Attention-aware attention (A^3) : combining coattention and self-attention for question answering

Stanford CS224N - Default Project - SQuAD IID Track

Pierre CHAMBON
Institute for Computational and Mathematical Engineering
Stanford University
pchambon@stanford.edu

Alban ZAMMIT
Institute for Computational and Mathematical Engineering
Stanford University
azammit@stanford.edu

Abstract

Attention has been one of the biggest recent breakthrough in NLP, paving the way for the improvement of state-of-art models in many tasks. In question answering, it has been successfully applied under many forms, especially with recurrent models (encoder-decoder fashion). Co-attention [1] and multihead self-attention [2] have been two interesting attention variations, but a larger study trying to combine them has never been conducted to the best of our knowledge. Hence, the purpose of this paper is to experiment different attention-based architecture types for question answering, as variations from one of the first successful recurrent encoder-decoder models for this task: BiDAF [3]. We test the performance of our model on the Stanford Question Answering Dataset 2.0 [4] and achieved a performance of EM = 62.730 and F1 = 66.283 on the dev set, and **EM = 60.490 and F1 = 64.081 on the test set**.

1 Introduction

Natural Langage Processing is currently one of the most active research area, especially since the breakthrough made possible by Deep Learning in the middle of the 2010s'. One of the most studied task in NLP is question answering, which consists in building models capable of answering a question based on a paragraph (called context or passage). Building efficient models at this task can improve the daily life of everyone whether it is to give you an answer as precise as possible when you query a browser, or when you are asking an information to your connected device.

Recent neural architectures managed to beat human performance on the SQuAD 2.0 dataset [4] [1], which is the *golden* evaluation tool to compare models between them and is the first choice when a team wants to experiment new models in question answering. These recent advances can be separated in two groups: those using Tranformers-like architectures with pretrained embeddings (such as BERT and all its derivatives) and those based on recurrent architectures such as BiDAF [3] and all its derivatives, which will be our baseline for the entire project. The latter, even though not being state-of-the-art anymore, offers more scope for creativity in the architecture and design to explore different techniques and develop intuitions.

Our approach explores both coattention and (multihead) self attention mechanisms in models based on recurrent architectures. We try to combine them, either by adding the two mechanisms or selecting

[1] https://rajpurkar.github.io/SQuAD-explorer/

the best from both worlds, to improve our baseline BiDAF model [3]. To this end, we keep most of the architecture and training techniques of the BiDAF model, and replace its bidirectional attention flow layer by our own attention layers.

2 Related Work

In recurrent models, the attention mechanism is used in a multi-stage hierarchical way to represent the question and context words. Every architecture aims at building a relevant representation of the words of the context, so that an accurate span for the answer can be found. For example, the BiDAF paper [3] uses a bi-directional attention flow mechanism to obtain question aware context representation (C2Q for context to question) and context-aware question vectors (Q2C for question to context). More precisely, an attention layer learns dependencies of the question words to context words, and further a modeling layer focuses on learning the interactions between the question-aware passage representation.

On the other hand, the coattention paper [1] also builds C2Q attentions and Q2C attentions, but instead of using them to feed the modeling layer, they are then mixed (using also a softmax ponderation of the question vectors for each document vector) and passed as input of an additional LSTM layer whose hidden states are namely the coattention encodings.

Another described variation of the attention mechanism is to not only use the scaled dot product attention but also the mechanism of self attention itself. In the R-Net paper for example [2], the first layer of attention is a self attention of the context to itself, and the question to itself, and then the bi-directional attention flow mechanism [3] is applied to these self-attended states.

3 Approach

Our baseline model follow the BiDAF architecture [3] as presented in the project handout provided by the CS224n teaching team[2]. Given the limited space allowed for this report, we cannot re-explain all the architecture. Instead, the reader should refer to the handout, especially part 4.1, where the different layers (*Embedding Layer*, *Encoder Layer*, *Attention Layer*, *Modeling Layer*, *Output Layer*) are presented. For the sake of simplicity, we will use exactly the same notations as those of handout's for the rest of this report.

Our approach consisted in modifying this baseline, especially its attention architecture to include coattention and self-attention. The next subsections describe the modifications we did from this baseline.

3.1 Coattention

Coattention [1] is a modification of the attention layer in the original BiDAF architecture. Let's note the context hidden states $c_1, ..., c_N \in \mathbb{R}^l$ and the question hidden states $q_1, ...q_M \in \mathbb{R}^l$, obtained from the encoder layer. Firstly, an activation is applied to the question hidden states to obtain projected question hidden state $q'_1, ..., q'_M$:

$$\forall j \in \{1, ..., M\} \qquad q'_j = \tanh(Wq_j + b) \in \mathbb{R}^l$$

with W and b learnable parameters. Next, we compute the affinity matrix $L \in \mathbb{R}^{N \times M}$, such that:

$$\forall i \in \{1, ..., N\} \quad \forall j \in \{1, ..., M\} \qquad L_{ij} = c_i^T q'_j$$

We then compute attention outputs for both directions. For the Context-to-Question (C2Q) Attention, we get attentions vectors a_i for $i \in \{1, ..., N\}$ such that

$$\alpha^{(i)} = \text{softmax}(L_{i,:}) \in \mathbb{R}^M \qquad \text{softmaxes over the i-th row of } L$$

[2] `http://web.stanford.edu/class/cs224n/project/default-final-project-handout-squad-track.pdf`

$$a_i = \sum_{j=1}^{M} \alpha_j^{(i)} q'_j \in \mathbb{R}^l \qquad \text{attention vector}$$

For the Question-to-Context (Q2C) Attention, we get attentions vectors b_j for $j \in \{1, ..., M\}$ such that

$$\beta^{(j)} = \text{softmax}(L_{:,j}) \in \mathbb{R}^N \qquad \text{softmaxes over the j-th column of } L$$

$$b_j = \sum_{i=1}^{N} \beta_i^{(j)} c_i \in \mathbb{R}^l \qquad \text{attention vector}$$

Then, we use the C2Q attention distributions $\alpha^{(i)}$ to take weighted sums of the Q2C attention outputs b_j. This gives us second-level attention outputs s_i for $i \in \{1, ..., N\}$, which takes into account Q2C attention through b_j, but consists in N vectors):

$$s_i = \sum_{j=1}^{M} \alpha_j^{(i)} b_j \in \mathbb{R}^l$$

Finally, we concatenate the second-level attention outputs s_i with the first-level C2Q attention outputs a_i, and feed the sequence through a bidirectional LSTM. The resulting hidden states u_i of the BiLSTM are known as the coattention encoding. This is the overall output of the Coattention Layer.

