Question-Answering With Pre-Trained Language Models

NLP Final Project, Spring 2020
Nathaniel Woodward

Abstract

This paper contains an introduction to the task of question-answering, the task I chose to solve as my final project, as well as the approaches, methodology, and concluding results. I decided to leverage pre-trained language models as the foundation of my solution. Then, on top of this through my research, implemented my own subsequent components to fine-tune the model for the task of question-answering in a different way. While my results fell short of current benchmarks, the knowledge gained is extremely valuable, and highlights how much goes into libraries that make solving downstream tasks manageable.

1 Introduction

The problem I decided to solve for my final project is generally speaking, question-answering. Question-answering has become one of the most important benchmarks to test state-of-art models on in the Natural Language Processing (NLP) space. The specific task that is being tested is not quite simply having the model return an answer to a question. Instead, the model, given the question as part of the pair with corresponding context or evidence, should identify the span within the context that it predicts contains the answer. This task is a great benchmark to test models on because it requires not only the baseline understanding of natural language, linguistics of the language being used, and word-meanings relative to each other, but requires the ability of the baseline model to adapt and translate to a very specific task that can be quite difficult.

I decided to choose question-answering as the problem I was going to solve because of its importance when setting state-of-art benchmarks. There is plenty of data on what the benchmarks are for any type of model, so however I chose to solve the problem would give me a benchmark for what to compare my results to. Also, opposed to say text classification, the value of a question-answering system is obvious as google uses it to show the answer to your question using the first returned link as context. Solving this problem is interesting, and in learning how to train a model for this specific task from a pre-trained language model and research, gives me the knowledge and skills going forward to apply towards many other downstream NLP tasks.

My first choice was which pre-trained language model to use as the baseline for my question-answering model. While ALBERT (A lite BERT model) has topped the BERT architecture on recent benchmarks, the results are a little misleading. The way in which ALBERT is "lite", is that all transformer layers share the same parameters. This saving in space gives the model the ability to expand wider and increase the word embedding space to 4096 in order to achieve better results than BERT – with roughly 70% of the parameters (Lan 2019). However, in order to train the model and inference with it, the data still needs forward pass through and conversely compute partial-derivatives of all transformer layers, regardless of parameter sharing. Between this added time to training and evaluation, and the fact that the increase in accuracy occurs only in large ensembles, I chose to use BERT as the baseline for my question-answering model. $\text{BERT}_{\text{BASE}}$ fits in the memory of the GPU available to me and therefore is the specific pre-trained language model I chose to work with.

The next part of the solution is adapting the pre-trained model to the specific task of question-answering. As previously described, the goal of the model is to predict the span in the context that contains the answer to the question or query. The best way to design this is to have the model predict the

index where the span containing the answer starts and the index where the span ends. By defining a loss function that calculates the cross-entropy, or difference, between the predicted span and actual answer span, we can minimize this loss across training examples thus fine-tune the model to predict the answer span amongst a question and context pairing. The specific methods and math used will be described in detail in the methodology section of this paper.

2 Related Work

2.1 Intro the BERT

As the final project is to identify a state-of-the-art approach and implement it as the baseline, a review of the original BERT paper is necessary. In the relevant sections of the original BERT paper referencing fine-tuning on the SQuADv1.1 dataset, both methods and results are well documented. In the paper, they introduce only an additional start vector, and end vector. These vectors have the same dimensionality as the hidden embedding layer. In $\text{BERT}_{\text{BASE}}$, this is 768. The reason, I deduced, was to have the same dimensionality as the token embeddings defined for BERT's vocabulary, which is necessary for how the loss is defined in the paper.

2.2 Related Methodology

The paper defines the probability of a word, i, being the start of the answer span as "a dot product between T_i and S followed by a softmax over all of the words in the paragraph" (Devlin 2018). In this description, T_i is the word embedding for a specific token, or word in the context paragraph, and S is the start vector described before. The dot product between the two vectors will produce a "similarity" metric between the two vectors, and thus the softmax calculation across all words or tokens in the context paragraph will produce a probability, where the word that has the probability essentially has values closest to the start vector than any other possible word. The cross-entropy loss is calculated across the word probabilities and a one-hot vector with a one contained in the index of the start/end word. The sum of this calculation for both start and end predictions is the total loss, and the training objective of the model. When inferencing for predictions, the i, j (start, end word) pair that is chose the one such that $S \cdot T_i + E \cdot T_j$ is the maximum, given j appears later in the text than i.

2.3 Related Results

When the BERT paper was published, they found their results on SQuADv1.1 "outperforms the top leaderboard system by +1.5 F1 in ensembling and +1.3 F1 as a single system" (Devlin 2018). The results of how their results compared to the leaderboard at the time is shown below.

System	Dev		Test	
	EM	F1	EM	F1
Top Leaderboard Systems (Dec 10th, 2018)				
Human	-	-	82.3	91.2
#1 Ensemble - nlnet	-	-	86.0	91.7
#2 Ensemble - QANet	-	-	84.5	90.5
Published				
BiDAF+ELMo (Single)	-	85.6	-	85.8
R.M. Reader (Ensemble)	81.2	87.9	82.3	88.5
Ours				
$\text{BERT}_{\text{BASE}}$ (Single)	80.8	88.5	-	-
$\text{BERT}_{\text{LARGE}}$ (Single)	84.1	90.9	-	-
$\text{BERT}_{\text{LARGE}}$ (Ensemble)	85.8	91.8	-	-
$\text{BERT}_{\text{LARGE}}$ (Sgl.+TriviaQA)	**84.2**	**91.1**	**85.1**	**91.8**
$\text{BERT}_{\text{LARGE}}$ (Ens.+TriviaQA)	**86.2**	**92.2**	**87.4**	**93.2**

The BERT model achieved fantastic results, beating the previous leaderboards even without first training on TriviaQA. The result that is the reference for the system that I built is the F1 score for the $\text{BERT}_{\text{BASE}}$ (Single) model. Given I am using the $\text{BERT}_{\text{BASE}}$ model and only training the one (not an ensemble of 7 models all hyper-parameter tuning), the F1 score that is the reference for my results is a score of 88.5.

3 Methodology

3.1 Main Components

My chosen methodology is a product of my research as well as my own experiments within the confines of using BERT. Therefore, while a lot of my implementation is meant to mirror state-of-the-art implementations, it is implemented with my own helper functions and metrics defined; this leaves some room for error, but to the benefit of trying a new approach – the current state has already been implemented before.

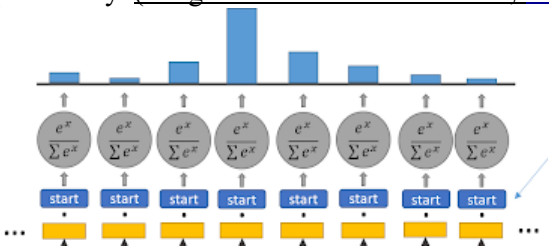
The dataset I used was Stanford's question-answering dataset, SQuADv1.1 (Rajpurkar 2016). To train and evaluate my model I used the files train-v.1.1.json and dev-v.1.1.json, respectively. The specific version of the pre-trained BERT transformer model I used was 'bert-base-uncased' from the transformers library (huggingface), published

by HuggingFace. While there exist PyTorch implementations that adapt the last hidden layer through a model-specific layer to the desired outputs for question answering, and scripts that perform fine-tuning with starter output weights to the output layer that yield very close to the 88.5 F1 baseline referenced earlier, I wrote all of the code myself. Both to force research to truly understand the problem and what's involved, but also to gain the knowledge for how to adapt these powerful language models to other downstream tasks in the future without relying on any specific library or architecture.

3.2 Model Adaptations

My model adaptations and training objectives were slightly different from those described in the original BERT paper. While theirs involves taking the BERT output (torch.size([768])), and applying two separate sets of weights, S_w and E_w to yield two 768 vectors, this implies that a lot of the code producing dot-products with possible tokens and normalizing this value among all other token possibilities, exists in the training loop, unless the defined model class has the dataset available to it during forward passes.

I adapted BERT to the task of question-answering by taking embedding dim (768 in our case) E, and the max sequence length (I chose 512 due to memory constraints), M, and putting the BERT output through two separate linear layers, applying a matrix of weights [768 x 512] to produce a start vector, and end vector, both of the dimensionality torch.size([1, 512]). Because the max sequence length is 512, and all sequences are either padded up to, or truncated at 512, each token from the tokenized input can be represented by the corresponding index in the output of this model. It follows that the probability of a token T_i being the start span of the answer is the value in the start vector, simple at index i, followed by a softmax over all tokens in the input. The same process follows for the end span probabilities as well. The figure below illustrates the computation of each indices probability. (Image credit Chris McCormick, site)



One special consideration is the fact that the question, special tokens and padding is included in the input_ids as well. It would be a waste for any of the probability distribution to go towards indices at these values. The attention mask should cover the padding tokens from being attended to, while the segment ids give information identifying the question span from the context span, which luckily includes special tokens, like [CLS], and [SEP], required by BERT as part of the input formatting. This only leaves the final [SEP] token following the end of the context. We can entertain the idea of this not being attended to if the BERT architecture allows this.

The next implementation detail is the training objective, what are we trying to minimize? Given the design of my model is to produce the probability distribution among the token indices, it follows that the loss function be defined as the cross entropy between the probability distribution and a one-hot vector containing all zeros, with the exception of a one in the index of the target. Depending on the vector, either the index containing the true start token or true end token of the answer span. This means, the less probability predicted in the index of the correct token index, the higher the cross-entropy between itself and a vector with just a one in the correct index, the higher the loss. The total loss is simply defined as the combination of the two losses (both start and end vectors):

```
nn.BCELoss()(start_pred, start_tar-
get)*0.5 + nn.BCEL-
oss()(end_pred, end_target)*0.5
```

3.3 Training Details

Based largely from what I've seen in my research, I decided on the following parameters during training. Due to memory constraints I chose 512 as the max token sequence length. I trained my model using GPUs for 3 epochs at a learning rate of 3e-5 (0.00003). Additionally, I used the AdamW optimizer function from the transformers library.

Using the model and dataset classes I defined, it allowed for easy creation of a data loader and training loop that has minimal steps from inferencing to calculating the loss. In fact, the way I designed the classes allowed the output of the models to go straight into the loss function with one-hots of the targets at hand. The training took roughly ~12 hours from my runs.

4 Experiments and Results

4.1 Experiment Setup

While I used the training set from SQuADv1.1 to train my model, I used the dev set for the experiment and evaluation part of this project. While the training set contains over 100,000 examples containing a single answer, the dev set contains roughly 15,000 examples with up to three possible "ground truth" answers that are all accepted.

I designed my classes for the model and dataset such that either dataset can be given to the parser and then to the dataset class. When queried for a specific index by the data loader, it will return the specific question, context pair from the index using a context map (No need to save the same context for each question), and search across the context for each answer pulled from the parsed data to return each start, end index paring for each "ground truth" answer. Because the design was intended to aid in easily training and evaluating from the same classes, the evaluation loop wasn't very much different. Using my defined data loader, I went through each example, only calculating metrics from inferencing rather than calculating a loss and back-propagating.

4.2 Evaluation Methods

I chose the implement a calculation of the F1 score that all other state-of-the-art baselines have also used as a metric. The F1 score is the harmonic mean between how much of the prediction is contained in the answer and how much of the answer is contained in the prediction. For example, given a prediction "Julian", and an answer "Julian Francis Edelman", the precision would be 100%, as all of the words in the prediction are contained in the answer. However, the recall would be 33%, as only half of the words in the answer are contained in the prediction. F1 is calculated as:

F1 = 2*Precision*Recall / (Precision + Recall)

This would give my Edelman example an F1 score of roughly 50%. Since the dev dataset contains up to three unique answers (most are the same) for each question-context pair, the F1 score for a given question-context pair, is the highest F1 score the model's prediction registers among the three "ground truth" answers.

4.3 Model Results

After training and evaluation as described in this paper, my model recorded an F1 accuracy across the dev set of **30.1%.**

These results do not approach near state-of-the-art examples. However, while not inspired by the results, I am happy that I tried a different approach to modeling the NLP question. I modeled the question how I would frame it being given the question otherwise, and wanted to see how the results would compare. With small tweaks back to how the original BERT paper described the implementation, results improve dramatically, which shines a huge light on how crowdsourcing data and how you design a problem both conceptually and architecturally plays a massive role in the results.

After the initial frustration, I realize that an F1 score of 30%, given the average size of the answer spans, means that more often than not, my model returned an answer span that contains a piece of the answer. With a probability distribution across up to 512 tokens, this means that the model was trained on this task to some success. Upon further investigation, I realize that this is simply the cost of how I designed the problem rather than a run-time error which would yield much worse results, likely not containing any of the answer.

4.4 Conclusions / Future Work

My future work in this space will be to implement the baseline as described in the BERT paper to first make sure I achieve a roughly 88% F1 score, and then extend the model with a few ideas.

A few experiments I was to run once the baseline is implemented are including a gateway "highway" architecture (though I see this being more useful in a SQuAD-v2.0 setting where there are unanswerable questions), and pre-training on SQuAD-v2.0 with highways and then on v1.1. Lastly I would likely to try weighting the loss of the start cross entropy higher than the end cross-entropy loss. My intuition is that the start span is more important in application settings, and could also help ensure that the end index isn't ever predicted before the start.

My first and main takeaway is that the most important piece of solving an NLP task, most especially a new task, or one with limited data, is how you design a problem. While my attempt using a different method didn't yield results near state-of-the-art numbers, I do take some pride in designing a different solution to the problem and seeing progress – no matter how small.

References

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. CoRR, abs/1810.04805, 2018.

Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. arXiv preprint arXiv:1909.11942, 2019.

Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100, 000+ questions for machine comprehension of text. CoRR, abs/1606.05250, 2016.