QACNNnet - Convolutional neural networks and self-attention for question answering

Gaetano Signorelli and Daniele Sirocchi
(Note: there is one less member with respect to the team that developed the assignments.)

Abstract. Question answering (QA) is one of the most exciting tasks in Natural Language Processing, and there is a lot of active research around it. The very first approaches used for solving QA problems were based on a common technique in this field, which consists in the use of recurrent neural networks (RNN) combined with attention. These approaches work pretty well, but they are typically slow at both training and testing times, due to the sequential nature of RNNs. A different approach that has gained in popularity is based on transformers; it seems to be very effective also on capturing long-term dependencies on text sequences, and it helps in overcoming the slowness problem. Finally, there exists a third approach that we decided to study and implement. It is based on the use of convolutional neural networks (CNN) in place of RNNs, coupled with self-attention. We found it to be very interesting, intrigued by experimenting the power of CNNs in finding local features with inputs different from images.

1 Introduction

1.1 Problem

Question answering represents a challenging task for natural language processing and, more generally, for artificial intelligence. The machine has to extract information from texts to answer a question given by the user, therefore text comprehension is required, which is a hard task.

The aim of the discussed work is trying to solve instances of this task in a simplified form: the chosen dataset, **SQuAD version 1.1**, contains texts that consist of relatively short passages extracted from longer paragraphs, and the only questions allowed are those whose answers are directly a part of the context (more in the "*Dataset*" section). Furthermore, all the questions have at least one answer belonging to the known context, making the problem much easier to tackle. If on one hand it would be more interesting to work with a more challenging dataset, such as the advanced SQuAD 2.0, where there are questions with no answers and other questions purposely structured so to mislead most of the neural models, on the other hand solving the task on the basic dataset still deserves attention, since finding the most performing strategy for a simpler task can be beneficial when it's time to pass to the more difficult one.

1.2 Previous works

Since its release in 2016, there have been numerous attempts to solve the question answering task on the examined dataset, starting from the baseline model proposed by the authors of the dataset itself [1]. Most of those attempts revolved around the usage of recurrent neural networks, such as in the *Match-LSTM* [2] and other initial tries that scored quite well. Results improved in two ways, first of all by creating models that became far more complex than the original one, for example the first *BiDAF* implementation [3], and secondly by introducing attention mechanisms, as implemented inside the DCN (*Dynamic Coattention Networks*) [4]. Thanks to these new evolving approaches, the metrics used to evaluate models' performances (F1 score and EM – Exact Match – score, see "*Metrics*" section later) produced better results until scores almost reached human ones. One of the most relevant results with an RNN approach has been achieved enhancing *BiDAF* with self-attention and *ELMo*'s embeddings [5].

Nowadays, new techniques have appeared, performing astonishingly well even on the more sophisticated *SQuAD* dataset 2.0, with higher scores with respect to human capabilities. Many of them are ensembles of models, but not only, as it is the case of the "*Retrospective Reader for Machine Reading Comprehension*" [6], which puts into the scene a lot of complex blocks (e.g., several transformers).

1.3 Selected choice

The proposed approach that has been implemented, tested and that will be discussed in the following sections is based on the **QANet** exposed in [7]. It is the first high scoring model that does not rely on recurrent neural models, but it is instead built on series of **convolutional layers** and **self-attention layers** used to catch, respectively, local and global interactions. Compared to the previous mentioned works, this one shows a double advantage. Firstly, it has slightly better performances than the RNNs models and it is definitely faster to train and run [8], making it feasible also for real-time applications. As a second point, despite having lower scores, it is easier to implement and test than the modern complex approaches, which need a high level of precision in model's design and implementation, combined with longer training times, in order to get their remarkable results [6]. In other words, the *QANet* represents a very good middle ground between a stable, reliable model with good performances and a not too delicate one to build and train on local machines with a limited computing power.

2 Dataset

As already mentioned, the central task revolves around the *SQuAD* dataset in its 1.1 version. It is absolutely the most common dataset for training models that try to tackle the question answering problem, even though it is by far not the most difficult one, due mainly to the answer always being part of the text and to the fact that it is always known in which context an answer should be

looked for [1]. Except for these points, this dataset remains reasonable from many points of view and it is, in a sense, the reference point. In this section it will follow a discussion of the analysis that have been conducted and the executed pre-processing steps.

2.1 Analysis

The dataset, officially released in *JavaScript Object Notation* (JSON), has been inspected both in a direct way (exploring the data methodically) and by statistical means. The following properties and peculiarities have emerged:

1. Structure: the dataset is made up of **442 paragraphs**, each containing a series of contexts (sub-paragraphs) extracted from it; a set of questions is associated to each context, each one with a specific identifier; finally, each question is followed by a list of possible valid answers, each with an index representing the starting character with respect to context's text.
2. There are **107.7K samples** (unique question-answer pairs).
3. Each question can have more than one answer.
4. Answers are always part of the context, so that they can be uniquely identified with a **span delimited by a start-end pair of indices**.
5. The language of all the paragraphs is **English**, but there are different characters coming from other languages (mostly ideograms) along with several symbols belonging to mathematical notations or others.
6. There are **6000+ unique words** inside the dataset.
7. There are around **1300 unique characters**, with the vast majority appearing less than 100 times in the whole dataset.

2.2 Preprocessing

To preprocess the entire dataset at best and build a dataframe to train the model on, a pipeline has been constructed, with a list of operations to adjust the text in a more convenient form for the network:

1. Text has been set to **lower case**.
2. Extra spaces and tabulations have been removed (**strip**).
3. **Special characters and punctuation have been removed** to reduce text's size and cleaning it with not necessary information. This operation has not been applied in all the tests and it has been excluded in the last ones for more precise results.
4. Text has been **lemmatized** to reduce the final vocabulary size. This step has been accomplished with the help of the *"nltk toolkit*" [9], which offers a set of utilities for text preprocessing, such as, exactly, a trained lemmatizer that also considers words' pos-tagging for better results. Lemmatization has not been included in all the tests and it has not showed any evident benefit: model's generalization capabilities are stronger without lemmatization.
5. Text has been split into a **sequence of tokens**, once again with the "*nltk*" library, using the "*WordNet*" tool, trained to do this job in a smart way.
6. No tokens have been removed (e.g., *stopwords*), to avoid losing important information and possibly deleting an entire answer.
7. The given character's starting index of each answer has been mapped into the correspondent **token's index** (indicating the position of the first answer's token inside context's tokens). This information has been stored into the dataframe together with the **end index** of the same answer.
8. Two tokenizers have been built, each one associating a unique integer respectively to **unique words and characters** considering all the texts (contexts and queries). Particularly, those words that owned a specific embedding inside **GloVe**'s embeddings dictionary have been taken into account, while the others have been mapped into an extra token value, called **<UNK>**. Finally, all the texts have been tokenized using the two tokenizers.
9. Samples having a context that exceeded a certain threshold have been **discarded**, to save memory and remove rare outliers. Depending on the removal of special symbols and punctuation, the average number of contexts' tokens varied from 250 (keeping all the characters) to 125 (with characters deletion); consequently, in the first case the **threshold has been set to 400 tokens** (because there are exceptionally long contexts), and 250 in the latter. **Padding** has been added to all contexts to reach the value indicated by the threshold.
10. Another limit regarding questions and answers has been set to 30, since they showed an average length of 10 tokens, with some longer exceptions also in this case. All of them being shorter than this limit have been padded, while the longer ones have been truncated to fit the desired size. The same policy has been applied to characters, submitted to **padding and truncation** with a threshold set to 16.
11. All different triplets *context-query-answer* have been stored as different samples, meaning that a question with multiple answers is repeated as many times as the number of those answers, that appear separately.

Samples have been split into **training and validation sets**. The second one represents around 10% of the entire dataset and it is selected in a way that keeps separated into different sets contexts belonging to different paragraphs. The test set is not extracted from the given dataset.

3 Model

Figure 1 depicts an overview of the model's architecture that has been implemented.

It is possible to observe that the network is basically composed by five different blocks:

1. Input embedding
2. Embedding encoder
3. Context-query-attention
4. Model encoder
5. Output layer

In the following sections, these blocks will be discussed in detail.

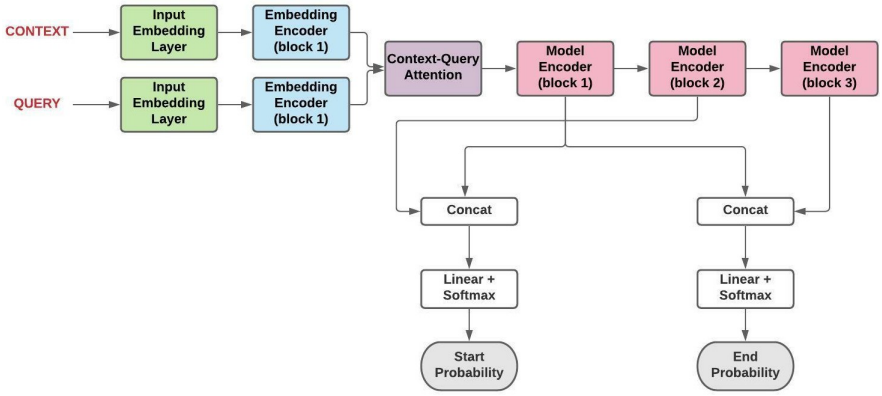

Fig. 1. General model's architecture

3.1 Input embedding

The input embedding layer is the very first block of the network, and it is responsible for converting the tokenized input into a vectorized representation, as depicted in **Figure 2**.

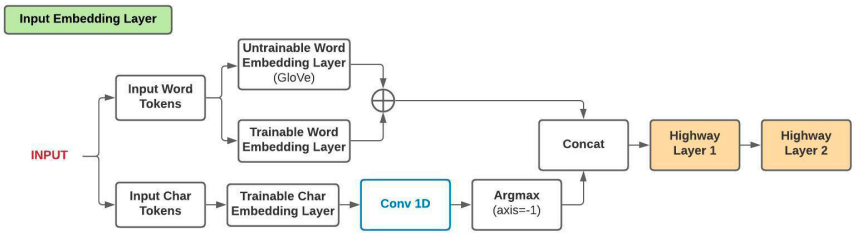

Fig. 2. Architecture of the input embedding layer

The **word embedding** is obtained by summing together the output of two different embedding layers:

- **Untrainable Embedding Layer (GloVe)**: this is a standard embedding layer whose weights are directly taken from *GloVe* [10] with a chosen dimension equal to 300. These weights are not trainable.
- **Trainable Embedding Layer**: this is another embedding layer which is responsible for learning just the <*UNK*> token. Its vector value is randomly initialized, and then updated during the training.

The **character embedding** is instead obtained as explained in [7]: the input is passed sequentially through an embedding layer, a convolution and then indices of the maximum values are extracted. These passages are required for obtaining the final characters' representation. This technique seems to be very effective for capturing patterns among nearby characters thanks to the convolutional operation.

Successively, as shown in **Figure 2**, the two branches are merged together with a concatenation operation, and passed via two highway layers.

As mentioned in the highway network paper [11], the goal of this block is to ease the problem of training deep networks. The main idea comes from the *Long short-term memory network* (LSTM), and its use of a **gating function** that allows to manipulate the signal in a way that makes possible to train deeper models.

Figure 6 shows how the structure of this layer has been implemented.

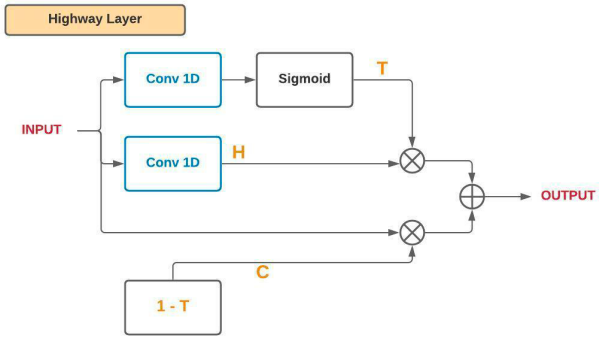

Fig. 6. Highway layer's architecture

There are 3 main branches:

- **H:** a main transformation to apply to the input signal. For the case of this implementation, the choice has fallen on a convolution operation.
- **T:** a non-linear transformation, called the **transform gate**.
- **C:** a non-linear transformation, called the **carry gate**. This is obtained as $(1 - T)$.

