NLP for Question Answering

Building a QA System with BERT on Wikipedia

A high-level code walk-through of an IR-based QA system with PyTorch and Hugging Face.

May 19, 2020 • 23 min read

PyTorch Hugging Face Wikipedia BERT Transformers

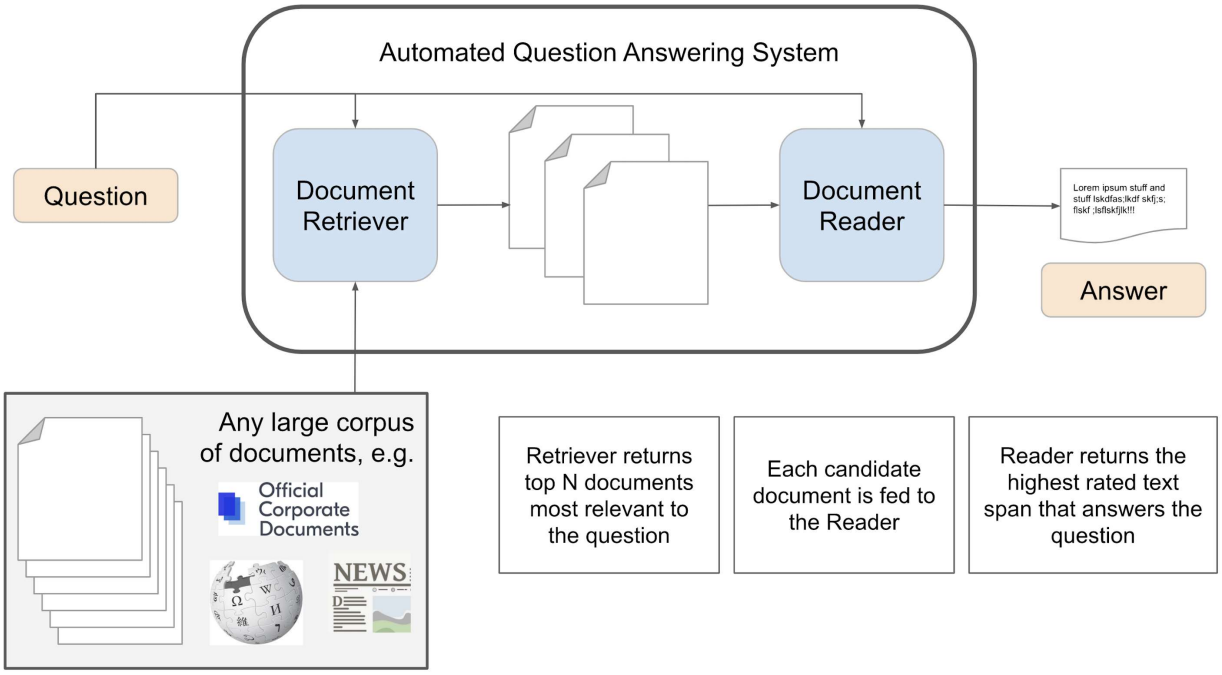
View On GitHub launch binder Open in Colab

Image by Markus Spiske at Unsplash.com

So you've decided to build a QA system

You want to start with something simple and general, so you plan to make it open domain, using Wikipedia as a corpus for answering questions. You want to use the best NLP that your compute resources allow (you're lucky enough to have access to a GPU), so you're going to focus on the big, flashy Transformer models that are all the rage these days.

Sounds like you're building an IR-based QA system. In our previous post (Intro to Automated Question Answering), we covered the general design of these systems, which typically require two main components: the document *retriever* (a search engine) that selects the n most relevant documents from a large collection, and a document *reader* that processes these candidate documents in search of an explicit answer span.



IR-based automated question answering workflow

Now we're going to build it!

This post is chock full of code that walks through our approach. We'll also highlight and clarify some powerful resources (including off-the-shelf models and libraries) that you can use to quickly get going on a QA system of your own. We'll cover all the necessary steps including:

- installing libraries and setting up an environment,
- training a Transformer style model on the SQuAD dataset,
- understanding Hugging Face's run_squad.py training script and output,
- and passing a full Wikipedia article as context for a question.

