FIT3181: Deep Learning (2024) - Assignment 2 (Transformers)

CE/Lecturer (Clayton): **Dr Trung Le** |
Lecturer (Clayton): **Prof Dinh Phung** |
Lecturer (Malaysia): **Dr Arghya Pal** |
Lecturer (Malaysia): **Dr Lim Chern Hong** |

Head Tutor 3181: **Miss Vy Vo** | []
Head Tutor 5215: **Dr Van Nguyen** | []

Faculty of Information Technology, Monash University, Australia ***

Student Information

Surname: **Ali Khan**
Firstname: **Mohib**
Student ID: **33370311**
Email: ****
Your tutorial time: **Friday 10PM**
***

Assignment 2 – Deep Learning for Sequential Data

Due: 11:55pm Sunday, 27 October 2024 (FIT3181)

Important note: This is an individual assignment. It contributes 15% to your final mark. Read the assignment instructions carefully.

Assignment 2's Organization

This assignment 2 has two (2) sections:

- Section 1: Fundamentals of RNNs (10 marks).
- Section 2: Deep Learning for Sequential Data (90 marks). This section is further divided into 4 parts.

The assignment 2 is organized in three (3) notebooks.

- Notebook 1 (link) [Total: 30 marks] includes Section 1 as well as Part 1 and Part 2 of Section 2.
- Notebook 2 (link) [Total: 40 marks] includes Part 3 of Section 2.
- Notebook 3 (this notebook) [Total: 30 marks] includes Part 4 of Section 2.

What to submit

This assignment is to be completed individually and submitted to Moodle unit site. **By the due date, you are required to submit one single zip file, named xxx_assignment02_solution.zip where `xxx` is your student ID, to the corresponding Assignment (Dropbox) in Moodle**. You can use Google Colab to do Assignment 2 but you need to save it to an `*.ipynb` file to submit to the unit Moodle.

More importantly, if you use Google Colab to do this assignment, you need to first make a copy of this notebook on your Google drive.

For example, if your student ID is 12356, then gather all of your assignment solutions to a folder, create a zip file named 123456_assignment02_solution.zip and submit this file.

Within this zip folder, you **must** submit the following files for each part:

1. **`FIT3181_DeepLearning_Assignment2_Official[Main].ipynb`**: this is your Python notebook solution source file.
2. **`FIT3181_DeepLearning_Assignment2_Official[Main].html`**: this is the output of your Python notebook solution *exported* in HTML format.
3. **`FIT3181_DeepLearning_Assignment2_Official[RNNs].ipynb`**
4. **`FIT3181_DeepLearning_Assignment2_Official[RNNs].html`**
5. **`FIT3181_DeepLearning_Assignment2_Official[Transformers].ipynb`**
6. **`FIT3181_DeepLearning_Assignment2_Official[Transformers].html`**
7. Any **extra files or folder** needed to complete your assignment (e.g., images used in your answers).

Section 2: Deep Learning for Sequential Data

Set random seeds

We need to install the package datasets for creating BERT datasets.

```
!pip install datasets
```

```
import os
import torch
import random
import requests
import pandas as pd
import numpy as np
import torch.nn as nn
from torch.utils.data import DataLoader
from torch.nn.utils.rnn import pad_sequence
from transformers import BertTokenizer
import os
from six.moves.urllib.request import urlretrieve
from sklearn import preprocessing
import matplotlib.pyplot as plt
plt.style.use('ggplot')
```

```
def seed_all(seed=1029):
    random.seed(seed)
    os.environ['PYTHONHASHSEED'] = str(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)  # if you are using multi-GPU.
    torch.backends.cudnn.benchmark = False
    torch.backends.cudnn.deterministic = True
seed_all(seed=1234)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
def seed_all(seed=1029):
    random.seed(seed)
```

```
    os.environ['PYTHONHASHSEED'] = str(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)  # if you are using multi-GPU.
    torch.backends.cudnn.benchmark = False
    torch.backends.cudnn.deterministic = True
seed_all(seed=1234)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
The dataset we use for this assignment is a question classification
        dataset for which the training set consists of $5,500$
        questions belonging to 6 coarse question categories including:
- abbreviation (ABBR),
- entity (ENTY),
- description (DESC),
- human (HUM),
- location (LOC) and
- numeric (NUM).

In this assignment, we will utilize a subset of this dataset,
        containing $2,000$ questions for training and validation. We
        will use 80% of those 2000 questions for trainning and the
        rest for validation.
```

```
class DataManager:
    """
    This class manages and preprocesses a simple text dataset for a
        sentence classification task.

    Attributes:
        verbose (bool): Controls verbosity for printing information
        during data processing.
        max_sentence_len (int): The maximum length of a sentence in
        the dataset.
        str_questions (list): A list to store the string
        representations of the questions in the dataset.
        str_labels (list): A list to store the string representations
        of the labels in the dataset.
        numeral_labels (list): A list to store the numerical
        representations of the labels in the dataset.
        maxlen (int): Maximum length for padding sequences. Sequences
        longer than this length will be truncated,
            and sequences shorter than this length will be padded with
        zeros. Defaults to 50.
```

```
        numeral_data (list): A list to store the numerical
        representations of the questions in the dataset.
        random_state (int): Seed value for random number generation to
        ensure reproducibility.
            Set this value to a specific integer to reproduce the same
        random sequence every time. Defaults to 6789.
        random (np.random.RandomState): Random number generator object
        initialized with the given random_state.
            It is used for various random operations in the class.


    Methods:
        maybe_download(dir_name, file_name, url, verbose=True):
            Downloads a file from a given URL if it does not exist in
        the specified directory.
            The directory and file are created if they do not exist.


        read_data(dir_name, file_names):
            Reads data from files in a directory, preprocesses it, and
        computes the maximum sentence length.
            Each file is expected to contain rows in the format "
        <label>:<question>".
            The labels and questions are stored as string
        representations.


        manipulate_data():
            Performs data manipulation by tokenizing, numericalizing,
        and padding the text data.
            The questions are tokenized and converted into numerical
        sequences using a tokenizer.
            The sequences are padded or truncated to the maximum
        sequence length.


        train_valid_test_split(train_ratio=0.9):
            Splits the data into training, validation, and test sets
        based on a given ratio.
            The data is randomly shuffled, and the specified ratio is
        used to determine the size of the training set.
            The string questions, numerical data, and numerical labels
        are split accordingly.
            TensorFlow `Dataset` objects are created for the training
        and validation sets.



    """

    def __init__(self, verbose=True, random_state=6789):
        self.verbose = verbose
        self.max_sentence_len = 0
        self.str_questions = list()
        self.str_labels = list()
```

```
        self.numeral_labels = list()
        self.numeral_data = list()
        self.random_state = random_state
        self.random = np.random.RandomState(random_state)

    @staticmethod
    def maybe_download(dir_name, file_name, url, verbose=True):
        if not os.path.exists(dir_name):
            os.mkdir(dir_name)
        if not os.path.exists(os.path.join(dir_name, file_name)):
            urlretrieve(url + file_name, os.path.join(dir_name,
        file_name))
        if verbose:
            print("Downloaded successfully {}".format(file_name))

    def read_data(self, dir_name, file_names):
        self.str_questions = list()
        self.str_labels = list()
        for file_name in file_names:
            file_path= os.path.join(dir_name, file_name)
            with open(file_path, "r", encoding="latin-1") as f:
                for row in f:
                    row_str = row.split(":")
                    label, question = row_str[0], row_str[1]
                    question = question.lower()
                    self.str_labels.append(label)
                    self.str_questions.append(question[0:-1])
                    if self.max_sentence_len <
        len(self.str_questions[-1]):
                        self.max_sentence_len =
        len(self.str_questions[-1])

        # turns labels into numbers
        le = preprocessing.LabelEncoder()
        le.fit(self.str_labels)
        self.numeral_labels = np.array(le.transform(self.str_labels))
        self.str_classes = le.classes_
        self.num_classes = len(self.str_classes)
        if self.verbose:
            print("\nSample questions and corresponding labels... \n")
            print(self.str_questions[0:5])
            print(self.str_labels[0:5])

    def manipulate_data(self):
```

```
        self.tokenizer = BertTokenizer.from_pretrained('bert-base-
uncased')
        vocab = self.tokenizer.get_vocab()
        self.word2idx = {w: i for i, w in enumerate(vocab)}
        self.idx2word = {i:w for w,i in self.word2idx.items()}
        self.vocab_size = len(self.word2idx)

        token_ids = []
        num_seqs = []
        for text in self.str_questions:  # iterate over the list of
text
          text_seqs = self.tokenizer.tokenize(str(text))  # tokenize
each text individually
          # Convert tokens to IDs
          token_ids = self.tokenizer.convert_tokens_to_ids(text_seqs)
          # Convert token IDs to a tensor of indices using your
word2idx mapping
          seq_tensor = torch.LongTensor(token_ids)
          num_seqs.append(seq_tensor)  # append the tensor for each
sequence

        # Pad the sequences and create a tensor
        if num_seqs:
          self.numeral_data = pad_sequence(num_seqs, batch_first=True)
# Pads to max length of the sequences
          self.num_sentences, self.max_seq_len =
self.numeral_data.shape

    def train_valid_test_split(self, train_ratio=0.8, test_ratio =
0.1):
        train_size = int(self.num_sentences*train_ratio) +1
        test_size = int(self.num_sentences*test_ratio) +1
        valid_size = self.num_sentences - (train_size + test_size)
        data_indices = list(range(self.num_sentences))
        random.shuffle(data_indices)
        self.train_str_questions = [self.str_questions[i] for i in
data_indices[:train_size]]
        self.train_numeral_labels =
self.numeral_labels[data_indices[:train_size]]
        train_set_data = self.numeral_data[data_indices[:train_size]]
        train_set_labels =
self.numeral_labels[data_indices[:train_size]]
        train_set_labels = torch.from_numpy(train_set_labels)
        train_set = torch.utils.data.TensorDataset(train_set_data,
train_set_labels)
        self.test_str_questions = [self.str_questions[i] for i in
data_indices[-test_size:]]
        self.test_numeral_labels = self.numeral_labels[data_indices[-
test_size:]]
        test_set_data = self.numeral_data[data_indices[-test_size:]]
```

```
        test_set_labels = self.numeral_labels[data_indices[-
        test_size:]]
        test_set_labels = torch.from_numpy(test_set_labels)
        test_set = torch.utils.data.TensorDataset(test_set_data,
        test_set_labels)
        self.valid_str_questions = [self.str_questions[i] for i in
        data_indices[train_size:-test_size]]
        self.valid_numeral_labels =
        self.numeral_labels[data_indices[train_size:-test_size]]
        valid_set_data = self.numeral_data[data_indices[train_size:-
        test_size]]
        valid_set_labels =
        self.numeral_labels[data_indices[train_size:-test_size]]
        valid_set_labels = torch.from_numpy(valid_set_labels)
        valid_set = torch.utils.data.TensorDataset(valid_set_data,
        valid_set_labels)
        self.train_loader = DataLoader(train_set, batch_size=64,
        shuffle=True)
        self.test_loader = DataLoader(test_set, batch_size=64,
        shuffle=False)
        self.valid_loader = DataLoader(valid_set, batch_size=64,
        shuffle=False)
```

```
print('Loading data...')
DataManager.maybe_download("data", "train_2000.label",
        "http://cogcomp.org/Data/QA/QC/")

dm = DataManager()
dm.read_data("data/", ["train_2000.label"])
```


```
print('Loading data...')
DataManager.maybe_download("data", "train_2000.label",
        "http://cogcomp.org/Data/QA/QC/")

dm = DataManager()
dm.read_data("data/", ["train_2000.label"])
```


```
Loading data...
Downloaded successfully train_2000.label

Sample questions and corresponding labels...
```

```
['manner how did serfdom develop in and then leave russia ?', 'cremat
        what films featured the character popeye doyle ?', "manner how
        can i find a list of celebrities ' real names ?", 'animal what
        fowl grabs the spotlight after the chinese year of the monkey
        ?', 'exp what is the full form of .com ?']
['DESC', 'ENTY', 'DESC', 'ENTY', 'ABBR']
```

```
dm.manipulate_data()
dm.train_valid_test_split(train_ratio=0.8, test_ratio = 0.1)
```

```
for x, y in dm.train_loader:
    print(x.shape, y.shape)
    break
```

```
We now declare the `BaseTrainer` class, which will be used later to
        train the subsequent deep learning models for text data.
```

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

class BaseTrainer:
    def __init__(self, model, criterion, optimizer, train_loader,
        val_loader):
        self.model = model
        self.criterion = criterion  #the loss function
        self.optimizer = optimizer  #the optimizer
        self.train_loader = train_loader  #the train loader
        self.val_loader = val_loader  #the valid loader

    #the function to train the model in many epochs
    def fit(self, num_epochs):
        self.num_batches = len(self.train_loader)

        for epoch in range(num_epochs):
            print(f'Epoch {epoch + 1}/{num_epochs}')
            train_loss, train_accuracy = self.train_one_epoch()
            val_loss, val_accuracy = self.validate_one_epoch()
            print(
                f'{self.num_batches}/{self.num_batches} - train_loss:
        {train_loss:.4f} - train_accuracy: {train_accuracy*100:.4f}% \
                - val_loss: {val_loss:.4f} - val_accuracy:
        {val_accuracy*100:.4f}%')

    #train in one epoch, return the train_acc, train_loss
```

```python
    def train_one_epoch(self):
        self.model.train()
        running_loss, correct, total = 0.0, 0, 0
        for i, data in enumerate(self.train_loader):
            inputs, labels = data
            inputs, labels = inputs.to(device), labels.to(device)
            self.optimizer.zero_grad()
            outputs = self.model(inputs)
            loss = self.criterion(outputs, labels)
            loss.backward()
            self.optimizer.step()

            running_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
        train_accuracy = correct / total
        train_loss = running_loss / self.num_batches
        return train_loss, train_accuracy

    #evaluate on a loader and return the loss and accuracy
    def evaluate(self, loader):
        self.model.eval()
        loss, correct, total = 0.0, 0, 0
        with torch.no_grad():
            for data in loader:
                inputs, labels = data
                inputs, labels = inputs.to(device), labels.to(device)
                outputs = self.model(inputs)
                loss = self.criterion(outputs, labels)
                loss += loss.item()
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()

        accuracy = correct / total
        loss = loss / len(self.val_loader)
        return loss, accuracy

    #return the val_acc, val_loss, be called at the end of each epoch
    def validate_one_epoch(self):
      val_loss, val_accuracy = self.evaluate(self.val_loader)
      return val_loss, val_accuracy
```

```
  Cell In[49], line 50
    The dataset we use for this assignment is a question
classification dataset for which the training set consists of $5,500$
questions belonging to 6 coarse question categories including:
        ^
SyntaxError: invalid syntax
```

We start with importing PyTorch and NumPy and setting random seeds for PyTorch and NumPy. You can use any seeds you prefer.

```python
import os
import torch
import random
import requests
import pandas as pd
import numpy as np
import torch.nn as nn
from torch.utils.data import DataLoader
from torch.nn.utils.rnn import pad_sequence
from transformers import BertTokenizer
import os
from six.moves.urllib.request import urlretrieve
from sklearn import preprocessing
import matplotlib.pyplot as plt
plt.style.use('ggplot')
```

```
/Users/mohibalikhan/Desktop/Deep Learning
A02/myenv/lib/python3.9/site-packages/urllib3/__init__.py:35:
NotOpenSSLWarning: urllib3 v2 only supports OpenSSL 1.1.1+, currently
the 'ssl' module is compiled with 'LibreSSL 2.8.3'. See:
https://github.com/urllib3/urllib3/issues/3020
  warnings.warn(
/Users/mohibalikhan/Desktop/Deep Learning
A02/myenv/lib/python3.9/site-packages/tqdm/auto.py:21: TqdmWarning:
IProgress not found. Please update jupyter and ipywidgets. See
https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
```

```python
def seed_all(seed=1029):
    random.seed(seed)
    os.environ['PYTHONHASHSEED'] = str(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
```

```
    torch.cuda.manual_seed_all(seed)  # if you are using multi-GPU.
    torch.backends.cudnn.benchmark = False
    torch.backends.cudnn.deterministic = True
seed_all(seed=1234)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Download and preprocess the data

The dataset we use for this assignment is a question classification dataset for which the training set consists of $5,500$ questions belonging to 6 coarse question categories including:

- abbreviation (ABBR),
- entity (ENTY),
- description (DESC),
- human (HUM),
- location (LOC) and
- numeric (NUM).

In this assignment, we will utilize a subset of this dataset, containing $2,000$ questions for training and validation. We will use 80% of those 2000 questions for trainning and the rest for validation.

Preprocessing data is a crucial initial step in any machine learning or deep learning project. The *TextDataManager* class simplifies the process by providing functionalities to download and preprocess data specifically designed for the subsequent questions in this assignment. It is highly recommended to gain a comprehensive understanding of the class's functionality by **carefully reading** the content provided in the *TextDataManager* class before proceeding to answer the questions.

```
class DataManager:
    """
    This class manages and preprocesses a simple text dataset for a
        sentence classification task.

    Attributes:
        verbose (bool): Controls verbosity for printing information
        during data processing.
        max_sentence_len (int): The maximum length of a sentence in
        the dataset.
        str_questions (list): A list to store the string
        representations of the questions in the dataset.
        str_labels (list): A list to store the string representations
        of the labels in the dataset.
```

```
        numeral_labels (list): A list to store the numerical
        representations of the labels in the dataset.
        maxlen (int): Maximum length for padding sequences. Sequences
        longer than this length will be truncated,
            and sequences shorter than this length will be padded with
        zeros. Defaults to 50.
        numeral_data (list): A list to store the numerical
        representations of the questions in the dataset.
        random_state (int): Seed value for random number generation to
        ensure reproducibility.
            Set this value to a specific integer to reproduce the same
        random sequence every time. Defaults to 6789.
        random (np.random.RandomState): Random number generator object
        initialized with the given random_state.
            It is used for various random operations in the class.

    Methods:
        maybe_download(dir_name, file_name, url, verbose=True):
            Downloads a file from a given URL if it does not exist in
        the specified directory.
            The directory and file are created if they do not exist.

        read_data(dir_name, file_names):
            Reads data from files in a directory, preprocesses it, and
        computes the maximum sentence length.
            Each file is expected to contain rows in the format "
        <label>:<question>".
            The labels and questions are stored as string
        representations.

        manipulate_data():
            Performs data manipulation by tokenizing, numericalizing,
        and padding the text data.
            The questions are tokenized and converted into numerical
        sequences using a tokenizer.
            The sequences are padded or truncated to the maximum
        sequence length.

        train_valid_test_split(train_ratio=0.9):
            Splits the data into training, validation, and test sets
        based on a given ratio.
            The data is randomly shuffled, and the specified ratio is
        used to determine the size of the training set.
            The string questions, numerical data, and numerical labels
        are split accordingly.
            TensorFlow `Dataset` objects are created for the training
        and validation sets.

    """
```

```
    def __init__(self, verbose=True, random_state=6789):
        self.verbose = verbose
        self.max_sentence_len = 0
        self.str_questions = list()
        self.str_labels = list()
        self.numeral_labels = list()
        self.numeral_data = list()
        self.random_state = random_state
        self.random = np.random.RandomState(random_state)

    @staticmethod
    def maybe_download(dir_name, file_name, url, verbose=True):
        if not os.path.exists(dir_name):
            os.mkdir(dir_name)
        if not os.path.exists(os.path.join(dir_name, file_name)):
            urlretrieve(url + file_name, os.path.join(dir_name,
        file_name))
        if verbose:
            print("Downloaded successfully {}".format(file_name))

    def read_data(self, dir_name, file_names):
        self.str_questions = list()
        self.str_labels = list()
        for file_name in file_names:
            file_path= os.path.join(dir_name, file_name)
            with open(file_path, "r", encoding="latin-1") as f:
                for row in f:
                    row_str = row.split(":")
                    label, question = row_str[0], row_str[1]
                    question = question.lower()
                    self.str_labels.append(label)
                    self.str_questions.append(question[0:-1])
                    if self.max_sentence_len <
        len(self.str_questions[-1]):
                        self.max_sentence_len =
        len(self.str_questions[-1])

        # turns labels into numbers
        le = preprocessing.LabelEncoder()
        le.fit(self.str_labels)
        self.numeral_labels = np.array(le.transform(self.str_labels))
        self.str_classes = le.classes_
        self.num_classes = len(self.str_classes)
        if self.verbose:
            print("\nSample questions and corresponding labels... \n")
```

```
            print(self.str_questions[0:5])
            print(self.str_labels[0:5])

    def manipulate_data(self):
        self.tokenizer = BertTokenizer.from_pretrained('bert-base-
        uncased')
        vocab = self.tokenizer.get_vocab()
        self.word2idx = {w: i for i, w in enumerate(vocab)}
        self.idx2word = {i:w for w,i in self.word2idx.items()}
        self.vocab_size = len(self.word2idx)


        token_ids = []
        num_seqs = []
        for text in self.str_questions:  # iterate over the list of
        text
          text_seqs = self.tokenizer.tokenize(str(text))  # tokenize
        each text individually
          # Convert tokens to IDs
          token_ids = self.tokenizer.convert_tokens_to_ids(text_seqs)
          # Convert token IDs to a tensor of indices using your
        word2idx mapping
          seq_tensor = torch.LongTensor(token_ids)
          num_seqs.append(seq_tensor)  # append the tensor for each
        sequence


        # Pad the sequences and create a tensor
        if num_seqs:
          self.numeral_data = pad_sequence(num_seqs, batch_first=True)
        # Pads to max length of the sequences
          self.num_sentences, self.max_seq_len =
        self.numeral_data.shape

    def train_valid_test_split(self, train_ratio=0.8, test_ratio =
        0.1):
        train_size = int(self.num_sentences*train_ratio) +1
        test_size = int(self.num_sentences*test_ratio) +1
        valid_size = self.num_sentences - (train_size + test_size)
        data_indices = list(range(self.num_sentences))
        random.shuffle(data_indices)
        self.train_str_questions = [self.str_questions[i] for i in
        data_indices[:train_size]]
        self.train_numeral_labels =
        self.numeral_labels[data_indices[:train_size]]
        train_set_data = self.numeral_data[data_indices[:train_size]]
        train_set_labels =
        self.numeral_labels[data_indices[:train_size]]
        train_set_labels = torch.from_numpy(train_set_labels)
        train_set = torch.utils.data.TensorDataset(train_set_data,
        train_set_labels)
```

```
        self.test_str_questions = [self.str_questions[i] for i in
        data_indices[-test_size:]]
        self.test_numeral_labels = self.numeral_labels[data_indices[-
        test_size:]]
        test_set_data = self.numeral_data[data_indices[-test_size:]]
        test_set_labels = self.numeral_labels[data_indices[-
        test_size:]]
        test_set_labels = torch.from_numpy(test_set_labels)
        test_set = torch.utils.data.TensorDataset(test_set_data,
        test_set_labels)
        self.valid_str_questions = [self.str_questions[i] for i in
        data_indices[train_size:-test_size]]
        self.valid_numeral_labels =
        self.numeral_labels[data_indices[train_size:-test_size]]
        valid_set_data = self.numeral_data[data_indices[train_size:-
        test_size]]
        valid_set_labels =
        self.numeral_labels[data_indices[train_size:-test_size]]
        valid_set_labels = torch.from_numpy(valid_set_labels)
        valid_set = torch.utils.data.TensorDataset(valid_set_data,
        valid_set_labels)
        self.train_loader = DataLoader(train_set, batch_size=64,
        shuffle=True)
        self.test_loader = DataLoader(test_set, batch_size=64,
        shuffle=False)
        self.valid_loader = DataLoader(valid_set, batch_size=64,
        shuffle=False)

print('Loading data...')
DataManager.maybe_download("data", "train_2000.label",
        "http://cogcomp.org/Data/QA/QC/")


dm = DataManager()
dm.read_data("data/", ["train_2000.label"])
```

```
Loading data...
Downloaded successfully train_2000.label


Sample questions and corresponding labels...


['manner how did serfdom develop in and then leave russia ?', 'cremat
what films featured the character popeye doyle ?', "manner how can i
find a list of celebrities ' real names ?", 'animal what fowl grabs
the spotlight after the chinese year of the monkey ?', 'exp what is
the full form of .com ?']
['DESC', 'ENTY', 'DESC', 'ENTY', 'ABBR']
```

```
dm.manipulate_data()
dm.train_valid_test_split(train_ratio=0.8, test_ratio = 0.1)
```

```
for x, y in dm.train_loader:
    print(x.shape, y.shape)
    break
```

```
torch.Size([64, 36]) torch.Size([64])
```

We now declare the `BaseTrainer` class, which will be used later to train the subsequent deep learning models for text data.

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

class BaseTrainer:
    def __init__(self, model, criterion, optimizer, train_loader,
     val_loader):
        self.model = model
        self.criterion = criterion  #the loss function
        self.optimizer = optimizer  #the optimizer
        self.train_loader = train_loader  #the train loader
        self.val_loader = val_loader  #the valid loader

    #the function to train the model in many epochs
    def fit(self, num_epochs):
        self.num_batches = len(self.train_loader)

        for epoch in range(num_epochs):
            print(f'Epoch {epoch + 1}/{num_epochs}')
            train_loss, train_accuracy = self.train_one_epoch()
            val_loss, val_accuracy = self.validate_one_epoch()
            print(
                f'{self.num_batches}/{self.num_batches} - train_loss:
     {train_loss:.4f} - train_accuracy: {train_accuracy*100:.4f}% \
                - val_loss: {val_loss:.4f} - val_accuracy:
     {val_accuracy*100:.4f}%')

    #train in one epoch, return the train_acc, train_loss
    def train_one_epoch(self):
        self.model.train()
        running_loss, correct, total = 0.0, 0, 0
        for i, data in enumerate(self.train_loader):
            inputs, labels = data
            inputs, labels = inputs.to(device), labels.to(device)
            self.optimizer.zero_grad()
            outputs = self.model(inputs)
            loss = self.criterion(outputs, labels)
            loss.backward()
            self.optimizer.step()
```

```
            running_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
        train_accuracy = correct / total
        train_loss = running_loss / self.num_batches
        return train_loss, train_accuracy

    #evaluate on a loader and return the loss and accuracy
    def evaluate(self, loader):
        self.model.eval()
        loss, correct, total = 0.0, 0, 0
        with torch.no_grad():
            for data in loader:
                inputs, labels = data
                inputs, labels = inputs.to(device), labels.to(device)
                outputs = self.model(inputs)
                loss = self.criterion(outputs, labels)
                loss += loss.item()
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()

        accuracy = correct / total
        loss = loss / len(self.val_loader)
        return loss, accuracy

    #return the val_acc, val_loss, be called at the end of each epoch
    def validate_one_epoch(self):
      val_loss, val_accuracy = self.evaluate(self.val_loader)
      return val_loss, val_accuracy
```

Part 4: Transformer-based models for sequence modeling and neural embedding

[Total marks for this part: 30 marks]

Question 4.1

Implement the multi-head attention module of the Transformer for the text classification problem. The provided code is from our tutorial. In this part, we only use the output of the Transformer encoder for the classification task. For further information on the Transformer model, refer to this paper.

[Total marks for this part: 10 marks]

Below is the code of `MultiHeadSelfAttention`, `PositionWiseFeedForward`, `PositionalEncoding`, and `EncoderLayer`.

```
class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        super(MultiHeadAttention, self).__init__()
        # Ensure that the model dimension (d_model) is divisible by
         the number of heads
        assert d_model % num_heads == 0, "d_model must be divisible by
         num_heads"

        # Initialize dimensions
        self.d_model = d_model # Model's dimension
        self.num_heads = num_heads # Number of attention heads
        self.d_k = d_model // num_heads # Dimension of each head's
         key, query, and value

        # Linear layers for transforming inputs
        self.W_q = nn.Linear(d_model, d_model) # Query transformation
        self.W_k = nn.Linear(d_model, d_model) # Key transformation
        self.W_v = nn.Linear(d_model, d_model) # Value transformation
        self.W_o = nn.Linear(d_model, d_model) # Output transformation

    def scaled_dot_product_attention(self, Q, K, V):
        # Calculate attention scores
        attn_scores = torch.matmul(Q, K.transpose(-2, -1)) /
         math.sqrt(self.d_k)

        # Apply mask if provided (useful for preventing attention to
         certain parts like padding)
        #if mask is not None:
            #attn_scores = attn_scores.masked_fill(mask == 0, -1e9)

        # Softmax is applied to obtain attention probabilities
        attn_probs = torch.softmax(attn_scores, dim=-1)

        # Multiply by values to obtain the final output
```

```
        output = torch.matmul(attn_probs, V)
        return output

    def split_heads(self, x):
        # Reshape the input to have num_heads for multi-head attention
        batch_size, seq_length, d_model = x.size()
        return x.view(batch_size, seq_length, self.num_heads,
         self.d_k).transpose(1, 2)

    def combine_heads(self, x):
        # Combine the multiple heads back to original shape
        batch_size, _, seq_length, d_k = x.size()
        return x.transpose(1, 2).contiguous().view(batch_size,
         seq_length, self.d_model)

    def forward(self, Q, K, V):
        # Apply linear transformations and split heads
        Q = self.split_heads(self.W_q(Q))
        K = self.split_heads(self.W_k(K))
        V = self.split_heads(self.W_v(V))

        # Perform scaled dot-product attention
        attn_output = self.scaled_dot_product_attention(Q, K, V)

        # Combine heads and apply output transformation
        output = self.W_o(self.combine_heads(attn_output))
        return output

class PositionWiseFeedForward(nn.Module):
    def __init__(self, d_model, d_ff):
        super(PositionWiseFeedForward, self).__init__()
        self.fc1 = nn.Linear(d_model, d_ff)
        self.fc2 = nn.Linear(d_ff, d_model)
        self.relu = nn.ReLU()

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

import math

class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_seq_length):
        super(PositionalEncoding, self).__init__()

        pe = torch.zeros(max_seq_length, d_model)
```

```
        position = torch.arange(0, max_seq_length,
        dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * -
        (math.log(10000.0) / d_model))

        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)

        self.register_buffer('pe', pe.unsqueeze(0))

    def forward(self, x):
        return x + self.pe[:, :x.size(1)]

class EncoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, d_ff, dropout):
        super(EncoderLayer, self).__init__()
        self.self_attn = MultiHeadAttention(d_model, num_heads)
        self.feed_forward = PositionWiseFeedForward(d_model, d_ff)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

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

Your task is to develop `TransformerClassifier` in which we input the embedding with the shape `[batch_size, seq_len, embed_dim]` to some `EncoderLayer` (i.e., num_layers specifies the number of EncoderLayer) and then compute the average of all token embeddings (i.e., `[batch_size, seq_len, embed_dim]`) across the `seq_len`. Finally, on the top of this average embedding, we build up a linear layer for making predictions.

```
class TransformerClassifier(nn.Module):
    def __init__(self, embed_dim, num_heads, ff_dim, num_layers,
        dropout_rate=0.2, data_manager = None):
        super(TransformerClassifier, self).__init__()
        self.vocab_size = data_manager.vocab_size
        self.num_classes = data_manager.num_classes
        self.embed_dim = embed_dim
        self.max_seq_len = data_manager.max_seq_len
        self.num_heads = num_heads
        self.ff_dim = ff_dim
```

```
        self.num_layers = num_layers
        self.dropout_rate = dropout_rate

    def build(self):
        #Insert your code here




    def forward(self, x):
        #Insert your code here




class TransformerClassifier(nn.Module):
    def __init__(self, embed_dim, num_heads, ff_dim, num_layers,
dropout_rate=0.2, data_manager=None):
        """
        Initializes the TransformerClassifier.

        Args:
            embed_dim (int): Dimension of the input embeddings
(d_model).
            num_heads (int): Number of attention heads.
            ff_dim (int): Dimension of the feed-forward network.
            num_layers (int): Number of encoder layers.
            dropout_rate (float): Dropout probability.
            data_manager (DataManager): Instance containing dataset
information.
        """
        super(TransformerClassifier, self).__init__()
        self.vocab_size = data_manager.vocab_size
        self.num_classes = data_manager.num_classes
        self.embed_dim = embed_dim
        self.max_seq_len = data_manager.max_seq_len
        self.num_heads = num_heads
        self.ff_dim = ff_dim
        self.num_layers = num_layers
        self.dropout_rate = dropout_rate

    def build(self):
        """
        Builds the TransformerClassifier architecture by initializing:
        - Embedding layer
        - Positional encoding
        - Encoder layers
```

```
        - Final classification layer
        """
        # Initialize Embedding Layer
        self.embedding = nn.Embedding(self.vocab_size, self.embed_dim)

        # Initialize Positional Encoding with d_model = embed_dim
        self.positional_encoding = 
PositionalEncoding(d_model=self.embed_dim, 
max_seq_length=self.max_seq_len)

        # Initialize a stack of Encoder Layers
        self.encoder_layers = nn.ModuleList([
            EncoderLayer(d_model=self.embed_dim, 
num_heads=self.num_heads, d_ff=self.ff_dim, 
dropout=self.dropout_rate)
            for _ in range(self.num_layers)
        ])

        # Initialize the final linear layer for classification
        self.fc = nn.Linear(self.embed_dim, self.num_classes)

    def forward(self, x):
        """
        Defines the forward pass of the TransformerClassifier.

        Args:
            x (torch.Tensor): Input tensor of shape [batch_size, 
seq_len]

        Returns:
            torch.Tensor: Logits tensor of shape [batch_size, 
num_classes]
        """
        # Apply Embedding
        x = self.embedding(x)  # Shape: [batch_size, seq_len, 
embed_dim]

        # Apply Positional Encoding
        x = self.positional_encoding(x)  # Shape: [batch_size, 
seq_len, embed_dim]

        # Pass through each Encoder Layer
        for encoder in self.encoder_layers:
            x = encoder(x)  # Shape remains [batch_size, seq_len, 
embed_dim]
```

```
        # Compute the average of all token embeddings across the
        sequence length
        x = torch.mean(x, dim=1)  # Shape: [batch_size, embed_dim]

        # Pass the averaged embedding through the final linear layer
        logits = self.fc(x)  # Shape: [batch_size, num_classes]

        return logits

transformer = TransformerClassifier(embed_dim=512, num_heads=8,
        ff_dim=2048, num_layers=12, dropout_rate=0.1, data_manager=
        dm)
transformer.build()
transformer = transformer.to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(transformer.parameters(), lr=1e-4, betas=
        (0.9, 0.98), eps=1e-9)
trainer = BaseTrainer(model= transformer, criterion=criterion,
        optimizer=optimizer, train_loader=dm.train_loader,
        val_loader=dm.valid_loader)
trainer.fit(num_epochs=30)
```

```
Epoch 1/30
26/26 - train_loss: 1.9181 - train_accuracy: 21.2367%
- val_loss: 0.9617 - val_accuracy: 12.6263%
Epoch 2/30
26/26 - train_loss: 1.7264 - train_accuracy: 21.6115%
- val_loss: 0.7609 - val_accuracy: 27.2727%
Epoch 3/30
26/26 - train_loss: 1.6812 - train_accuracy: 23.6727%
- val_loss: 0.8164 - val_accuracy: 26.2626%
Epoch 4/30
26/26 - train_loss: 1.6847 - train_accuracy: 22.2361%
- val_loss: 0.8395 - val_accuracy: 26.2626%
Epoch 5/30
26/26 - train_loss: 1.6815 - train_accuracy: 21.4241%
- val_loss: 0.7077 - val_accuracy: 27.2727%
Epoch 6/30
26/26 - train_loss: 1.6683 - train_accuracy: 20.5497%
- val_loss: 0.7249 - val_accuracy: 27.2727%
Epoch 7/30
26/26 - train_loss: 1.6892 - train_accuracy: 22.6109%
- val_loss: 0.7372 - val_accuracy: 27.2727%
Epoch 8/30
26/26 - train_loss: 1.3032 - train_accuracy: 45.2842%
- val_loss: 0.4200 - val_accuracy: 60.6061%
```

```
Epoch 9/30
26/26 - train_loss: 0.7795 - train_accuracy: 69.6440%
- val_loss: 0.2182 - val_accuracy: 85.8586%
Epoch 10/30
26/26 - train_loss: 0.3069 - train_accuracy: 89.2567%
- val_loss: 0.0751 - val_accuracy: 92.4242%
Epoch 11/30
26/26 - train_loss: 0.1498 - train_accuracy: 94.5034%
- val_loss: 0.0174 - val_accuracy: 94.9495%
Epoch 12/30
26/26 - train_loss: 0.0997 - train_accuracy: 96.0025%
- val_loss: 0.0052 - val_accuracy: 94.4444%
Epoch 13/30
26/26 - train_loss: 0.1182 - train_accuracy: 95.7527%
- val_loss: 0.0180 - val_accuracy: 95.4545%
Epoch 14/30
26/26 - train_loss: 0.0753 - train_accuracy: 97.3142%
- val_loss: 0.0036 - val_accuracy: 94.9495%
Epoch 15/30
26/26 - train_loss: 0.1091 - train_accuracy: 97.5640%
- val_loss: 0.0271 - val_accuracy: 94.4444%
Epoch 16/30
26/26 - train_loss: 0.1283 - train_accuracy: 97.7514%
- val_loss: 0.0131 - val_accuracy: 93.9394%
Epoch 17/30
26/26 - train_loss: 0.3086 - train_accuracy: 90.1312%
- val_loss: 0.0166 - val_accuracy: 93.9394%
Epoch 18/30
26/26 - train_loss: 0.1368 - train_accuracy: 95.0031%
- val_loss: 0.0212 - val_accuracy: 93.9394%
Epoch 19/30
26/26 - train_loss: 0.1322 - train_accuracy: 95.0656%
- val_loss: 0.0264 - val_accuracy: 93.9394%
Epoch 20/30
26/26 - train_loss: 0.1409 - train_accuracy: 94.8157%
- val_loss: 0.0936 - val_accuracy: 93.4343%
Epoch 21/30
26/26 - train_loss: 0.1337 - train_accuracy: 94.8782%
- val_loss: 0.0107 - val_accuracy: 93.9394%
Epoch 22/30
26/26 - train_loss: 0.1215 - train_accuracy: 95.3154%
- val_loss: 0.0034 - val_accuracy: 95.4545%
Epoch 23/30
```

```
26/26 - train_loss: 0.0941 - train_accuracy: 97.5640%
- val_loss: 0.0091 - val_accuracy: 95.9596%
Epoch 24/30
26/26 - train_loss: 0.0449 - train_accuracy: 98.4385%
- val_loss: 0.0193 - val_accuracy: 94.9495%
Epoch 25/30
26/26 - train_loss: 0.0680 - train_accuracy: 97.9388%
- val_loss: 0.0022 - val_accuracy: 94.4444%
Epoch 26/30
26/26 - train_loss: 0.0487 - train_accuracy: 98.6883%
- val_loss: 0.0008 - val_accuracy: 95.9596%
Epoch 27/30
26/26 - train_loss: 0.0314 - train_accuracy: 99.0631%
- val_loss: 0.0008 - val_accuracy: 91.4141%
Epoch 28/30
26/26 - train_loss: 0.0219 - train_accuracy: 99.4379%
- val_loss: 0.0005 - val_accuracy: 95.4545%
Epoch 29/30
26/26 - train_loss: 0.0475 - train_accuracy: 98.8132%
- val_loss: 0.0008 - val_accuracy: 95.9596%
Epoch 30/30
26/26 - train_loss: 0.0367 - train_accuracy: 99.2505%
- val_loss: 0.0001 - val_accuracy: 95.9596%
```

Question 4.2

Prefix prompt-tuning with Transformers: You need to implement the prefix prompt-tuning with Transformers. Basically, we base on a pre-trained Transformer, add prefix prompts, and do fine-tuning for a target dataset.

[Total marks for this part: 10 marks]

To implement prefix prompt-tuning with pretrained Transformers, we first need to create the Bert dataset.

```python
from transformers import AutoModel, AutoTokenizer, AdamW
from datasets import Dataset

model_name = "bert-base-uncased"  # BERT or any similar model

# Tokenize input and prepare model inputs
tokenizer = AutoTokenizer.from_pretrained(model_name)
```

```
dataset = Dataset.from_dict({"text": dm.str_questions, "label":
        dm.numeral_labels})


# Tokenize the dataset
def tokenize_function(examples):
    return tokenizer(examples["text"], padding="max_length",
        truncation=True, max_length= 36)


dataset = dataset.map(tokenize_function, batched=True)
dataset.set_format(type="torch", columns=["input_ids",
        "attention_mask", "label"])
print(dataset)
```

```
Map: 100%|██████████| 2000/2000 [00:00<00:00, 44370.55 examples/s]

Dataset({
    features: ['text', 'label', 'input_ids', 'token_type_ids',
'attention_mask'],
    num_rows: 2000
})
```

The following function splits the BERT dataset `dataset` into three BERT datasets for training, valid, and testing.

```
def train_valid_test_split(dataset, train_ratio=0.8, test_ratio =
        0.1):
    num_sentences = len(dataset)
    train_size = int(num_sentences*train_ratio) +1
    test_size = int(num_sentences*test_ratio) +1
    valid_size = num_sentences - (train_size + test_size)
    train_set = dataset[:train_size]
    train_set = Dataset.from_dict(train_set)
    train_set.set_format(type="torch", columns=["input_ids",
        "attention_mask", "label"])
    test_set = dataset[-test_size:]
    test_set = Dataset.from_dict(test_set)
    test_set.set_format(type="torch", columns=["input_ids",
        "attention_mask", "label"])
    valid_set = dataset[train_size:-test_size]
    valid_set = Dataset.from_dict(valid_set)
    valid_set.set_format(type="torch", columns=["input_ids",
        "attention_mask", "label"])
    train_loader = DataLoader(train_set, batch_size=64, shuffle=True)
    test_loader = DataLoader(test_set, batch_size=64, shuffle=False)
    valid_loader = DataLoader(valid_set, batch_size=64, shuffle=False)
    return train_loader, test_loader, valid_loader
```

```
train_loader, test_loader, valid_loader =
        train_valid_test_split(dataset)
```

You need to implement the class `PrefixTuningForClassification` for the prefix prompt fine-tuning. We first load a pre-trained BERT model specified by `model_name`. The parameter `prefix_length` specifies the length of the prefix prompts we add to the pre-trained BERT model. Specifically, given the input batch `[batch_size, seq_len]`, we input to the embedding layer of the pre-trained BERT model to obtain `[batch_size, seq_len, embed_size]`. We create the prefix prompts P of the size `[prefix_length, embed_size]` and concatenate to the embeddings from the pre-trained BERT to obtain `[batch_size, seq_len + prefix_length, embed_size]`. This concatenation tensor will then be fed to the encoder layers of the pre-trained BERT layer to obtain the last `[batch_size, seq_len + prefix_length, embed_size]`.

We then take mean across the seq_len to obtain `[batch_size, embed_size]` on which we can build up a linear layer for making predictions. Please note that **the parameters to tune include the prefix prompts P** and **the output linear layer**, and you should freeze the parameters of the BERT pre-trained model. Moreover, your code should cover the edge case when `prefix_length=None`. In this case, we do not insert any prefix prompts and we only do fine-tuning for the output linear layer on top.

```
class PrefixTuningForClassification(nn.Module):
    def __init__(self, model_name, prefix_length=None, data_manager =
        None):
        super(PrefixTuningForClassification, self).__init__()

        # Load the pretrained transformer model (BERT-like model)
        self.model = AutoModel.from_pretrained(model_name).to(device)
        self.hidden_size =  self.model.config.hidden_size
        self.prefix_length = prefix_length
        self.num_classes = data_manager.num_classes
        # Insert your code here


    def forward(self, input_ids, attention_mask):
        # Insert your code here
```

```
import torch
import torch.nn as nn
from transformers import AutoModel, AutoTokenizer
import math

class PrefixTuningForClassification(nn.Module):
    def __init__(self, model_name, prefix_length=None,
        data_manager=None):
```

```
        """
        Initializes the PrefixTuningForClassification model.

        Args:
            model_name (str): Name of the pre-trained Transformer
        model (e.g., 'bert-base-uncased').
            prefix_length (int, optional): Length of the prefix
        prompts. If None, no prefix is added.
            data_manager (DataManager): Instance containing dataset
        information such as vocab_size, num_classes, and max_seq_len.
        """
        super(PrefixTuningForClassification, self).__init__()

        # Load the pretrained Transformer model (e.g., BERT)
        self.model = AutoModel.from_pretrained(model_name)
        self.model.to(device)

        # Extract hidden size from the model configuration
        self.hidden_size = self.model.config.hidden_size

        # Set prefix length and number of classes from the data
        manager
        self.prefix_length = prefix_length
        self.num_classes = data_manager.num_classes

        # Freeze all parameters of the pre-trained model to prevent
        them from being updated
        for param in self.model.parameters():
            param.requires_grad = False

        if self.prefix_length is not None:
            # Initialize prefix prompts as learnable parameters
            # Shape: [prefix_length, hidden_size]
            self.prefix_embeddings =
        nn.Parameter(torch.randn(self.prefix_length,
        self.hidden_size))

        # Define the final classification layer
        # This layer maps the averaged embeddings to the number of
        classes
        self.classifier = nn.Linear(self.hidden_size,
        self.num_classes)

    def build(self):
        """
        Prepares the model for training by moving it to the
        appropriate device.
        This method can be expanded if additional setup is required.
```

```
        """
        # Currently, all setup is done in __init__
        pass

    def forward(self, input_ids, attention_mask):
        """
        Defines the forward pass of the model.

        Args:
            input_ids (torch.Tensor): Tensor of shape [batch_size, seq_len] containing token IDs.
            attention_mask (torch.Tensor): Tensor of shape [batch_size, seq_len] containing attention masks.

        Returns:
            torch.Tensor: Logits tensor of shape [batch_size, num_classes].
        """
        batch_size, seq_len = input_ids.size()

        if self.prefix_length is not None:
            # Expand prefix prompts to match the batch size
            # Shape after expansion: [batch_size, prefix_length, hidden_size]
            prefix = self.prefix_embeddings.unsqueeze(0).expand(batch_size, -1, -1)

            # Obtain input embeddings from the pre-trained model's embedding layer
            # Shape: [batch_size, seq_len, hidden_size]
            inputs_embeds = self.model.embeddings(input_ids)

            # Concatenate prefix prompts with input embeddings
            # Shape: [batch_size, prefix_length + seq_len, hidden_size]
            concatenated_embeds = torch.cat((prefix, inputs_embeds), dim=1)

            # Create a new attention mask that accounts for the prefix prompts
            # Prefix tokens have an attention mask of 1
            # Original attention_mask shape: [batch_size, seq_len]
            # New attention_mask shape: [batch_size, prefix_length + seq_len]
            prefix_attention_mask = torch.ones(batch_size, self.prefix_length).to(device)
            concatenated_attention_mask = torch.cat((prefix_attention_mask, attention_mask), dim=1)
```

```
            # Pass the concatenated embeddings and the new attention
            mask to the Transformer encoder
            outputs = self.model(
                inputs_embeds=concatenated_embeds,
                attention_mask=concatenated_attention_mask
            )

            # Extract the last hidden state
            # Shape: [batch_size, prefix_length + seq_len,
            hidden_size]
            last_hidden_state = outputs.last_hidden_state
        else:
            # If no prefix prompts, proceed as standard Transformer
            classification
            outputs = self.model(
                input_ids=input_ids,
                attention_mask=attention_mask
            )
            last_hidden_state = outputs.last_hidden_state

        # Compute the mean of the token embeddings across the sequence
        length
        # Shape: [batch_size, hidden_size]
        # This includes prefix prompts if they were added
        pooled_output = torch.mean(last_hidden_state, dim=1)

        # Pass the pooled output through the classification layer to
        obtain logits
        # Shape: [batch_size, num_classes]
        logits = self.classifier(pooled_output)

        return logits
```

You can use the following `FineTunedBaseTrainer` to train the prompt fine-tuning models.

```
class FineTunedBaseTrainer:
    def __init__(self, model, criterion, optimizer, train_loader,
        val_loader):
        self.model = model
        self.criterion = criterion  #the loss function
        self.optimizer = optimizer  #the optimizer
        self.train_loader = train_loader  #the train loader
        self.val_loader = val_loader  #the valid loader

    #the function to train the model in many epochs
    def fit(self, num_epochs):
```

```
        self.num_batches = len(self.train_loader)

        for epoch in range(num_epochs):
            print(f'Epoch {epoch + 1}/{num_epochs}')
            train_loss, train_accuracy = self.train_one_epoch()
            val_loss, val_accuracy = self.validate_one_epoch()
            print(
                f'{self.num_batches}/{self.num_batches} - train_loss: 
        {train_loss:.4f} - train_accuracy: {train_accuracy*100:.4f}% \
                - val_loss: {val_loss:.4f} - val_accuracy: 
        {val_accuracy*100:.4f}%')

    #train in one epoch, return the train_acc, train_loss
    def train_one_epoch(self):
        self.model.train()
        running_loss, correct, total = 0.0, 0, 0
        for batch in self.train_loader:
            input_ids = batch["input_ids"].to(device)
            attention_mask = batch["attention_mask"].to(device)
            labels = batch["label"].to(device)
            self.optimizer.zero_grad()
            outputs = self.model(input_ids= input_ids, attention_mask=
        attention_mask)
            loss = self.criterion(outputs, labels)
            loss.backward()
            self.optimizer.step()

            running_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
        train_accuracy = correct / total
        train_loss = running_loss / self.num_batches
        return train_loss, train_accuracy

    #evaluate on a loader and return the loss and accuracy
    def evaluate(self, loader):
        self.model.eval()
        loss, correct, total = 0.0, 0, 0
        with torch.no_grad():
            for batch in loader:
                input_ids = batch["input_ids"].to(device)
                labels = batch["label"].to(device)
                attention_mask = batch["attention_mask"].to(device)
```

```
                outputs = self.model(input_ids= input_ids,
        attention_mask= attention_mask)
                loss = self.criterion(outputs, labels)
                loss += loss.item()
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()

        accuracy = correct / total
        loss = loss / len(self.val_loader)
        return loss, accuracy

    #return the val_acc, val_loss, be called at the end of each epoch
    def validate_one_epoch(self):
      val_loss, val_accuracy = self.evaluate(self.val_loader)
      return val_loss, val_accuracy
```

We declare and train the prefix-prompt tuning model. In addition, you need to be patient with this model because it might converge slowly with many epochs.

```
prefix_tuning_model = PrefixTuningForClassification(model_name =
        "bert-base-uncased", prefix_length = 5, data_manager =
        dm).to(device)

if prefix_tuning_model.prefix_length is not None:
  optimizer =
        torch.optim.Adam(list(prefix_tuning_model.classifier.parameters())
        + [prefix_tuning_model.prefix_embeddings], lr=5e-5)
else:
  optimizer =
        torch.optim.Adam(prefix_tuning_model.classifier.parameters(),
        lr=1e-4)
criterion = nn.CrossEntropyLoss()
trainer = FineTunedBaseTrainer(model= prefix_tuning_model,
        criterion=criterion, optimizer=optimizer,
        train_loader=train_loader, val_loader=valid_loader)
trainer.fit(num_epochs=100)
```

```
Epoch 1/100
26/26 - train_loss: 1.7966 - train_accuracy: 16.5522%
- val_loss: 0.8773 - val_accuracy: 16.6667%
Epoch 2/100
26/26 - train_loss: 1.7542 - train_accuracy: 19.2380%
- val_loss: 0.8589 - val_accuracy: 21.7172%
Epoch 3/100
26/26 - train_loss: 1.7190 - train_accuracy: 23.8601%
- val_loss: 0.8476 - val_accuracy: 26.2626%
Epoch 4/100
```

```
26/26 - train_loss: 1.6917 - train_accuracy: 28.2324%
- val_loss: 0.8372 - val_accuracy: 32.3232%
Epoch 5/100
26/26 - train_loss: 1.6776 - train_accuracy: 32.4172%
- val_loss: 0.8295 - val_accuracy: 36.8687%
Epoch 6/100
26/26 - train_loss: 1.6574 - train_accuracy: 35.0406%
- val_loss: 0.8192 - val_accuracy: 41.4141%
Epoch 7/100
26/26 - train_loss: 1.6335 - train_accuracy: 38.4760%
- val_loss: 0.8094 - val_accuracy: 44.4444%
Epoch 8/100
26/26 - train_loss: 1.6178 - train_accuracy: 39.2255%
- val_loss: 0.7982 - val_accuracy: 46.4646%
Epoch 9/100
26/26 - train_loss: 1.5995 - train_accuracy: 40.7870%
- val_loss: 0.7920 - val_accuracy: 46.4646%
Epoch 10/100
26/26 - train_loss: 1.5868 - train_accuracy: 41.7864%
- val_loss: 0.7822 - val_accuracy: 46.9697%
Epoch 11/100
26/26 - train_loss: 1.5754 - train_accuracy: 42.9731%
- val_loss: 0.7730 - val_accuracy: 46.9697%
Epoch 12/100
26/26 - train_loss: 1.5610 - train_accuracy: 43.4728%
- val_loss: 0.7715 - val_accuracy: 47.4747%
Epoch 13/100
26/26 - train_loss: 1.5509 - train_accuracy: 47.0956%
- val_loss: 0.7635 - val_accuracy: 47.9798%
Epoch 14/100
26/26 - train_loss: 1.5447 - train_accuracy: 47.6577%
- val_loss: 0.7624 - val_accuracy: 49.4949%
Epoch 15/100
26/26 - train_loss: 1.5377 - train_accuracy: 49.7189%
- val_loss: 0.7588 - val_accuracy: 52.0202%
Epoch 16/100
26/26 - train_loss: 1.5217 - train_accuracy: 49.9063%
- val_loss: 0.7525 - val_accuracy: 53.0303%
Epoch 17/100
26/26 - train_loss: 1.5820 - train_accuracy: 52.4047%
- val_loss: 0.7518 - val_accuracy: 53.5354%
Epoch 18/100
26/26 - train_loss: 1.4978 - train_accuracy: 51.9051%
```

```
- val_loss: 0.7473 - val_accuracy: 54.0404%
Epoch 19/100
26/26 - train_loss: 1.4993 - train_accuracy: 51.4678%
- val_loss: 0.7416 - val_accuracy: 54.0404%
Epoch 20/100
26/26 - train_loss: 1.4805 - train_accuracy: 51.9051%
- val_loss: 0.7319 - val_accuracy: 53.5354%
Epoch 21/100
26/26 - train_loss: 1.4799 - train_accuracy: 52.2798%
- val_loss: 0.7276 - val_accuracy: 56.5657%
Epoch 22/100
26/26 - train_loss: 1.4791 - train_accuracy: 53.9663%
- val_loss: 0.7217 - val_accuracy: 58.5859%
Epoch 23/100
26/26 - train_loss: 1.4569 - train_accuracy: 57.8389%
- val_loss: 0.7166 - val_accuracy: 60.1010%
Epoch 24/100
26/26 - train_loss: 1.4457 - train_accuracy: 57.4641%
- val_loss: 0.7129 - val_accuracy: 61.1111%
Epoch 25/100
26/26 - train_loss: 1.4454 - train_accuracy: 58.1512%
- val_loss: 0.7129 - val_accuracy: 61.1111%
Epoch 26/100
26/26 - train_loss: 1.4318 - train_accuracy: 58.3385%
- val_loss: 0.7069 - val_accuracy: 64.6465%
Epoch 27/100
26/26 - train_loss: 1.4994 - train_accuracy: 61.0868%
- val_loss: 0.6965 - val_accuracy: 65.1515%
Epoch 28/100
26/26 - train_loss: 1.4105 - train_accuracy: 61.2742%
- val_loss: 0.6933 - val_accuracy: 65.1515%
Epoch 29/100
26/26 - train_loss: 1.4078 - train_accuracy: 61.3367%
- val_loss: 0.6913 - val_accuracy: 64.1414%
Epoch 30/100
26/26 - train_loss: 1.3973 - train_accuracy: 59.9001%
- val_loss: 0.6882 - val_accuracy: 64.6465%
Epoch 31/100
26/26 - train_loss: 1.3943 - train_accuracy: 61.0244%
- val_loss: 0.6848 - val_accuracy: 64.6465%
Epoch 32/100
26/26 - train_loss: 1.3743 - train_accuracy: 61.2117%
- val_loss: 0.6749 - val_accuracy: 65.1515%
```

```
Epoch 33/100
26/26 - train_loss: 1.3664 - train_accuracy: 60.9619%
- val_loss: 0.6666 - val_accuracy: 66.1616%
Epoch 34/100
26/26 - train_loss: 1.3710 - train_accuracy: 62.8982%
- val_loss: 0.6599 - val_accuracy: 67.6768%
Epoch 35/100
26/26 - train_loss: 1.3650 - train_accuracy: 63.7726%
- val_loss: 0.6567 - val_accuracy: 69.6970%
Epoch 36/100
26/26 - train_loss: 1.3419 - train_accuracy: 63.8351%
- val_loss: 0.6553 - val_accuracy: 69.6970%
Epoch 37/100
26/26 - train_loss: 1.3314 - train_accuracy: 64.4597%
- val_loss: 0.6481 - val_accuracy: 71.2121%
Epoch 38/100
26/26 - train_loss: 1.3270 - train_accuracy: 66.7083%
- val_loss: 0.6426 - val_accuracy: 71.7172%
Epoch 39/100
26/26 - train_loss: 1.3310 - train_accuracy: 65.0843%
- val_loss: 0.6360 - val_accuracy: 72.7273%
Epoch 40/100
26/26 - train_loss: 1.3201 - train_accuracy: 65.7714%
- val_loss: 0.6290 - val_accuracy: 72.7273%
Epoch 41/100
26/26 - train_loss: 1.3227 - train_accuracy: 66.3960%
- val_loss: 0.6320 - val_accuracy: 72.2222%
Epoch 42/100
26/26 - train_loss: 1.3270 - train_accuracy: 66.8332%
- val_loss: 0.6280 - val_accuracy: 71.7172%
Epoch 43/100
26/26 - train_loss: 1.2870 - train_accuracy: 69.0194%
- val_loss: 0.6221 - val_accuracy: 73.7374%
Epoch 44/100
26/26 - train_loss: 1.3164 - train_accuracy: 68.6446%
- val_loss: 0.6156 - val_accuracy: 73.7374%
Epoch 45/100
26/26 - train_loss: 1.3032 - train_accuracy: 69.3941%
- val_loss: 0.6155 - val_accuracy: 75.2525%
Epoch 46/100
26/26 - train_loss: 1.2816 - train_accuracy: 69.0194%
- val_loss: 0.6114 - val_accuracy: 76.7677%
Epoch 47/100
```

```
26/26 - train_loss: 1.2757 - train_accuracy: 69.8938%
- val_loss: 0.6127 - val_accuracy: 75.2525%
Epoch 48/100
26/26 - train_loss: 1.2547 - train_accuracy: 69.8938%
- val_loss: 0.6116 - val_accuracy: 76.7677%
Epoch 49/100
26/26 - train_loss: 1.3526 - train_accuracy: 71.0806%
- val_loss: 0.6058 - val_accuracy: 76.2626%
Epoch 50/100
26/26 - train_loss: 1.2515 - train_accuracy: 70.7683%
- val_loss: 0.6004 - val_accuracy: 76.7677%
Epoch 51/100
26/26 - train_loss: 1.2482 - train_accuracy: 71.8926%
- val_loss: 0.5958 - val_accuracy: 77.7778%
Epoch 52/100
26/26 - train_loss: 1.2546 - train_accuracy: 71.3929%
- val_loss: 0.5920 - val_accuracy: 77.7778%
Epoch 53/100
26/26 - train_loss: 1.2356 - train_accuracy: 72.1424%
- val_loss: 0.5877 - val_accuracy: 77.7778%
Epoch 54/100
26/26 - train_loss: 1.2351 - train_accuracy: 71.0181%
- val_loss: 0.5809 - val_accuracy: 77.2727%
Epoch 55/100
26/26 - train_loss: 1.2142 - train_accuracy: 72.5172%
- val_loss: 0.5795 - val_accuracy: 77.7778%
Epoch 56/100
26/26 - train_loss: 1.2214 - train_accuracy: 71.7676%
- val_loss: 0.5728 - val_accuracy: 77.7778%
Epoch 57/100
26/26 - train_loss: 1.1934 - train_accuracy: 72.5796%
- val_loss: 0.5736 - val_accuracy: 78.2828%
Epoch 58/100
26/26 - train_loss: 1.1982 - train_accuracy: 73.2042%
- val_loss: 0.5741 - val_accuracy: 78.2828%
Epoch 59/100
26/26 - train_loss: 1.2049 - train_accuracy: 73.2042%
- val_loss: 0.5685 - val_accuracy: 78.2828%
Epoch 60/100
26/26 - train_loss: 1.1991 - train_accuracy: 72.2049%
- val_loss: 0.5663 - val_accuracy: 79.7980%
Epoch 61/100
26/26 - train_loss: 1.1827 - train_accuracy: 73.8913%
```

```
- val_loss: 0.5608 - val_accuracy: 80.3030%
Epoch 62/100
26/26 - train_loss: 1.1840 - train_accuracy: 73.7664%
- val_loss: 0.5586 - val_accuracy: 79.7980%
Epoch 63/100
26/26 - train_loss: 1.1793 - train_accuracy: 72.7046%
- val_loss: 0.5544 - val_accuracy: 80.3030%
Epoch 64/100
26/26 - train_loss: 1.2102 - train_accuracy: 72.8919%
- val_loss: 0.5482 - val_accuracy: 79.7980%
Epoch 65/100
26/26 - train_loss: 1.1643 - train_accuracy: 73.3916%
- val_loss: 0.5424 - val_accuracy: 78.7879%
Epoch 66/100
26/26 - train_loss: 1.1646 - train_accuracy: 72.7670%
- val_loss: 0.5403 - val_accuracy: 80.8081%
Epoch 67/100
26/26 - train_loss: 1.1268 - train_accuracy: 74.8907%
- val_loss: 0.5411 - val_accuracy: 80.8081%
Epoch 68/100
26/26 - train_loss: 1.1512 - train_accuracy: 73.4541%
- val_loss: 0.5410 - val_accuracy: 79.7980%
Epoch 69/100
26/26 - train_loss: 1.1259 - train_accuracy: 75.2030%
- val_loss: 0.5366 - val_accuracy: 79.7980%
Epoch 70/100
26/26 - train_loss: 1.1457 - train_accuracy: 73.9538%
- val_loss: 0.5298 - val_accuracy: 79.2929%
Epoch 71/100
26/26 - train_loss: 1.1577 - train_accuracy: 75.0156%
- val_loss: 0.5249 - val_accuracy: 78.7879%
Epoch 72/100
26/26 - train_loss: 1.1124 - train_accuracy: 75.4528%
- val_loss: 0.5176 - val_accuracy: 78.7879%
Epoch 73/100
26/26 - train_loss: 1.1133 - train_accuracy: 75.2655%
- val_loss: 0.5182 - val_accuracy: 78.2828%
Epoch 74/100
26/26 - train_loss: 1.1247 - train_accuracy: 75.8901%
- val_loss: 0.5142 - val_accuracy: 78.2828%
Epoch 75/100
26/26 - train_loss: 1.1076 - train_accuracy: 75.5778%
- val_loss: 0.5137 - val_accuracy: 79.7980%
```

```
Epoch 76/100
26/26 - train_loss: 1.0773 - train_accuracy: 75.5778%
- val_loss: 0.5131 - val_accuracy: 78.7879%
Epoch 77/100
26/26 - train_loss: 1.0921 - train_accuracy: 75.8901%
- val_loss: 0.5077 - val_accuracy: 79.2929%
Epoch 78/100
26/26 - train_loss: 1.1060 - train_accuracy: 75.1405%
- val_loss: 0.5003 - val_accuracy: 79.2929%
Epoch 79/100
26/26 - train_loss: 1.1032 - train_accuracy: 73.8289%
- val_loss: 0.4982 - val_accuracy: 80.3030%
Epoch 80/100
26/26 - train_loss: 1.0889 - train_accuracy: 75.0156%
- val_loss: 0.4920 - val_accuracy: 79.7980%
Epoch 81/100
26/26 - train_loss: 1.0743 - train_accuracy: 74.9532%
- val_loss: 0.4931 - val_accuracy: 78.7879%
Epoch 82/100
26/26 - train_loss: 1.0769 - train_accuracy: 75.1405%
- val_loss: 0.4912 - val_accuracy: 80.3030%
Epoch 83/100
26/26 - train_loss: 1.0762 - train_accuracy: 75.2655%
- val_loss: 0.4887 - val_accuracy: 80.3030%
Epoch 84/100
26/26 - train_loss: 1.0528 - train_accuracy: 75.6402%
- val_loss: 0.4851 - val_accuracy: 80.8081%
Epoch 85/100
26/26 - train_loss: 1.0674 - train_accuracy: 76.9519%
- val_loss: 0.4784 - val_accuracy: 81.3131%
Epoch 86/100
26/26 - train_loss: 1.0488 - train_accuracy: 75.6402%
- val_loss: 0.4732 - val_accuracy: 80.8081%
Epoch 87/100
26/26 - train_loss: 1.0311 - train_accuracy: 77.0144%
- val_loss: 0.4713 - val_accuracy: 80.8081%
Epoch 88/100
26/26 - train_loss: 1.0458 - train_accuracy: 77.9513%
- val_loss: 0.4693 - val_accuracy: 81.8182%
Epoch 89/100
26/26 - train_loss: 1.0253 - train_accuracy: 75.9525%
- val_loss: 0.4699 - val_accuracy: 80.3030%
Epoch 90/100
```

```
26/26 - train_loss: 1.0454 - train_accuracy: 75.5778%
- val_loss: 0.4658 - val_accuracy: 81.3131%
Epoch 91/100
26/26 - train_loss: 1.0266 - train_accuracy: 76.3898%
- val_loss: 0.4629 - val_accuracy: 81.3131%
Epoch 92/100
26/26 - train_loss: 1.0014 - train_accuracy: 76.6396%
- val_loss: 0.4593 - val_accuracy: 81.3131%
Epoch 93/100
26/26 - train_loss: 1.0273 - train_accuracy: 76.4522%
- val_loss: 0.4563 - val_accuracy: 81.3131%
Epoch 94/100
26/26 - train_loss: 1.0082 - train_accuracy: 75.9525%
- val_loss: 0.4563 - val_accuracy: 81.8182%
Epoch 95/100
26/26 - train_loss: 1.0159 - train_accuracy: 76.0775%
- val_loss: 0.4502 - val_accuracy: 81.8182%
Epoch 96/100
26/26 - train_loss: 1.0058 - train_accuracy: 77.7639%
- val_loss: 0.4460 - val_accuracy: 81.3131%
Epoch 97/100
26/26 - train_loss: 1.0012 - train_accuracy: 77.5765%
- val_loss: 0.4430 - val_accuracy: 80.8081%
Epoch 98/100
26/26 - train_loss: 0.9897 - train_accuracy: 76.7645%
- val_loss: 0.4434 - val_accuracy: 80.8081%
Epoch 99/100
26/26 - train_loss: 0.9977 - train_accuracy: 77.2017%
- val_loss: 0.4381 - val_accuracy: 81.3131%
Epoch 100/100
26/26 - train_loss: 1.0084 - train_accuracy: 77.7639%
- val_loss: 0.4393 - val_accuracy: 80.8081%
```

```python
from sklearn.metrics import accuracy_score, classification_report

# Set the model to evaluation mode
prefix_tuning_model.eval()

all_preds = []
all_labels = []

with torch.no_grad():
    for batch in test_loader:
        # Move input data to the appropriate device
```

```
            input_ids = batch['input_ids'].to(device)
            attention_mask = batch['attention_mask'].to(device)
            labels = batch['label'].to(device)

            # Forward pass to get outputs/logits
            outputs = prefix_tuning_model(input_ids, attention_mask)

            # Get the predicted class by taking the argmax
            _, preds = torch.max(outputs, dim=1)

            # Append predictions and true labels to the lists
            all_preds.extend(preds.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())

    # Compute the overall accuracy
    test_accuracy = accuracy_score(all_labels, all_preds)

    print(f"The Test Accuracy we obtain is: {test_accuracy * 100}%")
```

```
The Test Accuracy we obtain is: 79.1044776119403%
```

Question 4.3

For any models defined in the previous questions (of all parts), you are free to fine-tune hyperparameters, e.g., `optimizer`, `learning_rate`, `state_sizes`, such that you get a best model, i.e., the one with the highest accuracy on the test set. You will need to report (i) what is your best model, (ii) its accuracy on the test set, and (iii) the values of its hyperparameters. Note that you must report your best model's accuracy with rounding to 4 decimal places, i.e., 0.xxxx. You will also need to upload your best model (or provide us with the link to download your best model). The assessment will be based on your best model's accuracy, with up to 9 marks available, specifically:

- The best accuracy ≥ 0.97: 10 marks
- 0.97 > The best accuracy ≥ 0.92: 7 marks
- 0.92 > The best accuracy ≥ 0.85: 4 marks
- The best accuracy < 0.85: 0 mark

For this question, you can put below the code to train the best model. In this case, you need to show your code and the evidence of running regarding the best model. Moreover, if you save the best model, you need to provide the link to download the best model, the code to load the best model, and then evaluate on the test set.

[10 marks]

Give your answer here.

(i) What is your best model?

(ii) The accuracy of your best model on the test set

(iii) The values of the hyperparameters of your best model

(iv) The link to download your best model

Answer

(i) --> I have used the RNN Model from 3.2.1 as my best model

(ii) --> The accuracy it gets is 0.98

(iii) --> There was no need to do any modifications on my model but rather I just went with these paramters

```
cell_type='gru',
state_sizes=[64, 128],
output_type='max',
data_manager= dm,
run_mode='init-fine-tune'
```

(iv) --> Link to download (https://drive.google.com/file/d/1908YpaD1CZGtTcnarlF69K2iX7ry-9Xn/view?usp=drive_link)

```
import os
import torch
import random
import requests
import pandas as pd
import numpy as np
import torch.nn as nn
from torch.utils.data import DataLoader
from torch.nn.utils.rnn import pad_sequence
from transformers import BertTokenizer
import os
from six.moves.urllib.request import urlretrieve # type: ignore
from sklearn import preprocessing
import matplotlib.pyplot as plt
```

```
plt.style.use('ggplot')

def seed_all(seed=1029):
    random.seed(seed)
    os.environ['PYTHONHASHSEED'] = str(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)  # if you are using multi-GPU.
    torch.backends.cudnn.benchmark = False
    torch.backends.cudnn.deterministic = True
seed_all(seed=1234)

class DataManager:
    """
    This class manages and preprocesses a simple text dataset for a
        sentence classification task.


    Attributes:
        verbose (bool): Controls verbosity for printing information
        during data processing.
        max_sentence_len (int): The maximum length of a sentence in
        the dataset.
        str_questions (list): A list to store the string
        representations of the questions in the dataset.
        str_labels (list): A list to store the string representations
        of the labels in the dataset.
        numeral_labels (list): A list to store the numerical
        representations of the labels in the dataset.
        numeral_data (list): A list to store the numerical
        representations of the questions in the dataset.
        random_state (int): Seed value for random number generation to
        ensure reproducibility.
            Set this value to a specific integer to reproduce the same
        random sequence every time. Defaults to 6789.
        random (np.random.RandomState): Random number generator object
        initialized with the given random_state.
            It is used for various random operations in the class.


    Methods:
        maybe_download(dir_name, file_name, url, verbose=True):
            Downloads a file from a given URL if it does not exist in
        the specified directory.
            The directory and file are created if they do not exist.


        read_data(dir_name, file_names):
            Reads data from files in a directory, preprocesses it, and
        computes the maximum sentence length.
            Each file is expected to contain rows in the format "
        <label>:<question>".
```

```
        The labels and questions are stored as string
    representations.


    manipulate_data():
        Performs data manipulation by tokenizing, numericalizing,
    and padding the text data.
        The questions are tokenized and converted into numerical
    sequences using a tokenizer.
        The sequences are padded or truncated to the maximum
    sequence length.


    train_valid_test_split(train_ratio=0.9):
        Splits the data into training, validation, and test sets
    based on a given ratio.
        The data is randomly shuffled, and the specified ratio is
    used to determine the size of the training set.
        The string questions, numerical data, and numerical labels
    are split accordingly.
        TensorFlow `Dataset` objects are created for the training
    and validation sets.



"""

def __init__(self, verbose=True, random_state=6789):
    self.verbose = verbose
    self.max_sentence_len = 0
    self.str_questions = list()
    self.str_labels = list()
    self.numeral_labels = list()
    self.maxlen = None
    self.numeral_data = list()
    self.random_state = random_state
    self.random = np.random.RandomState(random_state)

@staticmethod
def maybe_download(dir_name, file_name, url, verbose=True):
    if not os.path.exists(dir_name):
        os.mkdir(dir_name)
    if not os.path.exists(os.path.join(dir_name, file_name)):
        urlretrieve(url + file_name, os.path.join(dir_name,
    file_name))
    if verbose:
        print("Downloaded successfully {}".format(file_name))

def read_data(self, dir_name, file_names):
    self.str_questions = list()
```

```
        self.str_labels = list()
        for file_name in file_names:
            file_path= os.path.join(dir_name, file_name)
            with open(file_path, "r", encoding="latin-1") as f:
                for row in f:
                    row_str = row.split(":")
                    label, question = row_str[0], row_str[1]
                    question = question.lower()
                    self.str_labels.append(label)
                    self.str_questions.append(question[0:-1])
                    if self.max_sentence_len <
len(self.str_questions[-1]):
                        self.max_sentence_len =
len(self.str_questions[-1])

        # turns labels into numbers
        le = preprocessing.LabelEncoder()
        le.fit(self.str_labels)
        self.numeral_labels = np.array(le.transform(self.str_labels))
        self.str_classes = le.classes_
        self.num_classes = len(self.str_classes)
        if self.verbose:
            print("\nSample questions and corresponding labels... \n")
            print(self.str_questions[0:5])
            print(self.str_labels[0:5])

    def manipulate_data(self):
        self.tokenizer = BertTokenizer.from_pretrained('bert-base-
uncased')
        vocab = self.tokenizer.get_vocab()
        self.word2idx = {w: i for i, w in enumerate(vocab)}
        self.idx2word = {i:w for w,i in self.word2idx.items()}
        self.vocab_size = len(self.word2idx)


        token_ids = []
        num_seqs = []
        for text in self.str_questions:  # iterate over the list of
text
          text_seqs = self.tokenizer.tokenize(str(text))  # tokenize
each text individually
          # Convert tokens to IDs
          token_ids = self.tokenizer.convert_tokens_to_ids(text_seqs)
          # Convert token IDs to a tensor of indices using your
word2idx mapping
          seq_tensor = torch.LongTensor(token_ids)
```

```
            num_seqs.append(seq_tensor)  # append the tensor for each
sequence

        # Pad the sequences and create a tensor
        if num_seqs:
            self.numeral_data = pad_sequence(num_seqs, batch_first=True)
# Pads to max length of the sequences
            self.num_sentences, self.maxlen = self.numeral_data.shape

    def train_valid_test_split(self, train_ratio=0.8, test_ratio =
0.1):
        train_size = int(self.num_sentences*train_ratio) +1
        test_size = int(self.num_sentences*test_ratio) +1
        valid_size = self.num_sentences - (train_size + test_size)
        data_indices = list(range(self.num_sentences))
        random.shuffle(data_indices)
        self.train_str_questions = [self.str_questions[i] for i in
data_indices[:train_size]]
        self.train_numeral_labels =
self.numeral_labels[data_indices[:train_size]]
        train_set_data = self.numeral_data[data_indices[:train_size]]
        train_set_labels =
self.numeral_labels[data_indices[:train_size]]
        train_set_labels = torch.from_numpy(train_set_labels)
        train_set = torch.utils.data.TensorDataset(train_set_data,
train_set_labels)
        self.test_str_questions = [self.str_questions[i] for i in
data_indices[-test_size:]]
        self.test_numeral_labels = self.numeral_labels[data_indices[-
test_size:]]
        test_set_data = self.numeral_data[data_indices[-test_size:]]
        test_set_labels = self.numeral_labels[data_indices[-
test_size:]]
        test_set_labels = torch.from_numpy(test_set_labels)
        test_set = torch.utils.data.TensorDataset(test_set_data,
test_set_labels)
        self.valid_str_questions = [self.str_questions[i] for i in
data_indices[train_size:-test_size]]
        self.valid_numeral_labels =
self.numeral_labels[data_indices[train_size:-test_size]]
        valid_set_data = self.numeral_data[data_indices[train_size:-
test_size]]
        valid_set_labels =
self.numeral_labels[data_indices[train_size:-test_size]]
        valid_set_labels = torch.from_numpy(valid_set_labels)
        valid_set = torch.utils.data.TensorDataset(valid_set_data,
valid_set_labels)
        self.train_loader = DataLoader(train_set, batch_size=64,
shuffle=True) # you can change the batch size if needed
```

```
        self.test_loader = DataLoader(test_set, batch_size=64,
        shuffle=False) # you can change the batch size if needed
        self.valid_loader = DataLoader(valid_set, batch_size=64,
        shuffle=False) # you can change the batch size if needed
```

```
print('Loading data...')
DataManager.maybe_download("data", "train_2000.label",
        "http://cogcomp.org/Data/QA/QC/")


dm = DataManager()
dm.read_data("data/", ["train_2000.label"])



dm.manipulate_data()
dm.train_valid_test_split(train_ratio=0.8, test_ratio = 0.1)
```

```
Loading data...
Downloaded successfully train_2000.label

Sample questions and corresponding labels...

['manner how did serfdom develop in and then leave russia ?', 'cremat
what films featured the character popeye doyle ?', "manner how can i
find a list of celebrities ' real names ?", 'animal what fowl grabs
the spotlight after the chinese year of the monkey ?', 'exp what is
the full form of .com ?']
['DESC', 'ENTY', 'DESC', 'ENTY', 'ABBR']
```

```
for x, y in dm.train_loader:
    print(x.shape, y.shape)
    break
```

```
torch.Size([64, 36]) torch.Size([64])
```

```
#device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        -- i dont need this line
import torch
# Check if MPS is available
if torch.backends.mps.is_available():
    device = torch.device("mps")
else:
    device = torch.device("cpu")


class BaseTrainer:
    def __init__(self, model, criterion, optimizer, train_loader,
        val_loader):
        self.model = model
```

```
        self.criterion = criterion  #the loss function
        self.optimizer = optimizer  #the optimizer
        self.train_loader = train_loader  #the train loader
        self.val_loader = val_loader  #the valid loader

    #the function to train the model in many epochs
    def fit(self, num_epochs):
        self.num_batches = len(self.train_loader)

        for epoch in range(num_epochs):
            print(f'Epoch {epoch + 1}/{num_epochs}')
            train_loss, train_accuracy = self.train_one_epoch()
            val_loss, val_accuracy = self.validate_one_epoch()
            print(
                f'{self.num_batches}/{self.num_batches} - train_loss:
        {train_loss:.4f} - train_accuracy: {train_accuracy*100:.4f}% \
                - val_loss: {val_loss:.4f} - val_accuracy:
        {val_accuracy*100:.4f}%')

    #train in one epoch, return the train_acc, train_loss
    def train_one_epoch(self):
        self.model.train()
        running_loss, correct, total = 0.0, 0, 0
        for i, data in enumerate(self.train_loader):
            inputs, labels = data
            inputs, labels = inputs.to(device), labels.to(device)
            self.optimizer.zero_grad()
            outputs = self.model(inputs)
            loss = self.criterion(outputs, labels)
            loss.backward()
            self.optimizer.step()

            running_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
        train_accuracy = correct / total
        train_loss = running_loss / self.num_batches
        return train_loss, train_accuracy

    #evaluate on a loader and return the loss and accuracy
    def evaluate(self, loader):
        self.model.eval()
        loss, correct, total = 0.0, 0, 0
```

```
        with torch.no_grad():
            for data in loader:
                inputs, labels = data
                inputs, labels = inputs.to(device), labels.to(device)
                outputs = self.model(inputs)
                loss = self.criterion(outputs, labels)
                loss += loss.item()
                _, predicted = torch.max(outputs.data, 1)
                total += labels.size(0)
                correct += (predicted == labels).sum().item()

        accuracy = correct / total
        loss = loss / len(self.val_loader)
        return loss, accuracy

    #return the val_acc, val_loss, be called at the end of each epoch
    def validate_one_epoch(self):
      val_loss, val_accuracy = self.evaluate(self.val_loader)
      return val_loss, val_accuracy

class BaseRNN(nn.Module):
    def __init__(self, cell_type='gru', embed_size=128, state_sizes=
        [128, 128], output_type="mean", data_manager=None):
        super().__init__()
        self.cell_type = cell_type                    # Type of RNN
        cell: 'simple_rnn', 'gru', or 'lstm'
        self.state_sizes = state_sizes                # List of hidden
        sizes for each RNN layer
        self.embed_size = embed_size                  # Dimension of
        word embeddings
        self.output_type = output_type                # Output strategy:
        'last_state', 'mean', or 'max'
        self.data_manager = data_manager              # Data manager
        containing dataset information
        self.vocab_size = self.data_manager.vocab_size  # Size of the
        vocabulary

    # Static method to return the corresponding RNN layer based on
        cell_type
    @staticmethod
    def get_layer(cell_type='gru', input_size=128, state_size=128):
        if cell_type == 'gru':
            return nn.GRU(input_size=input_size,
        hidden_size=state_size, batch_first=True)
        elif cell_type == 'lstm':
            return nn.LSTM(input_size=input_size,
        hidden_size=state_size, batch_first=True)
        else:  # 'simple_rnn'
```

```
            return nn.RNN(input_size=input_size,
        hidden_size=state_size, batch_first=True)

    def build(self):
        # Embedding layer to convert word indices to embeddings
        self.embed = nn.Embedding(self.vocab_size, self.embed_size)

        # ModuleList to hold multiple RNN layers
        self.rnn_layers = nn.ModuleList()
        input_size = self.embed_size  # Initial input size is the
        embedding size

        # Create RNN layers based on state_sizes
        for state_size in self.state_sizes:
            rnn_layer = self.get_layer(self.cell_type,
        input_size=input_size, state_size=state_size)
            self.rnn_layers.append(rnn_layer)
            input_size = state_size  # Output size of current layer is
        input size for next layer

        # Fully connected layer for classification
        self.fc = nn.Linear(self.state_sizes[-1],
        self.data_manager.num_classes)

    def forward(self, x):
        # x: input tensor of shape [batch_size, seq_len]
        # Pass input through embedding layer
        e = self.embed(x)  # e: [batch_size, seq_len, embed_size]

        # Pass embeddings through the stacked RNN layers
        output = e  # Initial input to the first RNN layer
        for rnn_layer in self.rnn_layers:
            if self.cell_type == 'lstm':
                output, (h_n, c_n) = rnn_layer(output)
            else:
                output, h_n = rnn_layer(output)

        # Obtain the representation based on output_type
        if self.output_type == "last_state":
            # For last_state, use the last hidden state h_n
            # h_n shape: [num_layers * num_directions, batch_size,
        hidden_size]
            h = h_n[-1]  # Get the last layer's hidden state; shape:
        [batch_size, hidden_size]
        elif self.output_type == "mean":
            # For mean, average over the sequence length dimension
```

```
            h = torch.mean(output, dim=1)  # h: [batch_size,
hidden_size]
        elif self.output_type == "max":
            # For max, take the maximum over the sequence length
dimension
            h, _ = torch.max(output, dim=1)  # h: [batch_size,
hidden_size]
        else:
            raise ValueError("Invalid output_type. Choose from
'last_state', 'mean', or 'max'.")


        logits = self.fc(h)  # logits: [batch_size, num_classes]
        return logits

class RNN(BaseRNN):
    def __init__(self, cell_type='gru', embed_size=128, state_sizes=
[128, 128], output_type='mean', data_manager=None,
                 run_mode='scratch', embed_model='glove-wiki-gigaword-
100'):
        super().__init__(cell_type, embed_size, state_sizes,
output_type, data_manager)
        self.run_mode = run_mode
        self.embed_model = embed_model
        if not os.path.exists("embeddings"):
            os.makedirs("embeddings")
        self.embed_path = "embeddings/E.npy"
        if self.run_mode != 'scratch':
            self.embed_size = int(self.embed_model.split("-")[-1])
        self.word2idx = data_manager.word2idx
        self.word2vect = None
        self.embed_matrix = np.zeros((self.vocab_size,
self.embed_size))
        # Call self.build() after all attributes are set
        self.build()



    def build_embedding_matrix(self):
        # Check if the embedding matrix already exists
        if os.path.exists(self.embed_path):
            print(f"Loading embedding matrix from {self.embed_path}")
            self.embed_matrix = np.load(self.embed_path)
        else:
            # Download the pretrained embedding model
            print(f"Downloading embedding model
{self.embed_model}...")
            self.word2vect = api.load(self.embed_model)
            print("Building embedding matrix...")
            # Initialize the embedding matrix
```

```
            for word, idx in self.word2idx.items():
                if word in self.word2vect:
                    # If the word is in the pretrained embeddings, use
its vector
                    self.embed_matrix[idx] = self.word2vect[word]
                else:
                    # Otherwise, initialize a random vector
                    self.embed_matrix[idx] = np.random.uniform(-0.25,
0.25, self.embed_size)
            # Save the embedding matrix for future use
            np.save(self.embed_path, self.embed_matrix)
            print(f"Saved embedding matrix to {self.embed_path}")

    def build(self):
        # Build the embedding layer based on run_mode
        if self.run_mode == 'scratch':
            # Initialize the embedding layer from scratch
            self.embed = nn.Embedding(self.vocab_size,
self.embed_size)
        else:
            # Build the embedding matrix using the pretrained
embeddings
            self.build_embedding_matrix()
            # Create the embedding layer and load the pretrained
weights
            self.embed = nn.Embedding(self.vocab_size,
self.embed_size)

self.embed.weight.data.copy_(torch.from_numpy(self.embed_matrix))
            if self.run_mode == 'init-only':
                # Freeze the embedding layer weights
                self.embed.weight.requires_grad = False
            elif self.run_mode == 'init-fine-tune':
                # Allow the embedding layer to be fine-tuned
                self.embed.weight.requires_grad = True
            else:
                raise ValueError("Invalid run_mode. Choose from
'scratch', 'init-only', or 'init-fine-tune'.")
        # Build the RNN layers and the fully connected layer
        self.rnn_layers = nn.ModuleList()
        input_size = self.embed_size
        for state_size in self.state_sizes:
            rnn_layer = self.get_layer(self.cell_type,
input_size=input_size, state_size=state_size)
            self.rnn_layers.append(rnn_layer)
            input_size = state_size
```

```
        # Fully connected layer for classification
        self.fc = nn.Linear(self.state_sizes[-1],
        self.data_manager.num_classes)

# Insert your code here
print("\nTraining RNN with run_mode 'init-fine-tune'")

# Instantiate the RNN model
rnn_init_fine_tune = RNN(
    cell_type='gru',
    state_sizes=[64, 128],
    output_type='max',
    data_manager= dm,
    run_mode='init-fine-tune',
    embed_model='glove-wiki-gigaword-100'
)
rnn_init_fine_tune.to(device)

# Define the loss function
criterion_init_fine_tune = nn.CrossEntropyLoss()

# Define the optimizer
optimizer_init_fine_tune =
        torch.optim.Adam(rnn_init_fine_tune.parameters(), lr=0.001)

# Initialize the trainer
trainer_init_fine_tune = BaseTrainer(
    model=rnn_init_fine_tune,
    criterion=criterion_init_fine_tune,
    optimizer=optimizer_init_fine_tune,
    train_loader=dm.train_loader,
    val_loader=dm.valid_loader
)

# Train the model
trainer_init_fine_tune.fit(num_epochs=30)

# Evaluate on the validation set
val_loss_init_fine_tune, val_accuracy_init_fine_tune =
        trainer_init_fine_tune.validate_one_epoch()
print(f"Validation Loss: {val_loss_init_fine_tune:.4f} - Validation
        Accuracy: {val_accuracy_init_fine_tune*100:.2f}%")
```

```
Training RNN with run_mode 'init-fine-tune'
Loading embedding matrix from embeddings/E.npy
```

```
Epoch 1/30
26/26 - train_loss: 1.5941 - train_accuracy: 41.9738%
- val_loss: 0.5975 - val_accuracy: 54.0404%
Epoch 2/30
26/26 - train_loss: 1.0853 - train_accuracy: 68.8944%
- val_loss: 0.3698 - val_accuracy: 90.4040%
Epoch 3/30
26/26 - train_loss: 0.4852 - train_accuracy: 90.6933%
- val_loss: 0.1171 - val_accuracy: 93.4343%
Epoch 4/30
26/26 - train_loss: 0.2300 - train_accuracy: 93.3791%
- val_loss: 0.0450 - val_accuracy: 93.9394%
Epoch 5/30
26/26 - train_loss: 0.1311 - train_accuracy: 95.1280%
- val_loss: 0.0265 - val_accuracy: 93.9394%
Epoch 6/30
26/26 - train_loss: 0.0967 - train_accuracy: 96.6896%
- val_loss: 0.0159 - val_accuracy: 95.4545%
Epoch 7/30
26/26 - train_loss: 0.0617 - train_accuracy: 98.3760%
- val_loss: 0.0107 - val_accuracy: 94.9495%
Epoch 8/30
26/26 - train_loss: 0.0510 - train_accuracy: 98.5634%
- val_loss: 0.0064 - val_accuracy: 96.4646%
Epoch 9/30
26/26 - train_loss: 0.0397 - train_accuracy: 98.9382%
- val_loss: 0.0050 - val_accuracy: 96.9697%
Epoch 10/30
26/26 - train_loss: 0.0241 - train_accuracy: 99.4379%
- val_loss: 0.0037 - val_accuracy: 95.9596%
Epoch 11/30
26/26 - train_loss: 0.0303 - train_accuracy: 99.0631%
- val_loss: 0.0028 - val_accuracy: 97.4747%
Epoch 12/30
26/26 - train_loss: 0.0138 - train_accuracy: 99.6877%
- val_loss: 0.0022 - val_accuracy: 97.4747%
Epoch 13/30
26/26 - train_loss: 0.0075 - train_accuracy: 99.9375%
- val_loss: 0.0020 - val_accuracy: 96.9697%
Epoch 14/30
26/26 - train_loss: 0.0110 - train_accuracy: 99.7502%
- val_loss: 0.0016 - val_accuracy: 97.9798%
Epoch 15/30
```

```
26/26 - train_loss: 0.0046 - train_accuracy: 99.9375%
- val_loss: 0.0014 - val_accuracy: 97.9798%
Epoch 16/30
26/26 - train_loss: 0.0036 - train_accuracy: 100.0000%
- val_loss: 0.0014 - val_accuracy: 96.9697%
Epoch 17/30
26/26 - train_loss: 0.0048 - train_accuracy: 99.8751%
- val_loss: 0.0012 - val_accuracy: 96.9697%
Epoch 18/30
26/26 - train_loss: 0.0091 - train_accuracy: 99.7502%
- val_loss: 0.0010 - val_accuracy: 97.9798%
Epoch 19/30
26/26 - train_loss: 0.0031 - train_accuracy: 100.0000%
- val_loss: 0.0010 - val_accuracy: 96.9697%
Epoch 20/30
26/26 - train_loss: 0.0020 - train_accuracy: 100.0000%
- val_loss: 0.0008 - val_accuracy: 97.9798%
Epoch 21/30
26/26 - train_loss: 0.0016 - train_accuracy: 100.0000%
- val_loss: 0.0007 - val_accuracy: 97.9798%
Epoch 22/30
26/26 - train_loss: 0.0015 - train_accuracy: 100.0000%
- val_loss: 0.0007 - val_accuracy: 97.9798%
Epoch 23/30
26/26 - train_loss: 0.0013 - train_accuracy: 100.0000%
- val_loss: 0.0006 - val_accuracy: 97.9798%
Epoch 24/30
26/26 - train_loss: 0.0012 - train_accuracy: 100.0000%
- val_loss: 0.0006 - val_accuracy: 97.9798%
Epoch 25/30
26/26 - train_loss: 0.0011 - train_accuracy: 100.0000%
- val_loss: 0.0005 - val_accuracy: 97.9798%
Epoch 26/30
26/26 - train_loss: 0.0010 - train_accuracy: 100.0000%
- val_loss: 0.0005 - val_accuracy: 97.9798%
Epoch 27/30
26/26 - train_loss: 0.0010 - train_accuracy: 100.0000%
- val_loss: 0.0005 - val_accuracy: 97.9798%
Epoch 28/30
26/26 - train_loss: 0.0009 - train_accuracy: 100.0000%
- val_loss: 0.0004 - val_accuracy: 97.9798%
Epoch 29/30
26/26 - train_loss: 0.0009 - train_accuracy: 100.0000%
```

```
- val_loss: 0.0004 - val_accuracy: 97.9798%
Epoch 30/30
26/26 - train_loss: 0.0008 - train_accuracy: 100.0000%
- val_loss: 0.0004 - val_accuracy: 97.9798%
Validation Loss: 0.0004 - Validation Accuracy: 97.98%
```

```
trainer_init_fine_tune.fit(num_epochs=30)
```

```
test_loss, test_acc = trainer_init_fine_tune.evaluate(dm.test_loader)
print(f'test_loss: {test_loss:.4f} - test_accuracy:
        {test_acc*100:.4f}%')
```

```
test_loss: 0.4790 - test_accuracy: 98.0100%
```

```
torch.save(rnn_init_fine_tune.state_dict(),'final_best_model.pth')
rnn_init_fine_tune.load_state_dict(torch.load('final_best_model.pth',
        map_location=device))
test_loss, test_acc = trainer_init_fine_tune.evaluate(dm.test_loader)
print(f'test_loss: {test_loss:.4f} - test_accuracy:
        {test_acc*100:.4f}%')
```

```
test_loss: 0.4790 - test_accuracy: 98.0100%

/var/folders/gm/2ndchprn0czbhd7f3zbnh2f80000gn/T/ipykernel_77091/24528
FutureWarning: You are using `torch.load` with `weights_only=False`
(the current default value), which uses the default pickle module
implicitly. It is possible to construct malicious pickle data which
will execute arbitrary code during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-
models for more details). In a future release, the default value for
`weights_only` will be flipped to `True`. This limits the functions
that could be executed during unpickling. Arbitrary objects will no
longer be allowed to be loaded via this mode unless they are
explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control
of the loaded file. Please open an issue on GitHub for any issues
related to this experimental feature.

rnn_init_fine_tune.load_state_dict(torch.load('final_best_model.pth',
map_location=device))
```

GOOD LUCK WITH YOUR ASSIGNMENT 2!

END OF ASSIGNMENT