RNA_code_train_Data_

```
import pandas as pd
import os, gc
import numpy as np
from sklearn.model_selection import KFold

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader

import torch
from fastai.vision.all import *
def flatten(o):
    "Concatenate all collections and items as a generator"
    for item in o:
        if isinstance(o, dict): yield o[item]; continue
        elif isinstance(item, str): yield item; continue
        try: yield from flatten(item)
        except TypeError: yield item

from torch.cuda.amp import GradScaler, autocast
@delegates(GradScaler)
class MixedPrecision(Callback):
    "Mixed precision training using Pytorch's `autocast` and `GradScaler`"
    order = 10
    def __init__(self, **kwargs): self.kwargs = kwargs
    def before_fit(self):
        self.autocast,self.learn.scaler,self.scales = autocast(),GradScaler(**self.kwargs),L()
    def before_batch(self): self.autocast.__enter__()
    def after_pred(self):
        if next(flatten(self.pred)).dtype==torch.float16: self.learn.pred = to_float(self.pred)
    def after_loss(self): self.autocast.__exit__(None, None, None)
    def before_backward(self): self.learn.loss_grad = self.scaler.scale(self.loss_grad)
    def before_step(self):
        "Use `self` as a fake optimizer. `self.skipped` will be set to True `after_step` if gradients overflow
        self.skipped=True
        self.scaler.step(self)
        if self.skipped: raise CancelStepException()
        self.scales.append(self.scaler.get_scale())
    def after_step(self): self.learn.scaler.update()

    @property
    def param_groups(self):
        "Pretend to be an optimizer for `GradScaler`"
        return self.opt.param_groups
    def step(self, *args, **kwargs):
        "Fake optimizer step to detect whether this batch was skipped from `GradScaler`"
        self.skipped=False
    def after_fit(self): self.autocast,self.learn.scaler,self.scales = None,None,None

import fastai
fastai.callback.fp16.MixedPrecision = MixedPrecision

def seed_everything(seed):
    random.seed(seed)
    os.environ['PYTHONHASHSEED'] = str(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = True

#fname = 'example0'
#PATH = '/kaggle/input/stanford-ribonanza-rna-folding-converted/'
OUT = './'
bs = 256
```

```
num_workers = 2
SEED = 2023
nfolds = 4
device = 'cuda' if torch.cuda.is_available() else 'cpu'

class RNA_Dataset(Dataset):
    def __init__(self, df, mode='train', seed=2023, fold=0, nfolds=4,
                 mask_only=False, **kwargs):
        self.seq_map = {'A':0,'C':1,'G':2,'U':3}
        self.Lmax = 206
        df['L'] = df.sequence.apply(len)
        df_2A3 = df.loc[df.experiment_type=='2A3_MaP']
        df_DMS = df.loc[df.experiment_type=='DMS_MaP']

        split = list(KFold(n_splits=nfolds, random_state=seed,
                shuffle=True).split(df_2A3))[fold][0 if mode=='train' else 1]
        df_2A3 = df_2A3.iloc[split].reset_index(drop=True)
        df_DMS = df_DMS.iloc[split].reset_index(drop=True)

        m = (df_2A3['SN_filter'].values > 0) & (df_DMS['SN_filter'].values > 0)
        df_2A3 = df_2A3.loc[m].reset_index(drop=True)
        df_DMS = df_DMS.loc[m].reset_index(drop=True)

        self.seq = df_2A3['sequence'].values
        self.L = df_2A3['L'].values

        self.react_2A3 = df_2A3[[c for c in df_2A3.columns if \
                                 'reactivity_0' in c]].values
        self.react_DMS = df_DMS[[c for c in df_DMS.columns if \
                                 'reactivity_0' in c]].values
        self.react_err_2A3 = df_2A3[[c for c in df_2A3.columns if \
                                 'reactivity_error_0' in c]].values
        self.react_err_DMS = df_DMS[[c for c in df_DMS.columns if \
                                'reactivity_error_0' in c]].values
        self.sn_2A3 = df_2A3['signal_to_noise'].values
        self.sn_DMS = df_DMS['signal_to_noise'].values
        self.mask_only = mask_only

    def __len__(self):
        return len(self.seq)

    def __getitem__(self, idx):
        seq = self.seq[idx]
        if self.mask_only:
            mask = torch.zeros(self.Lmax, dtype=torch.bool)
            mask[:len(seq)] = True
            return {'mask':mask},{'mask':mask}
        seq = [self.seq_map[s] for s in seq]
        seq = np.array(seq)
        mask = torch.zeros(self.Lmax, dtype=torch.bool)
        mask[:len(seq)] = True
        seq = np.pad(seq,(0,self.Lmax-len(seq)))

        react = torch.from_numpy(np.stack([self.react_2A3[idx],
                                           self.react_DMS[idx]],-1))
        react_err = torch.from_numpy(np.stack([self.react_err_2A3[idx],
                                               self.react_err_DMS[idx]],-1))
        sn = torch.FloatTensor([self.sn_2A3[idx],self.sn_DMS[idx]])

        return {'seq':torch.from_numpy(seq), 'mask':mask}, \
               {'react':react, 'react_err':react_err,
