Alma Mater Studiorum · University of Bologna

Question Answering and Question Generation

Natural Language Processing Course Project

Giacomo Berselli (giacomo.berselli@studio.unibo.it)
Marco Cucè (marco.cuce@studio.unibo.it)
Riccardo De Matteo (riccardo.dematteo@studio.unibo.it)

February 8, 2022

Contents

Introduction

In this project we tackled the problems of Question Answering (QA) and Question Generation (QG) on the SQuAD v1.1 dataset. Those two tasks have an intrinsic connections and could be regarded as dual tasks. On one side, to answer dataset's question we developed three different architectures, a recurrent neural network and two pre-trained transformer-based models from the HuggingFace PyTorch library. On the other side, to generate a question given the answer and the context related, we implemented a modular Sequence-to-Sequence model, made up of different RNNs and Transformers architectures.

Part I

Question Answering

1 Summary

In this project we addressed the problem of *Question Answering* on the *Stanford Question Answering Dataset (SQuAD) v1.1* which consists in answering a given question about a paragraph by finding the correct span of text inside it.
For this task we had to set up an infrastructure to process the dataset, feed its examples to our models and analyze the results. We did it with the help of some useful external libraries like HuggingFace [7], WandB [12] and the famous machine learning framework Pytorch [9].

We implemented both Recurrent and Transformer-based models, for a total of 3 different architectures :

- DrQA (Document Reader Question Answering).
- BERT for Question Answering.
- Electra for Question Answering.

We built and trained from scratch the recurrent model (DrQA), while we decided to only fine-tune the two Transformer-based architectures (BERT and Electra) by adding on top of them a classification head suited for our task (different between the two models). This choice was dictated by the fact that we didn't had the necessary resources to train those models from scratch but was also encouraged by the great availability of pre-trained models on different tasks in the above mentioned HuggingFace library.

In order to fine-tune the transformer-base architectures, instead of using a frozen backbone and only let the gradients

flow in the classification head, we decided to take advantage of a learning rate scheduler specifically thought to make the backbone slowly adapt to the new data and task, while also successfully training our custom head.

The results we obtained in our experimental setup were aligned with our expectations and even exceeded them with the Electra architecture which was able to achieve 83% F1 and 67% EM on the validation set. These results are similar to those reported in the corresponding papers from which we took inspiration for our architectures and setup.

2 Background

Question Answering (QA) is a Natural Language Processing (NLP) and machine comprehension (MC) task, which has gained popularity over the past few years. The focus is to build a system which is capable to automatically answer questions about a passage of text provided in a natural language form. Clearly this is a challenging task for machines, requiring both understanding of natural language and knowledge about the world.

The Stanford Question Answering Dataset is a reading comprehension dataset made up of 100,000+ questions on 500+ articles posed by crowd workers on a collection of Wikipedia articles, where the answer to each question is a text segment, or span, from the relevant reading passage. The reading sections in SQuAD are taken from high-quality Wikipedia pages, and they cover a wide range of topics from music celebrities to abstract notions. A paragraph from an article is called a passage, and it can be of any length. The reading comprehension system given the query-passage pair in input, is expected to output the start and end position of the answer in the corresponding context.

Traditional approaches to the QA task include pipelined NLP models involving *Named Entity Recognition (NER)*, syntactic and semantic analysis. However, with the recent advance of deep learning, the top-performing models are all neural network-based end-to-end models. These systems, also called *Encoders*, are able to address the main issue with this task, the capability of translating the sentences into an internal numerical representation in order to generate valid answers. As of today, the two most popular neural architectures used are *Recurrent Neural Networks (RNN)* and *Transformers*.

A recurrent neural network is a special type of an artificial neural network adapted to work for time series or data that involves sequences. Ordinary feed forward neural networks are only meant to learn from data point which are independent of each other. However, if we have data in a sequence such that one data point depends upon the previous one, we need to modify the neural network to incorporate the dependencies between these data points. RNNs can thus exploit the inherent property of text sequences of being sequential, by having the concept of 'memory' that helps them store the states or information of previous inputs.

Transformers are also designed to handle sequential input data, such as natural language. Differently from RNNs, they do not necessarily process the data in order. Rather, the attention mechanism provides context for any position in the input sequence. For example, if the input data is a natural language sentence, the transformer does not need to process the beginning of the sentence before the end. Instead, it identifies the context that confers meaning to each word in the sentence. This feature allows for more parallelization compared to recurrent architectures.

3 System Description

3.1 Data Handling

3.1.1 Dataset Preparation

The *Stanford Question Answering Dataset* (SQuAD) is downloadable from the official site as a JSON file [11]. The file is composed of different Wikipedia articles, each one divided in paragraphs, and for each paragraph there is a set of questions. Each question comprises of: *id, text of the question, list of possible answers*. Some questions have only one correct answer, other may have multiple ones. Each answer is characterized by the text of the answer itself and by a number which is the start char of the text span in the paragraph corresponding to that answer. Given the mentioned data format, we decided to encode the dataset as a Pandas Dataframe where each row represents a different question and stores all the needed information for the task: *context, question, answer texts* and *answer start-end labels*.

3.1.2 Error Removal

Having built the dataframe, we analyzed it and discovered that it contained multiple errors. The labels of some examples are indeed wrong, since the words corresponding to the given span do not match with the given answer text. We decided to remove those rows from the dataset as they could have negatively affected the training process.

3.1.3 Training-Validation Split

The SQuAD datased is provided as a single file, so we decided to split it for training and validation purposes. The validation set consists of about the 20% of the original dataset. The splitting mechanism is designed to prevent that questions/paragraphs regarding the same title are in different splits, as in that case they would not serve the purpose of ensuring that the system is able to generalize well since they are too similar between each other.

3.1.4 Text Processing

In order to be able to feed the text (both context and questions) from SQuAD to our Neural Architectures, we needed to perform some processing. Our text processing pipeline includes different steps and we designed slightly different versions of it for the different models we implemented for this task.

As said, we developed two text-processing pipelines, one for our recurrent model and another for transformer-based models. Both of them make use of the HuggingFace NLP library [7]. Hugging Face is a large open-source community which provides easy-to-use frameworks with well-furnished API, to perform a plethora of NLP tasks and to leverage multiple deep learning architectures and pre-trained models.
In particular, we employed the Tokenizers library [6]. Tokenization is a way of separating a piece of text into smaller units called tokens. In NLP tasks, the most common type of tokenization is to split texts into their corresponding words, since they are the building blocks of Natural Language.

