CS224n Final Project: SQuAD 2.0 with BERT

Kevin M. Lalande
Department of Computer Science
Stanford University
klalande@stanford.edu

Abstract

This paper adapts pre-trained models of Bidirectional Encoder Representations from Transformers (BERT) to the natural language processing task of answer span prediction ("QA") on the Stanford Question Answering Dataset (SQuAD 2.0). The original BERT implementation ("`origBERT`") presented in Devlin et al. [2018] achieved SOTA performance of 88.5 F1 on a SQuAD 1.1 dataset with 100,000+ answerable context-question-answer tuples. However, when `origBERT` is used on the much more challenging SQuAD 2.0 dataset, released subsequently with 50,000+ additional questions that are impossible to answer from the context paragraph, performance drops to 73.1 F1. This paper presents a combination of refinements to the task-specific tuning in `origBERT`, including: a multi-layer QA architecture, an ensemble with a binary sequence classification model trained to identify impossible questions, a larger fine-tune training data set including examples from TriviaQA, and optimized hyperparameters. SQuAD 2.0 performance improves from a **baseline of 73.1 F1 to a best F1 score of 77.6 for $\text{BERT}_{\text{base}}$ and 81.4 for $\text{BERT}_{\text{large}}$**. Finally, this paper concludes with the observation that these refinements allow an optimized $\text{BERT}_{\text{base}}$ model to perform almost as well (-2.3 F1) as an $\text{origBERT}_{\text{large}}$ model, which has 3 times as many parameters and is considerably more challenging to train.

1 Introduction

BERT Devlin et al. [2018] introduce a new language representation model which achieves breakthrough performance on eleven different NLP benchmarks, including SQuAD, covering a wide range of useful token- and sentence-level tasks. Outperforming state-of-the-art on any single benchmark is a noteworthy accomplishment in and of itself, even though the new high water mark is often temporary. The BERT paper is particularly interesting because `origBERT` simultaneously beats a dozen well-established benchmarks with a single conceptual leap.

Furthermore, BERT achieves this remarkable performance while also employing a generalizable version of transfer learning, in which a universal set of pre-trained contextual representations are later trained again on supervised task-specific data jointly with a single additional output layer as BERT is "fine-tuned" to perform a particular downstream NLP task. This contrasts with the more commonly used “feature-based” strategy, in which a fixed set of possibly task-specific pre-trained representations are input into a model that has been custom engineered to perform well on a single task. The fact that BERT is capable of producing state-of-the-art results on a wide range of NLP benchmarks with a generalizable transfer learning strategy suggests that the authors have uncovered a fundamentally better language model representation.

SQuAD Motivated by the importance of large datasets of high-quality in the advancement other machine learning fields, like ImageNet for object recognition or Penn Treebank for syntactic parsing, Rajpurkar et al. [2016] released SQuAD 1.1, a reading comprehension dataset of 108,000 questions posed by human crowdworkers for which the answer can be extracted from a contiguous span of

text within the corresponding context paragraph. In the BERT paper, `origBERT` performed quite well on this benchmark, with the `base` model achieving 88.5 F1 on the Dev split and the `large` model achieving score of 91.8 F1 on the Test split, beating both previous SOTA and human-level performance.

After the BERT paper was submitted for publication, Rajpurkar et al. [2018] released a new SQuAD 2.0 benchmark which combines the prior data with an additional 54,000 unanswerable questions that were written adversarially by human crowdworkers to look similar to questions that do have answers. This is a more challenging task because QA systems must be able to identify questions that are not answerable from the context paragraph as well as correctly answer those which are. When an equivalent implementation of `origBERT` is applied to this new dataset, performance of the `base` model on the CS224n Dev split (which is half of the official SQuAD Dev split) drops to 73.1 F1.

This paper explores several potential changes to the fine-tuning process used by `origBERT` in order to improve performance on SQuAD 2.0 above this **baseline of 73.1 F1**.

2 Related Work

BERT is built on two conceptual foundations: 1. pre-trained contextual embeddings (PCEs), which are, in turn, an extension of distributed word embeddings, and 2. the Transformer architecture.