                'sn':sn, 'mask':mask}

class LenMatchBatchSampler(torch.utils.data.BatchSampler):
    def __iter__(self):
        buckets = [[]] * 100
        yielded = 0

        for idx in self.sampler:
            s = self.sampler.data_source[idx]
            if isinstance(s,tuple): L = s[0]["mask"].sum()
```

```
            else: L = s["mask"].sum()
            L = max(1,L // 16)
            if len(buckets[L]) == 0:  buckets[L] = []
            buckets[L].append(idx)

            if len(buckets[L]) == self.batch_size:
                batch = list(buckets[L])
                yield batch
                yielded += 1
                buckets[L] = []

        batch = []
        leftover = [idx for bucket in buckets for idx in bucket]

        for idx in leftover:
            batch.append(idx)
            if len(batch) == self.batch_size:
                yielded += 1
                yield batch
                batch = []

        if len(batch) > 0 and not self.drop_last:
            yielded += 1
            yield batch

def dict_to(x, device='cuda'):
    return {k:x[k].to(device) for k in x}

def to_device(x, device='cuda'):
    return tuple(dict_to(e,device) for e in x)

class DeviceDataLoader:
    def __init__(self, dataloader, device='cuda'):
        self.dataloader = dataloader
        self.device = device

    def __len__(self):
        return len(self.dataloader)

    def __iter__(self):
        for batch in self.dataloader:
            yield tuple(dict_to(x, self.device) for x in batch)

class SinusoidalPosEmb(nn.Module):
    def __init__(self, dim=16, M=10000):
        super().__init__()
        self.dim = dim
        self.M = M

    def forward(self, x):
        device = x.device
        half_dim = self.dim // 2
        emb = math.log(self.M) / half_dim
        emb = torch.exp(torch.arange(half_dim, device=device) * (-emb))
        emb = x[...,None] * emb[None,...]
        emb = torch.cat((emb.sin(), emb.cos()), dim=-1)
        return emb

class RNA_Model(nn.Module):
    def __init__(self, dim=192, depth=12, head_size=32, **kwargs):
        super().__init__()
        self.emb = nn.Embedding(4,dim)
        self.pos_enc = SinusoidalPosEmb(dim)
        self.transformer = nn.TransformerEncoder(
            nn.TransformerEncoderLayer(d_model=dim, nhead=dim//head_size, dim_feedforward=4*dim,
                dropout=0.1, activation=nn.GELU(), batch_first=True, norm_first=True), depth)
        self.proj_out = nn.Linear(dim,2)

    def forward(self, x0):
        mask = x0['mask']
        Lmax = mask.sum(-1).max()
        mask = mask[:, :Lmax]
```

```
        mask = mask[:,:Lmax]
        x = x0['seq'][:,:Lmax]

        pos = torch.arange(Lmax, device=x.device).unsqueeze(0)
        pos = self.pos_enc(pos)
        x = self.emb(x)
        x = x + pos

        x = self.transformer(x, src_key_padding_mask=~mask)
        x = self.proj_out(x)

        return x

def loss(pred,target):
    p = pred[target['mask'][:,:pred.shape[1]]]
    y = target['react'][target['mask']].clip(0,1)
    loss = F.l1_loss(p, y, reduction='none')
    loss = loss[~torch.isnan(loss)].mean()

    return loss

class MAE(Metric):
    def __init__(self):
        self.reset()

    def reset(self):
        self.x,self.y = [],[]

    def accumulate(self, learn):
        x = learn.pred[learn.y['mask'][:,:learn.pred.shape[1]]]
        y = learn.y['react'][learn.y['mask']].clip(0,1)
        self.x.append(x)
        self.y.append(y)

    @property
    def value(self):
        x,y = torch.cat(self.x,0),torch.cat(self.y,0)
        loss = F.l1_loss(x, y, reduction='none')
        loss = loss[~torch.isnan(loss)].mean()
        return loss

#df = pd.read_csv(os.path.join(PATH,r'C:\Users\VENKATESH\Downloads\Data_train.csv'))



seed_everything(SEED)
os.makedirs(OUT, exist_ok=True)
PATH="/content/drive/MyDrive/projects_RNA/train_data.parquet"
df=pd.read_parquet(PATH)

for fold in [0]: # running multiple folds at kaggle may cause OOM
    ds_train = RNA_Dataset(df, mode='train', fold=fold, nfolds=nfolds)
    ds_train_len = RNA_Dataset(df, mode='train', fold=fold,
                nfolds=nfolds, mask_only=True)
    sampler_train = torch.utils.data.RandomSampler(ds_train_len)
    len_sampler_train = LenMatchBatchSampler(sampler_train, batch_size=bs,
                drop_last=True)
    dl_train = DeviceDataLoader(torch.utils.data.DataLoader(ds_train,
                batch_sampler=len_sampler_train, num_workers=num_workers,
                persistent_workers=True), device)

    ds_val = RNA_Dataset(df, mode='eval', fold=fold, nfolds=nfolds)
    ds_val_len = RNA_Dataset(df, mode='eval', fold=fold, nfolds=nfolds,
               mask_only=True)
    sampler_val = torch.utils.data.SequentialSampler(ds_val_len)
    len_sampler_val = LenMatchBatchSampler(sampler_val, batch_size=bs,
               drop_last=False)
    dl_val= DeviceDataLoader(torch.utils.data.DataLoader(ds_val,
