Question Answering on Squad v1.1

A Bert based deep learning question answering model answering questions from context

Author and developer: Nikolaos Mallios

Intro on Data and approach

The squad question answering dataset is a sizeable dataset, where for each question there is q context where the given QA model answers from. The dataset is evaluated on whether the predicted answer is among the correct answers. Answers agree, but there are small differences. While questions strings tend to be short the context are pretty lengthy. We work on Squad v1.1.

Approach

In our approach we utilize a general use pre-trained Bert model. This model is loaded from Transformer's library. It is pretrained on a large corpus of English data in a self-supervised fashion, using Masked language modeling and Next sentence prediction. In other words, it is the classic Bert model which can be used for fine tuning.

We will fine-tune it on our downstream task, which is Question answering. We create a Tensorflow model training using SQUAD data.

The Bert model takes tokens sequences of maximum length of input tokens less than 512. This is enough in our dataset, as we are talking about tokens, Bert works with word and sub-words tokenization. But due, to the demand in resources, especially in memory for that, we only worked with Squad data samples where the tokenized context+question have length up to 200 max. This of course can easily change, from the project config file. The 200 max length was needed because the development was conducted on a Pc with 16gb or RAM. To reach up to 512 length and actually proceed all the Squad data, we estimate that it is needed RAM (or GPU RAM) or around 25 to 30GB.

Tools and libraries

The project use Python(3.8) and numpy for various data manipulations and operations. The model is developed in Tensorflow using mostly the tf.keras modules of Tensorflow. The project uses a pretrained model and a pretrained Tokenizer from Transformers. Finally, the API for making calls to get answers on the given context uses FastAPI.

See also the requirement.txt

Installation and deployment.

Install

Git checkout the project, create a virtual environment (suggested) using python 3.8 and pip install the requirements.

Prepare Data

Download the Squad v1.1 train and dev data(used as validation) from the official links bellow and place the inside Folder inputs:
https://rajpurkar.github.io/SQuAD-explorer/dataset/train-v1.1.json
https://rajpurkar.github.io/SQuAD-explorer/dataset/dev-v1.1.json

Prepare Configuration

Inside config.py you can set max length for the maximum length of model input and the batch. This also means the max length of tokenized question+context. Also, can change folders for downloading transformers.

Finally, can change the default Bert model. This is not advised without good knowledge of compatibility and API methods of transformers and Tensorflow!

TIP: based on the available RAM increase or reduce the batch as well as the max length. Very small batches reduce accuracy and scores but make the training lighter in terms of memory.

Train model

Run train_run.py with arguments

- epochs: the number of epochs to train (default 3)
- use_tpu: if to build model for tpu execution, if available.
- config: which of the available config dictionaries to use. Right now, only 1 is available, thus keep the default.

After finishing training, the model will be stored locally in the folder named model_{max_len}, for example as model_200. The trained model size is large!

API call predictions

Run app.py locally to start the API. Hitting the http://localhost:5000/docs will open the API documentation for more details or experiment.

To make a prediction call the /predict_questions providing a body of one or more context and question of max combined length equal to the configuration max. It returns the answer(s) the model predicted.

NOTE: the string inside the Post should be properly slash escaped as it is a Body call.

Implementation

Intro

Now let us explain in details the implementation and showcase some of the results of the training and predictions.

Starting the train_run.py will take the config files and the epochs proceed with downloading the tokenizer and pre-processing the Squad v1.1 train and dev data.

Tokenizer

We use the Pre-Trained Bert tokenizer from Transformers in order to transform the input string into tokens. The tokenizer tokenizes specifically for the Bert model, but not specifically for question answering tasks. As a result, the data requires some additional processing.

The tokenizers for Bert Transformers use WordPiece tokenization and not just simple word tokenization. WordPiece splits strings either as full words or as sub-strings(word-pieces). For example, surf-> [‘surf’] while surfing-> ['surf', '##ing']

Also, Bert tokenizer adds special tokens ‘[CLS]’ and ‘[SEP]’. These are required by Bert, as the input must start with CLS and the two strings, in our case context and question, must be separated by SEP.

Our pre-trained tokenizer outputs tokens of dimension 768.

Data

We use both train and dev files, in order to have a large enough dataset after we filter the max length to 200. We use around 70% from the total data.

We train on 65107 (filtered from total 87599 from train json).

We validate on 7654 (filtered from total 10570 from dev json).