By the end of this post we'll have a working IR-based QA system, with BERT as the document reader and Wikipedia's search engine as the document retriever - a fun toy model that hints at potential real-world use cases.

This article was originally developed in a Jupyter Notebook and, thanks to fastpages, converted to a blog post. For an interactive environment, click the "Open in Colab" button above (though we note that, due to Colab's system constraints, some of the cells in this notebook might not be fully executable.

We'll highlight when this is the case, but don't worry -- you'll still be able to play around with all the fun stuff.)

Let's get started!

Setting up your virtual environment

A virtual environment is always best practice and we're using `venv` on our workhorse machine. For this project, we'll be using PyTorch, which handles the heavy lifting of deep differentiable learning. If you have a GPU you'll want a PyTorch build that includes CUDA support, though most cells in this notebook will work fine without one. Check out PyTorch's quick install guide to determine the best build for your GPU and OS. We'll also be using the Transformers library, which provides easy-to-use implementations of all the popular Transformer architectures, like BERT. Finally, we'll need the wikipedia library for easy access and parsing of Wikipedia pages.

You can recreate our env (with CUDA 9.2 support -- but use the appropriate version for your machine) with the following commands in your command line:

```
$ python3 -m venv myenv
$ source myenv/bin/activate
$ pip install torch==1.5.0+cu92 torchvision==0.6.0+cu92 -f https://download.pytorch.org/whl/torch_stable.h
$ pip install transformers==2.5.1
$ pip install wikipedia==1.4.0
```

Note: Our GPU machine sports an older version of CUDA (9.2 -- we're getting around to updating that), so we need to use an older version of PyTorch for the necessary CUDA support. The training script we'll be using requires some specific packages. More recent versions of PyTorch include these packages; however, older versions do not. If you have to work with an older version of PyTorch, you might need to install `TensorboardX` (see the hidden code cell below).

Show Code

Conversely, if you're working in Colab, you can run the cell below.

```
!pip install torch  torchvision -f https://download.pytorch.org/whl/torch_stable.html
!pip install transformers==2.5.1
!pip install wikipedia==1.4.0
```

Hugging Face Transformers

The Hugging Face Transformers package provides state-of-the-art general-purpose architectures for natural language understanding and natural language generation. They host dozens of pre-trained models operating in over 100 languages that you can use right out of the box. All of these models come with deep interoperability between PyTorch and Tensorflow 2.0, which means you can move a model from TF2.0 to PyTorch and back again with just a line or two of code!

If you're new to Hugging Face, we strongly recommend working through the HF Quickstart guide as well as their excellent Transformer Notebooks (we did!), as we won't cover that material in this notebook. We'll be using `AutoClasses`, which serve as a wrapper around pretty much any of the base Transformer classes.

Fine-tuning a Transformer model for Question Answering

To train a Transformer for QA with Hugging Face, we'll need

1. to pick a specific model architecture,
2. a QA dataset, and
3. the training script.

With these three things in hand we'll then walk through the fine-tuning process.

1. Pick a Model

Not every Transformer architecture lends itself naturally to the task of question answering. For example, GPT does not do QA; similarly BERT does not do machine translation. HF identifies the following model types for the QA task:

- BERT
- distilBERT
- ALBERT
- RoBERTa
- XLNet
- XLM
- FlauBERT

We'll stick with the now-classic BERT model in this notebook, but feel free to try out some others (we will - and we'll let you know when we do). Next up: a training set.

2. QA dataset: SQuAD

One of the most canonical datasets for QA is the Stanford Question Answering Dataset, or SQuAD, which comes in two flavors: SQuAD 1.1 and SQuAD 2.0. These reading comprehension datasets consist of questions posed on a set of Wikipedia articles, where the answer to every question is a segment (or span) of the corresponding passage. In SQuAD 1.1, all questions have an answer in the corresponding passage. SQuAD 2.0 steps up the difficulty by including questions that cannot be answered by the provided passage.