               batch_sampler=len_sampler_val, num_workers=num_workers), device)
    gc.collect()

    data = DataLoaders(dl_train,dl_val)
```

```
    model = RNA_Model()
    model = model.to(device)
    learn = Learner(data, model, loss_func=loss,cbs=[GradientClip(3.0)],
                metrics=[MAE()]).to_fp16()
    #fp16 doesn't help at P100 but gives x1.6-1.8 speedup at modern hardware

    learn.fit_one_cycle(32, lr_max=5e-4, wd=0.05, pct_start=0.02)
    torch.save(learn.model.state_dict(),os.path.join(OUT,f'{fname}_{fold}.pth'))
    gc.collect()

import gc
import os
import time
import pandas as pd
import numpy as np
import json
import torch
from fastai.data.load import DataLoader

from datasets import DatasetEightInfer, DatasetTenInfer
from models import ModelThirtyNine, ModelThirtyTwo
from seed_all import seed_everything

SUBMISSION_NUMBER = 27  # the setup is shown in this repository for 27 and 23 only
MODEL_EPOCH_NUMBER = 27  # 27 for submission number 27, and 44 for submission number 23
# (how many epochs the model was trained, starting from zero)

BATCH = 128
COL_A = 'reactivity_2A3_MaP'
COL_D = 'reactivity_DMS_MaP'


def batch_to_csv(output, ids, main_path_for_parquets):
    # received a batch of outputs (B, 459, 2) and ids (B, 4) as numpy arrays
    name_of_csv = ids[0][0]
    dfs = []
    for i in range(output.shape[0]):
        start_id = ids[i][0]
        end_id = ids[i][1]
        start_index = ids[i][2]
        num_reactivities = ids[i][3]
        # Extract relevant reactivities from output[i]
        reactivities_a = output[i, start_index: start_index + num_reactivities, 0]
        reactivities_d = output[i, start_index: start_index + num_reactivities, 1]
        # Create a DataFrame for the current datapoint
        datapoint_df = pd.DataFrame({
            'id': np.arange(start_id, end_id + 1),
            COL_D: reactivities_d,
            COL_A: reactivities_a
        })
        dfs.append(datapoint_df)
    small_df = pd.concat(dfs, ignore_index=True)
    # the df will be written into .parquet
    path = os.path.join(main_path_for_parquets, f"{name_of_csv}.parquet")
    small_df.to_parquet(path, index=False, engine='pyarrow')
    return


# before running, folder ../submissions/{SUBMISSION_NUMBER}/all needs to already exist
# for submission number 23, it runs for a very long time (eight plus hours) because bpps are not saved
# and need to be calculated in dataset
if __name__ == '__main__':
    seed_everything()
    with open('SETTINGS.json') as f:
        data = json.load(f)
    path_to_test_data = data["TEST_DATA"]
    model_dir = data["MODEL_DIR"]
    submission_dir = data["SUBMISSION_DIR"]
    model_string = f"{SUBMISSION_NUMBER}/models/model_{MODEL_EPOCH_NUMBER}.pth"
    path_to_model = os.path.join(model_dir, model_string)
    main_path_string = f"{SUBMISSION_NUMBER}/all/"
    main_path_for_parquets = os.path.join(submission_dir, main_path_string)
```

```
    main_path_for_parquets = os.path.join(submission_dir, main_path_string)

    df = pd.read_parquet(path_to_test_data, engine='pyarrow')
    # device
    device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')
    print(device)

    if SUBMISSION_NUMBER == 27:
        dataset_skeleton = DatasetEightInfer
        model_skeleton = ModelThirtyNine
        num_workers = 0
    elif SUBMISSION_NUMBER == 23:
        dataset_skeleton = DatasetTenInfer
        model_skeleton = ModelThirtyTwo
        num_workers = 40

    # dataset and dataloader
    dataset = dataset_skeleton(df=df)
    loader = DataLoader(dataset=dataset, batch_size=BATCH, pin_memory=False, shuffle=False, device=device,
                        num_workers=num_workers)  # num_workers is set to 40 for bpps (submission number 23)

    # model
    model = model_skeleton()
    # load the state dict
    model.load_state_dict(torch.load(path_to_model))
    model.eval()
    model.to(device)

    # Start timer
    start_time = time.time()
    with torch.no_grad():
        i = 0
        for data, ids in loader:
            i += 1
            out = model(data)
            batch_to_csv(out.detach().cpu().numpy(), ids.detach().cpu().numpy(), main_path_for_parquets)
            if i % 50 == 0:
                print(f"step {i}")
    # End timer
    end_time = time.time()
    # Calculate elapsed time
    elapsed_time = end_time - start_time
    print("Elapsed time: ", elapsed_time)
```

```
/usr/local/lib/python3.10/dist-packages/torch/nn/modules/transformer.py:286: UserWarning: enable_nested_tensor
  warnings.warn(f"enable_nested_tensor is True, but self.use_nested_tensor is False because {why_not_sparsity_
/usr/local/lib/python3.10/dist-packages/torch/amp/autocast_mode.py:250: UserWarning: User provided device_type
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torch/cuda/amp/grad_scaler.py:126: UserWarning: torch.cuda.amp.GradSca
  warnings.warn(
```

0.00% [0/32 00:00<?]

