Open in Colab

```
In [ ]: !nvidia-smi
```

```
Mon Dec 20 20:29:36 2021
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 495.44       Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla P100-PCIE...  Off  | 00000000:00:04.0 Off |                    0 |
| N/A   39C    P0    27W / 250W |      0MiB / 16280MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

```
In [ ]: !pip install torchvision==0.8.2  -q -q -q
!pip install torchtext==0.8.1    -q -q -q
!pip install torchaudio==0.7.2   -q -q -q
!pip install torch==1.7.1        -q -q -q
!pip install tqdm==4.49.0        -q -q -q
!pip install transformers==4.2.2 -q -q -q
!pip install tensorflow          -q -q -q
!pip install tensorboard         -q -q -q
!pip install tensorboardX        -q -q -q
!pip install --upgrade virtualenv -q -q -q
```

```
     |████████████████████████████████| 776.8 MB 17 kB/s
ERROR: pip's dependency resolver does not currently take into account all the packages that are installed. This
 behaviour is the source of the following dependency conflicts.
torchtext 0.11.0 requires torch==1.10.0, but you have torch 1.7.1 which is incompatible.
torchaudio 0.10.0+cu111 requires torch==1.10.0, but you have torch 1.7.1 which is incompatible.
     |████████████████████████████████| 69 kB 3.5 MB/s
     |████████████████████████████████| 1.8 MB 5.0 MB/s
     |████████████████████████████████| 895 kB 41.8 MB/s
     |████████████████████████████████| 2.9 MB 50.1 MB/s
     |████████████████████████████████| 124 kB 5.2 MB/s
     |████████████████████████████████| 5.6 MB 5.0 MB/s
     |████████████████████████████████| 461 kB 51.5 MB/s
```

```
In [ ]: from google.colab import drive
import shutil
drive.mount('/content/gdrive')
#shutil.copy('/content/gdrive/MyDrive/MTUNLPA2/saved_pickle.tar.gz', '/tmp/saved_pickle.tar.gz')
```

```
Mounted at /content/gdrive
```

```
In [ ]: #!tar -xvzf /tmp/saved_pickle.tar.gz
#!rm -f /tmp/saved_pickle.tar.gz
#!du -sh /content/gdrive/MyDrive/NLP/NLP-Save-Train/train.tar.gz /tmp
```

```
In [ ]: %%writefile copy_train.sh
#!/bin/bash
cp /content/gdrive/MyDrive/NLP/NLP-Save-Train/train.tar.gz /tmp
cd /tmp
gunzip train.tar.gz
tar -xf train.tar
ls /tmp/train
```

```
Writing copy_train.sh
```

```
In [ ]: !/bin/bash copy_train.sh
```

```
attention_mask  id         offset_mapping              start_positions
end_positions   input_ids  overflow_to_sample_mapping
```

```
In [ ]: !cat /tmp/train/input_ids/config.json
```

```
{"length": 242304}
```

```
In [ ]: #!du -sh saved_pickle
#!pwd
#!mv /tmp/train /tmp/train.save
```

```
In [ ]: !git clone https://github.com/bhattacharjee/mtu-nlp-assignment.git
```

```
Cloning into 'mtu-nlp-assignment'...
remote: Enumerating objects: 1027, done.
remote: Counting objects: 100% (1027/1027), done.
remote: Compressing objects: 100% (892/892), done.
remote: Total 1027 (delta 508), reused 383 (delta 131), pack-reused 0
Receiving objects: 100% (1027/1027), 96.10 MiB | 45.41 MiB/s, done.
Resolving deltas: 100% (508/508), done.
```

args.py

```
In [ ]: %%writefile /tmp/args.py
import argparse

def get_train_test_args():
    parser = argparse.ArgumentParser()
    parser.add_argument('--batch-size', type=int, default=32)
    parser.add_argument('--num-epochs', type=int, default=3)
    parser.add_argument('--lr', type=float, default=3e-5)
    parser.add_argument('--num-visuals', type=int, default=10)
    parser.add_argument('--seed', type=int, default=42)
    parser.add_argument('--save-dir', type=str, default='save/')
    parser.add_argument('--train', action='store_true')
    parser.add_argument('--eval', action='store_true')
    parser.add_argument('--train-datasets', type=str, default='squad,nat_questions,newsqa')
    parser.add_argument('--run-name', type=str, default='multitask_distilbert')
    parser.add_argument('--recompute-features', action='store_true')
    parser.add_argument('--train-dir', type=str, default='datasets/indomain_train')
    parser.add_argument('--val-dir', type=str, default='datasets/indomain_val')
    parser.add_argument('--eval-dir', type=str, default='datasets/oodomain_test')
    parser.add_argument('--eval-datasets', type=str, default='race,relation_extraction,duorc')
    parser.add_argument('--do-train', action='store_true')
    parser.add_argument('--do-eval', action='store_true')
    parser.add_argument('--sub-file', type=str, default='')
    parser.add_argument('--visualize-predictions', action='store_true')
    parser.add_argument('--eval-every', type=int, default=2500)
    args = parser.parse_args()
    return args
```

```
Overwriting /tmp/args.py
```

chunkdata.py

```
In [ ]: %%writefile /tmp/ChunkData.py
#!/usr/bin/env python3

# Copyright 2021 Rajbir Bhattacharjee
#
# Permission is hereby granted, free of charge, to any person obtaining a copy
# of this software and associated documentation files (the "Software"), to deal
# in the Software without restriction, including without limitation the rights
# to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
# copies of the Software, and to permit persons to whom the Software is
# furnished to do so, subject to the following conditions:
#
# The above copyright notice and this permission notice shall be included in
# all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
# IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
# FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
# AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
# LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
# OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
# SOFTWARE.

import os
import shutil
import pathlib
import pickle
import json

import threading
import glob

class ChunkDataCommon():
    def __init__(self, directory:str, chunk_size=16):
        self.directory = directory
        self.chunk_size = chunk_size
        self.length = -1
        self.lock = None
        self.current_slice = None
        self.current_slice_start = -1

    def get_pickle_file_name(self, st, en):
        return f"{self.directory}/array_chunk-{st:08d}-{en:08d}.pickle"

    def get_conf_file_name(self):
        with self.lock:
            return f"{self.directory}/config.json"

    def get_slice_start_for_index(self, ind):
        return (ind // self.chunk_size) * self.chunk_size

    def load_slice(self, ind_start):
        with self.lock:
            ind_end = ind_start + self.chunk_size - 1
            filename =self.get_pickle_file_name(ind_start, ind_end)
            with open(filename, "rb") as f:
                self.current_slice = pickle.load(f)
                self.current_slice_start = ind_start
                return
            self.current_slice = None
            self.current_slice_start = -1
            raise IndexError

    def __len__(self):
        with self.lock:
            return self.length if self.length > 0 else 0


class ChunkDataReader(ChunkDataCommon):
    def __init__(self, d, chunk_size=16, lock=None):
        if isinstance(d, str):
            directory = d
        elif isinstance(d, ChunkDataCommon):
            directory = d.directory
            chunk_size = d.chunk_size

