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

Due: May 26, 2024

Welcome to the Assignment 2 of our course on Natural Language Processing. Similar to Lab 2, we will again be working on a common sense reasoning task and fine-tuning BERT to solve the same. Specifically, we will be looking at Choice Of Plausible Alternatives (COPA) dataset which was created to access common-sense causal reasoning of NLP models. This assignment should flow naturally from Lab 2, and we shall see with minimal changes we will be able to adapt what we learned for SocialIQA task on COPA.

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')
copa_data_dir = "gdrive/MyDrive/PlakshaTLF24-NLP/Assignment02/copa/"
```

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

```
# Install required libraries
# If using Colab, DO NOT INSTALL ANYTHING!
# !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
import xml.etree.ElementTree as ET
from pprint import pprint
```

```python
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
```

COPA Dataset

We start by discussing the dataset that we will making use of in today's Assignment. As described above, the COPA evaluation provides researchers with a tool for assessing progress in open-domain commonsense causal reasoning. COPA consists of 1000 questions, split equally into development and test sets of 500 questions each. Each question is composed of a premise and two alternatives, where the task is to select the alternative that more plausibly has a causal relation with the premise. Some examples from the dataset include:

```
Premise: The man broke his toe. What was the CAUSE of this?
Alternative 1: He got a hole in his sock.
Alternative 2: He dropped a hammer on his foot.

Premise: I tipped the bottle. What happened as a RESULT?
Alternative 1: The liquid in the bottle froze.
Alternative 2: The liquid in the bottle poured out.

Premise: I knocked on my neighbor's door. What happened as a RESULT?
Alternative 1: My neighbor invited me in.
Alternative 2: My neighbor left his house.
```

Below we load the dataset in memory. Since there is no seperate training set, we use dev set for training the model and evaluate on test set.

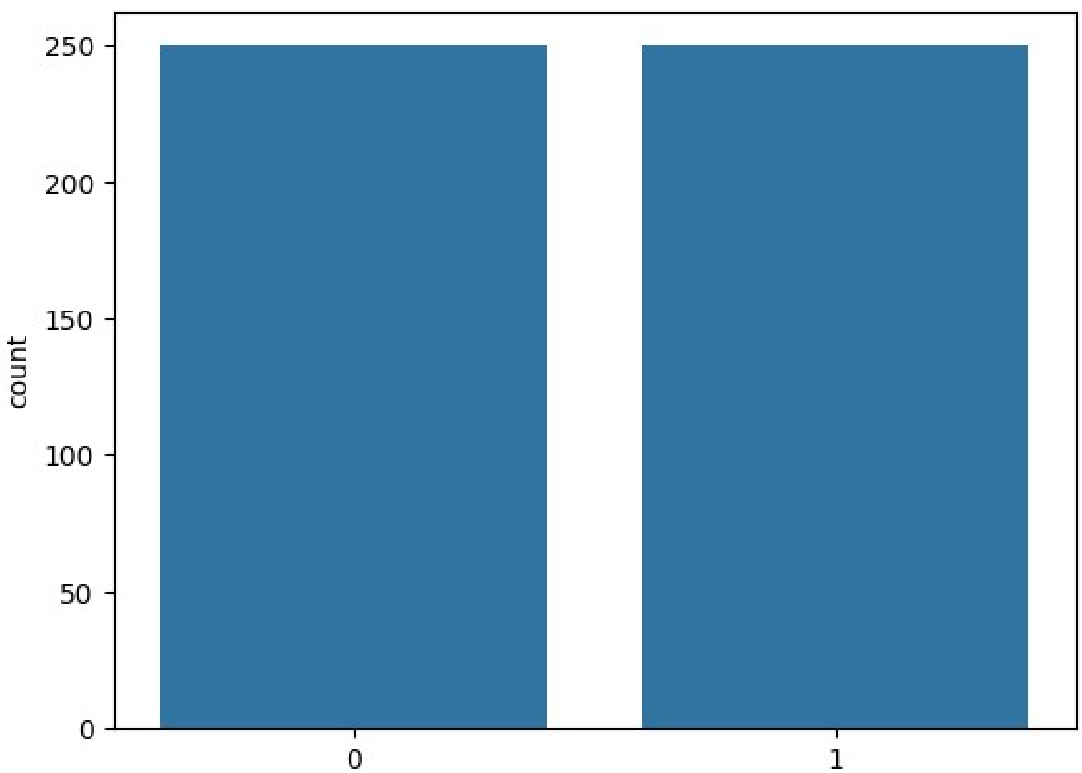
```python
def parse_copa_dataset(split="test"):
    tree = ET.parse(f"{copa_data_dir}/copa-{split}.xml")
    root = tree.getroot()
    items = root.findall("item")

    data = []
    labels = []
    for item in items:
        data.append(
            {
```

```
                "question": item.get("asks-for"),
                "premise": item.find("p").text,
                "choice1": item.find("a1").text,
                "choice2": item.find("a2").text,
            }
        )
        labels.append(int(item.get("most-plausible-alternative")) - 1)

    return data, labels

train_data, train_labels = parse_copa_dataset("dev")
test_data, test_labels = parse_copa_dataset("test")

print(f"Number of Training Examples: {len(train_data)}")
print(f"Number of Test Examples: {len(test_data)}")
```

```
Number of Training Examples: 500
Number of Test Examples: 500
```

```
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
{   'question': 'effect',
    'premise': 'The teacher took roll.',
    'choice1': 'She identified the students that were absent.',
    'choice2': 'She gave her students a pop quiz.'}
Label: 0
```

As you can see, the dataset is pretty much very similar as SocialIQA, with the main difference being that we have two answer choices instead of three. Hence, we just need to concatenate choice1 and choice2, seperately with premise and question this time.

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

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

example = train_data[100]
premise = example["premise"]
question = example["question"]
choice1 = example["choice1"]
choice2 = example["choice2"]

pqc1 = premise + bert_tokenizer.sep_token + question +
bert_tokenizer.sep_token + choice1
pqc2 = premise + bert_tokenizer.sep_token + question +
bert_tokenizer.sep_token + choice2

print(pqc1)
print(pqc2)

tokenized_pqc1 = bert_tokenizer(pqc1)
tokenized_pqc2 = bert_tokenizer(pqc2)
```

```
/usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/
_token.py:89: 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(
```

{"model_id":"836d9d90fd1f4f7da91792e271f5f7d3","version_major":2,"version_minor":0}

{"model_id":"a435f903868641b9af02f3885ecceb3b","version_major":2,"version_minor":0}

{"model_id":"9dceba19c9824bbcaf7b5e4626d954d9","version_major":2,"version_minor":0}

```
/usr/local/lib/python3.10/dist-packages/huggingface_hub/
file_download.py:1132: FutureWarning: `resume_download` is deprecated
and will be removed in version 1.0.0. Downloads always resume when
possible. If you want to force a new download, use
`force_download=True`.
  warnings.warn(
```

{"model_id":"6202a323186f4f0ebb255f8de99b8866","version_major":2,"version_minor":0}

```
The teacher took roll.[SEP]effect[SEP]She identified the students that
were absent.
The teacher took roll.[SEP]effect[SEP]She gave her students a pop
quiz.
```

Task 1: Setting up Custom Datasets and Dataloaders (4 Marks)

Task 1.1: Custom Dataset Class (2 Marks)

Similar to Lab 2, you will start by implementing a custom Dataset class for COPA dataset. The only difference will be that `__getitem__` should return tokenized outputs corresponding to the two choices choice1 and choice2, instead of three like in the case of SocialIQA dataset.

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

class COPABertDataset(Dataset):

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

        Inputs:
            - data (list) : A list COPA dataset examples
            - labels (list): A list of answer labels corresponding to
each example
            - bert_variant (str): A string indicating the variant of
BERT to be used.
        """
```

```
        self.data = None
        self.labels = None
        self.tokenizer = None

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


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

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

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

        Inputs:
            - idx (int): Index corresponding to the dataset example to
be returned

        Returns:
            - tokenized_input_dict (dict(str, dict)): A dictionary
corresponding to tokenizer outputs for the two resulting sequences due
to each answer choices as described above
            - label (int): Answer label for the corresponding
sentence. 0 for first answer choice and 1 for second answer choice.

        Example Output:
            - tokenized_input_dict: {
                "choice1": {'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, 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, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1]},
                "choice2": {'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, 17733, 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, 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, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]},
            }
            - label: 0

        """

        tokenized_input_dict = {"choice1": None, "choice2": None}
        label = None

        # YOUR CODE HERE
        example = self.data[idx]
        premise = example["premise"]
        question = example["question"]
        choice1 = example["choice1"]
        choice2 = example["choice2"]

        pqc1 = premise + self.tokenizer.sep_token + question +
self.tokenizer.sep_token + choice1
        pqc2 = premise + self.tokenizer.sep_token + question +
self.tokenizer.sep_token + choice2

        tokenized_input_dict["choice1"] = self.tokenizer(pqc1)
        tokenized_input_dict["choice2"] = self.tokenizer(pqc2)

        label = self.label[idx]

        return tokenized_input_dict, label

print("Running Sample Test Cases")

sample_dataset = COPABertDataset(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 = {'choice1': {'input_ids': [101, 2026,
```

```
2303, 3459, 1037, 5192, 2058, 1996, 5568, 1012, 102, 3426, 102, 1996,
3103, 2001, 4803, 1012, 102], 'token_type_ids': [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, 1, 1, 1]},
 'choice2': {'input_ids': [101, 2026, 2303, 3459, 1037, 5192, 2058,
1996, 5568, 1012, 102, 3426, 102, 1996, 5568, 2001, 3013, 1012, 102],
'token_type_ids': [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, 1, 1, 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 = {'choice1': {'input_ids': [101, 1996,
2450, 25775, 2014, 2767, 1005, 1055, 3697, 5248, 1012, 102, 3426, 102,
1996, 2450, 2354, 2014, 2767, 2001, 2183, 2083, 1037, 2524, 2051,
1012, 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, 0, 0, 0, 0, 0], 'attention_mask': [1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1]},
 'choice2': {'input_ids': [101, 1996, 2450, 25775, 2014, 2767, 1005,
1055, 3697, 5248, 1012, 102, 3426, 102, 1996, 2450, 2371, 2008, 2014,
2767, 2165, 5056, 1997, 2014, 16056, 1012, 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, 0, 0,
0, 0, 0], 'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 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 4: Checking if `__getitem__` is implemented correctly for `idx= 0` for a different bert-variant")
sample_dataset = COPABertDataset(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 = {'choice1': {'input_ids': [101, 1422, 1404, 2641, 170, 6464, 1166, 1103, 5282, 119, 102, 2612, 102, 1109, 3336, 1108, 4703, 119, 102], 'token_type_ids': [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, 1, 1, 1]},
 'choice2': {'input_ids': [101, 1422, 1404, 2641, 170, 6464, 1166, 1103, 5282, 119, 102, 2612, 102, 1109, 5282, 1108, 2195, 119, 102], 'token_type_ids': [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, 1, 1, 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:
 {'choice1': {'input_ids': [101, 2026, 2303, 3459, 1037, 5192, 2058, 1996, 5568, 1012, 102, 3426, 102, 1996, 3103, 2001, 4803, 1012, 102], 'token_type_ids': [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, 1, 1, 1]}, 'choice2': {'input_ids': [101, 2026, 2303, 3459, 1037, 5192, 2058, 1996, 5568, 1012, 102, 3426, 102, 1996, 5568, 2001, 3013, 1012, 102], 'token_type_ids': [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, 1, 1, 1]}}
Expected tokenized_input_dict:
 {'choice1': {'input_ids': [101, 2026, 2303, 3459, 1037, 5192, 2058,
1996, 5568, 1012, 102, 3426, 102, 1996, 3103, 2001, 4803, 1012, 102],
'token_type_ids': [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, 1, 1, 1]}, 'choice2': {'input_ids': [101, 2026, 2303, 3459, 1037,
5192, 2058, 1996, 5568, 1012, 102, 3426, 102, 1996, 5568, 2001, 3013,
1012, 102], 'token_type_ids': [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, 1, 1, 1]}}
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:
 {'choice1': {'input_ids': [101, 1996, 2450, 25775, 2014, 2767, 1005,
1055, 3697, 5248, 1012, 102, 3426, 102, 1996, 2450, 2354, 2014, 2767,
2001, 2183, 2083, 1037, 2524, 2051, 1012, 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, 0, 0,
0, 0, 0], 'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]}, 'choice2': {'input_ids':
[101, 1996, 2450, 25775, 2014, 2767, 1005, 1055, 3697, 5248, 1012,
102, 3426, 102, 1996, 2450, 2371, 2008, 2014, 2767, 2165, 5056, 1997,
2014, 16056, 1012, 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, 0, 0, 0, 0, 0],
'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1]}}
Expected tokenized_input_dict:
 {'choice1': {'input_ids': [101, 1996, 2450, 25775, 2014, 2767, 1005,
1055, 3697, 5248, 1012, 102, 3426, 102, 1996, 2450, 2354, 2014, 2767,
2001, 2183, 2083, 1037, 2524, 2051, 1012, 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, 0, 0,
0, 0, 0], 'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]}, 'choice2': {'input_ids':
[101, 1996, 2450, 25775, 2014, 2767, 1005, 1055, 3697, 5248, 1012,
102, 3426, 102, 1996, 2450, 2371, 2008, 2014, 2767, 2165, 5056, 1997,
2014, 16056, 1012, 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, 0, 0, 0, 0, 0],
'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1]}}
label:
 0
```

```
Expected label:
 0
Sample Test Case Passed!
****************************************

Sample Test Case 4: Checking if `__getitem__` is implemented correctly
for `idx= 0` for a different bert-variant
```

{"model_id":"903a6838d3f74b57b845e584c080da8d","version_major":2,"version_minor":0}

{"model_id":"b9978e4c28c6448499808c83af3878cb","version_major":2,"version_minor":0}

{"model_id":"2fd37eaf7c9f46809811e0a581037756","version_major":2,"version_minor":0}

{"model_id":"d6238f1f656543b88d0c0ae5da0222f9","version_major":2,"version_minor":0}

```
tokenized_input_dict:
 {'choice1': {'input_ids': [101, 1422, 1404, 2641, 170, 6464, 1166,
1103, 5282, 119, 102, 2612, 102, 1109, 3336, 1108, 4703, 119, 102],
'token_type_ids': [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, 1, 1, 1]}, 'choice2': {'input_ids': [101, 1422, 1404, 2641, 170,
6464, 1166, 1103, 5282, 119, 102, 2612, 102, 1109, 5282, 1108, 2195,
119, 102], 'token_type_ids': [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, 1, 1, 1]}}
Expected tokenized_input_dict:
 {'choice1': {'input_ids': [101, 1422, 1404, 2641, 170, 6464, 1166,
1103, 5282, 119, 102, 2612, 102, 1109, 3336, 1108, 4703, 119, 102],
'token_type_ids': [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, 1, 1, 1]}, 'choice2': {'input_ids': [101, 1422, 1404, 2641, 170,
6464, 1166, 1103, 5282, 119, 102, 2612, 102, 1109, 5282, 1108, 2195,
119, 102], 'token_type_ids': [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, 1, 1, 1]}}
label:
 0
Expected label:
 0
Sample Test Case Passed!
****************************************
```

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

```
train_dataset = COPABertDataset(train_data, train_labels,
bert_variant="bert-base-uncased")
test_dataset = COPABertDataset(test_data, test_labels,
bert_variant="bert-base-uncased")
```

Task 1.2: Custom `collate_fn` Class (2 Marks)

Similar to Lab 2, you will now implement a custom `colate_fn` for the `COPABertDataset` class. Remember, a `colate_fn` informs a dataloader on how to construct batches from a list of sequences in a batch. Implement `collate_fn` that takes a batch which is a list of tuples of the form `(tokenized_input_dict, label)` and constructs the batches of following:

- `input_ids_dict`: A dictionary containing batch of `input_ids` tensors corresponding to both choice1 and choice2. You will need to do padding here,
- `attn_mask_dict`: A dictionary containing batch of `attention_mask` tensors corresponding to both choice1 and choice2. You will need to do padding here,
- `labels`: A tensor of shape [batch_size,] containing the labels for each example in the batch

```
def collate_fn(tokenizer, batch):
    """
    Collate function to be used when creating a data loader for the
COPA 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 (input_ids_dict, attn_mask_dict, labels)
as described above
    """

    colated_batch = (None, None, None)

    # YOUR CODE HERE

    # defining the lists to store the input_ids, attention masks and
labels
    choice1_inputs = []
    choice2_inputs = []
    labels = []

    # iterating over the batch and appending the input_ids, attention
masks and labels to the respective lists
    for tokenized_input_dict, label in batch:
        choice1_inputs.append(tokenized_input_dict["choice1"])
        choice2_inputs.append(tokenized_input_dict["choice2"])
        labels.append(label)

    # converting the lists to tensors, with padding
```

```
    choice1_inputs = tokenizer.pad(choice1_inputs, padding=True,
return_tensors="pt")
    choice2_inputs = tokenizer.pad(choice2_inputs, padding=True,
return_tensors="pt")
    labels = torch.tensor(labels)

    input_ids_dict = {"choice1": choice1_inputs["input_ids"],
"choice2": choice2_inputs["input_ids"]}
    attn_mask_dict = {"choice1": choice1_inputs["attention_mask"],
"choice2": choice2_inputs["attention_mask"]}

    # creating the colated triple
    colated_batch = (input_ids_dict, attn_mask_dict, labels)

    return colated_batch

print("Running Sample Test Cases")

sample_dataset = COPABertDataset(train_data[:2], train_labels[:2],
bert_variant="bert-base-uncased")
batch = [sample_dataset.__getitem__(0), sample_dataset.__getitem__(1)]


print(f"Sample Test Case 1: Checking if the return output of
`collate_fn` is of the correct type")
colated_batch = collate_fn(bert_tokenizer, batch)
print(f"Output type: {type(colated_batch)}")
assert (type(colated_batch) == tuple)
print(f"Tuple Length: {len(colated_batch)}")
assert (len(colated_batch) == 3)
print(f"Tuple 0th element type: {type(colated_batch[0])}")
assert (type(colated_batch[0]) == dict)
print(f"Tuple 1st element type: {type(colated_batch[1])}")
assert (type(colated_batch[1]) == dict)
print(f"Tuple 2nd element type: {type(colated_batch[2])}")
assert (type(colated_batch[2]) == torch.Tensor)
print("Sample Test Case Passed!")
print("****************************************\n")

print(f"Sample Test Case 2: Checking if the return output of
`collate_fn` is of the correct shape")
print(f"Tuple 0th element shape for choice1: {colated_batch[0]
['choice1'].shape}")
assert (colated_batch[0]['choice1'].shape == torch.Size([2, 27]))
print(f"Tuple 0th element shape for choice2: {colated_batch[0]
['choice2'].shape}")
assert (colated_batch[0]['choice2'].shape == torch.Size([2, 27]))

print(f"Tuple 1st element shape for choice1: {colated_batch[1]
['choice1'].shape}")
```

```python
assert (colated_batch[1]['choice1'].shape == torch.Size([2, 27]))
print(f"Tuple 1st element shape for choice2: {colated_batch[1]
['choice2'].shape}")
assert (colated_batch[1]['choice2'].shape == torch.Size([2, 27]))

print(f"Tuple 2nd element shape: {colated_batch[2].shape}")
assert (colated_batch[2].shape == torch.Size([2]))

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

print(f"Sample Test Case 3: Checking if the return output of
`collate_fn` is of the correct values")
tup0_choice1_expected = torch.tensor([[  101,  2026,  2303,  3459,
1037,  5192,  2058,  1996,  5568,  1012,
           102,  3426,   102,  1996,  3103,  2001,  4803,  1012,
102,     0,
             0,     0,     0,     0,     0,     0,     0],
        [  101,  1996,  2450, 25775,  2014,  2767,  1005,  1055,
3697,  5248,
          1012,   102,  3426,   102,  1996,  2450,  2354,  2014,
2767,  2001,
          2183,  2083,  1037,  2524,  2051,  1012,   102]])
print(f"Tuple 0th element predicted values for choice1:
{colated_batch[0]['choice1']}")
print(f"Tuple 0th element expected values for choice1:
{tup0_choice1_expected}")
assert (torch.allclose(colated_batch[0]['choice1'],
tup0_choice1_expected))
```

```
Running Sample Test Cases
Sample Test Case 1: Checking if the return output of `collate_fn` is
of the correct type
Output type: <class 'tuple'>
Tuple Length: 3
Tuple 0th element type: <class 'dict'>
Tuple 1st element type: <class 'dict'>
Tuple 2nd element type: <class 'torch.Tensor'>
Sample Test Case Passed!
****************************************

Sample Test Case 2: Checking if the return output of `collate_fn` is
of the correct shape
Tuple 0th element shape for choice1: torch.Size([2, 27])
Tuple 0th element shape for choice2: torch.Size([2, 27])
Tuple 1st element shape for choice1: torch.Size([2, 27])
Tuple 1st element shape for choice2: torch.Size([2, 27])
Tuple 2nd element shape: torch.Size([2])
Sample Test Case Passed!
****************************************
```

```
Sample Test Case 3: Checking if the return output of `collate_fn` is
of the correct values
Tuple 0th element predicted values for choice1: tensor([[  101,  2026,
2303,  3459,  1037,  5192,  2058,  1996,  5568,  1012,
           102,  3426,   102,  1996,  3103,  2001,  4803,  1012,
102,     0,
             0,     0,     0,     0,     0,     0,     0],
        [  101,  1996,  2450, 25775,  2014,  2767,  1005,  1055,
3697,  5248,
          1012,   102,  3426,   102,  1996,  2450,  2354,  2014,
2767,  2001,
          2183,  2083,  1037,  2524,  2051,  1012,   102]])
Tuple 0th element expected values for choice1: tensor([[  101,  2026,
2303,  3459,  1037,  5192,  2058,  1996,  5568,  1012,
           102,  3426,   102,  1996,  3103,  2001,  4803,  1012,
102,     0,
             0,     0,     0,     0,     0,     0,     0],
        [  101,  1996,  2450, 25775,  2014,  2767,  1005,  1055,
3697,  5248,
          1012,   102,  3426,   102,  1996,  2450,  2354,  2014,
2767,  2001,
          2183,  2083,  1037,  2524,  2051,  1012,   102]])
```

Now that we have defined the collate_fn, lets create the dataloaders. It is common to use smaller batch size while fine-tuning these big models, as they occupy quite a lot of memory.

```
batch_size = 16
train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True, collate_fn=partial(collate_fn, train_dataset.tokenizer))
test_loader = DataLoader(test_dataset, batch_size=batch_size,
shuffle=True, collate_fn=partial(collate_fn, test_dataset.tokenizer))

batch_input_ids, batch_attn_mask, batch_labels =
next(iter(train_loader))
print(f"batch_input_ids:\n {batch_input_ids}")
print(f"batch_attn_mask:\n {batch_attn_mask}")
print(f"batch_labels:\n {batch_labels}")

batch_input_ids:
 {'choice1': tensor([[  101,  1996,  2450, 21319,  1996,  2336,  2013,
2014,  3200,  1012,
           102,  3426,   102,  1996,  2336,  2718,  1037,  3608,
2046,  2014,
          4220,  1012,   102,     0],
        [  101,  1996,  2450, 13671,  2014,  6904, 18796,  2102,
1012,   102,
          3426,   102,  1996,  6904, 18796,  2102,  2001, 17271,
2100,  1012,
```

```
          102,     0,     0,     0],
       [  101,  1996,  2879,  2018,  4390,  6462,  2075,  2010,
3797,  1012,
          102,  3466,   102,  2002,  4188,  2000,  4929,  1996,
3797,  1012,
          102,     0,     0,     0],
       [  101,  1996,  3237,  2787,  2025,  2000, 10887,  1996,
23761,  1012,
          102,  3426,   102,  1996, 23761,  3478,  1037,  4281,
4638,  1012,
          102,     0,     0,     0],
       [  101,  1996,  2450,  4907, 11533,  1996, 20377, 28168,
2682,  1996,
         3899,  1012,   102,  3466,   102,  1996,  3899,  5598,
2039,  1012,
          102,     0,     0,     0],
       [  101,  1996,  6302,  4042,  5067,  6497,  1999,  1996,
14832,  1012,
          102,  3426,   102,  2027,  2815, 22154,  2005,  2086,
1012,   102,
            0,     0,     0,     0],
       [  101,  1045,  2363,  1037,  7427,  1999,  1996,  5653,
1012,   102,
         3466,   102,  1996,  7427, 13330,  2026, 10628,  1012,
102,     0,
            0,     0,     0,     0],
       [  101,  1996,  6462,  2006,  2026,  3797,  3062,  2125,
1012,   102,
         3466,   102,  1045,  7367, 15557,  1996,  6462,  2067,
2006,  1012,
          102,     0,     0,     0],
       [  101,  2576,  4808,  3631,  2041,  1999,  1996,  3842,
1012,   102,
         3466,   102,  2116,  4480,  7448,  2000,  1996,  9424,
1012,   102,
            0,     0,     0,     0],
       [  101,  1996,  2450,  2001, 11908,  2005,  6467,  4611,
1012,   102,
         3466,   102,  2016, 11925,  2014,  5160,  1012,   102,
0,     0,
            0,     0,     0,     0],
       [  101,  1996,  8044,  2246,  2601,  1012,   102,  3466,
102,  1045,
         2716,  2026, 12191,  2000,  2147,  1012,   102,     0,
0,     0,
            0,     0,     0,     0],
       [  101,  1996,  2158,  1005,  1055,  4253,  4906, 11853,
1012,   102,
         3426,   102,  2002,  4149,  2068,  2006,  5096,  1012,
```

```
102,     0,
             0,     0,     0,     0],
        [  101,  1996,  8645,  3603,  1996,  4099,  1005,  1055,
3295,  1012,
           102,  3426,   102,  1996,  8645, 11829,  5999,  1996,
4099,  1005,
          1055,  3042,  1012,   102],
        [  101,  1996,  7596,  5520,  2039,  2046,  1996,  3712,
1012,   102,
          3426,   102,  1996,  2611,  3390,  2009,  1012,   102,
0,     0,
             0,     0,     0,     0],
        [  101,  1996,  2482,  2246, 18294,  1012,   102,  3466,
102,  1996,
          3954,  2165,  2009,  2000,  1996,  2482,  9378,  1012,
102,     0,
             0,     0,     0,     0],
        [  101,  1996,  2450,  2018,  2019,  8985,  1012,   102,
3466,   102,
          2016,  2165, 24479,  1012,   102,     0,     0,     0,
0,     0,
             0,     0,     0,     0]]), 'choice2': tensor([[  101,
1996,  2450, 21319,  1996,  2336,  2013,  2014,  3200,  1012,
           102,  3426,   102,  1996,  2336, 12517, 21132,  2083,
2014,  3871,
          1012,   102,     0,     0],
        [  101,  1996,  2450, 13671,  2014,  6904, 18796,  2102,
1012,   102,
          3426,   102,  1996,  6904, 18796,  2102,  2001,  2357,
2125,  1012,
           102,     0,     0,     0],
        [  101,  1996,  2879,  2018,  4390,  6462,  2075,  2010,
3797,  1012,
           102,  3466,   102,  2002,  2356,  2010,  2388,  2005,
2393,  1012,
           102,     0,     0,     0],
        [  101,  1996,  3237,  2787,  2025,  2000, 10887,  1996,
23761,  1012,
           102,  3426,   102,  1996, 23761,  2018,  3325,  2005,
1996,  3105,
          1012,   102,     0,     0],
        [  101,  1996,  2450,  4907, 11533,  1996, 20377, 28168,
2682,  1996,
          3899,  1012,   102,  3466,   102,  1996,  3899, 15047,
2049,  6519,
          1012,   102,     0,     0],
        [  101,  1996,  6302,  4042,  5067,  6497,  1999,  1996,
14832,  1012,
           102,  3426,   102,  1996,  2155,  2128, 25300, 11020,
```

```
2098,  2058,
          1996,  7760,  1012,   102],
        [  101,  1045,  2363,  1037,  7427,  1999,  1996,  5653,
1012,   102,
          3466,   102,  1045,  2165,  1996,  7427,  2000,  1996,
2695,  2436,
          1012,   102,     0,     0],
        [  101,  1996,  6462,  2006,  2026,  3797,  3062,  2125,
1012,   102,
          3466,   102,  1045, 22805,  1996,  6462,  2067,  2006,
1012,   102,
             0,     0,     0,     0],
        [  101,  2576,  4808,  3631,  2041,  1999,  1996,  3842,
1012,   102,
          3466,   102,  2116,  4480,  2165,  9277,  1999,  2060,
6500,  1012,
           102,     0,     0,     0],
        [  101,  1996,  2450,  2001, 11908,  2005,  6467,  4611,
1012,   102,
          3466,   102,  2016,  8014,  2014, 14651,  1012,   102,
0,     0,
             0,     0,     0,     0],
        [  101,  1996,  8044,  2246,  2601,  1012,   102,  3466,
102,  1045,
          2716,  2026, 12977,  2000,  2147,  1012,   102,     0,
0,     0,
             0,     0,     0,     0],
        [  101,  1996,  2158,  1005,  1055,  4253,  4906, 11853,
1012,   102,
          3426,   102,  2002,  2439,  3635,  1012,   102,     0,
0,     0,
             0,     0,     0,     0],
        [  101,  1996,  8645,  3603,  1996,  4099,  1005,  1055,
3295,  1012,
           102,  3426,   102,  1996,  8645,  2001,  5086,  2011,
1996,  2231,
          1012,   102,     0,     0],
        [  101,  1996,  7596,  5520,  2039,  2046,  1996,  3712,
1012,   102,
          3426,   102,  1996,  2611,  2881,  2009,  1012,   102,
0,     0,
             0,     0,     0,     0],
        [  101,  1996,  2482,  2246, 18294,  1012,   102,  3466,
102,  2002,
          3954,  2165,  2009,  1999,  2005,  1037,  6773,  3105,
1012,   102,
             0,     0,     0,     0],
        [  101,  1996,  2450,  2018,  2019,  8985,  1012,   102,
3466,   102,
```

```
          2016,  8871,  2014,  2398,  1012,   102,     0,     0,
0,     0,
             0,     0,     0,     0]])}
batch_attn_mask:
 {'choice1': tensor([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
0, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0,
0, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
0, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0,
0, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0,
0, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0,
0, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0,
0, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0,
0, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0,
0, 0, 0, 0]]), 'choice2': tensor([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 0, 0],
```

```
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
0, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0,
0, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0,
0, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0,
0, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0,
0, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
0, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0,
0, 0, 0, 0]])}
batch_labels:
 tensor([1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0])
```

Task 2: Implementing and Training BERT-based Multiple Choice Classifier (6 Marks)

```python
# Import BertModel from the library
from transformers import BertModel

# Create an instance of pretrained BERT
bert_model = BertModel.from_pretrained("bert-base-uncased")
bert_model
```

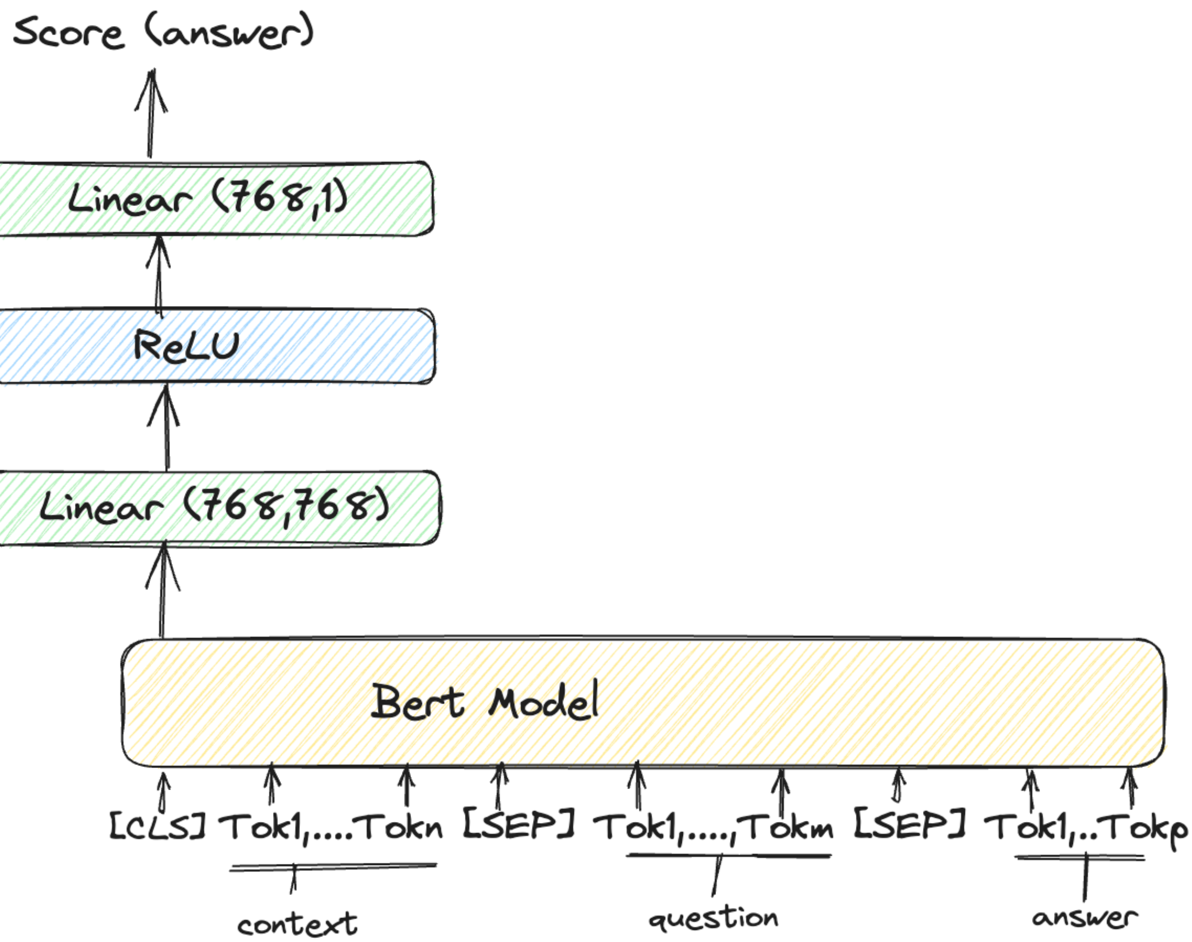
{"model_id":"2ae64a7dc84e484a855c6e030023bd15","version_major":2,"version_minor":0}

```
BertModel(
  (embeddings): BertEmbeddings(
    (word_embeddings): Embedding(30522, 768, padding_idx=0)
    (position_embeddings): Embedding(512, 768)
    (token_type_embeddings): Embedding(2, 768)
    (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
    (dropout): Dropout(p=0.1, inplace=False)
  )
  (encoder): BertEncoder(
    (layer): ModuleList(
      (0-11): 12 x BertLayer(
        (attention): BertAttention(
          (self): BertSdpaSelfAttention(
            (query): Linear(in_features=768, out_features=768,
bias=True)
```

```
            (key): Linear(in_features=768, out_features=768,
bias=True)
            (value): Linear(in_features=768, out_features=768,
bias=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
          (output): BertSelfOutput(
            (dense): Linear(in_features=768, out_features=768,
bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (intermediate): BertIntermediate(
          (dense): Linear(in_features=768, out_features=3072,
bias=True)
          (intermediate_act_fn): GELUActivation()
        )
        (output): BertOutput(
          (dense): Linear(in_features=3072, out_features=768,
bias=True)
          (LayerNorm): LayerNorm((768,), eps=1e-12,
elementwise_affine=True)
          (dropout): Dropout(p=0.1, inplace=False)
        )
      )
    )
  )
  (pooler): BertPooler(
    (dense): Linear(in_features=768, out_features=768, bias=True)
    (activation): Tanh()
  )
)
```

Task 2.1: Implementing BERT-based Classifier for Multiple Choice Classification (2 Marks)

We will be using trhe same exact architecture as SocialIQA dataset here as well i.e. we have the BERT model as the backbone, using which we obtain the contextualized representation of the [premise, question, answer] sequence. We then use the [CLS] token's embedding as the sequence representation and feed it to a 2 layer MLP (Linear(768, 768) -> ReLU -> Linear(768, 1)) that scores the answer.



The only change this time will be that to predict the correct answer, we need to score each of the two choices each instead of the three answers. Afterwards, like last time we ormalize the scores for the choices by applying softmax, that gives us the probability of each option being the correct answer.

Implement the architecture and forward pass in `BertMultiChoiceClassifierModel` class below:

```
class BertMultiChoiceClassifierModel(nn.Module):

    def __init__(self, d_hidden = 768, bert_variant = "bert-base-
uncased"):
        """
        Define the architecture of Bert-Based mulit-choice classifier.
        You will mainly need to define 3 components, first a BERT
layer
        using `BertModel` from transformers library,
        a two layer MLP layer to map the representation from Bert to
the output i.e. (Linear(d_hidden, d_hidden) -> ReLU ->
Linear(d_hidden, 1)),
```

```
        and a log sftmax layer to map the scores to a probabilities

        Inputs:
            - d_hidden (int): Size of the hidden representations of
bert
            - bert_variant (str): BERT variant to use
        """
        super(BertMultiChoiceClassifierModel, self).__init__()
        self.bert_layer = None
        self.mlp_layer = None
        self.log_softmax_layer = None

        # YOUR CODE HERE
        self.bert_layer = BertModel.from_pretrained(bert_variant)
        self.mlp_layer = nn.Sequential(nn.Linear(d_hidden, d_hidden),
nn.ReLU(), nn.Linear(d_hidden, 1))
        self.log_softmax_layer = nn.LogSoftmax(dim=1)

    def forward(self, input_ids_dict, attn_mask_dict):
        """
        Forward Passes the inputs through the network and obtains the
prediction

        Inputs:
            - input_ids_dict (dict(str,torch.tensor)): A dictionary
containing input_ids corresponding to each answer choice. Keys are
choice1 and choice2, and value is a torch tensor of shape [batch_size,
seq_len]
                                        representing the sequence of
token ids
            - attn_mask_dict (dict(str,torch.tensor)): A dictionary
containing attention mask corresponding to each answer choice. Keys
are choice1 and choice2, and value is a torch tensor of shape
[batch_size, seq_len]

        Returns:
          - output (torch.tensor): A torch tensor of shape
[batch_size,] obtained after passing the input to the network

        """
        output = None
        key_outs = []
        # YOUR CODE HERE
        for key in input_ids_dict.keys():
            bert_output =
self.bert_layer(input_ids=input_ids_dict[key],
attention_mask=attn_mask_dict[key])
            pooler_output = bert_output.pooler_output
            mlp_output = self.mlp_layer(pooler_output)
            key_outs.append(mlp_output)
```

```python
        # using torch.cat to concatenate the outputs of the two answer
choices
        output = self.log_softmax_layer(torch.cat(key_outs, dim=1))


        return output

print(f"Running Sample Test Cases!")
torch.manual_seed(42)
model = BertMultiChoiceClassifierModel()

print("Sample Test Case 1")
batch_input_ids, batch_attn_mask, batch_labels =
next(iter(train_loader))
bert_out = model(batch_input_ids, batch_attn_mask).detach().numpy()
expected_bert_out = np.array([[-0.69547975, -0.6908201 ],
                              [-0.6995947,  -0.68674093],
                              [-0.68830335, -0.6980145 ],
                              [-0.6899294,  -0.6963753 ],
                              [-0.6987722,  -0.68755364],
                              [-0.7084117,  -0.6781122 ],
                              [-0.6960533,  -0.6902495 ],
                              [-0.6748525,  -0.71178275],
                              [-0.6868652,  -0.699469  ],
                              [-0.68641526, -0.6999247 ],
                              [-0.6998995,  -0.68644005],
                              [-0.70553845, -0.6809075 ],
                              [-0.7043667,  -0.6820522 ],
                              [-0.6675567,  -0.7194098 ],
                              [-0.700077,   -0.6862651 ],
                              [-0.72046393, -0.6665568 ]])
print(f"Model Output: {bert_out}")
print(f"Expected Output: {expected_bert_out}")

assert bert_out.shape == expected_bert_out.shape
assert np.allclose(bert_out, expected_bert_out, 1e-4)
print("Test Case Passed! :)")
print("******************************\n")

print("Sample Test Case 2")
batch_input_ids, batch_attn_mask, batch_labels =
next(iter(test_loader))
bert_out = model(batch_input_ids, batch_attn_mask).detach().numpy()
expected_bert_out = np.array([[-0.6993066,  -0.6870254 ],
                              [-0.7057758,  -0.68067616],
                              [-0.670608,   -0.71620613],
                              [-0.6946109,  -0.69168556],
                              [-0.6930771,  -0.69321734],
                              [-0.6869541,  -0.6993789 ],
```

```
                            [-0.68364906, -0.7027364 ],
                            [-0.68642354, -0.6999164 ],
                            [-0.6944856,  -0.69181055],
                            [-0.6879125,  -0.6984094 ],
                            [-0.7094514,  -0.67710465],
                            [-0.68425775, -0.7021164 ],
                            [-0.6869471,  -0.6993859 ],
                            [-0.69160426, -0.6946925 ],
                            [-0.68354183, -0.7028456 ],
                            [-0.69150895, -0.69478804]])
print(f"Model Output: {bert_out}")
print(f"Expected Output: {expected_bert_out}")

assert bert_out.shape == expected_bert_out.shape
assert np.allclose(bert_out, expected_bert_out, 1e-4)
print("Test Case Passed! :)")
print("*******************************\n")
```

```
Running Sample Test Cases!
Sample Test Case 1
Model Output: [[-0.6954797  -0.69082016]
 [-0.69959474 -0.686741  ]
 [-0.6883034  -0.6980145 ]
 [-0.6899294  -0.6963753 ]
 [-0.69877225 -0.6875536 ]
 [-0.7084116  -0.6781122 ]
 [-0.6960533  -0.6902495 ]
 [-0.6748525  -0.71178275]
 [-0.6868652  -0.69946885]
 [-0.6864152  -0.6999248 ]
 [-0.6998995  -0.6864401 ]
 [-0.70553845 -0.68090755]
 [-0.7043666  -0.6820522 ]
 [-0.6675567  -0.7194098 ]
 [-0.70007694 -0.6862651 ]
 [-0.7204639  -0.6665568 ]]
Expected Output: [[-0.69547975 -0.6908201 ]
 [-0.6995947  -0.68674093]
 [-0.68830335 -0.6980145 ]
 [-0.6899294  -0.6963753 ]
 [-0.6987722  -0.68755364]
 [-0.7084117  -0.6781122 ]
 [-0.6960533  -0.6902495 ]
 [-0.6748525  -0.71178275]
 [-0.6868652  -0.699469  ]
 [-0.68641526 -0.6999247 ]
 [-0.6998995  -0.68644005]
 [-0.70553845 -0.6809075 ]
 [-0.7043667  -0.6820522 ]
 [-0.6675567  -0.7194098 ]
```

```
 [-0.700077   -0.6862651 ]
 [-0.72046393 -0.6665568 ]]
Test Case Passed! :)
******************************

Sample Test Case 2
Model Output: [[-0.6993066  -0.6870255 ]
 [-0.70577574 -0.68067616]
 [-0.67060804 -0.716206  ]
 [-0.69461083 -0.69168556]
 [-0.6930771  -0.6932172 ]
 [-0.686954   -0.69937897]
 [-0.6836491  -0.7027363 ]
 [-0.6864235  -0.6999164 ]
 [-0.6944856  -0.69181055]
 [-0.68791234 -0.69840944]
 [-0.7094514  -0.6771046 ]
 [-0.6842578  -0.70211625]
 [-0.6869471  -0.6993859 ]
 [-0.69160426 -0.6946925 ]
 [-0.6835418  -0.70284563]
 [-0.6915089  -0.6947881 ]]
Expected Output: [[-0.6993066  -0.6870254 ]
 [-0.7057758  -0.68067616]
 [-0.670608   -0.71620613]
 [-0.6946109  -0.69168556]
 [-0.6930771  -0.69321734]
 [-0.6869541  -0.6993789 ]
 [-0.68364906 -0.7027364 ]
 [-0.68642354 -0.6999164 ]
 [-0.6944856  -0.69181055]
 [-0.6879125  -0.6984094 ]
 [-0.7094514  -0.67710465]
 [-0.68425775 -0.7021164 ]
 [-0.6869471  -0.6993859 ]
 [-0.69160426 -0.6946925 ]
 [-0.68354183 -0.7028456 ]
 [-0.69150895 -0.69478804]]
Test Case Passed! :)
******************************
```

Task 2.2: Training and Evaluating the Model (3 Marks)

Now that we have implemented the custom Dataset and a BERT based classifier model, we can start training and evaluating the model as in Lab 2. You will need to implement the `train` and `evaluate` functions below.

```python
def evaluate(model, test_dataloader, device = "cpu"):
    """
    Evaluates `model` on test dataset

    Inputs:
        - model (BertMultiChoiceClassifierModel): A BERT based
multiple choice classification model to be evaluated
        - test_dataloader (torch.utils.DataLoader): A dataloader
defined over the test dataset

    Returns:
        - accuracy (float): Average accuracy over the test dataset
    """

    model.eval()
    model = model.to(device)
    accuracy = 0

    # YOUR CODE HERE
    with torch.no_grad():
        for test_batch in test_dataloader:
            input_ids_dict, attn_mask_dict, labels = test_batch
            # loading the inputs and labels to the device
            for key in input_ids_dict.keys():
                input_ids_dict[key] = input_ids_dict[key].to(device)
                attn_mask_dict[key] = attn_mask_dict[key].to(device)
            labels = labels.to(device)

            preds = model(input_ids_dict, attn_mask_dict)
            pred = torch.argmax(preds, dim=1)
            pred_accuracy = torch.sum(pred == labels).item()
            batch_accuracy = pred_accuracy / len(labels)
            accuracy += batch_accuracy

    accuracy /= len(test_dataloader)
    return accuracy


def train(model, train_dataloader, test_dataloader,
          lr = 1e-5, num_epochs = 3,
          device = "cpu"):
    """
    Runs the training loop. Define the loss function as BCELoss like
the last tine
    and optimizer as Adam and traine for `num_epochs` epochs.

    Inputs:
        - model (BertMultiChoiceClassifierModel):  A BERT based
multiple choice classification model to be trained
        - train_dataloader (torch.utils.DataLoader): A dataloader
defined over the training dataset
```

```
        - test_dataloader (torch.utils.DataLoader): A dataloader
defined over the test dataset
        - lr (float): The learning rate for the optimizer
        - num_epochs (int): Number of epochs to train the model for.
        - device (str): Device to train the model on. Can be either
'cuda' (for using gpu) or 'cpu'

    Returns:
        - test_accuracy (float): Test accuracy corresponding to the
last epoch
        Note that we are not doing model selection here since we do
not have access to a validation set for this task.
          It is not a good practice to do model selection on the test
set. Hence, we just return the performance we get after training the
model
    """
    epoch_loss = 0
    model = model.to(device)
    test_accuracy = None

    # YOUR CODE HERE
    loss_fn = nn.NLLLoss() # defining the loss function
    optimizer = Adam(model.parameters(), lr=lr) # defining the
optimizer

    for epochs in range(num_epochs):
        epoch_loss = 0

        # iterating over the training dataloader
        for train_batch in tqdm(train_dataloader):
            input_ids_dict, attn_mask_dict, labels = train_batch

            # zeroing out any gradients that might have been left from
the previous iteration
            optimizer.zero_grad()

            # loading the inputs and labels to the device
            for key in input_ids_dict.keys():
                input_ids_dict[key] = input_ids_dict[key].to(device)
                attn_mask_dict[key] = attn_mask_dict[key].to(device)
            labels = labels.to(device)

            # feeding the inputs to the model to get the output log-
probabilities
            preds = model(input_ids_dict, attn_mask_dict)

            # calculating the loss
            loss = loss_fn(preds, labels)

            # backpropagating the gradients
            loss.backward()
```

```python
            # taking an optimization step to update the model parameters
            optimizer.step()

            # adding the loss to the epoch loss
            epoch_loss += loss.item()

        epoch_loss /= len(train_dataloader)

        # printing the epoch loss - just to observe the change in loss with each epoch
        print(f"Epoch: {epochs}, Loss: {epoch_loss}")

        # evaluating the model on the test set
        test_accuracy = evaluate(model, test_dataloader, device)

        print(f"Test Accuracy: {test_accuracy}") # printing for observation as stated below

    return test_accuracy

torch.manual_seed(0)
model = BertMultiChoiceClassifierModel()
test_acc = train(model, train_loader, test_loader, num_epochs = 10, lr = 5e-6, device = "cuda")
```

{"model_id":"bcf76e7e2d2443f5978e221edfe68473","version_major":2,"version_minor":0}

```
Epoch: 0, Loss: 0.6893901154398918
Test Accuracy: 0.6171875
```

{"model_id":"71ee961b18e64441a8145b6b4519bbf8","version_major":2,"version_minor":0}

```
Epoch: 1, Loss: 0.6642385628074408
Test Accuracy: 0.6875
```

{"model_id":"121d8f4f3fe849efbba2655bcd47cfc9","version_major":2,"version_minor":0}

```
Epoch: 2, Loss: 0.5623010629788041
Test Accuracy: 0.666015625
```

{"model_id":"839d2219c1994eb382ba8f5f7962a676","version_major":2,"version_minor":0}

```
Epoch: 3, Loss: 0.3655265886336565
Test Accuracy: 0.65234375
```

```
{"model_id":"946f223dc06e4ba880f3b256215583a2","version_major":2,"version_minor":0}

Epoch: 4, Loss: 0.19504156010225415
Test Accuracy: 0.66796875

{"model_id":"2412e11d418a4b598604e914edc937ca","version_major":2,"version_minor":0}

Epoch: 5, Loss: 0.11494719388429075
Test Accuracy: 0.6640625

{"model_id":"797de3a23cae4808b54a837d85b9c6d7","version_major":2,"version_minor":0}

Epoch: 6, Loss: 0.07024043469573371
Test Accuracy: 0.6640625

{"model_id":"73020a7248f847c2801024c4322d7e3b","version_major":2,"version_minor":0}

Epoch: 7, Loss: 0.041234894830267876
Test Accuracy: 0.6640625

{"model_id":"db2c207b87e04e67b418250d25c5cb34","version_major":2,"version_minor":0}

Epoch: 8, Loss: 0.02742402773583308
Test Accuracy: 0.671875

{"model_id":"bbdd7449d90c45a281501d5fd2070b11","version_major":2,"version_minor":0}

Epoch: 9, Loss: 0.023502633892348967
Test Accuracy: 0.669921875
```

You should expect about ~65% test accuracy. Note that the model quickly overfits to the dataset in this case, i.e. the training loss reduces dramatically, but there isn't much improvement in the test accuracy after the first epoch. This happens because our training data consists of just 500 examples, which is usually not sufficient for training these large models. Next, we try out a simple strategy to improve the performance drastically.

Task 2.3: Continued Fine-tuning of BERT trained on SocialIQA Dataset (1 Mark)

In Lab2, we fine-tuned BERT on SocialIQA, which is also a common sense reasoning task and has a much larger training set. Deep learning models exhibhit a remarkable property of transfer learning where we can leverage a model trained on task to transfer it's knowledge for learning a new task much more effectively. The idea is that training on SocialIQA dataset would have endowed our model with some common-sense reasoning capabilities, which we can leverage to learn COPA task as well.

Below, you are needed to load the model that you trained in Lab2 and the train that model instead. You can read about how to load pre-trained models in pytorch here. Once you load the model, to train it, just call the `train` function as before.

```
model = BertMultiChoiceClassifierModel()
model_path = "gdrive/MyDrive/PlakshaTLF24-NLP/Lab02/models/model.pt" # Change this to the path of your model trained in Lab2.

# Step 1: Load the pre-trained model weights
# YOUR CODE HERE
model.load_state_dict(torch.load(model_path))

# Step 2: Train the loaded model on COPA dataset
# YOUR CODE HERE
test_acc = train(model, train_loader, test_loader, num_epochs = 10, lr = 5e-6, device = "cuda")
```

{"model_id":"eb82bb8dc4b64d33afff6b2b54e160dc","version_major":2,"version_minor":0}

```
Epoch: 0, Loss: 0.5564665822312236
Test Accuracy: 0.771484375
```

{"model_id":"f34784672e0c4385b38a91f6fb16b94c","version_major":2,"version_minor":0}

```
Epoch: 1, Loss: 0.34879961609840393
Test Accuracy: 0.76953125
```

{"model_id":"0f0807e51ddd45b48a14850fd2d401de","version_major":2,"version_minor":0}

```
Epoch: 2, Loss: 0.16341367724817246
Test Accuracy: 0.77734375
```

{"model_id":"dd0e18c0e8a84bd0bc6992c5b8ef6c08","version_major":2,"version_minor":0}

```
Epoch: 3, Loss: 0.05073971877573058
Test Accuracy: 0.779296875
```

{"model_id":"dc125419b56e48809a29c3fe5c61e7b1","version_major":2,"version_minor":0}

```
Epoch: 4, Loss: 0.019151839063852094
Test Accuracy: 0.7734375
```

{"model_id":"94347458c3284b23a7c90bc02440e647","version_major":2,"version_minor":0}

```
Epoch: 5, Loss: 0.010009308156440966
Test Accuracy: 0.78125
```

```
{"model_id":"b3b2c8f7aca7466384371152ca9ab701","version_major":2,"version_minor":0}

Epoch: 6, Loss: 0.00631643453380093
Test Accuracy: 0.77734375

{"model_id":"3a9f8cd377c64d958feb87aedb9ed05b","version_major":2,"version_minor":0}

Epoch: 7, Loss: 0.004664341278839856
Test Accuracy: 0.779296875

{"model_id":"0a1b5a0d2fae41e5a53d24849d8d2075","version_major":2,"version_minor":0}

Epoch: 8, Loss: 0.003735277970918105
Test Accuracy: 0.771484375

{"model_id":"93ee866c9e73489c9723fd902ba4040c","version_major":2,"version_minor":0}

Epoch: 9, Loss: 0.00313233029555704
Test Accuracy: 0.78125
```

You should expect ~77% accuracy with this, which is quite a large increase over the original 65% accuracy that we obtained by training the model from scratch. This illustrates the effectiveness of transfer-learning for NLP tasks, specially when the two tasks are related as they were in this case. Transfer learning had been the dominant paradigm in NLP since 2018. However, recently we have been witnessing a new paradigm emerge called "Prompting", which has taken the NLP community and in many ways the whole world by a storm. In the next lab and assignments, we will learn more about this new paradigm.