epoch	train_loss	valid_loss	mae	time

0.00% [0/531 00:00<?]

Double-click (or enter) to edit

```
# install packages
```

```
pip install datasets
```

```
Collecting datasets
  Downloading datasets-2.18.0-py3-none-any.whl (510 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 510.5/510.5 kB 6.0 MB/s eta 0:00:00
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages (from datasets) (3.13.1)
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.10/dist-packages (from datasets) (1.25.2)
Requirement already satisfied: pyarrow>=12.0.0 in /usr/local/lib/python3.10/dist-packages (from datasets) (14.
Requirement already satisfied: pyarrow-hotfix in /usr/local/lib/python3.10/dist-packages (from datasets) (0.6)
Collecting dill<0.3.9,>=0.3.0 (from datasets)
  Downloading dill-0.3.8-py3-none-any.whl (116 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 116.3/116.3 kB 13.1 MB/s eta 0:00:00
Requirement already satisfied: pandas in /usr/local/lib/python3.10/dist-packages (from datasets) (1.5.3)
Requirement already satisfied: requests>=2.19.0 in /usr/local/lib/python3.10/dist-packages (from datasets) (2.
Requirement already satisfied: tqdm>=4.62.1 in /usr/local/lib/python3.10/dist-packages (from datasets) (4.66.2
Requirement already satisfied: xxhash in /usr/local/lib/python3.10/dist-packages (from datasets) (3.4.1)
Collecting multiprocess (from datasets)
  Downloading multiprocess-0.70.16-py310-none-any.whl (134 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 134.8/134.8 kB 13.5 MB/s eta 0:00:00
Requirement already satisfied: fsspec[http]<=2024.2.0,>=2023.1.0 in /usr/local/lib/python3.10/dist-packages (f
Requirement already satisfied: aiohttp in /usr/local/lib/python3.10/dist-packages (from datasets) (3.9.3)
Requirement already satisfied: huggingface-hub>=0.19.4 in /usr/local/lib/python3.10/dist-packages (from datase
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-packages (from datasets) (23.2)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.10/dist-packages (from datasets) (6.0.1)
Requirement already satisfied: aiosignal>=1.1.2 in /usr/local/lib/python3.10/dist-packages (from aiohttp->data
Requirement already satisfied: attrs>=17.3.0 in /usr/local/lib/python3.10/dist-packages (from aiohttp->dataset
Requirement already satisfied: frozenlist>=1.1.1 in /usr/local/lib/python3.10/dist-packages (from aiohttp->dat
Requirement already satisfied: multidict<7.0,>=4.5 in /usr/local/lib/python3.10/dist-packages (from aiohttp->d
Requirement already satisfied: yarl<2.0,>=1.0 in /usr/local/lib/python3.10/dist-packages (from aiohttp->datase
Requirement already satisfied: async-timeout<5.0,>=4.0 in /usr/local/lib/python3.10/dist-packages (from aiohtt
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib/python3.10/dist-packages (from hug
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from reque
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests>=2.19.0-
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests>=2
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests>=2
Requirement already satisfied: python-dateutil>=2.8.1 in /usr/local/lib/python3.10/dist-packages (from pandas-
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas->datasets)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.8.
