Open Domain Natural Language Question Answering with Finetuned ALBERT Model

Lakshmi Sai Srikar Vadlamani (2K19/CO/208), Kushagra Singh (2K19/CO/204), Jayant Rana (2K19/CO/177)

Abstract

In our project we have created a Question Answering system that is capable of taking questions posed in natural language along with a corresponding context, and outputs the answer to that question. We have done so by fine tuning the ALBERT model on our dataset. This report details how we implemented the project.

1 Introduction

One of the core goals of artificial intelligence is to create systems that can answer complex questions posed to it in natural language based on any topic. These types of systems, known as Open Domain Question Answering Systems, are a benchmark task in the development of artificial intelligence, since the ability to understand text and answer questions about it is something we generally associate with intelligence. In our project we have built such a system which can take any question with a corresponding passage and output the answer to that question based on the passage. We have done this by finetuning the ALBERT model, which is a transformer based model, on the SQuAD(Stanford Question Answering Dataset) dataset.

2 Dataset

Stanford Question Answering Dataset(SQuAD) is a reading comprehension dataset, consisting of questions posed by crowdworkers on a set of Wikipedia articles, where the answer to every question is a segment of text, or span, from the corresponding reading passage, or the question might be unanswerable.
The SQuAD dataset contains many (context, question, answer) triples. Each context (sometimes called a passage, paragraph or document in other papers) is an excerpt from Wikipedia. The question (sometimes called a query in other papers) is the question to be answered based on the context. The answer is a span (i.e. excerpt of text) from the context.
This means our model will be given a paragraph, and a question about that paragraph, as input. The goal is to answer the question correctly.

3 Preprocessing of Dataset

3.1 Loading the Data

The first step in preprocessing the data is to load it in. There are two partitions for SQuAD given on their website, train-v2.0.json and dev-v2.0.json, and they are both given in the json format. In order to read the dataset we import python’s built-in library json. In the dataset there are several passages or ”contexts” which are text data, on each of which several questions are posed and their corresponding answers are given, making one data point for our dataset. The function we have written loads the context and the questions into lists of strings, and the answers are loaded as a list of dictionaries with each dictionary corresponding to a particular question including the answer string itself along with the starting character index of the answer.

3.2 Adjust indices and add end tokenposition

The next step is to add an end token index for each answer, i.e the index of the last token of the answer. We also adjust the indices to match the actual answers given in the dataset, since some of the answers in SQuAD are off by 1 or 2 indices dueto errors.

3.3 Tokenization of input

Our next step is to tokenize the input. The model we want to use for our project, ALBERT comes with its own tokenizer as it accepts inputs in a very particular form. This tokenizer converts each word in the question and context strings into integers which can be fed into the model (this is necessary since the model cannot accept strings in their raw form). The tokenizer take in parallel(corresponding) lists of contexts and questions and converts them to the form:

<CLS> + tokenized_context + <SEP>
tokenized_question> + <SEP>

where <CLS> and <SEP> are special tokens accepted by the model to indicate the start of the input and divide between the context and questions respectively. The tokenizer function also does padding, i.e it makes every input the same length by adding extra zeros to the side. It also creates an attention mask array, which is an array of 1's and 0's with the 1's corresponding to those token positions that have meaning or should be paid attention to, and the 0's corresponding to those that are padded on and shouldn't be paid attention to.

3.4 Convert char indices to token indices

The next step is to convert the answer_start and answer_end character indices to word(token) indices.

3.5 Put everything in a torch Dataset class

Finally, we put all of the processed data into a torch Dataset class. In order to define a torch Dataset class we need to inherit from torch.utils.data.Dataset and define three functions: an__init__() function which contains all the encodings of the data, a_len_() function which returns the length of the dataset and works with python's built in len function, and a_getitem_() function which returns a torch tensor containing a single data point given an index passed to the dataset.

4 Model

Bidirectional Encoder Representations from Transformers is a Transformer-based machine learning technique for natural language processing pre-training developed by Google. (ALBERT stands for A Lite BERT). The core idea of BERT is that the representation of a word (i.e word embedding) should be sensitive to the context of the word. This is in contrast to other word embedding techniques such as word2vec, which do not take the context of a word into account. BERT is based on transformers, which are attention-based models. The bidirectional part of BERT means that the model incorporates context on both sides of the word it is creating a representation of, rather than only looking at the left or right sides of the word.

BERT was a revolutionary model when it came out, achieving state of the art results in almost every NLP task. Transformer based models such as BERT have mostly taken over RNN based approaches as the dominant family in deep learning based NLP. BERT models can be easily adapted to several tasks by modifying the input and last layer and are called 'task agnostic' architectures. The original BERT models were pretrained on the concatenation of two massive unlabeled datasets extracted from the BooksCorpus and English Wikipedia with around 3300M words.

