Course Title: Neural Networks

Course Code: CSE425

Project Title: Implementation of Bangla ChatBot Using Transformer-Based Models

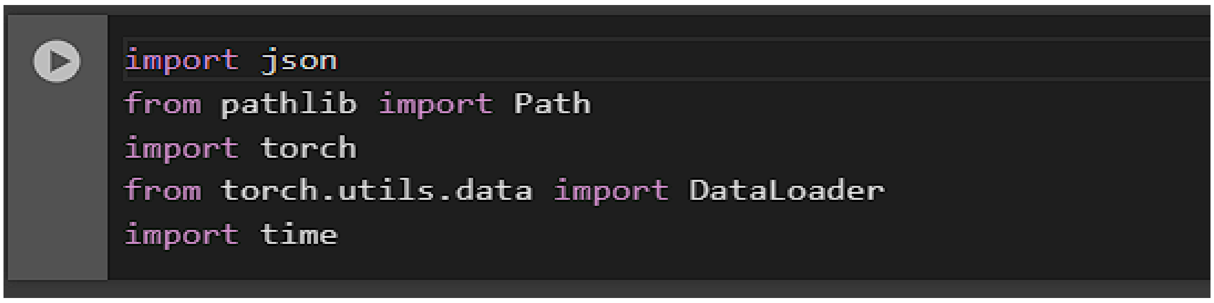
BRAC UNIVERSITY

Inspiring Excellence

Group No-03

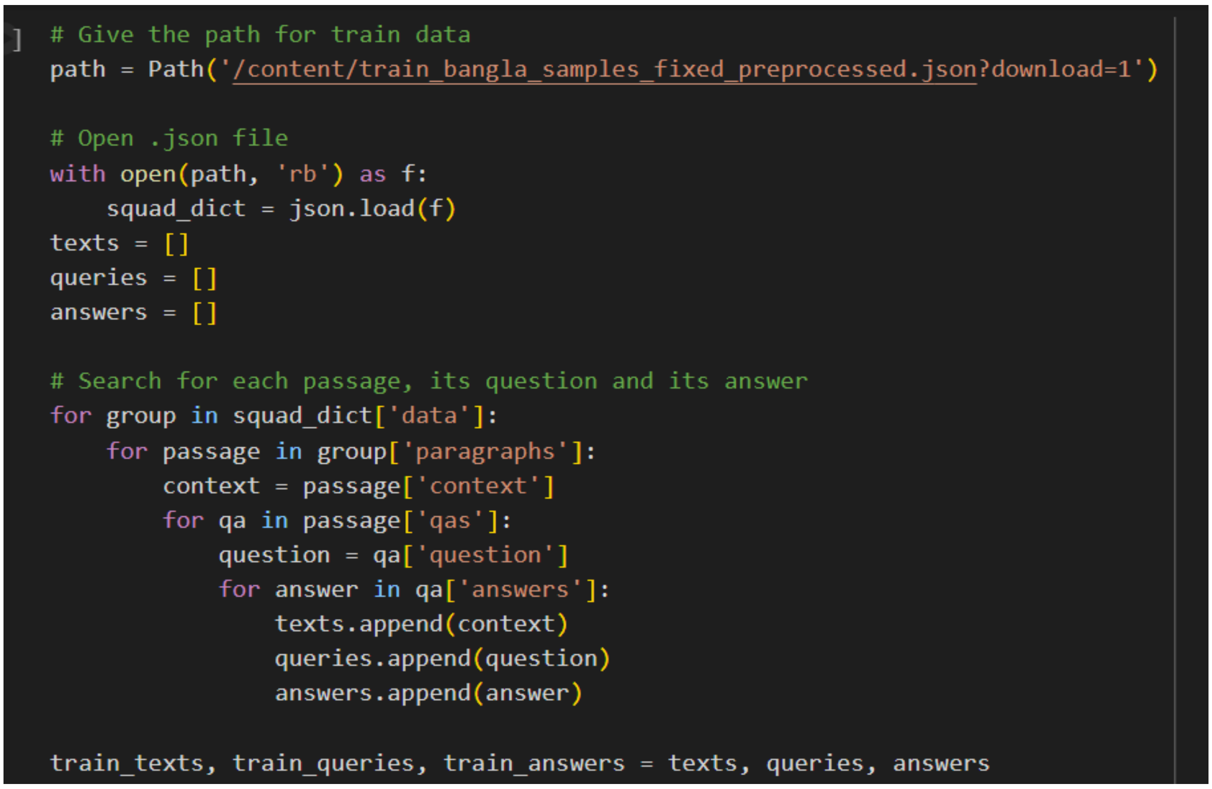
Name	Student ID	Section
Farhan Faruk	20301137	03
Aunon Halder	20301133	03
H.M Sarwer Alam	20301224	03
Saif Mokarrom	20301121	03
Shudeb Ghosh Barno	20301222	03

Code Documentation



```
import json
from pathlib import Path
import torch
from torch.utils.data import DataLoader
import time
```

Here, imports the json module, which provides methods for working with JSON (JavaScript Object Notation) data.Then. imports the Path class from the pathlib module. Path is a convenient way to work with file system paths.imports the PyTorch library, a popular deep learning framework for building and training neural networks.imports the DataLoader class from the torch.utils.data module. DataLoader is used to load and iterate over data in batches during the training of machine learning models.This line imports the time module, which provides various time-related functions. It is commonly used for measuring and controlling the execution time of code.



```
# Give the path for train data
path = Path('/content/train_bangla_samples_fixed_preprocessed.json?download=1')

# Open .json file
with open(path, 'rb') as f:
    squad_dict = json.load(f)
texts = []
queries = []
answers = []

# Search for each passage, its question and its answer
for group in squad_dict['data']:
    for passage in group['paragraphs']:
        context = passage['context']
        for qa in passage['qas']:
            question = qa['question']
            for answer in qa['answers']:
                texts.append(context)
                queries.append(question)
                answers.append(answer)

train_texts, train_queries, train_answers = texts, queries, answers
```

In this part of the code, we enrolled the train data.

This line creates a Path object named path, representing the file path '/content/train_bangla_samples_fixed_preprocessed.json?download=1'. Then ‘with’ statement to open the JSON file located at the specified path in binary read mode ('rb'). It then loads the contents of the JSON file into a Python dictionary named squad_dict using json.load(). After that, these lines initialize empty lists (texts, queries, and answers) to store information extracted from the JSON file.

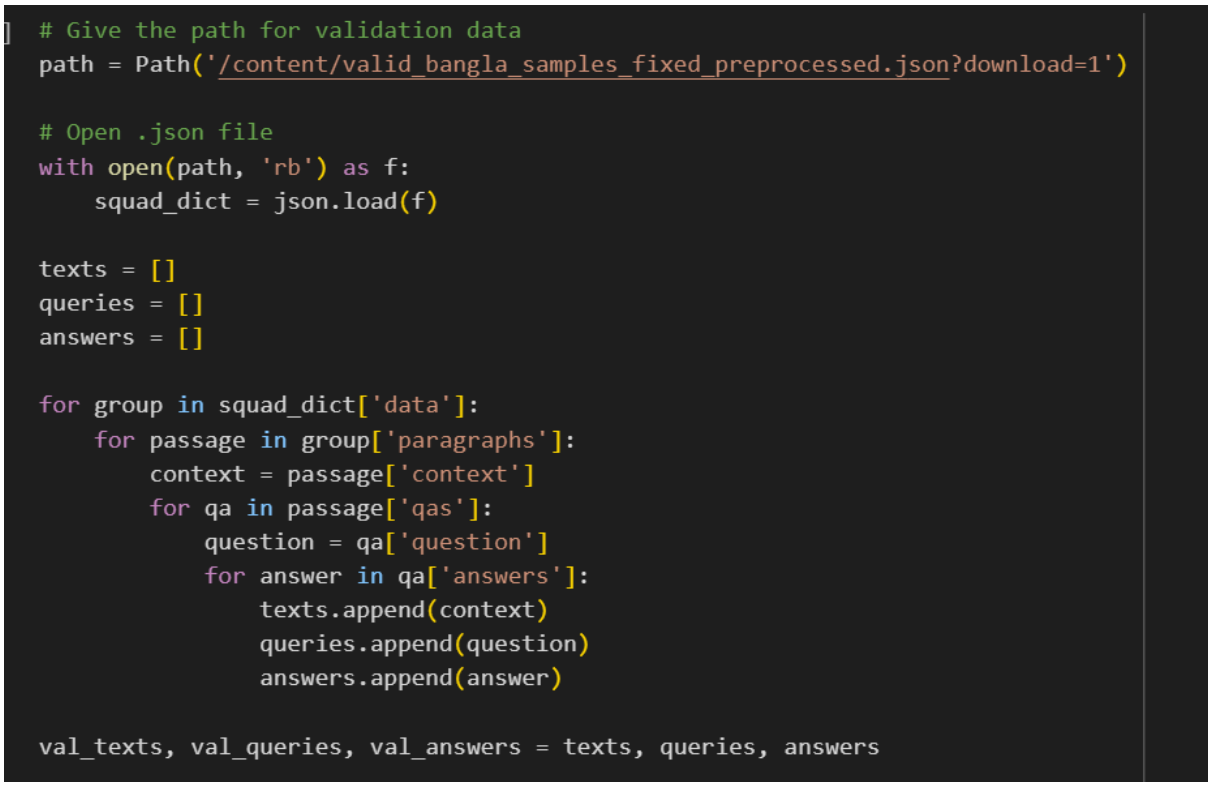
Assigns the contents of the three lists to new variables:

train_texts: stores the list of context texts.

train_queries: stores the list of question texts.

train_answers: stores the list of answer texts.

Overall this efficiently processes the training data by extracting the relevant information (context, question, and answer) and storing it in structured lists. These organized data structures will be essential for further tasks in training and developing the question-answering system.



```
# Give the path for validation data
path = Path('/content/valid_bangla_samples_fixed_preprocessed.json?download=1')

# Open .json file
with open(path, 'rb') as f:
    squad_dict = json.load(f)

texts = []
queries = []
answers = []

for group in squad_dict['data']:
    for passage in group['paragraphs']:
        context = passage['context']
        for qa in passage['qas']:
            question = qa['question']
            for answer in qa['answers']:
                texts.append(context)
                queries.append(question)
                answers.append(answer)

val_texts, val_queries, val_answers = texts, queries, answers
```

Here, We did the same thing for the validation data and continued the loop for this too. This line creates a Path object named path, representing the file path

'/content/valid_bangla_samples_fixed_preprocessed.json?download=1'. Then 'with' statement to open the JSON file located at the specified path in binary read mode ('rb'). It then loads the contents of the JSON file into a Python dictionary named squad_dict using json.load(). After that, these lines initialize empty lists (texts, queries, and answers) to store information extracted from the JSON file.

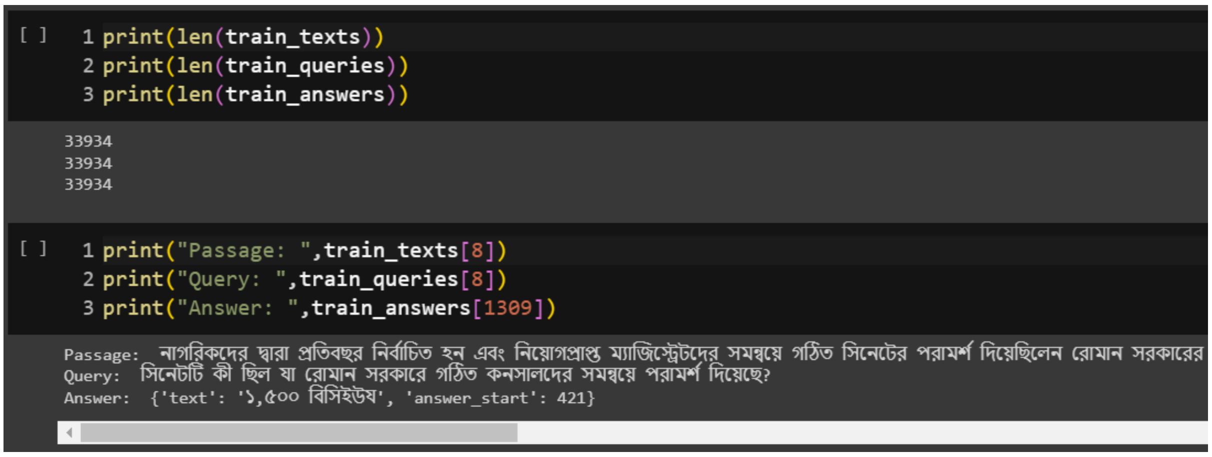
Assigns the contents of the three lists to new variables:

train_texts: stores the list of context texts.

train_queries: stores the list of question texts.

train_answers: stores the list of answer texts.

Overall this efficiently processes the training data by extracting the relevant information (context, question, and answer) and storing it in structured lists. These organized data structures will be essential for further tasks in training and developing the question-answering system. Now, this whole process is done for the validation data which shows how the model performs when given the unseen data.



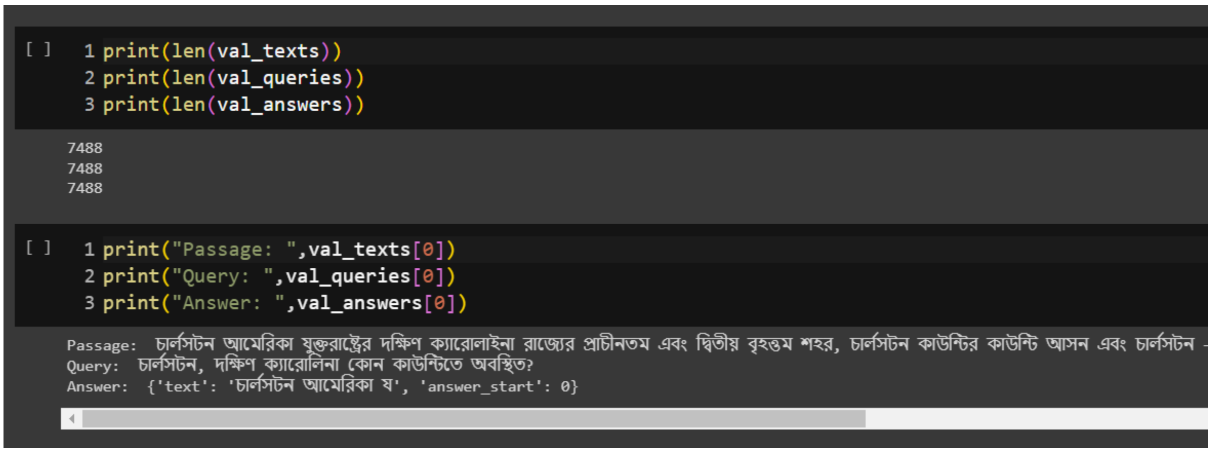
```
print(len(train_texts))
print(len(train_queries))
print(len(train_answers))
```

```
33934
33934
33934
```

```
print("Passage: ",train_texts[8])
print("Query: ",train_queries[8])
print("Answer: ",train_answers[1309])
```

```
Passage:  নাগরিকদের দ্বারা প্রতিবছর নির্বাচিত হন এবং নিয়োগপ্রাপ্ত ম্যাজিস্ট্রেটদের সমন্বয়ে গঠিত সিনেটের পরামর্শ দিয়েছিলেন রোমান সরকারের
Query:  সিনেটটি কী ছিল যা রোমান সরকারে গঠিত কনসালদের সমন্বয়ে পরামর্শ দিয়েছে?
Answer:  {'text': '১,৫০০ বিসিইউয', 'answer_start': 421}
```

In this part of the code, I incorporated a snippet that not only prints the lengths of three crucial lists—train_texts, train_queries, and train_answers—but also extracts and displays specific examples from these datasets. In this context, the code specifically focuses on the 9th element in the lists (index 8 due to zero-based indexing). The "Passage" line reveals the content of the 9th passage from train_texts, shedding light on the actual text data. Similarly, the "Query" line exposes the 9th query from train_queries, providing insights into the nature of the questions posed. Finally, the "Answer" line highlights the 1310th answer from train_answers, showcasing the information associated with that particular answer. To delve deeper into the specifics, one can inspect the content of train_texts[8], train_queries[8], and train_answers[1309], enabling a closer examination of the passage text, query, and answer details at these specific positions in the lists. This exploration serves as a valuable step in understanding the composition and structure of the training data, especially within the context of a question-answering task.



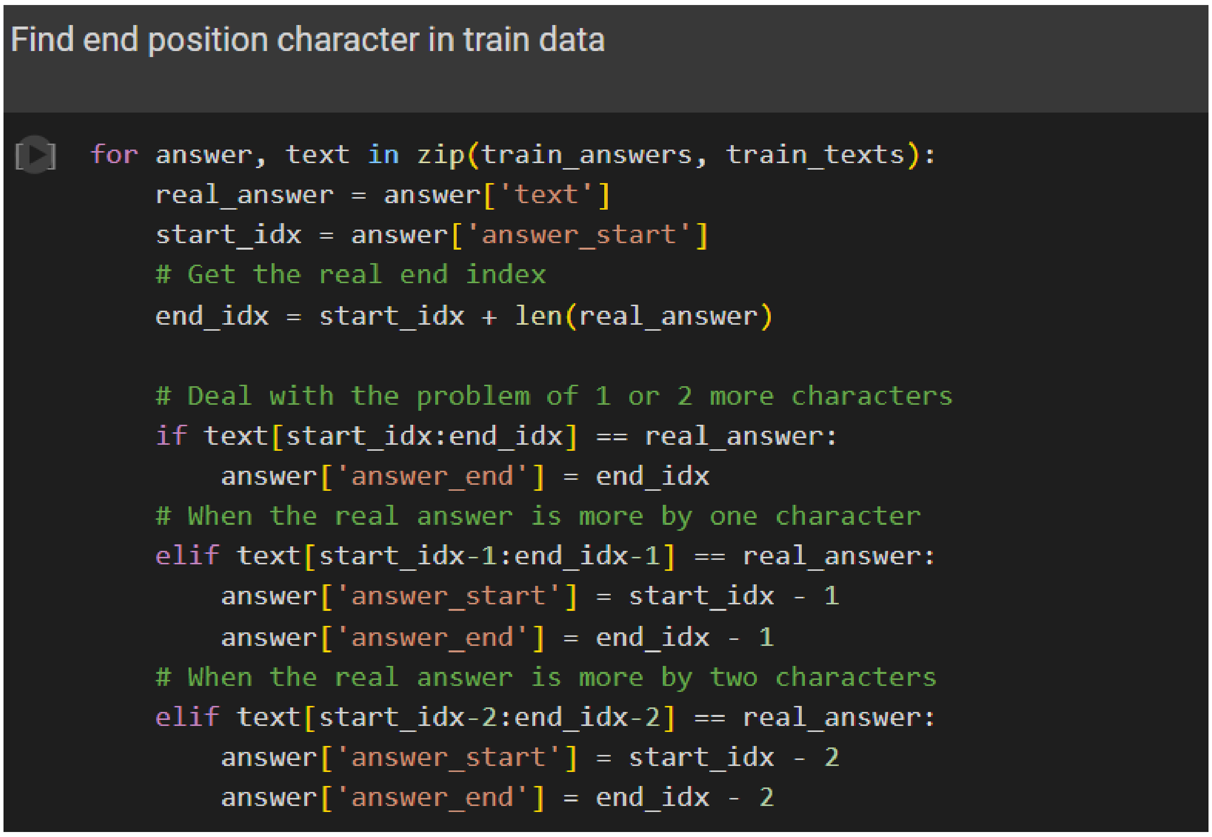
```
[ ]  print(len(val_texts))
     print(len(val_queries))
     print(len(val_answers))
```

```
7488
7488
7488
```

```
[ ]  print("Passage: ",val_texts[0])
     print("Query: ",val_queries[0])
     print("Answer: ",val_answers[0])
```

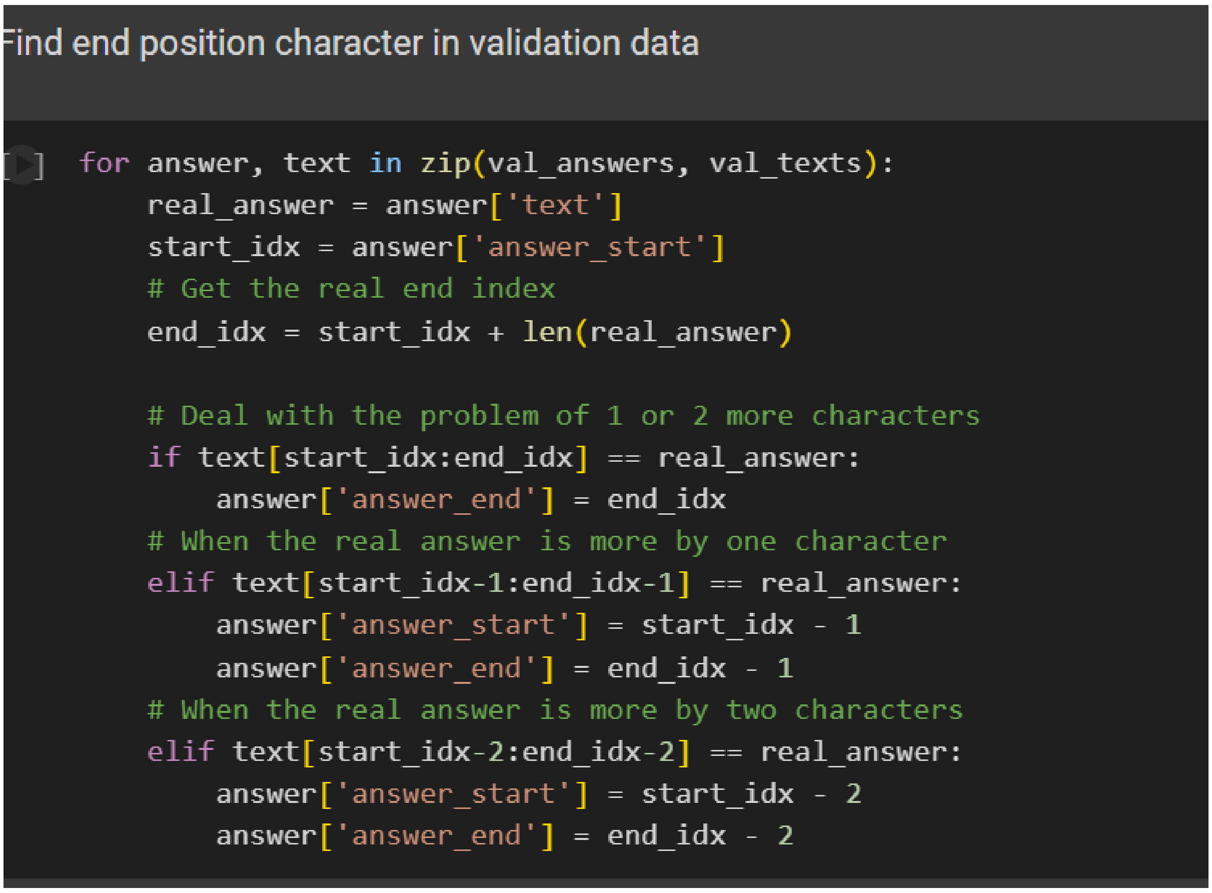
```
Passage:  চার্লসটন আমেরিকা যুক্তরাষ্ট্রের দক্ষিণ ক্যারোলাইনা রাজ্যের প্রাচীনতম এবং দ্বিতীয় বৃহত্তম শহর, চার্লসটন কাউন্টির কাউন্টি আসন এবং চার্লসটন -
Query:  চার্লসটন, দক্ষিণ ক্যারোলিনা কোন কাউন্টিতে অবস্থিত?
Answer:  {'text': 'চার্লসটন আমেরিকা য', 'answer_start': 0}
```

In this part of the code, I extended the exploration to the validation dataset by printing the lengths of three key lists: val_texts, val_queries, and val_answers. These lengths offer insights into the size of the validation data. Subsequently, the code highlights specific content from the validation dataset by printing the 1st passage, query, and answer from val_texts, val_queries, and val_answers, respectively. The "Passage" line reveals the content of the first passage from val_texts, while the "Query" line showcases the first query from val_queries, providing a glimpse into the nature of the validation questions. Finally, the "Answer" line displays the first answer from val_answers, presenting information associated with the initial answer in the validation dataset. To inspect the detailed content at these specific indices, one can examine val_texts[0], val_queries[0], and val_answers[0]. This exploration aids in understanding the composition and structure of the validation data, crucial for assessing the performance of a question-answering model on new, unseen examples.


Find end position character in train data

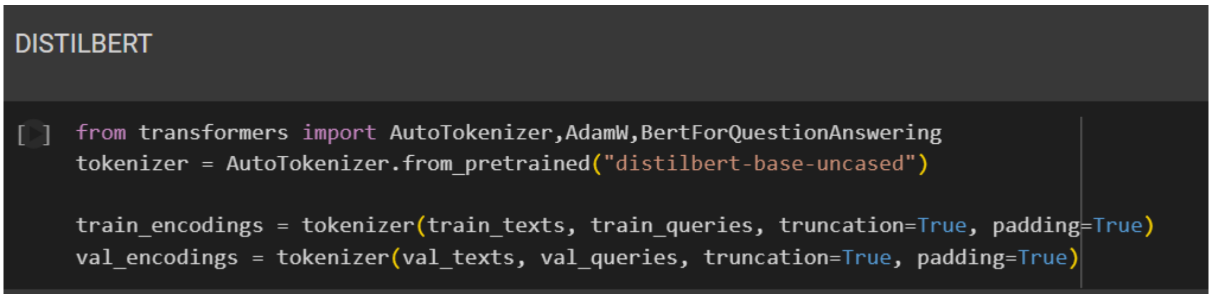
```
for answer, text in zip(train_answers, train_texts):
    real_answer = answer['text']
    start_idx = answer['answer_start']
    # Get the real end index
    end_idx = start_idx + len(real_answer)

    # Deal with the problem of 1 or 2 more characters
    if text[start_idx:end_idx] == real_answer:
        answer['answer_end'] = end_idx
    # When the real answer is more by one character
    elif text[start_idx-1:end_idx-1] == real_answer:
        answer['answer_start'] = start_idx - 1
        answer['answer_end'] = end_idx - 1
    # When the real answer is more by two characters
    elif text[start_idx-2:end_idx-2] == real_answer:
        answer['answer_start'] = start_idx - 2
        answer['answer_end'] = end_idx - 2
```

In this part of the code, I implemented a preprocessing step for a question-answering task, specifically addressing discrepancies between provided answers and the actual text in the training dataset. Through the use of a for loop that iterates over pairs of answer and text obtained by zipping train_answers and train_texts, the code dynamically adjusts the start and end indices of the answers to account for potential variations in the text. The code first extracts the actual answer text, start index, and computes the real end index. It then addresses the issue of one or two extra characters in the text by checking three conditions. If the substring of text from start_idx to end_idx matches the real answer, the end index is updated accordingly. In cases where the real answer has one or two more characters, the code adjusts both the start and end indices accordingly, providing a robust solution for aligning answers with their corresponding passages in the training data. This preprocessing step is essential for ensuring accurate training of a question-answering model on diverse and potentially noisy data.


ind end position character in validation data

```
for answer, text in zip(val_answers, val_texts):
    real_answer = answer['text']
    start_idx = answer['answer_start']
    # Get the real end index
    end_idx = start_idx + len(real_answer)

    # Deal with the problem of 1 or 2 more characters
    if text[start_idx:end_idx] == real_answer:
        answer['answer_end'] = end_idx
    # When the real answer is more by one character
    elif text[start_idx-1:end_idx-1] == real_answer:
        answer['answer_start'] = start_idx - 1
        answer['answer_end'] = end_idx - 1
    # When the real answer is more by two characters
    elif text[start_idx-2:end_idx-2] == real_answer:
        answer['answer_start'] = start_idx - 2
        answer['answer_end'] = end_idx - 2
```

In this part of the code, I extended the preprocessing methodology to the validation dataset for a question-answering task. Employing a for loop that iterates over pairs of answer and text obtained by zipping val_answers and val_texts, the code dynamically adjusts the start and end indices of the answers to reconcile any differences between the provided answers and the actual text in the validation dataset. The code extracts the real answer text, start index, and computes the real end index. It then addresses potential discrepancies, handling cases where there might be one or two extra characters in the text. If the substring of text from start_idx to end_idx matches the real answer, the end index is updated accordingly. For scenarios where the real answer has one or two more characters, the code adjusts both the start and end indices appropriately. This preprocessing step ensures the alignment of answers with their corresponding passages in the validation data, contributing to the accurate evaluation of a question-answering model on diverse and potentially nuanced validation examples.


DISTILBERT

```
from transformers import AutoTokenizer,AdamW,BertForQuestionAnswering
tokenizer = AutoTokenizer.from_pretrained("distilbert-base-uncased")

train_encodings = tokenizer(train_texts, train_queries, truncation=True, padding=True)
val_encodings = tokenizer(val_texts, val_queries, truncation=True, padding=True)
```

AutoTokenizer: Loads a pre-trained tokenizer that will preprocess text data for input into a transformer-based model.

AdamW: Optimizer for updating model weights during training.

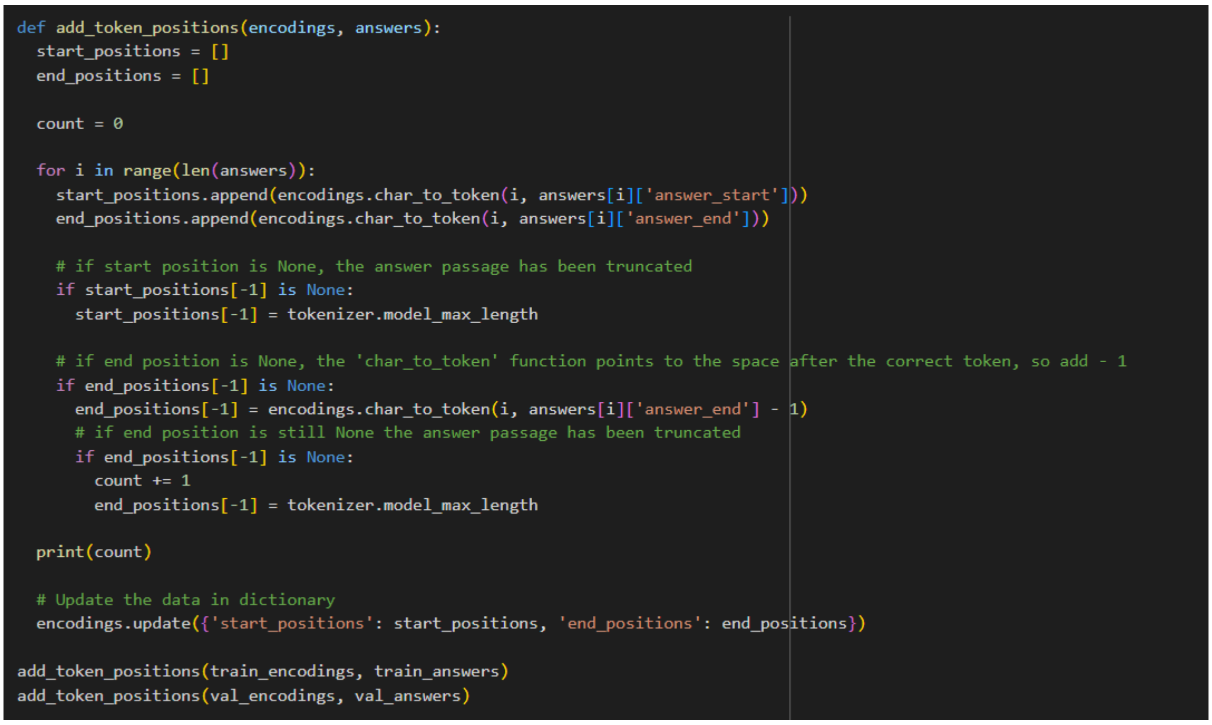
BertForQuestionAnswering: Pre-trained BERT-based model fine-tuned specifically for question answering tasks.

AutoTokenizer.from_pretrained(): Loads a tokenizer that corresponds to the specified pre-trained model. In this case, it's using the DistilBERT model (distilbert-base-uncased variant) and creating a tokenizer object.

tokenizer(): Tokenizes the text sequences and queries. The truncation=True parameter truncated sequences that exceed the maximum length, and padding=True pads sequences to ensure uniform length within a batch.

train_texts, train_queries: Lists/arrays containing the text data and corresponding queries for training.

val_texts, val_queries: Lists/arrays containing the text data and corresponding queries for validation.


```python
def add_token_positions(encodings, answers):
  start_positions = []
  end_positions = []

  count = 0

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

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

    # if end position is None, the 'char_to_token' function points to the space after the correct token, so add - 1
    if end_positions[-1] is None:
      end_positions[-1] = encodings.char_to_token(i, answers[i]['answer_end'] - 1)
      # if end position is still None the answer passage has been truncated
      if end_positions[-1] is None:
        count += 1
        end_positions[-1] = tokenizer.model_max_length

  print(count)

  # Update the data in dictionary
  encodings.update({'start_positions': start_positions, 'end_positions': end_positions})

add_token_positions(train_encodings, train_answers)
add_token_positions(val_encodings, val_answers)
```

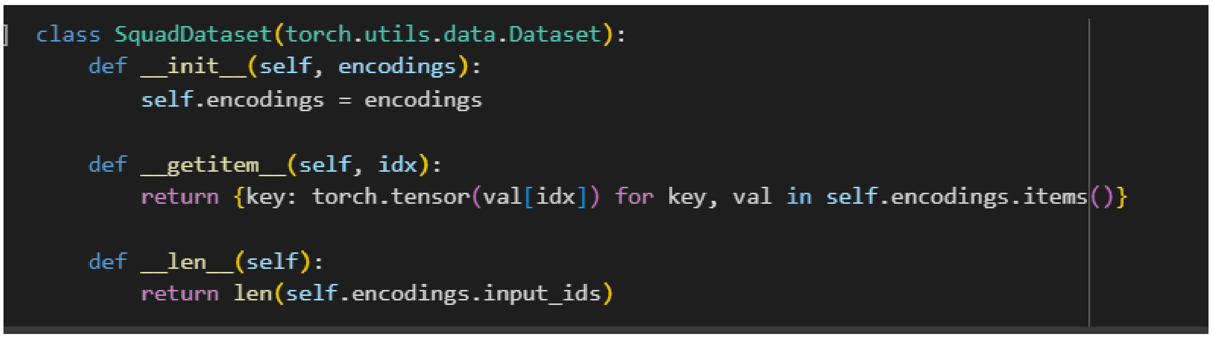
Compute token positions for answers within tokenized text or context.

Args:

- encodings (object): Tokenized text or context generated by a tokenizer.
- answers (list): Contains information about the start and end positions of answers within the context.

Returns:

- None: Updates the 'start_positions' and 'end_positions' within the 'encodings' object.



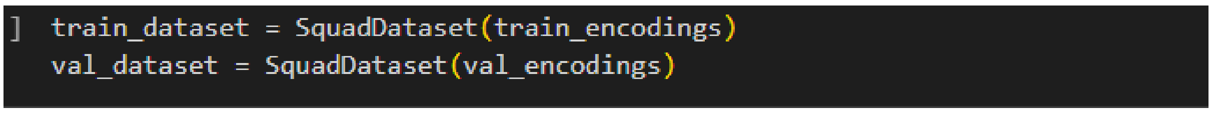
```python
class SquadDataset(torch.utils.data.Dataset):
    def __init__(self, encodings):
        self.encodings = encodings

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

    def __len__(self):
        return len(self.encodings.input_ids)
```

In this section, we designed a custom SquadDataset class for use with PyTorch. This class is a subclass of torch.utils.data.Dataset, which is a base class for representing a dataset in PyTorch. Here's a breakdown of its components:

__init__(self, encodings): This is the constructor of the class. It takes encodings as an argument, which should be a dictionary-like object containing tokenized input data. The encodings are stored in the instance variable self.encodings.

__getitem__(self, idx): This method is required by the PyTorch Dataset class. It defines how to access an individual item from the dataset. When the dataset is indexed, like dataset[idx], this method is called. It returns a dictionary where each key-value pair corresponds to a tensor of a specific feature (like input IDs, attention masks, etc.) for the indexed data point.

__len__(self): Another required method for PyTorch datasets. It simply returns the length of the dataset, which in this case is determined by the length of input_ids in encodings.



```python
train_dataset = SquadDataset(train_encodings)
val_dataset = SquadDataset(val_encodings)
```

In this section, we create an instance of the SquadDataset class for the training dataset. The train_encodings variable should contain the tokenized and encoded data for the training set. This data typically includes tokenized questions and contexts from the SQuAD dataset. Each entry in train_encodings is expected to be a dictionary with keys like input_ids, attention_mask, etc., which are common outputs from a tokenizer.

Then we create another instance of the SquadDataset for the validation dataset using val_encodings. The val_encodings should be structured like train_encodings, containing the tokenized and encoded validation data.

```
train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=16, shuffle=True)
```

In this section, we created a DataLoader for the training dataset (train_dataset). The batch_size=16 argument specifies that data should be loaded in batches of 16 samples. The shuffle=True argument indicates that the dataset should be shuffled at every epoch, which is a common practice to reduce model overfitting by ensuring that the model does not learn the order of the training data.

Then, we create another DataLoader for the validation dataset (val_dataset). It also uses a batch size of 16. While shuffling is typically not necessary for validation or testing data (and often set to False), setting shuffle=True here is not incorrect and depends on your specific requirements or preferences.

```
device = torch.device('cuda:0' if torch.cuda.is_available()
                        else 'cpu')
```

In this section, we need to set up a device for training or running a PyTorch model, determining whether to use a GPU (if available) or fall back to the CPU. Here's a breakdown of what it does:

torch.device('cuda:0' if torch.cuda.is_available() else 'cpu'):

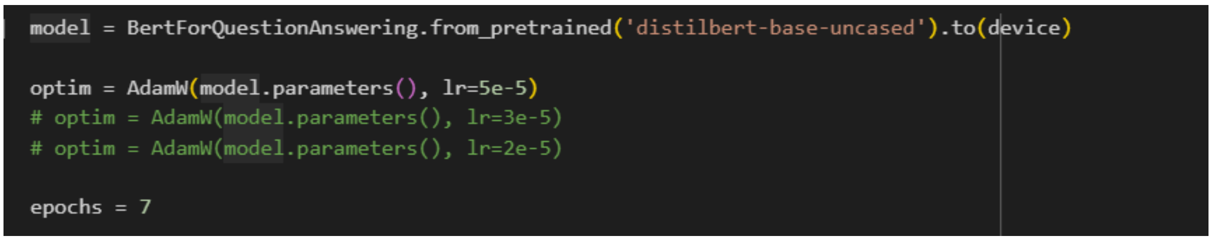
torch.device: This is a PyTorch function that creates a device object which can be used to move tensors to a GPU or CPU.

'cuda:0': This specifies the first GPU device. In systems with multiple GPUs, they are typically indexed as 'cuda:0', 'cuda:1', etc.

torch.cuda.is_available(): This function checks if a CUDA-enabled GPU is available on your system. If a CUDA GPU is available, it returns True.

'cpu': If no CUDA GPU is available, the device is set to use the CPU.

device = ...: This line assigns the created device object to the variable device.



```python
model = BertForQuestionAnswering.from_pretrained('distilbert-base-uncased').to(device)

optim = AdamW(model.parameters(), lr=5e-5)
# optim = AdamW(model.parameters(), lr=3e-5)
# optim = AdamW(model.parameters(), lr=2e-5)

epochs = 7
```

In this code snippet, we show the setup for training a BERT model for a question answering task using PyTorch and the Transformers library by Hugging Face. Let's break down the key parts:

model = BertForQuestionAnswering.from_pretrained('distilbert-base-uncased').to(device):

BertForQuestionAnswering.from_pretrained('distilbert-base-uncased'): This line initializes a BERT model pre-trained on the 'distilbert-base-uncased' architecture. The model is specifically designed for the question answering task. 'distilbert-base-uncased' refers to a distilled version of the BERT model which is smaller and faster.

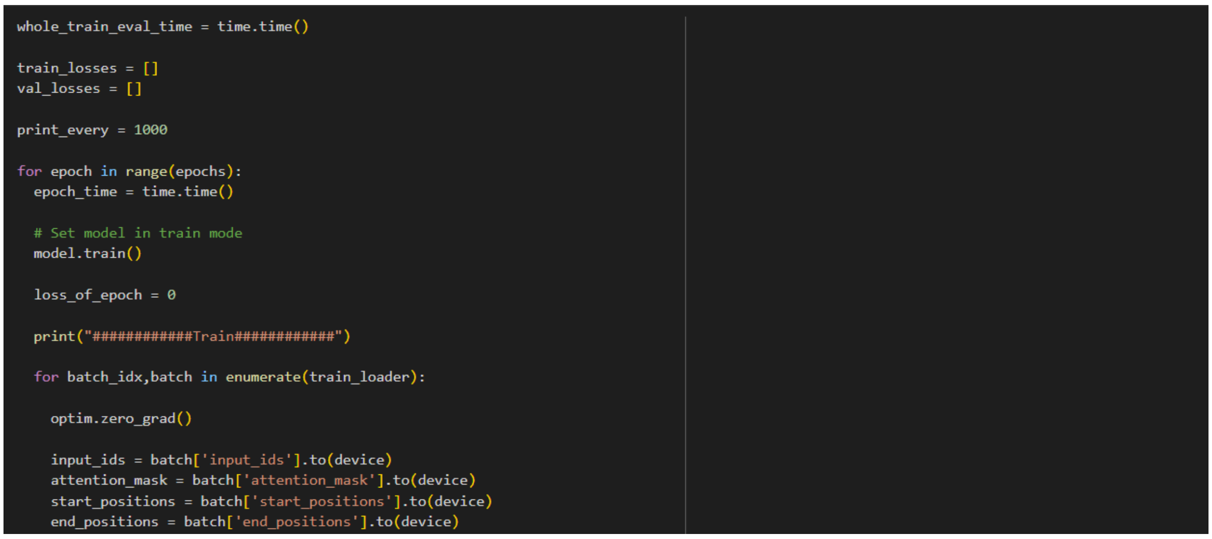
.to(device): This moves the model to the device (GPU or CPU) that you set up in the previous step. This is crucial for training the model with GPU acceleration if available.

optim = AdamW(model.parameters(), lr=5e-5):

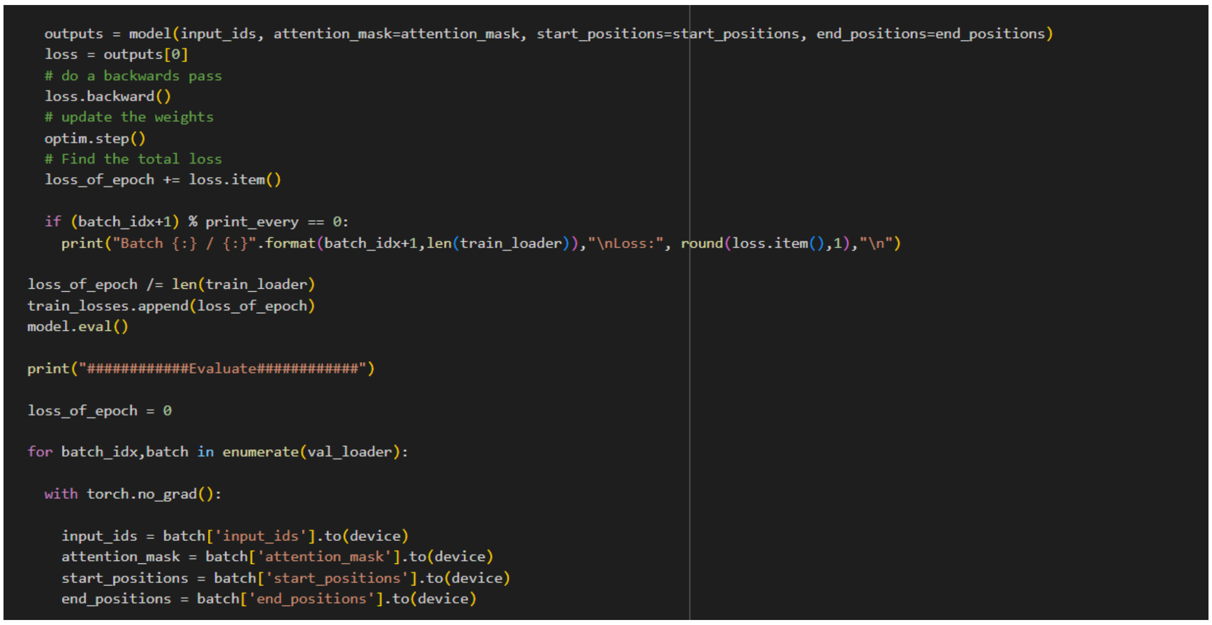
AdamW: This is an optimizer from the PyTorch library, often used with BERT models. It's a variant of the Adam optimizer with improved handling of weight decay.

model.parameters(): This tells the optimizer which parameters of the model it should optimize.

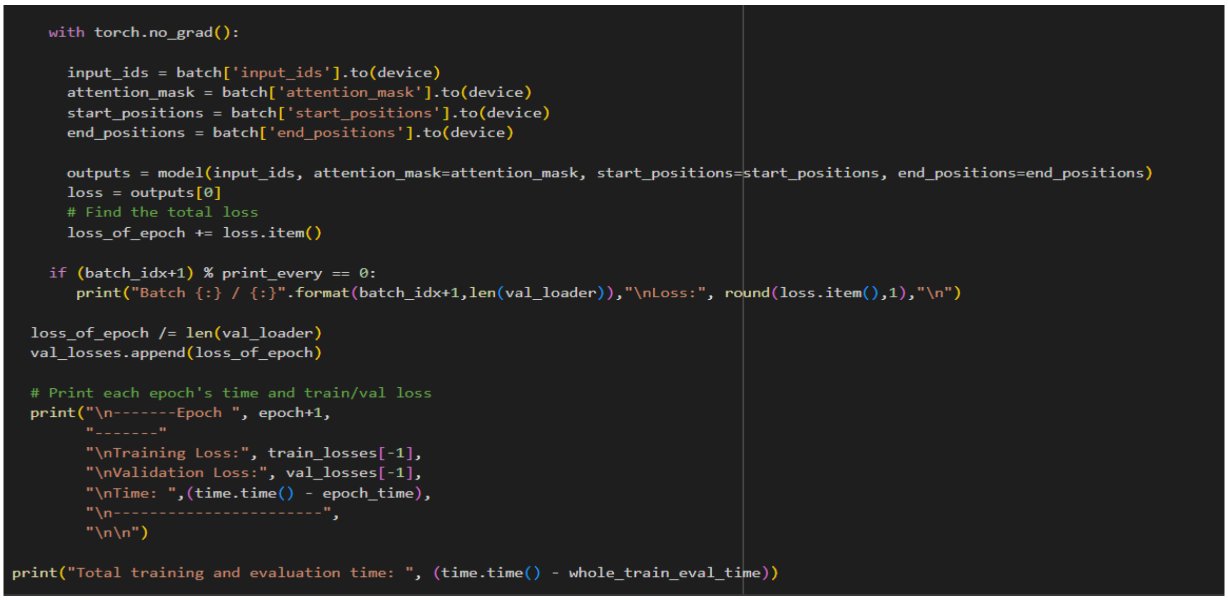
lr=5e-5: The learning rate is set to 5e-5 (0.00005). This is a hyperparameter that defines the step size during the gradient descent.



```python
whole_train_eval_time = time.time()

train_losses = []
val_losses = []

print_every = 1000

for epoch in range(epochs):
  epoch_time = time.time()

  # Set model in train mode
  model.train()

  loss_of_epoch = 0

  print("############Train############")

  for batch_idx,batch in enumerate(train_loader):

    optim.zero_grad()

    input_ids = batch['input_ids'].to(device)
    attention_mask = batch['attention_mask'].to(device)
    start_positions = batch['start_positions'].to(device)
    end_positions = batch['end_positions'].to(device)
```



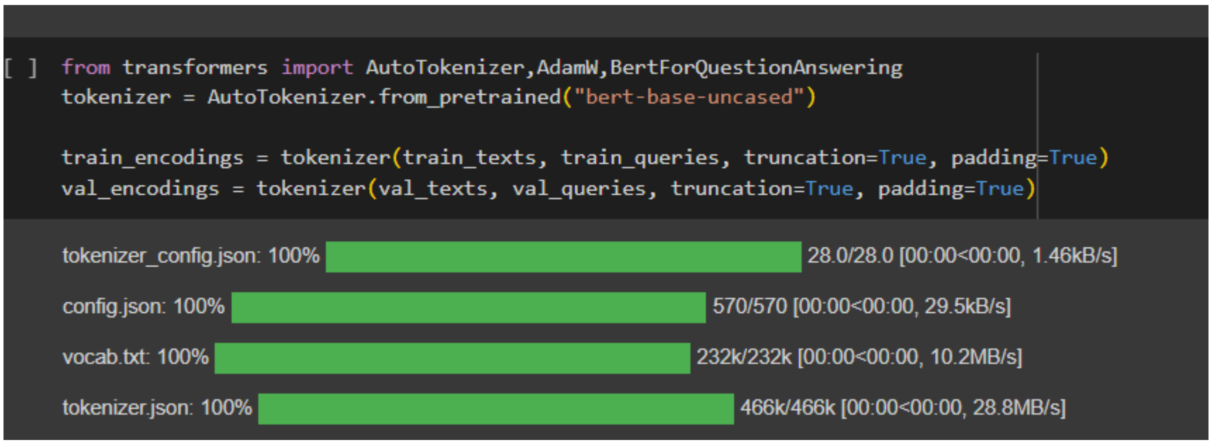
```python
    outputs = model(input_ids, attention_mask=attention_mask, start_positions=start_positions, end_positions=end_positions)
    loss = outputs[0]
    # do a backwards pass
    loss.backward()
    # update the weights
    optim.step()
    # Find the total loss
    loss_of_epoch += loss.item()

    if (batch_idx+1) % print_every == 0:
      print("Batch {:} / {:}".format(batch_idx+1,len(train_loader)),"\nLoss:", round(loss.item(),1),"\n")

  loss_of_epoch /= len(train_loader)
  train_losses.append(loss_of_epoch)
  model.eval()

  print("############Evaluate############")

  loss_of_epoch = 0

  for batch_idx,batch in enumerate(val_loader):

    with torch.no_grad():

      input_ids = batch['input_ids'].to(device)
      attention_mask = batch['attention_mask'].to(device)
      start_positions = batch['start_positions'].to(device)
      end_positions = batch['end_positions'].to(device)
```



```python
    with torch.no_grad():

      input_ids = batch['input_ids'].to(device)
      attention_mask = batch['attention_mask'].to(device)
      start_positions = batch['start_positions'].to(device)
      end_positions = batch['end_positions'].to(device)

      outputs = model(input_ids, attention_mask=attention_mask, start_positions=start_positions, end_positions=end_positions)
      loss = outputs[0]
      # Find the total loss
      loss_of_epoch += loss.item()

    if (batch_idx+1) % print_every == 0:
       print("Batch {:} / {:}".format(batch_idx+1,len(val_loader)),"\nLoss:", round(loss.item(),1),"\n")

  loss_of_epoch /= len(val_loader)
  val_losses.append(loss_of_epoch)

  # Print each epoch's time and train/val loss
  print("\n-------Epoch ", epoch+1,
        "-------"
        "\nTraining Loss:", train_losses[-1],
        "\nValidation Loss:", val_losses[-1],
        "\nTime: ",(time.time() - epoch_time),
        "\n-----------------------",
        "\n\n")

print("Total training and evaluation time: ", (time.time() - whole_train_eval_time))
```

First, we initialize our environment. We record the starting time for the entire training and evaluation process using whole_train_eval_time = time.time(). Lists like train_losses and val_losses are prepared to store the losses for training and validation phases, respectively, across each epoch. A key variable, print_every, is set, dictating how often we should print the loss information during training and validation loops. The heart of the process is the training loop, where for epoch in range(epochs), we iterate over a predefined number of epochs. Each epoch starts by recording its beginning time. The model is then set to training mode using model.train(). This is crucial as it enables certain features like dropout, which are relevant only during training. Inside the training loop, for each batch of data retrieved from train_loader, the gradients are first cleared using optim.zero_grad(). Then, input data and labels are loaded onto the device like a CPU or GPU. The model then performs a forward pass (outputs = model(...)), calculates the loss (loss = outputs[0]), and then back propagates the errors to update the model's weights (loss.backward() and optim.step()). Once training for an epoch is complete, the model switches to evaluation mode with model.eval(). In the validation phase, similar steps are followed but without the need for gradient computation, as indicated by the with torch.no_grad(): statement. This phase is crucial for assessing the model's performance on unseen data. After each epoch, a summary is printed, showing the average training and validation losses and the time taken for that epoch. This feedback loop is vital for understanding and improving the model's performance. Finally, once all epochs are completed, the code prints out the total time taken for training and evaluating the model, providing a sense of the computational effort involved. This structured process exemplifies the methodical approach required in the field of machine learning to train models effectively.

Bert



```
[ ]  from transformers import AutoTokenizer,AdamW,BertForQuestionAnswering
     tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")

     train_encodings = tokenizer(train_texts, train_queries, truncation=True, padding=True)
     val_encodings = tokenizer(val_texts, val_queries, truncation=True, padding=True)
```

```
tokenizer_config.json: 100% 28.0/28.0 [00:00<00:00, 1.46kB/s]
config.json: 100% 570/570 [00:00<00:00, 29.5kB/s]
vocab.txt: 100% 232k/232k [00:00<00:00, 10.2MB/s]
tokenizer.json: 100% 466k/466k [00:00<00:00, 28.8MB/s]
```

AutoTokenizer: Loads a pre-trained tokenizer that will preprocess text data for input into a transformer-based model.

AdamW: Optimizer for updating model weights during training.

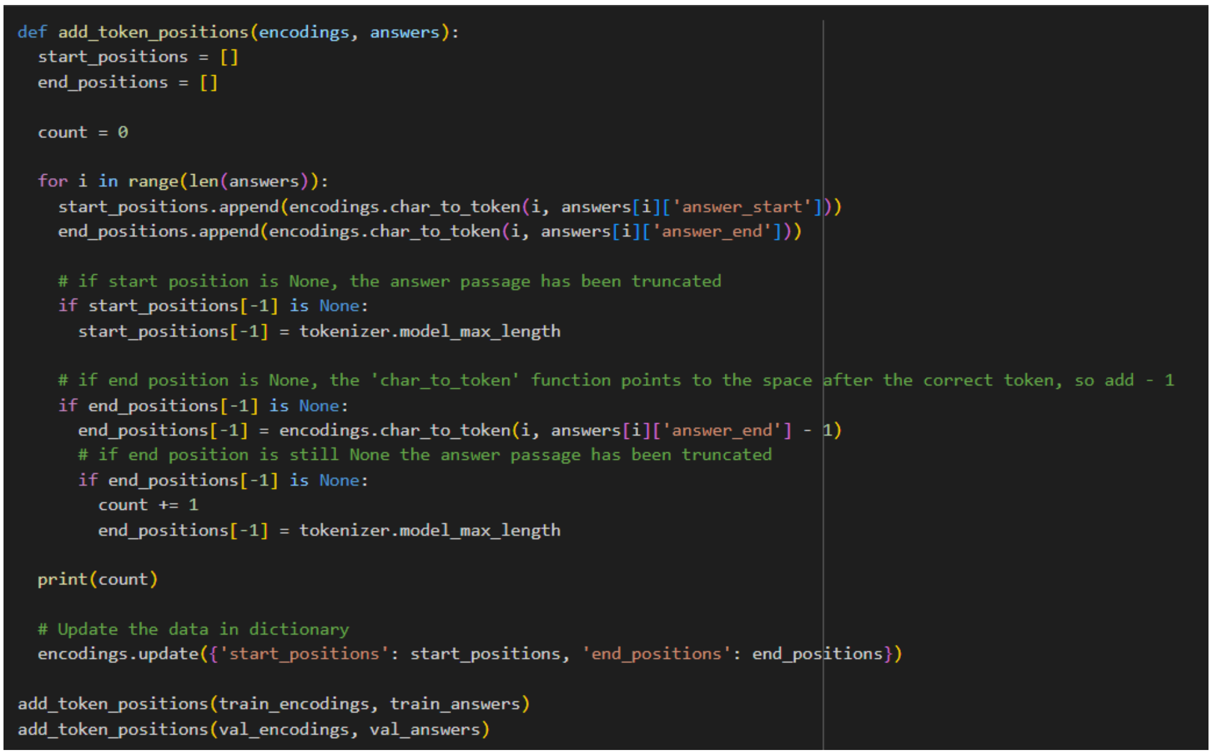
BertForQuestionAnswering: Pre-trained BERT-based model fine-tuned specifically for question answering tasks.

AutoTokenizer.from_pretrained(): Loads a tokenizer that corresponds to the specified pre-trained model. In this case, it's using the DistilBERT model (distilbert-base-uncased variant) and creating a tokenizer object.

tokenizer(): Tokenizes the text sequences and queries. The truncation=True parameter truncated sequences that exceed the maximum length, and padding=True pads sequences to ensure uniform length within a batch.

train_texts, train_queries: Lists/arrays containing the text data and corresponding queries for training.

val_texts, val_queries: Lists/arrays containing the text data and corresponding queries for validation.



```python
def add_token_positions(encodings, answers):
  start_positions = []
  end_positions = []

  count = 0

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

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

    # if end position is None, the 'char_to_token' function points to the space after the correct token, so add - 1
    if end_positions[-1] is None:
      end_positions[-1] = encodings.char_to_token(i, answers[i]['answer_end'] - 1)
      # if end position is still None the answer passage has been truncated
      if end_positions[-1] is None:
        count += 1
        end_positions[-1] = tokenizer.model_max_length

  print(count)

  # Update the data in dictionary
  encodings.update({'start_positions': start_positions, 'end_positions': end_positions})

add_token_positions(train_encodings, train_answers)
add_token_positions(val_encodings, val_answers)
```

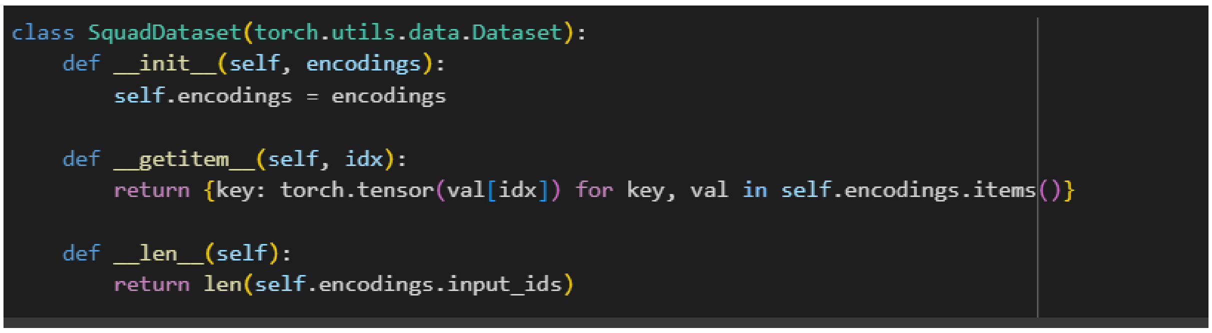
Compute token positions for answers within tokenized text or context.

Args:

- encodings (object): Tokenized text or context generated by a tokenizer.
- answers (list): Contains information about the start and end positions of answers within the context.

Returns:

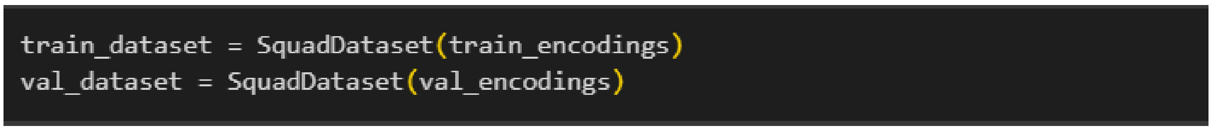
- None: Updates the 'start_positions' and 'end_positions' within the 'encodings' object.



```python
class SquadDataset(torch.utils.data.Dataset):
    def __init__(self, encodings):
        self.encodings = encodings

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

    def __len__(self):
        return len(self.encodings.input_ids)
```

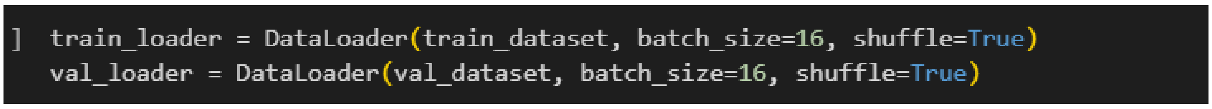
In this section, we designed a custom SquadDataset class for use with PyTorch. This class is a subclass of torch.utils.data.Dataset, which is a base class for representing a dataset in PyTorch. Here's a breakdown of its components:

__init__(self, encodings): This is the constructor of the class. It takes encodings as an argument, which should be a dictionary-like object containing tokenized input data. The encodings are stored in the instance variable self.encodings.
__getitem__(self, idx): This method is required by the PyTorch Dataset class. It defines how to access an individual item from the dataset. When the dataset is indexed, like dataset[idx], this method is called. It returns a dictionary where each key-value pair corresponds to a tensor of a specific feature (like input IDs, attention masks, etc.) for the indexed data point.
__len__(self): Another required method for PyTorch datasets. It simply returns the length of the dataset, which in this case is determined by the length of input_ids in encodings.



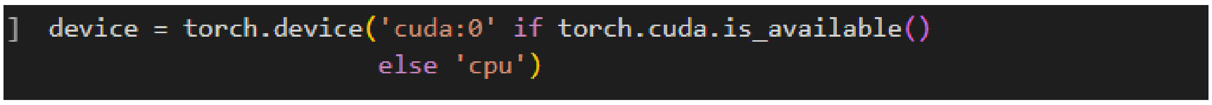
```
train_dataset = SquadDataset(train_encodings)
val_dataset = SquadDataset(val_encodings)
```

In this section, we create an instance of the SquadDataset class for the training dataset. The train_encodings variable should contain the tokenized and encoded data for the training set. This data typically includes tokenized questions and contexts from the SQuAD dataset. Each entry in train_encodings is expected to be a dictionary with keys like input_ids, attention_mask, etc., which are common outputs from a tokenizer.
Then we create another instance of the SquadDataset for the validation dataset using val_encodings. The val_encodings should be structured like train_encodings, containing the tokenized and encoded validation data.



```
] train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)
  val_loader = DataLoader(val_dataset, batch_size=16, shuffle=True)
```

In this section, we created a DataLoader for the training dataset (train_dataset). The batch_size=16 argument specifies that data should be loaded in batches of 16 samples. The shuffle=True argument indicates that the dataset should be shuffled at every epoch, which is a common practice to reduce model overfitting by ensuring that the model does not learn the order of the training data.
Then, we create another DataLoader for the validation dataset (val_dataset). It also uses a batch size of 16. While shuffling is typically not necessary for validation or testing data (and often set to False), setting shuffle=True here is not incorrect and depends on your specific requirements or preferences.



```
]  device = torch.device('cuda:0' if torch.cuda.is_available()
                          else 'cpu')
```

In this section, we need to set up a device for training or running a PyTorch model, determining whether to use a GPU (if available) or fall back to the CPU. Here's a breakdown of what it does:

torch.device('cuda:0' if torch.cuda.is_available() else 'cpu'):

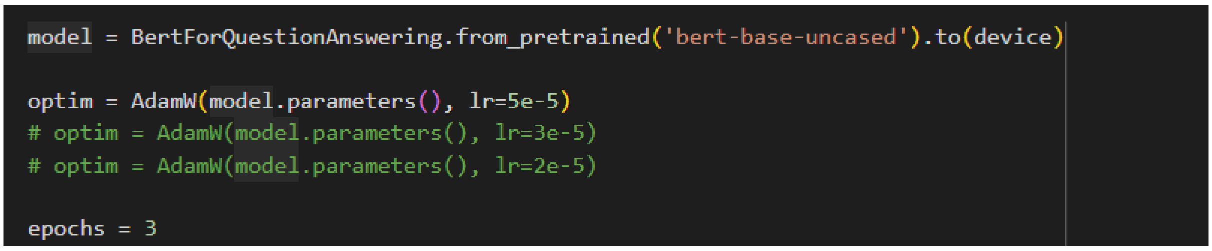
torch.device: This is a PyTorch function that creates a device object which can be used to move tensors to a GPU or CPU.

'cuda:0': This specifies the first GPU device. In systems with multiple GPUs, they are typically indexed as 'cuda:0', 'cuda:1', etc.

torch.cuda.is_available(): This function checks if a CUDA-enabled GPU is available on your system. If a CUDA GPU is available, it returns True.

'cpu': If no CUDA GPU is available, the device is set to use the CPU.

device = ...: This line assigns the created device object to the variable device.



```
model = BertForQuestionAnswering.from_pretrained('bert-base-uncased').to(device)

optim = AdamW(model.parameters(), lr=5e-5)
# optim = AdamW(model.parameters(), lr=3e-5)
# optim = AdamW(model.parameters(), lr=2e-5)

epochs = 3
```

In this code snippet, we show the setup for training a BERT model for a question answering task using PyTorch and the Transformers library by Hugging Face. Let's break down the key parts:

model = BertForQuestionAnswering.from_pretrained('distilbert-base-uncased').to(device):

BertForQuestionAnswering.from_pretrained('distilbert-base-uncased'): This line initializes a BERT model pre-trained on the 'distilbert-base-uncased' architecture. The model is specifically designed for the question answering task. 'distilbert-base-uncased' refers to a distilled version of the BERT model which is smaller and faster.

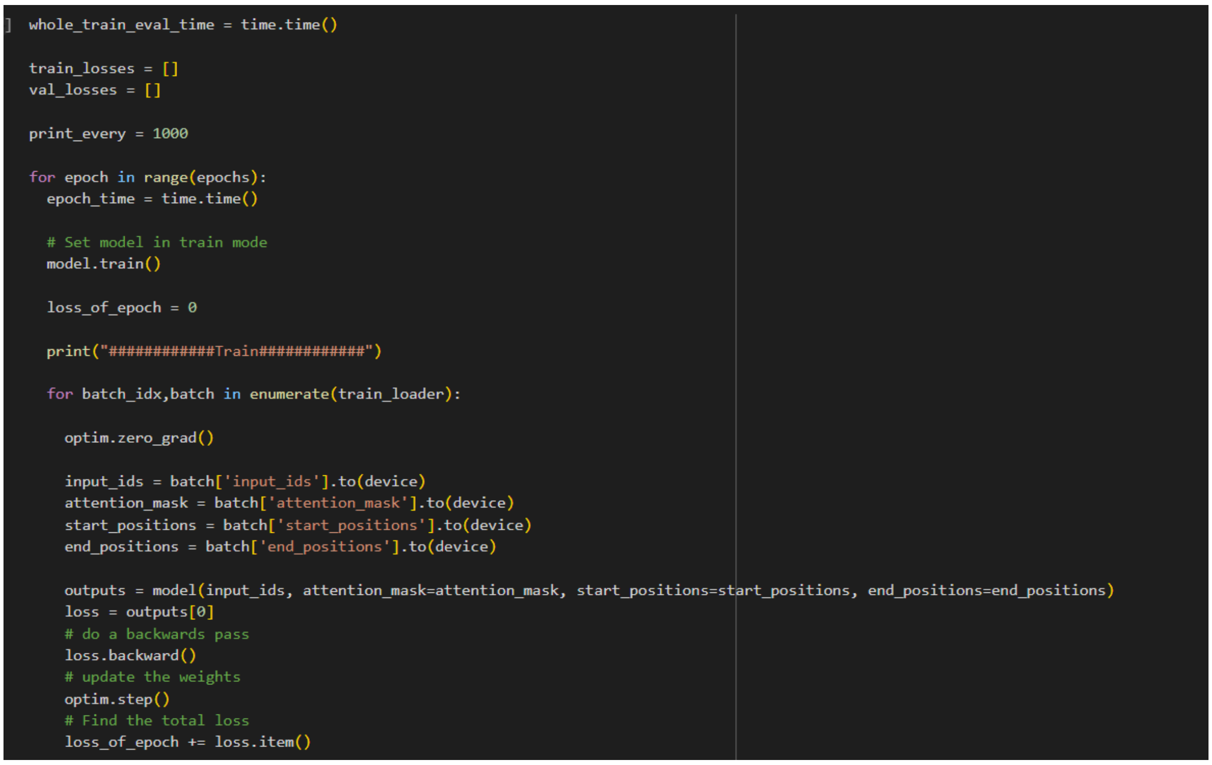
.to(device): This moves the model to the device (GPU or CPU) that you set up in the previous step. This is crucial for training the model with GPU acceleration if available.

optim = AdamW(model.parameters(), lr=5e-5):

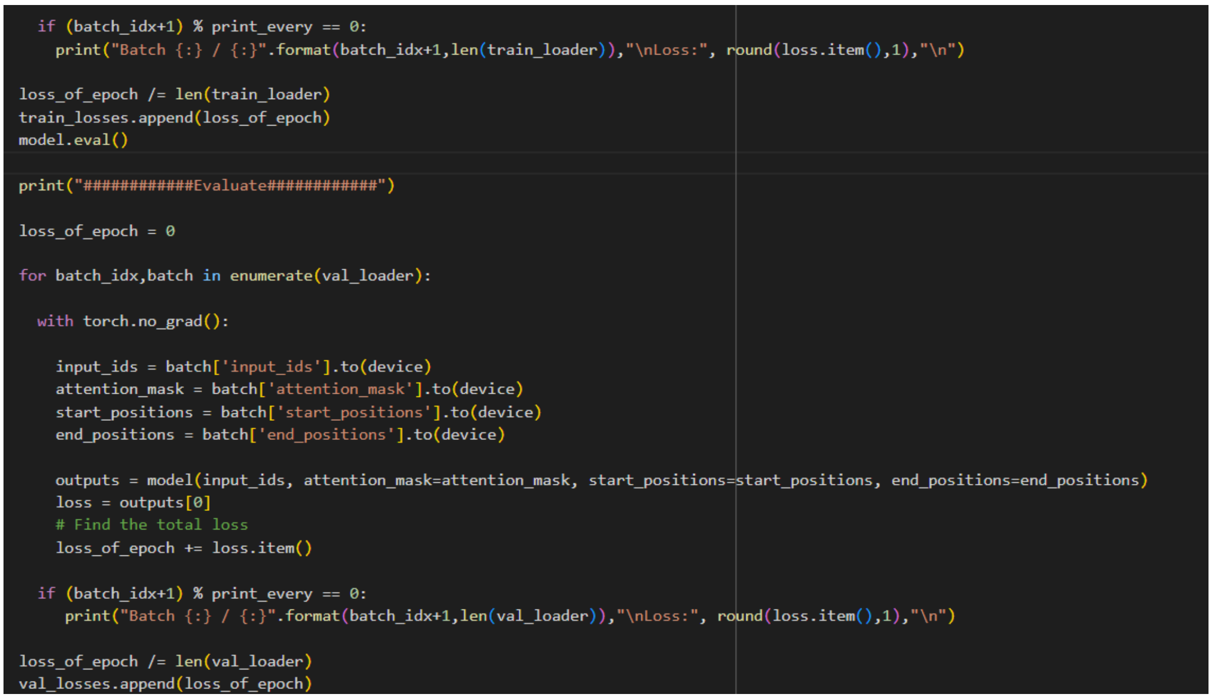
AdamW: This is an optimizer from the PyTorch library, often used with BERT models. It's a variant of the Adam optimizer with improved handling of weight decay.

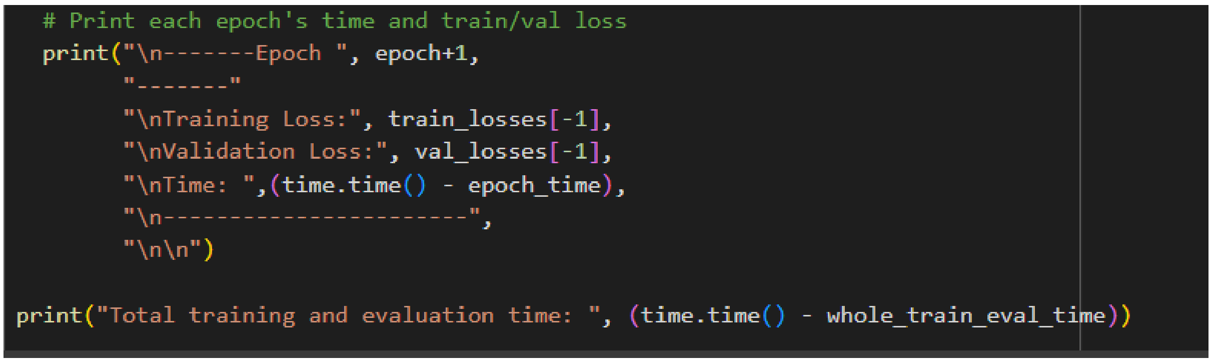
model.parameters(): This tells the optimizer which parameters of the model it should optimize.

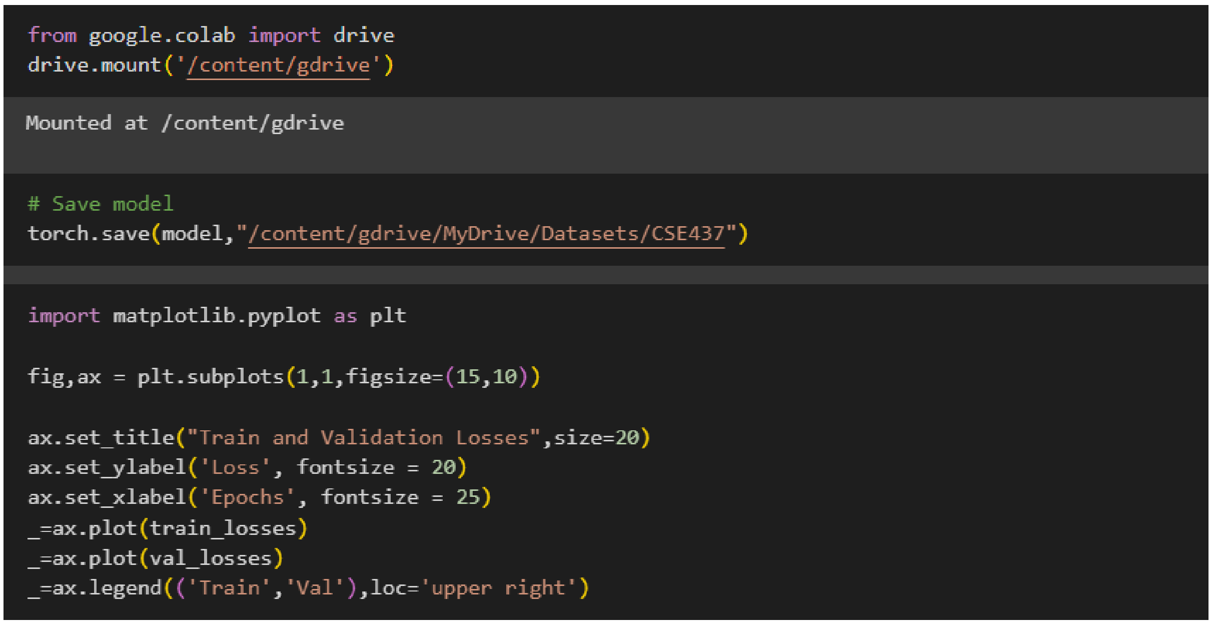
lr=5e-5: The learning rate is set to 5e-5 (0.00005). This is a hyperparameter that defines the step size during the gradient descent.



```
] whole_train_eval_time = time.time()

  train_losses = []
  val_losses = []

  print_every = 1000

  for epoch in range(epochs):
    epoch_time = time.time()

    # Set model in train mode
    model.train()

    loss_of_epoch = 0

    print("############Train############")

    for batch_idx,batch in enumerate(train_loader):

      optim.zero_grad()

      input_ids = batch['input_ids'].to(device)
      attention_mask = batch['attention_mask'].to(device)
      start_positions = batch['start_positions'].to(device)
      end_positions = batch['end_positions'].to(device)

      outputs = model(input_ids, attention_mask=attention_mask, start_positions=start_positions, end_positions=end_positions)
      loss = outputs[0]
      # do a backwards pass
      loss.backward()
      # update the weights
      optim.step()
      # Find the total loss
      loss_of_epoch += loss.item()
```


```
  if (batch_idx+1) % print_every == 0:
    print("Batch {:} / {:}".format(batch_idx+1,len(train_loader)),"\nLoss:", round(loss.item(),1),"\n")

loss_of_epoch /= len(train_loader)
train_losses.append(loss_of_epoch)
model.eval()

print("############Evaluate############")

loss_of_epoch = 0

for batch_idx,batch in enumerate(val_loader):

  with torch.no_grad():

    input_ids = batch['input_ids'].to(device)
    attention_mask = batch['attention_mask'].to(device)
    start_positions = batch['start_positions'].to(device)
    end_positions = batch['end_positions'].to(device)

    outputs = model(input_ids, attention_mask=attention_mask, start_positions=start_positions, end_positions=end_positions)
    loss = outputs[0]
    # Find the total loss
    loss_of_epoch += loss.item()

  if (batch_idx+1) % print_every == 0:
     print("Batch {:} / {:}".format(batch_idx+1,len(val_loader)),"\nLoss:", round(loss.item(),1),"\n")

loss_of_epoch /= len(val_loader)
val_losses.append(loss_of_epoch)
```


```
  # Print each epoch's time and train/val loss
  print("\n-------Epoch ", epoch+1,
        "-------"
        "\nTraining Loss:", train_losses[-1],
        "\nValidation Loss:", val_losses[-1],
        "\nTime: ",(time.time() - epoch_time),
        "\n-----------------------",
        "\n\n")

print("Total training and evaluation time: ", (time.time() - whole_train_eval_time))
```


```
from google.colab import drive
drive.mount('/content/gdrive')
```

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

```
# Save model
torch.save(model,"/content/gdrive/MyDrive/Datasets/CSE437")
```

```
import matplotlib.pyplot as plt

fig,ax = plt.subplots(1,1,figsize=(15,10))

ax.set_title("Train and Validation Losses",size=20)
ax.set_ylabel('Loss', fontsize = 20)
ax.set_xlabel('Epochs', fontsize = 25)
_=ax.plot(train_losses)
_=ax.plot(val_losses)
_=ax.legend(('Train','Val'),loc='upper right')
```

Here in print_every variable, determining the frequency of progress updates during the process. In the training loop, which iterates over epochs. In each epoch, the model is set to training mode, activating certain features like dropout, and it does following steps: loading data, forward pass, loss computation, backpropagation, and parameter updates. Following the training phase in each epoch is the evaluation phase, where the model, now in evaluation mode, processes the validation data without updating its parameters. This is crucial for observing the model's performance on unseen data. Finally, after completing all epochs, the code prints the total time taken for this training and evaluation cycle, which provides insights into the computational efficiency of the model training process.