x, col\_attn, row\_attn = x

# H x C x B x R x R -> B x H x C x R x R

col\_attn\_weights.append(col\_attn.permute(2, 0, 1, 3, 4))

# H x B x C x C -> B x H x C x C

row\_attn\_weights.append(row\_attn.permute(1, 0, 2, 3))

if (layer\_idx + 1) in repr\_layers:

hidden\_representations[layer\_idx + 1] = x.permute(2, 0, 1, 3)

x = self.emb\_layer\_norm\_after(x)

x = x.permute(2, 0, 1, 3) # R x C x B x D -> B x R x C x D

# last hidden representation should have layer norm applied

if (layer\_idx + 1) in repr\_layers:

hidden\_representations[layer\_idx + 1] = x

x = self.lm\_head(x)

result = {"logits": x, "representations": hidden\_representations}

if need\_head\_weights:

# col\_attentions: B x L x H x C x R x R

col\_attentions = torch.stack(col\_attn\_weights, 1)

# row\_attentions: B x L x H x C x C

row\_attentions = torch.stack(row\_attn\_weights, 1)

result["col\_attentions"] = col\_attentions

result["row\_attentions"] = row\_attentions

if return\_contacts:

contacts = self.contact\_head(tokens, row\_attentions)

result["contacts"] = contacts

return result

def predict\_contacts(self, tokens):

return self(tokens, return\_contacts=True)["contacts"]

@property

def num\_layers(self):

return self.args.layers

def max\_tokens\_per\_msa\_(self, value: int) -> None:

"""The MSA Transformer automatically batches attention computations when

gradients are disabled to allow you to pass in larger MSAs at test time than

you can fit in GPU memory. By default this occurs when more than 2^14 tokens

are passed in the input MSA. You can set this value to infinity to disable

this behavior.

"""

for module in self.modules():

if isinstance(module, (RowSelfAttention, ColumnSelfAttention)):

module.max\_tokens\_per\_msa = value

Висновки

В результаті ми отримали готову програмну реалізацію моделювання та прогнозування впливу мутацій на протеїнові функції. Було розроблено алгоритм та мову управління програмою. Програму було реалізовано за допомогою фреймворку pytorch. Більш того, хочеться зазначити, що

реалізована модель працює адекватно згідно з оцінки середнього коефіцієнту Спірмана порівняно з іншими методами, тобто прогнозування відбувається передбачувано.