Alma Mater Studiorum · University of Bologna

GoQU: A Tensorflow implementation of "Learning how to ask" model[1]

Natural Language Processing

Eric Rossetto (eric.rossetto@studio.unibo.it)
Salvatore Pisciotta (salvatore.pisciotta2@studio.unibo.it)
Tiberio Marras (tiberio.marras@studio.unibo.it)

July 6, 2022

[1]The code for the project is publicly available on GitHub

Contents

1 Executive summary

This paper aims to describe the working pipeline adopted for the implementation of our version of a Neural Network that tries to tackle the **Question Generation**(QG) problem on the SQuAD v1.1 dataset[15]. More specifically, our purpose is to develop a custom model using recent techniques able to generate human readable questions taking inspiration from the most relevant results in literature. Several steps were needed to reach appreciable performances and create questions that can be at least compared with the ones created by humans, even though to some extent. For training and testing we used the handy SQuAD dataset (which is generally used to train question-answering models), by extracting and pre-processing question-text pairs from it. In order to represent terms, our idea is to use pre-trained GloVe embeddings[14], to then build the embedding matrices using the processed question-answer pairs. The custom model we implemented is based on the work by Du et al in the paper "Learning to Ask: Neural Question Generation for Reading Comprehension" [4]. Their solution for QG task consists on using a encoder-decoder model, in a similar manner to what has been done in Seq2Seq[16]. Both the encoder and the decoder exploits the capabilities of the LSTMs[7] while implementing an attention head [17]. For the evaluation we based our analysis on several metrics like **ROUGE** [11] and **METEOR** [2], which compute a score comparing the true questions with the predicted ones. In addition, following the reasoning of Nema et al. in the paper "Towards a Better Metric for Evaluating Question Generation Systems" [12], we implemented the **Answerability** metric of a question that, combined with the already mentioned metrics, creates two question-specific variations of the aforementioned scoring functions called Q-ROUGE and Q-METEOR.

2 Introduction

Question generation (QG) is an important Natural Language Processing (NLP) task where given as input a text it is possible to create a natural question related to it. There are many possible applications regarding these QG technologies; some of them include education, e.g, for the evaluation of the knowledge and comprehension abilities; or in some advanced chat-bot systems, where correct questions are been provided to the user can be crucial in order to obtain information about a particular situation and to improve users' immersion; finally also in the entertainment field QG finds an interesting application in quiz games.

The challenge can be seen as a specific case of Natural Language Generation (NLG), where there is not only the necessity to generate grammatically and semantically correct questions, but also that the answers of produced questions must be present in the provided input text. In last years there has been a shift in QG tasks from *rule-based* approaches to the use of neural deep learning techniques, introducing the concept of **Neural Question Generation** (NQG). The objective of this project is to provide a neural network (NN) model able to generate questions (possibly) coherent with the provided input and with a good level of semantic and grammatical correctness.

Once we created the model able to produce human readable questions we faced the task of how to evaluate them.

3 Background

Generally, QG is tackled by rule-based methods that usually rely on burdensome hand-crafted rules. But lastly, in order to automate the generation process, the use of neural network-based methods, completely data-driven and trainable in an end-to-end fashion, became the most popular solution. In particular the most automated method reported thus so far utilizes RNNs as sequence transduction (Seq2Seq) model to generate questions from sentences or paragraphs. As the audience maybe already knows the most successful variant of a RNNs is the Long Short-Term Memory (LSTM) network. The Seq2Seq architecture is usually composed of an encoder and a decoder, which reads the input from left to right. The encoder takes the input and produces its own hidden representation, while the decoder takes the vectors produced by the encoder and tries to create an output token by token by using the information encoded by the encoder. An attention mechanism allows the decoder to deeply analyze the input sequence selectively. In this study, inspired by the work of Du e al. [4], we tried to automatically generate questions by developing a model using the Tensorflow [1] and Keras [3] frameworks, this is also motivated by the fact that currently we did not found any implementation of it using the mentioned frameworks in the literature.

3.1 Problem Definition

The goal of QG is to generate a valid and fluent question according to a given context and the target answer. In fact, given an input sentence x, our goal is to generate a natural question y related to the information given in the input. Additionally, we refer to y as a sequence of an

arbitrary length while the length of the input sequence is usually fixed and we call it M. As usually done in NLP, x could be represented as a sequence of tokens $[x_1, \ldots, x_M]$. The QG task is defined as finding $\hat{y}$, such that:

$$\hat{y} = \arg\max_{y} P(y|x)$$

where $P(y|x)$ is the conditional log-likelihood of the predicted question sequence y, given the sentence x[4].