        super(self.__class__, self).__init__(directory, chunk_size=chunk_size)

        self.lock = threading.RLock() if lock is None else lock
        self.length = -1

        self.current_slice = None
        self.current_slice_start = -1
        self.chunk_size = chunk_size
        self.directory = directory

        with open(self.get_conf_file_name(), "r") as f:
            conf = json.load(f)
            length = conf['length']
            self.length = length if length > 0 else -1

    def __getitem__(self, ind):
        with self.lock:
            if ind >= self.length:
                raise IndexError
            if (ind < 0):
                ind = self.length + ind
            with self.lock:
                ind_start = self.get_slice_start_for_index(ind)
                if ind_start != self.current_slice_start:
                    self.load_slice(ind_start)
                ind = ind - self.current_slice_start
                return self.current_slice[ind]



class ChunkDataWriter(ChunkDataCommon):
    """Uses pickle, but splits into multiple files.
    behaves like an array and supports the append() method,
    and finalize method()"""
    def __init__(self, directory:str, chunk_size=16):
        super(self.__class__, self).__init__(directory, chunk_size)
        self.lock = threading.RLock()

        # The reader object for __getitem__
        self.reader = None

        # the current slice of the array
        self.current_slice = list()

        # Length of all items
        self.length = 0

        # Where does the current slice start from
        self.current_slice_start = -1

        self.__setitem__ = None

        if not os.path.exists(self.directory):
            pathlib.Path(self.directory).mkdir(parents=True, exist_ok=False)

        if os.path.exists(self.get_conf_file_name()):
            self.load_existing()
        else:
            self.write_conf()

    def load_existing(self):
        with open(self.get_conf_file_name(), "r") as f:
            d = json.load(f)
            self.length = d['length']
            self.length = self.length if self.length > 0 else 0
            if self.length > 0:
                ind_start = self.get_slice_start_for_index(self.length - 1)
                self.load_slice(ind_start)
                self.current_slice_start = ind_start

    def finalize(self):
        with self.lock:
            self.write_conf()

    def get_current_slice_max(self):
        # Return the index of the last element that can
        # be stored in the current slice
        with self.lock:
            if self.current_slice_start == -1:
                return -1
            else:
                return self.current_slice_start + self.chunk_size - 1

    def get_current_slice_end(self):
        # Return the end of the current slice
        with self.lock:
            if self.current_slice_start == -1:
                return -1
            else:
                return self.current_slice_start + len(self.current_slice)

    def get_current_slice_size(self):
        # return the size of the current slice
        with self.lock:
            return len(self.current_slice)

    def get_current_slice_offsets(self):
        with self.lock:
            return self.current_slice_start, \
                self.get_current_slice_end(), \
                self.get_current_slice_max()

    def get_current_file_name(self):
        with self.lock:
            if -1 == self.current_slice_start:
                return None
            st = self.current_slice_start
            en = st + self.chunk_size - 1
            return self.get_pickle_file_name(st, en)

    def write_chunk(self):
        with self.lock:
            with open(self.get_current_file_name(), "wb") as f:
                pickle.dump(self.current_slice, f, pickle.HIGHEST_PROTOCOL)


    def write_conf(self):
        with self.lock:
            thedict = {}
            thedict["length"] = self.length
            #print("Exists?", os.path.exists(self.directory))
            with open(self.get_conf_file_name(), "w") as f:
                json.dump(thedict, f)

    def append(self, x):
        with self.lock:
            if 0 == len(self.current_slice) % self.chunk_size:
                self.current_slice = list()
                if self.current_slice_start != -1:
                    self.current_slice_start += self.chunk_size
            if self.current_slice_start == -1:
                self.current_slice_start = 0
            self.length += 1
            self.current_slice.append(x)
            self.write_chunk()
            self.write_conf()
        pass

    def __del__(self):
        with self.lock:
            try:
                # Directory may have been removed by now
                # because del might be called quite late
                if os.path.exists(self.directory):
                    self.write_conf()
            except Exception as e:
                traceback.print_tb(e.__traceback__, file=sys.stderr)

    def __getitem__(self, ind):
        with self.lock:
            try:
                # print(self.length, self.directory)
                if self.reader is None or self.reader.length <= ind:
                    self.reader = ChunkDataReader(self, lock=self.lock)
            except Exception as e:
                # print(self.length, self.directory)
                traceback.print_tb(e.__traceback__, file=sys.stderr)
                raise IndexError
            return self.reader[ind]
```

```
Overwriting /tmp/ChunkData.py
```

util.py

```
In [ ]: %%writefile /tmp/util.py

import json
import random
import os
import logging
import pickle
import string
import re
from pathlib import Path
from collections import Counter, OrderedDict, defaultdict as ddict
import torch
import numpy as np
from tqdm import tqdm
from torch.utils.data import Dataset
from ChunkData import *

def set_seed(seed):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)

def load_pickle(path):
    with open(path, 'rb') as f:
        obj = pickle.load(f)
    return obj

def save_pickle(obj, path):
    """
    with open(path, 'wb') as f:
        pickle.dump(obj, f)
    """
    return

def visualize(tbx, pred_dict, gold_dict, step, split, num_visuals):
    """Visualize text examples to TensorBoard.

    Args:
        tbx (tensorboardX.SummaryWriter): Summary writer.
        pred_dict (dict): dict of predictions of the form id -> pred.
        step (int): Number of examples seen so far during training.
        split (str): Name of data split being visualized.
        num_visuals (int): Number of visuals to select at random from preds.
    """
    if num_visuals <= 0:
        return
    if num_visuals > len(pred_dict):
        num_visuals = len(pred_dict)
    id2index = {curr_id : idx for idx, curr_id in enumerate(gold_dict['id'])}
    visual_ids = np.random.choice(list(pred_dict), size=num_visuals, replace=False)
    for i, id_ in enumerate(visual_ids):
        pred = pred_dict[id_] or 'N/A'
        idx_gold_dict = id2index[id_]
        question = gold_dict['question'][idx_gold_dict]
        context = gold_dict['context'][idx_gold_dict]
        answers = gold_dict['answer'][idx_gold_dict]
        gold = answers['text'][0] if answers else 'N/A'
        tbl_fmt = (f'- **Question:** {question}\n'
                   + f'- **Context:** {context}\n'
                   + f'- **Answer:** {gold}\n'
                   + f'- **Prediction:** {pred}')
        tbx.add_text(tag=f'{split}/{i+1}_of_{num_visuals}',
                     text_string=tbl_fmt,
                     global_step=step)


def get_save_dir(base_dir, name, id_max=100):
    for uid in range(1, id_max):
        save_dir = os.path.join(base_dir, f'{name}-{uid:02d}')
        if not os.path.exists(save_dir):
            os.makedirs(save_dir)
            return save_dir

    raise RuntimeError('Too many save directories created with the same name. \
                       Delete old save directories or use another name.')


def filter_encodings(encodings):
    filter_idx = [idx for idx, val in enumerate(encodings['end_positions'])
                 if not val]
    filter_idx = set(filter_idx)
    encodings_filtered = {key : [] for key in encodings}
    sz = len(encodings['input_ids'])
    for idx in range(sz):
        if idx not in filter_idx:
            for key in encodings:
                encodings_filtered[key].append(encodings[key][idx])
    return encodings_filtered

def merge(encodings, new_encoding):
    if not encodings:
        return new_encoding
    else:
        for key in new_encoding:
            encodings[key] += new_encoding[key]
        return encodings

def get_logger(log_dir, name):
    """Get a `logging.Logger` instance that prints to the console
    and an auxiliary file.