The following code will download the specified version of SQuAD.

```
# set path with magic
%env DATA_DIR=./data/squad

# download the data
def download_squad(version=1):
    if version == 1:
        !wget -P $DATA_DIR https://rajpurkar.github.io/SQuAD-explorer/dataset/train-v1.1.js
        !wget -P $DATA_DIR https://rajpurkar.github.io/SQuAD-explorer/dataset/dev-v1.1.jso
    else:
```

```
        !wget -P $DATA_DIR https://rajpurkar.github.io/SQuAD-explorer/dataset/train-v2.0.js
        !wget -P $DATA_DIR https://rajpurkar.github.io/SQuAD-explorer/dataset/dev-v2.0.jsor

download_squad(version=2)
```

```
env: DATA_DIR=./data/squad
--2020-05-11 21:36:52--  https://rajpurkar.github.io/SQuAD-explorer/dataset/train-v2.0.json
Resolving rajpurkar.github.io (rajpurkar.github.io)... 185.199.109.153, 185.199.108.153, 185.199.111.15
Connecting to rajpurkar.github.io (rajpurkar.github.io)|185.199.109.153|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 42123633 (40M) [application/json]
Saving to: ‘./data/squad/train-v2.0.json’

train-v2.0.json     100%[===================>]  40.17M  14.6MB/s    in 2.8s

2020-05-11 21:36:55 (14.6 MB/s) - ‘./data/squad/train-v2.0.json’ saved [42123633/42123633]

--2020-05-11 21:36:56--  https://rajpurkar.github.io/SQuAD-explorer/dataset/dev-v2.0.json
Resolving rajpurkar.github.io (rajpurkar.github.io)... 185.199.110.153, 185.199.111.153, 185.199.108.15
Connecting to rajpurkar.github.io (rajpurkar.github.io)|185.199.110.153|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 4370528 (4.2M) [application/json]
Saving to: ‘./data/squad/dev-v2.0.json’

dev-v2.0.json       100%[===================>]   4.17M  6.68MB/s    in 0.6s

2020-05-11 21:36:56 (6.68 MB/s) - ‘./data/squad/dev-v2.0.json’ saved [4370528/4370528]
```

3. Fine-tuning script

We've chosen a model and we've got some data. Time to train!

All the standard models that HF supports have been pre-trained, which means they've all been fed massive unsupervised training sets in order to learn basic language modeling. In order to perform well at specific tasks (like question answering), they must be trained further -- fine-tuned -- on specific datasets and tasks.

HF helpfully provides a script that fine-tunes a Transformer model on one of the SQuAD datasets, called `run_squad.py`. You can grab the script here or run the cell below.

```
# download the run_squad.py training script
!curl -L -O https://github.com/huggingface/transformers/blob/b90745c5901809faef3136ed09a689
```

This script takes care of all the hard work that goes into fine-tuning a model and, as such, it's pretty complicated. It hosts no fewer than 45 arguments, providing an impressive amount of flexibility and utility for those who do a lot of training. We'll leave the details of this script for another day, and focus instead on the basic command to fine-tune BERT on SQuAD 1.1 or 2.0.

Below are the most important arguments for the `run_squad.py` fine-tuning script.

```
# fine-tuning your own model for QA using HF's `run_squad.py`
# turn flags on and off according to the model you're training

cmd = [
    'python',
#    '-m torch.distributed.launch --nproc_per_node 2', # use this to perform distributed tr
    'run_squad.py',

    '--model_type', 'bert',                                # model type (one of the list under

    '--model_name_or_path', 'bert-base-uncased',           # specific model name of the given m
                                                           # on first execution this initiates
                                                           # can also be a local path to a dire

    '--output_dir', './models/bert/bbu_squad2',            # directory for model checkpoints ar

#    '--overwrite_output_dir',                             # use when adding output to a direct
                                                           # for instance, when training crashe

    '--do_train',                                          # execute the training method

    '--train_file', '$DATA_DIR/train-v2.0.json',           # provide the training data

    '--version_2_with_negative',                           # ** MUST use this flag if training

    '--do_lower_case',                                     # ** set this flag if using an uncas

    '--do_eval',                                           # execute the evaluation method on t
                                                           # if coupled with --do_train, evaluc
```

```
    '--predict_file', '$DATA_DIR/dev-v2.0.json',       # provide evaluation data (dev set)

    '--eval_all_checkpoints',                          # evaluate the model on the dev set

    '--per_gpu_eval_batch_size', '12',                 # evaluation batch size for each gpu

    '--per_gpu_train_batch_size', '12',                # training batch size for each gpu

    '--save_steps', '5000',                            # how often checkpoints (complete mo

    '--threads', '8',                                  # num of CPU threads to use for conv

    # --- Model and Feature Hyperparameters ---
    '--num_train_epochs', '3',                         # number of training epochs - usuall

    '--learning_rate', '3e-5',                         # learning rate for the default opti

    '--max_seq_length', '384',                         # maximum length allowed for the ful

    '--doc_stride', '128'                              # used for long documents that must
                                                       # this "sliding window" controls the
]
```