3.2 Related Works

For the implementation of the model we based our work mainly on the paper "Learning to Ask: Neural Question Generation for Reading Comprehension"[4]. The paper introduces a model that does not rely on hand-crafted rules or a sophisticated NLP pipeline; but it is instead trainable end-to-end via sequence-to-sequence learning. Other papers we consulted are "Question Generation Through Transfer Learning"[10] and "Automatic question generation model based on deep learning approach"[6]. For the metrics is noteworthy the paper "Towards a Better Metric for Evaluating Question Generation Systems" [12] in particular for the implementation of Q-metrics.

3.3 Dataset

The Stanford Question Answering Dataset (SQuAD) is a collection of question-answer pairs derived from Wikipedia articles. SQuAD v1.1 contains 100.000+ question-answer pairs on 500+ Wikipedia articles. This dataset is heterogeneous and is widely used in machine learning comprehension tasks. An instance of SQuAD dataset has:

- **ID**,
- **Title**, that is the title of the article,
- **Context**, the paragraph of the article,
- **Question**,
- **Answer Start**, an index representing the starting character of the answer in the context,
- **Answer**, the text containing the actual answer to the Question.

We decided, coherently with the problem to solve, to drop the **ID** and the **Title** columns since they are not useful data for tackling our task. From the remaining data, the **Question** column is our target, while the other columns are selected as data to pre-process in order to create the input for our neural network. The dataset we used for this project contains 87040 samples.

4 Data Preparation

4.1 Pre-processing Operations

The class `Dataset` is in charge of the loading the dataset and to perform the pre-processing steps. Firstly we take the context that we have in input and, throughout a method, extract all the sentences that contain the answer in the paragraph. In this way, in each sample we will have $(context, question)$ pairs. Hereinafter we will refer to context as the sentence that contains the answer. It is possible that the answer falls in more than one sentence and in that case we simply created a several pairs where the question remains the same for different contexts. Before tokenizing both the context and question sentences using the Keras Tokenizer [3] we implemented some pre-processing steps:

1. adding a space between a word and the punctuation following it,
2. keeping characters that belong to the regular expression [^a-zA-Z0-9?.!,¿]+ and replacing the others with a blank space,
3. adding a *start* and an *end* token (<sos> and <eos> respectively) to the sentences.

Once pre-processed we split the dataset in the training set (70% of data), validation set (20%) and test set (10%).

5 Model Description

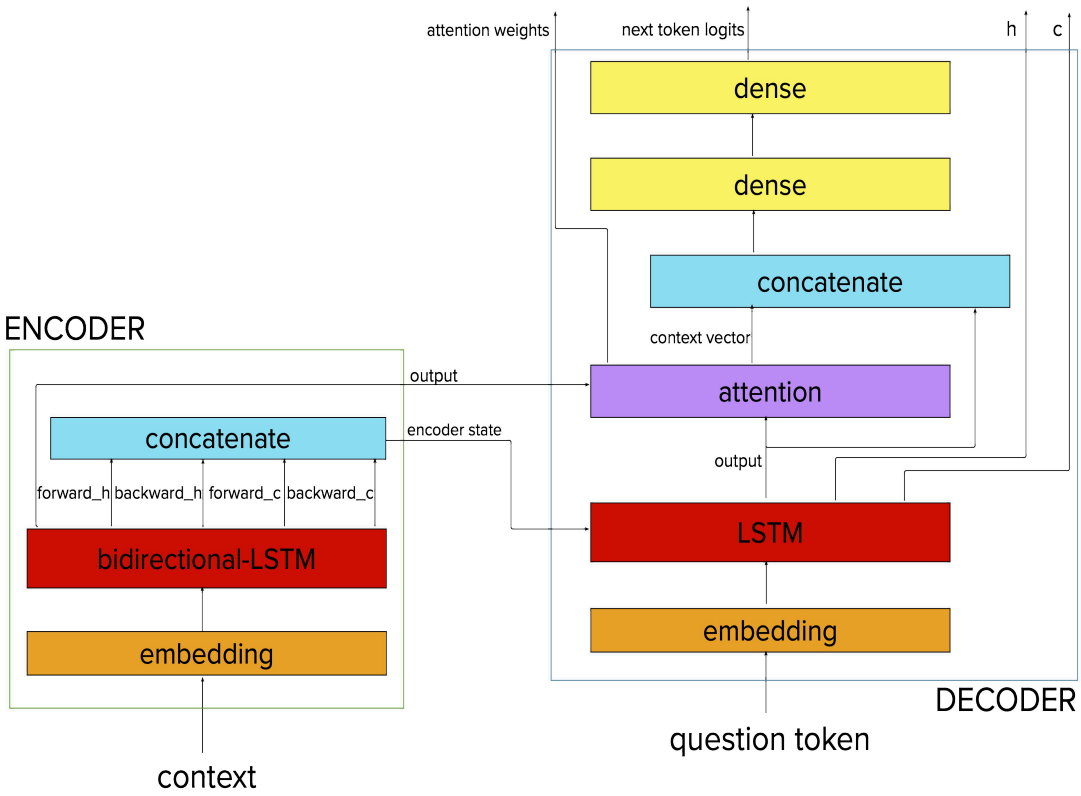
The model is a encoder-decoder neural network with an attention head.

