In the name of God

University of Tehran
College of Engineering
(School of Electrical and
Computer Engineering)

Natural Language Processing

CA Number: 6

Subject: QA and NLU

Full name: Mohammadreza Bakhtiari

Student Number: 810197468

(**Individual**)

June 2022

Contents

1-Question Answering

Introduction:

Reading comprehension, otherwise known as question answering systems, are one of the tasks that NLP tries to solve. The goal of this task is to be able to answer an arbitrary question given a context. For instance, given the following context:

"نیوزلند یک کشور جزیره ای مستقل در جنوب غربی اقیانوس آرام است. مساحت آن 268000 کیلومتر مربع (103500 مایل مربع) و 4.9 میلیون نفر جمعیت دارد. پایتخت نیوزلند ولینگتون و پرجمعیت ترین شهر آن اوکلند است."

We ask the question:

"چند نفر در نیوزلند زندگی می کنند؟"

We expect the QA system is to respond with something like this:

"4.9 میلیون"

Since 2017, transformer models have been shown to outperform existing approaches for this task. Many pretrained transformer models exist, including BERT, GPT-2, and XLNET. One of the newcomers to the group is ALBERT (A Lite BERT) which was published in September 2019. The research group claims that it outperforms BERT, with much fewer parameters (shorter training and inference time).

In this project, I want to demonstrate how you can fine-tune ALBERT/BERT for the task of QnA and use it for inference. For this project, I will use the transformer library built by Hugging Face, which is an extremely nice implementation of the transformer models (including ALBERT/BERT) in both TensorFlow and PyTorch. You can just use a fine-tuned model from their model repository.

Setup:

To get started, we first need to make the necessary settings and install the required packages during the project and check out what kind of GPU Google gave us.

Setup

```
[11] !pip install nltk
     !pip install random
     !pip install spellchecker
     !pip install pyspellchecker
     !pip install bpe
     !nvidia-smi
```

Fig. 1.1: Install the required packages

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 460.32.03    Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   46C    P0    27W /  70W |   1852MiB / 15109MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
+-----------------------------------------------------------------------------+
```

Fig. 1.2: GPU information

Importing important packages:

Then we should import the required packages:

Importing important packages

```
[16] import nltk
     nltk.download('punkt')
     from nltk import word_tokenize
     import random
     from spellchecker import SpellChecker
     import re
     from typing import Dict, Tuple
     from bpe import Encoder
```

Fig. 1.3: Import important packages

Loading dataset:

After that we need to load our dataset:

```
                                        # ParSQuAD
# automatic
  # Train :
            # https://www.dropbox.com/s/63l6iv2sz9ui8ld/train-v2.0.json?dl=1
  # Dev :
            # https://www.dropbox.com/s/iz7xzp9zrdgau5w/dev-v2.0.json?dl=1
# manual
  # Train :
            # https://www.dropbox.com/s/63l6iv2sz9ui8ld/train-v2.0.json?dl=1
  # Dev :
            # https://www.dropbox.com/s/iz7xzp9zrdgau5w/dev-v2.0.json?dl=1
# ------------------------------------------------------------------------------
```

Fig. 1.4: ParSQuAD dataset

```
                                        # pquad_public
  # Train :
            # https://www.dropbox.com/s/63l6iv2sz9ui8ld/train-v2.0.json?dl=1
  # Test :
            # https://www.dropbox.com/s/drhtfb6sfvjn1ic/test-v2.0.json?dl=1
  # Dev :
            # https://www.dropbox.com/s/iz7xzp9zrdgau5w/dev-v2.0.json?dl=1
# ------------------------------------------------------------------------------
```

Fig. 1.5: pquad_public dataset

```
                                        # PersianQA
  # Train :
            # https://www.dropbox.com/s/63l6iv2sz9ui8ld/train-v2.0.json?dl=1
  # Dev :
            # https://www.dropbox.com/s/iz7xzp9zrdgau5w/dev-v2.0.json?dl=1
```

Fig. 1.6: PersianQA dataset

```
!mkdir dataset \
&& cd dataset \
&& wget https://www.dropbox.com/s/63l6iv2sz9ui8ld/train-v2.0.json?dl=1 \
&& wget https://www.dropbox.com/s/iz7xzp9zrdgau5w/dev-v2.0.json?dl=1
# Loading dataset
file_train = open("/content/dataset/train-v2.0.json?dl=1", "r")
train = file_train.read()
file_dev = open("/content/dataset/dev-v2.0.json?dl=1", "r")
dev = file_dev.read()
```

Fig. 1.7: Loading the desired dataset

Now let's take a look at the data to select the appropriate preprocessors:

```
[24] train

'{\n  "data": [\n    {\n      "title": "شرکت فولاد مبارکه اصفهان",\n      "paragraphs": [\n        {\n          "qas": [\n
{\n              "answers": [\n                {\n                  "answer_start": 114,\n                  "answer_end":
131,\n                  "text": "در شرق شهر مبارکه"\n                }\n              ],\n              "question": "شرکت فولاد
مبارکه در کجا واقع شده است",\n              "is_impossible": false,\n              "id": 1\n            },\n            {\n
"answers": [\n                {\n                  "answer_start": 263,\n                  "answer_end": 264,\n
"text": "۶"\n                }\n              ],\n              "question": "فولاد مبارکه چند بار برنده جایزه شرکت دانشی را کسب کرده اس
ت؟",\n              "is_impossible": false,\n              "id": 2\n            },\n            {\n              "answer
s": [\n                {\n                  "answer_start": 413,\n                  "answer_end"... '
```

Fig. 1.8: ParSQuAD training dataset

Now look at examples of conventional preprocessing in natural language processing, and then select and implement the appropriate preprocessing in accordance with the ultimate goal, which is to train a model of question answering.

Text Pre-Processing:

0- BPE[1]

The BPE algorithm seeks to find a pair of characters in each step and combine them and add a new word to the vocabulary. The algorithm selects the most repeated pairs of characters in the text at each step.

We implement this algorithm once manually and once using existing libraries.

-Implement BPE manually:

In the first step, from the given set of texts, we extract the initial words that contain the unique letters used in the text and form a set of words.

Before starting work, we add the '_' character to the end of each word to indicate the end of each word.

In each step, find the two pairs of words that come with the most repetitions, and after combining them and specifying them as a new word, add that pair to the vocabulary and so on. We continue.

The condition for the end of this process can be considered in two ways. We can consider a certain number of repetitions and end the process after the loop is repeated to the desired number. Or we can consider a specific condition on the number of words in the vocabulary from the beginning.

A- *Implement BPE manually*

```
[19] # Find all unique letters in the corpus
     def Find_Vocabulary(Dictionary):
         Vocabulary = []
         for word in Dictionary:
             for string in word:
                 Vocabulary.append(string)
         Vocabulary = list(dict.fromkeys(Vocabulary))
         Vocabulary.remove(' ')
         return Vocabulary

     # Find all most frequent pairs in the corpus
     def Find_Most_Frequent_Pair(My_Dictionary: Dict[str, int]) -> Dict[Tuple[str, str], int]:
         Pair = {}
         for Vocab, Frequency in My_Dictionary.items():
             Letter = Vocab.split()
             for i in range(len(Letter) - 1):
                 pair = (Letter[i], Letter[i + 1])
                 New_Frequency = Pair.get(pair, 0)
                 Pair[pair] = New_Frequency + Frequency
         return Pair

     # Merge most frequent Consecutive letters into one word
     def Combine_Vocabulary(Most_Frequent_Pair: Tuple[str, str], Input_Vocabulary: Dict[str, int]) -> Dict[str, int]:
         Merged_Vocabulary = {}
         pattern = re.escape(' '.join(Most_Frequent_Pair))
         Substitution = ''.join(Most_Frequent_Pair)
         for Input_Word in Input_Vocabulary:
             Output_Word = re.sub(pattern, Substitution, Input_Word)
             Merged_Vocabulary[Output_Word] = Input_Vocabulary[Input_Word]
         return Merged_Vocabulary
```

Fig. 1.9: Implement BPE manually

[1] Byte Pair Encoding

-Implementation of BPE using the library:

Once again, we implement this algorithm using existing libraries:

B- *Implement BPE using library*

```
[26] encoder = Encoder(200, pct_bpe=0.88)  # params chosen for demonstration purposes
     encoder.fit(train.split('\n'))
```

Fig. 1.10: Implement BPE using existing libraries

Now, for example, we give a custom sentence as input and see the output:

```
[27] # Let's see an example
     example = "Let's see an example ."
     print(encoder.tokenize(example))
     print(next(encoder.transform([example])))
     print(next(encoder.inverse_transform(encoder.transform([example]))))
```

Fig. 1.11: Applying BPE on a desired sentence

And you see the following output with the number of repetitions as well as the input sentence after applying BPE:

['__sow', 'l', 'e', 't', '__eow', '__sow', '__unk', '__eow', '__sow', 's', '__eow', '__sow', 'se', 'e', '__eow', '__sow', 'an', '__eow', '__sow', 'ex', 'a', '__unk', '__unk', 'l', 'e', '__eow', '.']
[25, 83, 40, 43, 24, 25, 0, 24, 25, 51, 24, 25, 105, 40, 24, 25, 99, 24, 25, 88, 47, 0, 0, 83, 40, 24, 20]

"let __unk s see an exa__unk__unkle ."

In this project, we will use the second implementation (the implementation using the library)

1- Lowercasing

One of the most important preprocessors, especially for QA, is lowercasing.

1- *Lowercasing*

```
[28] # Lowercasing source dataset
     train = train.lower()
     dev = dev.lower()
```

Fig. 1.12: Lowercasing using .lower()

2- Remove Extra Whitespaces

Another preprocessor considered in QA is the removal of extra spaces that are available in English texts as white spaces (for example, " I go to school", which is converted to "I go to school" by removing the space before I) And in Persian it is referred to as space / half space.

2- *Remove Extra Whitespaces*

```
[29] def remove_whitespace(text):
         return  " ".join(text.split())

     # Removing Extra Whitespaces
     train = remove_whitespace(train)
     dev = remove_whitespace(dev)
```

Fig. 1.13: Remove extra whitespaces

These spaces do not give us any additional information and only increase the size of the text and to prevent this, we use the function shown to remove these extra spaces.

3- Tokenization

Another important and necessary preprocessor in language processing is tokenize the input text. In this way, we divide the given text into smaller parts called tokens.

3- *Tokenization*

```
[30] # Tokenization of source dataset
     train_token = word_tokenize(train)
     dev_token = word_tokenize(dev)
```

Fig. 1.14: Tokenize datasets

Now let's take a look at the tokens generated:

```
[31] # Let's take a look at some random tokens
     print("Some of the tokens at the train data set :")
     print(random.choices(train_token, k=10))
     print("Some of the tokens at the dev data set :")
     print(random.choices(dev_token, k=10))
```

```
Some of the tokens at the train data set :
[':', '```', ',', '{', 'مردم', '[', ']', 'is_impossible', 'ثوابت', ':']
Some of the tokens at the dev data set :
['با' ,'.' ,':' ,'و' ,'اروپا', '[', '{', 'id', '```', 'false']
```

Fig. 1.15: Random sample of generated tokens

In this section, we used tokenize the text at the word level, according to the existing needs and tasks. (In general, it is possible to tokenize at the sentence and character level)

4- Spelling Correction

Another important preprocessor in QA is word correction. Typing errors are common in textual databases, and we may want to correct existing spelling mistakes before performing the analysis, which ensures that we will get better results from our model.

4- *Spelling Correction*

```
[33] def spell_check(text):
         result = []
         spell = SpellChecker()
         for word in text:
             correct_word = spell.correction(word)
             result.append(correct_word)
         return result

     # Spelling Correction
     train_split = train.split()
     #train_Spelling_Correction = spell_check(train_split)
     dev_split = dev.split()
     #dev_Spelling_Correction = spell_check(dev_split)
```

Fig. 1.16: Correct spelling of words using the SpellChecker() command

The following is an example of spelling correction:

```
[6]  # Let's try an example of spelling correction
     text = "confuson matrx".split()
     spell_check(text)

     ['confusion', 'matrix']
```

Fig. 1.17: An example of correcting the spelling of English words

To have a good model, we need the existing data to have the lowest error rate so that the model can be well trained and achieve higher accuracy, so diagnosing and then correcting existing errors is an important and necessary task in the QA task.

5- Removal of Tags

If our collection of data is such that it contains multiple tags (HTML tags, ...) so that the existing tags do not have a meaningful load and do not provide us with additional information, it is better to use this preprocessing. And remove extra tags to prevent text from enlarging.

5- *Removal of Tags*

```
[35] def remove_tag(text):
         text=' '.join(text)
         html_pattern = re.compile('<.*?>')
         return html_pattern.sub(r'', text)