Installing collected packages: dill, multiprocess, datasets
Successfully installed datasets-2.18.0 dill-0.3.8 multiprocess-0.70.16
```

```
!pip install rotary_embedding_torch
```

```
Collecting rotary_embedding_torch
  Downloading rotary_embedding_torch-0.5.3-py3-none-any.whl (5.3 kB)
Collecting beartype (from rotary_embedding_torch)
  Downloading beartype-0.17.2-py3-none-any.whl (872 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 872.4/872.4 kB 18.8 MB/s eta 0:00:00
Collecting einops>=0.7 (from rotary_embedding_torch)
  Downloading einops-0.7.0-py3-none-any.whl (44 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 44.6/44.6 kB 6.3 MB/s eta 0:00:00
Requirement already satisfied: torch>=2.0 in /usr/local/lib/python3.10/dist-packages (from rotary_embedding_to
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages (from torch>=2.0->rotary_em
Requirement already satisfied: typing-extensions>=4.8.0 in /usr/local/lib/python3.10/dist-packages (from torch
Requirement already satisfied: sympy in /usr/local/lib/python3.10/dist-packages (from torch>=2.0->rotary_embed
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-packages (from torch>=2.0->rotary_em
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages (from torch>=2.0->rotary_embe
Requirement already satisfied: fsspec in /usr/local/lib/python3.10/dist-packages (from torch>=2.0->rotary_embe
Collecting nvidia-cuda-nvrtc-cu12==12.1.105 (from torch>=2.0->rotary_embedding_torch)
  Downloading nvidia_cuda_nvrtc_cu12-12.1.105-py3-none-manylinux1_x86_64.whl (23.7 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 23.7/23.7 MB 49.7 MB/s eta 0:00:00
Collecting nvidia-cuda-runtime-cu12==12.1.105 (from torch>=2.0->rotary_embedding_torch)
  Downloading nvidia_cuda_runtime_cu12-12.1.105-py3-none-manylinux1_x86_64.whl (823 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 823.6/823.6 kB 60.9 MB/s eta 0:00:00
Collecting nvidia-cuda-cupti-cu12==12.1.105 (from torch>=2.0->rotary_embedding_torch)
  Downloading nvidia_cuda_cupti_cu12-12.1.105-py3-none-manylinux1_x86_64.whl (14.1 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 14.1/14.1 MB 63.4 MB/s eta 0:00:00
Collecting nvidia-cudnn-cu12==8.9.2.26 (from torch>=2.0->rotary_embedding_torch)
  Downloading nvidia_cudnn_cu12-8.9.2.26-py3-none-manylinux1_x86_64.whl (731.7 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 731.7/731.7 MB 1.6 MB/s eta 0:00:00
Collecting nvidia-cublas-cu12==12.1.3.1 (from torch>=2.0->rotary_embedding_torch)
  Downloading nvidia_cublas_cu12-12.1.3.1-py3-none-manylinux1_x86_64.whl (410.6 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 410.6/410.6 MB 2.9 MB/s eta 0:00:00
Collecting nvidia-cufft-cu12==11.0.2.54 (from torch>=2.0->rotary_embedding_torch)
  Downloading nvidia_cufft_cu12-11.0.2.54-py3-none-manylinux1_x86_64.whl (121.6 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 121.6/121.6 MB 8.5 MB/s eta 0:00:00
```

```
Collecting nvidia-curand-cu12==10.3.2.106 (from torch>=2.0->rotary_embedding_torch)
  Downloading nvidia_curand_cu12-10.3.2.106-py3-none-manylinux1_x86_64.whl (56.5 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 56.5/56.5 MB 11.7 MB/s eta 0:00:00
Collecting nvidia-cusolver-cu12==11.4.5.107 (from torch>=2.0->rotary_embedding_torch)
  Downloading nvidia_cusolver_cu12-11.4.5.107-py3-none-manylinux1_x86_64.whl (124.2 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 124.2/124.2 MB 5.5 MB/s eta 0:00:00
Collecting nvidia-cusparse-cu12==12.1.0.106 (from torch>=2.0->rotary_embedding_torch)
  Downloading nvidia_cusparse_cu12-12.1.0.106-py3-none-manylinux1_x86_64.whl (196.0 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 196.0/196.0 MB 2.4 MB/s eta 0:00:00
Collecting nvidia-nccl-cu12==2.19.3 (from torch>=2.0->rotary_embedding_torch)
  Downloading nvidia_nccl_cu12-2.19.3-py3-none-manylinux1_x86_64.whl (166.0 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 166.0/166.0 MB 6.3 MB/s eta 0:00:00
Collecting nvidia-nvtx-cu12==12.1.105 (from torch>=2.0->rotary_embedding_torch)
  Downloading nvidia_nvtx_cu12-12.1.105-py3-none-manylinux1_x86_64.whl (99 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 99.1/99.1 kB 11.3 MB/s eta 0:00:00
Requirement already satisfied: triton==2.2.0 in /usr/local/lib/python3.10/dist-packages (from torch>=2.0->rota
Collecting nvidia-nvjitlink-cu12 (from nvidia-cusolver-cu12==11.4.5.107->torch>=2.0->rotary_embedding_torch)
  Downloading nvidia_nvjitlink_cu12-12.4.99-py3-none-manylinux2014_x86_64.whl (21.1 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 21.1/21.1 MB 69.1 MB/s eta 0:00:00
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-packages (from jinja2->torch>
Requirement already satisfied: mpmath>=0.19 in /usr/local/lib/python3.10/dist-packages (from sympy->torch>=2.0
Installing collected packages: nvidia-nvtx-cu12, nvidia-nvjitlink-cu12, nvidia-nccl-cu12, nvidia-curand-cu12,
Successfully installed beartype-0.17.2 einops-0.7.0 nvidia-cublas-cu12-12.1.3.1 nvidia-cuda-cupti-cu12-12.1.10
```

```python
import torch
import torch.nn as nn
import math
from rotary_embedding_torch import RotaryEmbedding

LEN = 457
LEN_EOS = 459
LEN_FOR_GENERALIZATION = 722

############################################################
# the code for building transformer (building blocks) is from
# https://towardsdatascience.com/build-your-own-transformer-from-scratch-using-pytorch-84c850470dcb

# the way how sinusoidal embedding is calculated is from https://www.kaggle.com/code/iafoss/rna-starter-0-186
class PosEnc(nn.Module):
    """
    sinusoidal embeddings
    """
    def __init__(self, dim=192, M=10000, num_tokens=LEN_EOS):
        super().__init__()
        positions = torch.arange(num_tokens).unsqueeze(0)
        half_dim = dim // 2
        emb = math.log(M) / half_dim
        emb = torch.exp(torch.arange(half_dim) * (-emb))
        emb = positions[..., None] * emb[None, ...]
        emb = torch.cat((emb.sin(), emb.cos()), dim=-1)
        self.pos = emb

    def forward(self, x):
        device = x.device
        pos = self.pos.to(device)
        res = x + pos
        return res


# https://towardsdatascience.com/build-your-own-transformer-from-scratch-using-pytorch-84c850470dcb
class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        super(MultiHeadAttention, self).__init__()
        assert d_model % num_heads == 0, "d_model must be divisible by num_heads"

        self.d_model = d_model
        self.num_heads = num_heads
        self.d_k = d_model // num_heads

        self.W_q = nn.Linear(d_model, d_model)
        self.W_k = nn.Linear(d_model, d_model)
```

```
        self.W_v = nn.Linear(d_model, d_model)
        self.W_o = nn.Linear(d_model, d_model)

    def scaled_dot_product_attention(self, Q, K, V, mask=None):
        attn_scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(self.d_k)
        if mask is not None:
            _MASKING_VALUE = -1e+30 if attn_scores.dtype == torch.float32 else -1e+4
            attn_scores = attn_scores.masked_fill(mask == 0, _MASKING_VALUE)
        attn_probs = torch.softmax(attn_scores, dim=-1)
        output = torch.matmul(attn_probs, V)
        return output

    def split_heads(self, x):
        batch_size, seq_length, d_model = x.size()
        return x.view(batch_size, seq_length, self.num_heads, self.d_k).transpose(1, 2)

    def combine_heads(self, x):
        batch_size, _, seq_length, d_k = x.size()
        return x.transpose(1, 2).contiguous().view(batch_size, seq_length, self.d_model)

    def forward(self, Q, K, V, mask=None):
        Q = self.split_heads(self.W_q(Q))
        K = self.split_heads(self.W_k(K))
        V = self.split_heads(self.W_v(V))

        attn_output = self.scaled_dot_product_attention(Q, K, V, mask)
        output = self.W_o(self.combine_heads(attn_output))
        return output


class AttentionRotary(nn.Module):
    def __init__(self, d_model, num_heads, rotary_emb):
        super(AttentionRotary, self).__init__()
        assert d_model % num_heads == 0, "d_model must be divisible by num_heads"

        self.d_model = d_model
        self.num_heads = num_heads
        self.d_k = d_model // num_heads
        self.rotary_emb = rotary_emb

        self.W_q = nn.Linear(d_model, d_model)
        self.W_k = nn.Linear(d_model, d_model)
        self.W_v = nn.Linear(d_model, d_model)
        self.W_o = nn.Linear(d_model, d_model)

    def scaled_dot_product_attention(self, Q, K, V, mask=None):
        attn_scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(self.d_k)
        if mask is not None:
            _MASKING_VALUE = -1e+30 if attn_scores.dtype == torch.float32 else -1e+4
            attn_scores = attn_scores.masked_fill(mask == 0, _MASKING_VALUE)
        attn_probs = torch.softmax(attn_scores, dim=-1)
        output = torch.matmul(attn_probs, V)
        return output

    def split_heads(self, x):
        batch_size, seq_length, d_model = x.size()
        return x.view(batch_size, seq_length, self.num_heads, self.d_k).transpose(1, 2)

    def combine_heads(self, x):
        batch_size, _, seq_length, d_k = x.size()
        return x.transpose(1, 2).contiguous().view(batch_size, seq_length, self.d_model)

    def forward(self, Q, K, V, mask=None):
        Q = self.split_heads(self.W_q(Q))
        Q = self.rotary_emb.rotate_queries_or_keys(Q)
        K = self.split_heads(self.W_k(K))
        K = self.rotary_emb.rotate_queries_or_keys(K)
        V = self.split_heads(self.W_v(V))

        attn_output = self.scaled_dot_product_attention(Q, K, V, mask)
        output = self.W_o(self.combine_heads(attn_output))
```

```
        return output


class CustomAttentionBPP(nn.Module):
    def __init__(self, d_model, num_heads=1):
        super(CustomAttentionBPP, self).__init__()
        assert d_model % num_heads == 0, "d_model must be divisible by num_heads"

        self.d_model = d_model
        self.num_heads = num_heads
        self.d_k = d_model // num_heads

        self.W_v = nn.Linear(d_model, d_model)
        self.W_o = nn.Linear(d_model, d_model)

    def scaled_dot_product_attention(self, bpp, V, mask=None):
        attn_scores = bpp.unsqueeze(1)
        _MASKING_VALUE = -1e+30 if attn_scores.dtype == torch.float32 else -1e+4
        attn_scores = attn_scores.masked_fill(attn_scores == 0, _MASKING_VALUE)
        if mask is not None:
            attn_scores = attn_scores.masked_fill(mask == 0, _MASKING_VALUE)
        attn_probs = torch.softmax(attn_scores, dim=-1)
        output = torch.matmul(attn_probs, V)
        return output

    def split_heads(self, x):
        batch_size, seq_length, d_model = x.size()
        return x.view(batch_size, seq_length, self.num_heads, self.d_k).transpose(1, 2)

    def combine_heads(self, x):
        batch_size, _, seq_length, d_k = x.size()
        return x.transpose(1, 2).contiguous().view(batch_size, seq_length, self.d_model)

    def forward(self, bpp, V, mask=None):
        V = self.split_heads(self.W_v(V))

        attn_output = self.scaled_dot_product_attention(bpp, V, mask)
        output = self.W_o(self.combine_heads(attn_output))
        return output


# https://towardsdatascience.com/build-your-own-transformer-from-scratch-using-pytorch-84c850470dcb
# gelu is used instead of relu
class PositionWiseFeedForward(nn.Module):
    def __init__(self, d_model, d_ff):
        super(PositionWiseFeedForward, self).__init__()
        self.fc1 = nn.Linear(d_model, d_ff)
        self.fc2 = nn.Linear(d_ff, d_model)
        self.gelu = nn.GELU()

    def forward(self, x):
        return self.fc2(self.gelu(self.fc1(x)))


# https://towardsdatascience.com/build-your-own-transformer-from-scratch-using-pytorch-84c850470dcb
# with minor modifications
class EncoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, d_ff, dropout):
        super(EncoderLayer, self).__init__()
        self.self_attn = MultiHeadAttention(d_model, num_heads)
        self.feed_forward = PositionWiseFeedForward(d_model, d_ff)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask):
        attn_output = self.self_attn(x, x, x, mask)
        x = self.norm1(x + self.dropout(attn_output))
        ff_output = self.feed_forward(x)
        x = self.norm2(x + self.dropout(ff_output))
        return x
```

```
190
191
192 class EncoderLayerRotary(nn.Module):
193     def __init__(self, d_model, num_heads, d_ff, dropout, rotary_emb):
194         super(EncoderLayerRotary, self).__init__()
195         self.self_attn = AttentionRotary(d_model, num_heads, rotary_emb)
196         self.feed_forward = PositionWiseFeedForward(d_model, d_ff)
197         self.norm1 = nn.LayerNorm(d_model)
198         self.norm2 = nn.LayerNorm(d_model)
199         self.dropout = nn.Dropout(dropout)
200
201     def forward(self, x, mask):
202         attn_output = self.self_attn(x, x, x, mask)
203         x = self.norm1(x + self.dropout(attn_output))
204         ff_output = self.feed_forward(x)
205         x = self.norm2(x + self.dropout(ff_output))
206         return x
207
208
209 # https://towardsdatascience.com/build-your-own-transformer-from-scratch-using-pytorch-84c850470dcb
210 class DecoderLayer(nn.Module):
211     def __init__(self, d_model, num_heads, d_ff, dropout):
212         super(DecoderLayer, self).__init__()
213         self.self_attn = MultiHeadAttention(d_model, num_heads)
214         self.cross_attn = MultiHeadAttention(d_model, num_heads)
215         self.feed_forward = PositionWiseFeedForward(d_model, d_ff)
216         self.norm1 = nn.LayerNorm(d_model)
217         self.norm2 = nn.LayerNorm(d_model)
218         self.norm3 = nn.LayerNorm(d_model)
219         self.dropout = nn.Dropout(dropout)
220
221     def forward(self, x, enc_output, src_mask, tgt_mask):
222         attn_output = self.self_attn(x, x, x, tgt_mask)
223         x = self.norm1(x + self.dropout(attn_output))
224         attn_output = self.cross_attn(x, enc_output, enc_output, src_mask)
225         x = self.norm2(x + self.dropout(attn_output))
226         ff_output = self.feed_forward(x)
227         x = self.norm3(x + self.dropout(ff_output))
228         return x
229
230
231 class DecoderLayerRotary(nn.Module):
232     def __init__(self, d_model, num_heads, d_ff, dropout, rotary_emb):
233         super(DecoderLayerRotary, self).__init__()
234         self.self_attn = AttentionRotary(d_model, num_heads, rotary_emb)
235         self.cross_attn = AttentionRotary(d_model, num_heads, rotary_emb)
236         self.feed_forward = PositionWiseFeedForward(d_model, d_ff)