Data Preprocessing

From each SQUAD Json, we process the questions and we create a SquadQuestionAnswer class object. This class has question, the answer, all answers (for evaluation) the context and answer start index as given by the dataset. Also, we easily calculate the answer end index, as it is not provided by Squad.

For our Bert based QA model, we need to also calculate the

- Input id: the tokenized text as token id from the tokenizer.
- Type id: is a mask with 0 for context (first sentence) and 1 for question. This allows the Bert to separate and understand the input.
- Attention mask: a mask used by Bert and generally Transformers to know where to apply attention and which parts of input are padded tokens (usually 0 pad).

After tokenization, the Wordpiece tokenizer makes the string based Squad indexes to be different in tokenized strings. Thus, we also calculate the new start and end indexes of the answer inside the tokenized context.

Finally, we pad with zeros to the max length. The inputs must have fixed length.

Lastly, when the input ids combined (context + question) are greater than the maximum length we skip them, as we said before.

Data as X and Y Model Inputs and Outputs

After pre-processing we transform the data into X and Y before passing them to the model. The X are the 2d numpy arrays of input ids, token_type_ids and attention_mask. We describe them above. For the Y target output is the start_token_index and end_token_index which we calculate after tokenization. Thus, our model tries to find correctly the start and end in the given context for the given question.

Model

The model is mostly composed from the Bert transformer encoder, which is trainable and two Question Answering heads. Which is two output neurons with softmax activation. Each one gives the probability of token input start and end position.

The loss function is sparse categorical cross entropy, where our categories are actually the tokenized sequence positions of the answer.

We use Adam optimizer with learning rate 0.00001. We use small LR, because this helps with fine tuning large models, especially when we have small batches of data.

We use a custom TensorFlow Callback, in order to calculate the F1 and Exact match scores at the end of each epoch. It is needed because the Squad Question answering has its own way of evaluating the data.

Bellow is the summary of the model, with max length 200 which we used for training. The trainable params are over 100.000.000 !!

```
Model: "model"
__________________________________________________________________________________________________
 Layer (type)                Output Shape              Param #   Connected to
==================================================================================================
 input_1 (InputLayer)        [(None, 200)]             0         []

 input_3 (InputLayer)        [(None, 200)]             0         []

 input_2 (InputLayer)        [(None, 200)]             0         []

 bert (TFBertMainLayer)      TFBaseModelOutputWi  109482240   ['input_1[0][0]',
                             thPoolingAndCrossAt               'input_3[0][0]',
                             tentions(last_hidde               'input_2[0][0]']
```

```
                    n_state=(None, 200,
                     768),
                     pooler_output=(Non
                    e, 768),
                     past_key_values=No
                    ne, hidden_states=N
                    one, attentions=Non
                    e, cross_attentions
                    =None)

 start_logit (Dense)          (None, 200, 1)      768       ['bert[0][0]']

 end_logit (Dense)            (None, 200, 1)      768       ['bert[0][0]']

 flatten (Flatten)            (None, 200)         0         ['start_logit[0][0]']

 flatten_1 (Flatten)          (None, 200)         0         ['end_logit[0][0]']

 activation (Activation)      (None, 200)         0         ['flatten[0][0]']

 activation_1 (Activation)    (None, 200)         0         ['flatten_1[0][0]']

==================================================================================================
Total params: 109,483,776
Trainable params: 109,483,776
Non-trainable params: 0
```

Model train results

After training for 3 epochs the model already gives very good results!

```
113/113 [==============================] - ETA: 0s - loss: 6.2452 - start_probs_loss: 3.2016 - end_probs_loss: 3.0436
On epoch: 1, Exact Match score: 0.34 and F1 score: 0.44 on train data.

On epoch: 1, Exact Match score: 0.33 and F1 score: 0.42 on validation data.
113/113 [==============================] - 2329s 21s/step - loss: 6.2452 - start_probs_loss: 3.2016 - end_probs_loss: 3.0436
Epoch 2/3
113/113 [==============================] - ETA: 0s - loss: 3.3892 - start_probs_loss: 1.7453 - end_probs_loss: 1.6439
On epoch: 2, Exact Match score: 0.58 and F1 score: 0.67 on train data.

On epoch: 2, Exact Match score: 0.51 and F1 score: 0.61 on validation data.
113/113 [==============================] - 2319s 21s/step - loss: 3.3892 - start_probs_loss: 1.7453 - end_probs_loss: 1.6439
Epoch 3/3
113/113 [==============================] - ETA: 0s - loss: 2.5159 - start_probs_loss: 1.3012 - end_probs_loss: 1.2147
On epoch: 3, Exact Match score: 0.67 and F1 score: 0.76 on train data.

On epoch: 3, Exact Match score: 0.58 and F1 score: 0.68 on validation data.
113/113 [==============================] - 2313s 21s/step - loss: 2.5159 - start_probs_loss: 1.3012 - end_probs_loss: 1.2147
```