Here's what to expect when executing `run_squad.py` for the first time:

1. Pre-trained model weights for the specified model type (i.e., `bert-base-uncased`) are downloaded.
2. SQuAD training examples are converted into features (takes 15-30 minutes depending on dataset size and number of threads).
3. Training features are saved to a cache file (so that you don't have to do this again *for this model type*).
4. If `--do_train`, training commences for as many epochs as you specify, saving the model weights every `--save_steps` steps until training finishes. These checkpoints are saved in `[--output_dir]/checkpoint-[step number]` subdirectories.
5. The final model weights and peripheral files are saved to `--output_dir`.
6. If `--do_eval`, SQuAD dev examples are converted into features.
7. Dev features are also saved to a cache file.
8. Evaluation commences and outputs a dizzying assortment of performance scores.

Time to train!

But first, a note on compute requirements. We don't recommend fine-tuning a Transformer model unless you're rocking at least one GPU and a considerable amount of RAM. For context, our GPU is several years old (GeForce GTX TITAN X), and while it's not nearly as fast as the Tesla V100 (the current Cadillac of GPUs), it gets the job done. Fine-tuning `bert-base-uncased` takes about 1.75 hours *per epoch*. Additionally, our workhorse machine has 32GB CPU and 12GB GPU memory, which is sufficient for data processing and training most models on either of the SQuAD datasets.

The following cells demonstrate two ways to fine-tune: on the command line and in a Colab notebook.

Training on the command line

We saved the following as a shell script (`run_squad.sh`) and ran on the command line (`$ source run_squad.sh`) of our workhorse GPU machine. Shell scripts help prevent numerous mistakes and mis-keys when typing args to a command line, especially for complex scripts like this. They also allow you to keep track of which arguments were used last (though, as we'll see below, the `run_squad.py` script has a solution for that). We actually kept two shell scripts -- one explicitly for training and another for evaluation.

```
#!/bin/sh
export DATA_DIR=./data/squad
export MODEL_DIR=./models
python run_squad.py  \
    --model_type bert   \
    --model_name_or_path bert-base-uncased  \
    --output_dir models/bert/ \
    --data_dir data/squad   \
    --overwrite_output_dir \
    --overwrite_cache \
    --do_train  \
    --train_file train-v2.0.json   \
    --version_2_with_negative \
    --do_lower_case  \
    --do_eval   \
    --predict_file dev-v2.0.json   \
    --per_gpu_train_batch_size 2   \
```

```
    --learning_rate 3e-5   \
    --num_train_epochs 2.0   \
    --max_seq_length 384   \
    --doc_stride 128   \
    --threads 10   \
    --save_steps 5000
```

Training in Colab

Alternatively, you can execute training in the cell as shown below. We note that standard Colab environments only provide 12GB of RAM. Converting the SQuAD dataset to features is memory intensive and may cause the basic Colab environment to fail silently. If you have a Colab instance with additional memory capacity (16GB+), this cell should execute fully.

```
!python run_squad.py  \
    --model_type bert   \
    --model_name_or_path bert-base-uncased  \
    --output_dir models/bert/ \
    --data_dir data/squad   \
    --overwrite_output_dir \
    --overwrite_cache \
    --do_train  \
    --train_file train-v2.0.json   \
    --version_2_with_negative \
    --do_lower_case  \
    --do_eval   \
    --predict_file dev-v2.0.json   \
    --per_gpu_train_batch_size 2   \
    --learning_rate 3e-5   \
    --num_train_epochs 2.0   \
    --max_seq_length 384   \
    --doc_stride 128   \
    --threads 10   \
    --save_steps 5000
```

Training Output

Successful completion of the `run_squad.py` yields a slew of output, which can be found in the `--output_dir` directory specified above. There you'll find...

Files for the model's tokenizer:

- `tokenizer_config.json`
- `vocab.txt`
- `special_tokens_map.json`

Files for the model itself:

- `pytorch_model.bin`: these are the actual model weights (this file can be several GB for some models)
- `config.json`: details of the model architecture

Binary representation of the command line arguments used to train this model (so you'll never forget which arguments you used!)

- `training_args.bin`

And if you included `--do_eval`, you'll also see these files:

- `predictions_.json`: the official best answer for each example
- `nbest_predictions_.json`: the top n best answers for each example

Providing the path to this directory to `AutoModel` or `AutoModelForQuestionAnswering` will load your fine-tuned model for use.

```
from transformers import AutoTokenizer, AutoModelForQuestionAnswering

# Load the fine-tuned model
tokenizer = AutoTokenizer.from_pretrained("./models/bert/bbu_squad2")
model = AutoModelForQuestionAnswering.from_pretrained("./models/bert/bbu_squad2")
```

Using a pre-fine-tuned model from the Hugging Face repository

If you don't have access to GPUs or don't have the time to fiddle and train models, you're in luck! Hugging Face is more than a collection of slick Transformer classes -- it also hosts a repository for pre-trained and fine-tuned models contributed from the wide community of NLP practitioners. Searching for "squad" brings up at least 55 models.

HUGGING FACE

Back to home

All Models and checkpoints

Also check out our list of **Community contributors** 🏆 and **Organizations** 🌍 .

squad | Tags: All | Sort: Default

ahotrod/albert_xxlargev1_squad2_512

ahotrod/roberta_large_squad2

ahotrod/xlnet_large_squad2_512

deepset/roberta-base-squad2-covid

deepset/roberta-base-squad2

elgeish/cs224n-squad2.0-albert-base-v2

elgeish/cs224n-squad2.0-albert-large-v2

elgeish/cs224n-squad2.0-albert-xxlarge-v1

Each of these links provides explicit code for using the model, and, in some cases, information on how it was trained and what results were achieved. Let's load one of these pre-fine-tuned models.

```
import torch
from transformers import AutoTokenizer, AutoModelForQuestionAnswering
```

```
# executing these commands for the first time initiates a download of the
# model weights to ~/.cache/torch/transformers/
tokenizer = AutoTokenizer.from_pretrained("deepset/bert-base-cased-squad2")
model = AutoModelForQuestionAnswering.from_pretrained("deepset/bert-base-cased-squad2")
```

Let's try our model!

Whether you fine-tuned your own or used a pre-fine-tuned model, it's time to play with it! There are three steps to QA:

1. tokenize the input
2. obtain model scores
3. get the answer span

These steps are discussed in detail in the HF Transformer Notebooks.

```
question = "Who ruled Macedonia"

context = """Macedonia was an ancient kingdom on the periphery of Archaic and Classical Gre
and later the dominant state of Hellenistic Greece. The kingdom was founded and initially r
by the Argead dynasty, followed by the Antipatrid and Antigonid dynasties. Home to the anci
Macedonians, it originated on the northeastern part of the Greek peninsula. Before the 4th
century BC, it was a small kingdom outside of the area dominated by the city-states of Athe
Sparta and Thebes, and briefly subordinate to Achaemenid Persia."""


# 1. TOKENIZE THE INPUT
# note: if you don't include return_tensors='pt' you'll get a list of lists which is easier
# exploration but you cannot feed that into a model.
inputs = tokenizer.encode_plus(question, context, return_tensors="pt")

# 2. OBTAIN MODEL SCORES
# the AutoModelForQuestionAnswering class includes a span predictor on top of the model.
# the model returns answer start and end scores for each word in the text
answer_start_scores, answer_end_scores = model(**inputs)
answer_start = torch.argmax(answer_start_scores)  # get the most likely beginning of answer
answer_end = torch.argmax(answer_end_scores) + 1  # get the most likely end of answer with

# 3. GET THE ANSWER SPAN
# once we have the most likely start and end tokens, we grab all the tokens between them
# and convert tokens back to words!
tokenizer.convert_tokens_to_string(tokenizer.convert_ids_to_tokens(inputs["input_ids"][0][a
```

```
'the Argead dynasty'
```

QA on Wikipedia pages

We tried our model on a question paired with a short passage, but what if we want to retrieve an answer from a longer document? A typical Wikipedia page is much longer than the example above, and we need to do a bit of massaging before we can use our model on longer contexts.

Let's start by pulling up a Wikipedia page.

```
import wikipedia as wiki
import pprint as pp

question = 'What is the wingspan of an albatross?'

results = wiki.search(question)
print("Wikipedia search results for our question:\n")
pp.pprint(results)

page = wiki.page(results[0])
text = page.content
print(f"\nThe {results[0]} Wikipedia article contains {len(text)} characters.")
```

```
Wikipedia search results for our question:

['Albatross',
 'List of largest birds',
 'Black-browed albatross',
 'Argentavis',
 'Pterosaur',
 'Mollymawk',
 'List of birds by flight speed',
 'Largest body part',
 'Pelican',
 'Aspect ratio (aeronautics)']

The Albatross Wikipedia article contains 38200 characters.
```

```
inputs = tokenizer.encode_plus(question, text, return_tensors='pt')
print(f"This translates into {len(inputs['input_ids'][0])} tokens.")
```

```
Token indices sequence length is longer than the specified maximum sequence length for this model (10 >
```

```
This translates into 8824 tokens.
```

The tokenizer takes the input as text and returns tokens. In general, tokenizers convert words or pieces of words into a model-ingestible format. The specific tokens and format are dependent on the type of model. For example, BERT tokenizes words differently from RoBERTa, so be sure to always use the associated tokenizer appropriate for your model.

In this case, the tokenizer converts our input text into 8824 tokens, but this far exceeds the maximum number of tokens that can be fed to the model at one time. Most BERT-esque models can only accept 512 tokens at once, thus the (somewhat confusing) warning above (how is 10 > 512?). This means we'll have to split our input into chunks and each chunk must not exceed 512 tokens in total.

When working with Question Answering, it's crucial that each chunk follows this format:

[CLS] question tokens [SEP] context tokens [SEP]

This means that, for each segment of a Wikipedia article, we must prepend the original question, followed by the next "chunk" of article tokens.

```
# time to chunk!
from collections import OrderedDict

# identify question tokens (token_type_ids = 0)
qmask = inputs['token_type_ids'].lt(1)
qt = torch.masked_select(inputs['input_ids'], qmask)
print(f"The question consists of {qt.size()[0]} tokens.")

chunk_size = model.config.max_position_embeddings - qt.size()[0] - 1 # the "-1" accounts fo
# having to add a [SEP] token to the end of each chunk
print(f"Each chunk will contain {chunk_size - 2} tokens of the Wikipedia article.")

# create a dict of dicts; each sub-dict mimics the structure of pre-chunked model input
chunked_input = OrderedDict()
```

```
for k,v in inputs.items():
    q = torch.masked_select(v, qmask)
    c = torch.masked_select(v, ~qmask)
    chunks = torch.split(c, chunk_size)

    for i, chunk in enumerate(chunks):
        if i not in chunked_input:
            chunked_input[i] = {}

        thing = torch.cat((q, chunk))
        if i != len(chunks)-1:
            if k == 'input_ids':
                thing = torch.cat((thing, torch.tensor([102])))
            else:
                thing = torch.cat((thing, torch.tensor([1])))

        chunked_input[i][k] = torch.unsqueeze(thing, dim=0)
```

```
The question consists of 12 tokens.
Each chunk will contain 497 tokens of the Wikipedia article.
```

```
for i in range(len(chunked_input.keys())):
    print(f"Number of tokens in chunk {i}: {len(chunked_input[i]['input_ids'].tolist()[0])}")
```

```
Number of tokens in chunk 0: 512
Number of tokens in chunk 1: 512
Number of tokens in chunk 2: 512
Number of tokens in chunk 3: 512
Number of tokens in chunk 4: 512
Number of tokens in chunk 5: 512
Number of tokens in chunk 6: 512
Number of tokens in chunk 7: 512
Number of tokens in chunk 8: 512
Number of tokens in chunk 9: 512
Number of tokens in chunk 10: 512
Number of tokens in chunk 11: 512
Number of tokens in chunk 12: 512
Number of tokens in chunk 13: 512
Number of tokens in chunk 14: 512
Number of tokens in chunk 15: 512
Number of tokens in chunk 16: 512
Number of tokens in chunk 17: 341
```

Each of these chunks (except for the last one) has the following structure:

[CLS], 12 question tokens, [SEP], 497 tokens of the Wikipedia article, [SEP] token = 512 tokens

Each of these chunks can now be fed to the model without causing indexing errors. We'll get an "answer" for each chunk; however, not all answers are useful, since not every segment of a Wikipedia article is informative for our question. The model will return the [CLS] token when it determines that the context does not contain an answer to the question.

```
def convert_ids_to_string(tokenizer, input_ids):
    return tokenizer.convert_tokens_to_string(tokenizer.convert_ids_to_tokens(input_ids))

answer = ''

# now we iterate over our chunks, looking for the best answer from each chunk
for _, chunk in chunked_input.items():
    answer_start_scores, answer_end_scores = model(**chunk)

    answer_start = torch.argmax(answer_start_scores)
    answer_end = torch.argmax(answer_end_scores) + 1

    ans = convert_ids_to_string(tokenizer, chunk['input_ids'][0][answer_start:answer_end])

    # if the ans == [CLS] then the model did not find a real answer in this chunk
    if ans != '[CLS]':
        answer += ans + " / "

print(answer)
```

```
3 . 7 m /
```

Putting it all together

Let's recap. We've essentially built a simple IR-based QA system! We're using `wikipedia`'s search engine to return a list of candidate documents that we then feed into our document reader (in this case, BERT fine-tuned on SQuAD 2.0). Let's make our code easier to read and more self-contained by packaging the document reader into a class.

```
from transformers import AutoTokenizer, AutoModelForQuestionAnswering


class DocumentReader:
    def __init__(self, pretrained_model_name_or_path='bert-large-uncased'):
        self.READER_PATH = pretrained_model_name_or_path
        self.tokenizer = AutoTokenizer.from_pretrained(self.READER_PATH)
        self.model = AutoModelForQuestionAnswering.from_pretrained(self.READER_PATH)
        self.max_len = self.model.config.max_position_embeddings
        self.chunked = False

    def tokenize(self, question, text):
        self.inputs = self.tokenizer.encode_plus(question, text, add_special_tokens=True, r
        self.input_ids = self.inputs["input_ids"].tolist()[0]

        if len(self.input_ids) > self.max_len:
            self.inputs = self.chunkify()
            self.chunked = True

    def chunkify(self):
        """
        Break up a long article into chunks that fit within the max token
        requirement for that Transformer model.

        Calls to BERT / RoBERTa / ALBERT require the following format:
        [CLS] question tokens [SEP] context tokens [SEP].
        """

        # create question mask based on token_type_ids
        # value is 0 for question tokens, 1 for context tokens
        qmask = self.inputs['token_type_ids'].lt(1)
        qt = torch.masked_select(self.inputs['input_ids'], qmask)
        chunk_size = self.max_len - qt.size()[0] - 1 # the "-1" accounts for
        # having to add an ending [SEP] token to the end

        # create a dict of dicts; each sub-dict mimics the structure of pre-chunked model i
        chunked_input = OrderedDict()
        for k,v in self.inputs.items():
            q = torch.masked_select(v, qmask)
            c = torch.masked_select(v, ~qmask)
            chunks = torch.split(c, chunk_size)

            for i, chunk in enumerate(chunks):
                if i not in chunked_input:
                    chunked_input[i] = {}
```

```
                thing = torch.cat((q, chunk))
                if i != len(chunks)-1:
                    if k == 'input_ids':
                        thing = torch.cat((thing, torch.tensor([102])))
                    else:
                        thing = torch.cat((thing, torch.tensor([1])))

                chunked_input[i][k] = torch.unsqueeze(thing, dim=0)
        return chunked_input

    def get_answer(self):
        if self.chunked:
            answer = ''
            for k, chunk in self.inputs.items():
                answer_start_scores, answer_end_scores = self.model(**chunk)

                answer_start = torch.argmax(answer_start_scores)
                answer_end = torch.argmax(answer_end_scores) + 1

                ans = self.convert_ids_to_string(chunk['input_ids'][0][answer_start:answer_
                if ans != '[CLS]':
                    answer += ans + " / "
            return answer
        else:
            answer_start_scores, answer_end_scores = self.model(**self.inputs)

            answer_start = torch.argmax(answer_start_scores)  # get the most likely beginni
            answer_end = torch.argmax(answer_end_scores) + 1  # get the most likely end of

            return self.convert_ids_to_string(self.inputs['input_ids'][0][
                                              answer_start:answer_end])

    def convert_ids_to_string(self, input_ids):
        return self.tokenizer.convert_tokens_to_string(self.tokenizer.convert_ids_to_tokens
```

Below is our clean, fully working QA system! Feel free to add your own questions.

Show Code

```
questions = [
    'When was Barack Obama born?',
    'Why is the sky blue?',
```

```
    'How many sides does a pentagon have?'
]

reader = DocumentReader("deepset/bert-base-cased-squad2")

# if you trained your own model using the training cell earlier, you can access it with thi
#reader = DocumentReader("./models/bert/bbu_squad2")

for question in questions:
    print(f"Question: {question}")
    results = wiki.search(question)

    page = wiki.page(results[0])
    print(f"Top wiki result: {page}")

    text = page.content

    reader.tokenize(question, text)
    print(f"Answer: {reader.get_answer()}")
    print()
```

```
Question: When was Barack Obama born?
Top wiki result: <WikipediaPage 'Barack Obama Sr.'>
Answer: 18 June 1936 / February 2 , 1961 /

Question: Why is the sky blue?
Top wiki result: <WikipediaPage 'Diffuse sky radiation'>
Answer: Rayleigh scattering /

Question: How many sides does a pentagon have?
Top wiki result: <WikipediaPage 'The Pentagon'>
Answer: five /
```

It got 2 out of 3 questions right!

Notice that, at least for the current questions we've chosen, the QA system fails because of Wikipedia's default search engine, not because of BERT! It pulls up the wrong page for two of our questions: a page about Barack Obama Sr. instead of the former US President, and an article about the US's Department of Defense building "The Pentagon" instead of a page about geometry. In the latter case, we ended up with the correct answer by coincidence! This illustrates that any successful IR-based QA system requires a search engine (document retriever) as good as the document reader.

Wrapping Up

There we have it! A working QA system on Wikipedia articles. This is great, but it's admittedly not very sophisticated. Furthermore, we've left a lot of questions unanswered:

1. Why fine-tune on the SQuAD dataset and not something else? What other options are there?
2. How good is BERT at answering questions? And how do we define "good"?
3. Why BERT and not another Transformer model?
4. Currently, our QA system can return an answer for each chunk of a Wiki article, but not all of those answers are correct -- How can we improve our `get_answer` method?
5. Additionally, we're chunking a wiki article in such a way that we could be ending a chunk in the middle of a sentence -- Can we improve our `chunkify` method?

Over the course of this project, we'll tackle these questions and more. By the end of this series we hope to demonstrate a snazzier QA model that incorporates everything we learn along the way. Stay tuned!

Subscribe

CFF builds a state-of-the-art QA application with the latest NLP techniques