In our projects we use the transformers library created by HuggingFace. The pretrained BERT base model has around 110M parameters which is massive compared to previous models. The reason we use a pretrained model is due to the fact that pretraining such a model is incredibly computationally expensive, with BERT-large(340M parameters) costing $7000 dollars and 4 days to train. This means in order to get state of the art results our only realistic option is to fine tune such a pretrained model for our dataset.

ALBERT

ALBERT (stands for A Lite Bert) is a variant of BERT created in late 2019 that reduces the number of parameters by almost 89% while maintaining accuracy. Their key idea was to allocate the model's capacity more efficiently and to share parameters across the layers, among other improvements. This model can be trained quicker and scales better compared to the original BERT model.

We used Pytorch and the HuggingFace transformers library to define our model and include pretrained weights in it. In order to create a model in pytorch, we need to subclass torch.nn.Module, and define two functions: an __init_() function that defines the parameters of the model, and a forward() function that defines how the output is computer, i.e the forward propagation of the model. Our model is a stack of 12 transformer encoder layers (with shared parameters) with a fully connected layer at the end which outputs the logits(log odds) of the answer_start and answer_end tokens.

5 Training

5.1 Dataloader

Our first step is to create a dataloader for the dataset that we have preprocessed already so that we can sample batches from the dataset to feed into our model. We will take training batch size as 16, and the validation batch size as 32.

5.2 Loss Function

The loss function in the model is a standard cross entropy loss. Cross entropy loss is used when the output of a model is a probability distribution, in our case the probability distribution is over all the token positions at which the answer could start and end. The loss will be high if the predicted output greatly diverges from the ground truth output, and will be small when they are similar. Therefore our goal is to reduce the loss in order to improve the performance of our model.

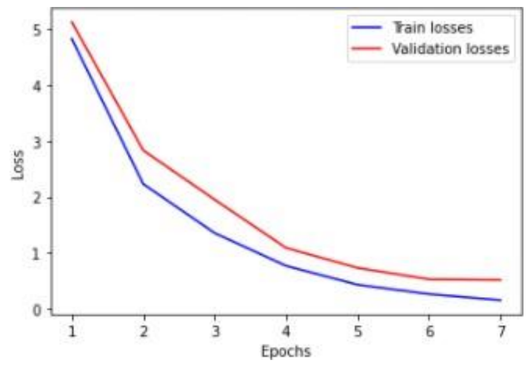
5.3 Optimizer

For our training loop we use the Adam optimizer (nn.optim.Adam) with a learning rate of 5e-5. We chose this since Adam is computationally efficient, has little memory requirements, is invariant to diagonal rescaling of the gradients, and is well suited for problems that are large in terms of data and/or parameters. Most of the research papers we have seen used Adam or some variant of it.

5.4 Main Training Loop

After sampling a batch from the data, it is passed through the model so that we can get the output for it. The first output for our model is the loss, which is the main output of interest for us here. After having computed the loss, we must backpropogate to compute the gradient of parameters wrt to the loss function by calling loss.backward(). We don't need to define our own backpropogation function since torch.autograd takes care of this for us automatically. Once we have computed the loss, we need to call optimizer.step(), which updates the parameters of the model and reduces our loss over many such subsequent steps. This is repeated for the length of the entire dataset for each epoch. We have set the number of epochs as 7.

We plotted the loss curves as shown below:



6 Evaluation and Results

In order to evaluate our model we use the F1 score which is the harmonic mean of precision and recall. To understand how this works, lets take an example: If our system outputs an answer 'Einstein' but the real answer was 'Albert Einstein', then our system would have 100% precision (as the output is a subset of the true answer) and 50% recall (as it only included one out of the two words in the ground truth). Therefore the F1 score would be

$$\frac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

We evaluate the model on a held out test set which we took from a portion of the dev set given in the dataset. Our model's F1 score is given in the table below, along with a few other model's F1 scores for comparison. (scores taken from the official leaderboard).

Model	**F1 score**
BiDAF(2016)	62.305
Unet(2018)	74.869
BERT base(2018)	77.402
Our model	84.126
Human performance	89.452

References

Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, Radu Soricut. 2019. *ALBERT: A Lite BERT for Self-supervised learning of language representations.* https://arxiv.org/abs/1909.11942

Pranav Rajpurkar, Robin Jia, Percy Liang 2018. *Know What You Don't Know: Unanswerable Questions for SQuAD. https://arxiv.org/abs/1806.03822*