     # Removal of Tags of source dataset
     train_ = remove_tag(train.split())
     dev = remove_tag(dev.split())
```

Fig. 1.18: Removal of Tags

Here is an example of tag removal:

```
[19] # Let's see an example of "Removal of Tags"
     text = "<HEAD> this is head tag </HEAD>"
     print("The input text is :")
     print(text,"\n")
     print("The output text after removing tags is :")
     print(remove_tag(text.split()))

     The input text is :
     <HEAD> this is head tag </HEAD>

     The output text after removing tags is :
      this is head tag
```

Fig. 1.19: An example of input text with its output after deleting the tag

Here we will not have to use this preprocessing because our text was in a way that did not contain specific tags.

6- Other Pre-processing

There are other preprocessors such as Stopwords Removal, Removing Punctuations, Removal of Frequent Words, Stemming, Lemmatization, Removal of URLs, some of which are less important or do not fit our ultimate goal (QA model) and in Some will even reduce the accuracy of the model.

First, we clone the Hugging Face transformer library from Github.

```
!git clone https://github.com/huggingface/transformers \
&& cd transformers \
&& git checkout a3085020ed0d81d4903c50967687192e3101e770
```

Fig. 1.20: cloning the Hugging Face transformer library from Github

Train Model:

This is where we can train our own model.

Get Training and Evaluation Data:

Our dataset contains question/answer pairs to for training the ALBERT/BERT model for the QA task.

Now get our dataset. train-v2.0.json is for training and dev-v2.0.json is for evaluation to see how well your model trained.

Run training:

We can now train the model with the training set.

Notes about parameters:

per_gpu_train_batch_size: specifies the number of training examples per iteration per GPU. In general, higher means more accuracy and faster training. However, the biggest limitation is the size of the GPU. 12 is what I use for a GPU with 16GB memory.

save_steps: specifies number of steps before it outputs a checkpoint file. I've increased it to save disk space.

num_train_epochs: I recommend two epochs here. It's currently set to one for the purpose of time.

Run training

```
[5]  !export SQUAD_DIR=/content/dataset \
     && python transformers/examples/run_squad.py \
       --model_type albert \
       --model_name_or_path albert-base-v2 \
       --do_train \
       --do_eval \
       --do_lower_case \
       --train_file $SQUAD_DIR/train-v2.0.json?dl=1 \
       --predict_file $SQUAD_DIR/dev-v2.0.json?dl=1 \
       --per_gpu_train_batch_size 12 \
       --learning_rate 3e-5 \
       --num_train_epochs 1.0 \
       --max_seq_length 384 \
       --doc_stride 128 \
       --output_dir /content/model_output \
       --save_steps 1000 \
       --threads 4 \
       --version_2_with_negative
```

Fig. 1.21: Training script

And here are the results and all metrics for each dataset, training on ALBERT(ALBERT-Persian) and BERT(ParsBERT) model:

ParSQuAD:

ALBERT:

Results: {'exact': 49.79253112033195, 'f1': 62.21746803873967, 'total': 964, 'HasAns_exact': 43.24693042291951, 'HasAns_f1': 59.58750230470004, 'HasAns_total': 733, 'NoAns_exact': 70.56277056277057, 'NoAns_f1': 70.56277056277057, 'NoAns_total': 231, 'best_exact': 49.79253112033195, 'best_exact_thresh': 0.0, 'best_f1': 62.217468038739625, 'best_f1_thresh': 0.0}

BERT:

Results: {'exact': 51.6597510373444, 'f1': 63.96133908315053, 'total': 964, 'HasAns_exact': 48.43110504774898, 'HasAns_f1': 64.609455492711, 'HasAns_total': 733, 'NoAns_exact': 61.904761904761905, 'NoAns_f1': 61.904761904761905, 'NoAns_total': 231, 'best_exact': 51.76348547717842, 'best_exact_thresh': 0.0, 'best_f1': 64.06507352298456, 'best_f1_thresh': 0.0}

PersianQA:

ALBERT:

Results: {'exact': 53.63070539419087, 'f1': 67.75508733355822, 'total': 964, 'HasAns_exact': 41.06412005457026, 'HasAns_f1': 59.639705579195315, 'HasAns_total': 733, 'NoAns_exact': 93.50649350649351, 'NoAns_f1': 93.50649350649351, 'NoAns_total': 231, 'best_exact': 53.63070539419087, 'best_exact_thresh': 0.0, 'best_f1': 67.75508733355822, 'best_f1_thresh': 0.0}

BERT:

Results: {'exact': 55.29045643153527, 'f1': 68.71182306247117, 'total': 964, 'HasAns_exact': 42.701227830832195, 'HasAns_f1': 60.352247520084866, 'HasAns_total': 733, 'NoAns_exact': 95.23809523809524, 'NoAns_f1': 95.23809523809524, 'NoAns_total': 231, 'best_exact': 55.29045643153527, 'best_exact_thresh': 0.0, 'best_f1': 68.7118230624712, 'best_f1_thresh': 0.0}

pquad_public:

ALBERT:

Results: {'exact': 60.995850622406635, 'f1': 74.91810360889745, 'total': 964, 'HasAns_exact': 51.29604365620737, 'HasAns_f1': 69.60580065344776, 'HasAns_total': 733, 'NoAns_exact': 91.77489177489177, 'NoAns_f1': 91.77489177489177, 'NoAns_total': 231, 'best_exact': 61.09958506224066, 'best_exact_thresh': 0.0, 'best_f1': 75.02183804873148, 'best_f1_thresh': 0.0}

BERT:

Results: {'exact': 63.27800829875519, 'f1': 76.5968135878235, 'total': 964, 'HasAns_exact': 55.38881309686221, 'HasAns_f1': 72.90494992996163, 'HasAns_total': 733, 'NoAns_exact': 88.31168831168831, 'NoAns_f1': 88.31168831168831, 'NoAns_total': 231, 'best_exact': 63.27800829875519, 'best_exact_thresh': 0.0, 'best_f1': 76.59681358782353, 'best_f1_thresh': 0.0}

Mix: pquad_public & PersianQA:

ALBERT:

Results: {'exact': 62.344398340248965, 'f1': 76.2921818289585, 'total': 964, 'HasAns_exact': 52.796725784447474, 'HasAns_f1': 71.14005904927151, 'HasAns_total': 733, 'NoAns_exact': 92.64069264069263, 'NoAns_f1': 92.64069264069263, 'NoAns_total': 231, 'best_exact': 62.344398340248965, 'best_exact_thresh': 0.0, 'best_f1': 76.2921818289585, 'best_f1_thresh': 0.0}

BERT:

Results: {'exact': 64.83402489626556, 'f1': 78.79446408356903, 'total': 964, 'HasAns_exact': 56.343792633015006, 'HasAns_f1': 74.70376995438004, 'HasAns_total': 733, 'NoAns_exact': 91.77489177489177, 'NoAns_f1': 91.77489177489177, 'NoAns_total': 231, 'best_exact': 64.83402489626556, 'best_exact_thresh': 0.0, 'best_f1': 78.79446408356905, 'best_f1_thresh': 0.0}

Setup prediction code:

Now we can use the Hugging Face library to make predictions using our newly trained model. Note that a lot of the code is pulled from run_squad.py in the Hugging Face repository, with all the training parts removed. This modified code allows running predictions we pass in directly as strings, rather a .json format like the training/test set.

Run predictions:

Now for the last part, we can test out our model on different inputs. Pretty rudimentary example here. But the possibilities are endless with this function.

Run predictions

```
[15] context = "New Zealand (Māori: Aotearoa) is a sovereign island country in the southwestern Pacific Ocean. It has a total
     questions = ["How many people live in New Zealand?",
                  "What's the largest city?"]

     # Run method
     predictions = run_prediction(questions, context)

     # Print results
     for key in predictions.keys():
       print(predictions[key])

     convert squad examples to features: 100%|██████████| 2/2 [00:00<00:00, 214.12it/s]
     add example index and unique id: 100%|██████████| 2/2 [00:00<00:00, 14665.40it/s]
     4.9 million.
     Auckland
```

Fig. 1.22: a simple example of QA

Attachment

- All requests and questions raised in the project form are fully answered separately in each section. Thank you in advance for reading the report patiently.
- All .ipynb files are available in the Codes folder. Also, all the codes have been executed and tested in the colab environment.
- Due to the conditions and implementations in this project and sometimes the heavy volume of processing, the execution of some cells will be time consuming, so thank you in advance for your patience and endurance to execute the codes.