6.864 Homework 4 - Question Answering

1 Introduction

In this homework, we'll put together all the tools we've learned in this class to build a model for a more complex task: answering questions about written passages.

For this assignment, we'll start with pre-trained sentence representations from a model called DistilBERT (a smaller version of the BERT model we discussed in lecture). See the paper (https://arxiv.org/pdf/1910.01108.pdf) for more info.

Note that the implementation of this homework for question answering is slightly different from the method introducted in the lecture - in order to let you know more possible solutions to the QA task.

1.1 Overview

To build a question answering model from DistilBERT, we need to do the following

1. Download a pretrained DistilBERT model.
2. Add a task-specific answer prediction layer on top of DistilBERT's representations.
3. Fine-tune both DistilBERT and the answer prediction layer on a Q&A task.
4. Evaluate the trained Q&A model.

This assignment will also introduce you to a set of libraries from an organization called HuggingFace (yes, really) that are commonly used to access NLP datasets and pre-trained transformer models.

1.2 Data

The dataset we will be using is the Stanford Question and Answer Dataset (SQuAD) v1.1. You can learn more about the dataset here (https://rajpurkar.github.io/SQuAD-explorer/explore/1.1/dev/). Just be careful to look at the v1.1 version, not v2. We'll download this model using the HuggingFace `datasets` package.

1.3 Pretrained model

You will install the HuggingFace `transformers` package. This package provides a wide variety of pretrained transformers. Check out out the documentation (https://huggingface.co/transformers/) for more information.

1.4 Hardware

Make sure you've enabled GPU as a hardware accelerator for this notebook.

1.5 Important: Using Google Drive

It is highly recommended that you mount your Google Drive to Colab. The code provided to you assumes that you've already done that. Create a folder named `6864_hw4` in your Google Drive root directory and use the code below to mount it. The code should save everything (dataset, feature-ized data, trained models etc.) in the `6864_hw4` folder in your drive.

In [ ]:

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

```
Drive already mounted at /content/gdrive; to attempt to forcibly remount,
call drive.mount("/content/gdrive", force_remount=True).
```

In [ ]:

```
%%bash
# Logistics #2: install the transformers package, create a folder, download the dataset
and a patch
pip -q install transformers
pip -q install datasets
pip -q install tqdm
pip -q install sentencepiece

# remove the directory if necessary
# rm -rf "/content/gdrive/MyDrive/6864_hw4/"

mkdir "/content/gdrive/MyDrive/6864_hw4/"
cd "/content/gdrive/MyDrive/6864_hw4/"
```

```
mkdir: cannot create directory ‘/content/gdrive/MyDrive/6864_hw4/’: File e
xists
```

Lets load the SQuAD dataset and observe its structure and data

In [ ]:

```
from datasets import load_dataset

squad = load_dataset('squad')
print(squad)
```

```
Reusing dataset squad (/root/.cache/huggingface/datasets/squad/plain_text/
1.0.0/4fffa6cf76083860f85fa83486ec3028e7e32c342c218ff2a620fc6b2868483a)

DatasetDict({
    train: Dataset({
        features: ['id', 'title', 'context', 'question', 'answers'],
        num_rows: 87599
    })
    validation: Dataset({
        features: ['id', 'title', 'context', 'question', 'answers'],
        num_rows: 10570
    })
})
```

In [ ]:

```
squad['train'][0].items()
```

Out[ ]:

```
dict_items([('answers', {'answer_start': [515], 'text': ['Saint Bernadette
Soubirous']}), ('context', 'Architecturally, the school has a Catholic cha
racter. Atop the Main Building\'s gold dome is a golden statue of the Virg
in Mary. Immediately in front of the Main Building and facing it, is a cop
per statue of Christ with arms upraised with the legend "Venite Ad Me Omne
s". Next to the Main Building is the Basilica of the Sacred Heart. Immedia
tely behind the basilica is the Grotto, a Marian place of prayer and refle
ction. It is a replica of the grotto at Lourdes, France where the Virgin M
ary reputedly appeared to Saint Bernadette Soubirous in 1858. At the end o
f the main drive (and in a direct line that connects through 3 statues and
the Gold Dome), is a simple, modern stone statue of Mary.'), ('id', '5733b
e284776f41900661182'), ('question', 'To whom did the Virgin Mary allegedly
appear in 1858 in Lourdes France?'), ('title', 'University_of_Notre_Dam
e')])
```

In [ ]:

```
for key, value in squad['train'][0].items():
    print(key)
    print(value)
    print('--------------------------')
```

```
answers
{'answer_start': [515], 'text': ['Saint Bernadette Soubirous']}
--------------------------
context
Architecturally, the school has a Catholic character. Atop the Main Buildi
ng's gold dome is a golden statue of the Virgin Mary. Immediately in front
of the Main Building and facing it, is a copper statue of Christ with arms
upraised with the legend "Venite Ad Me Omnes". Next to the Main Building i
s the Basilica of the Sacred Heart. Immediately behind the basilica is the
Grotto, a Marian place of prayer and reflection. It is a replica of the gr
otto at Lourdes, France where the Virgin Mary reputedly appeared to Saint
Bernadette Soubirous in 1858. At the end of the main drive (and in a direc
t line that connects through 3 statues and the Gold Dome), is a simple, mo
dern stone statue of Mary.
--------------------------
id
5733be284776f41900661182
--------------------------
question
To whom did the Virgin Mary allegedly appear in 1858 in Lourdes France?
--------------------------
title
University_of_Notre_Dame
--------------------------
```

Notice that the `answers` in this dataset always consist of substrings of the `contexts`. Thus, like we discussed in class, we'll build a question-answering model by predicting the *locations* of answers within contexts.

2 Introducing the Huggingface Toolkit

We'll be using two new packages:

- [transformers (https://github.com/huggingface/transformers)](https://github.com/huggingface/transformers). In this package,
  - **tokenizers** automatically convert strings into sequence of integer word piece IDs.
  - **models** provided architectures and weights for state-of-the-art pretrained transformer language models.
- [datasets (https://github.com/huggingface/datasets)](https://github.com/huggingface/datasets)
  - A toolkit that help you download and and evaluate your model on standard benchmarks easily
  - You can publish your own dataset on Huggingface [Datasets Hub (https://huggingface.co/datasets)](https://huggingface.co/datasets).

