Lab 2: Fine-tuning BERT To Perform Common Sense Reasoning

May 13, 2024

Welcome to the Lab 2 of our course on Natural Language Processing. As the name suggests in this lab you will learn how to fine-tune a pretrained model like BERT on a downstream task to improve much more superior performance compared to the methods discussed so far. We will be working with the [SocialIQA](#) dataset this week, which is a multiple choice classification dataset designed to learn and measure social and emotional intelligence in NLP models.

This assignment will also make heavy use of the [🤗 Transformers Library](#). Don't worry if you are not familiar with the library, we will discuss its usage in detail.

Note: Access to a GPU will be crucial for working on this assignment. So do select a GPU runtime in Colab before you start working.

Learning Outcomes from this Lab:

- Learn how to use 🤗 Transformer library to load and fine-tune pre-trained langauge models
- Learn how to solve common sense reasoning problems using Masked Language Models like BERT

Suggested Reading:

- [Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*](#)
- [Maarten Sap, Hannah Rashkin, Derek Chen, Ronan Le Bras, and Yejin Choi. 2019. Social IQa: Commonsense Reasoning about Social Interactions. In Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP), pages 4463–4473, Hong Kong, China. Association for Computational Linguistics.] (https://arxiv.org/pdf/1810.04805.pdf)

```
from google.colab import drive
drive.mount('/content/gdrive')
siqa_data_dir = "gdrive/MyDrive/PlakshaTLF24-NLP/Lab02/data/socialiqa-train-dev/" #renamed basis directory organization on my drive.
```

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

```
!ls -l gdrive/MyDrive/PlakshaTLF24-NLP/Lab02/data/socialiqa-train-dev/
```

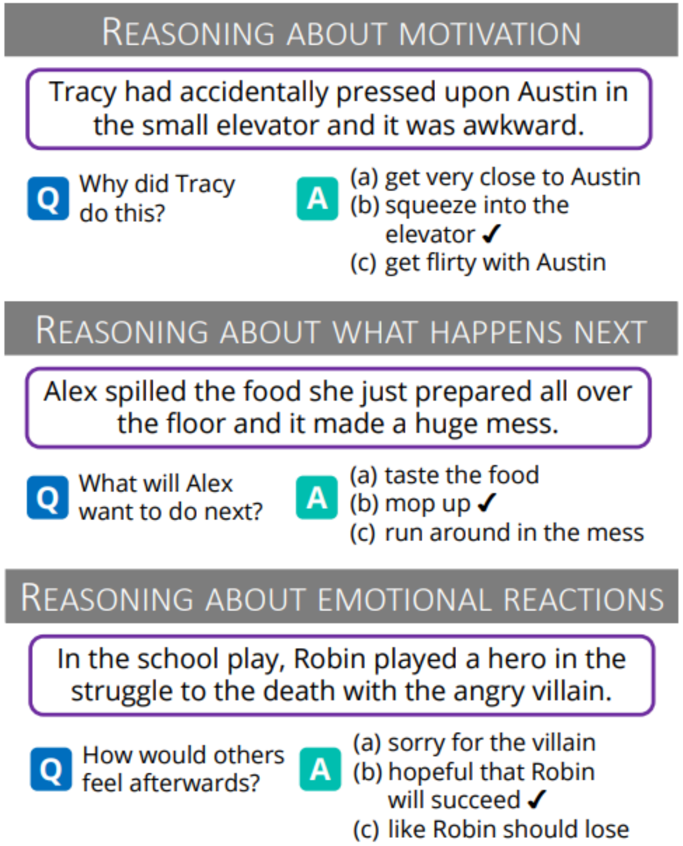
```
total 8479
-rw------- 1 root root  476394 May 13 13:39 dev.jsonl
-rw------- 1 root root    5862 May 13 13:39 dev-labels.lst
-rw------- 1 root root 8098489 May 13 13:39 train.jsonl
-rw------- 1 root root  100230 May 13 13:39 train-labels.lst
```

```
# If using Colab, NO NEED TO INSTALL ANYTHING
# Install required libraries
# !pip install numpy
# !pip install pandas
# !pip install torch
# !pip install tqdm
# !pip install matplotlib
# !pip install transformers
# !pip install scikit-learn
# !pip install tqdm
```

```
# We start by importing libraries that we will be making use of in the assignment.
import os
from functools import partial
import json
from pprint import pprint
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
from torch.optim import Adam
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
import copy
from tqdm.notebook import tqdm

from transformers.utils import logging
logging.set_verbosity(40) # to avoid warnings from transformers
```

SocialIQA Dataset

We start by discussing the dataset that we will making use of in today's Lab. As described above SocialIQA was designed to learn and measure social and emotional intelligence in NLP models. It is a multiple choice classification task, where you are given a context of some social situation, a question about the context and then three possible answers to the questions. The task is to predict which of the three options answers the question given the context.



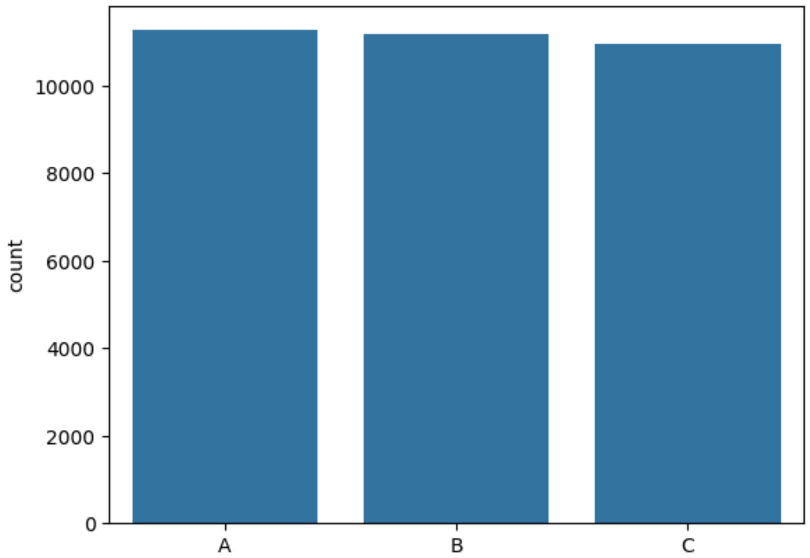
Below we load the dataset in memory

```python
def load_siqa_data(split):

    # We first load the file containing context, question and answers
    with open(f"gdrive/MyDrive/PlakshaTLF24-NLP/Lab02/data/socialiqa-train-dev/{split}.jsonl") as f:
        data = [json.loads(jline) for jline in f.read().splitlines()]

    # We then load the file containing the correct answer for each question
    with open(f"gdrive/MyDrive/PlakshaTLF24-NLP/Lab02/data/socialiqa-train-dev/{split}-labels.lst") as f:
        labels = f.read().splitlines()

    labels_dict = {"1": "A", "2": "B", "3": "C"}
    labels = [labels_dict[label] for label in labels]

    return data, labels


train_data, train_labels = load_siqa_data("train")
dev_data, dev_labels = load_siqa_data("dev")

print(f"Number of Training Examples: {len(train_data)}")
print(f"Number of Validation Examples: {len(dev_data)}")
```

```
Number of Training Examples: 33410
Number of Validation Examples: 1954
```

```python
sns.countplot(x = train_labels)
```

```
<Axes: ylabel='count'>
```



```
# View a sample of the dataset
print("Example from dataset")
pprint(train_data[100], sort_dicts=False, indent=4)
print(f"Label: {train_labels[100]}")
```

```
Example from dataset
{   'context': "Jordan's dog peed on the couch they were selling and Jordan "
               'removed the odor as soon as possible.',
    'question': 'How would Jordan feel afterwards?',
    'answerA': 'selling a couch',
    'answerB': 'Disgusted',
    'answerC': 'Relieved'}
Label: B
```

```
train_data[500]
```

```
{'context': 'kendall was a person who kept her word so she got my money the other day.',
 'question': 'What will Others want to do next?',
 'answerA': 'resent kendall',
 'answerB': 'support kendall',
 'answerC': 'hate kendall'}
```

Task 1: Tokenization and Data Preperation (1 hour)

As discussed in the lectures, BERT and other pretrained language models use sub-word tokenization i.e. individual words can also be split into constituent subwords to reduce the vocabulary size. The Transformer library provides tokenizer for all the popular language models. Below we demonstrate how to create and use these tokenizers.

```
# Import the BertTokenizer from the library
from transformers import BertTokenizer

# Load a pre-trained BERT Tokenizer
bert_tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
```

```
/usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/_token.py:88: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens), set it as secret in your Google Colab and restart your session.
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(
```

tokenizer_config.json: 100% 48.0/48.0 [00:00<00:00, 2.30kB/s]

vocab.txt: 100% 232k/232k [00:00<00:00, 2.07MB/s]

tokenizer.json: 100% 466k/466k [00:00<00:00, 2.71MB/s]

config.json: 100% 570/570 [00:00<00:00, 36.6kB/s]

`BertTokenizer.from_pretrained` is used to load a pre-trained tokenizer. Notice that we provide the argument `"bert-base-uncased"` to the method. This refers to the variant of BERT that we want to use. The term "base" means we want to use the smaller BERT variant i.e. the one with 12 layers, and "uncased" refers to the fact that it treats upper-case and lower-case characters identically. There are 4 variants available for BERT which are: - `bert-base-uncased` - `bert-base-cased` - `bert-large-uncased` - `bert-large-cased` Now that we have loaded the tokenizer, let's see how to use it.

`tokenize` method can be used to split the text into sequence of tokens

```
bert_tokenizer.tokenize("kendall was a person who kept her word exquisitely, so she got my money the other day")
```

```
['kendall',
 'was',
 'a',
 'person',
 'who',
 'kept',
 'her',
 'word',
 'exquisite',
 '##ly',
 ',',
 'so',
 'she',
 'got',
 'my',
 'money',
 'the',
 'other',
 'day']
```

Notice how the tokenizer not only splits the text into words but also subwords like "exquisitely" is split into "exquisite" and "ly".

Another use case of the tokenizer is to convert the tokens into indices. This is important because BERT and almost all language models takes as the inputs a sequence of token ids, which they use to map into embeddings. `convert_tokens_to_ids` method can be used to do this

```
sentence = "kendall was a person who kept her word exquisitely, so she got my money the other day"
tokens = bert_tokenizer.tokenize(sentence)
token_ids = bert_tokenizer.convert_tokens_to_ids(tokens)
print(token_ids)
```

```
[14509, 2001, 1037, 2711, 2040, 2921, 2014, 2773, 19401, 2135, 1010, 2061, 2016, 2288, 2026, 2769, 1996, 2060, 2154]
```

The two steps can also be combined by simply calling the tokenizer object

```
pprint(bert_tokenizer(sentence), sort_dicts=False, indent=4)
```

```
{   'input_ids': [   101,
                     14509,
                     2001,
                     1037,
                     2711,
                     2040,
                     2921,
                     2014,
                     2773,
                     19401,
                     2135,
                     1010,
                     2061,
                     2016,
                     2288,
                     2026,
                     2769,
                     1996,
                     2060,
                     2154,
                     102],
    'token_type_ids': [   0,
                          0,
                          0,
                          0,
                          0,
                          0,
                          0,
                          0,
                          0,
                          0,
                          0,
                          0,
                          0,
                          0,
                          0,
                          0,
                          0,
                          0,
                          0,
                          0,
                          0],
    'attention_mask': [   1,
                          1,
                          1,
                          1,
                          1,
                          1,
                          1,
                          1,
                          1,
                          1,
                          1,
                          1,
                          1,
                          1,
                          1,
                          1,
```

Notice that it returns a bunch of things in addition to the ids. The `"input_ids"` are just the token ids that we obtained in the previous cell. However you will notice that it has a few additional ids, it starts with 101 and ends with 102. These are what we call special tokens and correspond the [CLS] and [SEP] tokens used by BERT. [CLS] token is mainly added to beginning of each sequence, and its representations are used to perform sequence classification. More on [SEP] token later.

`"token_type_ids"` contains which sequence does a particular token belongs to.

`"attention_mask"` is a mask vector that indicates if a particular token corresponds to padding. Padding is extremely important when we are dealing with variable length sequences, which is almost always the case. Through padding we can ensure that all the sequences in a batch are of same size. However, while processing the sequence we need ignore these padding tokens, hence a mask is required to identify such tokens.

We can tokenize a batch of sequences by just providing a list instead of a string while calling the tokenizer and later pad them using the `.pad` method.

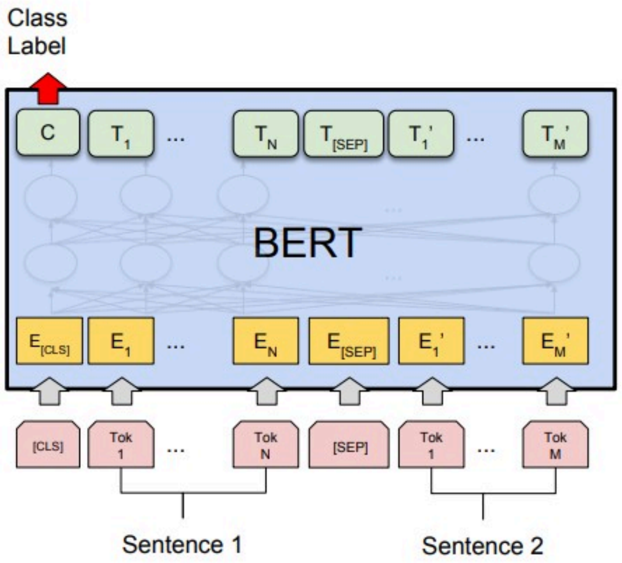
```
batch_size = 4
sentence_batch = [train_data[i]["context"] for i in range(batch_size)]

#Tokenize the batch of sequences
tokenized_batch = bert_tokenizer(sentence_batch)

# Pad the tokenized batch
tokenized_batch_padded = bert_tokenizer.pad(tokenized_batch, padding=True, max_length=32, return_tensors="pt")

input_ids = tokenized_batch_padded["input_ids"]
attn_mask = tokenized_batch_padded["attention_mask"]
print(f"Input Ids shape: {input_ids.shape}")
print(f"Attention Mask shape: {attn_mask.shape}")

pprint(f"Input Ids:\n {input_ids}\n")
pprint(f"Attention Mask:\n {attn_mask}\n")
```

```
Input Ids shape: torch.Size([4, 23])
Attention Mask shape: torch.Size([4, 23])
('Input Ids:\n'
 ' tensor([[  101,  7232,  2787,  2000,  2031,  1037, 26375,  1998,  5935,  '
 '2014,\n'
 '          2814,  2362,  1012,   102,     0,     0,     0,     0,     0,     '
 '0,\n'
 '             0,     0,     0],\n'
 '        [  101,  5553,  2734,  2000,  2507,  2041,  5841,  2005,  2019,  '
 '9046,\n'
 '          2622,  2012,  2147,  1012,   102,     0,     0,     0,     0,     '
 '0,\n'
 '             0,     0,     0],\n'
 '        [  101, 22712,  2001,  2019,  6739, 19949,  1998,  2001,  2006,  '
 '1996,\n'
 '          2300,  2007, 11928,  1012, 22712, 17395,  2098, 11928,  1005,  '
 '1055,\n'
 '          8103,  1012,   102],\n'
 '        [  101, 18403,  2435,  1037,  8549,  2000, 27970,  1005,  1055,  '
 '2365,\n'
 '          2043,  2027,  2020,  3110,  2091,  1012,   102,     0,     0,     '
```

```
 '0,\n'
 '            0,      0,      0]])\n')
('Attention Mask:\n'
 ' tensor([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, '
 '0],\n'
 '        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, '
 '0],\n'
 '        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, '
 '1],\n'
 '        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, '
 '0]])\n')
/usr/local/lib/python3.10/dist-packages/transformers/tokenization_utils_base.py:2692: UserWarning: `max_length` is ignored when `padding`=`True` and there is no truncation strategy. To
  warnings.warn(
```

Notice how 0s get appended to the input ids sequence, and the same is also reflected in the output of `attn_mask` where `0` indicates that the particular token was padded and `1` means otherwise. Setting `return_tensors="pt"` results in the outputs as torch tensors

Finally, for tasks involving reasoning over multiple sentences (like what we have for the SocialIQA dataset), it is common to seperate out each sentence using a [SEP] token:



(a) Sentence Pair Classification Tasks: MNLI, QQP, QNLI, STS-B, MRPC, RTE, SWAG

We can achieve this by adding concatenating all sentences with the `[SEP]` token before calling the tokenizer

```
example = train_data[100]
context = example["context"]
question = example["question"]
answerA = example["answerA"]

# Concatenate the context, question and answerA
cqa = context + bert_tokenizer.sep_token + question + bert_tokenizer.sep_token + answerA
print(cqa)

tokenized_cqa = bert_tokenizer(cqa)
pprint(tokenized_cqa, sort_dicts=False, indent=4)
```

```
Jordan's dog peed on the couch they were selling and Jordan removed the odor as soon as possible.[SEP]How would Jordan feel afterwards?[SEP]selling a couch
{   'input_ids': [   101,
                     5207,
                     1005,
                     1055,
                     3899,
                     21392,
                     2094,
                     2006,
                     1996,
                     6411,
                     2027,
                     2020,
                     4855,
                     1998,
                     5207,
                     3718,
                     1996,
                     19255,
                     2004,
                     2574,
                     2004,
                     2825,
                     1012,
                     102,
                     2129,
                     2052,
                     5207,
                     2514,
                     5728,
                     1029,
                     102,
                     4855,
                     1037,
                     6411,
                     102],
    'token_type_ids': [   0,
                          0,
                          0,
                          0,
                          0,
                          0,
                          0,
                          0,
                          0,
```

```
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
0,
```

For the reasons that will become clear once we work on the modeling part, we need three input tensors for each dataset example, one for concatenating each answer with the context and question.

```
example = train_data[100]
context = example["context"]
question = example["question"]
answerA = example["answerA"]

answerB = example["answerB"]
answerC = example["answerC"]

cqaA = context + bert_tokenizer.sep_token + question + bert_tokenizer.sep_token + answerA
cqaB = context + bert_tokenizer.sep_token + question + bert_tokenizer.sep_token + answerB
cqaC = context + bert_tokenizer.sep_token + question + bert_tokenizer.sep_token + answerC

print(cqaA)
print(cqaB)
print(cqaC)

tokenized_cqaA = bert_tokenizer(cqaA)
tokenized_cqaB = bert_tokenizer(cqaB)
tokenized_cqaC = bert_tokenizer(cqaC)
```

```
Jordan's dog peed on the couch they were selling and Jordan removed the odor as soon as possible.[SEP]How would Jordan feel afterwards?[SEP]selling a couch
Jordan's dog peed on the couch they were selling and Jordan removed the odor as soon as possible.[SEP]How would Jordan feel afterwards?[SEP]Disgusted
Jordan's dog peed on the couch they were selling and Jordan removed the odor as soon as possible.[SEP]How would Jordan feel afterwards?[SEP]Relieved
```

Task 1.1: Custom Dataset Class

Now that we know how to use the hugging face tokenizers we can define the custom `torch.utils.Dataset` class like we did in the previous assignments to process and store the data as well as provides a way to iterate through the dataset. Implement the `SIQABertDataset` class below. Recall to create a custom class you need to implement 3 methods `__init__`, `__len__` and `__getitem__`.

```
from torch.utils.data import Dataset, DataLoader

class SIQABertDataset(Dataset):

    def __init__(self, data, labels, bert_variant = "bert-base-uncased"):
        """
        Constructor for the `SST2BertDataset` class. Stores the `sentences` and `labels` which can then be used by
        other methods. Also initializes the tokenizer

        Inputs:
            - data (list) : A list SIQA dataset examples
            - labels (list): A list of labels corresponding to each example
            - bert_variant (str): A string indicating the variant of BERT to be used.
        """
        self.label2label_id = {"A": 0, "B": 1, "C": 2}
        self.data = None
        self.labels = None
        self.tokenizer = None

        # YOUR CODE HERE
        self.data = data
        self.labels = labels
        self.tokenizer = BertTokenizer.from_pretrained(bert_variant)

        if (not self.data) or (not self.labels) or (not self.tokenizer):
          raise NotImplementedError()

    def __len__(self):
        """
        Returns the length of the dataset
        """
        length = None

        # YOUR CODE HERE
        length = len(self.data)

        if length is None:
          raise NotImplementedError()

        return length

    def __getitem__(self, idx):
        """
        Returns the training example corresponding to review present at the `idx` position in the dataset

        Inputs:
            - idx (int): Index corresponding to the review,label to be returned

        Returns:
            - tokenized_input_dict (dict(str, dict)): A dictionary corresponding to tokenizer outputs for the three resulting sequences due to each answer choices as described above
            - label (int): Answer label for the corresponding sentence. We will use 0, 1 and 2 to represent A, B and C respectively.

        Example Output:
            - tokenized_input_dict: {
```

```
        "A": {'input_ids': [101, 5207, 1005, 1055, 3899, 21392, 2094, 2006, 1996, 6411, 2027, 2020, 4855, 1998, 5207, 3718, 1996, 19255, 2004, 2574, 2004, 2825, 1012, 102, 2129, 205
        "B": {'input_ids': [101, 5207, 1005, 1055, 3899, 21392, 2094, 2006, 1996, 6411, 2027, 2020, 4855, 1998, 5207, 3718, 1996, 19255, 2004, 2574, 2004, 2825, 1012, 102, 2129, 205
        "C": {'input_ids': [101, 5207, 1005, 1055, 3899, 21392, 2094, 2006, 1996, 6411, 2027, 2020, 4855, 1998, 5207, 3718, 1996, 19255, 2004, 2574, 2004, 2825, 1012, 102, 2129, 205
    }
    - label: 0

"""

tokenized_input_dict = {"A": None, "B": None, "C": None}
label = None

# YOUR CODE HERE
example = self.data[idx]
context = example["context"]
question = example["question"]
answerA = example["answerA"]
answerB = example["answerB"]
answerC = example["answerC"]

cqaA = context + self.tokenizer.sep_token + question + self.tokenizer.sep_token + answerA
cqaB = context + self.tokenizer.sep_token + question + self.tokenizer.sep_token + answerB
cqaC = context + self.tokenizer.sep_token + question + self.tokenizer.sep_token + answerC

tokenized_input_dict["A"] = self.tokenizer(cqaA)
tokenized_input_dict["B"] = self.tokenizer(cqaB)
tokenized_input_dict["C"] = self.tokenizer(cqaC)

label = self.label2label_id[self.labels[idx]]

if label is None:
  raise NotImplementedError()

return tokenized_input_dict, label
```

```
print("Running Sample Test Cases")

sample_dataset = SIQABertDataset(train_data[:2], train_labels[:2], bert_variant="bert-base-uncased")

print(f"Sample Test Case 1: Checking if `__len__` is implemented correctly")
dataset_len= len(sample_dataset)
expected_len = 2
print(f"Dataset Length: {dataset_len}")
print(f"Expected Length: {expected_len}")
assert len(sample_dataset) == expected_len
print("Sample Test Case Passed!")
print("****************************************\n")

print(f"Sample Test Case 2: Checking if `__getitem__` is implemented correctly for `idx= 0`")
sample_idx = 0
tokenized_input_dict, label = sample_dataset.__getitem__(sample_idx)
expected_tokenized_input_dict = {'A': {'input_ids': [101, 7232, 2787, 2000, 2031, 1037, 26375, 1998, 5935, 2014, 2814, 2362, 1012, 102, 2129, 2052, 2500, 2514, 2004, 1037, 2765, 1029, 102,
  'B': {'input_ids': [101, 7232, 2787, 2000, 2031, 1037, 26375, 1998, 5935, 2014, 2814, 2362, 1012, 102, 2129, 2052, 2500, 2514, 2004, 1037, 2765, 1029, 102, 2066, 6595, 2188, 102], 'token_
  'C': {'input_ids': [101, 7232, 2787, 2000, 2031, 1037, 26375, 1998, 5935, 2014, 2814, 2362, 1012, 102, 2129, 2052, 2500, 2514, 2004, 1037, 2765, 1029, 102, 1037, 2204, 2767, 2000, 2031, 1
expected_label = 0
print(f"tokenized_input_dict:\n {tokenized_input_dict}")
print(f"Expected tokenized_input_dict:\n {expected_tokenized_input_dict}")
assert (expected_tokenized_input_dict == tokenized_input_dict)

print(f"label:\n {label}")
print(f"Expected label:\n {expected_label}")
assert expected_label == label

print("Sample Test Case Passed!")
print("****************************************\n")


print(f"Sample Test Case 3: Checking if `__getitem__` is implemented correctly for `idx= 1`")
sample_idx = 1
tokenized_input_dict, label = sample_dataset.__getitem__(sample_idx)
expected_tokenized_input_dict =  {'A': {'input_ids': [101, 5553, 2734, 2000, 2507, 2041, 5841, 2005, 2019, 9046, 2622, 2012, 2147, 1012, 102, 2054, 2097, 2500, 2215, 2000, 2079, 2279, 1029,
                                  'B': {'input_ids': [101, 5553, 2734, 2000, 2507, 2041, 5841, 2005, 2019, 9046, 2622, 2012, 2147, 1012, 102, 2054, 2097, 2500, 2215, 2000, 2079, 2279, 1029,
                                'C': {'input_ids': [101, 5553, 2734, 2000, 2507, 2041, 5841, 2005, 2019, 9046, 2622, 2012, 2147, 1012, 102, 2054, 2097, 2500, 2215, 2000, 2079, 2279, 1029, 1


expected_label = 1
print(f"tokenized_input_dict:\n {tokenized_input_dict}")
print(f"Expected tokenized_input_dict:\n {expected_tokenized_input_dict}")
assert (expected_tokenized_input_dict == tokenized_input_dict)

print(f"label:\n {label}")
print(f"Expected label:\n {expected_label}")
assert expected_label == label

print("Sample Test Case Passed!")
print("****************************************\n")

print(f"Sample Test Case 4: Checking if `__getitem__` is implemented correctly for `idx= 0` for a different bert-variant")
sample_dataset = SIQABertDataset(train_data[:2], train_labels[:2], bert_variant="bert-base-cased")
```

```
sample_idx = 0
tokenized_input_dict, label = sample_dataset.__getitem__(sample_idx)
expected_tokenized_input_dict = {'A': {'input_ids': [101, 6681, 1879, 1106, 1138, 170, 2927, 3962, 27138, 1105, 5260, 1123, 2053, 1487, 119, 102, 1731, 1156, 8452, 1631, 1112, 170, 1871, 13
 'B': {'input_ids': [101, 6681, 1879, 1106, 1138, 170, 2927, 3962, 27138, 1105, 5260, 1123, 2053, 1487, 119, 102, 1731, 1156, 8452, 1631, 1112, 170, 1871, 136, 102, 1176, 6218, 1313, 102],
 'C': {'input_ids': [101, 6681, 1879, 1106, 1138, 170, 2927, 3962, 27138, 1105, 5260, 1123, 2053, 1487, 119, 102, 1731, 1156, 8452, 1631, 1112, 170, 1871, 136, 102, 170, 1363, 1910, 1106, 1
expected_label = 0
print(f"tokenized_input_dict:\n {tokenized_input_dict}")
print(f"Expected tokenized_input_dict:\n {expected_tokenized_input_dict}")
assert (expected_tokenized_input_dict == tokenized_input_dict)

print(f"label:\n {label}")
print(f"Expected label:\n {expected_label}")
assert expected_label == label

print("Sample Test Case Passed!")
print("****************************************\n")
```

```
Running Sample Test Cases
Sample Test Case 1: Checking if `__len__` is implemented correctly
Dataset Length: 2
Expected Length: 2
Sample Test Case Passed!
****************************************

Sample Test Case 2: Checking if `__getitem__` is implemented correctly for `idx= 0`
tokenized_input_dict:
 {'A': {'input_ids': [101, 7232, 2787, 2000, 2031, 1037, 26375, 1998, 5935, 2014, 2814, 2362, 1012, 102, 2129, 2052, 2500, 2514, 2004, 1037, 2765, 1029, 102, 2066, 7052, 102], 'token_ty
Expected tokenized_input_dict:
 {'A': {'input_ids': [101, 7232, 2787, 2000, 2031, 1037, 26375, 1998, 5935, 2014, 2814, 2362, 1012, 102, 2129, 2052, 2500, 2514, 2004, 1037, 2765, 1029, 102, 2066, 7052, 102], 'token_ty
label:
 0
Expected label:
 0
Sample Test Case Passed!
****************************************

Sample Test Case 3: Checking if `__getitem__` is implemented correctly for `idx= 1`
tokenized_input_dict:
 {'A': {'input_ids': [101, 5553, 2734, 2000, 2507, 2041, 5841, 2005, 2019, 9046, 2622, 2012, 2147, 1012, 102, 2054, 2097, 2500, 2215, 2000, 2079, 2279, 1029, 102, 21090, 2007, 5553, 102
Expected tokenized_input_dict:
 {'A': {'input_ids': [101, 5553, 2734, 2000, 2507, 2041, 5841, 2005, 2019, 9046, 2622, 2012, 2147, 1012, 102, 2054, 2097, 2500, 2215, 2000, 2079, 2279, 1029, 102, 21090, 2007, 5553, 102
label:
 1
Expected label:
 1
Sample Test Case Passed!
****************************************

Sample Test Case 4: Checking if `__getitem__` is implemented correctly for `idx= 0` for a different bert-variant
tokenizer_config.json: 100%    49.0/49.0 [00:00<00:00, 4.06kB/s]
vocab.txt: 100%    213k/213k [00:00<00:00, 3.66MB/s]
tokenizer.json: 100%    436k/436k [00:00<00:00, 2.56MB/s]
config.json: 100%    570/570 [00:00<00:00, 30.8kB/s]
tokenized_input_dict:
 {'A': {'input_ids': [101, 6681, 1879, 1106, 1138, 170, 2927, 3962, 27138, 1105, 5260, 1123, 2053, 1487, 119, 102, 1731, 1156, 8452, 1631, 1112, 170, 1871, 136, 102, 1176, 6546, 102], '
Expected tokenized_input_dict:
 {'A': {'input_ids': [101, 6681, 1879, 1106, 1138, 170, 2927, 3962, 27138, 1105, 5260, 1123, 2053, 1487, 119, 102, 1731, 1156, 8452, 1631, 1112, 170, 1871, 136, 102, 1176, 6546, 102], '
label:
 0
Expected label:
 0
Sample Test Case Passed!
****************************************
```

We can now create Dataset instances for both training and dev datasets

```
train_dataset = SIQABertDataset(train_data, train_labels, bert_variant="bert-base-uncased")
```

Before we instantiate the dataloaders for iterating over the dataset like last time, we need define a collate function, that creates batches from a list of dataset examples. In the last class we didn't have to create one, because all of our examples were of the same size, but that's not the case anymore, and we need to pad the sequences so that they all are of same size. We have implemented the collate_fn for you below, but we recommend going through it step by step, as it is used often in practice.

```
def collate_fn(tokenizer, batch):
    """
    Collate function to be used when creating a data loader for the SIQA dataset.
    :param tokenizer: The tokenizer to be used to tokenize the inputs.
    :param batch: A list of tuples of the form (tokenized_input_dict, label)
    :return: A tuple of the form (tokenized_inputs_dict_batch, labels_batch)
    """

    tokenized_inputsA_batch = []
    tokenized_inputsB_batch = []
    tokenized_inputsC_batch = []
    labels_batch = []
    for tokenized_inputs_dict, label in batch:
        tokenized_inputsA_batch.append(tokenized_inputs_dict["A"])
        tokenized_inputsB_batch.append(tokenized_inputs_dict["B"])
        tokenized_inputsC_batch.append(tokenized_inputs_dict["C"])
        labels_batch.append(label)

    #Pad the inputs
    tokenized_inputsA_batch = tokenizer.pad(tokenized_inputsA_batch, padding=True, return_tensors="pt")
    tokenized_inputsB_batch = tokenizer.pad(tokenized_inputsB_batch, padding=True, return_tensors="pt")
    tokenized_inputsC_batch = tokenizer.pad(tokenized_inputsC_batch, padding=True, return_tensors="pt")

    # Convert labels list to a tensor
    labels_batch = torch.tensor(labels_batch)
    return (
        {"A": tokenized_inputsA_batch["input_ids"], "B": tokenized_inputsB_batch["input_ids"], "C": tokenized_inputsC_batch["input_ids"]},
        {"A": tokenized_inputsA_batch["attention_mask"], "B": tokenized_inputsB_batch["attention_mask"], "C": tokenized_inputsC_batch["attentio
        labels_batch
    )
```