Distributed Word Embeddings Multi-dimensional vector representations of words like the skip-gram Word2Vec model by [Mikolov et al., 2013] and the hybrid GloVe model by Pennington et al. [2014], which adds global co-occurrence probability ratios, use unsupervised neural networks to efficiently encode semantic and syntactic relationships by unsupervised learning on a large corpora of unstructured text. When pre-trained distributed embeddings are later fed as fixed inputs into a natural language application, the resulting model outperforms statistical language strategies with atomic word representations, such as the popular N-gram model, no matter how much training data is supplied Brants et al. [2007]. This is due in part because of the transfer learning advantage conferred with multi-dimensional word representations learned over billion-word corpora. Furthermore, Word2Vec outperforms all prior work in its own category of neural-network derived word representations, including Collobert and Weston [2008], Mnih and Hinton [2009], Turian et al. [2010], and the hybrid GloVe model outperforms Word2Vec.

Pretrained Contextual Embeddings The concept of distributed word representations was extended to learn more than a single representation for a given word token, dependent upon the broader context beyond the local window in which the word is used. This was first proposed in the language model automated sequence tagger (TagLM) by Peters et al. [2017] and then improved shortly thereafter with Embeddings from Language Models (ELMo) by Peters et al. [2018]. EMLo produces contextualized embedding vectors derived from a deep bi-directional LSTM (BiLSTM) with self-attention that is trained with using a semi-supervized language model objective on a large text. Pre-trained ELMo weights are frozen and concatenated as input features into a supervised model with task-specific architecture. Like BERT, EMLo performed well on a broad range of natural language tasks including textual entailment, sentiment analysis, and question answering.

Transformer Architecture Vaswani et al. [2017] showed that the powerful but computationally slow sequential RNNs widely used in neural language models can be replaced with a simpler and more parallel network architecture, the Transformer, which is based solely on the self-attention mechanisms. This new architecture was adopted by Radford et al. [2018] in the Generative Pre-trained Transformer (OpenAI GPT).

Points of Comparison BERT, ELMo, and OpenAI GPT each produce PCEs. For downstream tasks, ELMo follows a feature-based approach in which its PCEs are fed as frozen inputs to a model-specific architecture, while BERT and OpenAI GPT both fine-tune their representations jointly with an additional output layer. ELMo's architecture is a BiLSTM, while BERT and OpenAI GPT both use Transformers. Perhaps the most significant difference among the three models is that BERT alone trains a bidirectionally conditioned language model, while OpenAI GPT trains a unidirectional Transformer and ELMo concatenates two independently trained unidirectional LSTMs.

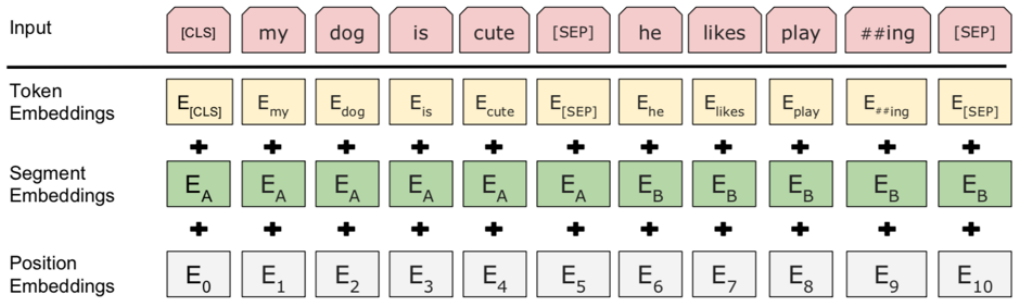


Figure 1: BERT input representation.