You can find more information and examples with the links above. The [demo notebook (https://drive.google.com/file/d/1_vx14SQkTyeWPW6lRR8_BHAO3urlfaQh/view?usp=sharing)](https://drive.google.com/file/d/1_vx14SQkTyeWPW6lRR8_BHAO3urlfaQh/view?usp=sharing) for recitation 6 is also very helpful.

2.1 Tokenizers

If we're going to pass text to pretrained models, we need to make sure that the input text is pre-processed into sequences of token IDs in the same way as the training data. `tokenizers` let us do this:

In [ ]:

```
import transformers

# Use a pretrained tokenizer with CLASS.from_pretrained() function
tokenizer = transformers.AutoTokenizer.from_pretrained('distilbert-base-cased')

context = 'You can protect yourself by wearing an N95 mask.'
answer = 'wearing an N95 mask'

context_ids = tokenizer.encode(context)
print(context_ids)
print(tokenizer.convert_ids_to_tokens(context_ids))
```

```
[101, 1192, 1169, 3244, 3739, 1118, 3351, 1126, 151, 1580, 1571, 7739, 11
9, 102]
['[CLS]', 'You', 'can', 'protect', 'yourself', 'by', 'wearing', 'an', 'N',
'##9', '##5', 'mask', '.', '[SEP]']
```

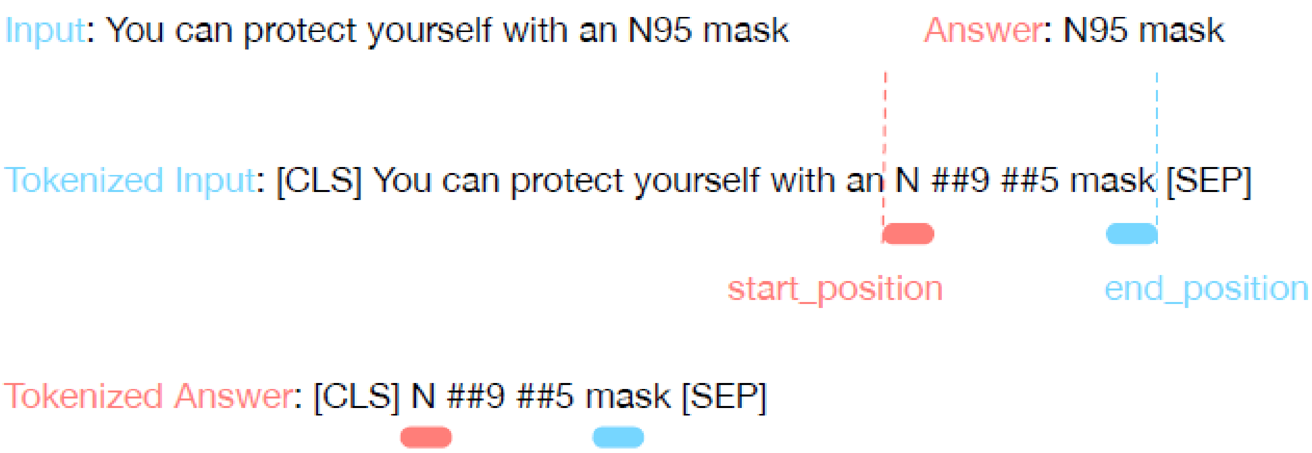
Using BPE for subword information in order to deal with missing words in the training set. Similar to BERT which uses WordPieces. Uses special tokans [CLS] and [SEP] to mark the start & end of the sentence. This has been done in order to train BERT in the global framework of MT, QA and other stuffs.

Notice that:

1. the tokenizer has inserted the special tokens `[CLS]` and `[SEP]` to mark the start and end of the sentence
2. the tokenizer has divided the word "N95" into three *word pieces* `N`, `9` and `5` (refer to the Transformers lecture for a discussion of why we use these kinds of subword units).

Task 1: Complete the `ans_loc()` function

To practice working with tokenized text, implement the function below, which identifies the start and end locations of a tokenized phrase within a larger tokenized string. We'll use this function later to identify start and end locations for answers when we train a question answering model.



For example, if

- Context = [CLS], protect, yourself, with, an, N, ##9, ##5, mask, [SEP]
- Answer = [CLS], N, ##9, ##5, mask, [SEP]

The the answer location is

- Start postition: 5 (N)
- End postion: 8 (mask)

Hint: you need `ctx_enc['offset_mapping']` to pass all test cases. Refer to this [document (https://huggingface.co/transformers/main_classes/tokenizer.html#transformers.PreTrainedTokenizer.__call__)](https://huggingface.co/transformers/main_classes/tokenizer.html#transformers.PreTrainedTokenizer.__call__) for information about `offset_mapping`. Briefly speaking, offset_mapping is the character-level position of each token in the input text. for the `i` -th token in a input sequence,

```
st_char, ed_char = txt_enc['offset_mapping'][i]
token_id = txt_enc['input_ids'][i]
token_txt = txt_raw[st_char: ed_char]
```

In [ ]:

```
def find_index_inside_tuple(index, list_of_tuples):
  for i, tup in enumerate(list_of_tuples):
    if (tup[0] <= index) and (index <= tup[1]):
      return i

def ans_loc(ctx, ans, verbose=False):

    start_loc = 0
    end_loc = 0
    ctx_enc = tokenizer(ctx, return_offsets_mapping=True, verbose=False)
    ans_enc = tokenizer(ans, return_offsets_mapping=True)

    if verbose:
        print('Input', ctx_enc['input_ids'])
        print('Answer', ans_enc['input_ids'])
        print('Input encoded', tokenizer.convert_ids_to_tokens(ctx_enc['input_ids']))
        print('Answer encoded', tokenizer.convert_ids_to_tokens(ans_enc['input_ids']))
    # ------------ Your Code Starts ------------- #
    ans_len = len(ans)
    ctx_num_tok = len(ctx_enc['input_ids'])

    if set(ans).issubset(ctx):
      start = ctx.index(ans)
      start_loc = find_index_inside_tuple(start, ctx_enc["offset_mapping"])
      end_loc = find_index_inside_tuple(start + ans_len, ctx_enc["offset_mapping"])


    # ------------ Your Code Ends --------------- #

    return start_loc, end_loc
```

Test your implementation with the following cases

In [ ]:

```
# ------------- Test Case 1 ---------------- #

print('------------- Test Case 1 -------------')
ctx_c1 = 'You can protect yourself by wearing an N95 mask.'
ans_c1 = 'wearing an N95 mask'

start_loc, end_loc = ans_loc(ctx_c1, ans_c1, verbose=True)
print(f'The start location is {start_loc}, and the end location is {end_loc}')

if (start_loc, end_loc) == (6, 11):
    print('\nYour implementation is correct for case 1')
else:
    print('\nYour implementation failed on case 1')

# ------------- Test Case 2 ---------------- #

print('\n------------- Test Case 2 -------------')
ctx_c2 = 'split with Luckett and Roberson'
ans_c2 = 'Luckett and Rober'

start_loc, end_loc = ans_loc(ctx_c2, ans_c2, verbose=True)
print(f'The start location is {start_loc}, and the end location is {end_loc}')
if (start_loc, end_loc) == (3, 7):
    print('\nYour implementation is correct for case 2')
else:
    print('\nYour implementation failed on case 2')

# ------------- Test Case 3 ---------------- #

print('\n------------- Test Case 3 -------------')
ctx_c2 = 'The UK government has spent £250 million in the construction of the island'
ans_c2 = '250 million'

start_loc, end_loc = ans_loc(ctx_c2, ans_c2, verbose=True)
print(f'The start location is {start_loc}, and the end location is {end_loc}')
if (start_loc, end_loc) == (6, 8):
    print('\nYour implementation is correct for case 3')
else:
    print('\nYour implementation failed on case 3')
```

```
------------- Test Case 1 -------------
Input [101, 1192, 1169, 3244, 3739, 1118, 3351, 1126, 151, 1580, 1571, 773
9, 119, 102]
Answer [101, 3351, 1126, 151, 1580, 1571, 7739, 102]
Input encoded ['[CLS]', 'You', 'can', 'protect', 'yourself', 'by', 'wearin
g', 'an', 'N', '##9', '##5', 'mask', '.', '[SEP]']
Answer encoded ['[CLS]', 'wearing', 'an', 'N', '##9', '##5', 'mask', '[SE
P]']
The start location is 6, and the end location is 11

Your implementation is correct for case 1

------------- Test Case 2 -------------
Input [101, 3325, 1114, 22311, 5912, 1105, 6284, 18608, 102]
Answer [101, 22311, 5912, 1105, 6284, 1200, 102]
Input encoded ['[CLS]', 'split', 'with', 'Luck', '##ett', 'and', 'Rob', '#
#erson', '[SEP]']
Answer encoded ['[CLS]', 'Luck', '##ett', 'and', 'Rob', '##er', '[SEP]']
The start location is 3, and the end location is 7

Your implementation is correct for case 2

------------- Test Case 3 -------------
Input [101, 1109, 1993, 1433, 1144, 2097, 24155, 11049, 1550, 1107, 1103,
2058, 1104, 1103, 2248, 102]
Answer [101, 4805, 1550, 102]
Input encoded ['[CLS]', 'The', 'UK', 'government', 'has', 'spent', '£2',
'##50', 'million', 'in', 'the', 'construction', 'of', 'the', 'island', '[S
EP]']
Answer encoded ['[CLS]', '250', 'million', '[SEP]']
The start location is 6, and the end location is 8

Your implementation is correct for case 3
```

Another useful feature of the tokenizer is the `batch_encode_plus function`, which returns both input IDs and attention masks. For example,

In [ ]:

```
ctx1 = 'I am a short sentence'
ctx2 = 'I am a long long long long long long long sentence'
ctx_list = [ctx1, ctx2]

inputs = tokenizer.batch_encode_plus(
    ctx_list,
    max_length = 12,
    truncation=True,
    padding='longest',
    return_attention_mask=True,
    return_tensors='pt'
)

for key, value in inputs.items():
    print(key)
    print(value)
    print('-------------------------')
```

```
input_ids
tensor([[ 101,  146, 1821,  170, 1603, 5650,  102,    0,    0,    0,    0,
0],
        [ 101,  146, 1821,  170, 1263, 1263, 1263, 1263, 1263, 1263, 1263,
102]])
-------------------------
attention_mask
tensor([[1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]])
-------------------------
```

2.2 Models

After converting the input texts into ids with corresponding attention masks, we can obtain the hidden state of each word by feeding the input IDs into a Transformer network. We will use the output hidden states to predict the start and end locations of the target answer.

Sanity Check: Input and Output Shapes of DistilBertModel

Please read the source code (https://huggingface.co/transformers/_modules/transformers/models/distilbert/modeling_distilbert.html#DistilBe and document (https://huggingface.co/transformers/model_doc/distilbert.html#transformers.DistilBertModel) of the DistilBertModel class.

Suppose we train a question-answering model based on a DistilBertModel. The batch size of the training inputs is `batch_size` , the maximum sequence length of the input batch is `seq_length` . In other words, the input shape of our model is `[batch_size, seq_length]` . Please answer the following questions to examine if you have understood the class we are going to use,

- What is the shape of the top-layer hidden states?
- According to the document (https://huggingface.co/transformers/model_doc/distilbert.html#transformers.DistilBertModel) and the `outputs` above, what are potential outputs of the `DistilBertModel` ?

If you are not sure if your answers are correct, we encourage you to practice with code to examine your answers. (No need to include this in your write-up---this is just here to help you debug).

3 Building a Question Answering Model

Now that we have access to pre-trained representations, let's start building a question answering model!