$$u_1, ..., u_N = \text{BiLSTM}([s_1; a_1], ..., [s_N; a_N])$$

In our experiments, if the trained model uses this version of the *Attention Layer*, we will note **CoAtt**

A slightly modification of this coattention layer is to add sentinel vectors $c_\emptyset \in \mathbb{R}^l$ and $q_\emptyset \in \mathbb{R}^l$ (which are trainable parameters of the model and make it possible to attend to none of the provided hidden states.) to both the context and question states c_i and q'_j and then follow the exact same calculation. We explored several initialization methods of these specific weights, the fact of using separate or common weights, as well as the best position to insert them in the input sequences. In our experiments, if the trained model uses this version of the *Attention Layer*, we will note **CoAttWithSent**

3.2 Self-Attention

A second step in improving the attention mechanism of our model was to include a self-attention layer, along with a scaled dot product and multiple heads, to our existing attention layer. We tried two architectures of self-attention in combination with the original coattention mechanism, and one including also the trilinear attention of the original BiDAF model.

The main goal is to complexify (and therefore hopefully improve) the attention by not only including a cross-attention layer, but also a self-attention layer, so that all types of interactions between the input tokens, from both the context and the query, can be easily modelized by our neural network. We observed that in the coattention implementation, there is a way for input tokens to interact with each other: through the second layer of coattention, which computes a context to query to context cross attention. But this remains a convoluted way for the model to make context tokens interact with each other; in addition, there is no direct way for query tokens to interact with each other at the attention layer level.

Using self-attention provides a way for both context and query tokens to interact with the tokens from the same class, before or along computing a cross-attention between the two.

3.2.1 Coattention with multi-headed Self-Attention

We explore two main architectures for the combination of self-attention with coattention:

- First, we start by inputting our query matrix and context matrix to a self-attention layer, therefore coding more information into each token about its role among its surrounding

tokens. These self-attended tokens, for both the context and the query, are then inputted in the cross attention mechanism of the coattention paper.

- Second, we keep the first layer of the coattention mechanism as it is, computing a cross-attention on the input tokens. The query tokens are previously passed through a dense layer. Then, the second layer of coattention is replaced with a self-attention and then a single direction cross attention, of the self-attended context tokens to the self-attended query tokens.

The goal is to use self-attention while still providing enough possibilities to the model to compute complex relations. Even though we start by a more simplistic model, removing the second layer of coattention and simply passing the tokens through a self-attention layer before a single direction dot-product cross-attention (see figure 1), we quickly find out that such a model is not capable of learning as complex relations.

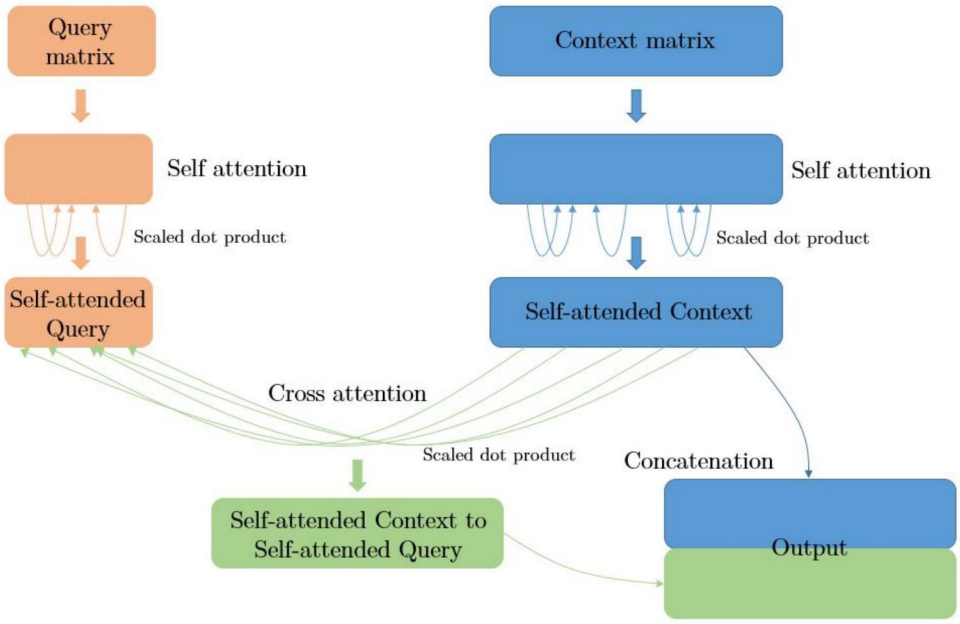

Figure 1: Illustration of our base architecture of self-attention with cross attention, to create an attention layer taking into account relationships between context tokens, query tokens, and themselves.

Our first attempt of coattention with self-attention aims to preprocess the input tokens with self-attention before passing them to the coattention layers. The objective is: in original coattention, we interpret the computations as answering a question at the token level, and answering a question contextualized with the context tokens at the token level. Using self-attention enables us to ask a question contextualized with the question tokens to a context contextualized wit the context tokens. This way, each token is aware of its surroundings, and we hope this will help the model better perform on the Question-Answering task.

We use the same notation as for the coattention layer description, that is to say we introduce $c_1, ..., c_N \in \mathbb{R}^l$ the context tokens as outputted by the encoder layer, and $q_1, ..., q_M \in \mathbb{R}^l$ the query tokens also as preprocessed by the encoder layer.

To break the symmetry between them, we remove the dense layer that was applied on the query tokens only, as used in the coattention. Instead, we introduce the weights of our self-attention layer, that is to say $W_C^Q, W_C^K, W_C^V \in \mathbb{R}^{att \times l}, W_C^O \in \mathbb{R}^{l \times att}$ and $W_Q^Q, W_Q^K, W_Q^V \in \mathbb{R}^{att \times l}, W_Q^O \in \mathbb{R}^{l \times att}$. We notice that we use different weight matrices, to still process query and context tokens in a different manner. Then:

$$\forall i \in \{1, ..., N\} \qquad c_i^Q = \text{relu}(W_C^Q c_i + b_C^Q) \in \mathbb{R}^{att}$$

$$c_i^K = \text{relu}(W_C^K c_i + b_C^K) \in \mathbb{R}^{att}$$
$$c_i^V = \text{relu}(W_C^V c_i + b_C^V) \in \mathbb{R}^{att}$$

$$\forall j \in \{1, ..., M\} \qquad q_j^Q = \text{relu}(W_Q^Q q_j + b_Q^Q) \in \mathbb{R}^{att}$$
$$q_j^K = \text{relu}(W_Q^K q_j + b_Q^K) \in \mathbb{R}^{att}$$
$$q_j^V = \text{relu}(W_Q^V c_i + b_Q^V) \in \mathbb{R}^{att}$$

For the sake of simplicity, we choose to map query, key and value tokens to a space of the same dimension. One could choose to rather distinguish the size of the query-key space and the value space, for instance to better optimize the computational time (one space could be smaller). In the case of several heads, we have a set of such matrices for each head, and each head performs its self-attention computation, with its own weights. We choose the att dimension to be a divisor of the input size l, so that $l = num_{heads} \times att$.

Then, we compute the two affinity matrices:

$$\forall i, j \in \{1, ..., N\} \qquad L_{Cij} = \frac{c_i^{QT} c_j^K}{\sqrt{att}}$$

$$\forall i, j \in \{1, ..., M\} \qquad L_{Qij} = \frac{q_i^{QT} q_j^K}{\sqrt{att}}$$

Then, we compute the attention scores of each key token for a particular query token, and the weighted sum of their value tokens, by their attention score:

$$\alpha_C^{(i)} = \text{softmax}(L_{Ci,:}) \in \mathbb{R}^N \qquad \text{attention scores of each key for the query } c_i^Q$$
$$c_i^{attention} = \sum_{j=1}^{N} \alpha_{Cj}^{(i)} c_j^V \in \mathbb{R}^{att} \qquad \text{attention vector of the ith context token}$$

$$\alpha_Q^{(i)} = \text{softmax}(L_{Qi,:}) \in \mathbb{R}^M \qquad \text{attention scores of each key for the query } q_i^Q$$
$$q_i^{attention} = \sum_{j=1}^{M} \alpha_{Qj}^{(i)} q_j^V \in \mathbb{R}^{att} \qquad \text{attention vector of the ith query token}$$

This gives us two matrices $C^{attention}$ and $Q^{attention}$. To get the output tokens of the self-attention layer, we finally run them through an output layer, such that:

$$\forall i \in \{1, ..., N\} \qquad c_i^{self-attended} = \text{relu}(W_C^O c_i^{attention} + b_C^O) \in \mathbb{R}^l$$

$$\forall j \in \{1, ..., M\} \qquad q_j^{self-attended} = \text{relu}(W_Q^O q_j^{attention} + b_Q^O) \in \mathbb{R}^l$$

We can then use the two matrices of self-attended context and self-attended query as an input to the coattention layer, as described in the previous section. This gives us the global architecture of the attention layer seen on figure 2.

In our experiments, if the trained model uses this version of the *Attention Layer*, we will note **CoSelfAttStacked**

Our second approach aims to merge the self-attention layer with the coattention, instead of stacking one on top of another. To do this, we choose to keep the first coattention layer as it is, running on the input context and query matrices. Then, we replace the second layer by a self-attention and a cross-attention between the self-attended tokens. This is described in figure 3.

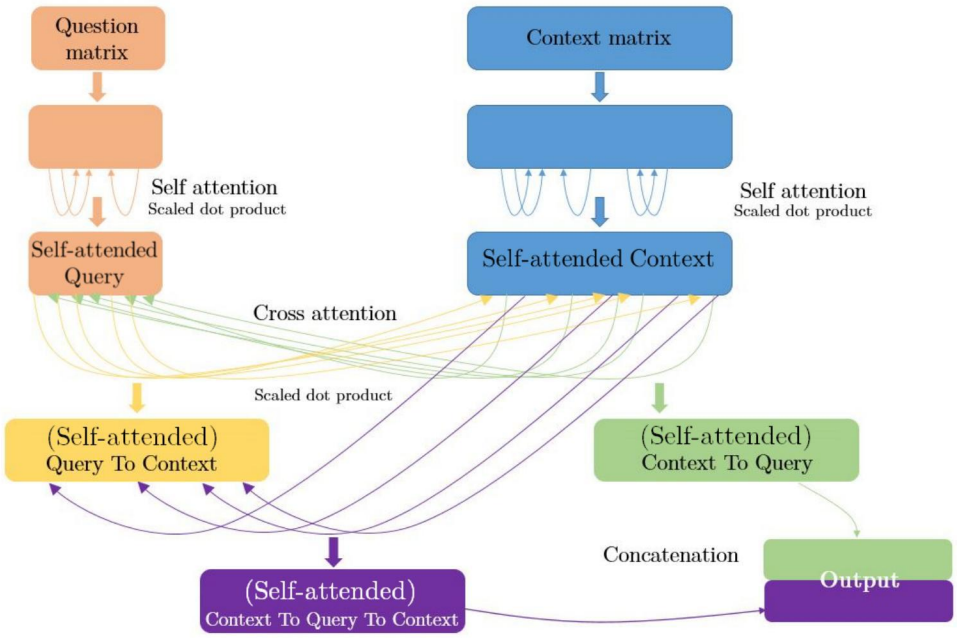

Figure 2: llustration of our architecture of self-attention with coattention. Context and query tokens are first passed through a self attention layer, before being used for the two layers of coattention: self-attended context to self-attended query, and self-attended context to self-attended query to self-attended context.