3 The BERT Model

3.1 Inputs

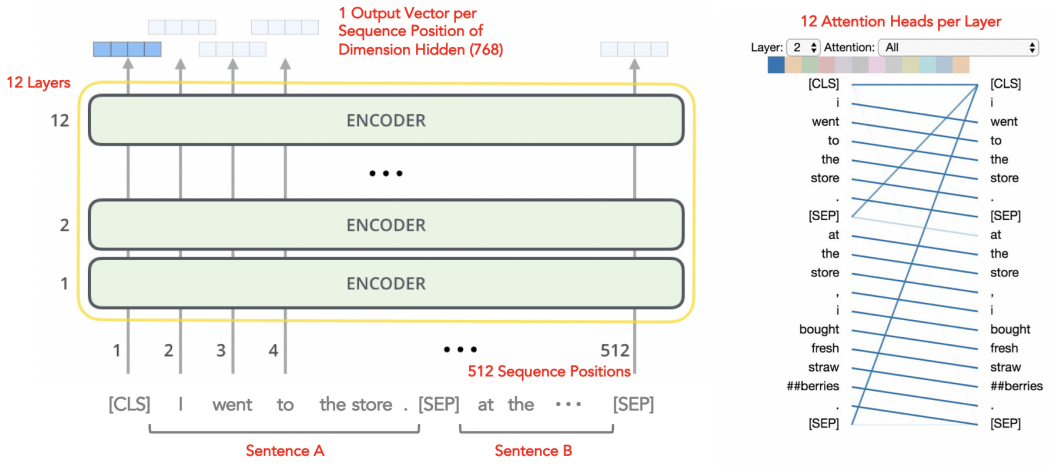
The input scheme is designed to represent either a single sentence or an A/B pair of sentences, where the word "sentence" is understood to mean any arbitrary span of contiguous text up to 512 total tokens rather than a well-formed linguistic sentence.

As shown in Figure 1, the BERT input representation is constructed by summing: (i) the corresponding WordPiece input token embedding Wu et al. [2016] using a 30,000 token vocabulary, (ii) a learned segment A embedding for every token in the first sentence and a segment B embedding for every token in the second sentence, and (iii) learned positional embeddings for every token in the sequence up to 512. Special tokens are used to designate the beginning of a sequence [CLS] and the end of a sentence [SEP]. A packed pair of A/B sentences is defined as an input sequence.

3.2 Architecture

BERT replaces the standard RNN-based language model with a purely attention-based architecture using a multi-layer bidirectional Transformer encoder as presented in Vaswani et al. [2017] to map an input sequence of discrete tokens $(x_1, \ldots, x_n)$ to a sequence of continuous representations $\mathbf{z} = (z_1, \ldots, z_n)$. Two different versions of the model are trained denoted by the number of layers L, the hidden size H, and the number of self-attention heads A. $\text{BERT}_{\text{base}}$ has 110 million parameters with L=12, H=768 and A=12. $\text{BERT}_{\text{large}}$ has 340 million parameters with L=24, H=1024 and A=16.

Figure 2: (Left) $\text{BERT}_{\text{base}}$ model with 12 Transformer Layers with 12 Attention Heads per Layer. (Right) Example visualization of attention weights for all tokens.



As illustrated in Figure 2 (Left), BERT uses multiple layers of attention (12 or 24), each with multiple attention heads per layer (12 or 16). Model weights are independent at each layer, so a single BERT model can have 144 or 384 different attention mechanisms. Figure 2 (Right) shows example output from visualization tool developed by Vig [2018] to provide insight into general patterns learned by BERT's attention mechanisms. The tool visualizes attention as lines connecting the sequence position being updated (left) with the position being attended to (right). Colors identify the corresponding attention head and line weight reflects magnitude of the attention score. General patterns include attention to: next word, previous word, delimiter tokens, bag-of-words, identical or related words in same sentence, identical or related words in other sentence, and words that are predictive of a given word.

3.3 Pre-training Procedure

Devlin et al. [2018]'s unique contribution in BURT is an innovative new unsupervised technique to train a deep bidirectional language model that does not suffer from the information leakage problem that constrains all prior conditional language models to be either unidirectional or a concatenation of independently trained directions. BERT trains using two novel unsupervised prediction tasks:

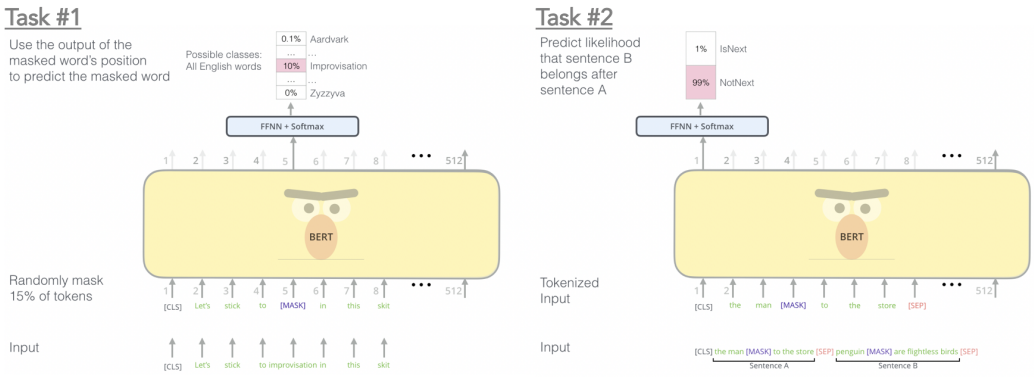
1. **Masked Language Model (MLM):** To train deep bidirectional representations, the authors randomly replace 15% of the input tokens with a special [MASK] designation and set the training objective to correctly predict each masked word using only the fused context from both the left and right side of the mask. This procedure, known as the Cloze task, is well-established in the literature Taylor [1953].
2. **Next Sentence Prediction (NSP):** Some NLP tasks, such as SQuAD, require an understanding of the relationship between two sentences, which is not directly captured by standard language models. To train BERT with more of an understanding of sentence relationships, an additional binary prediction task was added for each pre-training example in which a second sentence B `isNext` or is `notNext` the next sentence following sentence A.

Dataset and Hyperparameters The pre-training dataset includes BooksCorpus (800M words) Zhu et al. [2015] and English Wikipedia articles ignoring lists, tables and headings (2.5B words). The model is trained in batches of 256 sequences by 512 tokens each for 1 million steps, which requires 40 epochs over the entire 3.3 billion word corpus. Adam optimizer is used with a learning rate of 1e-4, β_1 = 0.9, $\beta_2 = 0.999$, L2 weight decay of 0.01 and a linear decay learning rate beginning after the first 10,000 steps. Dropout with a probability of 0.1 is used on all layers. Finally, instead of a ReLU activation, the authors use a Gaussian Error Linear Unit, or gelu, activation Hendrycks and Gimpel [2016].

3.4 Final Training Procedure ("fine-tuning") for SQuAD QA Task

The prediction task in SQuAD is to select the start and end token within a given paragraph of text which represent the answer to a given question. The (Question, Paragraph) input is represented as

Figure 3: BERT Unsupervised Pre-Training Tasks



a packed sequence with the question assigned the A embedding and the paragraph assigned the B embedding. As illustrated in Figure 3(c) in the paper, the final hidden state from pre-trained BERT for the i^{th} input token is $T_i \in \mathbb{R}^H$. In the fine-tuning process, new parameters are learned for the start vector $S \in \mathbb{R}^H$ and end vector $E \in \mathbb{R}^H$. The probability of a given token being the start or end of the answer span is calculated as the softmax dot-product between T_i and either S or E, with the additional constraint added that the end token must come after the start token in the paragraph span.

$$P_i = \frac{e^{S \circ T_i}}{\sum_j e^{(S \circ T_j)}}$$

4 Approach

My objective is to improve the F1 performance score of pre-trained $\text{BERT}_{\text{base}}$ and $\text{BERT}_{\text{large}}$ models that are fine-tuned for SQuAD v2.0. I began by integrating and extending the Codebase listed below as described in Testbed Setup to reproduce the implementation presented in Devlin et al. [2018] and to support experimentation. I then applied this model to SQuAD v2.0 to establish the Baseline below. Finally, as described in Improvements below, I pursued several potential refinements to the fine-tuning procedure used in `origBERT`, including: a multi-layer QA architecture, an ensemble with a binary sequence classification model trained to identify impossible questions, a larger fine tuning training data set with examples from TriviaQA, and optimized hyperparameters.

Codebase: I use DFP starter code and a PyTorch implementation of pretrained-BERT (Huggingface) that produces identical results to the TensorFlow-based model presented in Devlin et al. [2018], as well as an NVIDIA library for mixed-precision and distributed GPU training and a reading comprehension dataset with 650K additional question-answer tuples.