3.1 Pre-processing the SQuAD data

We preprocess each data point of SQuAD by calculating the start and end positions with the selected tokenizer. In this homework we use the `distilbert-base-cased` model, so we use the `distilbert-base-cased` tokenizer to preprocessing the data. If you want to try other models, please make sure you are using the correct pair of model and tokenizer.

Task 2: Complete the `proc_line()` function

In [ ]:

```
import json
import random
from multiprocessing import Pool
from tqdm import tqdm, trange

def proc_line_init(tokenizer_for_squad):
    global tokenizer
    tokenizer = tokenizer_for_squad

# Preprocess one SQuAD data point
def proc_line(sq):
    ctx = sq['context']
    ans = sq['answers']['text'][0]

    ctx_ids = tokenizer.encode(ctx, verbose=False)
    ans_ids = tokenizer.encode(ans)

    if len(ctx_ids) > 448:
        return None

    start_pos = None
    end_pos = None

    # Get the values of start and end pos
    #
    # ----- Your code: get the start and end positions ------ #
    # ----- with the `ans_loc` function you defined --------- #

    start_pos, end_pos = ans_loc(ctx, ans)

    # --------------- End of your code -------------------- #

    sq['start_position'] = start_pos
    sq['end_position'] = end_pos
    return sq

# Preprocess SQuAD corpus with tqdm multithreading
def preproc(squad_list, threads, tokenizer):

    with Pool(threads, initializer=proc_line_init, initargs=(tokenizer,)) as p:
        squad_proc = list(tqdm(p.imap(proc_line, squad_list), total=len(squad_list)))

    squad_proc = [x for x in squad_proc if x]
    json.dump(squad_proc, open("/content/gdrive/My Drive/6864_hw4/squad_proc.json", 'w'
))
    return squad_proc

squad_list = [x for x in squad['train']]
squad_proc = preproc(squad_list, 16, tokenizer)
```

```
100%|██████████| 87599/87599 [01:20<00:00, 1086.07it/s]
```

We test the correctness of the preprocessed data by randomly selecting data points.

In [ ]:

```
# Test if your preprocessing is correct
def test_preproc(sq, tokenizer):
    gt_ans = sq['answers']['text'][0]
    start_pos = sq['start_position']
    end_pos = sq['end_position']

    tok_outputs = tokenizer(sq['context'], return_offsets_mapping=True)
    ctx_ids = tok_outputs['input_ids']
    offsets = tok_outputs['offset_mapping']

    start_pos_char = offsets[start_pos][0]
    end_pos_char = offsets[end_pos][1]

    pred_ans = sq['context'][start_pos_char: end_pos_char]

    print(f'The annotated answer: {gt_ans} .')
    print(f'The predicted answer: {pred_ans} .')
    if gt_ans == pred_ans:
        print('Pass')
    else:
        print('Something went wrong')

test_preproc(random.choice(squad_proc), tokenizer)
```

```
The annotated answer: annually .
The predicted answer: annually .
Pass
```

Task 3: complete the `QuestionAnsweringModel` class

- Read the document of [DistilBERT.forward() ()] to understand the inputs and outputs of a Huggingface transformer model, and figure out how to extract representations from the encoder.
- Complete the `forward` function. This function should place logits over start positions and end positions (`log p(start | C, Q)` and `log p(end | C, Q)`), and return the predicted logits. It should also return losses if `start_positions` and `end_positions` are provided.
    - Feed the hidden states into the 1. dropout layer, and 2. linear layer to predict the start and end logits
    - Calculate `start_loss` and `end_loss` with `nn.CrossEntropyLoss` with predicted `start/end_logits` and labeled `start/end_positions` as two seperate classification tasks.
    - Return predicted `start_logits, end_logits` and `total_loss = (start_loss + end_loss) / 2` .

In [ ]:

```
import torch.nn as nn

class ModelOutputs:
    def __init__(self, start_logits=None, end_logits=None, loss=None):
        self.start_logits = start_logits
        self.end_logits = end_logits
        self.loss = loss

class QuestionAnsweringModel(nn.Module):

    def __init__(self, lm=None, dropout=0.2):
        '''
        lm:         a pretrained transformer language model
        dropout:    dropoutrate for the dropout layer
        '''
        super(QuestionAnsweringModel, self).__init__()
        self.qa_outputs = nn.Linear(lm.config.dim, 2) # predict the start and the end p
osition inside the question.
        self.attention = nn.Linear(in_features = 7, out_features = 1)
        self.lm = lm
        self.dropout = nn.Dropout(dropout)

    def forward(self, input_ids=None, attention_mask=None,
                start_positions=None, end_positions=None):
        '''
        input_ids:          ids of the concatenated input tokens
        attention_mask:     concatenated attention masks (ques+ctx)
        start_positions:    labels of the start positions of the answers
        end_positions:      labels of the end positions of the answers
        '''
        hiddens = []
        lm_output = self.lm(input_ids=input_ids, attention_mask=attention_mask, output_
hidden_states=True)
        hidden_states = lm_output.hidden_states
        for state in hidden_states:
          hiddens.append(torch.unsqueeze(state, -1))
        tri = torch.cat(hiddens, dim=-1)
        hidden_states = self.attention(tri)[:, :, :, 0]
        # hidden_states = lm_output.hidden_states[0] # size is (batch_size, sequence_le
ngth, hidden_size), we could do better: take the average over the hidden states, it has
proven
        # to work better in the BERT architecture
        hidden_states = self.dropout(hidden_states)

        start_logits = None
        end_logits = None

        # ----- Get start_logits and end_logits ----------------- #
        #
        # start_logits.size() should be [batch_size, seq_len]
        # end_logits.size() should also be [batch_size, seq_len]
        #
        start_logits = self.qa_outputs(hidden_states)[:, :, 0]
        end_logits = self.qa_outputs(hidden_states)[:, :, 1]


        total_loss = None
        if start_positions is not None and end_positions is not None:
```

```
            loss_fct = nn.CrossEntropyLoss()

            # ----- Getting training losses ------------------- #
            #
            # 1. calculate start_losses with
            #       a. start_logits
            #       b. start_positions
            #
            # 2. calculate end_losses with end_logits
            #       a. end_logits
            #       b. end_positions
            #
            # --------- Your code starts ---------------------- #

            start_loss = loss_fct(start_logits, start_positions)
            end_loss = loss_fct(end_logits, end_positions)

            # --------- Your code ends ------------------------ #

            total_loss = (start_loss + end_loss) / 2

        return ModelOutputs(
            start_logits=start_logits,
            end_logits=end_logits,
            loss=total_loss)
```

3.3 Training the DistilBERT Model

First, we move our model to the GPU.

In [ ]:

```
# Initialize the QA model and use GPU
lm_pretrained = transformers.AutoModel.from_pretrained('distilbert-base-cased')
model = QuestionAnsweringModel(lm_pretrained)
model = model.cuda()
```

Then we define the training hyper-parameters, the optimizer, and the learning rate scheduler. Read this [document (https://huggingface.co/transformers/main_classes/optimizer_schedules.html#transformers.get_linear_schedul](https://huggingface.co/transformers/main_classes/optimizer_schedules.html#transformers.get_linear_schedul) to understand how the linear learning rate scheduling influences the learning process.

In [ ]:

```
import torch

# Hyper-parameters: you could try playing with different settings
num_epochs = 1
learning_rate = 3e-5
weight_decay = 1e-5
eps = 1e-6
batch_size = 32
warmup_rate = 0.05
ques_max_length = 64
ctx_max_length = 448

# Calculating the number of warmup steps
num_training_cases = len(squad_proc)
t_total = (num_training_cases // batch_size + 1) * num_epochs
ext_warmup_steps = int(warmup_rate * t_total)

# Initializing an AdamW optimizer
ext_optim = torch.optim.AdamW(model.parameters(), lr=learning_rate,
                              eps=eps, weight_decay=weight_decay)

# Initializing the learning rate scheduler [details are in the BERT paper]
ext_sche = transformers.get_linear_schedule_with_warmup(
    ext_optim, num_warmup_steps=ext_warmup_steps, num_training_steps=t_total
)

print("***** Training Info *****")
print("  Num examples = %d" % t_total)
print("  Num Epochs = %d" % num_epochs)
print("  Batch size = %d" % batch_size)
print("  Total optimization steps = %d" % t_total)
```

```
***** Training Info *****
  Num examples = 2729
  Num Epochs = 1
  Batch size = 32
  Total optimization steps = 2729
```

We need a function that processes batch data into the input format needed by DistilBERT. We first gather contexts, questions, etc in a batch into their corresponding lists.

In [ ]:

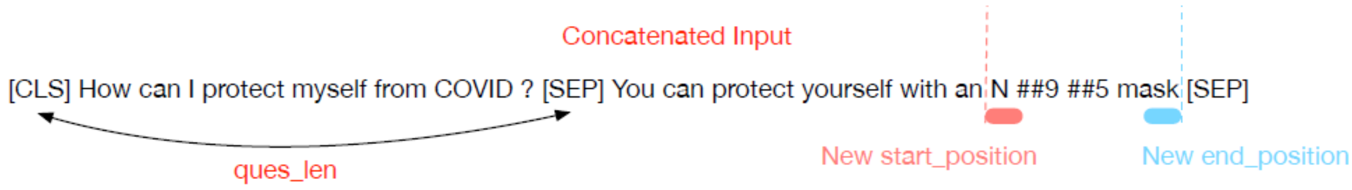
```
def gather_batch(batch):
    ctx_batch  = [x['context'] for x in batch]
    ques_batch = [x['question'] for x in batch]
    ans_batch  = [x['answers']['text'][0] for x in batch]

    start_positions = [x['start_position'] for x in batch]
    end_positions = [x['end_position'] for x in batch]

    return ctx_batch, ques_batch, ans_batch, start_positions, end_positions
```

and then we encode the texts with the DistilBERT tokenizer, and then process the inputs into the following format:

```
[CLS] <question ids> [SEP] <context ids> [SEP]
```

as you can find out, the inputs of our DistilBERT model is the concatenated question word IDs and context word IDs, marked and seperated by special tokens. Note that besides input IDs, we also need to re-organize the attention masks into the same format. We provide the function for this process, and we encourage you to read the code - should be helpful to your course project and future work.

After we concatenate the context to the question, the start and end positions in the context are off by a constant factor. Please fix start_positions and end_positions below.



In [ ]:

```
def vectorize_batch(batch, tokenizer):
    ctx_batch, ques_batch, ans_batch, start_positions, end_positions = gather_batch(bat
ch)

    # Encode the context passage
    ctx_encode = tokenizer.batch_encode_plus(
        ctx_batch,
        max_length = ctx_max_length,
        truncation = True,
        padding = 'longest',
        return_attention_mask = True,
        return_tensors = 'pt'
    )

    # Encode the questions
    ques_encode = tokenizer.batch_encode_plus(
        ques_batch,
        max_length = ques_max_length,
        truncation = True,
        padding = 'longest',
        return_attention_mask = True,
        return_tensors = 'pt'
    )

    # Get the actual sequence lengths of question tensors
    ques_seq_len = ques_encode['input_ids'].size(1)

    # Move the training batch to GPU
    ctx_ids        = ctx_encode['input_ids'].cuda()
    ctx_attn_mask  = ctx_encode['attention_mask'].cuda()
    ques_ids       = ques_encode['input_ids'].cuda()
    ques_attn_mask = ques_encode['attention_mask'].cuda()

    # Remove the [CLS] token of the contexts IDs before concatenation
    ctx_ids = ctx_ids[:, 1:]
    ctx_attn_mask = ctx_attn_mask[:, 1:]

    # Concatenate questions and contexts
    input_ids = torch.cat([ques_ids, ctx_ids], dim=1)
    input_attn_mask = torch.cat([ques_attn_mask, ctx_attn_mask], dim=1)

    # Move start and end positions to the GPU
    start_positions = torch.LongTensor(start_positions).cuda()
    end_positions   = torch.LongTensor(end_positions).cuda()

    # update the start_positions and end_positions variables accordingly
    # This is necessary for the following reasons
    #
    # 1. We concatenated the questions and contexts
    # 2. We removed the [CLS] token (the first token) of the contexts

    start_positions += ques_seq_len - 1
    end_positions += ques_seq_len - 1

    return input_ids, input_attn_mask, start_positions, end_positions
```

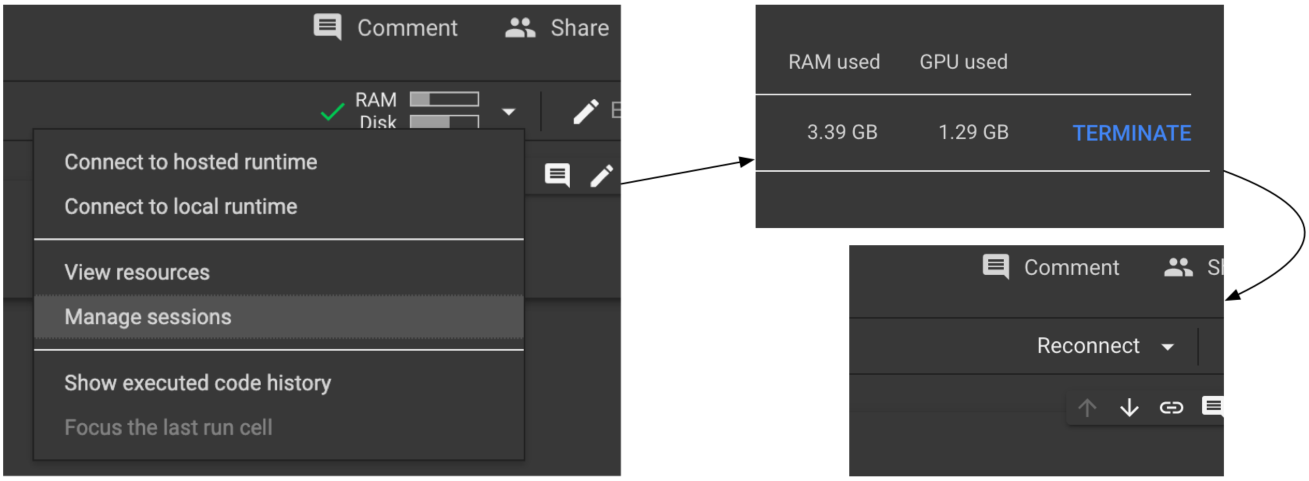
and we can start the training loop, which includes the following steps for processing each mini-batch

- Tokenize questions and contexts, then concatenate the input IDs of questions and corresponding contexts
- Feed the concatenated inputs into the Transformer model
- Optimize the model by back-propagating the loss signals

Task 4: Complete the code for feeding the inputs to the Transformer model

Hint: Read the implementation of the `forward()` method of the `QuestionAnsweringModel` class to decide the input format.

If you get CUDA out of memory error, do "Manage sessions" -> "TERMINATE" -> "Reconnect" -> Re-run necessary code cells



In [ ]:

```
class ModelOutputs:
    def __init__(self, start_logits=None, end_logits=None, loss=None):
        self.start_logits = start_logits
        self.end_logits = end_logits
        self.loss = loss
```

In [ ]:

```
Train = True


if Train:
  model.train()
  max_grad_norm = 1

  '''
  The training can take up to an hour (~50min in average)
  Consider using less training data to validate your implementation
  '''
  #squad_proc = squad_proc[:10000]
  num_training_cases = len(squad_proc)

  step_id = 0
  for _ in range(num_epochs):

      random.shuffle(squad_proc)

      for i in range(0, num_training_cases, batch_size):
          batch = squad_proc[i: i + batch_size]
          input_ids, input_attn_mask, start_positions, end_positions = vectorize_batch(
batch, tokenizer)

          model.zero_grad() # Does the same as ext_optim.zero_grad()

          # Get the model outputs, including (start, end) logits and losses
          # stored as a ModelOutput object
          outputs = model(input_ids, input_attn_mask, start_positions, end_positions)

          # Back-propagate the loss signal and clip the gradients
          loss = outputs.loss.mean()
          loss.backward()
          torch.nn.utils.clip_grad_norm_(model.parameters(), max_grad_norm)

          # Update neural network parameters and the learning rate
          ext_optim.step()
          ext_sche.step() # Update learning rate for better convergence


          if step_id % 100 == 0:
              print(f'At step {step_id}, the extraction loss = {loss}')

          step_id += 1

  print('Finished Training')
  torch.save(model, 'gdrive/MyDrive/6864_hw4/trained_DistillBERT_static_embeddings.pth'
)

else:
  model = torch.load('gdrive/MyDrive/6864_hw4/trained_DistillBERT_1.pth')
```

```
At step 0, the extraction loss = 6.046184539794922
At step 100, the extraction loss = 4.366264343261719
At step 200, the extraction loss = 2.6180243492126465
At step 300, the extraction loss = 2.6466798782348633
At step 400, the extraction loss = 1.630646824836731
At step 500, the extraction loss = 2.510709285736084
At step 600, the extraction loss = 1.9398633241653442
At step 700, the extraction loss = 1.8172781467437744
At step 800, the extraction loss = 1.9630472660064697
At step 900, the extraction loss = 1.868171215057373
At step 1000, the extraction loss = 1.9408320188522339
At step 1100, the extraction loss = 1.3014763593673706
At step 1200, the extraction loss = 1.7652891874313354
At step 1300, the extraction loss = 1.7547430992126465
At step 1400, the extraction loss = 1.2727305889129639
At step 1500, the extraction loss = 1.457627534866333
At step 1600, the extraction loss = 1.4718434810638428
At step 1700, the extraction loss = 1.8858411312103271
At step 1800, the extraction loss = 1.524166226387024
At step 1900, the extraction loss = 1.7534606456756592
At step 2000, the extraction loss = 1.601196527481079
At step 2100, the extraction loss = 1.4634039402008057
At step 2200, the extraction loss = 1.8657045364379883
At step 2300, the extraction loss = 2.1398770809173584
At step 2400, the extraction loss = 1.560817003250122
At step 2500, the extraction loss = 1.6034340858459473
At step 2600, the extraction loss = 1.9455184936523438
At step 2700, the extraction loss = 1.6933109760284424
Finished Training
```

4 Evaluating the Learned QA Model

Standard evaluation metrices for extractive, or span-based QA models are exact match (EM) and F1 scores.

- EM: how many predicted answers are exactly the same as the annotated answers
- F1: how many words in the predicted answers overlap the annotated answers.

In other words, the calculations of EM and F1 scores are:

- EM = num_same_answer / num_all_questions
- F1 = (2 *precision* recall) / (precision + recall), where
  - precision = num_overlap_words / num_predicted_answer_words
  - recall = num_overlap_words / num_annotated_answer_words

In [ ]:

```
def ans_pair_metric(a_pred, a_gt, tokenizer):
    '''
    a_pred: the predicted answer text
    a_gt: the groundtruth answer text
    '''

    # Exclude the special tokens
    pred_ids = tokenizer.encode(a_pred)[1: -1]
    gt_ids = tokenizer.encode(a_gt)[1: -1]

    len_pred = len(pred_ids)
    len_gt = len(gt_ids)
    num_same = 0
    for word_id in pred_ids:
        if word_id in gt_ids:
            num_same += 1.

    em = float(a_pred == a_gt)
    if num_same == 0:
        f1 = 0
    else:
        prec = num_same / len_pred
        recall = num_same / len_gt
        f1 = 2 * prec * recall / (prec + recall)

    return em, f1
```

In SQuAD, some context passages are annotated with many possible answers that are considered correct, and we compare our predicted answer with the most similar annotated answer.

In [ ]:

```
import numpy as np

def one_to_many_metric(a_pred, a_gt_list, tokenizer):
    '''
    a_pred: the predicted answer text
    a_gt_list: the provided ground-truth answer list
    '''
    metric = np.array([ans_pair_metric(a_pred, x, tokenizer) for x in a_gt_list])

    em = metric[:, 0]
    f1 = metric[:, 1]
    return em.max(), f1.max()
```

We will also need to have a function that infers the locations of answers based on the predicted start and end locations.

Task 5: complete the `logits_to_ans_loc` function

`logits_to_ans_loc` is a decoding function that converts predicted start and end logits to an actual answer span (i, j), where

- word w_i has a high `start_logit`
- word w_j has a high `end_logit` .

Inputs and outputs

- inputs: `start_logits and end_logits`
- outputs
  - st_loc: the index of the start TOKEN (not character) of the predicted answer
  - ed_loc: the index of the end TOKEN of the predicted answer

Note:

- we don't consider answers more than 30 tokens
- Make sure `ed_loc >= st_loc`
- Higher start/end logits stands for higher probability that a word could be the start/end point of an answer
- Please implement three strategies for getting answer span (i, j)
  - `greedy_left_to_right` - Select $i = argmax_i\ S^i_{start}$, then select $j = argmax_j\ S^j_{end}$. $(i \leq j)$
  - `greedy_right_to_lett` - Select $j = argmax_j\ S^j_{end}$, then select $i = argmax_i\ S^i_{start}$ $(i \leq j)$
  - `joint` - Select (i, j) by
$$i, j = argmax_{i,j}\ S^i_{start} + S^j_{end}\ (i \leq j)$$
  - Compare the performance

In [ ]:

```
def logits_to_ans_loc(start_logits, end_logits, mode='joint'):
    '''

    Input sizes -
        start_logits.size() = [batch_size, seq_len]
        end_logits.size() = [batch_size, seq_len]

    Output sizes -
        st_loc.size() = (batch_size,)
        ed_loc.size() = (batch_size,)

    '''
    bs, seq_len = start_logits.size()

    st_loc = None
    ed_loc = None

    # Find the span (i, j) that could be an answer
    # to the question, based on the predicted
    # start_logits and end_logits with three modes
    #   - greedy_left_to_right
    #   - greedy_right_to_left
    #   - joint
    #
    # ------------ Your code starts -------------- #

    # tensor( [[0, 1, ..., seq_len - 1]] )
    # pos_idx.size() = (1, seq_len)
    pos_idx = torch.range(0, seq_len - 1).cuda().unsqueeze(0)

    if mode == 'greedy_left_to_right':
        # Your code #
        st_loc = torch.argmax(start_logits, axis=1)
        ed_loc = []
        for i, st in enumerate(st_loc):
          ed = st + torch.argmax(end_logits[i, st:])
          ed_loc.append(ed)
        ed_loc = torch.tensor(ed_loc)
    if mode == 'greedy_right_to_left':
        # Your code #
        ed_loc = torch.argmax(end_logits, axis=1)
        st_loc = []
        for i, ed in enumerate(ed_loc):
          st = torch.argmax(start_logits[i, :ed+1])
          st_loc.append(st)
        st_loc = torch.tensor(st_loc)
    if mode == 'joint':
        # Your code #
        start_indices = []
        end_indices = []
        for b in range(bs):
          start_logit = start_logits[b]
          end_logit = end_logits[b]
          matrix = torch.unsqueeze(start_logit, 1) + torch.unsqueeze(end_logit, 0)
          considered_matrix = torch.triu(matrix) - torch.triu(matrix, 30)
          considered_matrix[considered_matrix==0] = -10e9
          indices = torch.argmax(considered_matrix).item()
          start_indices.append(indices//seq_len)
          end_indices.append(indices%seq_len)
```

```
        st_loc = torch.tensor(start_indices)
        ed_loc = torch.tensor(end_indices)



    # ------------ Your code ends ---------------- #

    return st_loc, ed_loc
```

and now we start implementing the evaluation loop

Task 6: complete the evaluation loop

In [ ]:

```
model.eval()

# Prepare the dev set of SQuAD for evaluation
dev_set = [x for x in squad['validation']]
num_dev_cases = len(dev_set)

eval_batch_size = 64

# `ans_pred_list` stores the predicted answers
# in the same order as the contexts of the dev set
ans_pred_list_ltr = []
ans_pred_list_rtl = []
ans_pred_list_joint = []
ans_gt_list = [x['answers']['text'] for x in dev_set]

for i in range(0, num_dev_cases, eval_batch_size):
    eval_batch = dev_set[i: i + eval_batch_size]
    ques = [x['question'] for x in eval_batch]
    ctx  = [x['context'] for x in eval_batch]

    # Encode the contexts
    ctx_encode = tokenizer.batch_encode_plus(
        ctx,
        max_length = ctx_max_length,
        truncation = True,
        padding = 'longest',
        return_attention_mask = True,
        return_tensors = 'pt',
        return_offsets_mapping = True
    )

    # Encode the questions
    ques_encode = tokenizer.batch_encode_plus(
        ques,
        max_length = ques_max_length,
        truncation = True,
        padding = 'longest',
        return_attention_mask = True,
        return_tensors = 'pt'
    )

    # get the actual question sequence lengths
    ques_len = ques_encode['input_ids'].size(1)

    # ------------- Your code Part 1 -------------- #
    #
    # concatenate the input ids and attention masks
    # of questions and contexts. Refer to the training
    # loop for implementation hints
    #
    # -------------------------------------------- #

    input_ids = torch.cat([ques_encode['input_ids'], ctx_encode['input_ids'][:, 1:]], d
im =1).cuda()
    input_attn_mask = torch.cat([ques_encode['attention_mask'], ctx_encode['attention_m
ask'][:, 1:]], dim = 1).cuda()

    # ------------- Your code Part 1 ends ---------- #
```

```
    with torch.no_grad():
        outputs = model(
            input_ids,
            attention_mask = input_attn_mask,
        )

    # ------------- Your code Part 2 --------------- #
    #
    # Obtain the predicted start and end logits
    # We drop the start and end logits of question tokens
    # and only keep the logits of
    #
    #       [SEP] or [PAD], ctx_1, ctx_2, ..., [SEP]
    #
    # keep the first special token, which is the tail
    # or padding token of the tokenized question, to
    # help you do later decoding more easily
    #
    # ---------------------------------------------- #
    start_logits_pred = outputs.start_logits[:, ques_len-1:]
    end_logits_pred = outputs.end_logits[:, ques_len-1:]
    # ------------- Your code Part 2 ends ---------- #

    st_locs_ltr, ed_locs_ltr = logits_to_ans_loc(
        start_logits_pred, end_logits_pred, mode='greedy_left_to_right'
    )

    st_locs_rtl, ed_locs_rtl = logits_to_ans_loc(
        start_logits_pred, end_logits_pred, mode='greedy_right_to_left'
    )

    st_locs_joint, ed_locs_joint = logits_to_ans_loc(
        start_logits_pred, end_logits_pred, mode='joint'
    )

    num_pred_answer = st_locs_ltr.size(0)

    # ---------------------------------------------- #
    #
    # Store predicted answer texts in `ans_pred_list`
    # 1. `ans_pred_list` should look like
    #           ['ans_txt_1', 'ans_txt_2', ....]
    #
    # 2. `len(ans_pred_list)` should equas to
    #           `len(dev_set)`
    #
    # ---------------------------------------------- #

    for j in range(num_pred_answer):
        st_char_ltr = ctx_encode['offset_mapping'][j][st_locs_ltr[j]][0]
        ed_char_ltr = ctx_encode['offset_mapping'][j][ed_locs_ltr[j]][1]
        ans_pred_list_ltr.append(ctx[j][st_char_ltr: ed_char_ltr])

        st_char_rtl = ctx_encode['offset_mapping'][j][st_locs_rtl[j]][0]  # ERROR HERE,
I FIXED IT
        ed_char_rtl = ctx_encode['offset_mapping'][j][ed_locs_rtl[j]][1]
        ans_pred_list_rtl.append(ctx[j][st_char_rtl: ed_char_rtl])
```

```
        st_char_joint = ctx_encode['offset_mapping'][j][st_locs_joint[j]][0]
        ed_char_joint = ctx_encode['offset_mapping'][j][ed_locs_joint[j]][1]
        ans_pred_list_joint.append(ctx[j][st_char_joint: ed_char_joint])

    # ---------------------------------------------- #

# Print the evaluation results
#
# The performance is decided by both training quality
# AND the implementation of the `logits_to_anc_loc` function

# Target result
# - EM: 67.97%
# - F1: 80.89%

"""
EM = 0.6393566669820246
F1 = 0.7910730123756418
"""

print('Evaluating greedy_left_to_right strategy')
metric = np.array([one_to_many_metric(x, y, tokenizer) for x, y in zip(ans_pred_list_lt
r, ans_gt_list)])
em = metric[:, 0].mean()
f1 = metric[:, 1].mean()

print(f'EM = {em}')
print(f'F1 = {f1}')

# Target result
# - EM: 67.97%
# - F1: 80.89%

print('\nEvaluating greedy_right_to_left strategy')
metric = np.array([one_to_many_metric(x, y, tokenizer) for x, y in zip(ans_pred_list_rt
l, ans_gt_list)])
em = metric[:, 0].mean()
f1 = metric[:, 1].mean()

print(f'EM = {em}')
print(f'F1 = {f1}')

# Target result
# - EM: 68.99%
# - F1: 81.77%

"""
EM = 0.6255439924314097
F1 = 0.7671115806330859
"""

print('\nEvaluating joint strategy')
metric = np.array([one_to_many_metric(x, y, tokenizer) for x, y in zip(ans_pred_list_jo
int, ans_gt_list)])
em = metric[:, 0].mean()
f1 = metric[:, 1].mean()

print(f'EM = {em}')
print(f'F1 = {f1}')
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:29: UserWarni
ng: torch.range is deprecated and will be removed in a future release beca
use its behavior is inconsistent with Python's range builtin. Instead, use
torch.arange, which produces values in [start, end).

Evaluating greedy_left_to_right strategy
EM = 0.6328287606433302
F1 = 0.7826945696972887

Evaluating greedy_right_to_left strategy
EM = 0.6327341532639545
F1 = 0.7810520066817371

Evaluating joint strategy
EM = 0.6411542100283822
F1 = 0.7876285578569755
```