Question Answering with DistilBert

Natural Language Processing: Final Project

F. Cichetti[*], M. Ceresini[†], D. Angelani[‡], L. Dans[§]

February 8, 2022

Contents

[*]federico.cichetti@studio.unibo.it

[†]marcello.ceresini@studio.unibo.it

[‡]davide.angelani@studio.unibo.it

[§]lucas.dans@studio.unibo.it

Summary

In this report, we give an overview of the Question Answering problem as presented on the Stanford Question Answering Dataset (SQuAD) and describe a neural architecture that solves it with good results, as well as showcasing different experiments that validate our choices and suggest potential future improvements.

Our proposed neural solution has at its core DistilBert, which is a lightweight but still very powerful version of Bert, itself a very flexible and well-known *transformer*-based language model. Bert can be extended to solve the Question Answering problem just by adding a few layers on top of it, but we discovered that aggregating some of the intermediate results before the final layers can be beneficial.

We provide an in-depth analysis of the SQuAD dataset and describe in details how we used the available data for our training, validation and testing experiments. Our choices for dataset pre-processing are validated by empirical evidence and some of them can be fine-tuned depending on the available hardware's capabilities. Furthermore, we explored different ways for loading the dataset in memory, which allow a trade-off between data retrieval time and memory occupation, allowing us to train the entire model with all the types of hardware at our disposal.

We also implemented some baseline algorithms which give an idea of how difficult the problem of Question Answering really is for ordinary non-learning techniques.

Motivated by some recently published articles, we also tried to enhance our initial model in different ways. Firstly, we studied the quality of the representation of semantic information at each level of the transformer, noticing interesting results caused by the nature of BERT pre-training tasks. We propose a Named Entity Recognition module, which tries to directly improve the representation level in the worst-performing layers by giving more weight to Named Entities tokens. Another enhancement we explored is that of using a transformer with higher representation capabilities, like the original Bert, and ensembling different transformer models with heterogeneous processing power.

All models were trained on cloud TPUs and the results we have obtained are promising: with our best method, we have reached an Exact Match precision of 64.33% and a Macro-Weighted F1 Score of 78.59%, which is lower than human performance but it's still impressive. Furthermore, a thorough error analysis shows that the model still produces humanly acceptable answers even when not in line with the provided ground truth.

Having said that, our model also has some weaknesses and limitations, which we discuss in-depth but are generally related to a weak understanding of numbers and of Named Entities. Our future work will be related to improving these aspects.

The code and instructions for downloading the weights for this report are available at the following link: https://github.com/MarcelloCeresini/QuestionAnswering.

1 Introduction

Question answering systems are designed to *reply* to *questions* posed in *natural language*, usually by extracting the answer from a *knowledge base* or some other kind of *context*. In this project we focus on a particular subset of questions: *factoid questions*, whose answer can be given by *citing* or extracting a span of text from a larger given body, like a book or an article.

Basically, such an answer is made of a subset of consecutive words whose start and end words are found in the body by some automatic process. Formally, the task can be defined like this: given a tuple (Q, C) consisting of two sequences of tokens extracted from sentences, a *question* $Q = [q_1, \ldots, q_t]$ and a related *context paragraph* $C = [c_1, \ldots, c_n]$, the goal of the problem is to return the *answer span* $[c_s, c_{s+1}, \ldots, c_{e-1}, c_e]$, with $0 \leq s \leq e \leq n$ that answers the question in the most precise and direct way.

2 Stanford Question Answering Data set (SQuAD)

Historically, large and realistic datasets have played a critical role for driving fields forward, like ImageNet for object recognition. Existing datasets for Question Answering experience a considerable trade off between data quality and size. Those that are high in quality are usually too small [1], while those that are large enough to train robust ML models are generated via automation or are not posing the problem as an explicit reading-comprehension task. To address the need for a large and high-quality reading-comprehension dataset, the Stanford Question Answering Dataset (*SQuAD*) [2] was presented in 2016.