- `https://github.com/chrischute/squad`
- `https://github.com/huggingface/pytorch-pretrained-BERT`
- `https://github.com/nvidia/apex`
- `https://github.com/mandarjoshi90/triviaqa`

Testbed Setup: Significant work was required to understand, integrate and extend the multiple repositories of code and data used in this project. The Huggingface codebase is the foundational scaffold, which I have made extensive modifications to and stack-traced every last line of `[modeling.py, run_squad.py, run_classifier.py, tokenization.py, optimization.py, file_utils.py]` to fully understand the implementation details and to make sure my integrations and code extensions are working properly. Here is a brief summary of the key elements of my testbed setup work:

- **Output File Management and Logging** ML projects need a consistent and automated output file naming convention to synchronize across multiple machines and keep track of dozens-to-hundreds of experimental file results (178+ as of this writing). This did not exist, so I implemented the functionality and added a new `-time_stamp` CLI argument which is set automatically to `YYDDMM-HH_MM` of script runtime if not specified, or is pre-pended to load specific `[log, bin, json, csv]` files if specified. I also extended the base logging to include a FileStream handler for saved log files in addition to the I/O console.
- **Tensorboard and Test-Val Model Checkpoints** Integrated PyTorch api for logging with Tensorboard from Hollander [2018]. Extended the existing script functionality to allow me to monitor train-val splits, save the best validation checkpoint models, and reload and continue training a checkpoint or previously fine-tuned model. Split the Train dataset into 90% train 10% validation so the Dev dataset can be used for Leaderboard submissions.
- **Tiny Data Set** Implemented a new `-tiny_data` CLI flag to load just a few examples from the `[train, dev].json` files to allow the entire code base to be rapidly debugged on a local machine without waiting too long for data processing between debugging runs.
- **Model Training Speed** Installed and debugged NVIDIA's Apex module, optimized the related `-[batch_size, fp16, loss_scale]` CLI settings by trial-and-error to speed up

training time from 7 days (estimated) on my local machine to 4-hours per epoch on an Azure NC6. This was still slower than I'd like at 8-12 hours per full model fine-tuning run, so I configured a second NC24s_v3 VM with four Tesla V100 GPUs which is more expensive but trains much faster at 35 mins per epoch. This capability, in turn, also allowed me to also train a $\text{BERT}_{\text{large}}$ model to compare its performance against my other changes, as shown in results below.

- **Leaderboard Submission** Extended the `write_predictions()` function to output a second version of `split-predictions.csv` that complies with the format specified for the DFP SQuAD Leaderboard.
- **Error Analysis** Added new code module to examine example-by-example prediction errors and to conduct split-level analytics of the types of questions the model gets `no_match, partial_match,` and `exact_match` as defined by the EM, F1 and AvNA calculations in the DFP starter code, which in turn was pulled from the original SQuAD paper Rajpurkar et al. [2016].

Baseline: Devlin et al. [2018] $\text{BERT}_{\text{base}}$(Single) produced an F1 of 88.5 on SQuAD 1.1. Applying my PyTorch equivalent version of their `origBERT` implementation to SQuAD 2.0 produces the project **baseline of 73.1 F1**. This result is what I expected based on the v2.0 Leaderboard lowest $\text{BERT}_{\text{base}}$ single model posted Jan 19, 2019 of 74.4 F1.

Improvements: I considered but ruled out potential improvements to the pre-trained BERT models and underlying Transformer architecture, like: a) larger WordPiece token vocabulary, b) a third unsupervised learning task, c) additional pre-train corpora like TriviaQA, d) more layers and/or attention heads. Pre-training times alone would make these impractical within the given time constraints. Therefore, I narrowed my exploration to the following potential improvements to the fine-tuning process in `origBERT`:

- **Deeper task-specific QA architecture** Since performance drops in v2.0 because a third of the questions are unanswerable, the most promising place to start is a task-specific training architecture that can learn more powerful representations than the single fully-connected layer in the baseline model.

  I built 3- and 6-layer linear QA models based on the `BertForQuestionAnswering` class. The first layer takes as input the output from the last layer of pre-trained BERT with `shape[batch, sequence_length, hidden_size]`. Dropout regularization with 10% probability is applied between each layer. In the 3-layer model, the size of the hidden units is halved at each layer from $768 \rightarrow 384 \rightarrow 192 \rightarrow 2$. The 6-layer model has two of each of the layers in the 3-layer model. The final two units in the output layer are split into `start_logits` and `end_logits` of `shape[batch, sequence_length]` to calculate the probabilities according to formula (1) above of each position in the sequence being the start or end of the correct answer span.
- **Concatenate multiple BERT output layers** In `origBERT`, only the hidden-states corresponding to the last encoding layer are fed into the QA architecture. I tried other combinations, including concatenating the last 4 encoding layers for an input to QA of `shape[batch, sequence_length, 4 * hidden_size]`, to see if access to more hidden-states from earlier layers could improve performance.
- **Weighted cost function** The classes of `is_impossible=True vs False` are unbalanced in the entire dataset (33/67) and, according to my Error Analysis, also between the Train and Dev splits. Furthermore, incorrect impossible questions are penalized more heavily (F1 score = 0) than partially incorrect answerable questions (0 < score < 1). For these reasons, a weighted loss function may help. I implemented class weighting in the binary `BertForSequenceClassification` discussed next. But in QA, loss on incorrect impossible questions is calculated based on the distance from position zero, which varies depending on question length. I couldn't come up with a convincing way to weight this function, so I opted instead to balance the Train dataset using more impossible examples from TriviaQA.
- **Ensemble two different BERT models** I trained a second class of model, `BertForSequenceClassification` (BSC), with its sole objective being the binary classification of whether an input sequence `is_impossible`. That required a full round of

code development and tuning, details skipped for space. I then ensemble the results of the QA and BSC model as follows: Default answer is QA prediction. If BSC predicts `is_impossible` then BSC overrides. If BSC predicts `not is_impossible` but QA predicted `is_impossible`, then QA replaces its NULL answer the next best score as the final predicted answer.

- **Failed experiment: Train on Larger QA Data Set** I spent several days writing, testing and troubleshooting the code necessary to extend the SQuADv2.0 train dataset with additional examples from the TriviaQA dataset (Joshie et al., 2017) as suggested in Devlin et al. [2018]. Unfortunately, I was not able to replicate their results. Models trained with augmented datasets perform significantly worse. I had to move on in the interest of time.

5 Experiments

	Run	Model	Epochs	Batch Size	LR	# QA Layers	Best Test-Val	Concat Multiple PCE Layers	Ensemble QA and Classifier	EM	FM	Positive Results	Negative Results	Conclusions
1	190209-21_59	bidaf	30	64	5.00E-01	NA	N	NA	NA	56.23	59.870			Baseline bidaf
2	190228-15_54	bert-base	2	32	3.00E-05	1	N	1	N	69.859	73.113	13.243		**Baseline bert-base**
3	190304-01_15	bert-base	2	32	3.00E-05	1	N	1	N	70.418	73.660	0.547		Model parameters tuned
4	190302-23_19	bert-base	6	32	3.00E-05	1	N	1	N	70.517	74.158	0.498		Modest improvement for 3x training
5	190304-02_46	bert-base	2	32	3.00E-05	3	N	1	N	72.902	76.045	1.887		3-layer QA is effective
6	190311-14_13	bert-base	2	32	3.00E-05	6	N	1	N	73.017	76.101	0.056		6-layer little additional benefit
7	190311-17_25	bert-base	6	32	3.00E-05	6	N	1	N	72.096	75.723		-0.378	6-layer overfitting; need Test-Val chkpts
8	190313-12_56	bert-base	2	32	3.00E-05	3	N	4	N	72.655	75.644		-0.457	Concatenating top 4 bert layers doesn't help
9	190316-00_48	bert-base	2	32	3.00E-05	3	Y	4	N	73.478	76.221	0.120		Train model until Test-Val diverges
10	190304-02_46--Ens	bert-base	2	32	3.00E-05	3	Y	1	Y	74.696	77.346	1.125		Ensemble BertForSequenceClassification
11	190304-02_46--Ens	bert-base	2	32	3.00E-05	3	Y	1	Y	75.074	**77.589**	0.243		**Best bert-base**
12	190303-05_18	bert-large	2	16	3.00E-05	1	N	1	N	76.802	79.883	2.294		**Baseline bert-large**
13	190304-04_13	bert-large	2	16	3.00E-05	3	N	1	N	75.930	78.762		-1.121	3-layer QA boost does not translate to bert-large
14	190304-13_02	bert-large	4	16	3.00E-05	3	N	1	N	75.304	78.443		-0.319	Performance decreases with more training
15	190317-00_41	bert-large	4	16	3.00E-05	3	Y	1	N	75.584	78.978		-0.905	Verified overfitting; drop to 3-Layer with Test-Val
16	190319-12_45	bert-large	2	16	3.00E-05	3	Y	1	N	78.005	**81.375**	1.492		**Best bert-large**

Figure 4: Summary of Experimental Results

Results from sixteen experimental model runs are listed in Figure 4 above. **Data:** Each uses data from the official SQuAD 2.0 training set to either train (BiDAF) or fine-tune (BERT) the model and modified splits for `dev` and `test` were created by CS224n faculty by dividing the official SQuAD 2.0 `dev` dataset roughly in half. All of the results shown are based on the `dev` split. **Evaluation Method:** EM and F1 scores from the PCE Leaderboard are listed. Each is defined in the DFP Handout and recalculated by my code, which produces results that match the Leaderboard. F1 is the primary objective, and any unlabeled performance number in this paper is intended to refer to F1. **Experimental details:** The type of model, number of training epochs, batch size, learning rate, and configuration of the task-specific QA architecture are also listed to help explain the differences among the various experimental runs. Other experimental details not listed are consistent with Devlin et al. 2018.

Results: I improved $\text{BERT}_{\text{base}}$ by 4.5 points above baseline to 77.6 F1. Notably, this is only 2.3 points below the `origBERT`$_{\text{large}}$ baseline performance, which is 3x larger and considerably more expensive to train. I achieved a maximum F1 of 81.4 with a modified $\text{BERT}_{\text{large}}$ model. A few observations:

- The big jump of 13.2 points from BiDAF baseline to $\text{BERT}_{\text{base}}$ baseline quantifies how much more powerful the PCE-based models are for the SQuAD task. It was exciting for me to re-create this result.
- The 6.8 point increase from baseline $\text{BERT}_{\text{base}}$ to baseline $\text{BERT}_{\text{large}}$ is driven by the increased power of a BERT model with 340M vs 110M parameters rather than by any improvements I made. That said, it was not trivial to debug the NVIDIA Apex modules or to pay for the more expensive NC24s_v3 machines capable of fine-tuning a $\text{BERT}_{\text{large}}$ in any reasonable amount of time. This capability has been helpful to me in exploring the effects of BERT model size, batch size, training epochs, and task-specific QA architecture.
- The modified 3-layer QA architecture described above produces a legitimate 2.4 point improvement. However, increasing QA depth again from 3 to 6 layers resulted in little

additional improvement (+0.05 F1). 6-layer performance then decreased by -0.4 F1 when the model was trained six epochs instead of two, suggesting the need for early stopping at Train-Val checkpoints to prevent overfitting. I therefore decided to use the 3-Level QA architecture. $\text{BERT}_{\text{large}}$ also benefits from the 3-Layer QA. But it is much quicker to overfit, so the benefit only became clear in Run #16 once Train-Val checkpoints was implemented.

- Concatenating the BERT's last 4 encoding layers for input to QA ended up hurting performance by (-0.5 F1), to my disappointment as I had high hopes for this approach. Performance improved once the Test-Val checkpoint was implemented in the next run, but the 4-layer concatenation was still a drag on overall performance so I abandoned it for the standard approach of using the last-level only.
- The two-model ensemble described above was effective in increasing performance by an additional 1.4 F1. I believe this is in part due to the weighted cost function used in the binary classifier to better capture actual class balance, and in part due to the advantage of one model focusing solely on whether a given sequence is answerable while the QA model focuses on both that and where the answer span begins and ends.