237         self.norm1 = nn.LayerNorm(d_model)
238         self.norm2 = nn.LayerNorm(d_model)
239         self.norm3 = nn.LayerNorm(d_model)
240         self.dropout = nn.Dropout(dropout)
241
242     def forward(self, x, enc_output, src_mask, tgt_mask):
243         attn_output = self.self_attn(x, x, x, tgt_mask)
244         x = self.norm1(x + self.dropout(attn_output))
245         attn_output = self.cross_attn(x, enc_output, enc_output, src_mask)
246         x = self.norm2(x + self.dropout(attn_output))
247         ff_output = self.feed_forward(x)
248         x = self.norm3(x + self.dropout(ff_output))
249         return x
250
251
252 # similar to DecoderLayer, but as cross_attn, it uses CustomAttentionBPP
253 class DecoderLayerTwo(nn.Module):
254     def __init__(self, d_model, num_heads, d_ff, dropout):
255         super(DecoderLayerTwo, self).__init__()
256         self.self_attn = MultiHeadAttention(d_model, num_heads)
257         self.cross_attn = CustomAttentionBPP(d_model)
258         self.feed_forward = PositionWiseFeedForward(d_model, d_ff)
259         self.norm1 = nn.LayerNorm(d_model)
260         self.norm2 = nn.LayerNorm(d_model)
```

```
        self.norm3 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, bpp, mask):
        attn_output = self.self_attn(x, x, x, mask)
        x = self.norm1(x + self.dropout(attn_output))
        attn_output = self.cross_attn(bpp=bpp, V=x, mask=mask)
        x = self.norm2(x + self.dropout(attn_output))
        ff_output = self.feed_forward(x)
        x = self.norm3(x + self.dropout(ff_output))
        return x

##########################################################
# models:


# first it is decoder layer to use bpp (with sinusoidal pos embeds), then uses rotary embeddings
# tgt, info1: seq_inds; info2: bpp; src or info3: struct_inds
class ModelThirtyTwo(nn.Module):
    def __init__(self, tgt_vocab=7, src_vocab=6, d_model=192, num_heads=6, num_layers=8,
                 d_ff=(192*4), dropout=0.1, num_tokens=LEN_EOS):
        super(ModelThirtyTwo, self).__init__()
        self.tgt_embedding = nn.Embedding(tgt_vocab, d_model)
        self.src_embedding = nn.Embedding(src_vocab, d_model)
        self.positional_enc = PosEnc(dim=d_model, num_tokens=num_tokens)
        self.rotary = RotaryEmbedding(dim=d_model//num_heads)
        self.decoder_one = DecoderLayerTwo(d_model, num_heads, d_ff, dropout)
        self.decoder = DecoderLayerRotary(d_model, num_heads, d_ff, dropout, self.rotary)
        self.encoder_layers = nn.ModuleList([EncoderLayerRotary(d_model, num_heads, d_ff, dropout, self.rotar
        self.fc = nn.Linear(d_model, 2)

    def forward(self, data):
        tgt = data['info1']
        bpp = data['info2']
        src = data['info3']
        mask = data['mask']

        mask = mask.unsqueeze(1).unsqueeze(2)
        src = self.src_embedding(src)
        tgt = self.positional_enc(self.tgt_embedding(tgt))

        output = self.decoder_one(x=tgt, bpp=bpp, mask=mask)

        output = self.decoder(x=output, enc_output=src, src_mask=mask, tgt_mask=mask)
        for enc_layer in self.encoder_layers:
            output = enc_layer(output, mask)

        output = self.fc(output)
        return output


class ModelThirtyNine(nn.Module):
    def __init__(self, tgt_vocab=7, src_vocab=6, d_model=384, num_heads=6, num_layers=8, d_ff=384, dropout=0
        super(ModelThirtyNine, self).__init__()
        self.tgt_embedding = nn.Embedding(tgt_vocab, d_model)
        self.src_embedding = nn.Embedding(src_vocab, d_model)
        self.positional_enc = RotaryEmbedding(dim=d_model//num_heads)
        self.decoder = DecoderLayerRotary(d_model, num_heads, d_ff, dropout, self.positional_enc)
        self.encoder_layers = nn.ModuleList([EncoderLayerRotary(d_model, num_heads, d_ff, dropout, self.posit
        self.fc = nn.Linear(d_model, 2)

    def forward(self, data):
        tgt = data['info1']
        src = data['info2']
        mask = data['mask']

        mask = mask.unsqueeze(1).unsqueeze(2)
        tgt = self.tgt_embedding(tgt)
        src = self.src_embedding(src)

        output = self.decoder(x=tgt, enc_output=src, src_mask=mask, tgt_mask=mask)
```

```
        for enc_layer in self.encoder_layers:
            output = enc_layer(output, mask)

        output = self.fc(output)
        return output
```