    Args:
        log_dir (str): Directory in which to create the log file.
        name (str): Name to identify the logs.

    Returns:
        logger (logging.Logger): Logger instance for logging events.
    """
    class StreamHandlerWithTQDM(logging.Handler):
        """Let `logging` print without breaking `tqdm` progress bars.

        See Also:
            > https://stackoverflow.com/questions/38543506
        """
        def emit(self, record):
            try:
                msg = self.format(record)
                tqdm.write(msg)
                self.flush()
            except (KeyboardInterrupt, SystemExit):
                raise
            except:
                self.handleError(record)

    # Create logger
    logger = logging.getLogger(name)
    logger.setLevel(logging.DEBUG)

    # Log everything (i.e., DEBUG level and above) to a file
    log_path = os.path.join(log_dir, f'{name}.txt')
    file_handler = logging.FileHandler(log_path)
    file_handler.setLevel(logging.DEBUG)

    # Log everything except DEBUG level (i.e., INFO level and above) to console
    console_handler = StreamHandlerWithTQDM()
    console_handler.setLevel(logging.INFO)

    # Create format for the logs
    file_formatter = logging.Formatter('[%(asctime)s] %(message)s',
                                       datefmt='%m.%d.%y %H:%M:%S')
    file_handler.setFormatter(file_formatter)
    console_formatter = logging.Formatter('[%(asctime)s] %(message)s',
                                          datefmt='%m.%d.%y %H:%M:%S')
    console_handler.setFormatter(console_formatter)

    # add the handlers to the logger
    logger.addHandler(file_handler)
    logger.addHandler(console_handler)

    return logger

class AverageMeter:
    """Keep track of average values over time.

    Adapted from:
        > https://github.com/pytorch/examples/blob/master/imagenet/main.py
    """
    def __init__(self):
        self.avg = 0
        self.sum = 0
        self.count = 0

    def reset(self):
        """Reset meter."""
        self.__init__()

    def update(self, val, num_samples=1):
        """Update meter with new value `val`, the average of `num` samples.

        Args:
            val (float): Average value to update the meter with.
            num_samples (int): Number of samples that were averaged to
                produce `val`.
        """
        self.count += num_samples
        self.sum += val * num_samples
        self.avg = self.sum / self.count

class QADataset(Dataset):
    def __init__(self, encodings, train=True):
        self.encodings = encodings
        self.keys = ['input_ids', 'attention_mask']
        if train:
            self.keys += ['start_positions', 'end_positions']
        assert(all(key in self.encodings for key in self.keys))

    def __getitem__(self, idx):
        return {key : torch.tensor(self.encodings[key][idx]) for key in self.keys}

    def __len__(self):
        return len(self.encodings['input_ids'])

def read_squad(path):
    path = Path(path)
    with open(path, 'rb') as f:
        squad_dict = json.load(f)
    data_dict = {'question': [], 'context': [], 'id': [], 'answer': []}
    for group in squad_dict['data']:
        for passage in group['paragraphs']:
            context = passage['context']
            for qa in passage['qas']:
                question = qa['question']
                if len(qa['answers']) == 0:
                    data_dict['question'].append(question)
                    data_dict['context'].append(context)
                    data_dict['id'].append(qa['id'])
                else:
                    for answer in  qa['answers']:
                        data_dict['question'].append(question)
                        data_dict['context'].append(context)
                        data_dict['id'].append(qa['id'])
                        data_dict['answer'].append(answer)
    id_map = ddict(list)
    for idx, qid in enumerate(data_dict['id']):
        id_map[qid].append(idx)

    data_dict_collapsed = {'question': [], 'context': [], 'id': []}
    if data_dict['answer']:
        data_dict_collapsed['answer'] = []
    for qid in id_map:
        ex_ids = id_map[qid]
        data_dict_collapsed['question'].append(data_dict['question'][ex_ids[0]])
        data_dict_collapsed['context'].append(data_dict['context'][ex_ids[0]])
        data_dict_collapsed['id'].append(qid)
        if data_dict['answer']:
            all_answers = [data_dict['answer'][idx] for idx in ex_ids]
            data_dict_collapsed['answer'].append({'answer_start': [answer['answer_start'] for answer in all_ans
                                                  'text': [answer['text'] for answer in all_answers]})
    return data_dict_collapsed

def add_token_positions(encodings, answers, tokenizer):
    start_positions = []
    end_positions = []
    for i in range(len(answers)):
        start_positions.append(encodings.char_to_token(i, answers[i]['answer_start']))
        end_positions.append(encodings.char_to_token(i, answers[i]['answer_end']))

        # if start position is None, the answer passage has been truncated
        if start_positions[-1] is None:
            start_positions[-1] = tokenizer.model_max_length

        # if end position is None, the 'char_to_token' function points to the space before the correct token -
        if end_positions[-1] is None:
            end_positions[-1] = encodings.char_to_token(i, answers[i]['answer_end'] + 1)
    encodings.update({'start_positions': start_positions, 'end_positions': end_positions})


def add_end_idx(answers, contexts):
    for answer, context in zip(answers, contexts):
        gold_text = answer['text']
        start_idx = answer['answer_start']
        end_idx = start_idx + len(gold_text)

        # sometimes squad answers are off by a character or two - fix this
        if context[start_idx:end_idx] == gold_text:
            answer['answer_end'] = end_idx
        elif context[start_idx-1:end_idx-1] == gold_text:
            answer['answer_start'] = start_idx - 1
            answer['answer_end'] = end_idx - 1     # When the gold label is off by one character
        elif context[start_idx-2:end_idx-2] == gold_text:
            answer['answer_start'] = start_idx - 2
            answer['answer_end'] = end_idx - 2     # When the gold label is off by two characters

def convert_tokens(eval_dict, qa_id, y_start_list, y_end_list):
    """Convert predictions to tokens from the context.

    Args:
        eval_dict (dict): Dictionary with eval info for the dataset. This is
            used to perform the mapping from IDs and indices to actual text.
        qa_id (int): List of QA example IDs.
        y_start_list (list): List of start predictions.
        y_end_list (list): List of end predictions.
        no_answer (bool): Questions can have no answer. E.g., SQuAD 2.0.