Answerability of Question

	no_match	exact_match	partial_match	total	
is_impossib	617	2551	0	3168	52%
is_possibl	653	1989	268	2910	48%
	1270	**4540**	**268**	**6078**	
	21%	75%	4%		

Length of Question

	no_match	exact_match	partial_match	total	
32	80	300	29	409	7%
64	830	3061	178	4069	67%
96	1193	4292	252	5737	94%
more	1270	4540	268	6078	100%

Type of Question

	no_match	exact_match	partial_match	total	
what	765	2802	159	3726	59%
who	146	543	25	714	11%
how	139	430	36	605	10%
when	113	436	12	561	9%
where	66	198	11	275	4%
which	63	202	9	274	4%
why	22	54	16	92	1%
other	14	36	9	59	1%
	1328	**4701**	**277**	**6306**	

	no_match	exact_match	partial_match	total
what	21%	75%	4%	
who	20%	76%	4%	
how	23%	71%	6%	
when	20%	78%	2%	
where	24%	72%	4%	
which	23%	74%	3%	
why	24%	59%	17%	

Figure 5: Error Analysis by Question Type

6 Analysis

Error Analysis: I analyzed model accuracy by `is_impossible[True, False]`, by type of question asked `[what, who, how, when, where, which, why, other]`, and by token length of `[context, question, gold\_answer, predicted\_answer]` to try to find patterns where various models perform better or worse. An example of this analysis is shown in Figure 5 above, which is based on the 190304-04_13 run in which a $\text{BERT}_{\text{large}}$ model with 3-level QA architecture unexpectedly performed worse than the same model with default QA architecture. A few quick observations:

- The split between possible-impossible questions is 48/52 rather than the 67/33 that I expected. I have double-checked these results and believe they are correct. I need to check the distributions in the `train` and `test` sets, and to work these findings back into my model as described above.
- My analysis found that the default `max_query_length` of 64 covered only 67% of the Dev set questions, while setting it to 96 covers 94% and improves performance. The same did not hold `max_seq_length` from 384 to 512.
- The question type [what, who, ...] is not an exact 1:1 correspondence, because some questions have more than one word on the type list in the question, but it is close (within about 5%). Questions types are dominated by 'what' and 'who,' both of which are usually answered with concrete nouns. I am curious how the models perform on 'why' questions, which seem conceptually harder to me, but I'm not going to spend much time here because they make up such a small portion of the question set.

References

Thorsten Brants, Ashok C Popat, Peng Xu, Franz J Och, and Jeffrey Dean. Large language models in machine translation. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, 2007.

Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167. ACM, 2008.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Dan Hendrycks and Kevin Gimpel. Bridging nonlinearities and stochastic regularizers with gaussian error linear units. *arXiv preprint arXiv:1606.08415*, 2016.

Branislav Hollander. Logging in tensorboard with pytorch. *Medium*, Septemer 2018. URL `https://becominghuman.ai/logging-in-tensorboard-with-pytorch-or-any-other-library-c549163dee9e`.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

Andriy Mnih and Geoffrey E Hinton. A scalable hierarchical distributed language model. In *Advances in neural information processing systems*, pages 1081–1088, 2009.

Jeffrey Pennington, Richard Socher, and Christopher Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

Matthew E Peters, Waleed Ammar, Chandra Bhagavatula, and Russell Power. Semi-supervised sequence tagging with bidirectional language models. *arXiv preprint arXiv:1705.00108*, 2017.

Matthew E Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. *arXiv preprint arXiv:1802.05365*, 2018.

Alec Radford, Karthik Narasimhan, Time Salimans, and Ilya Sutskever. Improving language understanding with unsupervised learning. Technical report, Technical report, OpenAI, 2018.

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

Pranav Rajpurkar, Robin Jia, and Percy Liang. Know what you don't know: Unanswerable questions for squad. *arXiv preprint arXiv:1806.03822*, 2018.

Wilson L Taylor. "cloze procedure": A new tool for measuring readability. *Journalism Bulletin*, 30 (4):415–433, 1953.

Joseph Turian, Lev Ratinov, and Yoshua Bengio. Word representations: a simple and general method for semi-supervised learning. In *Proceedings of the 48th annual meeting of the association for computational linguistics*, pages 384–394. Association for Computational Linguistics, 2010.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

Jesse Vig. Deconstructing bert: Distilling 6 patterns from 100 million parameters. *Medium*, December 2018. URL `https://towardsdatascience.com/deconstructing-bert-distilling-6-patterns-from-100-million-parameters-b49113672f77`.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

Yukun Zhu, Ryan Kiros, Rich Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *Proceedings of the IEEE international conference on computer vision*, pages 19–27, 2015.