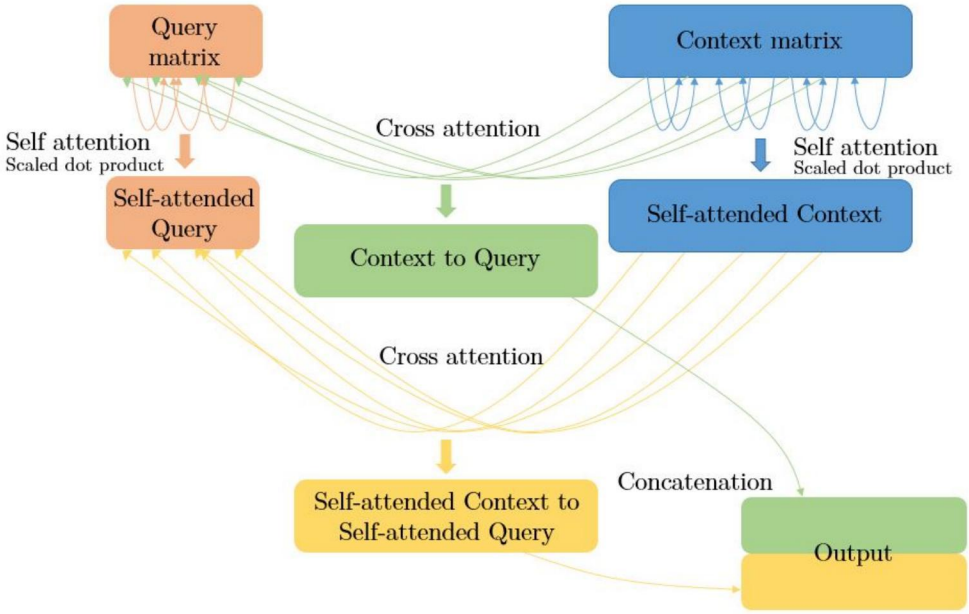

Figure 3: Illustration of our final combination of self-attention and coattention: the first layer of coattention is not modified, only the second layer of coattention is replaced with a self-attention and then a cross-attention of self-attended context to self-attended query.

In particular, this version of our attention layer takes as input the context hidden states $c_1, ..., c_N \in \mathbb{R}^l$ and the query hidden states $q_1, ...q_M \in \mathbb{R}^l$, obtained from the encoder layer. These hidden states are passed through the first layer of coattention, as described in the section above, the get the output tokens $a_1, ..., a_N \in \mathbb{R}^l$. We use the sentinel vectors to try to improve the performance of this part. At

the same time, the input hidden states are passed through the self-attention layer as described above. It outputs the vectors $c_1^{self-attended}, ..., c_N^{self-attended} \in \mathbb{R}^l$ and $q_1^{self-attended}, ..., q_M^{self-attended} \in \mathbb{R}^l$. These are used to compute a one-direction cross attention, by starting with an affinity matrix:

$$\forall i \in \{1, ..., N\} \quad \forall j \in \{1, ..., M\} \qquad L'_{ij} = c_i^{self-attendedT} q_j^{self-attended}$$

We then compute attention outputs of context to query. This attention is oriented in the same direction as the first layer of coattention, but the difference is that we run it on self-attended vectors. Therefore, we get attention vectors b_i for $i \in \{1, ..., N\}$ such that

$$\beta^{(i)} = \text{softmax}(L'_{i,:}) \in \mathbb{R}^M \qquad \text{softmax over the i-th row of } L'$$

$$b_i = \sum_{j=1}^{M} \beta_j^{(i)} q_j^{self-attended} \in \mathbb{R}^l \qquad \text{attention vector of the i-th context token}$$

At the end, we concatenate the second-level attention outputs b_i with the first-level attention outputs a_i, and feed the sequence through the bidirectional LSTM of the coattention. This gives us the outputs u_i of the attention layer:

$$u_1, ..., u_N = \text{biLSTM}([a_1; b_1], ..., [a_N; b_N])$$

For this version of attention, it is worth noticing that context and query vectors are handled in an asymmetric manner at each step:

- For the vectors run through the first layer of coattention, the query vectors are first prepossessed with a dense layer, as handled by the coattention.
- For the vectors run through the self-attention and the second cross-attention, the self-attention matrices are distinct for context and query vectors.

In general, we try to use non-linearities as frequently as possible, after using each self-attention matrix. We also try different number of heads, and query-key-value space dimensions such that either $l = num_{heads} \times att$ or not, where l is the input size dimension. In particular, we do not respect the equation to make the query-space dimension times the number of heads smaller than the input dimension, so that we may save some computational time.

To accelerate the learning process, we also try a version where we replace the value vectors with the initial vectors passed as input of the attention layer. This way, the model may benefit from the knowledge already contained in the embedding and the encoder layer.

In our experiments, if the trained model uses this version of the *Attention Layer*, we will note **CoSelfAttMerged**

3.2.2 Trilinear Self-Attention

Our last step in improving the attention layer was to use the benefits from both self-attention and coattention, as explored in the previous section, but also take advantage from the BiDAF attention layer and its trilinear attention. As described in the QANet paper [5], this cross-attention mechanism seems to provide a good inductive bias to our model. Instead of sticking with dot-product cross-attention as in the coattention paper, we therefore choose to replace it with the trilinear attention, and keep the other improvements made by combining self-attention with coattention.

In particular, this layer takes as input the hidden states as preprocessed by the encoder layer, $c_1, ..., c_N \in \mathbb{R}^l$ for the context and $q_1, ..., q_M \in \mathbb{R}^l$ for the query. These tokens are then passed through a self-attention layer, as described previously, except that the matrices $W_C^Q, W_C^K, W_C^V \in \mathbb{R}^{att \times l}, W_C^O \in \mathbb{R}^{l \times att}$ and $W_Q^Q, W_Q^K, W_Q^V \in \mathbb{R}^{att \times l}, W_Q^O \in \mathbb{R}^{l \times att}$ are now put in common between the context and the query. They form one unique set $W^Q, W^K, W^V \in \mathbb{R}^{att \times l}, W^O \in \mathbb{R}^{l \times att}$. This is due to the fact that the trilinear attention will provide different weights for the self-attended context, the self-attended query, as well as their element-wise multiplication. We can therefore delegate to the

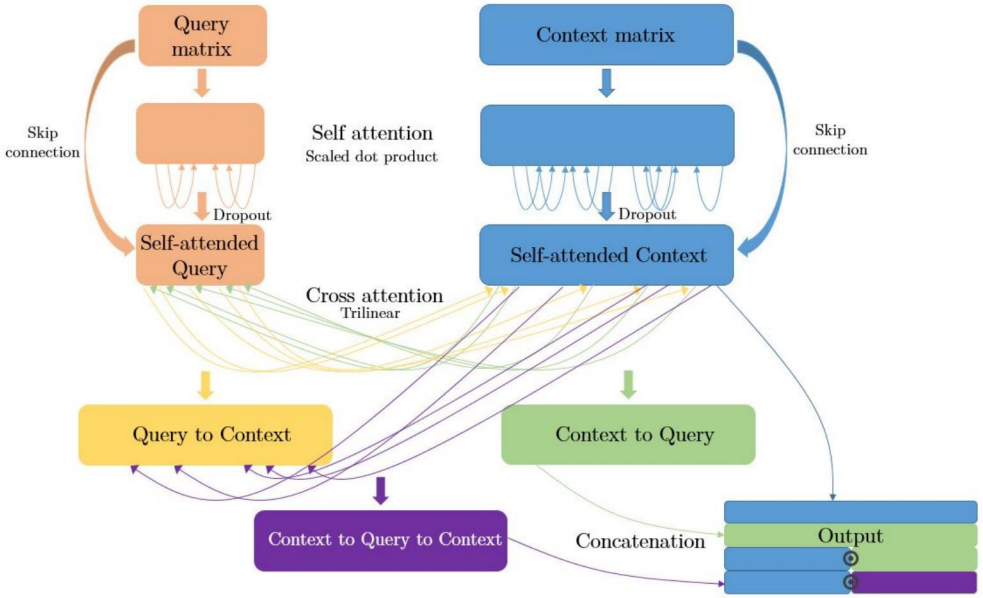

Figure 4: llustration of our architecture of self-attention with trilinear attention. Context and query tokens are first passed through a self attention layer, before being added to the input vectors through a skip connection. Then we compute the matrix of self-attended context to self-attended query, and self-attended context to self-attended query to self-attended context. The vectors are concatenated in an output that is not passed through a BiLSTM such as in coattention.

trilinear attention mechanism the task of asymmetrizing query and context, instead of the previously used separate self-attention matrices, or before that the preprocessed query tokens through a single dense layer. In addition, reducing the number of weights in the self-attention layer will hopefully fasten the computations and ease the convergence. Note that when we use multiple heads, we still have several sets of these matrices (but for each head, in common for the query and the context).

The self-attention layer then outputs the vectors $c_1^{self-attended}, ..., c_N^{self-attended} \in \mathbb{R}^l$ and $q_1^{self-attended}, ..., q_M^{self-attended} \in \mathbb{R}^l$. Here, instead of directly inputting the vectors into our cross-attention, we choose to use a skip connection. The goal is to accelerate and ease the training process, in particular at the beginning. For the version of coattention with self-attention, where we had a self-attention before the two layers of coattention, we had observed difficulties, without using any skip-connection (further discussed in the analysis section).

We form the following vectors:

$$\forall i \in \{1, ..., N\} \qquad c_i^{self-attended'} = c_i^{self-attended} + c_i \in \mathbb{R}^l$$

$$\forall j \in \{1, ..., M\} \qquad q_j^{self-attended'} = q_j^{self-attended} + q_j \in \mathbb{R}^l$$

Then, these vectors are used to create an affinity matrix, defined as:

$$\forall i, j \qquad L_{ij} = W_{trilinear}[q_j^{self-attended'}; c_i^{self-attended'}; q_j^{self-attended'} \odot c_i^{self-attended'}]$$

Using these affinity weights, we then compute the context to query, query to context and context to query to context matrices, as in the coattention layers previously defined. This gives us the vectors a_i for $i \in \{1, ..., N\}$ (context to query) and b_i for $i \in \{1, ..., N\}$ (context to query to context). The vectors are concatenated into the final output matrix, defined as:

$$\forall i \in \{1, ..., N\} \qquad s_i = [c_i^{self-attended'}; a_i; c_i^{self-attended'} \odot a_i; c_i^{self-attended'} \odot b_i] \in \mathbb{R}^{4l}$$

It is important to note that we use this complex concatenation to transmit more information, instead of using a smaller matrix then inputted into a BiLSTM such as in the coattention mechanism. This idea, which comes from the BiDAF model, enables us to get rid of the BiLSTM weights, what is counterbalanced by the additional trilinear attention weights. In addition, the goal is to fasten the computations by removing this BiLSTM.

The general architecture of our combined trilinear self-attention with coattention model is summarized on the figure 4. In our experiments, if the trained model uses this version of the *Attention Layer*, we will note **TriSelfAtt**.

3.3 Character-level Embeddings

Following the idea developed in [6], we modified the word embedding layer to include character level information. Given character embeddings of the letters of a given words, we can judiciously merge them with convolutions to get another complementary word representation.

More precisely, let's denote $c_1, c_2, ...c_k \in \mathbb{R}^d$ the k d-dimensional pretrained characters embeddings of a given word w. We first define 3 channel size $s_2, s_3, s_4 \in \mathbb{N}$ such that $s_2 = s_3 = \left\lfloor \frac{H}{3} \right\rfloor$, and $s_4 = H - s_2 - s_3$. (Where $H \in \mathbb{N}$ is the target hidden size for our final word embeddings)

Then, we apply three one dimensional convolutions (denoted C_2, C_3 and C_4) on the zero-padded sequence $[0_{\mathbb{R}^d}, c_1, c_2, ...c_k, 0_{\mathbb{R}^d}]$ with output channel sizes respectively being s_2, s_3 and s_4 and kernel sizes respectively being 2, 3 and 4. (And of course, given the input, input channel sizes is d for the 3 convolutions).

We further max-pool each channel output to keep only one coefficient for each channel. For example, C_2 applied on our sequence $[0_{\mathbb{R}^d}, c_1, c_2, ...c_k, 0_{\mathbb{R}^d}]$ will output a $(k+1)$-dimensional vector for each output channel. We thus take the maximum of theses $k+1$ coefficients, and concatenate these s_2 maxima (one for each channel) to get a representation of w of size s_2. We then concatenate the 3 representations get with the 3 convolutions C_2, C_3 and C_4, so that finally we obtain on character-based representation of w of size $s_2 + s_3 + s_4 = H$. The figure 5, adapted from [6] gives a better visual explanation of this process.

Then, we concatenate this H-dimensional character based representation of w with the other H-dimensional representation of w, obtained from the projection and sigmoid-activation of the GloVe embedding for w. The concatenated $2H$-dimensional vector is further projected with a linear layer to a H-dimensional space, so that it can be fed into a Highway Network [7] exactly as presented in the project handout.

In our experiments, if the trained model uses this version of the *Embedding Layer*, we will note **CharEmb**

3.4 Conditioning End Prediction on Start Prediction

The BiDAF baseline predicts the start location and the end location independently, given the final layer's activations, whereas one can think that they depend on each other. Using the Answer Pointer architecture [8], we can take into account this dependency to replace the *Output Layer*.

Given a representation of the context words $C_r = [c_1, ..., c_N] \in \mathbb{R}^{N \times l}$ (in our case, each vector c_i is the concatenation of the outputs of the `Attention Layer` and the `Modeling Layer`), we build the output probabilities in the following way:

$$h_s = 0_{\mathbb{R}^l} \qquad \text{initial hidden state of a RNN cell of hidden size } l$$

$$F_s = \tanh\left(C_r.V + e_N \otimes (W_a h_s + b_a)\right) \in \mathbb{R}^{N \times l}$$

$$\beta_s = \text{softmax}(F_s.v) \in \mathbb{R}^N \qquad \text{attention weights to the first hidden state } h_1$$

where $V \in \mathbb{R}^{l \times l}$, $W_a \in \mathbb{R}^{l \times l}$, $b_a \in \mathbb{R}^l$ and $v \in \mathbb{R}^l$ are learnable parameters, and $e_N \otimes$ simply mean repeating N times the multiplied vector (so that $e_N \otimes (W_a h_0 + b_a)$ is a $\mathbb{R}^{N \times l}$ matrix whose all rows are equal to $(W_a h_0 + b_a)$).

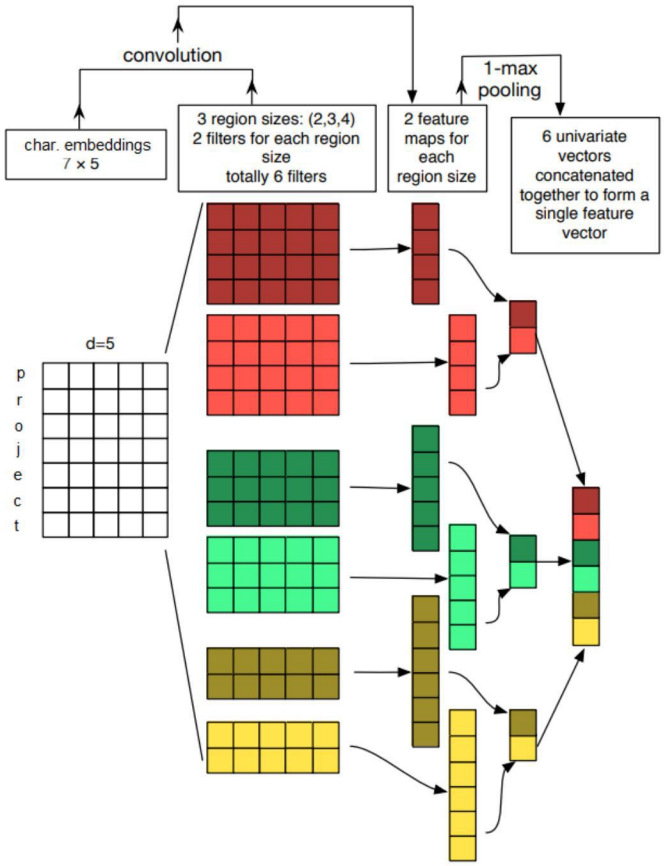

Figure 5: Illustration of a CNN architecture to build a word embedding from character embeddings (source [6]). We depict three kernel sizes: 2, 3 and 4, each of which has $s_2 = s_3 = s_4 = 2$ filters (thus $H = 6$ on this illustration)

β_s if the probabilistic output for the starting point of the answer span. Then, the resulting attention $a_s = \sum_{i=1}^{N} (\beta_s)_i c_i \in \mathbb{R}^l$ if fed to a RNN cell with hidden state h_1 and we repeat the same procedure. Thanks to this feeding, the end probability vector depends on the start probability vector.

$$h_e = \text{RNNCell}(a_s, h_s) \qquad \text{initial hidden state of a RNN cell of hidden size } l$$

$$F_e = \tanh\left(C_r.V + e_N \otimes (W_a h_e + b_a)\right) \in \mathbb{R}^{N \times l}$$

$$\beta_e = \text{softmax}(F_e.v) \in \mathbb{R}^N \qquad \text{attention weights to the second hidden state } h_e$$

β_s if the probabilistic output for the end point of the answer span. In our experiments, if the trained model uses this version of the *Output Layer*, we will note **CondOutput**

3.5 Hyperparameter optimization and training scheduler

Aside from the architecture modifications, we decided to explore a portion of the hyperparameter space as well as some fancy training methods. In particular, we implemented a learning rate finder, using a preexisting algorithm [3], in order to quickly explore the learning values that are convenient for our model. This was very useful to avoid unsuccessful runs, which were costly both in terms of computational time and credits. The learning rate finder output can be seen on figure 6. The optimal learning rate value can be the argmin of the loss divided by ten (at least a good candidate).

[3] `https://https://github.com/davidtvs/pytorch-lr-finder/`

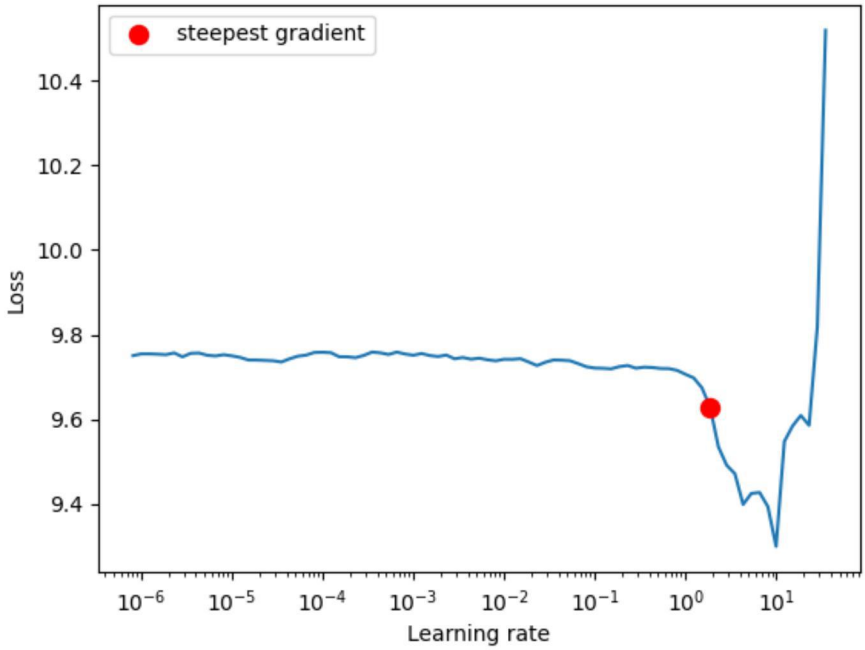

Figure 6: LR finder output. At each learning rate value, the finder runs a few batches and measures the evolution of the loss function. Therefore, it gives insights on which values are susceptible to lead to the steepest loss decays. In particular, from our experience and the recommendations in the literature, we start by a value of 1/10 the argmin of the loss on the plot.

In addition, we try to use higher values of the learning rate, in particular at the beginning of the training. This can be done using a learning rate scheduler, for instance by steps. At the same time, we reduce the other forms of regularization, by increasing the batch size and reducing the dropout, to allow hopefully for higher values of the learning rate. We also explore several initialization methods of the attention layer weights and select the most appropriate one.

3.6 Final model architecture

We provide an illustration of our final model architecture on figure 7.

4 Experiments

4.1 Data

We used the SQuAD 2.0 dataset [4] to test our different candidate architecture. Our training set contains 129,941 examples, our dev set 6078 examples and our test set 5915 examples. Each example consists in a tuple (question, context, answer).

In addition, the SQuAD 2.0 dataset contains around 33% of unanswerable questions [4] given the context (that is the amswer to the question is not present in the context), which forces our model to adapt.

4.2 Evaluation method

As explained in the project handout, performance is measured via two metrics: **Exact Match (EM)** score and **F1 score**. We cannot dive more into this topic given the limited space we have, but as previously, the reader can refer to `http://web.stanford.edu/class/cs224n/project/`

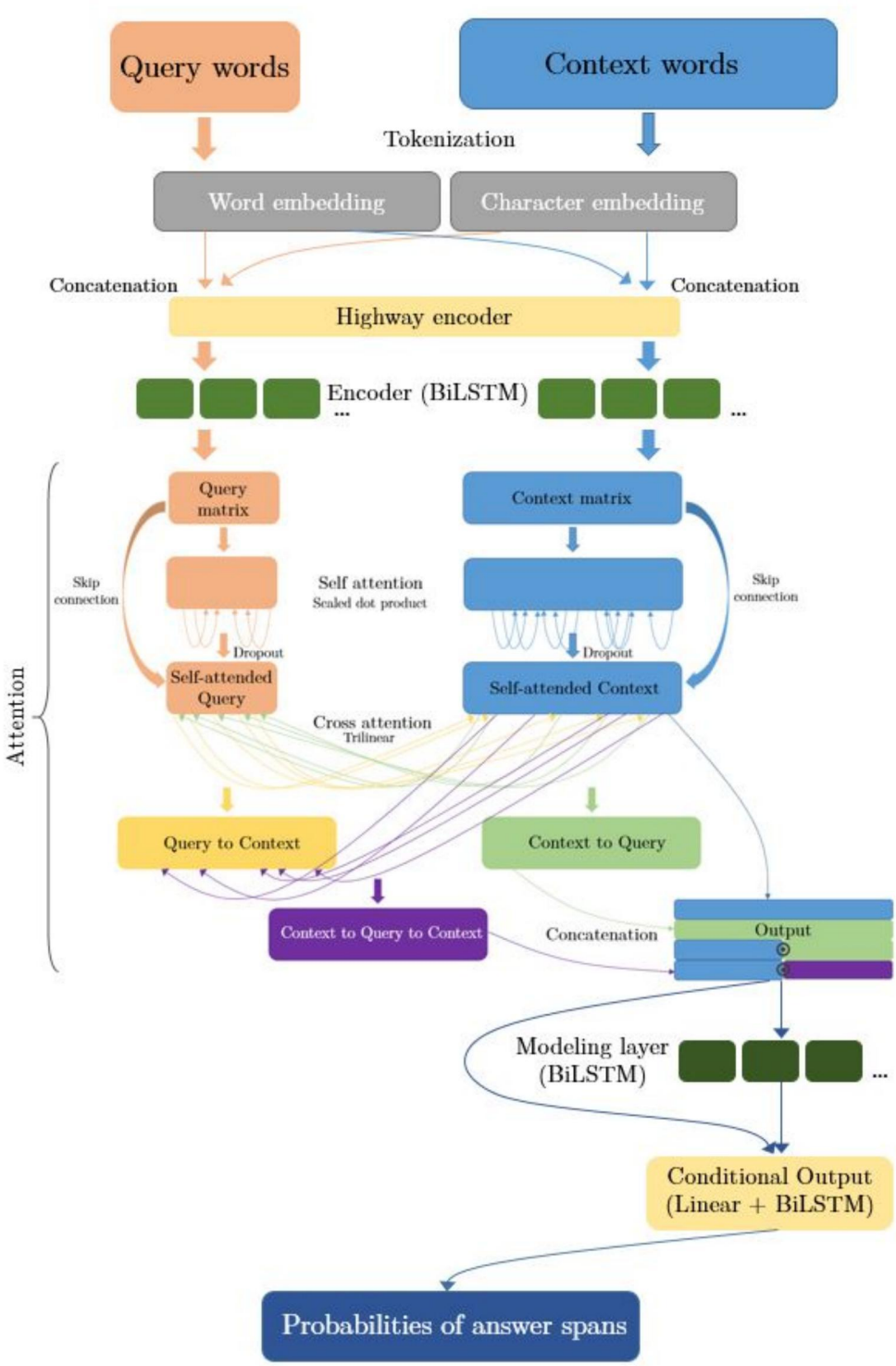

Figure 7: Illustration of our final model architecture. It combines the basic archicture of the BiDAF model, including a character embedding aside the word embedding, with our attention layer: combining self-coattention with trilinear cross-attention and a portion of coattention mechanism. Finally, the output layer is a conditional output.

`default-final-project-handout-squad-track.pdf` (page 2) where these two metrics are properly defined for our task.

4.3 Experimental details

The final hyperparameters we used for all our trainings are finally:

- Batch size: 64
- Drop out probability: 0.2
- Exponential Moving Average weight decay: 0.999
- Number of training example between each evaluation on dev set: 50 000
- Hidden size: 100
- Learning Rate: 0.5
- Number of epochs: 30
- Typical associated training time: 10h to 15h each time

These were determined using a learning rate finder as well as a reasonable hyperparameter space exploration.