    Returns:
        pred_dict (dict): Dictionary index IDs -> predicted answer text.
        sub_dict (dict): Dictionary UUIDs -> predicted answer text (submission).
    """
    pred_dict = {}
    sub_dict = {}
    for qid, y_start, y_end in zip(qa_id, y_start_list, y_end_list):
        context = eval_dict[str(qid)]["context"]
        spans = eval_dict[str(qid)]["spans"]
        uuid = eval_dict[str(qid)]["uuid"]
        start_idx = spans[y_start][0]
        end_idx = spans[y_end][1]
        pred_dict[str(qid)] = context[start_idx: end_idx]
        sub_dict[uuid] = context[start_idx: end_idx]
    return pred_dict, sub_dict

def metric_max_over_ground_truths(metric_fn, prediction, ground_truths):
    if not ground_truths:
        return metric_fn(prediction, '')
    scores_for_ground_truths = []
    for ground_truth in ground_truths:
        score = metric_fn(prediction, ground_truth)
        scores_for_ground_truths.append(score)
    return max(scores_for_ground_truths)


def eval_dicts(gold_dict, pred_dict):
    avna = f1 = em = total = 0
    id2index = {curr_id : idx for idx, curr_id in enumerate(gold_dict['id'])}
    for curr_id in pred_dict:
        total += 1
        index = id2index[curr_id]
        ground_truths = gold_dict['answer'][index]['text']
        prediction = pred_dict[curr_id]
        em += metric_max_over_ground_truths(compute_em, prediction, ground_truths)
        f1 += metric_max_over_ground_truths(compute_f1, prediction, ground_truths)

    eval_dict = {'EM': 100. * em / total,
                 'F1': 100. * f1 / total}
    return eval_dict

def postprocess_qa_predictions(examples, features, predictions,
                               n_best_size=20, max_answer_length=30):
    all_start_logits, all_end_logits = predictions
    # Build a map example to its corresponding features.
    example_id_to_index = {k: i for i, k in enumerate(examples["id"])}
    features_per_example = ddict(list)
    for i, feat_id in enumerate(features['id']):
        features_per_example[example_id_to_index[feat_id]].append(i)

    # The dictionaries we have to fill.
    all_predictions = OrderedDict()

    # Let's loop over all the examples!
    #for example_index in tqdm(range(len(examples['id']))):
    for example_index in range(len(examples['id'])):
        example = {key : examples[key][example_index] for key in examples}
        # Those are the indices of the features associated to the current example.
        feature_indices = features_per_example[example_index]
        prelim_predictions = []

        # Looping through all the features associated to the current example.
        for feature_index in feature_indices:
            # We grab the predictions of the model for this feature.
            start_logits = all_start_logits[feature_index]
            end_logits = all_end_logits[feature_index]
            seq_ids = features.sequence_ids(feature_index)
            non_pad_idx = len(seq_ids) - 1
            while not seq_ids[non_pad_idx]:
                non_pad_idx -= 1
            start_logits = start_logits[:non_pad_idx]
            end_logits = end_logits[:non_pad_idx]
            # This is what will allow us to map some the positions in our logits to span of texts in the origi
            # context.
            offset_mapping = features["offset_mapping"][feature_index]
            # Optional `token_is_max_context`, if provided we will remove answers that do not have the maximum
            # available in the current feature.
            token_is_max_context = features.get("token_is_max_context", None)
            if token_is_max_context:
                token_is_max_context = token_is_max_context[feature_index]


            # Go through all possibilities for the `n_best_size` greater start and end logits.
            start_indexes = np.argsort(start_logits)[-1 : -n_best_size - 1 : -1].tolist()
            end_indexes = np.argsort(end_logits)[-1 : -n_best_size - 1 : -1].tolist()
            for start_index in start_indexes:
                for end_index in end_indexes:
                    # Don't consider out-of-scope answers, either because the indices are out of bounds or corr
                    # to part of the input_ids that are not in the context.
                    if (
                        start_index >= len(offset_mapping)
                        or end_index >= len(offset_mapping)
                        or offset_mapping[start_index] is None
                        or offset_mapping[end_index] is None
                    ):
                        continue
                    # Don't consider answers with a length that is either = 0 or > max_answer_length.
                    if end_index <= start_index or end_index - start_index + 1 > max_answer_length:
                        continue
                    # Don't consider answer that don't have the maximum context available (if such information
                    # provided).
                    if token_is_max_context is not None and not token_is_max_context.get(str(start_index), Fals
                        continue
                    prelim_predictions.append(
                        {
                            "start_index": start_index,
                            "end_index": end_index,
                            "offsets": (offset_mapping[start_index][0], offset_mapping[end_index][1]),
                            "score": start_logits[start_index] + end_logits[end_index],
                            "start_logit": start_logits[start_index],
                            "end_logit": end_logits[end_index],
                        }
                    )
        # Only keep the best `n_best_size` predictions.
        predictions = sorted(prelim_predictions, key=lambda x: x["score"], reverse=True)[:n_best_size]

        # Use the offsets to gather the answer text in the original context.
        context = example["context"]
        for pred in predictions:
            offsets = pred['offsets']
            pred["text"] = context[offsets[0] : offsets[1]]

        # In the very rare edge case we have not a single non-null prediction, we create a fake prediction to a
        # failure.
        if len(predictions) == 0:
            predictions.insert(0, {"text": "empty", "start_logit": 0.0, "end_logit": 0.0, "score": 0.0})

        # Compute the softmax of all scores (we do it with numpy to stay independent from torch/tf in this file
        # the LogSumExp trick).
        scores = np.array([pred.pop("score") for pred in predictions])
        exp_scores = np.exp(scores - np.max(scores))
        probs = exp_scores / exp_scores.sum()

        # Include the probabilities in our predictions.
        for prob, pred in zip(probs, predictions):
            pred["probability"] = prob

        # need to find the best non-empty prediction.
        i = 0
        while i < len(predictions):
            if predictions[i]['text'] != '':
                break
            i += 1
        if i == len(predictions):
            import pdb; pdb.set_trace();

        best_non_null_pred = predictions[i]
        all_predictions[example["id"]] = best_non_null_pred["text"]

    return all_predictions



# All methods below this line are from the official SQuAD 2.0 eval script
# https://worksheets.codalab.org/rest/bundles/0x6b567e1cf2e041ec80d7098f031c5c9e/contents/blob/
def normalize_answer(s):
    """Convert to lowercase and remove punctuation, articles and extra whitespace."""

    def remove_articles(text):
        regex = re.compile(r'\b(a|an|the)\b', re.UNICODE)
        return re.sub(regex, ' ', text)

    def white_space_fix(text):
        return ' '.join(text.split())

    def remove_punc(text):
        exclude = set(string.punctuation)
        return ''.join(ch for ch in text if ch not in exclude)

    def lower(text):
        return text.lower()

    return white_space_fix(remove_articles(remove_punc(lower(s))))

def get_tokens(s):
    if not s:
        return []
    return normalize_answer(s).split()

def compute_em(a_gold, a_pred):
    return int(normalize_answer(a_gold) == normalize_answer(a_pred))


def compute_f1(a_gold, a_pred):
    gold_toks = get_tokens(a_gold)
    pred_toks = get_tokens(a_pred)
    common = Counter(gold_toks) & Counter(pred_toks)
    num_same = sum(common.values())
    if len(gold_toks) == 0 or len(pred_toks) == 0:
        # If either is no-answer, then F1 is 1 if they agree, 0 otherwise
        return int(gold_toks == pred_toks)
    if num_same == 0:
        return 0
    precision = 1.0 * num_same / len(pred_toks)
    recall = 1.0 * num_same / len(gold_toks)
    f1 = (2 * precision * recall) / (precision + recall)
    return f1
```