When T=0, then the input is passed directly as the output, and this creates something known as **information highway**. This is the reason why the layer contains the word *Highway* in its name.

Whereas, when T=1, just the non-linear transformed input is passed as the output.

Thanks to the sigmoid function that oscillates between 0 and 1, during the training the network can learn to adaptively pass the input transformed by means of H, or just pass the original input to the next layer.

This layer has been empirically tested by the authors, and the results show that thanks to its structure it is possible to train deeper networks without stalling.

3.2 Embedding encoder

The encoder block is made up of a sequential series of **encoding layers**, whose aim is to catch as many features as possible to give the best input tensors' representation. Encoders are used multiple times across the network and particularly in two different steps:

- To encode the embedding of both contexts and queries, extracting their features;
- To build a final encoding for the model, to be used by the last block in order to make sensible predictions.

For each of the two tasks, the same kind of encoding layer has been adopted, with the first case just making use of a single layer and the latter stacking 7 of them, as suggested by [7].

Going down into the details of the discussed layer, its main goal is to find out the **local and global relations** (features) between the current representation of each token. In particular, tokens come into this sub-network in their embedded form, as described in the previous section, and then they are immediately projected into a fixed **128d-representation**, which is **maintained for the rest of the model**. A general overview of an encoding layer is given in **Figure 7**.

To get some information about local interactions between tokens, **1D convolutional** neural networks have been exploited. Specifically, the model is based on **depthwise separable convolutions**, a modern approach that has proved to be a very good approximation of the standard convolution, using a lowered number of trainable variables, which helps reducing both overfitting and training time [12]. While global features are managed by a single **self-attention layer**, in its multi-head implementation, as described in [8]. Finally, a standard **feed-forward layer** concludes the block, and it has been implemented through a couple of 1D convolutions (although, classical dense layers have also been tested, with similar results). In general, these design choices allowed to avoid the use of the more common recurrent neural networks approach, that can heavily affect the training time, and not only [8].

Before passing through the first convolutional layer, encoding (especially its self-attention mechanism) needs some information about the position of each token inside of a context or a query. This is achieved by means of **positional encoding**, a technique introduced in [8] that allows adding position-related information about a token into its current representation. This information is computed through the proposed formulas (reported below), based on sine and cosine signals' concatenation, and it is summed to the current representation. This kind of spatial information is also well propagated through the network thanks to residual blocks (see below), and its application is repeated when entering a new encoding block.

$$PE(pos, 2i) = sin\left(\frac{pos}{1000^{(i/d_{model})}}\right)$$

$$PE(pos, 2i+1) = cos\left(\frac{pos}{1000^{(i/d_{model})}}\right)$$

Apart from the layers responsible for the main encoding part, each block is structured so to prevent as much as possible **overfitting** and **vanishing gradients** (which can be caused by the high number of parameters involved and the considerable number of sequential layers); moreover, stabilizing the model to let it train faster has also been considered:

- **Layer normalization** is applied after every single layer to improve model's stability and to reduce training time [13]; this

technique normalizes all the variables, for both training and testing, through their mean and variance.

- **Residual blocks** have been applied whenever possible with the goal of making the network more flexible and also reducing issues involving vanishing gradients [14]; this approach consists in adding a layer's input to its output to let the most relevant features (those coming from the shallowest layers) propagate through the network.
- **Stochastic dropout** intervenes to randomly disable some layers with a certain probability in order to improve generalization and fight against overfitting as in the more typical dropout layer [15]. Since, as stated in the previous point, the outputs of the first layers are the most useful ones, the probability of a layer to be set off depends on its depth (more on this later, see "Overfitting" section).

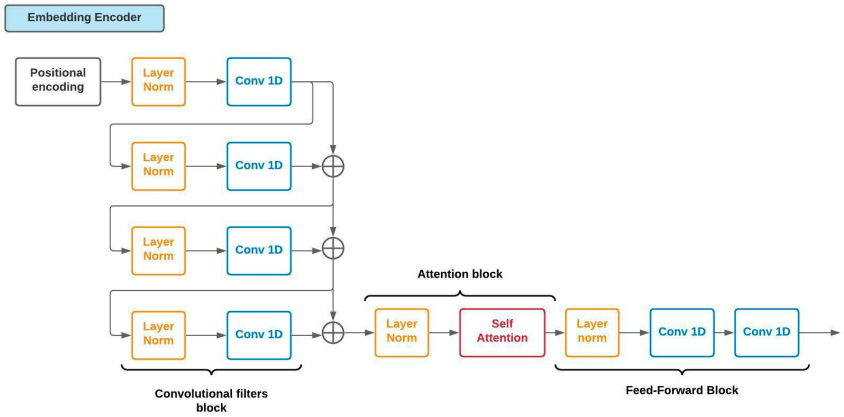

Fig. 7. Structure of an embedding encoding block

3.3 Context-query attention

The presence of a dedicated **attention block** inside the model is typical of most of the best performing models used to tackle this task [4, 7, 3]. It is a block which is mainly employed to build a query-dependant context representation. It is crucial to know, from a logical point of view, what part of the context is asked for by the query. Having a tensor-based representation of context and query, the attention is built on the common strategy of creating the **attention weights** (values that represent how much each token has an influence on the others), then changing them into probabilities that sum up to one and finally multiplying them by the query, getting a **context-query** attention tensor. Furthermore, the experimented approach follows the same strategy utilized in [3, 7], which consists in producing also a **query-context** attention, multiplying the transposed attention weights by the context. This second attention is not necessary to let the model learn the interactions between the two, but it adds a minor benefit. In the end, context-query and query-context attentions are combined to get a single representation.

All the applied operations are described by [7] and [3]. Attention weights come from the so-called **similarity matrix** ($\boldsymbol{S}$), obtained by concatenating context elements with query elements and with their respective element-wise products; then this matrix is compacted through a linear function with trainable variables, that are the only parameters of this block, while the other operations are well-defined mathematical manipulations. The entire process takes the name of "**trilinear function**" and the just described passages are synthesized in Equation (1)

$$f(q,c) = W[q, c, q \odot c] \tag{1}$$

Where f is the trilinear function, W represents the linear weights, q the embedded query and c the embedded context.

