An Efficient Conversational Smart Compose System

Yun Zhu, Xiayu Chen, Lei Shu, Bowen Tan, Xinying Song,
Lijuan Liu, Maria Wang, Jindong Chen, Ning Ruan
Google Inc.
yunzhu@google.com

Abstract

Online conversation is a ubiquitous way to share information and connect everyone but repetitive idiomatic text typing takes users a lot of time. This paper demonstrates a simple yet effective cloud based smart compose system to improve human-to-human conversation efficiency. Heuristics from different perspectives are designed to achieve the best trade-off between quality and latency. From the modeling side, the decoder-only model exploited the previous turns of conversational history in a computation lightweight manner. Besides, a novel phrase tokenizer is proposed to reduce latency without losing the composing quality further. Additionally, the caching mechanism is applied to the serving framework. The demo video of the system is available at `https://youtu.be/U1KXkaqr60g`. We open-sourced our phrase tokenizer in `https://github.com/tensorflow/text`.

1 Introduction

Online conversations are happening in every corner of the world in every second. People relies on different channels like daily chatting apps, e.g. Messages, WhatsApp, or online customer service to communicate with friends, families, colleagues and even acquaintance. Within conversational applications, efficient and smart assistant functions for users are long-desired. Smart compose (Chen et al., 2019) is a well-known smart writing feature that actively predicts the next couple of words in real-time to assist human writing. It saves user's time by cutting back on repetitive idiomatic writing. In this paper, we build a smart compose system for chatting applications to improve user's communication efficiency and enhance user experience. The outlook of the demo is shown in Fig 1. We assume two or multiple users are discussing some random topic in a chat room. And smart compose is working under the hood to suggest what to type next. The full demo is in `https://youtu.be/U1KXkaqr60g`.

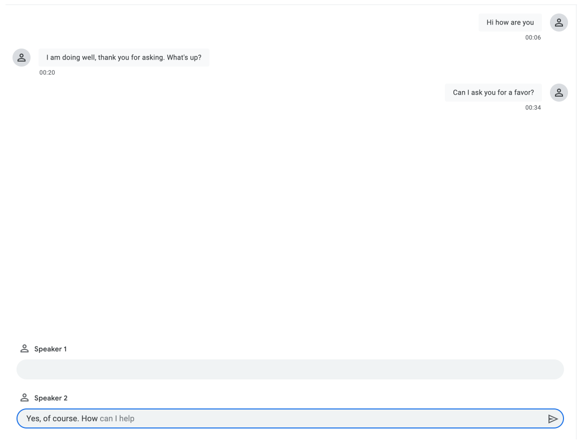
Hi how are you
00:06
I am doing well, thank you for asking. What's up?
00:20
Can I ask you for a favor?
00:34
Speaker 1
Speaker 2
Yes, of course. How can I help

Figure 1: Demo of smart compose experiment. We type in "How" in the input box, the smart compose system suggests "can I help" in real time, which is based on the conversation context.

To the best of our knowledge, although smart compose is a well-known application, it has not been well-designed for conversation scenarios. One basic requirement of smart compose is the low inference latency. If the next words are not shown up in less than 100ms, users will likely type them themselves. And there are two challenges out of this. The first challenge is the conflict between latency and long conversation context. During the human-to-human conversation within the chatting applications, there are multiple turns of conversation from multiple users, making the conversation history informative. Attention over the whole conversation history would bring too much latency. However, simply ignoring the conversation history or using an average embedding of history to represent previous context (Chen et al., 2019) is not exploiting the rich information within the conversational flow between users. Therefore how to effectively and efficiently extract information from conversation history is the key consideration for model designing. The second challenge comes from the fact that users would prefer a longer suggestion for conversation but generating a longer sequence is likely to involve more latency. In the auto-regressive generative model for example, the decoding time is linearly growing with the decoding steps. There is no clear solution on how to avoid the extra latency on the longer suggestion.

In this paper, we proposed a new solution to address above challenges and built an end-to-end cloud based smart compose system for human-to-human chatting applications. Our smart compose system can achieve high exact match rate with low latency (less then 100ms) under acceptable suggestion length. Our contributions are three-fold.

- We designed a novel architecture based on transformer-XL (Dai et al., 2019) [1] to effectively encode history and context information for composing. Also, it incorporates conversational features like user id and local position id to retrain the information effectively.
- We proposed a novel efficient tokenizer called phrase tokenizer which can increase the suggestion length for composing without sacrificing the quality and latency. This tokenizer can significantly reduce the latency of the system. We open-source our phrase tokenizer in `https://github.com/tensorflow/text` as an extension of the TensorFlow text library.
- We designed the cloud serving mechanism and addressed the practical issues such as caching.

2 Related work

One of the most successful applications for smart compose is Gmail (Chen et al., 2019), where the title of the email, the previous paragraphs, and the previous text of the current typing are combined to predict the next words. Following this success, similar models have also been used in other applications like document editing (e.g., Google Docs) or input method (e.g., Gboard). Although this feature is not new, previous work mainly focuses on general writing assist and not is specially designed for conversation scenarios.

Approach wise, there are two major categories. The first category is the dual encoder based model (Gillick et al., 2018; Karpukhin et al., 2020; Yang et al., 2019) which exploited contrastive learning: the predefined whitelist is built in advance, and the prefix tries to match the most possible one in the list. This kind of solution is also widely used in practical retrieval or recommendation systems. Although the dual encoder is extremely fast, the predefined whitelist limits the headroom for model quality from smart compose.

The second category is language modeling, a fundamental and indispensable component of many natural language processing. Recently it is becoming even more powerful with a larger model size (Floridi and Chiriatti, 2020; Thoppilan et al., 2022; Chowdhery et al., 2022). However, with the latency requirement, models should be designed smaller and more light-weighted for the smart compose (Van et al., 2020; Ciurumelea et al., 2020; Chen et al., 2019). Compared to dual-encoder, it removes the dependency on the whitelist and could achieve better matching accuracy. However, as it is involved with the autoregressive decoding process, the latency is generally higher and positive relative to the number of decoding steps. Although non-autoregressive based approaches (Zou et al., 2021) could potentially reduce the latency for text generation, it has been well-known to suffer from quality regression.

Our smart compose system is more conversational oriented and can achieve the best trade-off between quality and latency. Our model share the benefit of higher quality with the language modeling, but keep the latency low with proposed techniques.

3 Model Design

In this section, we will outline the modeling design. Firstly, we will first introduce the conversational inputs for the model and what features we are extracting from the conversation history. Then we will go through the model architecture used to take the conversational input and output the predicted suggestions. Finally, we will discuss how the loss is calculated during training and some implementation details.

3.1 Conversational Inputs

Our smart compose is based on conversational data, and our data processing strategy is to process the conversation with the sliding window of fixed size `N`, which is the number of conversation turns (in practice, we choose `N` equals 10). We tokenize all `N` sentences and concatenate the `N` turns together

[1] In practice, we found transformer based system has higher quality compared to LSTM and thus we build our system with Transformer.

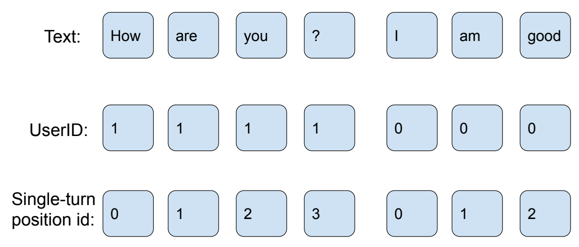

Figure 2: Example of conversational features.

to make a single input. Since the conversation could happen between multiple people, we also introduce the user id into the model input. The user id is associated with every token inside each turn conversation. Besides, we make use of the single turn position id. Different from the normal position embedding (Vaswani et al., 2017), which is calculated for all the input sequences, the single-turn position id is only for a sub-section of the input sequence. Furthermore, the N single-turn position id is concatenated together. The conversational features are in Fig 2. We found that using the user id and single-turn position id could achieve lower perplexity.

3.2 Model Architecture

We use separate embedding matrices to process the text id, user id, and single-turn position id. Specifically, we have separate embedding matrices for text-token, user-id, and single-turn position id. We add embedded vectors together as the final input embedding.

We feed this input embedding to our transformer decoder. Our architecture is a decoder-only Transformer XL (Dai et al., 2019) model. Note that the relative position is still in place, although we have an extra single-turn position id. The overall architecture is shown in Figure 3.

To reduce the latency, we adopted local attention (Beltagy et al., 2020) instead of full attention. We found that even with limited look-back length, the prediction can be accurate enough. By joint using Transformer-Xl and local attention, the transformer architecture is able to perform with lower computation, and similar to LSTM it can pass the state during processing the sequence. This helps capture the context information with little overhead. More details will be discussed in Section 5.

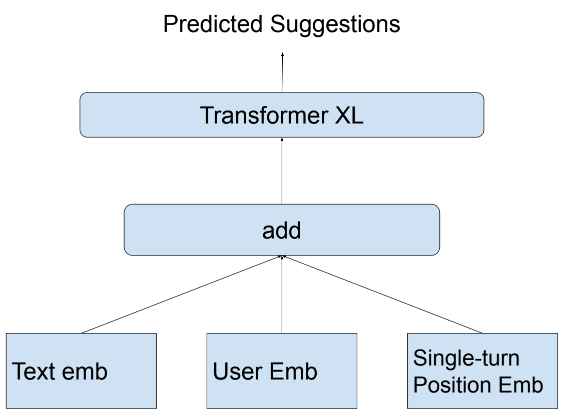

Figure 3: Model Architecture used for conversational smart compose.

3.3 Loss and Training Framework

We right-shift the input as the target and use the simple cross-entropy loss for the language model training. To be aligned with our sliding window data processing, we use a loss mask that only masks on the "last" turn. This is to ensure we are re-calculating the cross entropy for each turn. Our training framework is based on Tensorflow Model Gardens (Banna et al., 2021).

4 Phrase Tokenizer

Autoregressive decoding is the primary way of generating text with a Transformer decoder. One drawback of this mechanism is the latency issue: with N output units, the model will forward pass N times in a sequential manner. In this paper, we design a token that is beyond a single word (a phrase with n-gram) and a tokenizer that emits a group of units at a time, with the fallback mechanism to emit a single unit so that we can decrease the N (i.e., number of output tokens) here to reduce the latency. Note that the basic element of the phrase is a word following the findings of (Chen et al., 2019) that the word-based vocabulary can achieve better performance for smart compose. Nevertheless, our approach can also be applied to word-piece or sentence-piece models.

4.1 Ngram phrase Tokens

We build the Ngram dictionary based on the text. By definition, we treat P_1 as the set of unigrams with higher frequency. Likewise, to build P_2 we collect all word bigrams that occur in the training set with higher frequency (e.g., greater than a certain threshold) and augment this set of bigrams with the P_1. Moreover, we can build P_3 and P_k similarly.

4.2 Random Tokenizing Process

Once we have the Ngram phrase token set ready, the intuitive solution for tokenization is to tokenize the input text string from left to right to maximize the current `N` in NGram. This greedy method, however, can not work well in the smart compose application. The reason is that a user can stop at any word. Using the example of the phrase “How are you", if the model always treats this as a single token during training, it can never make the right next word prediction when the user types “How are". Therefore we should consider both phrase and individual word simultaneously.

To solve this problem we designed a random picking process when tokenizing the input text. The main idea is whenever there is multiple way to tokenize a phrase or sentence, we randomly pick one of them. We still enforce the left-to-right manner here. To efficiently find all the possible phrase from certain point, we leverage the Double Array Trie (Yata et al., 2007), which provides the function of iteratively finding the prefix match of a given text string. We introduce the additional parameter `prob` as input, which denotes the probability of choosing the current prefix match as opposed to finding a longer one. This randomness ensures that the model can see both phrase tokens and single-word tokens. We summarize our tokenization process in Algorithm 1.

Algorithm 1 Random-picking tokenization

```
Require: raw_input
Require: A Double Array Trie that contains all phrases trie
Require: The prob of emitting the phrase prob
0: tokens = []
0: wordlist = WhiteSpaceTokenizer(raw_input)
0: concat wordlist with space as input
0: len = len(input)
0: while do$i < len$
0:   matches = trie.IteratePrefixMatches(input(i:))
0:   for do$match \in matches$
0:     if RandomGen() > prob then
0:       $i+ = match.length$
0:       tokens.append($match.id$)
0:       Break
0:     end if
0:   end for
0: end while
0: Output tokens =0
```

5 Serving System Design

We will discuss the serving system design and heuristics in this section. Specifically, we will introduce how we design the inference graph to take advantage of the caching of conversation history. Besides, we will illustrate the server-side components of the system.

5.1 Inference Heuristics

To alleviate the burden of the server, we move complicated logic into the inference graph and serve this inside a standard servo. Specifically We put both the phrase tokenizer and the greedy(beam) search process inside the inference graph. We also customized the greedy or beam search process. To make sure `<UNK>` will not become the output, we regard the `<UNK>` token as a `<EOS>` token and update the log probability as the previous one when stopping at the `<UNK>`.

Since the conversation for chat is usually multi-turn and short, we use Transformer XL to encode the previous turns into model states and provide two sub-inference graphs to handle the caching and suggestions. The first sub graph takes in the previous conversation history and encode the states as

$$\text{States} = Model(\text{ConversationHistory}) \qquad (1)$$

The `States` is tensors of the intermediate output for TransformerXL. We will cache this `States` and do compose for any current user input as.

$$\text{Suggestions} = Model(\text{CurrentInput}, \text{States}) \quad (2)$$

5.2 Serving Flow

The serving infra of smart compose consists of the API front-end, prediction, and cache layers. The front-end layer receives conversational data, truncated according to sliding window size N, separated into the conversation history storing the previous conversation turns, and the current input storing the current message being typed. Such data can then be packed into sequential servo prediction requests sent to the prediction layer for states and suggestions. Furthermore, by adding the cache layer, we can further store the value of states keyed by their conversation history so that when multiple smart compose suggestions are requested at various keystrokes of the same current input message, the prediction request for a state can be swapped by a cache retrieval as the conversation history stays the same, and hereby reduce the latency.

6 Experiments

6.1 Data and Model Setup

Our experimental data is based on conversational data collected in existing conversational apps. We use the previous 10 conversation as conversation history and restrict the max sequence length as 256 tokens. For the model configuration, we used the the transformer with 8 attention heads. The input dimension as well as model dimension is set to 256 and the dimension of feed-forward network is set to 1024. We did not find dropout useful, so the models are trained without dropout. All the models for experiments presented in this paper are trained on 8x8 Dragonfish TPU with a per-core batch size of 32 (effective batch size of 4096). The learning rate schedule is ramped up linearly from 0 to $8.0e{-}4$ during the first 10,000 steps, and then it decays exponentially to zero till 500,000 steps. We use CPU for serving. The total model size is around 5MB.

6.2 Results

We mainly use the exact match to measure the accuracy of the model. For latency, the cpu based servo load-test is used.

6.2.1 XL and Local Attention

We first evaluate how much XL and local attention could help in the smart compose tasks. In the composing task, the latency is related to the number of decoding steps. Meanwhile, for the quality side, the more decoding steps, the worse the exact match.

Using local attention can effectively reduce latency because of two reasons. First, it will have a smaller payload for RPC requests, as the local attention span decides the tensor size of model states. Secondly, it reduces the model computation, especially the computation of self-attention. Table 1 compares the latency with different attention spans. We found that local attention can effectively reduce the latency by a large ratio. However using the local attention does not give much penalty for exact match. As shown in Table 2.

	Server latency (ms)	Total latency (ms)
full attn	19.3	23.4
attn=32	14.1	16.6
attn=8	12.6	13.4

Table 1: Comparison of Latency of transformer XL with location attention to its counterparts. Attn means the ”length of local attention".

	DS=1	DS=2	DS=3	DS=4	DS=5
full attn	88.6	69.1	56.5	49.4	45.9
attn=32	88.5	69.0	56.5	49.4	45.8
attn=8	87.4	68.0	54.6	48.4	44.7

Table 2: Comparison of Exact Match of transformer XL with location attention to its counterparts. "DS" means "decoding steps".

6.2.2 Phrase Tokenizer

We evaluate the effectiveness of the proposed phrase tokenizer in this section. To better illustrate the suggestion length, another critical factor for smart compose, we will look at two more additional metrics besides exact match. The first one is average length, which measures the length of the phrase (i.e., number of words) in the prediction when it is a correct prediction. The second one is effective length. This is also considering the correctness when calculating the prediction length, i.e., if the prediction is wrong, we will treat it as 0. Please note that these two metrics only give partial evidence of the composing quality.

For both the word level and phrase level tokens, we use the vocabulary of 60,000 tokens. For phrase level, we combine single words and phrases: we pick 30,000 single-word tokens and 30,000 phrase tokens which have 2-5 single words. We found that combining them can achieve the best quality.

We compare two tokenizers using the same model architecture, the word-based tokenizer, and the phrase-based tokenizer. The results are summarized in Table 4. For the word-based compose, we decode up to 5 steps. While for phrase-based compose, we only need to decode up to 2 steps. When the phrase-based model decodes two steps, and the word-based model decodes 4 or 5 steps, they achieve similar prediction length and exact match performance. In other words, with a phrase tokenizer, decoding two steps has the same quality effect as decoding 4-5 steps with a word tokenizer.

Prefix	Phrase tokenization	Steps	Word tokenization	Steps
Hi	"How are you"	1	"How", "are", "you"	3
I would	"like to thank you"	1	"like", "to", "thank", "you"	4
May I	"know when you have", "time"	2	"know", "when", "you", "have", "time"	5
Don't	"worry about", "it"	2	"worry", "about", "it"	3
How	"can I help you"	1	"can", "I", "help", "you"	4
We	"have to", "send it", "to", "him"	4	"have", "to", "send", "it", "to", "him"	6
I'll be	"more than happy to", "help you"	2	"more", "than", "happy", "to", "help", "you"	6
Can you	"tell me", "what is", "wrong"	3	"tell", "me", "what", "is", "wrong"	5

Table 3: Case Study of Phrase tokenzier VS Word tokenizer. When using the phrase tokenizer, fewer decoding steps is required to compose the same length as the word level tokenizer.

Metric	Tokenization	DS=1	DS=2	DS=3	DS=4	DS=5
Exact Match	Word	87.4	68.0	54.6	48.4	44.7
	Phrase	67.2	46.2	-	-	-
Average Length	Word	1	1.991	2.879	3.692	4.43
	Phrase	2.132	4.384	-	-	-
Effective Length	Word	0.874	1.345	1.572	1.786	1.98
	Phrase	1.432	2.027	-	-	-

Table 4: Performance comparison between phrase tokenizer and word tokenizer. The phrase tokenizer can achieve a similar composing length and exact matching rate with fewer decoding steps. "DS" refers to "decoding steps".

Latency (ms) / Steps				
1	2	3	4	5
3.9	7.5	9.5	12.6	14.8

Table 5: Latency Per steps.

However, decoding only two steps can achieve significantly lower latency, as shown in Table 5.

To illustrate how the phrase tokenizer behaves differently from the word level tokenizer, we show examples in Table 4. Our phrase tokenizer contains common phrases for daily conversations, such as "How are you", "can I help you", and "more than happy to". With these phrases as tokens, our solution can largely reduce the decoding steps and the latency. Furthermore, besides common phrases, phrase tokenizer contains single words as tokens, for example, "to", "it", "him", "wrong". These single-word tokens extend the granularity and diversity of tokens and ensure composing quality.

6.3 Demonstration

In this section, we demonstrate how the conversational smart compose system works. More details are described in the demo video. The user input interface is shown in Figure 1. While a user can type messages in the text box, the smart compose model will provide real-time suggestions. If the user is satisfied with the suggested text, the user can accept this suggestion; if not, the user can continue typing, and new suggestions will be shown along with user types. The lower latency comes from the model architecture, phrase tokenizer, and caching mechanism during serving.

7 Conclusion

In this paper, we demonstrate a simple and efficient conversational smart compose system. Our model is adopted from Transformer XL and effectively encodes history and context information to be used during composing. We proposed a novel and efficient tokenizer called phrase tokenizer to reduce latency for composing applications. Efficient serving heuristics and caching is also applied. With all the merit above, our demo achieved long, accurate and real-time typing suggestions for the conversation. We open-sourced the phrase tokenizer as a part of tensorflow text library.

References

Vishnu Banna, Akhil Chinnakotla, Zhengxin Yan, Anirudh Vegesana, Naveen Vivek, Kruthi Krishnappa, Wenxin Jiang, Yung-Hsiang Lu, George K Thiruvathukal, and James C Davis. 2021. An experience report on machine learning reproducibility: Guidance for practitioners and tensorflow model garden contributors. *arXiv preprint arXiv:2107.00821*.

Iz Beltagy, Matthew E Peters, and Arman Cohan. 2020. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*.

Mia Xu Chen, Benjamin N Lee, Gagan Bansal, Yuan Cao, Shuyuan Zhang, Justin Lu, Jackie Tsay, Yinan Wang, Andrew M Dai, Zhifeng Chen, et al. 2019. Gmail smart compose: Real-time assisted writing. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2287–2295.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2022. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*.

Adelina Ciurumelea, Sebastian Proksch, and Harald C Gall. 2020. Suggesting comment completions for python using neural language models. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 456–467. IEEE.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V Le, and Ruslan Salakhutdinov. 2019. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*.

Luciano Floridi and Massimo Chiriatti. 2020. Gpt-3: Its nature, scope, limits, and consequences. *Minds and Machines*, 30(4):681–694.

Daniel Gillick, Alessandro Presta, and Gaurav Singh Tomar. 2018. End-to-end retrieval in continuous space. *arXiv preprint arXiv:1811.08008*.

Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense passage retrieval for open-domain question answering. *arXiv preprint arXiv:2004.04906*.

Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, et al. 2022. Lamda: Language models for dialog applications. *arXiv e-prints*, pages arXiv–2201.

Hoang Van, David Kauchak, and Gondy Leroy. 2020. Autometss: the autocomplete for medical text simplification. *arXiv preprint arXiv:2010.10573*.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Yinfei Yang, Daniel Cer, Amin Ahmad, Mandy Guo, Jax Law, Noah Constant, Gustavo Hernandez Abrego, Steve Yuan, Chris Tar, Yun-Hsuan Sung, et al. 2019. Multilingual universal sentence encoder for semantic retrieval. *arXiv preprint arXiv:1907.04307*.

Susumu Yata, Masaki Oono, Kazuhiro Morita, Masao Fuketa, Toru Sumitomo, and Jun-ichi Aoe. 2007. A compact static double-array keeping character codes. *Information processing & management*, 43(1):237–247.

Yicheng Zou, Zhihua Liu, Xingwu Hu, and Qi Zhang. 2021. Thinking clearly, talking fast: Concept-guided non-autoregressive generation for open-domain dialogue systems. *arXiv preprint arXiv:2109.04084*.