```
Writing /tmp/util.py
```

train.py

In [ ]:

```python
%%writefile /tmp/train.py

import argparse
import json
import os
from collections import OrderedDict
import torch
import csv
import util
from transformers import DistilBertTokenizerFast
from transformers import DistilBertForQuestionAnswering
from transformers import AdamW
from tensorboardX import SummaryWriter


from torch.utils.data import DataLoader
from torch.utils.data.sampler import RandomSampler, SequentialSampler
from args import get_train_test_args

from tqdm import tqdm
from ChunkData import *

#torch.set_default_tensor_type('torch.cuda.FloatTensor')

def prepare_eval_data(dataset_dict, tokenizer):
    tokenized_examples = tokenizer(dataset_dict['question'],
                                   dataset_dict['context'],
                                   truncation="only_second",
                                   stride=128,
                                   max_length=384,
                                   return_overflowing_tokens=True,
                                   return_offsets_mapping=True,
                                   padding='max_length')
    # Since one example might give us several features if it has a long context, we need a map from a feature t
    # its corresponding example. This key gives us just that.
    sample_mapping = tokenized_examples.pop("overflow_to_sample_mapping")

    # For evaluation, we will need to convert our predictions to substrings of the context, so we keep the
    # corresponding example_id and we will store the offset mappings.
    tokenized_examples["id"] = []
    for i in tqdm(range(len(tokenized_examples["input_ids"]))):
        # Grab the sequence corresponding to that example (to know what is the context and what is the questio
        sequence_ids = tokenized_examples.sequence_ids(i)
        # One example can give several spans, this is the index of the example containing this span of text.
        sample_index = sample_mapping[i]
        tokenized_examples["id"].append(dataset_dict["id"][sample_index])
        # Set to None the offset_mapping that are not part of the context so it's easy to determine if a token
        # position is part of the context or not.
        tokenized_examples["offset_mapping"][i] = [
            (o if sequence_ids[k] == 1 else None)
            for k, o in enumerate(tokenized_examples["offset_mapping"][i])
        ]

    return tokenized_examples



def prepare_train_data(dataset_dict, tokenizer, remove_data=False):
    import gc
    import shutil
    tokenized_examples = tokenizer(dataset_dict['question'],
                                   dataset_dict['context'],
                                   truncation="only_second",
                                   stride=128,
                                   max_length=384,
                                   return_overflowing_tokens=True,
                                   return_offsets_mapping=True,
                                   padding='max_length')
    orig_tokenized_examples = tokenized_examples
    sample_mapping = orig_tokenized_examples["overflow_to_sample_mapping"]
    offset_mapping = orig_tokenized_examples["offset_mapping"]


    # Let's label those examples!
    tokenized_examples["start_positions"] = []
    tokenized_examples["end_positions"] = []
    tokenized_examples['id'] = []

    temp_tokenized_examples = tokenized_examples
    if remove_data:
        for c in temp_tokenized_examples.keys():
            try:
                shutil.rmtree(f"/tmp/train/{str(c)}")
            except:
                print("Could not remove ", c)

    # TODO: RAJBIR: my additions, not sure if they work
    tokenized_examples = {str(c): ChunkDataWriter(f"/tmp/train/{str(c)}") for c in temp_tokenized_examples.keys

    for key in temp_tokenized_examples.keys():
        key = str(key)
        if key not in ["start_positions", "end_positions", "id", "sequence_ids", "overflow_to_sample_mapping",
            v_from = temp_tokenized_examples[key]
            v_to = tokenized_examples[key]
            #print(f"Copying {key}... {len(v_from)} {type(v_from)}")
            for i in range(len(v_from)):
                v_to.append(v_from[i])
                if key not in ["offset_mapping", "overflow_mapping"]:
                    v_from[i] = None
                    gc.collect(0)
                    gc.collect(1)
                    gc.collect(2)
            temp_tokenized_examples[key] = None
            gc.collect(0)
            gc.collect(1)
            gc.collect(2)
            # print("done")
            v_to.finalize()

    inaccurate = 0
    for i, offsets in enumerate(offset_mapping):
        # We will label impossible answers with the index of the CLS token.
        input_ids = tokenized_examples["input_ids"][i]
        cls_index = input_ids.index(tokenizer.cls_token_id)

        # Grab the sequence corresponding to that example (to know what is the context and what is the questio
        sequence_ids = orig_tokenized_examples.sequence_ids(i)

        # One example can give several spans, this is the index of the example containing this span of text.
        sample_index = sample_mapping[i]
        answer = dataset_dict['answer'][sample_index]
        # Start/end character index of the answer in the text.
        start_char = answer['answer_start'][0]
        end_char = start_char + len(answer['text'][0])
        tokenized_examples['id'].append(dataset_dict['id'][sample_index])
        # Start token index of the current span in the text.
        token_start_index = 0
        while sequence_ids[token_start_index] != 1:
            token_start_index += 1

        # End token index of the current span in the text.
        token_end_index = len(input_ids) - 1
        while sequence_ids[token_end_index] != 1:
            token_end_index -= 1

        # Detect if the answer is out of the span (in which case this feature is labeled with the CLS index).
        if not (offsets[token_start_index][0] <= start_char and offsets[token_end_index][1] >= end_char):
            tokenized_examples["start_positions"].append(cls_index)
            tokenized_examples["end_positions"].append(cls_index)
        else:
            # Otherwise move the token_start_index and token_end_index to the two ends of the answer.
            # Note: we could go after the last offset if the answer is the last word (edge case).
            while token_start_index < len(offsets) and offsets[token_start_index][0] <= start_char:
                token_start_index += 1
            tokenized_examples["start_positions"].append(token_start_index - 1)
            while offsets[token_end_index][1] >= end_char:
                token_end_index -= 1
            tokenized_examples["end_positions"].append(token_end_index + 1)
            # assertion to check if this checks out
            context = dataset_dict['context'][sample_index]
            offset_st = offsets[tokenized_examples['start_positions'][-1]][0]
            offset_en = offsets[tokenized_examples['end_positions'][-1]][1]
            if context[offset_st : offset_en] != answer['text'][0]:
                inaccurate += 1

    total = len(tokenized_examples['id'])
    # print(f"Preprocessing not completely accurate for {inaccurate}/{total} instances")
    # print([key for key in tokenized_examples.keys()])
    # TODO: RAJBIR: my additions, not sure if they work
    tokenized_examples = {k: ChunkDataReader(v) for k, v in tokenized_examples.items()}
    return tokenized_examples



