GOOGLE GMAIL SMART COMPOSE

TECHNICAL REPORT

Giovanni Jiayi Hu
Department of Mathematics
University of Padua, Italy I-35121
Email: giovannijiayi.hu@studenti.unipd.it

June 2, 2020

ABSTRACT

This report describes an educational implementation of Gmail Smart Compose which predicts email completion within the browser application. The training involves the Enron email dataset and a sequence-to-sequence model implemented in Keras and then used within the browser with Tensorflow.js .

1 Gmail Smart Compose

Gmail Smart Compose [1] is a feature introduced in the popular Google email service back in 2018, which helps to save time on repetitive writing by suggesting relevant contextual phrases. In this report, I explore the approach taken to reproduce a Proof-of-Concept implementation of the same feature using the Enron Email Dataset [2].

1.1 Previous work

As the first step into the task, I searched for some information about the technical implementation. The Google Teams published a paper called "Gmail Smart Compose: Real-Time Assisted Writing" [3] in July 2019 covering much of the challenges and implementation details needed to try to experiment it on my own.

An additional article "Building Gmail style smart compose with a char ngram language model" [4] from the Machine Learning community helped to gather additional information and implementation details.

Then most of the code contributions came from the official Keras documentation for sequence-to-sequence models: "A ten-minute introduction to sequence-to-sequence learning in Keras" [5] and "Sequence to sequence example in Keras (character-level)" [6].

1.2 Environment limits

The project has been developed mainly using the Colab free plan, thus subject to the limit of 13GB RAM. Also, because of the educational purposes of the project, I aimed for training time within the range of 30-60mins.

2 Architecture

The fundamental task in Smart Compose is to predict a sequence of tokens of variable length, conditioned on the prefix token sequence typed by a user and additional contextual information. During training, the objective is to maximize the log probability of producing the correct target sequence given the input for all data samples in the training corpus.

2.1 Data

The biggest public email dataset available is the Enron email dataset, which contains approximately 500,000 emails generated by employees of the Enron Corporation. It was obtained by the Federal Energy Regulatory Commission

during its investigation of Enron's collapse. Despite its relatively big size (Google trained their models on billions of emails), the usage of this dataset has faced different challenges.

2.1.1 Previous email

An important information for the prediction of the completion is the content of the previous email in case the composed one is a response. Unfortunately, the Enron dataset is a flatten set of emails with no information about conversations other than the subject, such ash `Re: hello`. The subject, however, is too fragile to use to safely group emails as related, and it's also often missing as value within the samples. According to the paper, merely joining the subject and the previous email to the model input reduces the log perplexity (the metric used to evaluate the models) is reduced by 0.13, which is a significant improvement.

Another important consequence is that the only relevant information to use as input of the model is the sentence the user is composing.

2.1.2 Preprocessing

Preprocessing was a very important step of the project and required careful handling. This is the biggest difference I experienced compared to usual tutorials or exercises, where the dataset is just loaded, and it's already prepared for the task.

The emails are tokenized into words and essential punctuation marks like `. ? ! , '`. Other special marks like new lines are removed and multiple white spaces are compacted as single whitespaces.

Quoted and forwarded messages are removed from the dataset, along with emails longer than 100 words. The latter removal has been done for two purposes: long emails tend to be uncommon to be written again by the user and the increase the required memory and training time by a great amount. Likewise, sentences longer than 20 words are not considered. The model is aimed to learn common sentences which are usually below the threshold of 20 words.

This process resulted in approximately 57 000 sentences.

2.1.3 Data sequences

In order to generate the context of a sentence, we train the sequence-to-sequence model to predict the sentence completion from pairs of split sentences of variable length. For instance, the sentence `here is our forecast` is split in the following pairs within the dataset:

```
[
    ('<start> here is <end>', '<start> our forecast <end>')
    ('<start> here is our <end>', '<start> forecast <end>')
    ('<start> way to <end>', '<start> go !!! <end>')
    ('<start> way to go <end>', '<start> !!! <end>')
    ('<start> let's shoot <end>', '<start> for tuesday at . <end>')
    ('<start> let's shoot for <end>', '<start> tuesday at . <end>')
    ('<start> let's shoot for tuesday <end>', '<start> at . <end>')
    ('<start> let's shoot for tuesday at <end>', '<start> . <end>')
]
```

Unique tokens `<start>` and `<end>` are added to delimit the limits on the input and output sequences. This is both similar to usual Neural Machine Translation (NMT) where the model is given the task to predict from the input sequence `<start> The weather is nice <end>` the output sequence `<start> Il fault beau <end>`

This resulted in approximately 500 000 pairs of sequences.

2.1.4 Tokenization

The previous text corpora are transformed into sequences of integers (each integer being the index of a token in a dictionary) by using Keras `Tokenizer`. I also put a limit to the 10k most frequent words, deleting uncommon words from sentences. This reduces the number of parameters needed to learn in the model and thus the training time.

Sequences are then padded to have the same length, shuffled to avoid pairs from the same sentence to be contiguous, and they are finally ready to be served to the neural model.

2.2 Natural Language Model

Considering the limits of the available dataset, as mentioned in the previous section, the project used the Sequence-to-Sequence Model (seq2seq) similar to NMT, where the source sequence is the start of the sentence and the target sequence the completion. The original paper also uses an attention mechanism to provide better a understanding of the context, but it has been left as potential improvement of the project.

2.2.1 Training model

The following image provides an overview of the final seq2seq model used for training. At its core, it's an Encoder-Decoder model which uses GRU units to encode the input context as in Figure 1.

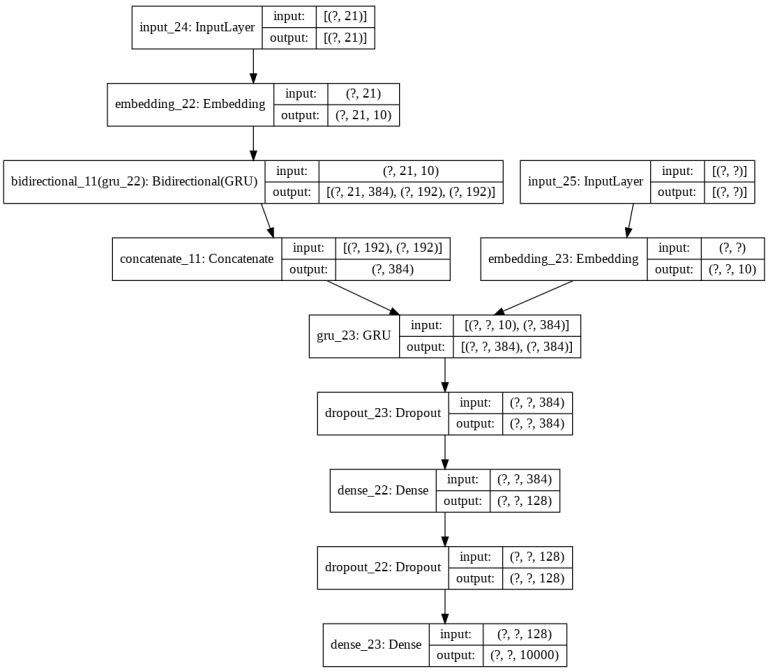

Figure 1: The layers of the seq2seq model

- The first layer takes as input the start of the sentences as sequences of integers returned by the `Tokenizer`and padded to have the same length, 21 integers.
- An embedding layer is used to improve the training. The dimension is set as 10 by following the general advice of using $vocabulary\_size^{0.25}$ as embedding size as suggested in "Introducing TensorFlow Feature Columns" [7]. The main motivation of the usage of an embedding layer is to introduce the ability to recognize that two sequences are similar without losing the ability to encode both sequences as distinct from the other, while sharing the statistical strength between the two of them and their context. For instance, the two sequences `Have a great` and `Have a good` should have the same completion `<start> weekend <end>`.
- The Encoder is comprised of a Bidirectional Gated Recurrent Units (GRU) layer. GRU units have been used as a gating mechanism to better encode the input sequences compared to common RNN. The latter suffers the vanishing gradient issue, especially with the case of the sequences in this dataset which reach a length of 20 words. Compared instead to Long short-term memory (LSTM) units, GRU units have been proved in practice to perform better with the task under consideration. Furthermore, GRU units have only one state, the Hidden State, whereas an Encoder with LSTM units requires working with both the Hidden States and the Cell states. Finally, since the usage of the Bidirectional model, both the forward and backward Hidden States are concatenated to form a single Encoded Hidden State.
- During the training, the model uses **teacher forcing** as shown in Figure 2. Specifically, it is trained to turn the target sequences into the same sequences but offset by one time-step in the future. After the model is trained, the `<start>` token can be used to start the process and the generated word in the output sequence is used as input on the subsequent time step. However, during the training, we obtain values which are likely

to be different from the ground truth and they set the model off track because every subsequently generated word will be based on a wrong history. For this reason, during the training, the model receives the ground truth output y^t as input at time `t + 1`. This explains the usage of an additional Input Layer and Embedding, containing the ground truth sequence.

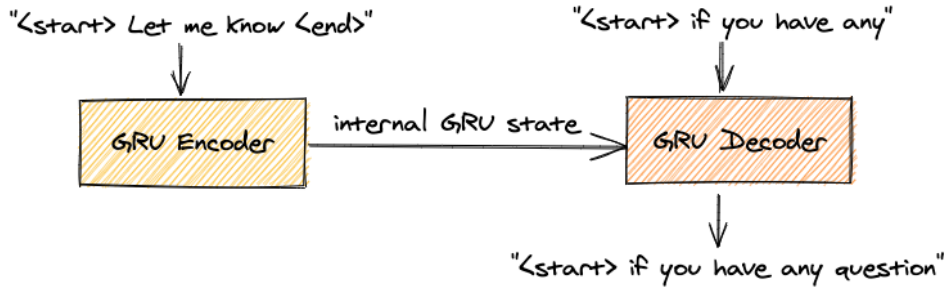

Figure 2: Teacher forcing during training

- The Decoder is comprised of a GRU layer which takes as input the Encoded Hidden State and the ground truth output from the previous time-step. Then a Dropout layer is added to its output to avoid overfitting, and an additional Dense layer is used to improve inference capabilities. Finally, the last Dense layer produces the logits for each word in the dictionary, assigning to each of them a probability of being the correct word to predict. Since the categories are provided as integers, the function used as loss is `sparse_categorical_crossentropy` whereas `perplexity` is the metric.

$$Perplexity(x) = -\sum_{x} p(x)\ log\ p(x)$$

x is the ground truth label and $p(x)$ is the model. The lower is the perplexity, and the higher is the probability of assigning the true target tokens. Perplexity is a typical metric used to evaluate language model and it's the same used in the Google paper.

In Table 1 the results of different architectures or hyper-parameters.

Architecture	# Parameters	Training time	Perplexity
Forward-only GRU 128 outputs with a Decoder Dense Layer of 128 units	1,614,032	32min	2.0357
Bidirectional LSTM 128 outputs with a Decoder Dense Layer of 128 units	1,938,640	35min	1.9327
Bidirectional GRU 128 outputs with a Decoder Dense Layer of 128 units	1,836,240	34min	1.8793
Bidirectional GRU 128 outputs with a Decoder Dense Layer of 256 units	3,149,136	61min	1.6835
Bidirectional GRU 64 outputs with a Decoder Dense Layer of 128 units	941,200	24min	2.3211
Bidirectional GRU 192 outputs with a Decoder Dense Layer of 192 units	2,895,120	57min	1.7118
Bidirectional GRU 192 outputs with a Decoder Dense Layer of 128 units	2,230,480	41min	1.8352

Table 1: Results of different model hyperparameters or memory units

Despite the best model is 128 GRU outputs with 256 units in the hidden Dense layer, the model has much more parameters and therefore incurs also in slower inference time later. Since inference latency is an important metric for the task under consideration, I preferred to keep the model with 192 GRU outputs and 128 units in the hidden layer. It is the second best-scoring model but with 2.2mln parameters compared to 3.2mln parameters of the best-absolute model.

2.2.2 Inference Model

During the inference, the model uses separately the Encoder to encode the input sequence whereas the Decoder will not be fed with the true output and it's approximated with the model's output at the previous time-step as shown in Figure 3.

The Encoder in Figure 4 uses the same layers of the training Encoder-Decoder.

Likewise, the Inference Decoder in Figure 5 uses the same layers as before, with the only difference that it uses the word predicted at the previous step as input along with the Encoded Hidden State.

The inference Decoder predicts the sentence completion by doing the following steps:

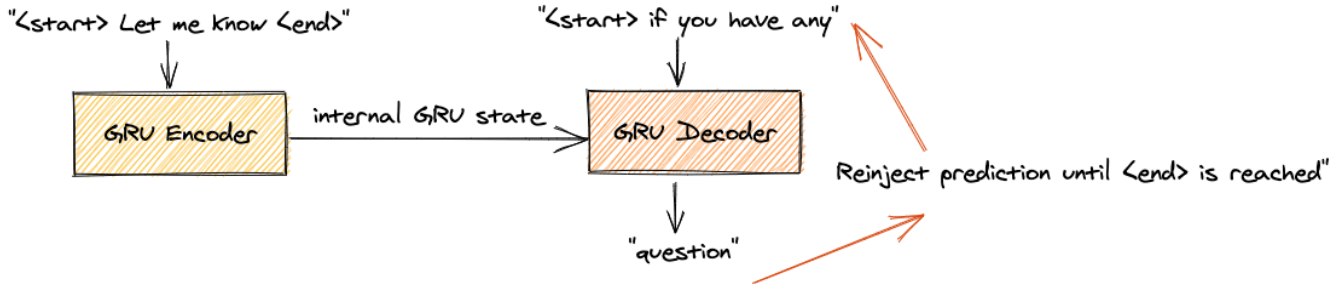

Figure 3: Prediction re-injected during inference

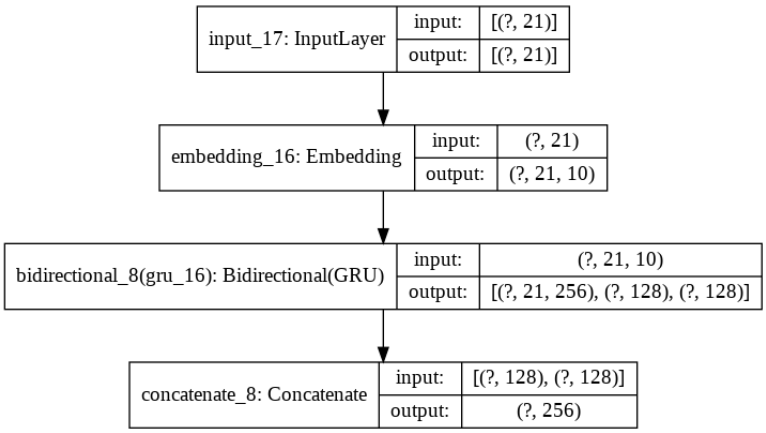

Figure 4: The Inference Encoder

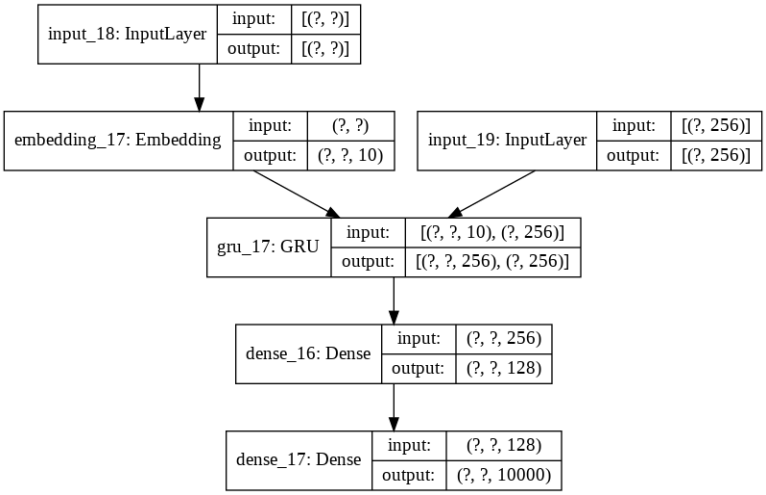

Figure 5: Layers in the Inference Decoder model

1. It tokenizes and embeds the input sequence using the same preprocessing of the training inputs
2. The Encoder returns the Encoded Hidden State
3. The target sequence starts with only the `<start>` token and is conditioned to generate an output work token based on the context provided by the Encoded Hidden State. The
4. Step 3 is repeated until the token `<end>` is generated. The final sequence is converted back from integer tokens to words using the reverse dictionary of the `Tokenizer`.

3 Evaluation

Below there are some results provided by the inference.

Input	Output
here is	the latest version of the usec transaction <end>
have a	good weekend <end>
please review	and let's discuss <end>
please call me	at <end>
thanks for	the help <end>
let me	know if you have any questions . <end>
Let me know	if you have any questions . <end>
Let me know if you	have any questions <end>
this sounds	good <end>
is this call going to	get a look at this ? thanks <end>
can you get	a look at this problem ? thanks <end>
is it okay	? <end>
it should	be the absolute latest <end>
call if there's	okay with you <end>
gave her a	call <end>
i will let	you know if you have any questions <end>
i will be	there <end>
may i get a copy of all the	invoices ? thanks <end>
how is our trade	? <end>
this looks like a	good idea <end>
i am fine with the changes	. <end>
please be sure this	is the morale booster <end>

Table 2: Input-Output predictions of the inference model

The predicted outputs are actually quite good. The inference model is especially good with short common sentences like *"Let me know if you have any questions"* or *"I will let you know"*. Some predictions also show that the predictions are personalized based on the Enron dataset, for instance in the case of *"here is the latest version of the usec transaction"*.

Unfortunately, some other cases such as *"call if there's okay with you"* or *"please be sure this is the morale booster"* seem to be a bit off. If we had a bigger dataset and more training time these completions should have been discarded as very unlikely to be correct.

4 Inference in the browser

Because the model must be used to predict email autocompletion, the next step is to save the model and make the inference on the fly while the user types. This can be done on the server or on directly in the browser. Because of the low-latency requirements and the cost of renting an ever-running server, the decision has been to try using the model within the browser.

For this goal, both the `Tokenizer` dictionary and the Keras model must be saved and loaded later in the browser. The word-to-integer dictionary is just saved as JSON, whereas I found that `H5` is the best working format for the Keras model. Attempts to use the Tensorflow `SavedModel` [8] resulted in errors while converting the model to the web using `tensorflowjs-converter` [9].

The model is then loaded in the browser along with the pre-trained weights using `Tensorflow.js` [10], a JavaScript implementation of the framework. In particular, because JavaScript executes in a single main thread shared with UI rendering, in order to avoid freezing the page while loading the model or during the inference calculations, all the Keras processing is done in a separate thread via `Web Worker` [11] as shown in Figure 6.

The main thread communicates with the Web Worker via messages, and the latter executes the tokenization of the input sequence and the subsequent model inference. Then the result is returned to the browser main thread via message again.

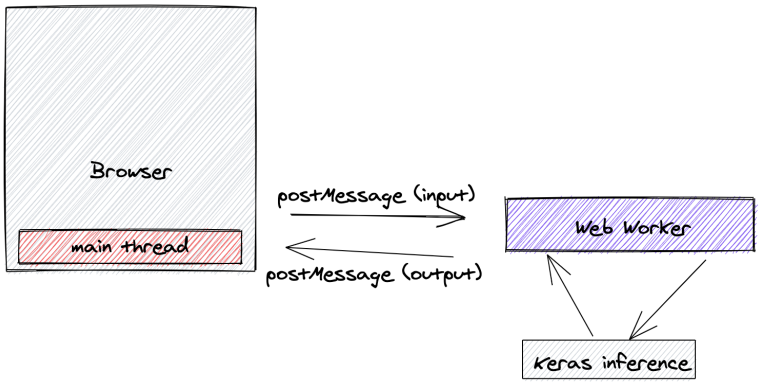

Figure 6: A representation of Model inference in the browser

The final source code are available on the repository at github.com/jiayihu/gmail-smart-compose. For easier evaluation, there is also a notebook on Colab.

5 Conclusions

The project has been the first real-world usage of NLP notions and Deep Learning techniques. The most important lessons have been:

1. Before even thinking about the learning model, there is a lot of work that needs to be done during the preprocessing phase. The available dataset to be used as input for the model assumes an important role on the subsequent training and goodness of the results. I would dare to say that data quality is as important as the training model. Unfortunately the Enron dataset is the only meaningful dataset but is lacking important information like linking responses.
2. Training real-world models requires a lot of RAM and computing power. The project has been trained using Colab free plan which limits the available RAM to 13GB and there are also limitations on the amount of GPU usage. Although this prohibited the training of more complex models or for a long time, at first it forced me to think better about the data structures I was using and the amount of input used to generate the dataset. I started to be more careful about what to include in the dataset, which resulted in both less RAM usage and more accurate inference predictions. At the same time, there are obviously strongs disadvantages in limited RAM and training time. According to the paper, Google ran their models on billions of emails and for at least 3 days using high-end TPUs. But for educational purposes, 13GB should be enough to learn something meaningful.

References

[1] "SUBJECT: Write emails faster with Smart Compose in Gmail". `https://www.blog.google/products/gmail/subject-write-emails-faster-smart-compose-gmail/`

[2] Kaggle - "Enron Email Dataset". `https://www.kaggle.com/wcukierski/enron-email-dataset`

[3] Mia Xu Chen et al. "Gmail smart compose: Real-time assisted writing". 2019. `https://dl.acm.org/doi/abs/10.1145/3292500.3330723`

[4] "Building Gmail style smart compose with a char ngram language model". `https://towardsdatascience.com/gmail-style-smart-compose-using-char-n-gram-language-models-a73c09550447`

[5] "A ten-minute introduction to sequence-to-sequence learning in Keras". `https://blog.keras.io/a-ten-minute-introduction-to-sequence-to-sequence-learning-in-keras.html`

[6] "Sequence to sequence example in Keras (character-level)". https://keras.io/examples/lstm_seq2seq/.

[7] "Introducing TensorFlow Feature Columns". https://developers.googleblog.com/2017/11/introducing-tensorflow-feature-columns.html

[8] TensorFlow.js. "Using the SavedModel format". https://www.tensorflow.org/guide/saved_model

[9] TensorFlow.js. "Model conversion". https://www.tensorflow.org/js/guide/conversion

[10] TensorFlow.js. https://www.tensorflow.org/js

[11] Mozilla Developer Network. "Using Web Workers". https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers