Machine Comprehension with pytorch-transformers

by Roberto Silveira

8 min read • *August 18, 2019*

CATEGORIES machine_learning, nlp, pytorch

Step-by-step guide to finetune and use question and answering models with pytorch-transformers

I have used question and answering systems for some time now, and I'm really impressed how these algorithms evolved recently. My first interaction with QA algorithms was with the BiDAF model (Bidirectional Attention Flow) [1] from the great AllenNLP team. It was back in 2017, and ELMo embeddings [2] were not even used in this BiDAF model (I believe they were using GLove vectors in this first model). Since then, a lot of stuff is happened in the NLP arena, such as the Transformer [3], BERT [4] and the many other members of the Sesame Street family (now there are a whole BERT-like-family such as Facebook RoBERTa [4], VilBERT and maybe(why not?) one day, DilBERT).

There are lots of great materias out there (see Probe Further section for more details), so it will be much easier to go on and watch these awesome video materials instead of detailing each model in a blog post.

I would really want to spend time in the practical usage of question and answering models, as they can be very helpful for real-life applications (besides some challenges that will be addressed in other posts - such as model size, response time, model quantization/pruning, etc).

In this regard, all the ML community should give a massive shout-out to Hugging Face team. They are really pushing the limits to make the latest and greatest algorithms available for the broader community, and it is really cool to see how their project is growing rapidly in github (at the time I'm writing this they already surpassed more than 10k

⭐on github for the pytorch-transformer repo, for example). I will focus on SQuAD 1.1 dataset, more details on how fine-tune/use these models with SQuAD 2.0 dataset will be described in further posts.

Inside pytorch-transformers

The `pytorch-transformers` lib has some special classes, and the nice thing is that they try to be consistent with this architecture independently of the model (BERT, XLNet, RoBERTa, etc). These 3 important classes are:

> **Config** → this is the class that defines all the configurations of the model in hand, such as number of hidden layers in Transformer, number of attention heads in the Transformer encoder, activation function, dropout rate, etc. Usually, there are 2 *default* configurations [`base` , `large`], but it is possible to tune the configurations to have different models. The file format of the configuration file is a `.json` file.

> **Tokenizer** → the tokenizer class deals with some linguistic details of each model class, as specific tokenization types are used (such as WordPiece for BERT or SentencePiece for XLNet). It also handles begin-of-sentence (bos), end-of-sentence (eod), unknown, separation, padding, mask and any other special tokens. The tokenizer file can be loaded as a `.txt` file.

> **Model** → finally, we need to specify the model class. In this specific case, we are going to use special classes for Question and Answering [`BertForQuestionAnswering` , `XLNetForQuestionAnswering`], but there are other classes for different downstream tasks that can be used. These downstream classes inherit [`BertModel` , `XLNetModel`] classes, which will then go into more specific details (embedding type, Transformer configuration, etc). The weights of a fine-tuned downstream task mode are stored in a `.bin` file.

Download Fine-tuned models

BERT Model for SQuAD 1.1 **XLNet Model for SQuAD 1.1**

Watch out! The BERT model I downloaded directly from Hugging Face repo, the XLNet model I fine-tuned myself for 3 epochs in a Nvidia 1080ti. Also, I noticed that the XLNet model maybe needs some more training - see Results section

Finetuning scripts

To run the fine-tuning scripts, the Hugging Face team makes available some dataset-specific files that can be found here. These fine-tuning scripts can be highly customizable, for example by passing a config file for a model specified in `.json` file e.g. `--config_name xlnet_m2.jsonn`. The examples below are showing BERT finetuning with `base` configuration, and `xlnet` configuration with specific parameters (`n_head` , `n_layer`). The models provided for download both use the `large` config.

Finetuning BERT

```
python run_squad.py \
 --model_type bert \
 --model_name_or_path bert-base-cased \
 --do_train \
 --do_eval \
 --evaluate_during_training \
 --do_lower_case \
 --train_file $SQUAD_DIR/train-v1.1.json \
 --predict_file $SQUAD_DIR/dev-v1.1.json \
 --save_steps 10000 \
 --learning_rate 3e-5 \
 --num_train_epochs 5.0 \
 --max_seq_length 384 \
 --doc_stride 128 \
 --output_dir /home/roberto/tmp/finetuned_xlnet \
  --overwrite_output_dir \
  --overwrite_cache
```

Finetuning XLNet

```
python -u run_squad.py \
  --model_type xlnet \
  --model_name_or_path xlnet-large-cased \
  --do_train \
  --do_eval \
  --config_name xlnet_m2.json \
  --evaluate_during_training \
  --do_lower_case \
  --train_file $SQUAD_DIR/train-v1.1.json \
  --predict_file $SQUAD_DIR/dev-v1.1.json \
  --save_steps 10000 \
  --learning_rate 3e-5 \
  --num_train_epochs 5.0 \
  --max_seq_length 384 \
  --doc_stride 128 \
  --per_gpu_train_batch_size 1 \
  --output_dir /home/roberto/tmp/finetuned_xlnet \
```

```
  --overwrite_output_dir \
  --overwrite_cache
```

Config `xlnet_m2.json`


```
{
  "attn_type": "bi",
  "bi_data": false,
  "clamp_len": -1,
  "d_head": 64,
  "d_inner": 4096,
  "d_model": 1024,
  "dropatt": 0.1,
  "dropout": 0.1,
  "end_n_top": 5,
  "ff_activation": "gelu",
  "finetuning_task": null,
  "init": "normal",
  "init_range": 0.1,
  "init_std": 0.02,
  "initializer_range": 0.02,
  "layer_norm_eps": 1e-12,
  "max_position_embeddings": 512,
  "mem_len": null,
  "n_head": 16,
  "n_layer": 18,
  "n_token": 32000,
  "num_labels": 2,
  "output_attentions": false,
  "output_hidden_states": false,
  "reuse_len": null,
  "same_length": false,
  "start_n_top": 5,
  "summary_activation": "tanh",
  "summary_last_dropout": 0.1,
  "summary_type": "last",
  "summary_use_proj": true,
  "torchscript": false,
  "untie_r": true
}
```


Using the trained models

Now to the fun part: using these models for question and answering!

First things first, let's import the model classes from `pytorch-transformers`

```
import os
import time
import torch
from pytorch_transformers import BertConfig, BertTokenizer, BertForQuestionAnswering
from pytorch_transformers import XLNetConfig, XLNetForQuestionAnswering, XLNetTokenizer
```

These are the 3 important classes:

```
MODEL_CLASSES = {
    'bert': (BertConfig, BertForQuestionAnswering, BertTokenizer),
    'xlnet': (XLNetConfig, XLNetForQuestionAnswering, XLNetTokenizer)
}
```

I've made this special class to handles all the feature preparation and output formating for both BERT and XLNet, but this could be done in different ways:

```
class QuestionAnswering(object):
    def __init__(self, config_file, weight_file, tokenizer_file, model_type ):
        self.device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        self.config_class, self.model_class, self.tokenizer_class = MODEL_CLASSES[model_typ
        self.config = self.config_class.from_json_file(config_file)
        self.model = self.model_class(self.config)
        self.model.load_state_dict(torch.load(weight_file, map_location=self.device))
        self.tokenizer = self.tokenizer_class(tokenizer_file)
        self.model_type = model_type

    def to_list(self, tensor):
        return tensor.detach().cpu().tolist()

    def get_reply(self, question, passage):
        self.model.eval()
        with torch.no_grad():
            input_ids, _ , tokens = self.prepare_features(question, passage)
            if self.model_type == 'bert':
                span_start,span_end= self.model(input_ids)
                answer = tokens[torch.argmax(span_start):torch.argmax(span_end)+1]
                answer = self.bert_convert_tokens_to_string(answer)
            elif self.model_type == 'xlnet':
                input_vector = {'input_ids': input_ids,
                                'start_positions': None,
                                'end_positions': None }
                outputs = self.model(**input_vector)
```

```
            answer = tokens[self.to_list(outputs[1])[0][torch.argmax(outputs[0])]:self.
            answer = self.xlnet_convert_tokens_to_string(answer)
    return answer

def bert_convert_tokens_to_string(self, tokens):
    out_string = ' '.join(tokens).replace(' ##', '').strip()
    if '@' in tokens:
        out_string = out_string.replace(' ', '')
    return out_string

def xlnet_convert_tokens_to_string(self, tokens):
    out_string = ''.join(tokens).replace('_', ' ').strip()
    return out_string

def prepare_features(self, question,  passage, max_seq_length = 300,
             zero_pad = False, include_CLS_token = True, include_SEP_token = True):
    ## Tokenzine Input
    tokens_a = self.tokenizer.tokenize(question)
    tokens_b = self.tokenizer.tokenize(passage)
    ## Truncate
    if len(tokens_a) > max_seq_length - 2:
        tokens_a = tokens_a[0:(max_seq_length - 2)]
    ## Initialize Tokens
    tokens = []
    if include_CLS_token:
        tokens.append(self.tokenizer.cls_token)
    ## Add Tokens and separators
    for token in tokens_a:
        tokens.append(token)
    if include_SEP_token:
        tokens.append(self.tokenizer.sep_token)
    for token in tokens_b:
        tokens.append(token)
    ## Convert Tokens to IDs
    input_ids = self.tokenizer.convert_tokens_to_ids(tokens)
    ## Input Mask
    input_mask = [1] * len(input_ids)
    ## Zero-pad sequence lenght
    if zero_pad:
        while len(input_ids) < max_seq_length:
            input_ids.append(0)
            input_mask.append(0)
    return torch.tensor(input_ids).unsqueeze(0), input_mask, tokens
```

Finally we just need to instantiate these models and start using them!

BERT:

```
bert = QuestionAnswering(
    config_file =   'bert-large-cased-whole-word-masking-finetuned-squad-config.json',
    weight_file=    'bert-large-cased-whole-word-masking-finetuned-squad-pytorch_model.bin'
    tokenizer_file= 'bert-large-cased-whole-word-masking-finetuned-squad-vocab.txt',
    model_type =    'bert'
)
```

XLNet:

```
xlnet = QuestionAnswering(
    config_file =   'xlnet-cased-finetuned-squad.json',
    weight_file=    'xlnet-cased-finetuned-squad.bin',
    tokenizer_file= 'xlnet-large-cased-spiece.txt',
    model_type =    'xlnet'
)
```

Results

I've included some sample `facts` and `questions` to give these algorithms a go:

```
facts = " My wife is great. \
My complete name is Roberto Pereira Silveira. \
I am 40 years old. \
My dog is cool. \
My dog breed is jack russel. \
My dog was born in 2014.\
My dog name is Mallu. \
My dog is 5 years old. \
I am an engineer. \
I was born in 1979. \
My e-mail is rsilveira79@gmail.com."

questions = [
    "What is my complete name?",
    "What is dog name?",
    "What is my dog age?",
    "What is my age?",
    "What is my dog breed?",
    "When I was born?",
    "What is my e-mail?"
]
```

And here are the results! As you could see I should have trained XLNet a bit more, but it is already returning good results:

```
QUESTION: What is my complete name?
  BERT: roberto pereira silveira
  XLNET: Roberto Pereira Silveira
--------------------------------------------------
QUESTION: What is dog name?
  BERT: mallu
  XLNET: Roberto Pereira Silveira. I am 40 years old. My dog is cool. My dog breed is jack
--------------------------------------------------
QUESTION: What is my dog age?
  BERT: 5 years old
  XLNET: 40 years old
--------------------------------------------------
QUESTION: What is my age?
  BERT: 40
  XLNET: 40 years old
--------------------------------------------------
QUESTION: What is my dog breed?
  BERT: jack russel
  XLNET: jack russel
--------------------------------------------------
QUESTION: When I was born?
  BERT: 1979
  XLNET: 1979
--------------------------------------------------
QUESTION: What is my e-mail?
  BERT: rsilveira79@gmail.com
  XLNET: rsilveira79@gmail.com
--------------------------------------------------
```

Hope you enjoyed and till the next post!

References

1. **Bidirectional Attention Flow for Machine Comprehension** PDF
   Minjoon Seo and Aniruddha Kembhavi and Ali Farhadi and Hannaneh Hajishirzi, 2016
2. **Deep contextualized word representations** PDF
   Peters, Matthew and Neumann, Mark and Iyyer, Mohit and Gardner, Matt and Clark, Christopher and Lee, Kenton and Zettlemoyer, Luke, 2018
3. **Attention Is All You Need** PDF
   Ashish Vaswani and Noam Shazeer and Niki Parmar and Jakob Uszkoreit and Llion Jones and Aidan N. Gomez and Lukasz Kaiser and Illia Polosukhin, 2017

4. **BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding** PDF
   Jacob Devlin and Ming-Wei Chang and Kenton Lee and Kristina Toutanova, 2018
5. **RoBERTa: A Robustly Optimized BERT Pretraining Approach** PDF
   Yinhan Liu and Myle Ott and Naman Goyal and Jingfei Du and Mandar Joshi and Danqi Chen and Omer Levy and Mike Lewis and Luke Zettlemoyer and Veselin Stoyanov, 2019

To Probe Further

1. **The Illustrated Transformer** Link
   Jay Alammar
2. **Stanford CS224n NLP Class w/Ashish Vaswani & Anna Huang** Link
   Professor Christopher Manning
3. **ELMo - Paper Explained** Link
   ML Papers Explained - A.I. Socratic Circles - AISC
4. **Transformer - Paper Explained** Link
   ML Papers Explained - A.I. Socratic Circles - AISC
5. **BERT - Paper Explained** Link
   ML Papers Explained - A.I. Socratic Circles - AISC
6. **XLNet - Paper Explained** Link
   ML Papers Explained - A.I. Socratic Circles - AISC

Next
Text classification with RoBERTa →