"""
def read_and_process(args, tokenizer, dataset_dict, dir_name, dataset_name, split):
    #TODO: cache this if possible
    cache_path = f'{dir_name}/{dataset_name}_encodings.pt'
    if os.path.exists(cache_path) and not args.recompute_features:
        tokenized_examples = util.load_pickle(cache_path)
    else:
        if split=='train':
            tokenized_examples = prepare_train_data(dataset_dict, tokenizer)
        else:
            tokenized_examples = prepare_eval_data(dataset_dict, tokenizer)
        util.save_pickle(tokenized_examples, cache_path)
    return tokenized_examples
"""

def get_cached_dataset():
    import os
    import glob
    if os.path.exists("/tmp/train/input_ids/config.json"):
        g = glob.glob("/tmp/train/*")
        g = [x.split('/')[-1] for x in g]
        out_dict = {}
        for name in g:
            val = ChunkDataReader(f"/tmp/train/{name}", chunk_size=16)
            out_dict[name] = val
        return out_dict
    return None

def read_and_process(args, tokenizer, dataset_dict, dir_name, dataset_name, split):
    #TODO: cache this if possible
    def get_dataset_slice(dataset_dict, slice_start, slice_length):
        out_dict = {k: [] for k in dataset_dict.keys()}
        for k in dataset_dict.keys():
            arr = dataset_dict[k]
            out_dict[k] = arr[slice_start:slice_start+slice_length]
        return out_dict

    def get_dataset_length(dataset_dict):
        column = [c for c in dataset_dict.keys()][0]
        return len(dataset_dict[column])

    cache_path = f'{dir_name}/{dataset_name}_encodings.pt'
    for key in dataset_dict.keys():
        dataset_dict[key] = dataset_dict[key]
    if split=='train':
        cached_dataset = get_cached_dataset()
        if not cached_dataset is None:
            print("USING CACHED DATASET THAT HAS ALREADY BEEN PREPROCESSED")
            return cached_dataset
        cleardir = True
        #dataset_dict = get_dataset_slice(dataset_dict, 0, 8192)
        stride = 512
        for i in tqdm(range(0, get_dataset_length(dataset_dict), stride), "Preparing training data"):
            temp_dict = get_dataset_slice(dataset_dict, i, stride)
            tokenized_examples = prepare_train_data(temp_dict, tokenizer, cleardir)
            cleardir = False
            # for c in tokenized_examples.keys():
            #     print(c, len(tokenized_examples[str(c)]))
            # print('-' * 80)
    else:
        if os.path.exists(cache_path) and not args.recompute_features:
            tokenized_examples = util.load_pickle(cache_path)
            return tokenized_examples
        tokenized_examples = prepare_eval_data(dataset_dict, tokenizer)
    util.save_pickle(tokenized_examples, cache_path)
    return tokenized_examples



#TODO: use a logger, use tensorboard
class Trainer():
    def __init__(self, args, log):
        self.lr = args.lr
        self.num_epochs = args.num_epochs
        self.device = args.device
        self.eval_every = args.eval_every
        self.path = os.path.join(args.save_dir, 'checkpoint')
        self.num_visuals = args.num_visuals
        self.save_dir = args.save_dir
        self.log = log
        self.visualize_predictions = args.visualize_predictions
        if not os.path.exists(self.path):
            os.makedirs(self.path)

    def save(self, model):
        model.save_pretrained(self.path)

    def evaluate(self, model, data_loader, data_dict, return_preds=False, split='validation'):
        device = self.device

        # print("\nEvaluating")
        model.eval()
        pred_dict = {}
        all_start_logits = []
        all_end_logits = []
        with torch.no_grad(): #, \
                #tqdm(total=len(data_loader.dataset)) as progress_bar:
            for batch in data_loader:
                # Setup for forward
                input_ids = batch['input_ids'].to(device)
                attention_mask = batch['attention_mask'].to(device)
                batch_size = len(input_ids)
                outputs = model(input_ids, attention_mask=attention_mask)
                # Forward
                start_logits, end_logits = outputs.start_logits, outputs.end_logits
                # TODO: compute loss

                all_start_logits.append(start_logits)
                all_end_logits.append(end_logits)
                #progress_bar.update(batch_size)

        # Get F1 and EM scores
        start_logits = torch.cat(all_start_logits).cpu().numpy()
        end_logits = torch.cat(all_end_logits).cpu().numpy()
        preds = util.postprocess_qa_predictions(data_dict,
                                                 data_loader.dataset.encodings,
                                                 (start_logits, end_logits))
        if split == 'validation':
            results = util.eval_dicts(data_dict, preds)
            results_list = [('F1', results['F1']),
                            ('EM', results['EM'])]
        else:
            results_list = [('F1', -1.0),
                            ('EM', -1.0)]
        results = OrderedDict(results_list)
        if return_preds:
            return preds, results
        return results

    def train(self, model, train_dataloader, eval_dataloader, val_dict):
        device = self.device
        model.to(device)
        optim = AdamW(model.parameters(), lr=self.lr)
        global_idx = 0
        best_scores = {'F1': -1.0, 'EM': -1.0}
        tbx = SummaryWriter(self.save_dir)

        for epoch_num in range(self.num_epochs):
            self.log.info(f'Epoch: {epoch_num}')
            with torch.enable_grad(), tqdm(total=len(train_dataloader.dataset)) as progress_bar:
                for batch in train_dataloader:
                    optim.zero_grad()
                    model.train()
                    input_ids = batch['input_ids'].to(device)
                    attention_mask = batch['attention_mask'].to(device)
                    start_positions = batch['start_positions'].to(device)
                    end_positions = batch['end_positions'].to(device)
                    outputs = model(input_ids, attention_mask=attention_mask,
                                    start_positions=start_positions,
                                    end_positions=end_positions)
                    loss = outputs[0]
                    loss.backward()
                    optim.step()
                    progress_bar.update(len(input_ids))
                    progress_bar.set_postfix(epoch=epoch_num, NLL=loss.item())
                    tbx.add_scalar('train/NLL', loss.item(), global_idx)
                    if (global_idx % self.eval_every) == 0:
                        #self.log.info(f'Evaluating at step {global_idx}...')
                        preds, curr_score = self.evaluate(model, eval_dataloader, val_dict, return_preds=True)
                        results_str = ', '.join(f'{k}: {v:05.2f}' for k, v in curr_score.items())
                        #self.log.info('Visualizing in TensorBoard...')
                        for k, v in curr_score.items():
                            tbx.add_scalar(f'val/{k}', v, global_idx)
                        self.log.info(f'Eval {results_str}')
                        if self.visualize_predictions:
                            util.visualize(tbx,
                                           pred_dict=preds,
                                           gold_dict=val_dict,
                                           step=global_idx,
                                           split='val',
                                           num_visuals=self.num_visuals)
                        if curr_score['F1'] >= best_scores['F1']:
                            best_scores = curr_score
                            self.save(model)
                    global_idx += 1
        preds, curr_score = self.evaluate(model, eval_dataloader, val_dict, return_preds=True)
        results_str = ', '.join(f'{k}: {v:05.2f}' for k, v in curr_score.items())
        self.log.info(f'Final eval results after epoch: {results_str}')
        self.log.info(f'After epoch, best scores so far: {best_scores}')
        return best_scores

def get_dataset(args, datasets, data_dir, tokenizer, split_name):
    datasets = datasets.split(',')
    dataset_dict = None
    dataset_name=''
    for dataset in datasets:
        dataset_name += f'_{dataset}'
        dataset_dict_curr = util.read_squad(f'{data_dir}/{dataset}')
        dataset_dict = util.merge(dataset_dict, dataset_dict_curr)
    data_encodings = read_and_process(args, tokenizer, dataset_dict, data_dir, dataset_name, split_name)
    return util.QADataset(data_encodings, train=(split_name=='train')), dataset_dict

#import torch_xla.core.xla_model as xm
def main():
    # define parser and arguments
    args = get_train_test_args()

    util.set_seed(args.seed)
    model = DistilBertForQuestionAnswering.from_pretrained("distilbert-base-uncased")
    tokenizer = DistilBertTokenizerFast.from_pretrained('distilbert-base-uncased')

    if args.do_train:
        if not os.path.exists(args.save_dir):
            os.makedirs(args.save_dir)
        args.save_dir = util.get_save_dir(args.save_dir, args.run_name)
        log = util.get_logger(args.save_dir, 'log_train')
        log.info(f'Args: {json.dumps(vars(args), indent=4, sort_keys=True)}')
        log.info("Preparing Training Data...")
        args.device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')
        if torch.cuda.is_available():
            print("USING CUDA !!!!")
        else:
            #print(xm.xla_device())
            #if xm.xla_device() != None:
            if False:
                args.device = xm.xla_device()
                print("USING TPU!!! ")
            else:
                printt("USING CPU !!!")
        trainer = Trainer(args, log)
        train_dataset, _ = get_dataset(args, args.train_datasets, args.train_dir, tokenizer, 'train')
        log.info("Preparing Validation Data...")
        val_dataset, val_dict = get_dataset(args, args.train_datasets, args.val_dir, tokenizer, 'val')
        train_loader = DataLoader(train_dataset,
                                  batch_size=args.batch_size,
                                  sampler=RandomSampler(train_dataset))
        val_loader = DataLoader(val_dataset,
                                batch_size=args.batch_size,
                                sampler=SequentialSampler(val_dataset))
        best_scores = trainer.train(model, train_loader, val_loader, val_dict)
    if args.do_eval:
        args.device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')
        split_name = 'test' if 'test' in args.eval_dir else 'validation'
        log = util.get_logger(args.save_dir, f'log_{split_name}')
        trainer = Trainer(args, log)
        checkpoint_path = os.path.join(args.save_dir, 'checkpoint')
        model = DistilBertForQuestionAnswering.from_pretrained(checkpoint_path)
        model.to(args.device)
        eval_dataset, eval_dict = get_dataset(args, args.eval_datasets, args.eval_dir, tokenizer, split_name)
        eval_loader = DataLoader(eval_dataset,
                                 batch_size=args.batch_size,
                                 sampler=SequentialSampler(eval_dataset))
        eval_preds, eval_scores = trainer.evaluate(model, eval_loader,
                                                   eval_dict, return_preds=True,
                                                   split=split_name)
        results_str = ', '.join(f'{k}: {v:05.2f}' for k, v in eval_scores.items())
        log.info(f'Eval {results_str}')
        # Write submission file
        sub_path = os.path.join(args.save_dir, split_name + '_' + args.sub_file)
        log.info(f'Writing submission file to {sub_path}...')
        with open(sub_path, 'w', newline='', encoding='utf-8') as csv_fh:
            csv_writer = csv.writer(csv_fh, delimiter=',')
            csv_writer.writerow(['Id', 'Predicted'])
            for uuid in sorted(eval_preds):
                csv_writer.writerow([uuid, eval_preds[uuid]])


if __name__ == '__main__':
    main()
```

```
Overwriting /tmp/train.py
```

run.sh

In [ ]:

```bash
%%writefile run.sh
#!/bin/bash

# Modify this if python version is different
export PATH=$PATH:/usr/lib/python3.7/venv/scripts/common/:/usr/lib/python3.6/venv/scripts/common/activate
cd mtu-nlp-assignment/assignment2/
rm -rf robustqa
unzip -o robustqa.zip
cd robustqa
mv datasets_50k.tar.gz  datasets_50k.tar
tar -xf datasets_50k.tar
source activate robustqa
cp -f /tmp/args.py ./args.py
cp -f /tmp/train.py ./train.py
cp -f /tmp/util.py ./util.py
cp -f /tmp/ChunkData.py ./ChunkData.py
grep batch.size args.py
python train.py --do-train --run-name baseline
```

```
Overwriting run.sh
```

Run the model

In [15]:

```
!bash ./run.sh
```

```
Archive:  robustqa.zip
   creating: robustqa/
  inflating: robustqa/datasets_50k.tar.gz
   creating: __MACOSX/
   creating: __MACOSX/robustqa/
  inflating: __MACOSX/robustqa/._datasets_50k.tar.gz
  inflating: robustqa/convert_to_squad.py
  inflating: robustqa/environment.yml
  inflating: robustqa/util.py
  inflating: robustqa/README.md
  inflating: __MACOSX/robustqa/._README.md
  inflating: robustqa/train.py
  inflating: robustqa/args.py
    parser.add_argument('--batch-size', type=int, default=32)
Downloading: 100% 483/483 [00:00<00:00, 584kB/s]
Downloading: 100% 268M/268M [00:06<00:00, 44.1MB/s]
Some weights of the model checkpoint at distilbert-base-uncased were not used when initializing DistilBertForQu
estionAnswering: ['vocab_transform.weight', 'vocab_transform.bias', 'vocab_layer_norm.weight', 'vocab_layer_nor
m.bias', 'vocab_projector.weight', 'vocab_projector.bias']
- This IS expected if you are initializing DistilBertForQuestionAnswering from the checkpoint of a model traine
d on another task or with another architecture (e.g. initializing a BertForSequenceClassification model from a
BertForPreTraining model).
- This IS NOT expected if you are initializing DistilBertForQuestionAnswering from the checkpoint of a model th
at you expect to be exactly identical (initializing a BertForSequenceClassification model from a BertForSequenc
eClassification model).
Some weights of DistilBertForQuestionAnswering were not initialized from the model checkpoint at distilbert-bas
e-uncased and are newly initialized: ['qa_outputs.weight', 'qa_outputs.bias']
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
Downloading: 100% 232k/232k [00:00<00:00, 2.96MB/s]
Downloading: 100% 466k/466k [00:00<00:00, 4.76MB/s]
[12.21.21 10:42:31] Args: {
    "batch_size": 32,
    "do_eval": false,
    "do_train": true,
    "eval": false,
    "eval_datasets": "race,relation_extraction,duorc",
    "eval_dir": "datasets/oodomain_test",
    "eval_every": 2500,
    "lr": 3e-05,
    "num_epochs": 8,
    "num_visuals": 10,
    "recompute_features": false,
    "run_name": "baseline",
    "save_dir": "save/baseline-01",
    "seed": 42,
    "sub_file": "",
    "train": false,
    "train_datasets": "squad,nat_questions,newsqa",
    "train_dir": "datasets/indomain_train",
    "val_dir": "datasets/indomain_val",
    "visualize_predictions": false
}
[12.21.21 10:42:31] Preparing Training Data...
USING CUDA !!!!
USING CACHED DATASET THAT HAS ALREADY BEEN PREPROCESSED
[12.21.21 10:42:35] Preparing Validation Data...
100% 38888/38888 [00:02<00:00, 15390.09it/s]
[12.21.21 10:42:55] Epoch: 0
[12.21.21 10:47:11] Eval F1: 07.68, EM: 00.05
[12.21.21 11:17:40] Eval F1: 63.55, EM: 47.31
[12.21.21 11:48:08] Eval F1: 66.63, EM: 50.57
[12.21.21 12:18:39] Eval F1: 68.67, EM: 52.48
100% 242304/242304 [1:36:29<00:00, 41.86it/s, NLL=1.04, epoch=0]
[12.21.21 12:19:24] Epoch: 1
[12.21.21 12:49:09] Eval F1: 69.21, EM: 53.00
[12.21.21 13:19:39] Eval F1: 69.51, EM: 53.75
[12.21.21 13:50:09] Eval F1: 70.56, EM: 54.41
100% 242304/242304 [1:32:15<00:00, 43.77it/s, NLL=0.686, epoch=1]
[12.21.21 13:51:40] Epoch: 2
[12.21.21 14:20:41] Eval F1: 69.77, EM: 53.81
[12.21.21 14:51:11] Eval F1: 70.21, EM: 54.16
[12.21.21 15:21:41] Eval F1: 70.66, EM: 54.58
100% 242304/242304 [1:32:17<00:00, 43.76it/s, NLL=0.491, epoch=2]
[12.21.21 15:23:57] Epoch: 3
[12.21.21 15:52:12] Eval F1: 70.23, EM: 54.42
[12.21.21 16:22:44] Eval F1: 70.22, EM: 53.96
[12.21.21 16:53:18] Eval F1: 70.37, EM: 54.66
100% 242304/242304 [1:32:22<00:00, 43.72it/s, NLL=0.354, epoch=3]
[12.21.21 16:56:19] Epoch: 4
[12.21.21 17:23:53] Eval F1: 69.78, EM: 53.65
[12.21.21 17:54:25] Eval F1: 69.90, EM: 53.60
[12.21.21 18:24:53] Eval F1: 69.82, EM: 53.51
100% 242304/242304 [1:32:19<00:00, 43.74it/s, NLL=0.321, epoch=4]
[12.21.21 18:28:39] Epoch: 5
[12.21.21 18:55:21] Eval F1: 69.63, EM: 53.08
[12.21.21 19:25:48] Eval F1: 69.19, EM: 52.96
[12.21.21 19:56:16] Eval F1: 69.53, EM: 53.51
100% 242304/242304 [1:32:08<00:00, 43.83it/s, NLL=0.35, epoch=5]
[12.21.21 20:00:47] Epoch: 6
[12.21.21 20:26:45] Eval F1: 69.40, EM: 52.89
[12.21.21 20:57:13] Eval F1: 69.25, EM: 53.11
[12.21.21 21:27:45] Eval F1: 69.23, EM: 52.95
100% 242304/242304 [1:32:14<00:00, 43.78it/s, NLL=0.269, epoch=6]
[12.21.21 21:33:02] Epoch: 7
[12.21.21 21:58:17] Eval F1: 69.18, EM: 52.38
[12.21.21 22:28:51] Eval F1: 68.90, EM: 52.54
[12.21.21 22:59:21] Eval F1: 68.82, EM: 52.32
100% 242304/242304 [1:32:20<00:00, 43.73it/s, NLL=0.169, epoch=7]
[12.21.21 23:09:40] Final eval results after epoch: F1: 69.22, EM: 52.60
[12.21.21 23:09:40] After epoch, best scores so far: OrderedDict([('F1', 70.66428123994659), ('EM', 54.58034615
10609)])
```

Save pre-processed dataset

Save the tokenized and pre-processed dataset for future use

In [16]:

```bash
%%writefile save_train.sh
#!/bin/bash
THEDATE=`date +%Y-%m-%d_%H-%M-%S`

echo "COPYING SAVE"
echo "------------"
THEDATE=`date +%Y-%m-%d_%H-%M-%S`
cd mtu-nlp-assignment/assignment2/robustqa
tar -cf "save-${THEDATE}.tar" save
gzip "save-${THEDATE}.tar"
cp "save-${THEDATE}.tar.gz" /content/gdrive/MyDrive/NLP/NLP-Save-Train
du -sh "/content/gdrive/MyDrive/NLP/NLP-Save-Train/save-${THEDATE}.tar.gz"

echo "---------------------------------------------------------"

if [[  -f /tmp/train/input_ids/config.json ]]
then
    echo "COPYING PREPROCESSED TRAINING DATA"
    echo "----------------------------------"
    cd /tmp
    rm -f train.tar train.tar.gz
    tar -cf "train-${THEDATE}.tar" train
    gzip "train-${THEDATE}.tar"
    mkdir -p /content/gdrive/MyDrive/NLP/NLP-Save-Train
    cp "train-${THEDATE}.tar.gz" "/content/gdrive/MyDrive/NLP/NLP-Save-Train/train-${THEDATE}.tar.gz"
    du -sh "/content/gdrive/MyDrive/NLP/NLP-Save-Train/train-${THEDATE}.tar.gz"
fi
```

```
Writing save_train.sh
```

In [17]:

```
!/bin/bash save_train.sh
```

```
COPYING SAVE
------------
235M    /content/gdrive/MyDrive/NLP/NLP-Save-Train/save-2021-12-21_23-09-43.tar.gz
---------------------------------------------------------
COPYING PREPROCESSED TRAINING DATA
----------------------------------
48M     /content/gdrive/MyDrive/NLP/NLP-Save-Train/train-2021-12-21_23-09-43.tar.gz
```

In [ ]:

```
!find /tmp/train -name config.json -exec ls {} \; -exec cat {} \; -exec echo \; -exec echo \;
```

```
/tmp/train/start_positions/config.json
{"length": 242304}

/tmp/train/id/config.json
{"length": 242304}

/tmp/train/overflow_to_sample_mapping/config.json
{"length": 0}

/tmp/train/attention_mask/config.json
{"length": 242304}

/tmp/train/end_positions/config.json
{"length": 242304}

/tmp/train/input_ids/config.json
{"length": 242304}

/tmp/train/offset_mapping/config.json
{"length": 0}
```

In [18]: