Improving Numerical Reasoning in Reading Comprehension

Jambay Kinley Raymond Lin

Abstract

With the advent of models that are reaching and exceeding human performance on standard reading comprehension datasets such as SQuAD (Rajpurkar et al., 2016), DROP (Dua et al., 2019) has recently been proposed as a more challenging dataset. The dataset includes questions requiring symbolic discrete reasoning, including arithmetic, counting, and sorting, as well as strong contextual understanding of the text, that are difficult to answer with existing passage span based models. Taking advantage of BERT (Devlin et al., 2018) word embeddings, we propose a new baseline NABERT that exceeds the performance of the NAQANet baseline from the original DROP paper, with 54.67 EM and 57.64 F1 on the dev set. We also describe improvements on NABERT by adding additional "standard" numbers and expression templates to create NABERT+, which further increases performance to 62.75 EM and 65.70 F1 on the dev set. These gains are particularly strong on questions requiring arithmetic.

1. Introduction and Background

Reading comprehension is a problem where given a passage and questions related to that passage, the task is produce the correct answers to those questions. One of the most commonly used reading comprehension datasets in recent years is SQuAD (Rajpurkar et al., 2016). In this dataset, the answers to the questions are spans of text contained within the passages. Numerous attention-based models have been developed that have achieved high performance nearing or exceeding human performance, such as QANet (Yu et al., 2018) and BiDAF (Seo et al., 2017), as well as many models taking advantage of BERT (Devlin et al., 2018) word embeddings.

To provide a more challenging dataset for reading comprehension, DROP (Dua et al., 2019) was recently proposed. The dataset is intended to be difficult through a combination of requiring a model to perform symbolic discrete reasoning (such as arithmetic, counting, and sorting) in addition to being able to have strong contextual understanding of the passage and question. Constructed in an adversarial manner, the dataset only includes crowdsourced questions that could not be properly answered by a BiDAF model. Existing SQuAD-style reading comprehension models that perform well on SQuAD fail to do similarly well on DROP. According to Dua et al. (2019) the best performing model on SQuAD 1.1, QANet, sees its exact match accuracy fall from 72.7% on SQuAD to 25.5% on DROP, while even the highest performing SQuAD-style model on DROP, which uses BERT, sees its exact match accuracy fall from 84.7% on SQuAD to 29.5% on DROP. These drops are primarily due to these SQuAD-style models requiring the answer to be present as a span within the passage text, and lacking the numerical reasoning capabilities to answer the many such questions in DROP.

In this paper, we propose a new baseline for the DROP dataset that takes advantage of BERT, which we call NABERT. We analyze the shortcomings of this baseline, and propose augmentations that improve performance, especially on questions that require arithmetic calculations.

2. Related Work

2.1. NAQANet

To provide a first baseline for the DROP dataset, Dua et al. (2019) propose numerically-aware QANet (NAQANet), which consists of QANet with some augmentations. These augmentations take the form of modules that use the QANet architecture up to the output layer and produce answers using four methods: 1) passage spans, 2) question spans, 3) counting, and 4) arithmetic expressions, as well as a predictor that, for a given question and passage, picks which of these modules to use to produce the answer.

More specifically, NAQANet first uses the QANet architecture to get a question representation $\mathbf{Q}$ and question-aware passage representation $\mathbf{P}$. Self-attention on $\mathbf{Q}$ and $\mathbf{P}$ is used to produce question and passage summary vectors $\mathbf{h}^Q$ and $\mathbf{h}^P$, respectively. The QANet encoder is also applied to P four times to produce $\mathbf{M}_0$, $\mathbf{M}_1$, $\mathbf{M}_2$, and $\mathbf{M}_3$. Number representations $\mathbf{h}_i^N$ are created by indexing into a concatenation of $\mathbf{M}_0$ and $\mathbf{M}_3$. Following the standard output layer of QANet, the passage span module uses $\mathbf{M}_0$, $\mathbf{M}_1$, and $\mathbf{M}_2$ to produce start and end indices for an answer taken from

the passage text. The question span module makes use of $\mathbf{Q}$ and $\mathbf{h}^P$ to get start and end indices for an answer taken from the question text. The counting module uses $\mathbf{h}^P$ to generate a count from 0 to 9 as a categorical variable. The arithmetic module uses $\mathbf{h}_i^N$ to produce a sign in $\{0, +, -\}$ for each number in the passage, and produces as the answer the numbers summed up with their assigned signs. Finally, to choose which answer module to use, the answer choice predictor uses $\mathbf{h}^P$ and $\mathbf{h}^Q$.

We follow the structure of NAQANet fairly closely in our NABERT baseline, with the same four modules and a predictor for which module to use, although we modify how we get the various passage and question representation vectors to use BERT embeddings.

2.2. Math Word Problems

To direct our work on improving arithmetic performance, we turn to the math word problem literature, which involves work to produce a correct numerical answer given an algebraic word problem. This bears resemblance to our attempts to perform numerical reasoning in reading comprehension, as we usually need to evaluate a mathematical expression involving numbers in the passage to arrive at the correct answer for a question requiring numerical reasoning. In Kushman et al. (2014) and Wang et al. (2017), the general approach is to generate equation templates based on the word problem, and map numbers into the templates. Kushman et al. (2014) use the training set to obtain a set of possible templates from which templates are selected and filled at inference time, while Wang et al. (2017) use a RNN to encode the word problem to a context vector and another RNN to decode the context vector to an equation template.

In the cases of Kushman et al. (2014) and Wang et al. (2017), the training datasets they use have some or all of the word problems labeled with the correct mathematical expressions used to solve the problem. We unfortunately do not have this labeling, making expression generation more difficult. However, we can still take advantage of this idea of equation templating to allow our model to use a larger variety of mathematical expressions when we work to improve our numerical reasoning performance.

3. Model and Training

3.1. Problem Setting

In this reading comprehension problem, we have a vocabulary $\mathcal{V}$. For a pair of passage x^P and question x^Q which are both sequences of words in $\mathcal{V}$, we wish to predict the answer y. The answer can be a number or a set of spans which are subsequences of either x^P or x^Q. We want to estimate $p(y; x^P, x^Q)$.

3.2. Model Formulation

Given x^P and x^Q, we treat the distribution of y as a mixture parameterized by x^P, x^Q and model parameters θ where the components correspond to different answer modules.

Formally, for each pair (x^P, x^Q) we introduce a latent variable z such that

$$p(y; x^P, x^Q, \theta) = \sum_{z=1}^{K} p(z; x^P, x^Q, \theta) p(y|z; x^P, x^Q, \theta)$$

where $z \in \{1, \ldots, K\}$ is a categorical random variable and K is the number of answer modules. Each component of the mixture is a neural network such that

$$p(y|z; x^P, x^Q, \theta) = \mathrm{NN}_z(y, x^P, x^Q, \theta)$$

An important point to note is that unlike standard mixture models, an individual component of our mixture assigns non-zero probability to only a subset of the range of y. For instance, a component that predicts answers as spans of the passage will give non-zero probability only to answers that are subsequences of x^P.

3.3. NABERT Baseline

We implement our baseline to be similar to NAQANet (Dua et al., 2019) with the QANet replaced by a pretrained BERT and necessary modifications made to incorporate BERT. It uses four different answer modules on top of BERT to produce four different kinds of answers.

Summary Vectors. Following Devlin et al. (2018)'s approach to SQuAD, we represent x^Q and x^P as a single packed sequence with the `A` embedding for x^Q and the `B` embedding for x^P and obtain the final hidden vectors as tags $\mathbf{T}$ from BERT. The model then uses self-attention to produce contextualized summary vectors $\mathbf{h}^P$ and $\mathbf{h}^Q$ for the passage and the question, respectively.

In more mathematical terms, let $\mathbf{T}^P$ and $\mathbf{T}^Q$ be subsequences of $\mathbf{T}$ that correspond to x^P and x^Q respectively. Let us also define `BERTdim` as the dimension of the tags in $\mathbf{T}$, and have $\mathbf{W}^P \in \mathbb{R}^{\texttt{BERTdim}}$ and $\mathbf{W}^Q \in \mathbb{R}^{\texttt{BERTdim}}$ as learned linear layers. Then, the summary vectors are computed as

$$\boldsymbol{\alpha}^P = \mathrm{softmax}(\mathbf{W}^P \mathbf{T}^P) \qquad \boldsymbol{\alpha}^Q = \mathrm{softmax}(\mathbf{W}^Q \mathbf{T}^Q)$$
$$\mathbf{h}^P = \boldsymbol{\alpha}^P \mathbf{T}^P \qquad \mathbf{h}^Q = \boldsymbol{\alpha}^Q \mathbf{T}^Q$$

Answer module prediction. The answer module is predicted using a categorical variable with the component probabilities computed as

$$\mathbf{p}^{\text{module}} = \mathrm{softmax}(\mathbf{FFN}([\mathbf{h}^P; \mathbf{h}^Q]))$$

where **FNN** is a feedforward neural network.

Passage span. We use the method used by Devlin et al. (2018) for passage span. If we define $\mathbf{W}^S \in \mathbb{R}^{\texttt{BERTdim}}$ and $\mathbf{W}^E \in \mathbb{R}^{\texttt{BERTdim}}$ as learned vectors, the probability of the start and end positions in a passage span are computed as

$$\mathbf{p}^{\text{p\_start}} = \text{softmax}(\mathbf{W}^S \mathbf{T}^P)$$
$$\mathbf{p}^{\text{p\_end}} = \text{softmax}(\mathbf{W}^E \mathbf{T}^P)$$

Question span. The probabilities of the start and end positions in a question span are computed as

$$\mathbf{p}^{\text{q\_start}} = \text{softmax}(\mathbf{FNN}([\mathbf{T}^Q, \mathbf{e}^{|\mathbf{T}^Q|} \otimes \mathbf{h}^P]))$$
$$\mathbf{p}^{\text{q\_end}} = \text{softmax}(\mathbf{FNN}([\mathbf{T}^Q, \mathbf{e}^{|\mathbf{T}^Q|} \otimes \mathbf{h}^P]))$$

where $\mathbf{e}^{|\mathbf{T}^Q|} \otimes \mathbf{h}^P$ repeats $\mathbf{h}^P$ for each component of $\mathbf{T}^Q$

Count. Counting is treated as a multi-class prediction problem with the numbers 0-9 as possible labels. The probabilities for the numbers are computed as

$$\mathbf{p}^{\text{count}} = \text{softmax}(\mathbf{FFN}(\mathbf{h}^P))$$

Arithmetic. The model obtains all of the numbers from the passage and assigns a plus, minus or zero for each number. BERT uses word piece tokenization so some numbers are broken up into multiple tokens. We experiment with two ways to create contextual number representations from the BERT tags (from **T**) of all word piece tokens that make up a number.

If $\mathbf{N}^i$ is the set of tags corresponding to the i^{th} number, we use the following two methods to obtain the number representations $\mathbf{h}_i^N$

1. First tag:
$$\mathbf{h}_i^N = \mathbf{N}_0^i$$
2. Self attention:
$$\boldsymbol{\alpha}_i^N = \text{softmax}(\mathbf{W}^N \mathbf{N}^i)$$
$$\mathbf{h}_i^N = \boldsymbol{\alpha}_i^N \mathbf{N}^i$$

The selection of the sign for each number is a multi-class prediction problem with options $\{0, +, -\}$, so the probabilities for the signs are given by

$$\mathbf{p}_\mathbf{i}^{\text{sign}} = \text{softmax}(\mathbf{FFN}(\mathbf{h}_i^N))$$

3.4. NABERT+

NABERT+ is a modified version of NABERT primarily directed towards improving its arithmetic capabilities.

3.4.1. "Standard" Numbers

In addition to the numbers present in the passage, we add a list $\mathcal{L}$ of "standard" numbers that the model always has access to. For each number $n \in \mathcal{L}$, the model learns an embedding $\mathbf{E}_n \in \mathbb{R}^{\texttt{BERTdim}}$. The arithmetic module uses these embeddings as the contextual representation of the standard numbers.

Let us illustrate this with a brief example. Suppose that in our model $\mathcal{L} = [100, 1]$. Then, we have corresponding embeddings $\mathbf{E}_1, \mathbf{E}_{100}$. If we encounter a passage with numbers $[2, 3, 2, 100]$, we will have contextual representations $[\mathbf{h}_0^N, \ldots, \mathbf{h}_3^N]$ for these numbers. The arithmetic module will assign a sign to each of the numbers in $[100, 1, 2, 3, 2, 100]$, and will use as contextual representations $[E_{100}, E_1, \mathbf{h}_0^N, \ldots, \mathbf{h}_3^N]$ when choosing those signs.

3.4.2. Expression Templates

To make the arithmetic module more expressive while also keeping the search for solutions tractable, we replace the baseline arithmetic module with one that uses templates $\mathcal{T} = \{t_0, \ldots\}$ to generate candidate arithmetic expressions. We consider a set of templates, each one of which has predetermined operations and slots for numbers. Some examples of templates might be $x_0 - x_1$, $(x_0 + x_1) \cdot x_2$, or $x_0 \cdot x_0$.

The module first produces a distribution over templates $\mathcal{T}$. Then, for each slot in a template, it computes the probability of each number being in that slot. Formally, the distribution of the answer in the arithmetic module is a mixture where the latent variable is the template. A mixture component assigns non-zero probability to only those values that can be obtained using the corresponding template and the numbers available.

Arithmetic summary vectors. We generate summary vectors $\mathbf{h}^{PA}$ and $\mathbf{h}^{QA}$ in a similar way as how we get $\mathbf{h}^P$ and $\mathbf{h}^Q$. That is, with new learned vectors $\mathbf{W}^{PA} \in \mathbb{R}^{\texttt{BERTdim}}$ and $\mathbf{W}^{QA} \in \mathbb{R}^{\texttt{BERTdim}}$, we have

$$\boldsymbol{\alpha}^{PA} = \text{softmax}(\mathbf{W}^{PA} \mathbf{T}^P) \quad \boldsymbol{\alpha}^{QA} = \text{softmax}(\mathbf{W}^{QA} \mathbf{T}^Q)$$
$$\mathbf{h}^{PA} = \boldsymbol{\alpha}^{PA} \mathbf{T}^P \quad \mathbf{h}^{QA} = \boldsymbol{\alpha}^{QA} \mathbf{T}^Q$$

Template prediction. The template is predicted as a categorical variable with probabilities computed as

$$\mathbf{p}^{\text{template}} = \text{softmax}(\mathbf{FFN}([\mathbf{h}^{PA}, \mathbf{h}^{QA}]))$$

Template completion. Let template t_j have s_j slots. For instance, if our template t_j were $x_0 \cdot x_1 - x_2$, it would have $s_j = 3$ slots. Let us define **U** to be the concatenation of the list of number embeddings for the standard numbers $\mathbf{E}_n$ and the list of contextual number representations $\mathbf{h}_\mathbf{i}^N$ and set

$$\mathbf{R} = [\mathbf{U}; \mathbf{e}^{|\mathbf{U}|} \otimes \mathbf{h}^{P_A}]$$

For each template t_j, we have learned weights $\mathbf{W}^{A,j} \in \mathbb{R}^{2 \cdot \texttt{BERTdim} \times s_j}$. Then, probabilities of the different numbers being chosen for slot k in template t are given by

$$\mathbf{p}_k^{\text{slot},j} = \text{softmax}(\mathbf{W}_k^{A,j}\mathbf{R})$$

3.5. Training

We train our model by optimizing the likelihood of the answer $p(y; x^P, x^Q)$. As described in Section 3.2, the distribution of y parameterized by x^P, x^Q, θ is a mixture. We enumerate over all answer types and compute $p(y; x^P, x^Q)$.

Our preprocessing method described in Section 4.2.1 gives all possible ways to get the answer using each answer type. Due to the brute force method of obtaining these possibilities, there might be more than one way to get the answer using an answer type. We will consider all of these possibilities correct although some might not really be "correct". For instance, we might have an arithmetic expression that arrives at the correct answer but not using the intended method, like if we have passage "Jambay has 3 apples and 2 oranges. Raymond has 3 apples and 4 bananas." and question "How many apples are there in total?" an expression that arrives at the correct answer but using the incorrect method would be $4+2$, while one that arrives at the correct answer with the correct answer would be $3+3$.

For a datapoint y, x^P, x^Q, let the set of all possible ways to get y be $\boldsymbol{\chi} = \{\boldsymbol{\chi}^{PS}, \boldsymbol{\chi}^{QS}, \boldsymbol{\chi}^{C}, \boldsymbol{\chi}^{A}\}$ where $\boldsymbol{\chi}^{PS}$ is the set of correct passage spans (represented as tuples with the start and end positions), $\boldsymbol{\chi}^{QS}$ is set of correct question spans (also tuples with the start and end positions), $\boldsymbol{\chi}^{C}$ is the correct count value and $\boldsymbol{\chi}^{A}$ is the set of correct arithmetic expressions. Let $\{PS, QS, C, A\}$ be the choice of the passage span, question span, count, and arithmetic modules, respectively, so $z \in \{PS, QS, C, A\}$, and if we chose to use the passage span module, $z = PS$.

With this notation, we can write out

$$p(z; x^P, x^Q, \theta) = \mathbf{p}_z^{\text{module}}$$

and the conditional probabilities $p(y|z; x^P, x^Q, \theta)$.

For the passage span, question span, and count modules, we have

$$p(y|z = PS; x^P, x^Q) = \sum_{(s,e)\in\boldsymbol{\chi}^P} \mathbf{p}_s^{\text{p\_start}} \cdot \mathbf{p}_e^{\text{p\_end}}$$

$$p(y|z = QS; x^P, x^Q) = \sum_{(s,e)\in\boldsymbol{\chi}^Q} \mathbf{p}_s^{\text{q\_start}} \cdot \mathbf{p}_e^{\text{q\_end}}$$

$$p(y|z = C; x^P, x^Q) = \begin{cases} \mathbf{p}_{\boldsymbol{\chi}^C}^{\text{count}} & 0 \le \boldsymbol{\chi}^C \le 9 \\ 0 & \text{otherwise} \end{cases}$$

For the arithmetic module, the conditional probability depends on which model use (NABERT or NABERT+).

NABERT. Let μ be the number of numbers in the passage and $\boldsymbol{\chi}^A$ be the set of correct sign assignments to the numbers (represented as a tuple of sign indices for the numbers). Then, we have

$$p(y|z = A; x^P, x^Q) = \sum_{\text{S}\in\boldsymbol{\chi}^A} \prod_{i=0}^{\mu-1} \mathbf{p}_{i,\text{S}_i}^{\text{sign}}$$

NABERT+. For the model with additional numbers, the conditional probability is the same as that for NABERT with the product now over all standard numbers and the numbers in the passage instead of just over numbers in the passage.

For the model with templates added as well, let $\boldsymbol{\chi}^A = \{\boldsymbol{\chi}^{A,0}, \ldots, \boldsymbol{\chi}^{A,|\mathcal{T}|-1}\}$ where $\boldsymbol{\chi}^{A,j}$ is the set of correct placements of numbers in the template t_j represented as a tuple of the indices of the correct number for each slot. Then, we have

$$p(y|z = A; x^P, x^Q) = \sum_{j=0}^{|\mathcal{T}|-1} \mathbf{p}_j^{\text{template}} \sum_{N\in\boldsymbol{\chi}^{A,j}} \prod_{k=0}^{s_j-1} \mathbf{p}_{k,N_j}^{\text{slot},j}$$

4. Methods

4.1. Dataset

Table 1 contains specs on the DROP (Dua et al., 2019) train and dev sets. Passage and question length as well as vocabulary size are given in terms of number of tokens, which in our case are generated using wordpiece tokenization.

Statistic	Train	Dev
Num passages	5565	582
Num questions	77409	9536
Avg questions / passage	13.91	16.38
Avg answers / question	1.00	3.35
Avg passage len (tokens)	278.39	256.10
Avg question len (tokens)	12.89	13.39
Passage vocab size (tokens)	20986	11102
Question vocab size (tokens)	14051	5794

Table 1. DROP dataset specs.

Table 2 has some breakdowns of the answer types that appear in the train and dev sets. In cases where more than one answer is provided in the dev set in the form of extra validated answers, we use the primary answer to label the answer type.

Answer type	%	
	Train	Dev
Date	1.59	1.65
Number	60.68	61.35
Single-span	31.65	31.06
Multiple-span	6.07	5.95

Table 2. Breakdown of answer types.

4.2. Implementation

We base much of our implementation off of the NAQANet implementation by Dua et al. (2019) which is available from allenai[1]. Our implemetations of our models can be found at `https://github.com/raylin1000/drop_bert`.

4.2.1. Preprocessing

Our models require information not present in the original dataset, which contains only the raw passage and question text, as well as the raw number, date, and span answers.

One thing we need to add is the start and end indices of passage and question span answers. We generate these similar to how it is done by Dua et al. (2019). We tokenize the passage, question, and all answer texts using the bert-base-uncased BERT tokenizer from huggingface[2] and truncate the total length of the passage and question tokens to 512 so we can feed the concatenation of all the tokens into BERT (Devlin et al., 2018). We then go through the passage and question tokens to find matching answer text tokens.

We also need the numbers in the passage as well as the indices of those numbers. While in Dua et al. (2019) NAQANet uses space-based tokenization, the wordpiece tokenization we need to use for BERT (Devlin et al., 2018) sometimes breaks up a number into multiple tokens. Therefore, we identify numbers in the passage along with the indices of all tokens that make up that number.

Finally, we need information on how the numbers in the passage (and any additional standard numbers) can be used to get a numerical answer. For the baseline, this takes the form of sign assignments to all the numbers so that when they are summed up with the signs, they produce the right numerical answer. We generate these following Dua et al. (2019), where we enumerate over all pairs of numbers and all sign assignments to find ones that produce correct answers. For our template augmented models, we again brute force, enumerate over our templates, as well as all possible assignments without replacement of numbers to slots in the templates, to get ones that return the right answer.

[1] `https://github.com/allenai/allennlp`

[2] `https://github.com/huggingface/pytorch-pretrained-BERT`

4.2.2. Model

We use a pretrained BERT model (the bert-base-uncased model from huggingface [2]) that we finetune when training. For standard numbers, we experiment with adding 100. As for templates, we consider the forms $(x_0 + x_1) \cdot x_2, (x_0 - x_1) \cdot x_2, (x_0 + x_1)/x_2, (x_0 - x_1)/x_2, x_0 \cdot x_1/x_2$ and use standard number 1 in addition to any other standard numbers to capture templates such as $x_0 + x_1, x_0 - x_1$. We consider exact match (EM) and F1 scores, where F1 is the harmonic mean of precision and recall, when evaluating models.

4.2.3. Training

For training all our models, we use the BERT Adam optimizer provided from huggingface [2] with default settings and learning rate $1e{-}5$. We use batch size of 4 and train for 10 epochs, using F1 score as our validation metric and taking the model with with the best F1 score on the dev set.

5. Results

5.1. NABERT Baseline

The EM and F1 scores on the dev set from different number representations are in the Table 3. Our best baseline performance was from using the first tag number representation, achieving scores of **54.57 EM** and **57.64 F1**, improving on the 46.20 EM and 49.24 F1 scores that Dua et al. (2019) report for NAQANet.

Representation	EM	F1
First tag	54.67	57.64
Self-attention	53.78	56.87

Table 3. NABERT with different number representations.

Table 4 contains EM and F1 on the dev set split by answer type for NAQANet and NABERT. The NAQANet results are from our own trained version of NAQANet. We used the model implementation and training parameters provided by allenai[1]. Gains are made across all answer types, with the most significant improvement on numerical answer types.

Answer type	Model			
	NAQANet		NABERT	
	EM	F1	EM	F1
Date	33.76	42.26	37.58	46.38
Number	44.72	44.73	54.27	54.29
Single-span	59.18	64.99	65.56	70.62
Multi-span	5.29	23.57	6.53	27.48
All	46.69	49.72	54.67	57.64

Table 4. Comparison of NAQANet and NABERT performance.

To further analyze the performance of NABERT, we break down the answer module used according to answer type in the dev set in Table 5. We also find break down the performance based on answer module selected and answer type in the dev set in Table 6.

From Table 5 see that the model appears to make good choices of answer module. When the answer was a number, the model tended to almost always select the count and arithmetic modules, and when the answer was a span, the model overwhelmingly selected the passage and question span modules. We see in Tables 4 and 6 that NABERT performs by far the worst on date and multi-span types, likely due to the model not explicitly handling dates and being unable to generate more than one span as an answer. However, as we see in Table 2, these answer types are a fairly small subset of the data. Of the remaining answer types, the model performs significantly worse on numerical answers than on single-span answers. From Table 2, much of the dataset has numerical answers, so we feel it is worth attempting to improve the poor performance on numerical answers. As we can see in Table 6, the question span has the strongest performance and the arithmetic module performs the worst. From Table 5, the arithmetic module is used most frequently chosen on numerical answers, and in Table 6, the performance using that module is poor. Given that we wish to improve numerical answer performance, the arithmetic module is used for computing most of the numerical answers, and the arithmetic module performs poorly, we are motivated, when creating NABERT+, to focus on improving the arithmetic module.

5.2. NABERT+

The EM and F1 scores on the dev set from varying number representation and components of NABERT+ used (including standard numbers only and standard numbers + expresison templates) are in the Table 7. Our best performance was from using the first tag number representation with just standard numbers, which had **62.75 EM** and **65.70 F1**. The increased performance of the standard only models may be because we trained those models for longer (we were unable to complete full training due to time constraints), so the ones with standard number and templates may perform better in the long run. The performance of all NABERT+ models is a significant improvement over the NABERT baseline performance.

Representation	**Additions**	**EM**	**F1**
First tag	Standard	62.75	65.70
Self-attention	Standard	62.65	65.58
First tag	Standard + template	62.19	65.24
Self-attention	Standard + template	61.43	64.39

Table 7. NABERT with different number representations and components added

To further analyze the performance of the versions of NABERT+, we break down the answer module used according to answer type in the dev set in Table 8. We also find break down the performance based on answer module selected and answer type in the dev set in Tables 9 and 10. Here, the models are the best one for each version of NABERT+, so the best standard number only version and the best standard number + expression template version.

Answer type	**Answer module**			
	Passage span	Question span	Count	Arithmetic
Date	15.29	12.10	0.00	72.61
Number	0.14	0.02	22.21	77.64
Single-span	66.31	31.77	0.34	1.59
Multiple-span	92.24	7.05	0.00	0.71
All	26.42	10.5	13.73	49.36

Table 5. Answer module selected (%) broken down by answer type for NABERT.

Answer type	**Answer module**									
	Passage span		Question span		Count		Arithmetic		All	
	EM	F1	EM	F1	EM	F1	EM	F1	EM	F1
Date	29.17	35.96	57.89	60.53	-	-	35.96	46.22	37.58	46.38
Number	50.00	58.38	0.00	0.00	58.97	58.97	52.95	52.96	54.27	54.29
Single-span	64.31	69.05	68.44	74.39	30.00	30.00	68.09	69.51	65.56	70.62
Multiple-span	4.59	25.74	32.50	52.85	-	-	0.00	2.00	6.53	27.48
All	51.53	59.71	66.73	73.19	58.75	58.75	52.64	52.91	54.67	57.64

Table 6. Model performance broken down by answer module selected and answer type for NABERT.

Answer type	NABERT+ model							
	Standard only				Standard + templates			
	Passage span	Question span	Count	Arithmetic	Passage span	Question span	Count	Arithmetic
Date	18.47	11.46	0.0	70.06	49.68	23.57	0.00	26.75
Number	0.17	0.05	22.09	77.69	7.49	0.02	21.90	70.60
Single-span	69.35	28.83	0.24	1.59	67.59	31.47	0.3	0.64
Multiple-span	92.59	6.53	0.0	0.88	94.71	4.59	0.0	0.71
All	27.45	9.56	13.62	49.36	32.04	10.44	13.53	43.99

Table 8. Answer module selected (%) broken down by answer type for NABERT+ (standard numbers only and standard + templates)

Answer type	Answer module									
	Passage span		Question span		Count		Arithmetic		All	
	EM	F1	EM	F1	EM	F1	EM	F1	EM	F1
Date	27.59	32.24	50.00	50.00	-	-	31.82	39.25	33.12	39.19
Number	30.0	42.9	0.00	0.00	59.75	59.75	70.21	70.21	67.79	67.82
Single-span	64.41	69.66	66.74	72.76	57.14	57.14	68.09	69.51	65.12	70.52
Multiple-span	5.14	25.09	24.32	39.46	-	-	20.00	23.4	6.53	26.01
All	51.99	60.20	64.47	70.72	59.74	59.74	69.24	69.43	62.75	65.70

Table 9. Model performance broken down by answer module selected and answer type for NABERT+ (standard numbers only).

Answer type	Answer module									
	Passage span		Question span		Count		Arithmetic		All	
	EM	F1	EM	F1	EM	F1	EM	F1	EM	F1
Date	34.62	50.72	62.16	66.22	-	-	23.81	23.81	38.22	47.17
Number	83.79	84.29	0.00	0.00	53.16	53.16	69.01	69.03	66.63	66.69
Single-span	61.89	67.56	73.18	77.86	55.56	55.56	36.84	36.84	65.26	70.57
Multiple-span	5.03	26.14	46.15	59.35	-	-	0.00	0.00	6.88	27.48
All	54.34	62.24	71.99	76.87	53.18	53.18	68.34	68.37	62.19	65.24

Table 10. Model performance broken down by answer module selected and answer type for NABERT+ (standard numbers + templates)

From Table 8 we see that just like NABERT, both versions of NABERT+ appear to make good choices of answer module. Between the two NABERT+ versions, module choices are quite similar, except that for date answers in the standard number only model, the module much more strongly favors the arithmetic module. We see in Tables 9 and 10 that NABERT+ performs by far the worst on date and multi-span types likely for the same reasons as NABERT. However, we see that significant gains have been made on numerical answers compared to NABERT, as well as on the performance of the arithmetic module.

6. Discussion

We built NABERT to serve as a baseline and gauge the potential of BERT for this task. As can be seen in Table 4, we found that NABERT does significantly better than NAQANET on all answer types. Numerical answers make up a significant proportion of the dataset at around 60% while NABERT only achieves 54.27% EM on this subset so we chose to focus on improving the numerical ability of NABERT.

Upon adding 100 as a "standard" number, we achieved significant boost in the performance of our model on numbers with an absolute improvement of 12.8% EM on the number subset and 8.37% EM overall. An analysis done by Dua et al. (2019) on the prefixes of questions with number type answers found that a sizable proportion of the questions had the forms "how many percent of ...", "how many percent were ...", "how many percent are ...", "how many in percent ...". We expected to observe some improvement in the numerical performance of our model but the large improvement achieved by the addition of 100 to NABERT suggests that a good portion of the questions have solutions of the form $100 - n$ where n is a number from the passage. This was confirmed by a cursory examinations of a sample of questions, many of which were asking for the complement of a percentage.

NABERT+ with "standard" numbers and templates has performance comparable to that of NABERT+ with "standard" numbers but it learns faster reaching the reported performance in 5 epochs compared to NABERT+ with "standard" numbers that takes 8 epochs. The faster training could be due to the inductive bias introduced by the structure of the templates. We also observe that NABERT+ with "standard" numbers and templates chooses the passage span module more frequently. This is because some numerical answers that are just spans cannot be answered using a template that requires more than one number. We think this model might beat our current best model with further training.

The decisions to add "standard" numbers and use templates although reasonable in the context of the DROP dataset are ad-hoc and not good for generalization to other domains of passages and questions. To this extent, we attempted to build a more general arithmetic module that dynamically generates expressions using a seq2seq model similar to that in Wang et al. (2017) with modifications made to use BERT embeddings. However, the space of expressions grows combinatorially large with increasing number of numbers and operations. Even after restricting the search space to 3 numbers and 2 arithmetic operations, our brute force search gave too many "correct" expressions which restricted us from using techniques like teacher forcing while training the model. We were thus unable to make the model learn to generate expressions. We think that given better supervision through validated correct answers, such models would work well since we see significant improvements in different tasks when using pretrained BERT.

Our focus in this project has been on the arithmetic module but there are many other areas of potential improvements. Some of the questions with number answer types require specific knowledge about football such as point values of touchdowns and field goals, and format of season records. Answering such questions might require models that incorporate domain knowledge. Questions that require counting and comparison are also of particular interest. Given the decent performance of our simple count module, we see promise in the potential improvements from more sophisticated modules.

The different modules in NABERT and NABERT+ share the final outputs of a fine-tuned pretrained BERT. The relatively simple architectures of the modules and their reasonable performance raise some interesting questions : Can a single pretrained language model like BERT be finetuned to be used for different tasks? How sophisticated do the modules need to be? There has been recent work like Liu et al. (2019) towards answering these questions and our work is a small step in this direction.

7. Conclusion

We took advantage of BERT word embeddings (Devlin et al., 2018) to build a new baseline model, NABERT, for the DROP dataset (Dua et al., 2019), by adapting the architecture of NAQANet, the existing DROP baseline, for use with BERT. We also make the further augmentations of using "standard" numbers and expression templates to create NABERT+, a version of NABERT with improve performance on questions with numerical answers.

NABERT improves substantially on the performance of NAQANet, managing to reach performance of 54.67 EM and 57.64 F1. NABERT+ makes further gains, raising the performance to 62.75 EM and 65.70 F1. The improvement appears to be driven by better performance of the arithmetic module.

We believe that to further improve numerical performance, the addition of validated correct expressions for numerical questions in the dataset would allow us to build more a more general arithmetic module capable of generating general expressions, and not limited to just a small set of templates.

NABERT+ pushes the baseline for DROP and opens up multiple avenues for future work. For future models to do better, they might need to perform more sophisticated reasoning and require domain knowledge.

References

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

Dua, Dheeru, Wang, Yizhong, Dasigi, Pradeep, Stanovsky, Gabriel, Singh, Sameer, and Gardner, Matt. DROP: A reading comprehension benchmark requiring discrete reasoning over paragraphs. In *NAACL*, 2019.

Kushman, Nate, Artzi, Yoav, Zettlemoyer, Luke, and Barzilay, Regina. Learning to automatically solve algebra word problems. In *ACL*, 2014.

Liu, Xiaodong, He, Pengcheng, Chen, Weizhu, and Gao, Jianfeng. Multi-task deep neural networks for natural language understanding. In *arXiv preprint arXiv:1901.11504*, 2019.

Rajpurkar, Pranav, Zhang, Jian, Lopyrev, Konstantin, and Liang, Percy. SQuAD: 100,000+ questions for machine comprehension of text. In *EMNLP*, 2016.

Seo, Min Joon, Kembhavi, Aniruddha, Farhadi, Ali, and Hajishirzi, Hannaneh. Bidirectional attention flow for machine comprehension. In *ICLR*, 2017.

Wang, Yan, Liu, Xiaojiang, and Shi, Shuming. Deep neural solver for math word problems. In *EMNLP*, 2017.

Yu, Adams Wei, Dohan, David, Luong, Minh-Thang, Zhao, Rui, Chen, Kai, Norouzi, Mohammad, and Le, Quoc V. QANet: Combining local convolution with global self-attention for reading comprehension. In *ICLR*, 2018.