The built matrix is then submitted twice to a softmax function, one for the context-related attention ($\bar{S}$) and the other one, applied on the transposed matrix, for the query ($\bar{\bar{S}}$), producing two matrices (Equations (2) and (3)) to be respectively multiplied by query and context to get the desired attentions.

$$A = \bar{S} \cdot Q^T \tag{2}$$

$$B = \bar{S} \cdot \bar{\bar{S}}^T \cdot C \tag{3}$$

With A and B being respectively the context-to-query and query-to-context attention vectors.

The last step, merging the two attention tensors, involves a concatenation of the two together with their products and the full encoded context, as shown in Equation (4)

$$\text{output} = [c, a, c \odot a, c \odot b] \tag{4}$$

A general scheme of the operations of this block is depicted in **Figure 8**.

3.4 Model Encoder

As already explained, the above discussed encoder block has been exploited several times, for both embedding encoding and model encoding. Thus, the same sub-network has also been adopted in the subsequent step, taking as input the output of the attention block. The only sensible difference between this implementation and the embedding one relies on a different (reduced) number of

convolutional layers, as it can be observed in **Figure 9**.

This core layer has been stacked 7 times (compared to the single unit used for the embedding encoder) to build up a complete block. And, as represented in **Figure 1**, 3 of these blocks, sharing the same inner parameters (also to avoid overfitting due to an excessive number of variables) have been concatenated to reinforce the learning.

3.5 Output

Following the empirical results achieved in [7], the outputs of the 3 previous blocks have been concatenated in two different ways (one for each head of the network), using the first and the second output to compute start probabilities, and first and third outputs for end probabilities. These probabilities are basically obtained by a sequential application of a linear layer, followed by a softmax function.

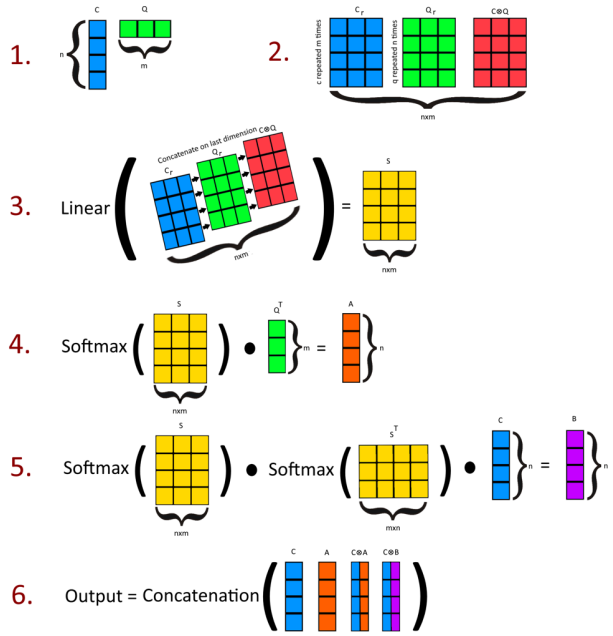

Fig. 8. Operations computed into the context-query attention block; the d-model dimension has not been represented for simplicity

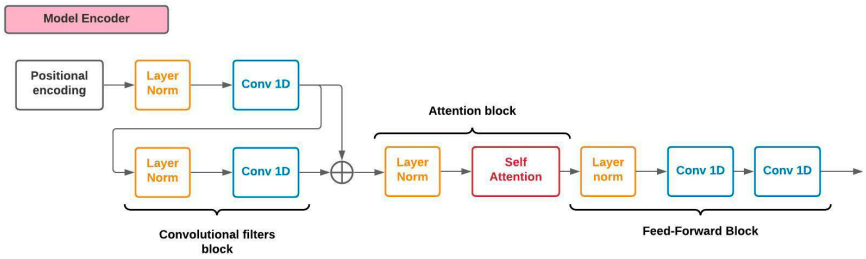

Fig. 9. Structure of a model encoder block

4 Training

The training has been divided in two phases.

The **first phase** had the aim of finding the best hyper-parameters. For this purpose, the validation set has been exploited for evaluating network's generalization performances.

Once defined suitable hyper-parameters, the **second phase** started, consisting in using the entire dataset as the training set, with no validation anymore.

This procedure has led the network to better results on unseen data (generalization improvement).

4.1 Loss function

The properties of the network, and the presence of a double output, led to the construction of a **custom loss function** to train the model on. Particularly, the adopted approach is the same proposed by [7] and it is based on the following loss function:

$$L(\vartheta) = -\frac{1}{N}\sum_{i}^{N}\left[log\left(p_{y_i^s}^{s}\right) + log\left(p_{y_i^e}^{e}\right)\right] \tag{5}$$

The main idea is trying to make the model assign high probabilities to those tokens that perfectly represent the start and the end positions of the correct answer. So, probabilities associated with tokens of the real answers are selected and the network has to find suitable values for the parameters, aiming at maximizing them. Equation (5) does nothing but exposing a function that, if minimized, maximizes the right probabilities. This is done by multiplying the two probabilities in output (the ones with the indices of interest), computing the negative of their logarithms (to transform the problem into a minimization task that can be dealt by the optimizer), and finally taking the mean over the entire batch of samples.

It is worth to point out that an alternative and more typical loss function could have been used instead. A standard sparse categorial cross entropy [16] could have been applied twice for start and end probabilities. In this view, each token can be considered as a word of a vocabulary and the entire process as a multiclass classification. So, the two computed losses could be summed to get the final loss of a single sample. This different strategy has been tested, with similar results compared to the chosen option, which has been preferred for its more logical way of operating with respect to the problem.

4.2 Metrics

For evaluating the model's performance, **F1** and **Exact Match** (*EM*) scores have been used.

The **F1** score is a measure that considers the tokens in common between those predicted by the model and the ground truth, regardless of their positions. The following formulas show how the score is computed:

$$PRECISION = \frac{number\ of common\ tokens}{number\ of\ predicted\ tokens}$$

$$RECALL = \frac{number\ of common\ tokens}{number\ of\ true\ tokens}$$

$$F1 = 2 * \frac{(PRECISION * RECALL)}{(PRECISION + RECALL)} \tag{6}$$

The **EM** score (Equation (7)), instead, measures the exact overlapping between answer's tokens predicted by the model and the ground truth:

$$EM = \frac{number\ of\ predicted\ and\ true\ tokens\ in\ the\ same\ positions}{number\ of\ true\ tokens} \tag{7}$$

Inspired by the official *SQuAD* evaluation script [1, 17], a cleaning on the predictions and true answers have been applied before computing these metrics. More precisely, **punctuations and articles have been removed**.

Additionally, in order to avoid penalizing too much these scores with incorrect predictions, an **inference algorithm** has been created, whose goal is to consider only those predictions so that the product between **start** and **end predictions** of the span is maximized, keeping the constraint that *end* indices cannot come before *start* ones.

The algorithm proceeds as follows:

1. The model returns two tensors from each of its heads: **y_pred_start** and **y_pred_end** that represent the predictions (distribution of probabilities) for the starting and ending positions of the answer, respectively.
2. **y_pred_start** and **y_pred_end** are multiplied, to have a matrix that contains all the combinations of the output probabilities of the model.
3. The above matrix is multiplied by a mask containing only the valid positions (e.g., the starting position index less than or equal to the ending position index).
4. Indices of the highest masked probability product are extracted, then the extraction of the answer starting/ending positions is performed, to eventually compute the F1 and EM scores.

4.3 Overfitting

Initially, the entire model has been created without applying any technique against overfitting, and it has been tested on a dummy dataset. In this way, it was possible to verify whether the network was behaving correctly or not (such as learning and overfitting very quickly).

Afterward, the model has been trained on the full training and validation sets, and a totally expected outcome emerged: the network was performing very good on the training set, but its performances were decreasing on the validation set just after few epochs of training. In other words, the network was heavily affected by overfitting.

Therefore, also inspired by [7], three different techniques against overfitting have been used: **dropout**, **L2 regularization** and **layer stochastic dropout**.

The **dropout** [18] has been applied in different positions, depending on the network block. In the embedding encoder and the model encoder, the dropout has been applied with a rate of 0.1 every 2 convolutional layers, before the self-attention, and before the feed-forward layer. **In the input embedding**, the dropout has been applied after each embedding layer, with a rate of 0.1 for the words, and 0.05 for the characters. In the **context-query-attention** the dropout has been applied with a rate of 0.1 to both query and

context coming from the previous layer. No dropout has been applied in the **output layer**.

The **L2 regularization [19]**, instead, has been applied on all layers that have trainable parameters, except for the embedding layers. The L2 rate used is 3e-7.

The **stochastic dropout** has been applied in the **embedding encoder** and **model encoder**, and it is used for deciding whether to maintain a layer or just remove it [15]. In the latter case, only the residual block connection survives. The deeper a layer is, the higher is the probability of being dropped, following Equation (8).

$$1 - \frac{current\ layer\ depth}{max\ depth} \cdot (1 - survival\ probability) \tag{8}$$

Where $survival_probability = 0.9$. The depths are reset at the beginning of each embedding encoder or model encoder block.

As a final note, it is important to remark that all these techniques are used only during training phase, and **removed for the inference phase**.

4.4 Other training details

Besides the aforementioned fundamental points implemented to let the training work properly, the following other techniques have been used (or at least tested) for better and more stable results:

1. **Warmup routine**: the learning rate of the optimizer can be static or evolve over time while the model learns, and losses move toward points of minima. Through several simulations, all the typical **best practices** have been tried to find a reasonable strategy to handle the learning rate at best, ensuring convergence and reducing as much as possible the training time. In the end, the most successful attempt has revealed to be a starting **warmup routine** followed by a fixed learning rate: for the first 1000 steps (about half of the total number of steps using 32 as batch size), learning rate is increased following a logarithmic growth from 0 to the value (found empirically) of 0.001; then it remains constant. This is computed through Equation (9). This kind of warmup, as explained in [14] , is intended to avoid an early overfitting and it is one of the best practices for cases when there could be an unlucky imbalance inside the batches. If the shuffled data include a cluster of related, strongly-featured observations, the model's initial training can skew badly toward those features (or worse, toward incidental features that aren't truly related to the topic at all). Warmup is a way to reduce the primacy effect of the early training examples. Without it, it may need to run a few extra epochs to get the convergence desired, as the model un-trains those wrong early discoveries.
2. **Exponential learning rate decay**: as the name suggests, consists in gradually (exponentially) reducing its value step after step, intending to support convergence, that can be more complicated to reach with the loss approaching a minimum. These tests produced considerably bad results, with the model unable to learn from a certain (early) point on (convergence on high loss); this is due to the presence of many local minima that cannot be escaped from with a low learning rate.
3. **Clipping**: one of the most concerning issues encountered in training sessions, among overfitting and others, has been the "**vanishing gradients**" effect, caused by the complexity of the network in terms of number of layers and parameters. This phenomenon has been diminished with the introduction of already examined techniques (residual blocks, L2 regularization), but without reaching a complete solution: for a long time, after a critical number of epochs (on average, 5 or 6), the computed loss on the training and validation sets increased a lot and apparently with no reason at all. To solve this issue, all the gradients have been subjected to clipping: their value has been normalized in a fixed range, so to avoid them to go down the threshold that caused the vanishing effect. This other best practice, that is also suggested to decrease the training time [20], solved the problem in question.
4. **Exponential moving average**: the last best practice that has been tested involves assigning different weights to run the model on the validation/test set compared to those used for training. In particular, the exponential moving average method consists in computing the weighted mean of a variable in time, so that each value in a specific time step (training step) contributes differently to the final mean and older values are weighted using a **decay factor of 0.999**. This computed value is then assigned to the correspondent parameter at evaluation time. The intended goal is to take into account also, in a small part, previous discoveries of the network. Although better results were expected [21], this extension has not shown any benefit during the evaluation; thus, it has been discarded in the end.

$$\min\left(l_{rate} \cdot \frac{\log(\text{train}_{step} + 1)}{log(1000)}, l_{rate}\right) \tag{8}$$

5 Experiments

Some parameters have been fixed by taking their values directly from [7]. In particular, *GloVe word embedding* size set to 300, the *embedding layer convolutional kernel size* set to 5, the *number of highway layers* set to 2, the *encoder convolutional kernel size* set to 7, *the number of convolutional layers* in the embedding encoder and model encoder set to 4 and 2, respectively. Finally, the *number of repeated encoder blocks* for the embedding encoding is 1, whereas it is 7 for the model encoding.

Nevertheless, some parameters have also been changed with respect to [7].

The *character embedding size* changed from 200 to 64, because of a discovery: the model generalizes better in this way if there is an inferior number of heads (next point below).

The *number of heads* used in the multi-head attention has been set to 5 instead of 8, due to memory issues when using too many

heads. For the same reason, also the hidden size of the inner layers has been lowered from 128 to 96.

The used optimizer is **Adam**, with $\beta_1 = 0.8$ and $\beta_2 = 0.999$, which represents the best combination tested so far.

As mentioned before, *dropout* and *L2 regularization* have also been adopted. However, before fixing their rates, several combinations have been tried by changing both the L2 rate, with values between 1e-2 and 3e-7, and the dropout rate, with values between 0.1 and 0.5 (see the "*Overfitting*" section for further details).

The last hyper-parameter that is worth to mention is the number of epochs. Different strategies have been experimented for deciding the right number of epochs, by initially monitoring the validation loss, and then the validation metrics. However, empirical discoveries showed that training the model for 60 epochs let it possible to obtain the best possible results both on the validation set and the test set.

Once fixed all the necessary hyper-parameters, an ablation study has also been performed, especially on the applied anti-overfitting techniques. **Figures 10** and **11** clearly depict how much smoother and stable is the loss when applying all these techniques:

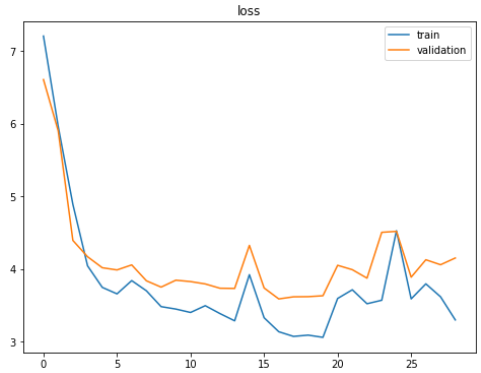

Fig. 10. Loss history for train and validation sets with no countermeasures against the overfitting

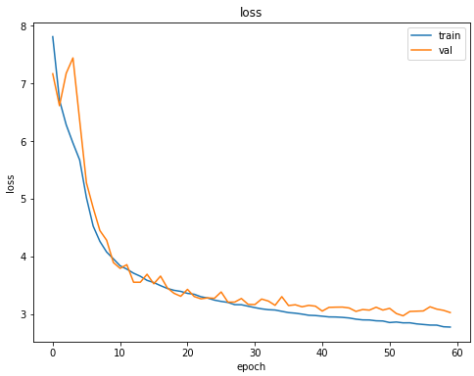

Fig. 11. Loss history for train and validation sets using all the techniques against the overfitting

It is important to notice that the L2 regularization affects heavily, and positively, the training loss, with stabilizing effects. This smoothness allowed to train the network for much more epochs (60), leading to better results.

Finally, as previously remarked, the network has been trained on the full training set. Figures 12 and 13 show the metric scores obtained during this stage.

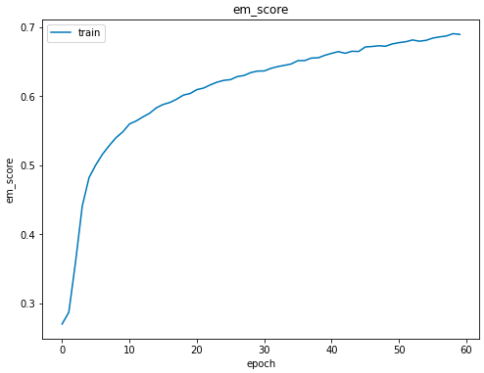

Fig. 12. Exact match score history during the training on the entire training set

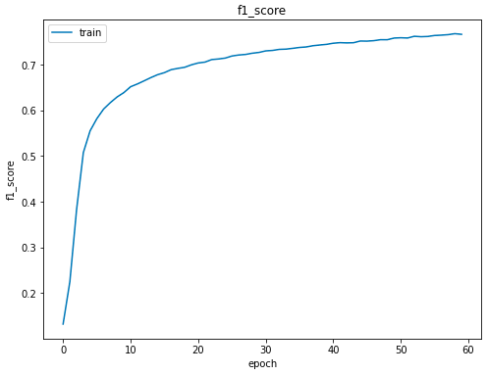

Fig. 13. F1 score history during the training on the entire training set

6 Results

To evaluate each model and get comparable results with the ones obtained by other authors, the official evaluation script [1, 17] along with the official test set have been utilized. It deserves attention that the referenced script of evaluation computes the metrics in the same fashion as done by the model during the validation step, with the exception of simultaneously considering all the possible true answers to a given question.

Running the model with the best tuned hyper-parameters, the obtained **scores for EM and F1 are 63.7 and 74.8** respectively, which can be considered good results, based on the implemented architecture.

As a reference, Table 1 compares these results to older approaches that are mainly based on RNNs. These scores are taken directly from the SQuAD leaderboard [17].

It is important to remark that obtained scores are lower than those mentioned in the *QANet* of [7] because they performed additional operations, which were very important so as to increase the scores. First, they used **more trainable parameters,** for instance by adding a higher number of heads in the attention layer. Moreover, they implemented a noticeable extension to their work, by introducing transformers intended to increase the number of samples (**data augmentation**) through a double-translation mechanism.

Finally, it is relevant to say that in the last few years, new architectures and approaches are becoming more and more popular

for solving many NLP tasks, and question answering is among those. These techniques are typically based on transformers, and they are frequently very sophisticated, for example they often require the training of multiple models that lead to the creation of an ensembled one. Additionally, they usually need much more computational power than any previously proposed architecture.

Single model	EM / F1
Dynamic Chunk Reader (Yu et al., 2016)	62.5 / 71.0
Match-LSTM with Ans-Ptr (Wang & Jiang, 2016)	64.7 / 73.7
Multi-Perspective Matching (Wang et al., 2016)	70.4 / 78.8
Dynamic Coattention Networks (Xiong et al., 2016)	66.2 / 75.9
FastQA (Weissenborn et al., 2017)	68.4 / 77.1
BiDAF (Seo et al., 2016)	68.0 / 77.3
SEDT (Liu et al., 2017a)	68.5 / 78.0
FastQAExt (Weissenborn et al., 2017)	70.8 / 78.9
ReasoNet (Shen et al., 2017b)	70.6 / 79.4
Document Reader (Chen et al., 2017)	70.7 / 79.4
Ruminating Reader (Gong & Bowman, 2017)	70.6 / 79.5
jNet (Zhang et al., 2017)	70.6 / 79.8
Interactive AoA Reader (Cui et al., 2017)	73.6 / 81.9
Reg-RaSoR (Salant et al., 2017)	75.8 / 83.3
DCN+ (Xiong et al., 2017)	74.9 / 82.8
AIR-FusionNet	76.0 / 83.9
R-Net (Wang et al., 2017)	76.5 /84.3
BiDAF + Self Attention + ELMo	77.9/ 85.3
Reinforced Mnemonic Reader (Hu et al., 2017)	73.2 / 81.8
Dev set: QANet	73.6 / 82.7
Dev set: QANet + data augmentation ×2	74.5 / 83.2
Dev set: QANet + data augmentation ×3	75.1 / 83.8
QACNNnet (discussed model)	**64.8 / 75.6**

7 Error Analysis

In order to find some valuable information about ways for improving the model, a set of analysis has been executed. The aim of this step was to find out which were the weakest points of the network, meaning which features were responsible for partially or completely wrong predictions.

7.1 Statistical analysis

The total number of features that could be considered to fully analyse such a complex process, that involves 5 different inputs and 2 different outputs, is huge; furthermore, most of them are not humanly explainable. To handle this task, predictions have been split into three categories (***solved***, ***partially solved*** and ***unsolved***), then just the most relevant features have been considered and studied:

1. **Answers' lengths**: one of the most "obvious" properties to look for is represented by the length of the span to be predicted, which can negatively affect at least the *EM* score. As shown in **Figure 14**, the distributions of these lengths have been compared involving the three mentioned classes. Results do not fulfil the expectations, since the distributions are very similar in terms of peaks and troughs: the length of an answer's span does not have any visible effects on the quality of the model's predictions; this represents another strength point of the discussed approach.
2. **Most frequent tokens**: another reasonable explanation for errors can be the presence of certain frequent tokens with destabilising consequences. Exploiting a *quantile-cut* (set to 0.99), once again for each class, lists of the most repeated tokens have been produced and examined. However, also this time no correlation has appeared, with the three lists showing almost the same tokens.
3. **Type of questions:** the last relevant feature that has been taken into account is the influence that different types of questions can have on the network. More specifically, they usually contain one token that represents the object of the question itself; these tokens are "*why*", "*who*", "*what*", "*where*", "*when*", "*which*" and "*how*". Results, depicted in **Figure 15-16-17**, highlight the first small correlation found, as some types are proportionally more frequent in some categories than others. In particular, the following conclusions can be drawn:

   - "*what*", "*where*" and "*which*" questions do not cause any sensible effect on the predictions;
   - "*who*" questions are among the most difficult ones to solve (no matching between tokens);
   - "*how*" questions have a negative effect on both the *EM* and *F1* scores.

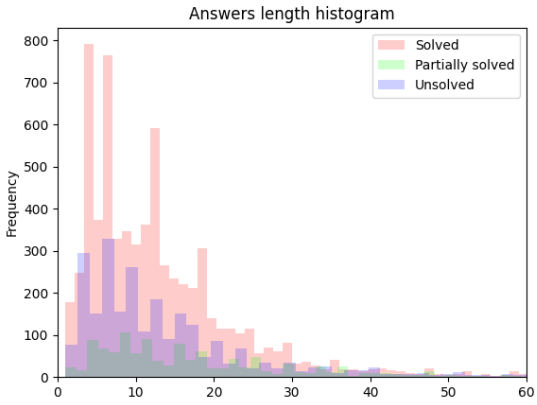

Fig. 14. The answers' lengths distribution

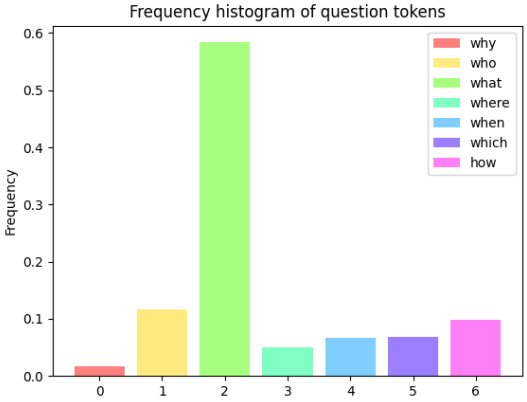

Fig. 15. Frequency distribution for SOLVED questions

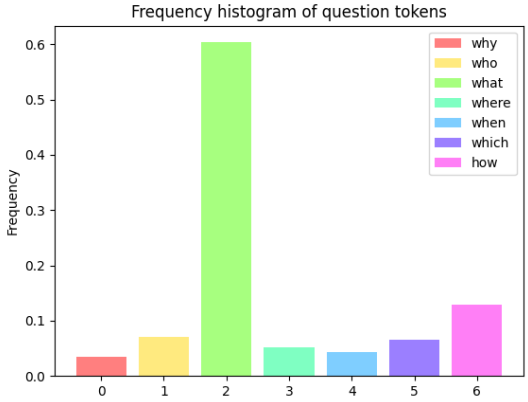

Fig. 16. Frequency distribution for PARTIALLY SOLVED questions

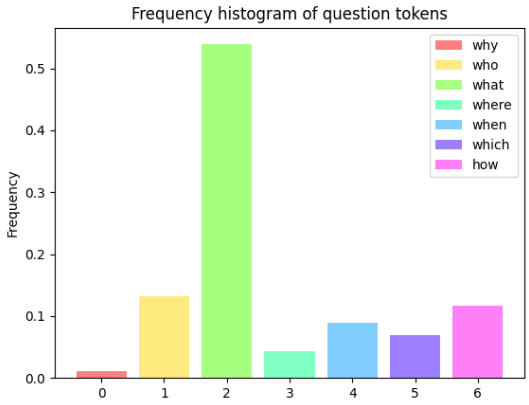

Fig. 17. Frequency distribution for UNSOLVED questions

7.2 Extension: deep learning-based analysis

As it can be noticed by the previous observations, the outcomes of the statistical analysis did not add many significant information (even results of point "3", that include relatively rare types of questions, compared with the predominant "*what*" question), not pointing toward a specific feature that impacts the model. In an attempt to fill this gap, a further and more complex step has been taken. Deeper features have been sought by means of another neural architecture. The main idea was to catch, in an environment rich in texts and variables, the discriminant features, owned by a question, that led to successes or failures. The core of this point is a theory according to which most of the errors come from questions structured in a way that lets the model go wrong [22].

To prove this theory, a full extension has been implemented, with a new straightforward model. Going down into the details, a classification problem has been defined, with three possible classes: "***easy***" (the *QACNNnet* will predict an answer with an exact match, equivalent to "*solved*" of the previous section), "***medium***" (a prediction with an *F1* score below the 0.5 threshold is expected, equivalent to "*partially solved*") and "***difficult***" (complete failure, equivalent to "*unsolved*"). So, a basic model has been built to tackle this problem. Its components are, sequentially, an **embedding layer** initialized with *GloVe* embeddings [10] (including the "*UNK*" token, as for the previous model), a **bidirectional RNN** layer based on **LSTM** (with 0.1 ***dropout*** *rate*) and a final **dense layer** with three outputs and a softmax activation function. **Inputs are only questions**, submitted to the same pre-processing and words tokenization as for the *QACNNnet*.

Results show a final accuracy on test and validation sets *above 95%,* which is not so high, considering a little class imbalance ("*easy*" labels appear slightly more than the other two classes on average) and the limited number of classes. On the other hand, this score can be achieved just by considering questions and completely ignoring contexts and possible answers. This is a significant fact, and it represents an empirical proof of the above-mentioned theory, meaning that, with different structures in questions, results could be improved for the *QACNNnet*. Moreover, the released extension can be adopted by other useful tasks, such as:

- More difficult datasets can be created by means of an adversarial mechanism to mislead the network.
- A *seq-to-seq* model can be associated to transform the query of a user into one with the same meaning but with a structure that increases the chances of an exact match (or a higher *F1* score, at least).

8 Conclusions and possible improvements

The implementation of the discussed model has brought to many interesting results. Even though this is far from being the new state of the art for question answering, the ratio between architectural complexity and performances is definitely positive. The network proved to be solid and to be able to reach impressive scores in a quite limited amount of time (even after only 30 epochs, which corresponds to roughly 6 hours of training), showing an overwhelming difference compared to equally complex models proposed in the past. In fact, they came to similar results with networks based on RNNs, thus much slower and inadequate for real-time applications.

Except for the previous point, the proposed work should be interpreted as another proof of the superiority of the CNNs combined with self-attention with respect to RNNs; without having any pretension to overtake the modern approaches, which put back into the scene difficulties related to training times. Moreover, this work showed the strength of most of the new techniques adopted to handle typical issues such as overfitting, vanishing gradients and many more.

The combination of all the chosen strategies and tuned hyper-parameters led to quite successful out-comes, even with lighter versions of the network (removing some layers and decreasing base dimensions), making it a stable and reliable alternative for systems that require fast responses.

One of the possible ways for improving the network's performances would be to increment the model complexity, by further leveraging the CNNs and attentions mechanisms (e.g., increase the layers' hidden size from 96 to 128, increase the number of heads in the multi-head-attention layer, etc...). Moreover, based on the error analysis considerations in the previous paragraph, with the aim of other neural networks (e.g., seq-to-seq) the QACNNnet's inputs could be enhanced in order to increase its performances.

As a final note, it would be interesting to investigate the data augmentation extension implemented by the referenced paper [7], in an attempt to achieve even better results.

References

1. Rajpurkar, P., et al.: SQuAD: 100,000+ Questions for Machine Comprehension of Text (2016)
2. Wang, S., et al.: Machine Comprehension Using Match-LSTM and Answer Pointer. In: ICLR 2017 (2016)
3. Seo, M., et al.: Bidirectional Attention Flow for Machine Comprehension (2016)
4. Xiong, C., et al.: Dynamic Coattention Networks For Question Answering (2016)
5. Peters, M.E., et al.: Deep contextualized word representations (2018)
6. Zhang, Z., et al.: Retrospective Reader for Machine Reading Comprehension (2020)
7. Yu, A.W., et al.: QANet: Combining Local Convolution with Global Self-Attention for Reading Comprehension (2018)
8. Vaswani, A., et al.: Attention Is All You Need (2017)
9. NLTK: Natural Language Toolkit
10. Pennington, J., et al.: GloVe: Global Vectors for Word Representation (2014)
11. Srivastava, R.K., et al.: Highway Networks (2015)
12. Chollet, F.: Xception: Deep Learning with Depthwise Separable Convolutions (2016)
13. Ba, J.L., et al.: Layer Normalization (2016)
14. He, K., et al.: Deep Residual Learning for Image Recognition (2015)
15. Huang, G., et al.: Deep networks with stochastic depth. In: ECCV 2016 (2016)
16. Zhang, Z., et al.: Generalized Cross Entropy Loss for Training Deep Neural Networks with Noisy Labels (2018)
17. The Stanford Question Answering Dataset
18. Srivastava, N., et al.: Dropout: A Simple Way to Prevent Neural Networks from Overfitting (2014)
19. Krogh, A., et al.: A Simple Weight Decay Can Improve Generalization (1991)
20. Zhang, J., et al.: Why gradient clipping accelerates training: A theoretical justification for adaptivity (2019)
21. Cai, Z., et al.: Exponential Moving Average Normalization for Self-supervised and Semi-supervised Learning (2021)
22. Rondeau M. et al: Systematic Error Analysis of the Stanford Question Answering Dataset (2018)