4.4 Results

Model	EM	F1
BiDAF (baseline for all layers)	58.01	61.31
BiDAF + CoAtt	55.47	59.33
BiDAF + CoAttWithSent	56.41	60.08
BiDAF + CoSelfAttStacked	52.19	52.19
BiDAF + CoSelfAttMerged	55.6	59.32
BiDAF + CharEmb	62.29	65.38
BiDAF + CharEmb + CoAtt	59.27	63.23
BiDAF + CharEmb + TriSelfAtt	62.71	66.22
BiDAF + CharEmb + CondOutput	62.36	65.86
BiDAF + CharEmb + TriSelfAtt + CondOutput	**62.73**	**66.28**

Figure 8: Comparisons of the models performance per architecture on the dev set

Finally, as expected, our best model is the more complex one: **BiDAF + CharEmb + TriSelfAtt + CondOutput**. It achieved a performance of EM = 62.730 and F1 = 66.283 on the dev set, and **EM = 60.490 and F1 = 64.081 on the test set**.

Overall, we were a bit disappointed by the coattention, which was said to be a game changer [1]: we observed results that never beat the baseline, when implementing the coattention layer as originally described in its paper.

On the contrary, modifying more heavily the coattention layer helped us increase the performance compared to the original model. By adding self-attention, using a trilinear cross-attention in place of the dot-product cross-attention, and removing the final BiLSTM of the coattention layer we achieved higher scores than our BiDAF baseline.

It is important to note that the biggest improvement we can see from figure 8 is the **CharEmb** layer. There is a huge performance gap between *BiDAF* and *BiDAF + CharEmb*.

5 Analysis

Looking at figure 8, it seems that the largest gap in terms of performance was made possible when we added the character embeddings. This can be explained by the fact that a lot of answers in the dataset requires dates or person/location names, which are probably out-of-vocabulary if we only use pretrained embeddings. Using character embeddings helps to build a relevant representation for all theses words and improved significantly the accuracy.

Exploring various modifications to the attention mechanism was very interesting to better capture the role of attention, in particular for the QA task. At the end, when we compare the original BiDAF

model with character embeddings, and our modified BiDAF model with our own attention layer and the character embeddings, we see an improvement of +0.42 EM score and +0.84 F1 score. If we compare with our original coattention implementation, we see an improvement of +7.24 EM score and +6.89 F1 score, even though this is certainly mostly due to the character embeddings. We would need to further explore our modified architecture to provide additional improvements and comparisons with existing architectures.

It is important to note that the self-attention layer needed numerous runs to adjust its hyperparameters. We explored a variety of hidden dimensions, sometimes distinct between the query-key space and the value-space. We tried various number of heads, and different relations between number of heads, embedding size and hidden dimensions of the self-attention matrices. At the end, we discovered that having the number of heads times the hidden dimension being equal to the original embedding size was a good equation to get good scores.

Aside the dimensions, using a skip connection in the self-attention layer really helped us ease the learning process of our algorithm. We do not consider anymore not using it.

We also explored modifications of the training process through hyperparameters and scheduling. We tried to use higher values of the learning rate, by reducing other forms of regularization, such as using a higher batch size and a lower dropout probability. Nevertheless, as we were using a constant learning rate across all the layers of our model, we observed catastrophic forgetting phenomenons[4]: we suspect that higher learning rate values induce too large variations of the embeddings, thus impacting the long-term training performance and the validation performance. This is why we chose at the end to use the original (and lower) learning rate value.

If we were to further improve the hyperparameter optimization, we would implement a discriminative learning rate accross the different layers, allowing us to use lower values for the embeddings (preventing any form of catastrophic forgetting) and higher values for the top layers.

6 Conclusion

Our final attention-aware attention model (A^3) is a combination of the existing BiDAF architecture, with an additional character embedding, a conditional output and a modified attention layer. In terms of performance, the character embedding was the most important improvement of our model. The conditional output is responsible for a minor part of the final model performance. The attention layer, which improved at the end the EM and F1 score by a reasonable amount (+0.42 EM score and +0.84 F1 score) is in its final configuration a combination of self-attention, trilinear cross-attention and coattention mechanisms.

We explored a variety of other modifications to the coattention layer to come up to its final version: we tried using self-attention to preprocess tokens before the original implementation of coattention; we then used self-attention in place of the second layer of coattention, which performed slightly better than the original coattention; and at the end we returned to having self-attention before the coattention, using a skip connection, replaced the dot-product cross-attention by a trilinear attention, and removed the final BiLSTM, replaced by a modified concatenated output matrix, as in the BiDAF implementation.

Our model certainly needs a more thorough hyperparameter optimization, in particular in regards to the self-attention layer: modifying the number of heads or the hidden sizes has a huge impact on the layer's performance. We would also like to use fancier training scheduler tools, such a discriminative learning rate, to enable the use of higher learning rates for top layers. This would enable us to avoid the catastrophic forgetting phenomenons that happened at certain stages of our exploration.

[4] `https://en.wikipedia.org/wiki/Catastrophic_interference`

References

[1] Caiming Xiong, Victor Zhong, and Richard Socher. Dynamic coattention networks for question answering. In *arXiv preprint arXiv:1611.01604*, 2016.

[2] Microsoft Research Asia Natural Language Computing Group. R-net: Machine reading comprehension with self-matching networks. In *https://www.microsoft.com/en-us/research/wp-content/uploads/2017/05/r-net.pdf*, 2019.

[3] Minjoon Seo, Aniruddha Kembhavi, Ali Farhadi, and Hannaneh Hajishirzi. Bidirectional attention flow for machine comprehension. In *arXiv preprint arXiv:1611.01603*, 2016.

[4] Pranav Rajpurkar, Robin Jia, and Percy Liang. Know what you don't know: Unanswerable questions for SQuAD. In *Association for Computational Linguistics (ACL)*, 2018.

[5] Adams Wei Yu, David Dohan, Minh-Thang Luong, Rui Zhao, Kai Chen, Mohammad Norouzi, and Quoc V. Le. Qanet: Combining local convolution with global self-attention for reading comprehension. In *arXiv preprint arXiv:1804.09541*, 2018.

[6] Ye Zhang and Byron Wallace. A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification. In *arXiv preprint arXiv:1510.03820*, 2015.

[7] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. In *arXiv preprint arXiv:1505.00387*, 2015.

[8] Shuohang Wang and Jing Jiang. Machine comprehension using match-lstm and answer pointer. In *arXiv preprint arXiv:1608.07905*, 2016.