We start with Exact Match: 0.34 and F1 score: 0.44 on train data and Exact Match 0.33 and F1 score: 0.42 on validation data on first epoch. This is not that good. But on 3rd epoch we reach 0.58 EM and 0.68 F1 score. F1 68% is pretty good. Training for 5 – 10 more epochs could give excellent results. We stop here due to heavy load on CPU.

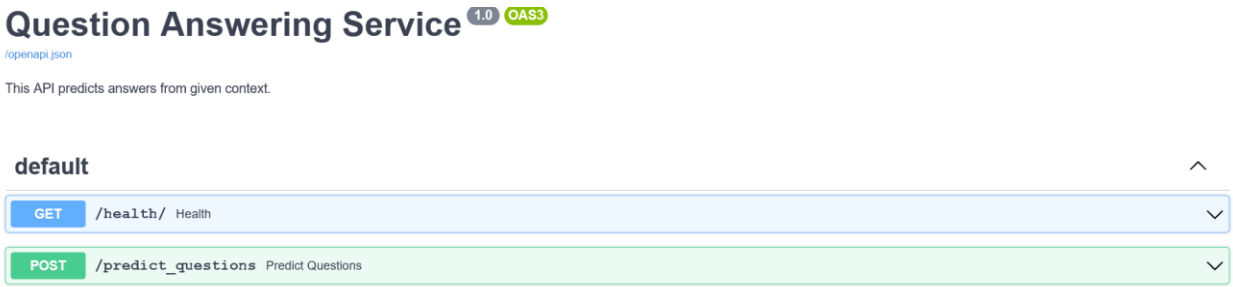
Model Evaluation and scoring

The evaluation of Squad data is performed from the Callback as we said. It calculates the F1 score and exact match. Those two are the benchmarks for the Squad dataset. The implementation is custom, but it follows exactly the way they are calculated in Squad v1.1 dataset. To note here that the v2 has a bit different method.

Model Predictions – API

API

We use the FastAPI api which can take any pair of context and question strings. For testing the trained model we provide a post /predict_questions. Try the http://localhost:5000/docs to see the documentation and experiment with methods as we see bellow.



The API takes a json of many Context and question pairs (as strings):

```
{
  "data": [
    {
      "context": "string",
      "question": "string"
    }
  ]
}
```

The FastAPi automatically validates the type(string) as they are defined in the api_models.py. Then we load the locally stored tokenizer and model, we preprocess the data similarly as before but using ApiQuestion class. This similarly calculates the input to the Bert model, but does not have target

outputs. Finally, we get the two predicted start and end points and return the answer text from the given context.

Let us evalatuate how it answers on some random context and questions we wrote.

```
curl -X 'POST' \
  'http://localhost:5000/predict_questions' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "data": [
    {
      "context": "The happiest place on planet earth is Greece after Italy
and Spain",
      "question": "What is the happiest place on earth?"
    },
    {
      "context": "Tonight I am going to eat Pizza with my friends and drink
juice.",
      "question": "What i am going to do tonight?"
    }
  ]
}'
```

For the above we get the response body

{
 "answers": [
  "Greece",
  "eat Pizza with my friends and drink juice"
 ]
}

We see that the first tricky question confused the Model as it is Italy instead of Greece. For the second simple question we get a really nice answer. Finally, let us make a question on larger context, using again our trained model.

{
 "data": [
  {
   "context": "BERT came up with the clever idea of using the word-piece tokenizer concept which is nothing but to break some words into sub-words. For example in the above image ‘sleeping’ word is tokenized into ‘sleep’ and ‘##ing’. This idea may help many times to break unknown words into some known words. If I am saying known words I mean the words which are in our vocabulary. We will see this with a real-world example later.",
   "question": "With what clever idea did BERT had?"
  }
 ]}

```
{
  "answers": [
    "word-piece tokenizer"
  ]
}
```


Yes! Our model can even answer questions about the model’s architecture!