2.1 Dataset Analysis

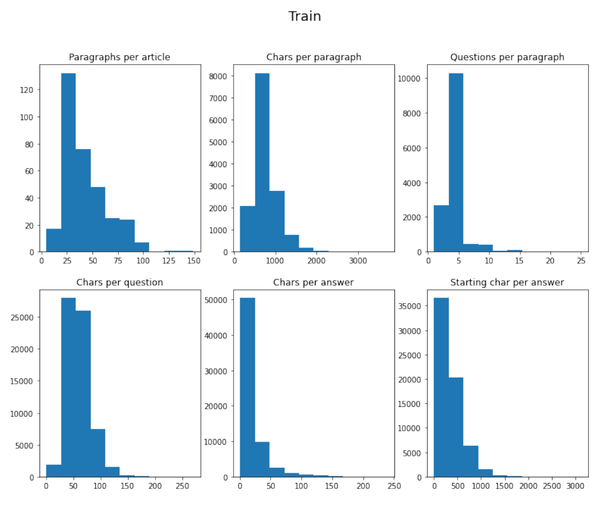
The dataset consists of a large set of questions posed on a set of Wikipedia articles by a team of crowdworkers, each with one or more related answers extracted directly from the article's text. SQuAD contains in total $107,785$ question-answer pairs on 536 different articles.

The articles were originally distributed across dataset splits so that the original *training set* contains 442 articles, the validation set contains 48 articles and the test set has the remaining 46. Unfortunately, the original test set is not public, therefore we randomly split the original training set into a smaller training set (331 articles) and a validation set (111 articles), while we used the original validation set as test set. Splitting by articles ensures that no split contains the same paragraphs or questions as another. We assured that the three splits had overall the same properties, as presented in Figure 1. An important thing to note is that the test set has multiple answers for each questions, while the others don't: on average there are 3.29 answers per question, with a maximum of 6 and a minimum of 1 over the 10,570 questions in that split.

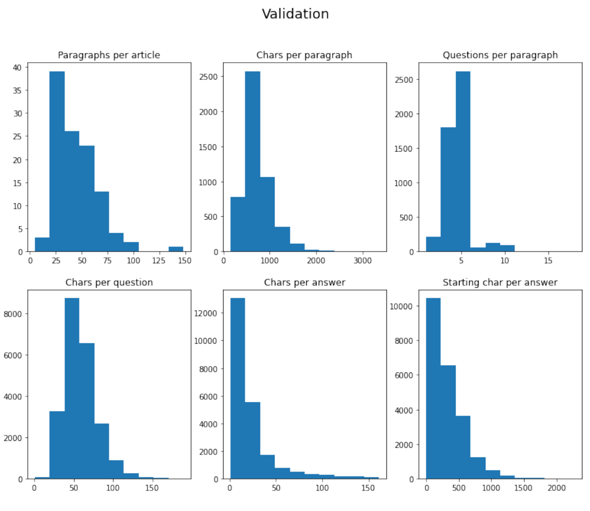
Who		What		Why		How	
by the	114	of the	448	due to	25	more than	78
of the	57	such as	316	of the	16	there are	61
led by	47	known as	230	because of	16	there were	56
such as	42	as the	229	to the	11	of the	46
designed by	24	in the	201	in order	8	total of	33
When		**Where**		**Which**		**Others**	
in the	183	in the	134	of the	67	in the	380
founded in	41	at the	83	in the	50	of the	338
during the	29	to the	28	such as	47	to the	108
of the	27	located in	27	by the	35	such as	97
at the	21	from the	16	to the	32	from the	80

Table 1: Most common groups of words for each category of question (counts on the right)

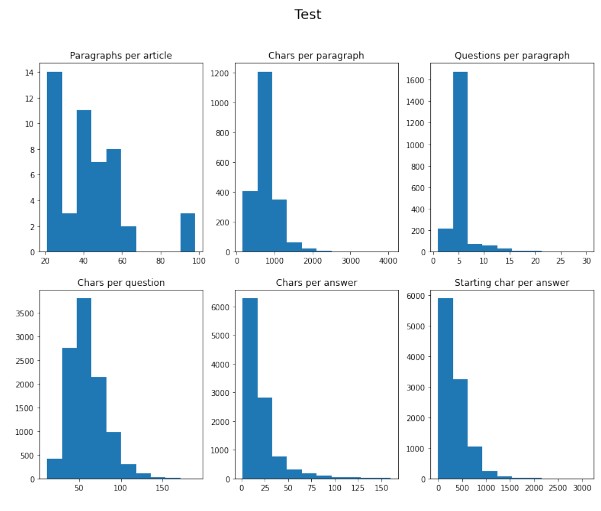
We also analyzed whether we could incorporate linguistic knowledge based on the *type* of a question into the answer-searching process. A question can be assigned a type by looking at whether it contains one



(a) Train split properties



(b) Val split properties



(c) Test split properties

Figure 1: Dataset splits properties

of the most common interrogative words: `("Who", "What", "Why", "How", "When", "Where", "Whose", "Which")`. We split the training set by these types, then for each split we extract which are the previous two words of the answer in the context. Table 1 shows the first 5 words for each category.