5.1 Implementation Details

As already mentioned in section 4.1 we decided to define two tokenizers, one for the questions and one for the sentences, so we have defined two different vocabularies $\mathcal{U}$ and $\mathcal{V}$ fitted on the training and validation sets. To internally represents the terms in the network we used pre-trained GloVe word-embeddings. A key step is to define a procedure to build an embedding matrix $M \in \mathbb{R}^{|V| \times d}$, where V is the set containing all the terms present in the corpus and d is the dimension associated to the embedding vectors. The vocabulary provided by GloVe contains 400k unique words and a handy associated embedding. There may be words that are not present in such dictionary but that appear in our dataset. Such words, called *Out-of-Vocabulary* (OOV) words, need to be handled carefully. Indeed, since the pre-trained embedding model has associated a vector to each word in its vocabulary, we have decided to assign to each OOV word a randomly generated vector of dimension d sampled from a uniform distribution ranging in $[-0.5, 0.5]$.

The class `Trainer` represents our model and is what has been used on the training phase. It exploits the functionalities of the sub-classing API of Tensorflow and is instantiated with the setting parameters, the encoder and the decoder, which by the way are implemented by using the Functional API. In order to train it correctly we had to implement both the train step called at

training time, containing the forward and backward pass, and the test step, called at validation time, containing solely the forward pass. A proper loss was also defined in 6.1 along with the performance metrics used to observe during the tuning phase.



Architecture of the model

5.2 Encoder

The encoder model is composed of following layers:

- **Input layer**: takes the context sentences as inputs;
- **Embedding layer**: converts the positive integers (tokens) into a dense vectors representation of fixed size;
- **Bidirectional LSTM**: takes as input the embedding vector produced by the previous layer. Since both the directions of are enabled it computes the actual output along with the hidden state and cell state of the forward and backwards LSTMs;
- **Concatenate layer**: merges the hidden states and the cell states of the two LSTM layers and returns a state composed only of one hidden state and a cell state;

The encoder will return an encoded representation of the input and the last state of the LSTMs to be eventually be passed to the decoder.

5.3 Decoder

The decoder is composed of following layers:

- **Input layer**: takes as input:
  1. **Token**: the last token generated of shape, namely the token obtained in the previous time step of the decoder. At start time it is equal to the start-of-sequence token;
  2. **Encoder Output**: this is the representation produced by the Encoder;
- **Embedding layer**: takes as input the output of input layer and produces as before a dense vector representation of fixed size;
- **LSTM layer**: returns the output of each time step. Its state is initialized with the state of the encoder at the start of decoding and eventually it updates it with the decoder state at each time step;
- **Bahdanau Attention layer**: takes as input the output of the LSTM layer and the output of the encoder. It outputs a context vector c_t and the weights of the attention computed along the context length;
- **Concatenate layer**: concatenates the output of the LSTM layer and the context vector produced by the attention layer;
- **Dense layer**: takes as input the output of concatenate and produces an attention vector a_t, we chose *arctan* as activation function;
- **Dense layer**: takes as input the attention vector a_t and produces a vector of logits, which are unscaled probabilities over the elements of the dictionary.

The next word for each sentence in the batch will be sampled by the distribution produced by the last layer.

Partially following the indication of the paper and some of our intuitions, in order to obtain the best possible configuration of the model, we have set LSTM layers of the encoder and the decoder at 512 units, with an L2 regularization value of 0.001 and some dropout layers in between the LSTM and Dense layers with a dropout rate of 0.3; both of them were inserted in order to deal with possible over-fitting situations.

6 Training

6.1 Loss Function

Du et al. work shows that the conditional probability could be factorized in:

$$P(y|x) = \prod_{t=1}^{|y|} P(y_t|x, y_{<t})$$

where the probability of each y_t is predicted based on all the words that have been generated upon time t, namely $y_{<t}$.

This means that given a training corpus of sentence-question pairs $\mathcal{S} = \{(x^{(i)}, y^{(i)})\}_{i=1}^{N}$, the objective is to minimize the negative log-likelihood:

$$\begin{aligned}\mathcal{L} &= -\sum_{i=1}^{N} \log P(y^{(i)}|x^{(i)}; \theta) \\ &= -\sum_{i=1}^{N} \sum_{j=1}^{|y^{(i)}|} \log P(y_j^{(i)}|x^{(i)}, y_{<j}^{(i)}; \theta)\end{aligned}$$

We parameterize the probability of decoding each word y_j by using a LSTM:

$$P(y_j|y_{<j}, s) = W_s \tanh(W_t[h_t; c_t])$$

where h_t is the hidden state and c_t is the context vector, both at time step t. This means that we can express our objective as the minimization of the cross-entropy loss. In order to obtain more trustful results we suppressed the loss values when it encountered a padding tag.

6.2 Training Parameters

During our experimentation we tried different configurations of parameters for the training, in particular we started using the parameters used by the authors of the paper that trained the model using a Stochastic Gradient Descent optimizer (SGD). They set a learning rate equal to 1.0 with a scheduling that halves it at 0.5 after 8 epochs, the model was trained for 15 epochs but in our case this training configuration was too aggressive and led to questionable results. This could be motivated by the fact that the model configuration is slightly different and that we trained the model on far less samples. Indeed, the loss after each epoch had an exponential increasing trend that did not stop even after some epochs. For this reasons we decided to change the optimizer from SGD to Adam[9] and using far smaller learning rate values that do not allow the loss to have an exponential trend.

In order to find the best parameters for the training we decided to execute a *Keras Tuner*[13] on the model, this means that we looked for the best model that came out after letting the Keras Tuner trying different combinations of hyper-parameters such as regularizer impact, dropout rate and optimizer's learning rate.

6.3 Training Metrics

For the training phase of the project we used two metrics:

- **perplexity**: is a measurement of how well a probability model predicts a sample. It is easily deducted by the cross-entropy loss:
$$p = e^{H(p,q)}$$
where $H(p,q)$ is the loss defined in 6.1 between p, the true labels, and q, the logits produced by the decoder;
- **accuracy**: is the classic metric for evaluating a classification model. It is the number of right predictions divided by the number of total predictions. In our case the comparison is between the predicted question and the right one.

In particular, talking about the accuracy, we monitored the standard **accuracy** and the so called **masked accuracy**. The **masked accuracy** the accuracy that, through a masking mechanism, does not take into account the presence of the '<pad>' tag for the calculation in the predicted question and in the true question.

6.4 Training History

We trained for several trials and we experienced that our model tends to suffer from a over-fitting phenomena when the train and validation loss reach an approximate value of 5.0. In particular, the steady decay of the training loss continues while the validation loss starts to increase, creating an evident gap between the two. In 7.3 we will discuss about the consequences of this situation.

The figures below show the loss curves of two training situations, the figure 6.2 represents one of our first runs where it can be seen the over-fitting situation in 30 epochs described before. In 6.1 we have a training stopped at 20 epochs, with a smaller learning rate. The results of these runs have been reported in 6.1 with the first row representing the values in the over-fitting situation and the second one representing the values of the training without over-fitting.

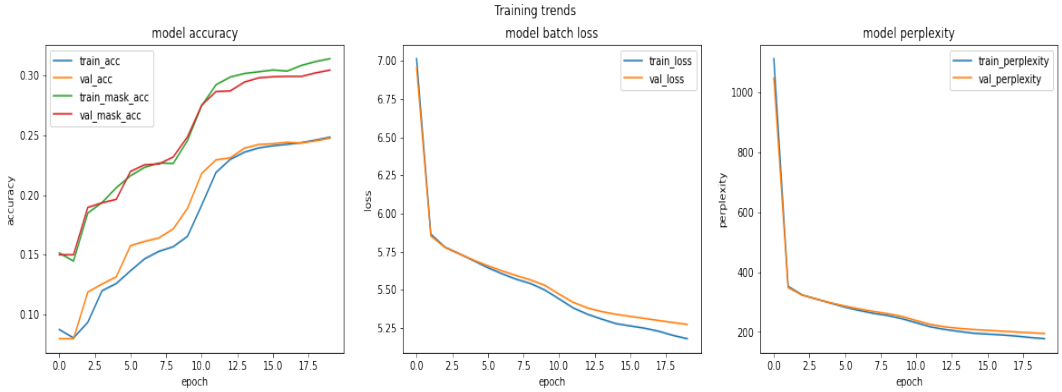


Figure 6.1: Training curves without over-fitting.

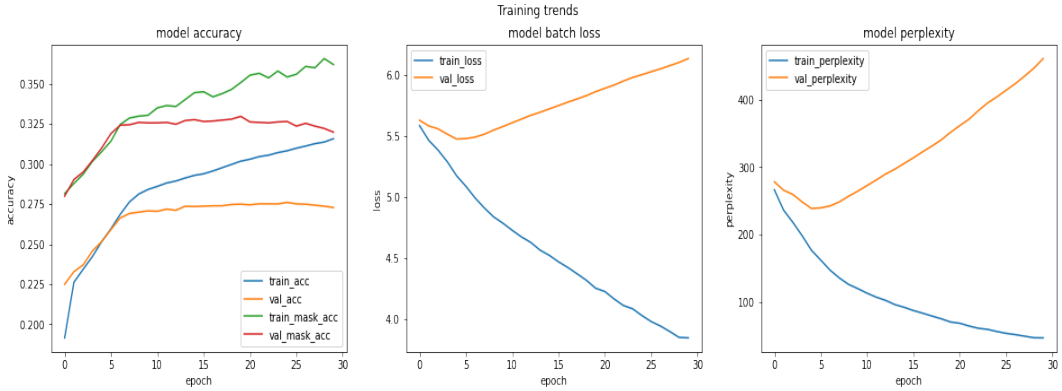


Figure 6.2: Training curves in case of over-fitting situation.

Epochs	Train_loss	Train_mask_acc	Perplex	Val_loss	Val perplex	Mask_val_acc
30	3.8496	0.3621	46.9757	6.1328	460.7069	0.3199
20	4.9818	0.3352	145.7350	5.1912	179.6878	0.3255

Table 6.1: Table

7 Evaluation

The evaluation is implemented in the `Evaluator` class. The class constructor takes as input the trained model `Trainer` and the tokenizer fitted on the dataset as mentioned earlier, finally it initializes the ROUGE and the METEOR metrics. The evaluate function iterates throughout the input, generating the questions calling the `predict_step` and computes the metrics comparing them with the true questions.

7.1 Inference

In order to obtain the predictions we implemented in the `Evaluator` class a `predict_step`, which computes the questions starting from the context sentence provided a maximum output length. That is, we encode the input and run the encoder to produce a token at each time step, while doing that we have to stop generation of new tokens whenever the decoder generates the end-of-sentence tag and assign to next token in the sequence padding values; eventually, in order to deal with unknown words, which they are still possible since the two vocabularies have not been fitted in the test set, we substitute each unknown token by the one in the context that have received the highest attention score. From the logits produced, we extract the next token by using a *freezing function* which is parameterized by a temperature value ranging in $t \in [0, 1]$, when $t \to 0$ it behaves like a greedy decoding achieving a greater grammatical correctness whereas when $t \to 1$ it achieves more variety by directly scaling the results.

7.2 Metrics

For the evaluation we used two metrics:

- **ROUGE (Recall-Oriented Understudy for Gisting Evaluation)**: is a metric generally used for evaluating automatic summarization and machine translation. ROUGE-N measures the number of matching '*n-grams*' between our model-generated text and a '*reference*'. In particular for the ROUGE we calculate 3 different measures:
  - **recall**: counts the number of overlapping n-grams found in both the model output and reference — then divides this number by the total number of n-grams in the reference.

$$Recall = \frac{\#\text{ ngrams} \in \{\text{model} \cup \text{reference}\}}{\#\text{ n-grams} \in \text{reference}}$$

  - **precision**: same as recall, but rather than dividing by the reference n-gram count, we divide by the model n-gram count.

$$Precision = \frac{\#\text{ ngrams} \in \{\text{model} \cup \text{reference}\}}{\#\text{ n-grams} \in \text{model}}$$

  - **F1-score**: the harmonic mean of precision and recall.

$$F1\_score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

  ROUGE-L measures the longest common sub-sequence (LCS) between our model output and reference. All this means is that we count the longest sequence of tokens that is shared between both. We apply our recall and precision calculations like for ROUGE-N, but this time we replace the match with LCS.