For the RNN model, we used the WordLevel tokenizer. The processing is done separately for context and questions, and comprises the following steps: a normalization step, in which we remove all accents, set all words to lower case, strip all unnecessary spaces, and handle special characters; a tokenization step, in which, based on white space and punctuation, we break down the text into tokens, and assign an id to each unique token; a final padding step, in which all sentences in a batch are padded in order to create vectors with the same length.

For the Transformers, we leveraged the BERT tokenizer pre-trained on the 'bert-base-uncased' vocabulary [6], available on the Hugging Face website. It applies the same cleaning and normalization steps mentioned above but in addition splits words either into the full forms or into word pieces; one word can thus be broken into multiple tokens to help reduce the number of Out Of Vocabulary words. This tokenization is applied to the combined question-context sequence, and in order to distinguish between the two some special tokens are added: [CLS] at the beginning of the sentence, [SEP] between question and context. The input sequence is truncated to the maximum number of input tokens that Bert-like models can accept (512). We also make sure that if the original sequence is truncated, the retained part contains the answer. Finally, each sentence is padded to the right, at the maximum length inside the batch.

3.1.5 Dataset and Dataloader

Both the train and validation dataframes are wrapped into an Hugging Face dataset, which allows us to perform the tokenization steps one batch at a time when needed, instead of on the whole dataset beforehand. A Pytorch Sampler is used to retrieve multiple examples from the above mentioned dataset and collate them together in a batch of fixed size.

3.2 Models

For this project, we decided to implement models from both the two prominent neural architectures suited for this task, recurrent and transformer-based, in order to compare the performance. In particular, we deployed a recurrent model *DrQA* [2], and two transformer-based models, *Bert* [4] and *Electra* [3].

3.2.1 DrQA

Document Reading Question Answering (DrQA) is an end-to-end system for open domain question answering which involves an information retrieval system from a text passage as well. In particular, DrQA is targeted at the task of "machine reading at scale", in which we are searching for an answer to a question in a potentially very large corpus of unstructured documents. Thus, the system has to combine the challenges of document retrieval (finding the relevant documents) with that of machine comprehension of text (identifying the answers from those documents). Therefore, it seemed perfect to address our problem. The implemented model proposed is inspired by the paper 'Reading Wikipedia to Answer Open-Domain Questions' [2].

The architecture [Figure 1] is composed of the following sub-modules:

- **Word Embedding**: Both question and context tokens are passed through an embedding layer initialized with pre-trained *GloVe* word vectors of 300 dimensions [5]. Each word id is converted into the corresponding 300d floating point vector which encodes various features associated with the word into its dimensions.

- **Align Question Context**: This layer produces a weighted representation of question embedding for each context token, that express which portion of context is more relevant w.r.t. the question. For each word p_i in the context, $f_{align}(p_i)$ is computed as $\sum_j a_{i,j} E(q_j)$. Specifically, $a_{i,j}$ represents the weights applied to each question embedding vector, which is computed by the dot-product between non-linear mappings (obtained via a dense layer followed by *ReLu*) of context and question embeddings $E(q_j)$.

$$a_{i,j} = \frac{exp(\alpha E(p_i) \cdot \alpha E(q_j))}{\sum_{j'} exp(\alpha E(p_i) \cdot \alpha E(q_{j'})}$$

  This features' vectors add soft-alignment between similar words.

- **Context Encoding**: This layer generates an internal representation of the paragraph tokens by concatenating each layer hidden units of a *multi-layer bidirectional LSTM* [Figure 2], which takes in input paragraph token embeddings enriched with the aligned embeddings generated by the Align Question-Context Layer .

- **Question Encoding**: An internal representation of the question tokens is generated by passing the embedded question through a *multi-layer Bi-LSTM*. Once again, the hidden states of all the timesteps from all the layers are concatenated together. Then, a *Linear Self Attention* is computed over the question encodings to determine the importance of each word in the question. The scores coming from this computation b_i are used to weight the output q_j of the previous LSTM.

$$weights = b_i = \frac{exp(w \cdot q_j)}{\sum_{j'} exp(w \cdot q_{j'})} \quad ; \quad q = \sum_j q_j \cdot b_j$$

- **Bilinear Attention**: Since the goal is to predict the span of tokens that correspond to the answer inside the context, two linear classifiers are independently trained for predicting the two ends of the span. Two different projections of the question vectors are computed which are then both multiplied by the context vectors to compute similarities.

$$e_t = s^T W h_t$$

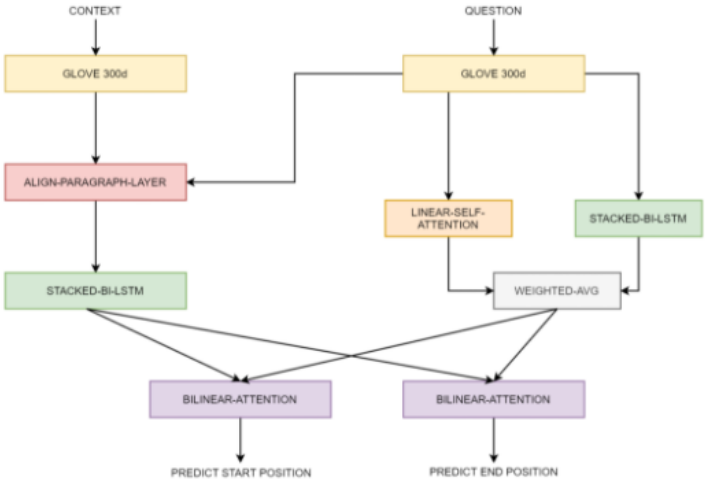

Figure 1: DrQA Model architecture

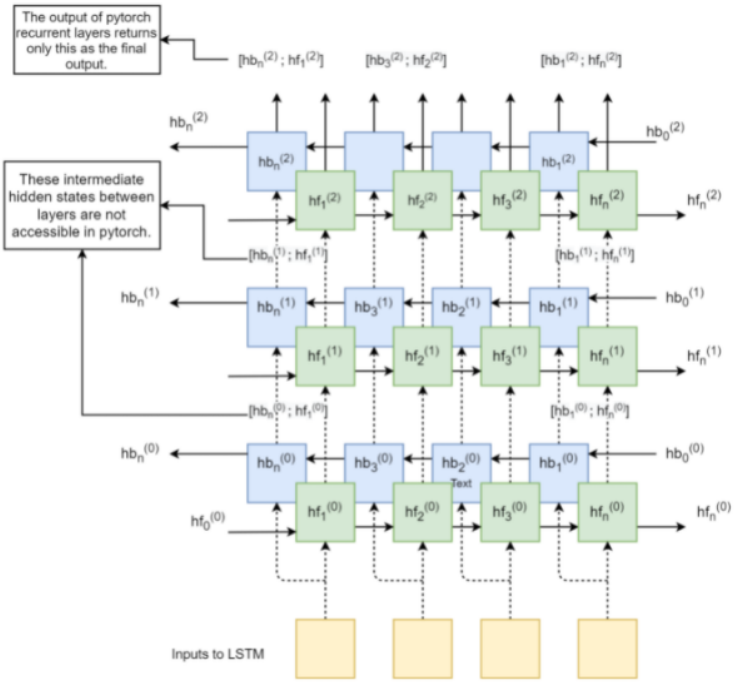

Figure 2: Multi-layer Stacked Bilinear LSTM

3.2.2 BERT

Bidirectional Encoder Representations from Transformers (BERT) is a bidirectional transformer, pretrained using a combination of masked language modeling objective and next sentence prediction on a large corpus comprising the *Toronto Book Corpus* and *Wikipedia*.

The first model was proposed in 'BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding' by Devlin et al [4]. Unlike other language representation models, *BERT* is designed to pre-train deep bidirectional representations from unlabeled text by jointly conditioning on both left and right context in all layers at the same time performing a particular self attention, without having to perform a sequential scan of the input, as instead done by RNNs. The paper's results show that a language model which is bidirectionally trained can have a deeper sense of language context and flow than single-direction language models. In the paper, the researchers detail a novel technique named *Masked LM (MLM)* which consists in replacing the 15% of the words in each sequence with a *[MASK]* token before feeding the words into the transformer. The model then attempts to predict the original value of the masked words, based on the context provided by the other, *non-masked*, words in the sequence. Furthermore, the paper of the authors explains another technique named *Next Sequence Prediction (NSP)*, applied in the training process where the model receives pairs of sentences as input and learns to predict if the second sentence in the pair is the subsequent sentence in the original document.

Usually, BERT comes in different pre-trained versions, which can be fine-tuned with just one additional output layer to create state-of-the-art models for a wide range of tasks, such as question answering. The pre-trained models are able to achieve great performance and to generalize well, since they were already prepared on many different tasks.
For our project we decided to use the base uncased version of BERT from the Hugging Face library, which consists of 12 layers and has a hidden size of 768. This pre-trained version needs in input the concatenation of question and context ids, already processed as described in section 'Text Processing', and two additional vectors, 'type_ids' and 'attention_mask'. The first one specifies for each token to which original sequence it belongs among question and context vector, while the second is a vector which masks the padding tokens. In order to predict the text span corresponding to the answer we decided to process the output of the model with two separate linear layers to distinguish between start and end tokens classification [Figure 3]

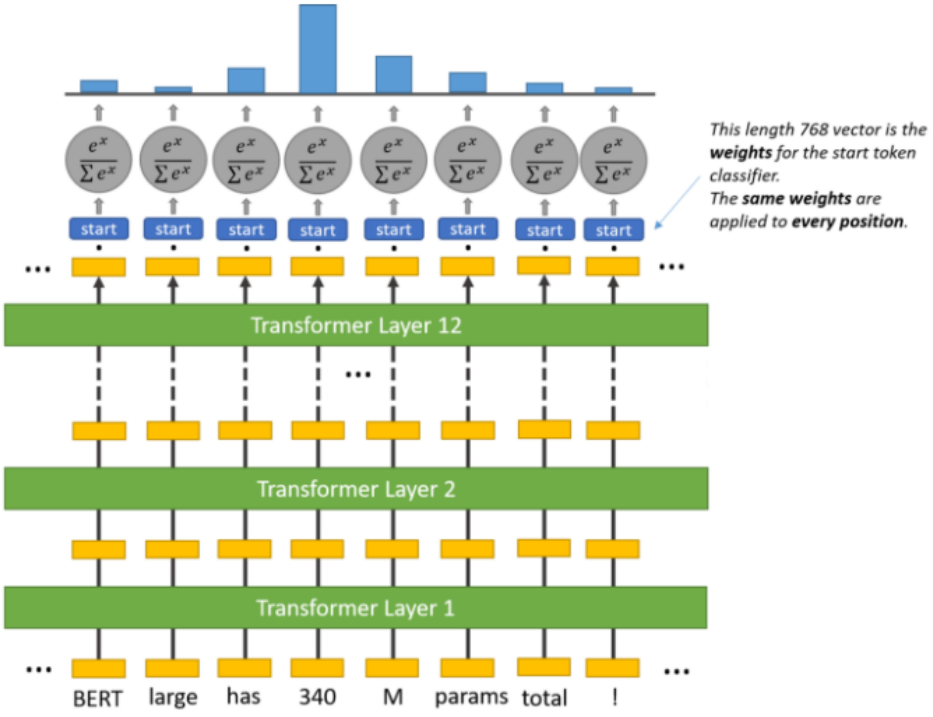

Figure 3: Bert Model

3.2.3 Electra

Electra replaces the *MLM* task of BERT with *Replaced Token Detection (RTD)*, which looks to be more efficient and produces better results. Instead of masking the input, the approach corrupts it by replacing some input tokens with plausible alternatives sampled from a small generator network. Then, instead of training a model that predicts the original identities of the corrupted tokens, a *discriminative model* is trained that predicts whether each token in the corrupted input was replaced by a generator sample or not. This new pre-training task is more efficient than MLM because the model learns from all input tokens rather than just the small subset that was masked out.

For our classification head, this time, instead of just fine-tuning Electra by adding only one additional output layer, as we did with Bert, we decided to set up a more elaborate classifier to put on top of the pre-trained model. The outputs from Electra Encoder are first passed through a *Bi-LSTM* layer and then a non-linear projection is applied by means of a Linear layer followed by *ReLU*. Finally a single Linear layer act as a classifier for both the start and end token position.

4 Experimental Setup and Results

4.1 Environment

The environment used to train and test the models consist in different third-party libraries. The architectures are built using *PyTorch* as framework, an open source machine learning library based on the Python programming language and the *Torch* library. It is one of the preferred platforms for deep learning research. As already described, the Transformers pre-trained models were taken from the *HuggingFace* platform and were fine-tuned on the *SQuAD* dataset to fit our QA task.
Since the models are really huge and heavy to train, we take advantage of Google Colaboratory, a platform well suited to deploy machine learning models which allows a free use of their GPUs. In particular, to train and evaluate our models we used the NVIDIA Tesla T4 GPU, with 16GB of RAM. The metrics, along with model checkpoints and results, were directly logged into the Weights & Biases (WandB) platform [12], to keep track of hyperparameters, system metrics and many other useful information about the models' performance.

4.2 Metrics

To evaluate our models performances, at the end of each train epoch to keep track of progressive improvements and after the training to get a sense of the generalization capabilities on the validation/test set, we made use of many different metrics. The official SQuAD documentation already comes with an evaluation script which we extended in order to compute other metrics that helped us to better understand the different behaviours of the multiple models deployed in this work.
The main metrics we employed are:

- **F1 score**: defined as the armonic mean between precision and recall that are in turn computed by first counting the number of tokens (as strings) in common between the predicted answer and the ground truth.
- **Exact Match (EM)**: this is the number of predicted answers that has the same exact wording of the true answer, divided by the total number of answers
- **Numerical Accuracy**: this is just a normal accuracy but this time computed on the raw numerical output (start and end tokens) of the networks instead of on the normalized strings that correspond to that span in the text. It has been useful to keep an eye on overfitting since it doesn't express how similar predictions and ground truths are but the mere match between numerical labels.
- **Mean Token Distance**: it is the average absolute distance between the correct token and the predicted token (for both end and start tokens). It highlights examples where the models are struggling the most and is particularly interesting for determining the irreducible error deriving from the available data (for example when the model indeed produce a correct answer for a given question but the ground truth in the dataset is somehow different).

The list of strings that compose an answer for both the prediction and the ground truth of each example in a batch are always normalized first, in order to remove differences between the two that are not meaningful for our task and should not penalize the numerical results expressed by the metrics.

All the metrics are computed for each question in each batch and then averaged out across all examples in the dataset seen during either the train or the evaluation loops.

4.3 Setup

As a first step, we implemented and setup the models accordingly to the corresponding original papers, in order to have a well-established benchmark. Then, on the basis of the previously described metrics, we tried to improve the models performance by tuning the hyperparameters and employing specific techniques.

As a baseline, we trained the *DrQA* model as described in the paper. In particular, the architecture is composed of 3 *LSTM* layers with a hidden dimension of 128. The size of the batches more suitable for our available resources was 32. Furthermore, since the model was overfitting on the training set and we saw degraded performance on the validation set we decided to introduce a regularization layer. In particular, a dropout of 30% was introduced after the *LSTMs* layers, in order to prevent overfitting. Since we decided to make the weights of the Embedding layer trainable, we added a dropout layer also after that. In addition, knowing that *LSTMs* suffer from 'exploding gradients', we performed gradient clipping during backpropagation to 2.0. Finally, the original learning rate used by the authors in the paper $1 \cdot e - 3$ was too low for our setting, while the best results were obtained with a value of $2 \cdot e - 3$.
In the original paper the authors used as optimizer *Adamax* with the default parameters of PyTorch. *Adamax* is a special case of *Adam* and it is supposed to be used in a setup that has sparse parameter updates (e.g. word embeddings) since it is more robust to noise in the gradients.

Again, regarding Transformer-based models, we started from the predefined parameter configuration suggested by the HuggingFace library. For Bert-based model, we decided to set the learning rate of the optimizer to $3 \cdot e - 5$, as seen in [10], while for Electra we set it to $5 \cdot e - 5$. In addition, we introduced a linear learning rate scheduler with a warm-up of 2000 steps, to reduce the possibility of preventing the model from learning when exposed to sudden new data while fine-tuning. In the previous cited paper, the authors reported that their model was overfitting a lot on the training data, so they recommended to introduce some regularization techniques. Since we faced the same problem, we added dropout and a weight decay term.
We decided to use the *AdamW* optimizer which offers the advantages of Adam together with a more precise weight decay handling. Due to limited resources availability, we were only able to feed batches of 8 sentences to the models.

For every architecture we made use of Pytorch *Cross Entropy Loss*.

4.4 Results

The table below shows the best results for each of the implemented architectures obtained on both training and validation set from the *SQuAD v1.1* dataset.

	Training		**Validation**	
	F1 (%)	**EM** (%)	**F1** (%)	**EM** (%)
DrQA	72.4	58.1	61.5	44.7
BERT	91.2	82.3	78.7	62.9
Electra	93.1	84.5	82.4	66.7

Table 1: Best results

From the table [1], we can clearly see how the Transformer-based models achieve better performance compared to RNNs. In particular, *DrQA* obtains acceptable results but is not able to generalize well as the Trasformers . Moreover the expected improvement in the results given by Electra when compared to BERT are definitively there. A visible improvement was also given by the additional *LSTM* layer which gave us an additional 1-2%. Even though the model slightly overfits on the training set, it is able to achieve an F1 score on the validation set of 82.4% which is incredibly higher than our recurrent implementation.

5 Error analysis

To get a better understanding of the obtained performances, it is useful to take a look at some errors made by the implemented models.

- **Example 1**:
  - **Context**: ... the Battle of Lobositz on 1 October 1756, Frederick prevented the isolated Saxon army from being reinforced by an Austrian army under General Browne. The Prussians then occupied Saxony; after the Siege of Pirna, the Saxon army surrendered in October 1756, and was forcibly incorporated into the Prussian army. The attack on neutral Saxony caused outrage across Europe and led to the strengthening of the anti-Prussian coalition ...
  - **Question**: What happened to the Saxon army?
  - **Correct Answer**: Saxon army surrendered in October 1756, and was forcibly incorporated into the Prussian army
  - **DrQA answer**: forcibly
  - **Electra answer**: surrendered in October 1756, and was forcibly incorporated into the Prussian army

Here is an example of a kind of error that the *DrQA* model makes . It seems to capture the meaning of the

question and it was able to correctly determine the position of the answer in the context, but could not predict the correct span. Probably, with a better Attention layer, such as the total-attention of Electra-based model, it would have chose more precise start and end tokens as the Transformer is able to do in these cases.

- **Example 2**:
  - **Context**: ... At the outbreak of the war in the Pacific the Dutch Admiral in charge of the naval defense of the East Indies, Conrad Helfrich, gave instructions to wage war aggressively. His small force of submarines sank more Japanese ships in the first weeks of the war than the entire British and US navies together, an exploit which earned him the nickname 'Ship-a-day Helfrich' ...
  - **Question**: What was the nickname given to the Dutch Admiral in charge of the East Indies?
  - **Correct Answer**: Ship-a-day Helfrich
  - **DrQA answer**: Conrad Helfrich
  - **Electra answer**: Ship-a-day Helfrich

  These kind of examples are extremely difficult since there are multiple text span that seems to match with the question. Here DrQA often provide the wrong answer showing less reading comprehension capability than Transformers-based models. As we could see, Electra indeed gave the correct answer.

- **Example 3**:
  - **Context**: ... the House of Saud took over the Hijaz, and regimes led by army officers came to power in Iran and Turkey. B̈]oth illiberal currents of the modern Middle East,ẅrites de Bellaigne, Ïslamism and militarism, received a major impetus from Western empire-builders.Äs often happens in countries undergoing social ...
  - **Question**: de Bellaigne attributed the growth of Islamism and militarism to what Western catalyst?
  - **Correct Answer**: : empire-builders
  - **DrQA answer**: writes de Bellaigne, Islamism and militarism, received a major impetus from Western empire-builders
  - **Electra answer**: Western empire-builders

  There are some examples, as the one shown above, for which both the models are able to determine the correct answer but the one provided in the dataset is very specific and the recurrent model in particular adds unnecessary text. Here to obtain superior results it could be possible to incorporate a penalty factor in the loss based on the lenght of the answer but in all fairness in these cases most of the times the answers given by Electra are even more accurate than the ones in the dataset.

- **Example 4**:
  - **Context**: ... The Church of the Holy Apostles in Thessaloniki was built in 1310ŏ01314. Although some vandal systematically removed the gold tesserae of the background it can be seen that the Pantokrator and the prophets in the dome follow the traditional Byzantine pattern. Many details are similar to the Pammakaristos mosaics so it is supposed that the same team of mosaicists worked in both buildings. Another building with a related mosaic decoration is the Theotokos Paregoritissa Church in Arta. ...
  - **Question**: The same team of mosaicists worked on the Church of the Holy Apostles in Thessaloniki as which other building?
  - **Correct Answer**: Pammakaristos
  - **DrQA answer**: mosaicists
  - **Electra answer**: Theotokos Paregoritissa Church in Arta

  Here is an example where both the models gave the wrong answer but we can see a difference. While DrQA has totally misunderstood the question, giving an answer which is completely off, Electra seems to have a better understanding of the question but still gave a wrong answer. This happens for particular convoluted questions.

6 Discussion

In the end, the performances as expressed by the metrics show what we expected: the transformer-based models are able to obtain much better results than the recurrent one, and to generalize very well. Nonetheless it could be possible to improve the results of *DrQA*.
As explained in [2], the authors, in order to enrich the information produced by the context encoding, added some features beyond just the token embeddings. We have implemented the alignnment feature between context and question tokens but they also added an 'exact match' feature. It is composed of three simple binary features, indicating whether a context word can be exactly matched to one question word, either in its original, lowercase or lemma form. This simple features turned out to be extremely useful to them. In addition, they also added some other features which reflect some specific properties of each token in its context, which include its *part-of-speech (POS)* and *named entity recognition (NER)* tags and its (normalized) term frequency.

Our custom Electra-based model outperformed all the other architectures, and it was able to reach near human-level performance, when evaluated on the *F1 score* metric of the official *SQuAD v1.1* dev set. Again, even though our solutions achieve extremely satisfactory results, more advanced or recent models like T5, with new state-of-the-art techniques could probably improve the performance on this task.

Part II

Question Generation

1 Summary

In this project we addressed the problem of *Question Generation* on the *Stanford Question Answering Dataset (SQuAD) v1.1* which consists in generating questions given a specific answer related to a paragraph of the dataset. For this task we extended the infrastructure designed to solve the Question Answering problem, in order to process the dataset accordingly to the new specific task, feed the examples to the QG models and analyze the subsequent results. We did it with the help of some useful external libraries like HuggingFace [7], WandB [12] and the famous machine learning framework Pytorch [9].

We implemented a total of three Encoders and two Decoders, that make use of different strategies to accomplish their specific task. Since each Decoder make use of an Attention mechanism, we also decided to implement two Attention strategies. With them, we were able to develop three different architectures, made up of different combination of the Encoder-Decoder pair

- Baseline model.
- RefNet for Question Generation with Bahdanau attention.
- BERT for Question Generation with Bahdanau attention.

We built and trained from scratch the two RNN Encoders of the Baseline and RefNet modules, while only fine-tuning the transformer-base module.

The results we obtained were not astonishing but overall satisfactory, given the difficulty of the task.

2 Background

Question Generation (QG) is an important yet challenging problem in Natural Language Processing, whose objective is to be able to automatically generate correct and relevant questions from textual data. Previous works mainly made use of rigid heuristic rules to transform a sentence into related questions, and only in recent years the latest state-of-the-art neural text generation architectures have become so popular.
In this project we addressed the Question Generation task as an extension of the QA project, by conducting a preliminary study on *Neural Question Generation (NQG)* on the same *SQuAD v1.1* dataset used for Question Answering. The general goal of NQG is to generate natural language questions from text without pre-defined rules, and in this case to generate answer focused questions.

A model which is able to take a sequence in input and producing one in output is often called Sequence-to-Sequence. A *Sequence-to-Sequence (Seq2Seq)* model is a special class of Neural Network architectures and it is typically implemented as an Encoder-Decoder architecture. The *Encoder* turns each input item into a corresponding hidden vector which represents its internal numerical representation, while the *Decoder* reverses the process, turning the vector into an output item, using the previous output at each step as the input context. It is therefore clear how this behaviour could be perfectly represented using *Long Short-Term Memory (LSTM)* networks for both the components. In recent years, the progress given by Transformers in the NLP field, led to the successful substitution of RNNs as components of Seq2Seq architectures.

3 System Description

As already mentioned, this task is the dual of the Question Answering task. For this reason we were able to re-utilise part of the infrastructure already built for it. We were extremely careful in implementing a modular structure that could be reused with the least possible change for multiple NLP problems which aimed at similar goals. To avoid being redundant, in the next sections we will explain the key concepts about the implementation we provided for this task, without repeating the parts that we kept the same from the QA work.

3.1 Text Processing

In order to address the Question Generation task, a *Seq2Seq* model needs to be developed and implemented. The Encoder component reads in input a text sequence which could be composed of two independent inputs, typically a context and the answer associated to it, or a single input composed by the combination of the answer and the context. Then it needs to compress the information provided in the *'internal state vectors'*, and the final state produced will then initialize the Decoder initial state. The Decoder instead, generates the output sequence one token at a time, with the number possible classes to be predicted equal to the number of words belonging to the output question vocabulary.

The management of the input and output vocabulary must be carefully studied, to allow the model to obtain the best results in an already difficult task. To be more specific, the model should be able to predict a number of classes equal to the number of the output question-vocabulary's words. Since the size of this vocabulary is often considerably large, in order to avoid making the task unnecessary tougher, we decided not to exploit the HuggingFace pre-trained tokenizers [6] which come with their vocabulary already built-in, but to take advantage of a custom trainable tokenizer, in order to train it later on the text corpus available from the *SQuAD v1.1* dataset. Moreover, since the Encoder and Decoder are two separate architectures, where the former takes in input only the context and the answer, while the latter outputs the predicted question, we thought that maintaining two separate tokenizers with two different vocabularies (answer-context vocabulary and question vocabulary) was the best choice to built lightweights but effective architectures. We also limited the size of the output vocabulary including only the words appearing at least twice in the text corpus, to reduce it's size.

Nevertheless, both the tokenizers perform the same preprocessing pipeline, by normalizing the input text, removing accents and unnecessary white spaces, and handling special characters. Then, a tokenization step is performed, in which, based on white space and punctuation, we break down the text into tokens, and assign an *id* to each unique token. Finally a padding step is performed, in which all sentences in a batch are padded in order to create vectors with the same length. We also decided to add to the sequence provided in input to the decoder (the question) two special tokens *[SOS]* and *[EOS]* indicating, respectively,the start and end of the sequence.

3.2 Models

Clearly, a model that is able to generate questions starting from a paragraph and a given answer comprised in that paragraph, needs to take in input one or more text sequences, and to produce in output another text sequence. As already mentioned a model with this characteristics is called *Seq2Seq* (Sequence to Sequence).

We decided to implement a single *Seq2Seq* prototype which is responsible for taking in input the context and the answer and producing in output the question relative to that answer, by delegating the two phases respectively to an *Encoder* and a *Decoder*. Then, the different models with which we experimented are built by putting together different combinations of the multiple Encoders and Decoders we implemented, that make use of different strategies to accomplish to their specific task.
Since each Decoder make use of an Attention mechanism, we also decided to implement two Attention strategies.

We are now going to describe in details the implementation of the building blocks of our Seq2Seq models : Encoders, Decoders and Attention Mechanisms.

Encoders:

- **BaseEncoder**

  It takes in input a context and an answer , which are both passed through an Embedding layer. Each *word id* is converted into the corresponding 300d floating point vector.
  Context and answer embeddings are then 'aligned' by an alignment module, which determines the importance of each word in the context with respect to the answer. We then run a bidirectional *LSTM* network on the previously computed context embeddings enriched by concatenating the alignment feature. The recurrent module produces *Annotation Vectors* as the concatenation of the network's forward and backward hidden states for each input token.
  Each encoder should generate two distinct outputs, *context_vectors* which encode the internal representation of the context conditioned on the answer (this vectors will be used by our attention mechanism) and an *initial hidden state* for the decoder.
  To compute the initial state of the decoder, we run another bidirectional *LSTM*, this time on the answer embeddings. We form the condition encoding h^a by concatenating the final hidden states from each direction of the *Bi-LSTM* . These vectors are passed through a single linear layer which performs a projection that should

encode the general meaning of the answer, producing the vector r. This projected vector is then summed with the mean of all the context annotation vectors h_i to also include a compressed representation of the context in what will be the first hidden state of the decoder.
Finally to produce the vector s_0 whose dimension match the one of the decoder hidden states, a non linear projection is applied to r by means of a linear layer followed by a $\tanh$ activation function. The process to obtain an initial state for the decoder s_0, is expressed by the two formulas below:

$$r = Lh^a + \frac{1}{n}\sum_{i}^{|D|} h_i^d \quad , \quad s_0 = \tanh W_0 r + b_0$$

- **RefNetEncoder**

  The implementation of this encoder was inspired by [8]. In the paper the authors use a 3 layered encoder consisting of: Embedding, Contextual and Passage-Answer Fusion layers [Figure 4].

  - **Embedding layer**: here a d-dimensional embedding for every word in the passage and the answer is computed. Additionally, for passage words, we also compute a positional embedding based on the relative position of the word w.r.t. the answer span by adding a binary feature which express whether or not a specific word in the context is also part of the answer. For every passage word, this positional embedding is concatenated to the word embedding to form h_i.

  - **Contextual Layer**: in this layer, we compute a contextualized representation for every word in the context by passing the word embeddings (as computed above) through a bidirectional LSTM. We then concatenate the forward and backward hidden states. Since the answer correspond to a span in the passage, each answer embedding is concatenated with the hidden state of the previous LSTM layer from the corresponding position inside the context. That is, we take the concatenation of the answer embeddings with the representation of the corresponding answer words in the context. We then obtain a contextualized representation for the n answer words by passing this concatenated vectors through an LSTM. As usual only the final state $h^a = [\overrightarrow{h_n^a}; \overleftarrow{h_n^a}]$ of this Bi-LSTM is used as the answer representation in the subsequent stages.

  - **Passage-Answer Fusion Layer**: in this layer, the representation of the context words is refined based on the answer representation, as follows:

    $$\tilde{h}_i^p = \tanh\left(W_u[h_i^p; h^a; h_i^p \odot h^a]\right) \quad \forall i \in [1, m]$$

    We use the output of this non linear projection as the fused passage-answer representation which will be used by the decoder.
    Finally, to compute an initial state for the decoder we simply perform a linear projection from the encoder hidden dimension to the decoder hidden dimension, of the concatenation of the final answer hidden states from each direction of the *Bi-LSTM*.

The overall encoder module is represent in [Figure 4]

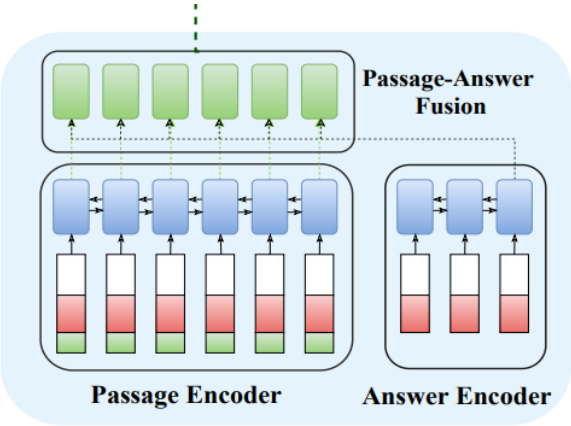

Figure 4: Encoder

- **BertEncoder**

  This is the 'simpler' encoder we implemented. It makes use of the pretrained Bert model from HuggingFace [7]. It takes in input the combination of the context and answer and returns as output the internal representation of each word in the sequence produced by *BERT*. The initial hidden state for the decoder is taken as the pooled output of all the hidden states which contains a compressed representation of each sequence in a batch.

Decoders:

- **BaseDecoder**

  This module produces the prediction of the next word in the sequence (question) given the state from the previous iteration. It makes use of the attention module to compute a weighted sum of the encoder hidden states (the internal representation of each word in the context). The obtained vector, together with the question embedding (one word at a time per example in the batch), is fed to an LSTM, which is also given the hidden state from the previous iteration.
  A linear classification layer is then applied to the output of the recurrent module, for predicting the next word of the question.

- **BertDecoder**

  This decoder is nearly identical to the Base Decoder. The fundamental difference here is that we first run a *LSTM* network on the question embedding conditioned on the previous hidden state of the decoder, and only after that we compute the attention between its output and encoder hidden states. This is because Bert already makes use of an attention mechanism to generate its internal representation while doesn't employ a recurrent module, so we inverted the two operations. As usual a final classification layer is applied to obtain the final prediction.

Attention Mechanism:

- **Custom Attention**

  The attention module is where we calculate the attention values over the source sentence. These values will then be used to weight each word of the context so that the decoder can compute a prediction by attending to specific parts of the paragraph. The inputs to this module are the encoder states that are the internal representation of each word in the context and the decoder hidden state at time t, which will determine the predicted output.
  To determine the weights for the context vectors, two linear layer , interleaved by a *tanh* activation function, are used to compute similarities between the decoder vector and each of the encoder vectors.
  These weights are turned into scores by the application of a *softmax* function. They express the relevance of each of the passage representation vectors with respect to the current state of the decoder.
  A weighted sum of the passage hidden states, based on the obtained scores, is then computed and returned to be used by the decoder module.

- **Bahdanau Attention**

  In addition to our custom attention, we implemented also a particular type of attention, taken from the paper [1], and called *Bahdanau attention*, from the name of the paper's author. The idea behind is to allow the model to automatically (soft-)search for parts of a source sentence that are relevant to predicting a target word.

  We parametrized the alignment model as a feedforward network which takes in input the concatenation of the previous hidden state of the decoder (s_{i-1}) and a sequence of annotations ($h_i, ..., h_{T_x}$) given by the encoder states. Each annotation contains information about the whole input sequence with a strong focus on the parts surrounding the i-th word of the input sequence.
  A non linear mapping is applied to the result of this computation to produce what is called 'energy' . The dot product between the encoder states and the energy vectors is used to produce scores which are then transformed into weights via a *softmax* function. This produces for each annotation h_j a weight α_{ij}.
  A context vector c_i is then computed as a weighted sum of the annotations h_i with the weights computed previously, and finally the outputs are applied as attention weights to the original encoder states.

  This approach allows the decoder to decide to which parts of the source sentence to pay attention to. With this approach the information can be spread throughout the sequence of annotations, which can be selectively retrieved by the decoder accordingly.

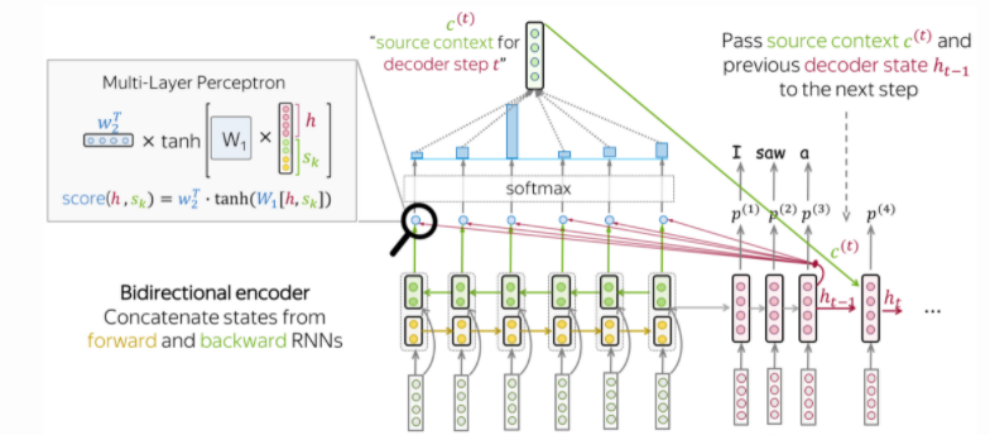

Figure 5: Bahdanau Attention

Seq2Seq model

Finally we implemented a prototype of *Seq2Seq* model which wraps a pair of Encoder-Decoder, as follows:
The model takes in input the ids of the context, answer and question, with the first two being passed to the desired Encoder, while the third being the input of the Decoder. The Encoder returns as output a tensor composed of all its hidden states and a single hidden state tensor to be used as the decoder's first *LSTM* hidden block. Then, one step at a time, the Decoder is fed with the tokens of the target question. In particular, the first token passed to the Decoder will always be *[SOS]*, then all the question's tokens, one at a time are fed to the network, until the *[EOS]* token is reached. In practice, we implemented a teacher forcing technique with a probability of 50%. Thus, at any time step, there will be a probability of 50% to choose as Decoder's next input token the actual target word or the previous predicted one.

In order to put together our trainable models, we just had to subclass the base *Seq2Seq* model and specify the combination of Encoder and Decoder, among the ones that we had implemented.

We built and trained three different models to address the Question Generation task:

- **Baseline**: Base Encoder - Base Decoder - Custom Attention
- **RefNet**: RefNet Encoder - Base Decoder - Bahdanau Attention
- **BERT for QG**: BERT Encoder - BERT Decoder - Bahdanau Attention

Everytime an Embedding Layer is cited in the above detailed explanation of our models, we are referring to the same implementation also employed for the *Question Answering* task, so it is initialized with the pre-trained vectors of 300 dimensions from GloVe.

4 Experimental setup and metrics

To train and test the models the same metrics for the Question Answering task were used. In addition to the standard metrics, for this specific task we decided to implement also the BLEU and perplexity score:

- *The Bilingual Evaluation Understudy (BLEU)* works by counting the match of n-grams in the candidate translation to n-grams in the reference text, where an unigram would be each token and a bigram comparison would be each word pair. The counting of matching n-grams is modified to ensure that it takes the occurrence of the words in the reference text into account, not rewarding a candidate translation that generates an abundance of reasonable words.
- *Perplexity* is the multiplicative inverse of the probability assigned to the train/test set by the language model, normalized by the number of words in the train/test set.

We implemented and setup the models accordingly to the corresponding original papers, in order to have a well-established benchmark. Then, on the basis of the previously described metrics, we tried to improve the models performance by tuning the hyperparameters and employing specific techniques.

The first architecture we implemented is the Baseline. The dimension of the Encoder and Decoder's hidden state is 512, which becomes 1024 in output since we used *Bi-LSTM* networks. The size of the batches more suitable for our available resources was 64. Furthermore, since the model was overfitting a lot on the training set and we saw degraded performance on the validation set, we decided to introduce a regularization layer. In particular, a dropout of 30% was introduced after the *LSTMs* layers. Since we wanted to use this model as a baseline, we took advantage of the Adam optimizer with the default parameters of PyTorch.

The RefNet model was the first architecture we developed to try to increase the performance w.r.t. the Baseline. The setup is similar to the one of the previous model, with few precise improvements. Since we know that *LSTMs* suffer from 'exploding gradients', we clipped the gradients during backpropagation to 1.0, we set the dropout to 30% and added a weight decay of 0.01 on the optimizer.

Finally, the latest architecture is the one which has BERT as Encoder layer. We took advantage of the same setup used in the QA task, since we obtained great results in that case.

5 Results and possible improvements

All the results are shown in table below [2]. As we can see, the results expressed by the metrics are acceptable but not that great. This is understandable since the QG task is extremely challenging w.r.t. the QA one. Indeed, here we ask to the model not only to locate the correct span of the target inside the context but also to generate a possible question in a non-extractive way. In addition, the *SQuAD* dataset is designed accurately for a extractive question answering task, with all the related problems.

Nevertheless, the results we obtained are not that far from the ones of the original papers from which we took the models.

	Training			**Validation**		
	F1 (%)	**BLEU** (%)	**Perplexity**	**F1** (%)	**BLEU** (%)	**Perplexity**
Baseline	39.5	21.9	25.5	21.5	16.1	563.0
RefNet	29.5	18.8	108.7	20.7	15.9	580.9
BERT	26.8	14.7	159.6	25.5	15.1	293.3

Table 2: Best results

The performance obtained on the training set by the Baseline model are much higher than the ones on the validation set, symptom that the dropout layer helped to reduce a lot the overfitting but was not completely effective. Despite this, the outcomes on the validation set are promising for a simple *Seq2Seq* model with a naive attention. The *SQuAD v1.1* dataset indeed is not ideal for a Question Generation task, since the context texts are huge and the models struggle in finding the correct span of text from which to take the words of the desired question. For this reason, we also made a naive preprocessing step to reduce the size of the context for each target question. In practice, the preprocessing algorithm splits each context in sentences and overwrite the original context with only the sentences in which the answer is present. The results with this freshly created dataset were computed only for the Baseline model, to understand if a more accurate preprocessing on the *SQuAD* dataset could be considered a possible improvements for this task. We obtained a *BLEU* score of 16.9 with an F1 of 24.8 on the validation set, demonstrating a clear improvement in the general performance.

The RefNet model is the direct improvement of the Baseline, featuring a well established and recognized attention, the Bahdanau Attention. The results on the validation set are similar to the Baseline ones, but we could clearly see how the model was overfitting much less on the training set. Moreover, while the Baseline architecture achieved the best outcomes at epoch 15, at the same time the RefNet was still improving, so maybe in few more epochs it would get even better than the Baseline on the validation set.

Finally, the table shows that the BERT model, although performing similar on the validation set compared to the previous models, succeed in lowering the perplexity and overfits a lot less while also obtaining the highest f1 score. Moreover, due to limited resources of computation, we were able to fine-tune the transformer only for 2 epochs, while the RNNs models were trained for 15 epochs, and that all the metrics were increasing, symptoms that more epochs would be needed and that we would for sure obtain higher results . In addition, the perplexity on the validation is almost half of the one of the previous models making this the best model.

As already said, the results were not astonishing but overall satisfactory, given the difficulty of the task. Those results can be certainly increased by employing state-of-the-art models and fancier techniques for the training phase.
Possible improvements could be related to the *BLEU* score for instance. The default implementation takes in input both multiple possible references and multiple predictions in order to choose the best match; this is because in tasks like this there is not just one possibility to generate the target text.

Due to the way the *SQuAD* dataset is formed, we didn't have multiple ground truths available for each question. Similarly we didn't make our models produce multiple predictions and to address that we will talk Beam search.
The Beam search is a typical technique used in text generation task which keeps track of k states rather than just one. It begins with k randomly generated states. At each step, all the successors of all k states are generated. If any one is a goal, the algorithm halts. Otherwise, it selects the k best successors from the complete list and repeats. It is very useful in text generation tasks because there may be many different ways to write a particular sentence without altering its meaning. Nonetheless, we decided to not implement the Beam search in our Decoder due to limited resources of computation. Indeed, generate all the successors of all k states at each step is an extremely expensive task.

References

[1] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. 2016. arXiv: 1409.0473 [cs.CL].

[2] Danqi Chen, Adam Fisch, Jason Weston, and Antoine Bordes. *Reading Wikipedia to Answer Open-Domain Questions*. 2017. arXiv: 1704.00051 [cs.CL].

[3] Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. *ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators*. 2020. arXiv: 2003.10555 [cs.CL].

[4] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: 1810.04805 [cs.CL].

[5] *GloVe: Global Vectors for Word Representation*. URL: https://nlp.stanford.edu/projects/glove/.

[6] *HuggingFace Tokenizers*. URL: https://huggingface.co/docs/tokenizers/python/latest/.

[7] *Huggingface webpage*. URL: https://huggingface.co.

[8] Preksha Nema, Akash Kumar Mohankumar, Mitesh M. Khapra, Balaji Vasan Srinivasan, and Balaraman Ravindran. *Let's Ask Again: Refine Network for Automatic Question Generation*. 2019. arXiv: 1909.05355 [cs.CL].

[9] *PyTorch webpage*. URL: https://pytorch.org/.

[10] Sam Schwager and John Solitario. *Question and Answering on SQuAD 2.0: BERT Is All You Need*. URL: https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1194/reports/default/15812785.pdf.

[11] *SQuAD dataset webpage*. URL: https://deepai.org/dataset/squad1-1-dev.

[12] *WandB webpage*. URL: https://wandb.ai/site.