Since these words indeed reflect the kind of question that is being asked, we think that this analysis can potentially be adapted into a baseline algorithm for finding the starting point of an answer.

3 Bert and DistilBert

The heart of our model is the popular transformer-based language representation model *Bert*, presented in [3]. In 2018, Bert was the state of the art model on many natural language processing tasks and its architecture has been further improved and refined over the years.

In particular, since reaching high accuracy was not our main focus, we used a variant of Bert called *DistilBert* [4] which, according to the paper, reduces the size of Bert by 40% while retaining 97% of its language understanding capabilities and being 60% faster. Indeed, this architectural change allowed us to cut down training and testing times by more than half.

3.1 Question Answering with Bert

Both Bert and DistilBert are very *flexible* architectures that, once pre-trained (usually on the tasks of Masked Language Modeling (MLM) and Next Sentence Prediction (NSP)), can be adapted and fine-tuned on any task of interest.

While both BERT and DistilBert can receive as input two separate sentences (through a special `[SEP]` token in the middle), only BERT requires another vector to indicate to which sentence a given token belongs. DistilBert encompasses this step into the model, removing the need for additional inputs.

In order to adapt the model to QA, we face the problem to design a way in which we can transform the network's output into an answer for the given question. The transformer outputs *token embeddings* (denoted as $(T_0, T_1 \ldots, T_n)$, where n is the length of the sequence), which are low dimensional but high-level semantic representations of each token. The solution proposed by Bert paper is to compute dot products between each of these embeddings and two learnt vectors with the same dimensionality: a start vector S and an end vector E. The results are *start and end scores* for each token that with a softmax become probability distributions for each token to be the starting element and the ending element of the answer in the paragraph.

Once we have the probabilities, the sequence is scanned with a simple algorithm that returns the span with *maximum score*, where the score of each span $[T_i, T_{i+1} \ldots, T_j]$ is $S \cdot T_i + E \cdot T_j$, where only spans where the start is before the end (i.e. $i > j$) are considered.

3.2 DistilBert

As mentioned before, we actually used DistilBert in our model for efficiency. DistilBert has the same general architecture as Bert, but the number of layers is halved (6 instead of 12 in our implementation) and the *segment embedding* is not present.

The model was trained using a technique called *distillation* (also known as *teacher-student training*), in which a *small model* (DistilBert) is trained to reproduce the behaviour of a *larger model* (Bert). In particular, the student model learns to also mimic the *prediction uncertainty* of the teacher model, with the principle that learning from uncertainty will allow the student model to generalize in the same way as the teacher.

In practice, this means that rather than using a one-hot encoded ground truth, the student uses the teacher's prediction as a feedback signal. This is a vector of *soft* probabilities, therefore as (part of the) training loss, the Kullback-Leibler divergence between the two probability distributions is minimized rather than the usual Cross-Entropy:

$$KL(p||q) = E_p \log(\frac{p}{q}) = \sum_i p_i \times \log(p_i) - \sum_i p_i \times \log(q_i)$$

4 Implementation

In this section we present in details the tools and techniques we used to implement our main model and describe our experiments.

4.1 Dataset pre-processing

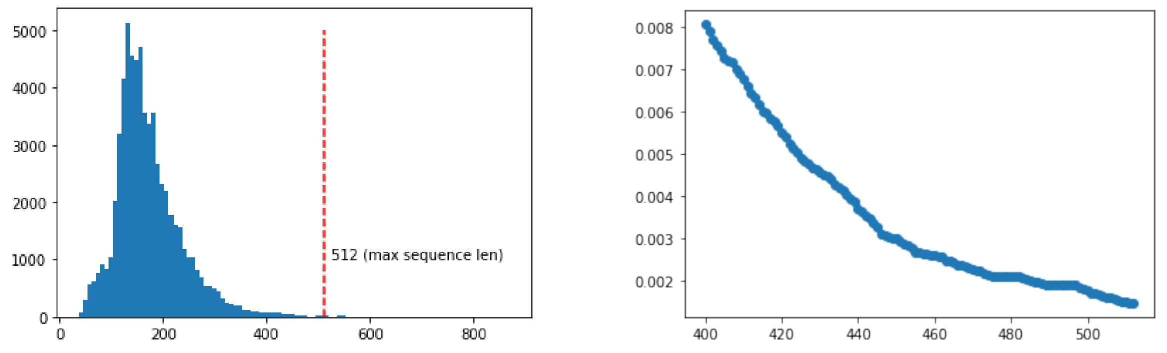
The questions and paragraphs have to be transformed from *raw text* to sequences of *tokens* through a *tokenizer*. Bert and DistilBert's tokenizers are based on *WordPiece*, which is an algorithm to automatically build a dictionary given a large corpus of text. WordPiece adds to the dictionary every Unicode character, then adds the most frequent groups of 2-grams, then 3-grams and so on, until a maximum vocabulary size is reached. In this way, if a word is not present in the dictionary it gets iteratively decomposed into sub-words until either all sub-words are in the dictionary or the word has become a sequence of characters. This implies that we have no Out Of Vocabulary (OOV) words in the dataset.

The tokenizer we used was pre-trained on BookCorpus [5] and English Wikipedia and was distributed by HuggingFace together with the *transformers* library that we used for implementing the rest of the model

[6]. Since the source of text in our dataset are also English Wikipedia articles, we did not deem necessary to fine-tune the tokenizer on SQuAD. Furthermore, we applied no text pre-processing on the dataset, since all text had already been cleaned of non-Unicode characters, HTML tags and such.

The maximum number of tokens that the model can accept as input is set to 512. The input is formed by the `[CLS]` token, followed by the tokenized question, the `[SEP]` token, the tokenized context paragraph and additionally some padding tokens to reach 512. Longer sequences are automatically truncated, but in the official training set we only found 95 pairs (0.001% of the total pairs) that exceed this limit (Figure 2a).

Although we decided to settle for 512 max tokens, we analyzed that the trade-off between maximum sequence length (and thus, memory requirements) and number of truncated sequences could be further improved: in particular, reducing the maximum input size from 512 to 400 would only increase the truncated sequences by 0.006% (Figure 2b). This could be exploited in edge applications where memory allocation and inference time is essential, but since this is not our case, we decided to keep 512 and maximize performance.



(a) Histogram of sequence lengths distribution. (b) Truncated seq. % over maximum seq. length.

Figure 2: Sequence length analysis.

Since we implemented the model using Keras, we could take advantage of TensorFlow's `Dataset` class, a high level utility class to feed data to neural models. We implemented 2 different dataset loading logics: one that builds the whole `Dataset` in RAM, which allows for faster loading during execution but requires a long pre-processing phase and more memory, and one that utilizes a `generator` to create data on-the-fly whenever it's requested, which has minimal impact on RAM but it's very slow at producing data.

Whatever loading type is used, the data is organized into batches of tuples, each containing *features* (the 512-d (padded) tensor of token IDs and the 512-d attention mask), an optional tensor of *question IDs* that identifies the questions in the dataset (used for evaluation) and *labels*, which are one-hot encoded tensors that map the start and end of ground truth answers to the tokens produced by the tokenizer.

4.2 Model implementation

The model and all training/testing operations have been implemented using TensorFlow 2 and Keras. Additionally, we used HuggingFace's *transformers* library [6], which is one of the most well-known implementations for transformer-based neural models. Apart from being an extremely well-documented library, it has a rich API that provides pre-trained models that can be fine-tuned on Question Answering just by adding a few layers on top. We implemented the following pipeline:

1. DistilBert receives as inputs the *tokens IDs* of the sequence that was pre-processed by the tokenizer and the *attention masks*, which indicates which tokens are eligible for attention and which aren't (i.e. 0 for padding and special tokens). These are (batches of) 512-d vectors.
2. DistilBert has 6 hidden layers, each of which outputs a *hidden state*, which is a 768-d vector that represents the self-attended representation of each token at that processing level.
3. Then, we decide how to combine these hidden states to form a *final representation* of the tokens. We conducted several experiments (presented in section 4.4.1) which suggest that just taking the last of

these hidden states may not be the best choice. Our flagship model (that we refer to as *normal model*) concatenates the hidden states of layer 3, 4, 5 and 6, so to obtain a 3072-d vector.

4. Once we have a final representation for each token, the next step should be computing two dot products between them and the learnt *start vector* and *end vector*. These vectors are implemented as weights of two distinct *fully connected layers* with a *single output*. Indeed a FC layer with no activation function is simply a dot product with a matrix with shape $dim\_input \times dim\_output$, so in case of the *normal* model, we would learn a pair of 3072×1 vectors that have exactly the same role of the start and end vectors.
5. We flatten this sequence of scores and apply a *softmax* function to obtain probabilities for both the starting and ending tokens in the sequence.

Figure 3 shows the architecture of the normal model.

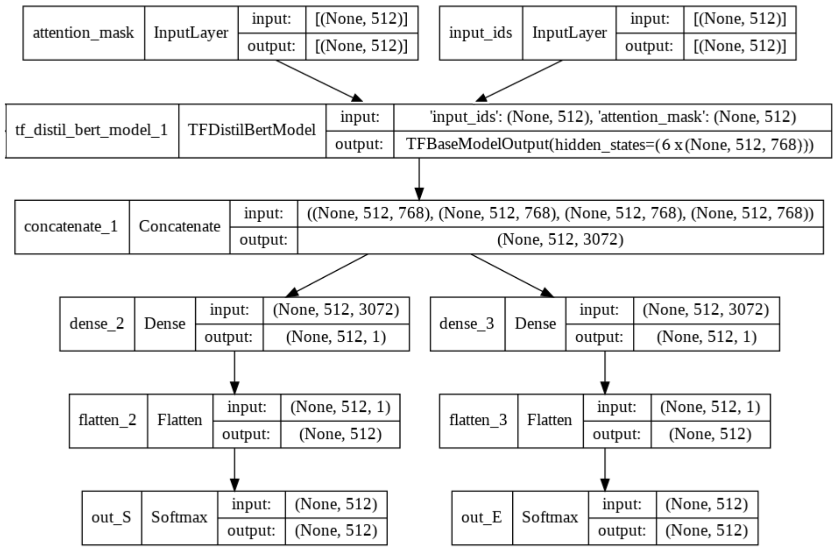

attention_mask | InputLayer | input: [(None, 512)] | output: [(None, 512)]
input_ids | InputLayer | input: [(None, 512)] | output: [(None, 512)]
tf_distil_bert_model_1 | TFDistilBertModel | input: 'input_ids': (None, 512), 'attention_mask': (None, 512) | output: TFBaseModelOutput(hidden_states=(6 x (None, 512, 768)))
concatenate_1 | Concatenate | input: ((None, 512, 768), (None, 512, 768), (None, 512, 768), (None, 512, 768)) | output: (None, 512, 3072)
dense_2 | Dense | input: (None, 512, 3072) | output: (None, 512, 1)
dense_3 | Dense | input: (None, 512, 3072) | output: (None, 512, 1)
flatten_2 | Flatten | input: (None, 512, 1) | output: (None, 512)
flatten_3 | Flatten | input: (None, 512, 1) | output: (None, 512)
out_S | Softmax | input: (None, 512) | output: (None, 512)
out_E | Softmax | input: (None, 512) | output: (None, 512)

Figure 3: The architecture of the "normal" model.

4.3 Training

We trained all models using cloud TPUs available for free on Google Colab. TPUs were chosen over GPUs because they provide a huge gain in training time, with only the minor drawback of not being able to use callbacks that save intermediate results and TensorBoard logs locally.

We also checked whether training the model on GPU could produce different results, but no significant difference was noticed.

We trained all models for a maximum of 100 epochs, setting an Early Stopping callback to stop the training after the validation loss had not improved for 3 epochs. Most trainings were stopped around epochs 10-15, with some exceptions. Using TPUs, each epoch took about 2 minutes.

We used an Adam optimizer with an initial learning rate of $3e^{-6}$ and a batch size of 64. On GPUs, the batch size was reduced to 8 or 16 due to memory limitations. We saved the best weights for each model and used them for the evaluations presented in table 3.

4.4 Experiments and Variants

Starting from the initial architecture described above, we studied how we could further improve our results with additional modules and architectural changes and made some experiments to validate these ideas.

4.4.1 Separate Layers

A good final representation is paramount to achieve a better classification, so we investigated what could be the best way to use the transformer's hidden states. We froze the transformer's weights and trained 6 different models called *Layer i*, with $i \in 1, \ldots, 6$, where the final hidden state was directly the 768-d output of hidden layer i of the transformer, rather than an aggregation. The results are shown in Table 2.

The main thing to notice is that the more an embedding is processed through additional layers, the semantically richer its representation becomes, thus allowing its weights to better understand and answer questions. The poor results obtained when classifying the starting and ending token directly from the hidden states at layers 1 and 2 tell us that those hidden states are in general not able to have an understanding of the text that is general enough for question answering. For this reason, in our *normal* model, we only concatenated layers 3 to 6, which offer a richer semantic value.

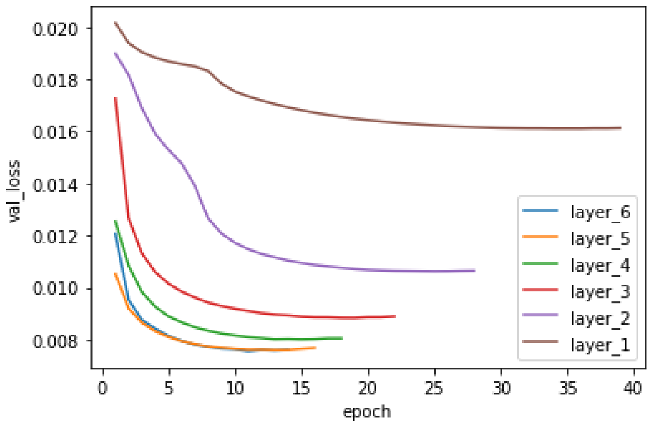


Figure 4: Losses of the separate layers training

Method	Split	Exact Match	Macro F1
Layer 6	val	48.32%	68.39%
	test	60.75%	75.15%
Layer 5	val	47.84%	67.92%
	test	60.24%	74.58%
Layer 4	val	46.79%	66.21%
	test	57.60%	72.40%
Layer 3	val	41.54%	61.10%
	test	51.82%	66.89%
Layer 2	val	33.13%	51.58%
	test	39.72%	55.38%
Layer 1	val	9.50%	20.44%
	test	10.49%	20.90%

Table 2: Results on the separate layers models

It's also very interesting to notice that less epochs are needed to train deeper layers, probably because their weights have better representation of the sentence and require less fine tuning. Moreover, it can be noticed that layer 6 starts with higher loss than layer 5. The weights of the final layer, in fact, are not mainly used to summarize tokens in their context, but for predictions on the pre-training tasks on which the model is trained, MLM and NSP. This means that they are, in a way, too specific to be directly used as they are for other tasks, such as QA. Nevertheless, they are so semantically rich that with just a few epochs they reach the best result.

4.4.2 Ensemble

Similarly to [7], we tried to ensemble our classic *DistilBert* model with a model using *Bert*. We pass the question and paragraph to both models, obtaining for each their output start and end probability distribution. Then, we add up the probabilities and finally we output the prediction with the highest aggregated probability.

Ensembling provides a better accuracy with respect to just using DistilBert, but only using Bert actually seems to be even better and, of course, more performing. We believe that ensembling using a larger pool of simpler models might actually prove to be more beneficial.

4.4.3 Named Entity Recognition module

Following the ideas in [8] we tried to incorporate prior knowledge into our model to improve its performance. The idea is not to create a new task with an augmented dataset and make use of fine tuning, but to directly inject knowledge inside the layers of the model, specifically inside the multi-head self attention layer. One of the main benefits of this method is the avoidance of additional training time, neither through more data nor more tasks during the training process.

In [8] the final task for optimization is Semantic Textual Matching, so the augmentation is focused on word similarity. Instead, during our initial data analysis we discovered, as could be intuitively hypothesized, that many question and answers involve Named Entities (NE). This brought us to the idea that pushing the model to focus more of its attention towards NE would actually improve its performance on QA.

Firstly, we used SpaCy [9], an useful library for NLP which also incorporates Named Entities Recognition, to extract the NE in the sequence of tokens. We aligned this information with the tokenization provided by HuggingFace to create a simple vector (NEVector) with two possible values: 1+*NER_ value* for tokens corresponding to NE, 1-*NER_ value* for all others, where *NER_ value* is an hyperparameter.

Afterwards, we implemented this idea into *DistilBert*'s multi-head self attention: this module takes as input three vectors (query Q, key K and values V) and outputs a vector of attention strengths. In our case, Q, K and V are all equal to the same vector, the hidden state of the previous layer. We decided to follow the direction described in [8] and multiply the K vector by the NEvector, before any processing occurs inside the layer. This should increase the scores (and, accordingly, the attention) of NE tokens.

Finally, we decided to insert this augmentation only in one layer of our model. Since, through the previous analysis on separated layers, the first layer was the worst performing one, it was the one with the highest margin of improvement and that would benefit most from augmentation.

However, the fact that we are utilizing a pre-trained model and TPUs meant that modifying layers of the model and keeping the same pre-trained weights wasn't possible. All of the trials that we made brought to worse results, because the initial weights of the modified layer were randomly initialized, completely changing the attention scores and the performance of the entire model. Through GPU utilization the weights could be kept, but training would have required more time than we had available when we discovered this problem. More details of our future intentions for this study will be discussed in the conclusions.

4.5 Baselines

We implemented two simple baselines to quantify the improvements that neural language modeling brings over traditional algorithms.

The first baseline algorithm simply produces *random* probabilities for both start and end vectors.

The second baseline is an algorithm called *Sliding Window*. The original implementation was first proposed in [1], but it was slightly modified in [2].

Given a passage $P = [P_1, \ldots, P_n]$ and a question Q, we create the set of tokens in the passage PW and a set of words in a question QW. Then, following the changes in [2], we generate a set of *proposal answers* $A = A_1, \ldots, A_q$ taking all *constituents* in the passage (ignoring punctuation and articles). For each A_i, we create a set of tokens $S_i = A_i \cup QW$ and then *score* each answer using the formula:

$$sw_i = \max_{j=1\ldots|P|} \sum_{w=1\ldots|S_i|} \begin{cases} IC(P_{j+w}) & \text{if } P_{j+w} \in S_i \\ 0 & \text{otherwise} \end{cases} \quad \text{with} \quad \begin{array}{c} IC(w) = \log(1 + \dfrac{1}{C(w)}) \\ C(w) = \sum_i (one\_hot(P_i = w)) \end{array}$$

Basically, scores for a proposal answer are computed by sliding a window over all words in the sentence containing the proposal. Index j represents the start of the window, while the window width is given by the cardinality of S_i. If a word w of the window is also in S_i, we assign it a number $(0 < n < 1)$ that gets smaller if there are many instances of that word into the paragraph, because a word that appears many times in an answer isn't probably very discriminative. Scores are summed for the words in the window, then we repeat for all windows in the paragraph and keep the maximum window-score as global score of the answer. The max-scoring span is the final answer to the question.

5 Results

In Table 3 we present the results of the various tests we have conducted and described in the previous sections. We used the official evaluation script for the SQuAD dataset, which compares answers given by our systems to the ground truth answers in the dataset by matching the *question IDs.* The script evaluates our answers by means of two metrics:

Exact Match Measures the percentage of predictions that match any one of the ground truth answers *exactly.*

Macro-Averaged F1 Score Measures the *overlap* between the predicted and ground truth answers, treating them as sets of unigrams. The F1-Score is computed using predicted words that are not in the GT answer as *False Positives* and missed words in the ground truth as *False Negatives.* The F1-Scores are averaged over all answers.

Method	Split	Exact Match	Macro F1	Notes
Human	test	80.3	90.5	As reported in [2]
BERT	val	**51.13**	**71.46**	Uses BERT, 12 epochs training on TPU. Concatenates layers [9,10,11,12]
	test	**64.33**	**78.59**	
Ensemble (BERT + Distil)	val	51.35	71.32	Ensembles Bert (see above) and DistilBert (see below).
	test	64.30	78.32	
Normal	val	**48.71**	**69.03**	Uses DistilBERT, 16 epochs training on TPU. Concatenates hidden states [3,4,5,6]
	test	**61.17**	**75.53**	
Sliding Window Baseline	val	1.09	15.56	Re-implementation of baseline in [1] and [2]
	test	1.76	17.54	
Baseline Random	val	0.004	6.21	Randomly initialized prediction vectors
	test	0.016	6.96	

Table 3: Table containing evaluation results for our experiments, in terms of exact match precision and Macro-weighted F1 score.

5.1 Error analysis

For analysis purpose, we partitioned the questions into their *types.* To do so, we fist analyze the related answers and split them between numerical and non-numerical. Then, the non-numerical answer are further classified, using their NER tags extracted using the SpaCy Python library [9]. We also add two categories that are not NER tags: *clause* for long answers and *other* if none applies. In this way we can see the evaluation scores divided by each category, as shown in table 4.

5.1.1 Common mistakes

From the previous analysis, we can see that our model gets wrong a lot of questions about *numbers*, be it cardinal numbers, money or quantities. We can explain this by the fact that the tokenizer probably splits numbers that are not *common*, ignoring the fact that a number has value not just as a word but also as a mathematical entity.

Another general problem we noticed is that when there are multiple Named Entities of the same type in the paragraph, the model might select one that is not the correct answer but has the same functionality. This is often true with numbers (eg. answering with a date or a quantity that is different to the one that is being asked), but also happens with other kinds of entities, like citing a different TV show episode present in

category	explanation	f1 score	exact score	number of elements
other	Answers in no other categories	74.05%	61.1%	3820
person	People, including fictional	81.58%	75.91%	1475
cardinal	Numerals in no other categories	71.25%	51.05%	1138
date	Dates or periods	73.49%	54.39%	1103
org	Companies, institutions, ...	79.85%	67.46%	1005
clause	Long answers in no other categories	74.55%	56.03%	887
gpe	Countries, cities, states	71.35%	54.34%	449
norp	Political/religious groups	77.22%	57.20%	229
loc	Non-GPE locations	79.33%	64.00%	125
ordinal	Ordinal numbers	68.61%	48.72%	78
other cat	10 uncommon categories, for brevity	83.19%	70.96%	261

Table 4: Scores for the most common categories

the paragraph. This leads us to believe that the model might not fully understand how are entities related, but it *does* understand *which kind of entity* is expected in the answer.

Finally, we notice that a lot of "wrong" answers are simply longer or shorter than the ground truth, as they extract more or less context surrounding the exact answer. These are not exactly *wrong* answers: they are just a little imprecise and unfocused. We believe that this is a limit of using accuracy and F1 score as metrics for the task.

As an example of some common wrong answers:

1. Question: What was the name of the first Doctor Who story released as an LP?
   GT answer: *Doctor Who and the Pescatons.*
   Prediction: *The Chase*

2. Question: In Nepalese private schools, what is the primary language of instruction?
   GT answer: *English.*
   Prediction: *English, but as a compulsory subject, Nepali*

5.1.2 Error gap between validation and test set

From Table 3 we can see that there is a huge gap between the results we obtain in the validation set and in the test set. We noticed that the validation set contains some junk questions and answers, but in small number, so this does not justify such a large gap. Furthermore, the data distribution and the error distribution seem to overall be the same between the two splits.

The real culprit here is the fact that the test set contains *more than one answer* to each question, giving more flexibility to the predictions of the model, which in turn increases the score. The multiple answers often have the same text, but are starting in different positions of the paragraph, or are slightly longer/shorter, adding or removing context to the answer. For example:

```
"answers": [{ "answer_start": 109, "text": "polynomial time" },
            { "answer_start": 109, "text": "polynomial" }]
// or
"answers": [{ "answer_start": 392, "text": "multiplication" },
            { "answer_start": 520, "text": "multiplication" }]
```

Listing 1: Examples of multiple answers to the same question

As a proof of this claim, since the evaluation script provided by SQuAD selects the *max scoring* between all the available ground truth answers, we modified it to select one *random* ground truth answer between them and re-evaluated the normal model. As expected, the performance on the test set dropped and became similar to the validation score (for the "Normal" model, exact score 48.23%, F1 67.90%, similar to validation exact score 48.71% and F1 69.03% and lower than the test scores presented in Table 3).

6 Conclusions

The results we have obtained during our experiments are still far from human performance, but they represent a significant improvement over our baseline algorithms and the final model overall provides good answers to many of the questions in the dataset.

Apart from gathering more data for training or using a deeper encoder that could produce better token representations, both of which are always good possible improvements in ML, we could try to go deeper in the study of the different layers, finding how a different combination of the final layers and the merge of their weights influences performance.

We would also like to further exploit the advantages of ensembling: in our experiments it did not improve on previous results, so we would like to try and ensemble smaller but faster models: for example [10] implements models even smaller than DistilBert that could greatly benefit from ensembling.

The NER module idea seemed promising despite our implementation problems, so we will definitely explore this and possibly other similar enhancements in the future. For example, we could try and find a method to help the model differentiate between NE of the same category, thus improving on one of our main sources of error.

At the end of this project, the thing that has revealed to be the most interesting was the possibility to explore how to inject external knowledge into the model in order to help the encoder build a knowledge more specialized for the task, while searching for the part of the model that lacks that specific knowledge the most. This will be our main focus in the future.

References

[1] Richardson, C. J. Burges, and E. Renshaw. Mctest: A challenge dataset for the open-domain machine comprehension of text. pages 193–203, 2013.

[2] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. Squad: 100,000+ questions for machine comprehension of text, 2016.

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[4] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter, 2020.

[5] Yukun Zhu, Ryan Kiros, Richard 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 *arXiv preprint arXiv:1506.06724*, 2015.

[6] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics. URL `https://www.aclweb.org/anthology/2020.emnlp-demos.6`.

[7] W. Zhou. Ensemble bert with data augmentation and linguistic knowledge on squad 2.0. 2019.

[8] Tingyu Xia, Yue Wang, Yuan Tian, and Yi Chang. Using prior knowledge to guide bert's attention in semantic textual matching tasks. *Proceedings of the Web Conference 2021*, Apr 2021. doi: 10.1145/3442381.3449988. URL `http://dx.doi.org/10.1145/3442381.3449988`.

[9] Matthew Honnibal and Ines Montani. spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing. To appear, 2017.

[10] Iulia Turc, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Well-read students learn better: On the importance of pre-training compact models, 2019.