- **METEOR (Metric for Evaluation of Translation with Explicit ORdering)**: is based on the harmonic mean of uni gram precision and recall, with recall weighted higher than precision. It also has some features that are not present in BLEU metric, such as stemming and synonymy matching, along with the standard exact word matching.

Both the metrics described are imported from Hugging Face libraries [18][11][2].

Since the model produces questions that can be seen as a particular category of text there was also the need to evaluate the results with some more specific metrics. In order to do this, based on the work by Nema et al. [12], we implemented **Q-ROUGE** and **Q-METEOR** metrics. These metrics are a modification of existing metrics that do not correlate well with human judgments about answerability [12]. They are computed as a weighted sum between the ROUGE and METEOR values respectively and the **answerability** value. The answerability is a value that is based on the comparison between the generated question and gold-standard question, in our case the prediction. The value is based on the matching between the two question of four kinds of words:

- **Relevant Words**: words of the question computed using the keyword extraction function of spaCy [8];
- **Named Entity Recognition (NER)**: the entities, computed for every question using also in this case the keyword extraction function of spaCy;
- **Question Words**: the **wh question words**:what,when,how,which,where,whom,who,why,whose with the addition by us of the '?' character in order to reward the more structured questions;
- **Functional Words**: taking as reference to the list of English functional words (or stop words) defined in spaCy.

The answerability is then computed as:

$$Answerability = 2 \cdot \frac{P_{avg} \cdot R_{avg}}{P_{avg} + R_{avg}}$$

where:

$$P_{avg} = \sum_i w_i \cdot \frac{c(S_i)}{|l_i|} \qquad R_{avg} = \sum_i w_i \cdot \frac{c(S_i)}{|l_i|}$$

with $i \in \{r, n, q, f\}$ representing the already mentioned groups of words, $c(S_i)$ is the count of matching words, belonging to one of the four categories, between generated and reference questions and $|l_i|, |r_i|$ are the total number of words of generated and reference questions respectively for each group of terms.
There was not a clearly indication in the paper of what could be the optimal value for the set of weights, this because they strictly depend on the dataset and the type of question to generate. We decided to keep, by default, all the weights at 0.25 in order to obtain a more balanced value of answerability. To deal with the possible absence of words belonging to one or more groups, the correspondent weight w_i is imposed at 0 and the functional words weight assumes more importance because it is calculated as the difference between 1 and the sum of all the other weights.

Generally the answerability alone is not so explicable of how a question is well posed, so we can combine this answerability score with any existing metric obtaining the Q-variant of the metric, calculated as follow:

$$Q\_metric = \delta \cdot Answerability + (1 - \delta) \cdot \text{metric\_value}$$

With parameter δ representing the weight of the sum between the two values. Also in this case the assigned value is not explicitly given in the paper so we can assume is dependent, again, from the task. In our case we gave it a default value of 0.7.

7.3 Results

Table 7.1 below above reported are produced by the model that does not encounter, as explained in 6.4, an overfitting situation during the training.

METEOR	ROUGE-1 F1	ROUGE-2 F1	ROUGE-L F1	QROUGE-L F1	QMETEOR
0.5084	0.1328	0.0158	0.1251	0.2703	0.3814

Table 7.1: Table of ROUGE and METEOR metrics with their Q-variants metrics value for test set

7.4 Analysis of Results

By looking at the results obtained we can observe that the model get pretty good at predicting unigram words in the questions, as reported by METEOR. On the contrary, when looking at the different ROUGE scores we can say that there an issue at predicting bi-grams or sub-sequences of words that are correlated with the natural language, thus lacking of sense. Still, the Q-variants of the metrics that the answerability helps in reporting higher values, this means that words that are correlated with what humans consider a question are predicted correctly but still it misses in putting them in order or in way that assign the right semantic and grammatical meaning to the questions.

8 Conclusions and Future Works

To conclude, we are quite satisfied by the work done as well as the pipeline followed in order to obtain the results produced, even though they are far from being competitive with the SOtA techniques from the literature. For educational and for fun purposes we can say that we have reached a decent implementation of a custom model for a challenging task like question generation. The first future task is to investigate in a deeper way the evident problems of over-fitting and implement some solution to avoid it, or at least mitigate it. This is because the classical mentioned techniques do not provide satisfactory effects. The following step could be to augment each word by adding the corresponding POS and the NER tag. Another solution could be using the entire paragraphs instead of only the sentences that contain the answer, still [4] shown that this will lead to similar results. Furthermore, an improvement in the performances could be achieved implementing a Beam Search[5] method, to achieve more variety in the predictions. Finally, it can be a good idea also to use different metrics in order to analyze in a more detailed way the results obtained.

Bibliography

[1] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[2] Satanjeev Banerjee and Alon Lavie. "METEOR: An Automatic Metric for MT Evaluation with Improved Correlation with Human Judgments". In: *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*. Ann Arbor, Michigan: Association for Computational Linguistics, June 2005, pp. 65–72. URL: https://aclanthology.org/W05-0909.

[3] François Chollet et al. *Keras*. https://keras.io. 2015.

[4] Xinya Du, Junru Shao, and Claire Cardie. *Learning to Ask: Neural Question Generation for Reading Comprehension*. 2017. DOI: 10.48550/ARXIV.1705.00106. URL: https://arxiv.org/abs/1705.00106.

[5] Markus Freitag and Yaser Al-Onaizan. "Beam Search Strategies for Neural Machine Translation". In: *Proceedings of the First Workshop on Neural Machine Translation*. Association for Computational Linguistics, 2017. DOI: 10.18653/v1/w17-3207. URL: https://doi.org/10.18653%2Fv1%2Fw17-3207.

[6] Salma Doma Hala Abdel-Galil Mai Mokhtar. *Automatic question generation model based on deep learning approach*. URL: https://ijicis.journals.ekb.eg/article_189182_6a0cd1a1a15c33b47fa9cea92863883e.pdf.

[7] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

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

[9] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. DOI: 10.48550/ARXIV.1412.6980. URL: https://arxiv.org/abs/1412.6980.

[10] Yin-Hsiang Liao Jia-Ling Koh. *Question Generation Through Transfer Learning*. URL: https://link.springer.com/chapter/10.1007/978-3-030-55789-8_1.

[11] Chin-Yew Lin. "ROUGE: A Package for Automatic Evaluation of Summaries". In: *Text Summarization Branches Out*. Barcelona, Spain: Association for Computational Linguistics, July 2004, pp. 74–81. URL: https://www.aclweb.org/anthology/W04-1013.

[12] Preksha Nema and Mitesh M. Khapra. "Towards a Better Metric for Evaluating Question Generation Systems". In: (2018). DOI: 10.48550/ARXIV.1808.10192. URL: https://arxiv.org/abs/1808.10192.

[13] Tom O'Malley, Elie Bursztein, James Long, François Chollet, Haifeng Jin, Luca Invernizzi, et al. *Keras Tuner*. https://github.com/keras-team/keras-tuner. 2019.

[14] Jeffrey Pennington, Richard Socher, and Christopher Manning. "GloVe: Global Vectors for Word Representation". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1532–1543. DOI: 10.3115/v1/D14-1162. URL: https://aclanthology.org/D14-1162.

[15] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. "SQuAD: 100,000+ Questions for Machine Comprehension of Text". In: *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Austin, Texas: Association for Computational Linguistics, Nov. 2016, pp. 2383–2392. DOI: 10.18653/v1/D16-1264. URL: https://aclanthology.org/D16-1264.

[16] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. *Sequence to Sequence Learning with Neural Networks*. 2014. DOI: 10.48550/ARXIV.1409.3215. URL: https://arxiv.org/abs/1409.3215.

[17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. *Attention Is All You Need*. 2017. DOI: 10.48550/ARXIV.1706.03762. URL: https://arxiv.org/abs/1706.03762.

[18] Thomas Wolf et al. "HuggingFace's Transformers: State-of-the-art Natural Language Processing". In: *CoRR* abs/1910.03771 (2019). arXiv: 1910.03771. URL: http://arxiv.org/abs/1910.03771.