Deep Learning – Cover Sheet

ASSIGNMENT

Instruction:

- Marks will be awarded for good presentation and thoroughness in your approach.
- Referencing Code: If you use some code, or ideas for code, which are taken or adapted from another source (book, magazine, internet, discussion forum, etc), then this **must** be cited and referenced using the APA Name convention within your source code. Failure to reference code properly is considered as plagiarism.
- Complete this cover sheet and attach it to your project.

Student declaration:

I declare that:

- *I understand what is meant by plagiarism*
- *The implication of plagiarism have been explained to me by our lecturer*
- *This project is all my work and I have acknowledged any use of the published or unpublished works of other people.*

Student Signature: *ckc* ..... Date: 29/02/2024 .....

Assignment Title	Blenderbot Space: A Persona Based Conversational Model	**Intake Code**	APDMF2303AI
Student Name	Chong Kah Chun Kelvin	**TPNo**	TP073978

Table of Contents

List of Figures

List of Tables

Blender-Bot-Space: A Persona Based Conversational Model

Abstract. Chatbot is one of the most popular application in Natural Language Processing, however there are various challenges involved when building a chatbot. In order for a chatbot to be successful, engagement and consistency is the key. Recent works have been focusing on developing an engaging chatbot through introducing personality to it. This study aims to develop an engaging chatbot through implementing persona information into the chatbot by introducing latent features space to memorize entailment relation and to facilitate discourse coherent. A pre-trained language model that followed a transformer-based architecture was used in this experiment, namely, the BlenderBot by fine-tuning it into a model capable of conducting persona consistent conversation. Experiment was done using two publicly available large datasets, ConvAI2 and DNLI to train and evaluate the performance of the conversation models. Evaluation metrics such as Hits@1, Perplexity, and F1 were done to compared with several state-of-the-art models as benchmark. The proposed model shows a good performance evaluated on the ConvAI2 test set.

Keywords Natural Language Processing • Deep Learning • Pre-trained Model • Chatbot • Latent Space

1.0 Introduction

Natural Language Processing (NLP) is one of the major fields in Artificial Intelligence (AI). In the early years, architectures such as Recurrent Neural Network (RNN) (McCulloch & Pitts, 1943) and Long Short-Term Memory (Hochreiter & Schmidhuber, 1997) have been successful in solving various NLP tasks but they are prone to vanishing gradients and long-term dependencies problems. Additionally, these models have very limited parallelization capability causing inefficiency in utilization of resources when training. With the recent proposed architecture call Transformer (Vaswani et al., 2017), some these weaknesses were mitigated, allowing models that built upon the transformer architecture to have state-of-the-art results. Popular pre-trained models that followed the transformers architecture includes Bert (Devlin et al., 2018), GPT (Liu et al., 2023), and BART (Lewis et al., 2019). These pre-trained models have contributed to the succession of various down-stream tasks through fine tuning or transfer learning. One of the tasks that were able to achieved excellent performance with the help of the pre-trained models is the conversational model. Conversational models or chatbots have

two major branches, namely, open domain and closed domain, open domain chatbots typically are more challenging to build as they need to be able to answer various types of queries by the user. Conversely, close domain chatbots answer only domain specific query, usually these types of chatbots obtained the responses through a knowledge based containing the domain information. There are two methods to build a chatbot, retrieval-based (discriminative) and generative-based (Pandey & Sharma, 2023). The decision of method depends on the tasks, usually task that are more knowledge dependent and require the response to be consistent uses retrieval-based method. On the other hand, generative-based methods are more suitable for open conversations. However, despite the availability of various pre-trained models trained on large amount of data there are still several concerns involved to make the chatbot practical in real world, such as lacking of personality (Chen et al., 2023) and response inconsistency (Chen et al., 2023). The aims of this paper are to study the methods to apply persona and improve consistency on responses generated based on the context (k previous dialogue) and apply the suitable methods on pre-trained encoder-decoder text generation model. This paper has 7 sections, **Section 2.0** will discuss about the major architectures applicable for building chatbots, **Section 3.0** discussed how deep learning can be implemented in various components that made up the chatbot, **Section 4.0** discussed the related works, **Section 5.0** discussed the model implemented in this paper, **Section 6.0** will show the implementation codes, **Section 7.0** discussed about the fine tuning and evaluation processes. **Section 8.0** shows the results and critical analysis for the experiment. The contributions of this paper are two-folds, first this paper can act as a proof-of-concept for the usability of the latent space technique (Chen et al., 2023) on other transformer-based architectures. Secondly, this paper added a new sequence classification layer on top of the original Blenderbot decoder layer to help the model generalize better on the learning task. As an example, the model will now include a classification loss to the final loss which force the model to generate responses closely related to the positive response.

2.0 Deep Learning Architectures

2.1 Sequential Models

Over the past decade, various deep learning techniques have been explored to solve different types of data as real-word data contains non-linear characteristics which deep learning excel in interpreting such patterns (Shickel & Rashidi, 2020). Since there are different types of data structures, different types of models have been developed to solve them, natural language

processing is considered as a sequential problem as utterances exhibit a sequential pattern (Shickel & Rashidi, 2020), hence sequential models are excellent choice for solving these problems. Popular sequential models for deep learning includes Recurrent based neural network (Schmidt, 2019) and Attention based neural network (Vaswani et al., 2017). In the following sections major sequential models will be discussed, for the experiment in this paper Transformers architecture was selected as the major building block for the implementation.

2.2 Recurrent Neural Network

RNN is one of the earliest sequential models for solving sequential data. The primary motivation behind RNNs was to capture temporal dependencies in time series, speech, and text data (Schmidt, 2019). **Figure 1** shows the differences between artificial neural network (ANN) and recurrent neural network (RNN). The main differences lie within the ability to capture temporal dependency, in an ANN, an input independent variable passed through the network and output a predicted value. This structure is effective for various classification or regression tasks but not effective for sentences input where the dependencies between words are important, to solve this, RNN was introduced. Similar to the ANN, RNN accept input word of

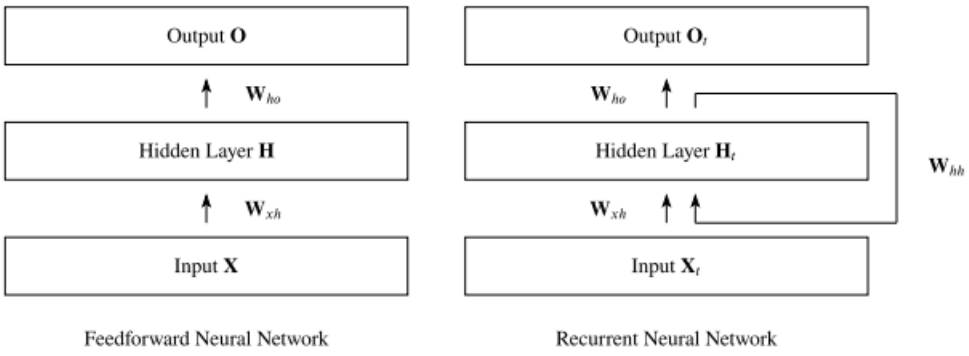

Figure 1. ANN vs RNN (Schmidt, 2019)

a sentence in order. Current input word represented as x^t, where t represent the current timestep are affected by word input from previous timestep x^{t-1} through passing hidden state between each corresponding hidden layer (Otter et al., 2020). Originally, each ANN layer output is computed as $\sigma(W_{xh}x + b_h)$, where W_{xh} is the weight vector of the hidden layer, b_h is the bias term of the hidden layer and σ is the activation function. In RNN however, each layer output is computed as $\sigma(W_{xh}x^t + W_{hh}h^{t-1} + b_h)$ (Xiao & Zhou, 2020) (Schmidt, 2019), where W_{xh} is the weight of hidden layer, $W_{hh}h^{t-1}$ is the dot product of previous hidden state and the hidden state weight vector and b_h is the bias term of the hidden layer and σ is the

activation function. This allowed the previous information (words) to be past to the future for prediction.

2.3 Long Short-Term Memory

Long Short-Term Memory (LSTM) is an architecture that based upon the RNN, the motivation behind this architecture is to mitigate the diminishing gradient problem in RNN (Staudemeyer et al., 2019). The diminishing gradient problem happen when there are long dependencies between words (long sentence), this make the backpropagation process difficult, as the gradient backward through each layer the gradient value starting to get smaller and smaller causing the value to be so small that it has no effect as backpropagation happen. The LSTM have a memory mechanism that pass through the whole sentence to memorize important information in each time step (Van Houdt et al., 2020). In order to decide what information that should be memorize there are 3 types gates, namely, the forget gate, the input gate, and the output gate as shown in **Figure 2** (Van Houdt et al., 2020). The forget gate ca be denoted as $F^{\sigma(Wf[h(t-1)+x(t)]+Bf)}$, where $h(t-1)$ represent the previous hidden state, $x(t)$ represent the current time step input, Wu represent the weight vector and Bf represent the bias term for the forget gate respectively and σ represent the sigmoid activation function. Similarly, the input and output gates have

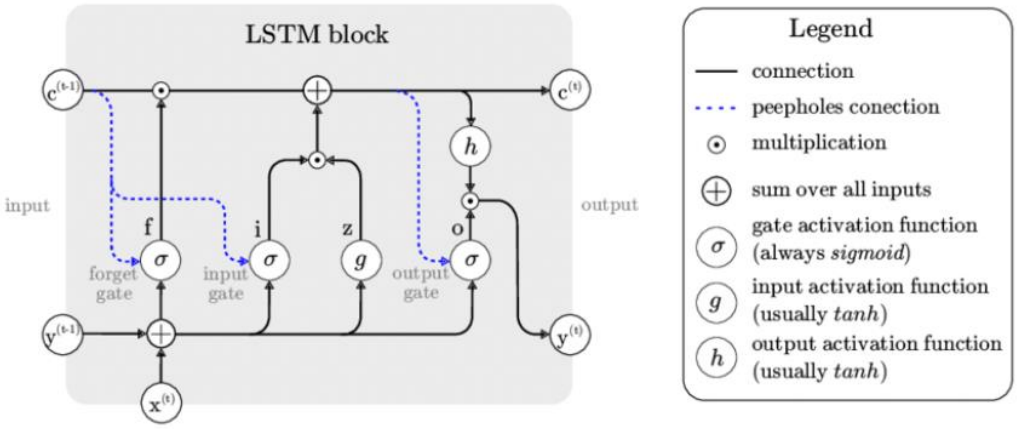

Figure 2. Long Short-Term Memory (Van Houdt et al., 2020)

similar notation of $I^{\sigma(Wi[h(t-1)+x(t)]+Bi)}$ and $O^{\sigma(Wo[h(t-1)+x(t)]+Bo)}$ with their own learnable weight vector. The forget gate decide which information in the cell state $c^{<t-1>}$ should be discarded. The input gate decides whether the current input contain important information that should be updated to the cell state $c^{<t>}$. The output gate is use to output the hidden state of current input step. In short, similarly to RNN the LSTM also output a hidden state $a^{<t>}$ at each time step which slight modification of $a^{<t>} = O * \tanh\ c^{<t>}$, which essentially uses cell state

information instead of previous hidden state information. The formal notation of cell state is $c^{<t>} = U * z^{<t>} + F * c^{<t-1>}$, where $z^{<t>} = \tanh(Wc[h(t-1) + x(t)] + Bc)$ is the current candidate for the cell state.

2.4 Transformers

Transformers is the recent attention-based model. As of the time of writing this paper, transformer is the most popular and effective sequential model for solving sequential problem. It's attention mechanism allow it to mitigate the vanishing gradient problem of the recurrent-based model and also allow it to run parallelly unlike recurrent networks where previous input need to be done then only the next input can be compute (Sarker, 2021). **Figure 3** shows the architecture of the transformer, a transformer consists of two main layer the encoder layer and the decoder layer (Vaswani et al., 2017). The encoder layer is in charge of encoding the input words into contextual feature representation for each word. The decoder layer is in charge of using the encoded feature representation to predict the output based on the task. Using a text translation task as example, the text to be translated will be passed into the encoder and output a set of feature vectors, then the decoder will be fed with the translated labels during training which passes through the masked multi-head attention component to mask the future and current predicting labels, leaving the previous time step labels for predicting the masked position's label. This

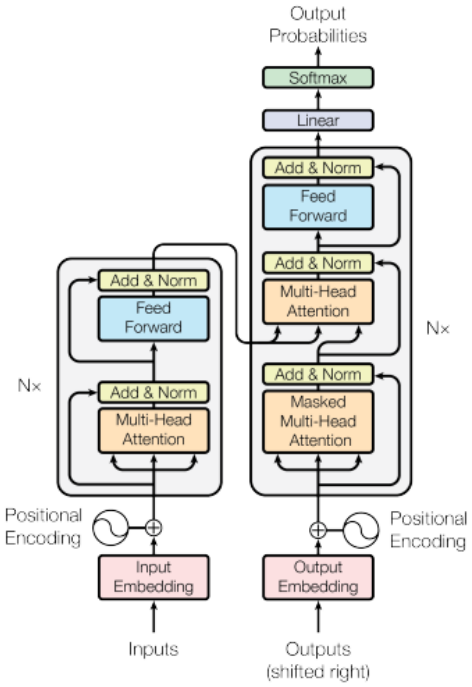

Figure 3. Transformer Architecture (Vaswani et al., 2017)

mechanism allows the model to learn more effectively by addressing prediction one word at a time. This architecture also has better flexibility. For instance, instead of sequence labelling task (predicting word by word) we want to do classification task, a shallow layer can be added on top of the decoder to output probability distribution for the classes instead.

Figure 4 shows the attention layer of the overall transformer architecture. The attention layer is a shallow network, which is also the main component of transformer-based models. On the left of Figure 4 is the self-attention computation flow. Generally, each word in a sequence will be passed to a self-attention module, which have 3 component Query (Q), Key (K), Value (V) (Vaswani et al., 2017) so a self-attention of a word is essentially $A(q, K, V)$ which

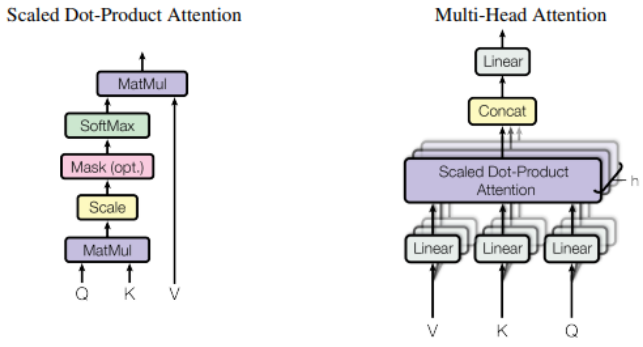

Figure 4. Attention Mechanism (Vaswani et al., 2017)

equivalent to the attention-based vector representation of a word (Vaswani et al., 2017). For a loose analogy, query is "what to attend to", which is a vector representing the position of which the attention scores should be computed, the key provides the information of the position's content so that it can know how important current position is to other positions when computing attention scores. The value is the feature representation of the actual content of the position, usually the word embedding of the word in current position. After understanding the multi-head attention is just the multiple stacks of self-attention often denoted as $MultiHead(Q, K, V) = Concatenate(attention^{head1}, \ldots., attention^{headn})$, where each attention head can have multiple self-attention.

3.0 Deep Learning for Chatbot

3.1 Natural Language Understanding and Generation

In order to develop a chatbot, the two major problems to be solved are natural language understanding (NLU) and natural language generation (NLG). The primary goal of NLU is to facilitate the machine's ability in understanding human language (Khurana et al., 2022), in the

case of conversation bot, it needs to be able to understand user query to generate appropriate responses. Next, NLG is concern with the machine capability on generating human text based on its' understanding (Khurana et al., 2022). The syntax, grammar and language structures are all that need to be taken into account in NLG. Both of this task can be split into two different processing problem, namely features representation for NLU and response generation for NLG. Hence, recently many research on chatbot have been done on transformer architecture using the encoder layer to solve the NLU and decoder layer for NLG (Chen et al., 2023) (Lee et al., 2023) (Lewis et al., 2019) (Liu et al., You Impress Me: Dialogue Generation via Mutual Persona Perception, 2020) (Liu et al., GPT understands, too, 2023).

3.2 Features Representation

In typical machine learning tasks, features representation involves converting text data into number representation encoding. However, encoding does not provide contextual information for the model to learn or understand user query, which was why word embeddings were introduced. Word embeddings are distributed representations of words in a continuous vector space typically represented in a higher dimension as real-valued vectors. There are two major types of embeddings, namely, static embeddings and dynamic embeddings. Some models that produce static embeddings includes Glove and Word2Vec (Dharma et al., 2022). Static embeddings mean that the embeddings are produced by unsupervised models such as Glove and Word2Vec before being used in downstream task. These embeddings contain contextual information that were hardcoded (outputs of the unsupervised models) (Dharma et al., 2022) and can be used for training various NLP models. With the advent of encoder-decoder models dynamic embeddings techniques became more popular. The definition of dynamic embedding mean that embeddings were produced during runtime through the encoder layer to produce more meaningful feature representation. Typically, people use pre-trained large language model to produce such embeddings as they were trained on large amount of text data. Some of the popular dynamic word embeddings models includes BERT (Devlin et al., 2018), ELMO (Ethayarajh, 2019), and GPT (Ethayarajh, 2019).

3.3 Response Generation

For chatbot there are mainly two types of response generation: retrieval-based and generation-based. Based on the task different response generation technique can be chosen. Retrieval-based methods can be considered as classification problem as it involves ranking and choosing

candidates responses from the knowledge base (Liu, S, et al., 2023). Retrieval-based responses are typically easier to model and evaluate, but the responses are dull and cannot answer out-of-domain queries. Generation-based methods would be a task of text generation. This task is concern with the similarity between words, such that the probability of a word appearing after another word and what other similar words can be used as variant for generating creative responses. There are various challenges in generation-based chatbot, contextual consistency, personality, language structuring are all problems that need to be solved to produce a practical generation-based chatbot which is suitable for open-domain conversation. Such conversation agent would require huge amount of data to train which limit its accessibility in the early years. With the advent of conversational agent like ChatGPT have raised the popularity of open-source pre-trained transformer-based model making it significantly more accessible for researchers to enhance the capability of chatbot. Pre-trained large language models such as Bart (Chen et al., 2023) , DiaglogGPT (Lin et al., 2021), and others transformer based pre-trained model provide good opportuning for building a much feasible generative-based chatbot than before.

4.0 Related Work

Table 1 shows the summarization of the reviewed papers.

4.1 Conversation Models

Pre-trained transformer-based models served as a good starting point for building a conversation model. Recent works, involves the utilization of pre-trained models by fine-tuning and transfer learning to build their conversation models and showing good results. Adapter-Bot a model uses DialGPT as backbone for dialogue generation, which is a pre-trained transformer-based model was proposed by Lin et al (2021), **Figure 5** shows the architecture of their model (Lin et al., 2021). The DialGPT was used as a backbone model and fine-tuned using residual adapter (Houlsby et al., 2019) to learn retrieval of dialogue skills from different domain such as weather, persona, movie recommendation, and emphatic response (Lin et al., 2021). They also used an external BERT model to decide the adapter to be used and passing the corresponding pre-trained weights to the DialGPT model to generate suitable responses (Lin et al., 2021). This method allowed the conversation models to continuously learn new task without having to fine-tune the backbone model and also avoid forgetting previously learned tasks. In their experiment, they used a medium size DialoGPT model consisting of 345M

parameters and a Bert base model as classifier, for training the adapters were fine-tuning on different datasets (based on task), the hyperparameter for the adapters $h = 200$ bottleneck size was used, which result in an additional of 9.85M trainable parameters (Lin et al., 2021). Additionally, they adjusted their batch size, learning rate through running multiple time and evaluating the model performance on the validation sets of each task and conclude that batch size of 16, learning rate of $6 * 10^{-4}$, and implementing early stopping technique provide the best performances for learning the adapter weights. In their experiment, they use Perplexity (PPL), BLEU, and F1 evaluation metrics on the CONVAI2 (Dinan et al., 2019) dataset, which

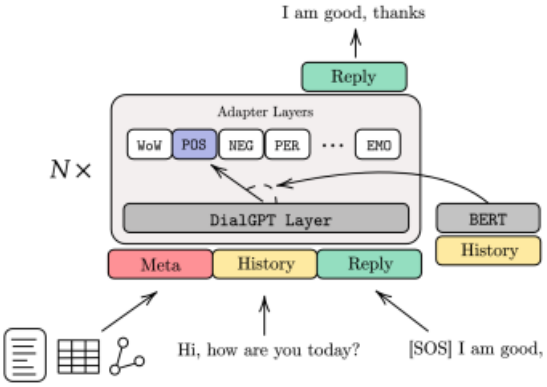

Figure 5. Adapter-Bot Architecture (Lin et al., 2021)

was designed for building persona based chatbot. They were able to obtained BLEU (1.55), PPL (11.08), and F1 (15.0) scores respectively (Lin et al., 2021). According to their benchmark, as shown in **Figure 6** their results does not show state-of-the-art performance, but satisfactory results, additionally their proposed adapter-based transfer learning method serve as a good research direction for future work on multiple tasks learning model.

Model	BLEU	Ppl.	F1
PC (Zhang et al., 2018b)	-	38.08	-
TT (Wolf et al., 2019)	-	17.51	-
BERT (Lin et al., 2020a)	1.79	16.08	-
GPT-2 (Lin et al., 2020b)	**2.17**	13.13	-
BART (Lewis et al., 2020)	-	11.9	20.7
DDD (Shuster et al., 2020b)		10.8	**21.7**
BST (Roller et al., 2020)	-	**8.36**	-
Adapter-Bot	1.55	11.08	15.0

Figure 6. Adapter-Bot Benchmark Comparison (Lin et al., 2021)

In recent work, Bart was proposed as backbone architecture for building conversation model (Chen et al., 2023). **Figure 7** shows the architecture for their proposed model. Bart is made up of a bi-directional encoder layer and an autoregressive decoder layer (Lewis et al., 2019). The original motivation of Bart was to be used as a text denoiser, but the nature of its architecture

allowing it to be effective when fine-tuned on text generation tasks as well (Chen et al., 2023) (Lewis et al., 2019). In their experiment they added custom token: [PER], [QRY], [RSP] to separate the input of the persona information, meaning the input is $x = \{Q, R, P\}$, where Q is the user query, R is the target response and P is the Persona texts (Chen et al., 2023). In order for the decoder to generate consistent responses that take the persona information and dialogue contexts they introduced two stage learning, namely, the entailment relation learning

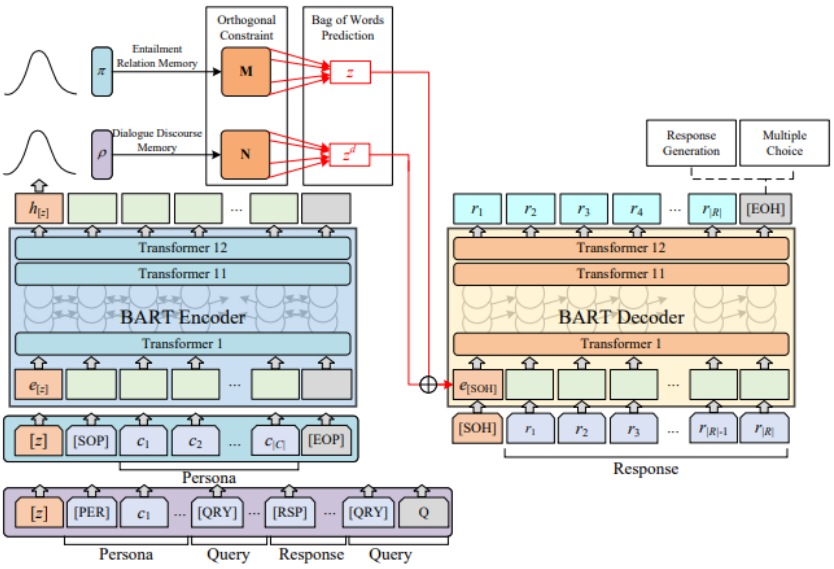

Figure 7. Persona consistent Bart Architecture (Chen et al., 2023)

and persona and dialogue consistent generation learning, where entailment relation learning involves memorizing entailments using the DNLI dataset (Welleck et al., 2019), which is an inference dataset built based on PersonaChat dataset (Zhang et al., 2018). The DNLI dataset contains sentence pairs labelled as entailment, neutral, or contradict allowing the model to learn entailment between sentence pairs (Chen et al., 2023). On the other hand, the dialogue consistent generation learning involves mapping discourse information into latent space and used together with the learnt latent features in entailment memory as latent variable (Chen et al., 2023) for the decoder as controlled generation. They use mainly two datasets the DNLI dataset (Welleck et al., 2019) for training entailment and the ConvAI2 PersonaChat dataset (Dinan et al., 2019) for training persona consistent dialogue generation.

Model	Original		
	Hits@1	PPL	F1
KV Profile Memory	54.8	-	14.25
Generative Profile Memory	10.2	35.01	16.29
LIC	17.3	-	17.79
Transfertransfo	82.1	17.51	19.09
P^2BOT	81.9	15.12	19.77
Our	**89.5**	**10.99**	**21.99**

Figure 8. Bart with latent space benchmark result (Chen et al., 2023)

In their experiment, their model yielded a state-of-the-art result having hits@1 (89.5), perplexity (10.99), and f1 score (21.99) on the ConvAI2 test set as shown in **Figure 8** (Chen et al., 2023). The hits@1 was used to evaluate the model capability in choosing the positive response among the 20 candidate responses which 19 of them is negative responses, as candidates were passed for the model to filter negative responses and learn similar response with the positive response as multiple-choice task for helping the model to generalize better (Chen et al., 2023). The perplexity was to compute the exponential of negative log-likelihood between the generated response and the truth response and the F1 is to compute the precision and recall on word level between the generated and truth response (Chen et al., 2023). Their experiment uses a learning rate of 8e-6 for two stages learning and different batch size for different training stages. Here they used batch size of 64 for entailment learning stage, and batch size of 2 with gradient accumulation of 8 for dialogue generation training stage (Chen et al., 2023).

A novel approach using multiple pre-trained BERT models to build a conversation agent was proposed (Song et al., 2021). **Figure 9** shows the architecture of their proposed model. Their proposed framework is a very large model consisting of three Bert layer, one as encoder and two as decoder. They disentangled the dialogue generation into two tasks, which are response generation and entailment understanding (Song et al., 2021). Which is why they have two BERT layer in their decoder. Their concept was similar to Chen et al., (2023), which

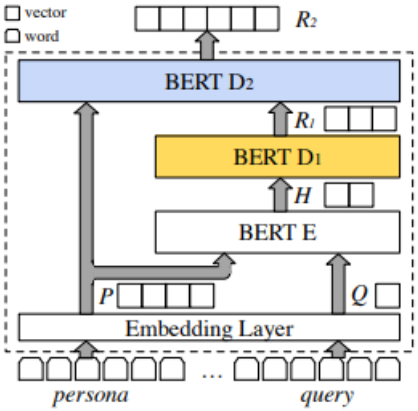

Figure 9. Bert over Bert Architecture (Song et al., 2021)

they also use non-dialogue inference data to train their model's entailment understanding, but the differences lie with they use a separate model as another layer to do so. In their implementation, they employed Adam optimizer with learning rate α to 5e-3 and β to 0.1 (Song et al., 2021). Since their model is large a 32GB GPU would be required to train their model. Similarly, they evaluated their model with the ConvAI2 dataset and uses multiple inference dataset MNLI (Williams et al., 2017), DNLI (Welleck et al., 2019), and some Chinese dataset to learn the entailment. They used evaluation methods of PPL, distinct ½ which is the ratio of distinct unigrams / bi-grams where higher distinct mean better response diversity and some other human evaluation methods such as fluency (Flue), informativeness (Info) and such (Song et al., 2021). Their model was able to achieved a 7.8 (PPL), 8.64 (Distinct), 4.12 (Flue), and 4.03 (Info). **Figure 10** shows the full benchmark result for their model.

	PPL	Dist.1	Dist.2	D.AVG	p.Ent	p.Ctd	ΔP	C.Score	Flue.	Info.	Relv.	Per.C.
Transformer	28.8	3.14	17.80	10.47	31.5	35.5	4.0	1.20	3.05	2.57	2.72	0.05
CMAML	36.7	1.00	2.10	1.55	32.3	37.5	5.2	6.96	3.36	2.40	3.09	0.24
GDR	16.7	3.76	23.10	13.43	19.7	32.3	12.6	7.89	3.38	2.74	3.13	0.21
LIC	17.3	6.29	28.99	17.64	13.7	20.4	6.7	14.12	3.70	3.53	3.47	0.39
GPT2	14.4	7.29	28.12	17.71	12.0	20.2	8.2	15.88	3.79	3.22	3.79	0.47
BoB (Ours)	**7.8**	**8.40**	**36.08**	**22.24**	**7.3**	**83.4**	**76.1**	**17.18**	**4.12**	**4.03**	**4.09**	**0.60**

Figure 10. BOB benchmark result (Song et al., 2021)

p^2 bot was proposed by Liu et al., (2020) as a chatbot that can remember the persona of theirs correspond interlocutor (Liu et al., 2020). **Figure 11** shows the general architecture of p^2 bot. Their model is made up of two main components, namely, the transmitter and the receiver. The transmitter is responsible for generating dialogue taking into account their correspond interlocutor personality H^p which will be learned as conversation goes on (Liu et al., 2020). In order to determine if the impression is correct the receiver component come into

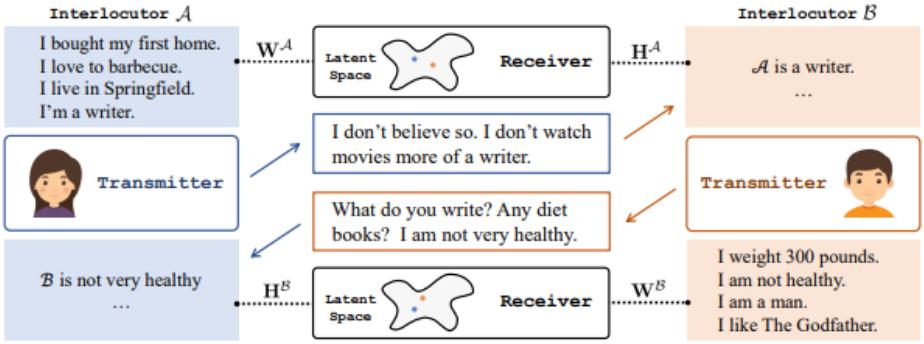

Figure 11. Architecture of p^2 bot (Liu et al., 2020)

play. The receiver is responsible for measuring the similarity between impression from the transmitter with the actual persona. The transmitter architecture follows the GPT architecture which consist of only decoder (Liu et al., 2020) for sequence generation. A special [PS] token

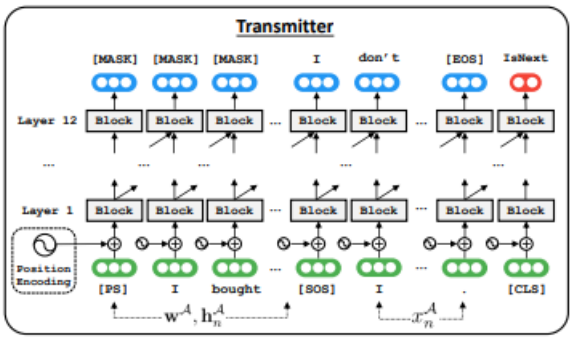

Figure 12. Transmitter architecture (Liu et al., 2020)

was added to differentiate between the query input and persona input. The transmitter was also trained to discriminate between negative responses and positive response by adding a classifier (Liu et al., 2020). **Figure 12** shows the architecture of the transmitter which uses the transformer architecture. For the receiver component, it was trained by negative sampling incorrect persona, where it needs to identify the real persona within the sample (Liu et al., 2020). **Figure 13** shows the architecture of the receiver, which contains two encoders, one for impression and another for persona (Liu et al., 2020). Reinforcement learning was introduced

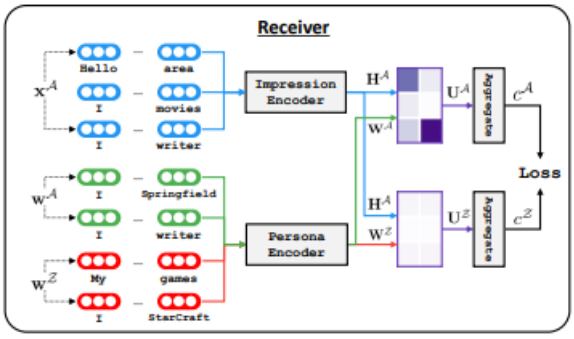

Figure 13. Receiver architecture (Liu et al., 2020)

to improve the p^2 bot where two transmitters were trained by communicating with each other where one will be frozen and another will be the learnable agent for N turn (Liu et al., 2020). Three components design are important in reinforcement learning, namely, the state, action, and policy. Considering B as the learnable agent, conversing with the frozen party A. Here the state is the persona and the dialogue history defined as S_n^B and action is the response to be generated defined as a_n^B and the goal is to learn the policy parameter θ so that the it can guide the agent to select action the generated response which maximize the rewards (Liu et al., 2020). **Figure 14** shows the reinforcement learning procedure.

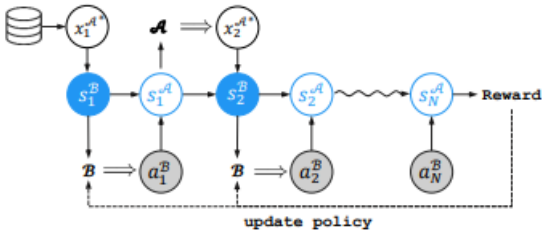

Figure 14. Reinforcement Learning for p^2 bot (Liu et al., 2020)

The proposed p^2 bot uses Hits@1 (81.9), PPL (15.12), and F1 (19.77) as evaluation metrics **Figure 15** shows their overall benchmarking results. The dataset used is the CONVAI2 dataset (Dinan et al., 2019). They used Adam as their optimizer and set the learning rate to 6.25e-5 for both their receiver and transmitter. Whereas, during reinforcement learning the transmitter learning rate was set to 1e-6. The hyperparameter $m, \alpha, \beta, \lambda_1, \lambda_2, \lambda_3$ were set to 0.4, 0.1, 1e-4, 0.5, 0.4, 0.1, 0.5 respectively (Liu et al., 2020). They run their regular training for 2 epochs and reinforcement learning for 3 turns. They used beam search strategy during inference which size is set to 2 (Liu et al., 2020).

Category	Model	Original		
		Hits@1(%) ↑	ppl ↓	F1(%) ↑
Retrieval	KV Profile Memory	54.8	-	14.25
	Dually Interactive Matching	78.8	-	-
Generative	Generative Profile Memory	10.2	35.01	16.29
	Language Model	-	50.67	16.30
	SEQ2SEQ-ATTN	12.5	35.07	16.82
Pretrain Fintune	Lost In Conversation	17.3	-	17.79
	Transfertransfo	**82.1**	17.51	19.09
	$\mathcal{P}^2$ BOT (Our)	81.9 [0.1]	**15.12** [0.16]	**19.77** [0.08

Figure 15. p^2 bot benchmark results

4.2 Persona-based Dialogues

Residual adapter (Houlsby et al., 2019) was proposed in an Adapter-Bot model (Lin et al., 2021) for learning persona-based dialogue generation, the persona information was fed separately using special token embedding as meta information to the DialoGPT for training

(Lin et al., 2021). Chen et al., (2023) proposed using entailment latent space for controlling decoder generation to be consistent with persona (Chen et al., 2023). The entailment was learnt using inference dataset providing the model the concept of hypothesis and premise which effectively allowing it to memorize and utilize persona information to search for generated responses (premise) that entail with the persona (hypothesis). Song et al., (2021) proposed disentangling the generation process and the persona entailment process into separate task resulting in two separate model for learning the tasks (Song et al., 2021). In their proposed method, multiple inference datasets were used to make the first layer of their model's decoder layer (consisting a pre-trained Bert) to learn entailment so during decoding the model can generate hypothesis to test if the generated sentences are consistent with the persona.

4.3 Context-Consistent Dialogues

Context consistent or discourse coherent are important aspect in a conversation to ensure the fluency and logical response generation. The memorization of discourse information in the latent space have been proposed recently (Chen et al., 2023) to enhance model response generation, the conversations were memorized into latent space as features and used in the generation of texts. A model proposed consisting of multiple BERT decoder layer by Song et al., (2021) uses different layer for helping to generate context-consistent sentences. The last Bert layer in **Figure 9** receive input from the embedding layer which consist of embedded persona features and output from the decoder layer containing entailment and conversation history information. The inputs received were fuses to generate discourse coherent responses (Song et al., 2021).

4.4 Ranking Generation

In order to develop a better controlled generation model, a discriminative head can be added to the model to assist the decoder layer to generalize better according to the task. Proposed by Chen et al., (2023) uses the built-in pre-trained Bart sentence classification head to make the model to learn extra weight (Chen et al., 2023). The idea is that the model will be fed independently from the text generation decoder with 1 positive response along with multiple wrong responses and the model need to classify the positive response as the top candidate (Chen et al., 2023). Specifically, the output from the text generation decoder (last hidden state) will be fed into the classifier to predict score $\sim y$ for the candidates (Chen et al., 2023). Liu et

al., (2020) proposed training a discriminator by introducing response distractor to improve the text generation (Liu et al., 2020). Specifically, a special [CLS] token was appended to the tail of their generated token where a discriminator was built on top of the hidden state of the token and the hidden state fed into the discriminator should rank the correct response as the top candidates amongst the distractors (Liu et al., 2020).

4.5 Related Work's Model Summarization

Table 1 Benchmark Models and Performances

Paper	Model	ConvAI2 Original Benchmark		
		Hits@1	PPL	F1
Learning to Memorize Entailment and Discourse Relations for Persona Consistent Dialogues (Chen et al., 2023)	**Bart: Transformer**	**89.5**	10.99	**21.99**
TransferTransfo: A Transfer Learning Approach for Neural Network Based Conversational Agents (Wolf et al., 2019)	**Transformer**	82.1	17.51	19.09
You Impress Me: Dialogue Generation via Mutual Persona Perception (Liu et al., 2020)	**GPT: Transformer**	81.9	15.21	19.77
The Adapter-Bot: All-In-One Controllable Conversational Model (Lin et al., 2021)	**DialGPT: Transformer**	-	11.08	15.0
BoB: BERT Over BERT for Training Persona-based Dialogue Models from Limited Personalized Data (Song et al., 2021)	**Bert: Transformer**	-	**7.8**	-

5.0 Methodology

5.1 Model Architecture

For the experiment in this paper the implementation for building the conversation model will follow the transformer architecture as recent works have shown that such architecture gives better result on conversation model as oppose to architecture such as RNN and LSTM.

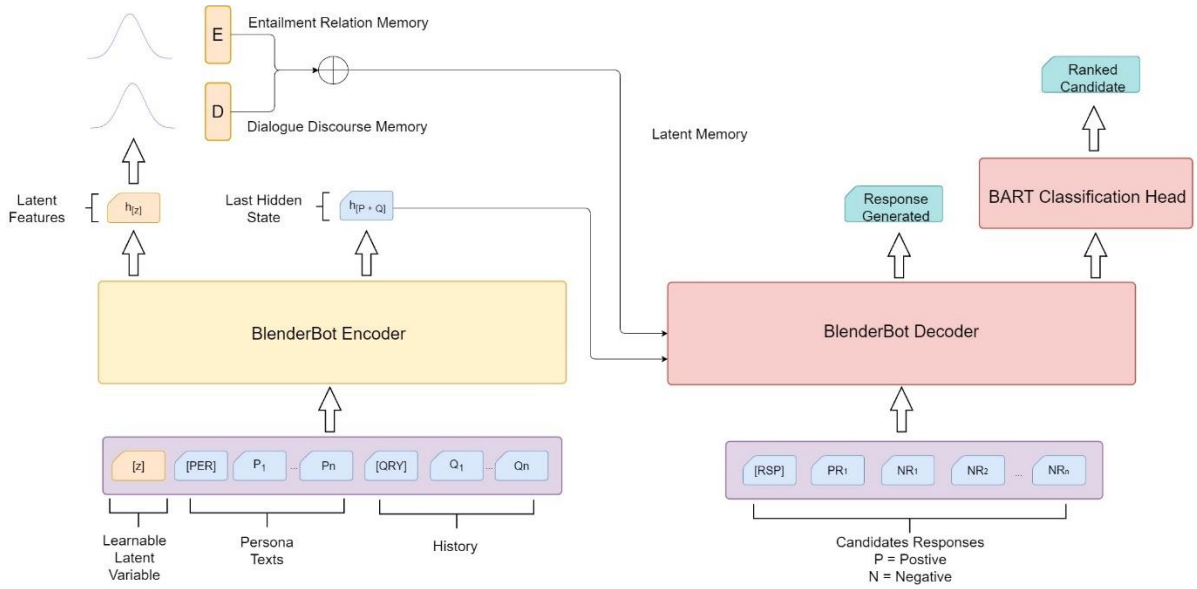

Figure 16. Blenderbot Space Architecture

Figure 16 shows the overall architecture of Blenderbot Space. Blenderbot Space uses pre-trained BlenderBot model (Roller et al., 2020) as backbone for building persona-based chatbot. Inspired by recent works (Chen et al., 2023) (Lee et al., 2023) (Song et al., 2021) (Lin et al., 2021) (Liu et al., You Impress Me: Dialogue Generation via Mutual Persona Perception, 2020) have shown that fine-tuning large pre-trained models have proved to provide good results. In this experiment, BlenderBot was chosen as the backbone model for fine-tuning to downstream conversation agent is because:

1. Blenderbot is a model built on top of transformer architecture which such architecture exhibits excellent performance on conversational tasks
2. Most research works reviewed in this paper uses encoder-decoder-based architecture, so using Blenderbot allow me to transfer useful algorithms introduced in their paper more easily
3. Blenderbot was originally pre-trained for conversation purposes, making it very much suitable for this experiment objectives

4. Most pre-trained models have huge amounts of parameters (>1billion), whereas Blenderbot provide variety of size, which allowed this experiment to run on the more lightweight version due to computational constraints

Following the work of Chen et al., (2023), where they introduced the latent space concept for improving pre-trained Bart text generation consistency on persona and dialogue context and was able to produce a state-of-the-art chatbot model on the ConvAI2 dataset (Dinan et al., 2019). The aim of this experiment is to utilize their entailment and discourse memory algorithms on a different architecture to observe how well the model would perform. Other than that, unlike Bart the original BlenderBot architecture does not contains a classification head layer for sequence classification, considering work by Liu et al., (2020) and Chen et al., (2023) both state-of-the-art model uses sequence classification to help control the text generation, I introduced a classification head (same model as Bart classification head) to my model. The proposed model of this paper will run two stages of training, the entailment learning and the dialogue generation learning. The entailment learning will be discussed in section **5.2 Entailment Learning**. For the second stage training dialogue generation, the training flow follows **Figure 16** where the inputs to the encoder are process as $M(P, Q, R)$ where $P = [p_1, p_2, \dots p_n]$ is the set of persona texts, $Q = [q_1, q_2, \dots q_n]$ is the queries, and $R = [rp, rn_2, \dots rn_n]$ is the set of candidate response where rp represent the one positive response. Assuming that stage 1 training (entailment learning) have be done, the input $x = \{P, Q\}$ will be fed into the Blenderbot encoder layer which will encode the $\{Q\}$ into features representation as last hidden state and the last hidden state will also be used to mapped into the latent space in the Dialog Discourse Memory. Independently, the Blenderbot encoder will also encode the $\{P\}$ into the latent space in the Entail Relation Memory using the weights learnt from entailment learning. Then the entail relation variable and dialog discourse variable will be summed as a latent variable and fed into the decoder layer along with the last hidden state.

In the decoder layer, three inputs are provided. The latent variable, the last hidden state of the encoder and the candidate responses. The decoder will generate response autoregressively control by the latent variable and the last hidden state. Once the decoder finish generating its last hidden state will be used on the Bart sequence classification head to predict the $\sim y$ score for each candidate responses. Overall, there are 4 types of losses summed together to compute the gradient. The classification loss (Chen et al., 2023) (Liu et al., 2020) produced from the classification head, the language model loss from the text generation decoder, the

memory loss for the latent variable (Chen et al., 2023) and the bag of word loss (Chen et al., 2023).

5.2 Entailment Learning

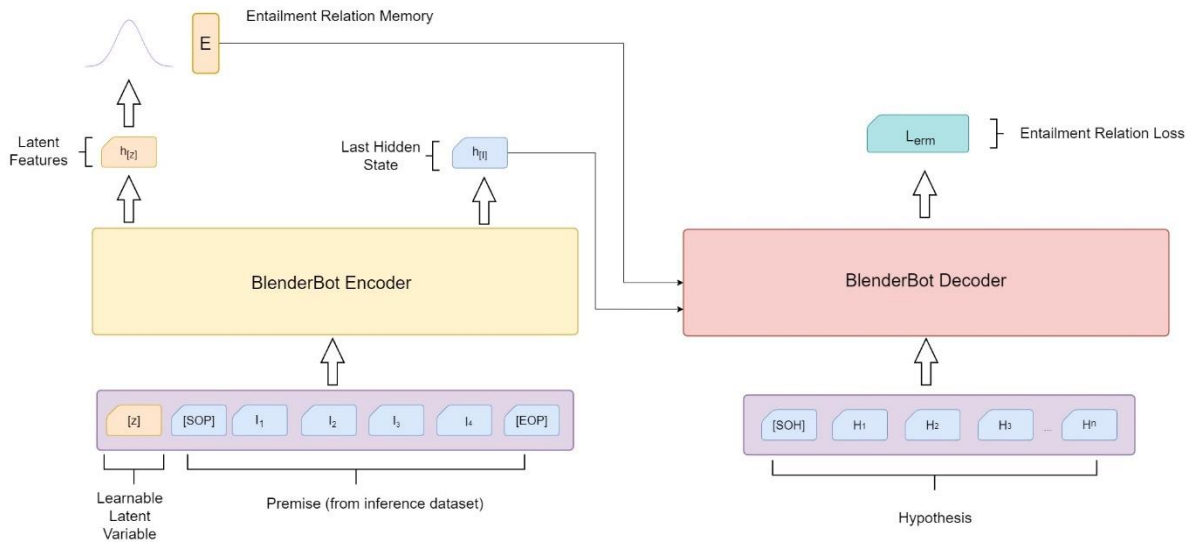

Figure 17. Entailment Learning

The first stage of training is to learn entailment relation using inference dataset as shown in **Figure 17**. Inference dataset would contain pairs of sentences annotated with labels such as entailment, contradiction, and neutral (Williams et al., 2017) as hypothesis where the model would need to learn. The encoder will receive the premises which is the sentence pairs and encode it into feature representation as last hidden state. The last hidden state was also mapped into the entailment latent memory using the learnable entailment weight, which will be tuned throughout the entailment learning process. After that, the encoder last hidden state and the entailment relation memory will be passed into the decoder to compute the loss for backpropagation (Chen et al., 2023).

5.3 Evaluation Methods

For this experiment my model with be benchmarking with various models such as BOB (Song et al., 2021), Learning to Memorize entail and discourse relations (Chen et al., 2023), Adapter-bot (Lin et al., 2021), p^2 bot (Liu et al, 2020), Transfertransfo (Wolf et al., 2019), and GPT-2 (Lin et al., 2021). Since, most of them used different metrics, in order to benchmark properly in this experiment, I will select three metrics that most of the papers used and does not require

human evaluation. The three metrics selected are Hits@1 for evaluating the model capability on ranking positive candidate correctly, Perplexity to compare between response and generation with negative log-likelihood, and F1 score to compute the precision and recall at word level between generated text and response. These metrics covered most of the important aspect that need to be evaluated for a chatbot in the context of automatic evaluation.

5.4 Dataset Selection

The main dataset that will be used is the CONVAI2 dataset (Dinan et al., 2019) which is a popular for developing a persona-based chatbot. The dataset was built with the motivation of organizing an open-domain chatbots competition (Dinan et al., 2019). The dataset was created through collecting conversations between crowd-workers paired which they were required to act as the persona randomly given to them and converse accordingly (Dinan et al., 2019). The dataset format is similar to what is shown in **Figure 18** having a two person with their respective persona and their dialogues. The details of the training, validation and test set is shown in **Figure 19**. The dataset contains 1155 personas with each having at least 5 profile sentences for training, whereas validation and test set have 100 personas each (Dinan et al., 2019).

Persona 1	**Persona 2**
I like to ski	I am an artist
My wife does not like me anymore	I have four children
I have went to Mexico 4 times this year	I recently got a cat
I hate Mexican food	I enjoy walking for exercise
I like to eat cheetos	I love watching Game of Thrones

[PERSON 1:] Hi
[PERSON 2:] Hello ! How are you today ?
[PERSON 1:] I am good thank you , how are you.
[PERSON 2:] Great, thanks ! My children and I were just about to watch Game of Thrones.
[PERSON 1:] Nice ! How old are your children?
[PERSON 2:] I have four that range in age from 10 to 21. You?
[PERSON 1:] I do not have children at the moment.
[PERSON 2:] That just means you get to keep all the popcorn for yourself.
[PERSON 1:] And Cheetos at the moment!
[PERSON 2:] Good choice. Do you watch Game of Thrones?
[PERSON 1:] No, I do not have much time for TV.
[PERSON 2:] I usually spend my time painting: but, I love the show.

Figure 18. CONVAI2 dataset sample

	# examples	# dialogues	# personas
Training set	131,438	17,878	1,155
Validation set	7,801	1,000	100
Hidden test set	6,634	1,015	100

Figure 19. CONVAI2 training, validation, and test set

Another dataset will need to be used for inference training (entailment learning), there were several inference datasets available as shown in Chen et al., (2023), and Song et al., (2021). For this experiment I chose to use the DNLI dataset (Welleck, Weston, Szlam, & Cho, 2019).

This dataset consists of sentence pairs labelled with either entailment (E), contradiction (C), or neutral (N) allowing the model to learn the entailment relation dataset (Welleck, Weston, Szlam, & Cho, 2019). **Figure 20** shows the detail of the dataset, for this experiment I only need to train the entailment relation where the evaluation will be done together with the dialogue generation, hence, only the training set containing 310,110 records will be used.

		Train		Valid		Test		Test-Gold	
Data Type	**Label**	(u,p)	(p,p)	(u,p)	(p,p)	(u,p)	(p,p)	(u,p)	(p,p)
Matching Triple	E	43,000	57,000	5,000	500	4,500	900	3,712	615
Misc. Utterance	N	50,000	-	3,350	-	3,000	-	2,282	-
Persona Pairing	N	20,000	10,000	2,000	-	2,000	-	1,466	-
Relation Swap	N	20,000	-	150	-	400	-	260	-
Relation Swap	C	19,116	2,600	85	14	422	50	279	44
Entity Swap	C	47,194	31,200	4,069	832	3,400	828	2,246	591
Numerics	C	10,000	-	500	-	1,000	-	881	-
Dialogue NLI Overall		**310,110**		**16,500**		**16,500**		**12,376**	

Figure 20. DNLI training, validation, and test set

6.0 Model Implementation

6.1 Data pre-processing

Since there are two datasets involve in this experiment, two data pre-processing process were conducted for each dataset. For the first dataset - CONVAI2 (Dinan et al., 2019) was in text file format, which the sample is shown in **Figure 21**, each set of conversations start with at least 5 persona texts, then every first sentence is the query following a huge blank space to indicate the candidate responses for that query. For instance, "hi, how are you doing? I'm getting ready to do some cheetah chasing to stay in shape" is the query and the first sentence after it is the positive candidate following several other negative candidates. The query above the current query can be considered as the chat history. To split the data into query, persona,

```
1 your persona: i like to remodel homes.
2 your persona: i like to go hunting.
3 your persona: i like to shoot a bow.
4 your persona: my favorite holiday is halloween.
5 hi , how are you doing ? i'm getting ready to do some cheetah chasing to stay in shape .  you must be very fast . hunting is one of my favorite h
6 i am ! for my hobby i like to do canning or some whittling .  i also remodel homes when i am not out bow hunting .        hello i am doing well h
7 that's neat . when i was in high school i placed 6th in 100m dash !   that's awesome . do you have a favorite season or time of year ?       yes
8 i do not . but i do have a favorite meat since that is all i eat exclusively .    what is your favorite meat to eat ?     not really , it is just
9 i would have to say its prime rib . do you have any favorite foods ?  i like chicken or macaroni and cheese .     i am listening to system of a d
10 do you have anything planned for today ? i think i am going to do some canning . i am going to watch football . what are you canning ?       tha
11 i think i will can some jam . do you also play footfall for fun ?    if i have time outside of hunting and remodeling homes . which is not much
```

Figure 21. CONVAI2 dataset format sample

history pairs, a function shown in **Figure 22** was created. The function accepts the file path toward the dataset and split the item into persona, query, and response pairs accordingly. For each conversation the "your persona:" keyword will be check, if exist meaning it is a new

conversation and the persona will be grouped together. After that if "your persona:" keyword was not found it indicate that it is currently in the first query and "\t" was used to split to get the query, response, and negative response respectively. After running through the whole text file the separated query, history, response, and candidate (negative responses) will be returned.

```
def create_persona_data(file_path):
  with open(file_path, "r", encoding="utf8") as f:
      persona =[]
      query = []
      response = []
      cand = []
      is_persona = False
      tmp_persona = []
      tmp_query = []
      tmp_response = []
      tmp_cand = []
      first = True
      cnt = 0
      sum_u = 0
      for line in f:
          cnt += 1
          line = line.strip()
          if "your persona: " in line:
              if not is_persona and not first:
                  query.append(tmp_query)
                  response.append(tmp_response)
                  cand.append(tmp_cand)
                  sum_u += len(tmp_query)
                  tmp_query = []
                  tmp_response = []
                  tmp_cand = []
              first = False
              is_persona = True
              line = line.split(": ", maxsplit=1)[1]
              tmp_persona.append(line)
          else:
              if is_persona:
                  persona.append(tmp_persona)
                  is_persona = False
                  tmp_persona = []
              line = line[line.find(" ")+1:]
              tmp_query.append(line.split("\t")[0])
              tmp_response.append(line.split("\t")[1])
              tmp_cand.append(line.split("\t")[3].split("|"))
      query.append(tmp_query)
      response.append(tmp_response)
      cand.append(tmp_cand)
      sum_u += len(tmp_query)
      assert len(query) == len(response) == len(persona) == len(cand)

  return persona, query, response, cand
```

Figure 22. Pre-processing for CONVAI2 dataset text file

After getting the respectively persona, query, history, and candidate pairs I need to tokenize and encode them into numeric format so that the model can accept the input. This is where several functions are needed. First, I need to retrieve the encoding ids for the special token, so that it can be used to separate the inputs. **Figure 23** shows the function code for retrieving special token ids.

```
def get_token_id(tokenizer):
  bos_id = tokenizer.bos_token_id
  eos_id = tokenizer.eos_token_id
  pad_id = tokenizer.pad_token_id
  sep_id = tokenizer.sep_token_id
  query_id, res_id, latent_id, persona_id = tokenizer.convert_tokens_to_ids(
      ['<query>', '<response>', '<latent>', '<persona>'])
  return bos_id, eos_id, pad_id, sep_id, query_id, res_id, latent_id, persona_id
```

Figure 23. Special Token Retrieval Function

After that, the main function for converting the persona, query, response, and candidate was created as shown in **Figure 24**. This function takes the tokenizer (passed into it as parameter, it is a pre-trained blenderbot tokenizer) to tokenize and convert the text into numeric encodings for each of the persona, query, response, and candidate pairs. After that, the special token ids such as '<persona>' are added in front of the persona ids as an indicator to the model. This was done similarly for query, and response. As the query, persona are inputs for the encoder and responses are for the decoder, two sub-functions were used to handle their processing described previously as shown in **Figure 25** for handling encoder input pre-processing and **Figure 26** for handling decoder input pre-processing. This way the code dependencies are decoupled promoting better readability.

```
def load_data_persona(persona, query, response, cand, tokenizer, max_history=4, n_cand=5, use_all=False, max_seq=128):
  bos_id, eos_id, pad_id, sep_id, query_id, res_id, latent_id, persona_id = get_token_id(tokenizer)
  persona_data = defaultdict(list)
  for i in range(len(persona)):
      persona_ = persona[i]
      per_list = []
      for per in persona_:
          persona_ids = tokenizer.convert_tokens_to_ids(tokenizer.tokenize(per, add_prefix_space=True))
          per_list.append(persona_ids)
      query_ = query[i]
      response_ = response[i]
      cand_ = cand[i]
      history = []
      assert len(query_) == len(response_)
      for j in range(len(query_)):
          if use_all:
              noise_candidate = cand_[j][:-1]
          else:
              noise_candidate = random.sample(cand_[j][:-1], n_cand-1)

          query_ids = tokenizer.convert_tokens_to_ids(tokenizer.tokenize(query_[j], add_prefix_space=True))
          response_ids = tokenizer.convert_tokens_to_ids(tokenizer.tokenize(response_[j], add_prefix_space=True))

          noise_cand_ids_list = [tokenizer.convert_tokens_to_ids(tokenizer.tokenize(text, add_prefix_space=True))
                         for text in noise_candidate]
          history.append(query_ids)
          history.append(response_ids)
          tmp_history = history[-2 * max_history: -1]

          encoder_input_items = create_encoder_input(per_list, tmp_history, query_id, res_id, latent_id, persona_id, sep_id, eos_id, max_seq=max_seq)
          if encoder_input_items is None:
            continue
          else:
            encoder_input_ids, attention_mask, per_input_ids, per_attention_mask = encoder_input_items

          decoder_input_items = create_decoder_input(response_ids, res_id, eos_id, max_seq=max_seq, golden=True)
          if decoder_input_items is None:
            continue
          else:
            decoder_lmlabel, decoder_input_ids, decoder_cls_idx, decoder_attention_mask = decoder_input_items
```

```
        persona_data["input_ids"].append(encoder_input_ids)
        persona_data["attention_mask"].append(attention_mask)
        persona_data["per_input_ids"].append(per_input_ids)
        persona_data["per_attention_mask"].append(per_attention_mask)
        persona_data["lmlabels"].append(decoder_lmlabel)
        persona_data["decoder_input_ids"].append(decoder_input_ids)
        persona_data["decoder_attention_mask"].append(decoder_attention_mask)
        persona_data["cls_index"].append(decoder_cls_idx)
        persona_data["clslabel"].append([0])
        for k in range(len(noise_cand_ids_list)):
            decoder_lmlabel, decoder_input_ids, decoder_cls_idx, decoder_attention_mask = create_decoder_input(noise_cand_ids_list[k], res_id, eos_id, golden=False)
            persona_data["input_ids"].append(encoder_input_ids)
            persona_data["attention_mask"].append(attention_mask)
            persona_data["per_input_ids"].append(per_input_ids)
            persona_data["per_attention_mask"].append(per_attention_mask)
            persona_data["lmlabels"].append(decoder_lmlabel)
            persona_data["decoder_input_ids"].append(decoder_input_ids)
            persona_data["decoder_attention_mask"].append(decoder_attention_mask)
            persona_data["cls_index"].append(decoder_cls_idx)


for item_name, item in persona_data.items():
    if item_name == "input_ids" or item_name == "per_input_ids":
        item = pad_sequence([torch.from_numpy(np.array(x)) for x in item],
                                        batch_first=True, padding_value=pad_id)

        persona_data[item_name] = item
    elif item_name == "lmlabels":
        item = pad_sequence([torch.from_numpy(np.array(x)) for x in item],
                            batch_first=True, padding_value=-100)
        persona_data[item_name] = item
    elif item_name == "attention_mask" or item_name == "decoder_attention_mask" or item_name == "per_attention_mask":
        item = pad_sequence([torch.from_numpy(np.array(x)) for x in item],
                            batch_first=True, padding_value=0)
        persona_data[item_name] = item
    elif item_name == "decoder_input_ids":
        item = pad_sequence([torch.from_numpy(np.array(x)) for x in item],
                            batch_first=True, padding_value=pad_id)
        persona_data[item_name] = item
    elif item_name == "clslabel":
        persona_data[item_name] = torch.tensor(item).view(-1,1)
    elif item_name == "cls_index":
        item = pad_sequence([torch.from_numpy(np.array(x)) for x in item],
                            batch_first=True, padding_value=-100) # automatically determine the max length, using the longest sequence among the input
        persona_data[item_name] = item
return persona_data
```

Figure 24. Main processing function for encoding persona dataset

```
def create_encoder_input(per, history, query_id, res_id, latent_id, persona_id, sep_id, eos_id, max_seq):
    encoder_input_ids = []
    per_input_ids = [latent_id] + [persona_id]

    for x in per:
        per_input_ids += x + [sep_id]

    encoder_input_ids += per_input_ids
    for i in range(len(history)):
        if i % 2 == 0:
            encoder_input_ids += [query_id] + history[i] + [eos_id]
        else:
            encoder_input_ids += [res_id] + history[i] + [eos_id]

    attention_mask = [1] * len(encoder_input_ids)
    per_attention_mask = [1] * len(per_input_ids)

    # Remove sample longer than max input
    if len(encoder_input_ids) > max_seq:
      return None

    return encoder_input_ids, attention_mask, per_input_ids, per_attention_mask
```

Figure 25. Function for processing input for the encoder layer of Blenderbot

```
def create_decoder_input(response_ids, res_id, eos_id, max_seq, golden=None):
  assert golden != None

  decoder_lmlabel= response_ids + [eos_id]
  decoder_input_ids = [res_id] + response_ids # <response> = res_id
  decoder_cls_index = [-100] * (len(decoder_lmlabel) - 1) + [eos_id] # cls_indices
  decoder_attention_mask = [1] * len(decoder_input_ids)

  if golden == False:
      decoder_lmlabel = [-100] * len(decoder_lmlabel)

  assert len(decoder_lmlabel) == len(decoder_input_ids)

  # Remove sample longer than max allowed input
  if len(decoder_input_ids) > max_seq:
    return None

  return decoder_lmlabel, decoder_input_ids, decoder_cls_index, decoder_attention_mask
```

Figure 26. Function for processing input for decoder layer of Blenderbot

Lastly, the processed data need to be re-arranged and their dimensions need to be amended to fit the model input dimension (Blenderbot), so a function was created to do the final processing, the function is shown in **Figure 27**.

```
def prepare_data_persona(persona_train, persona_val, args):
  KEYS = ["input_ids", "attention_mask", "lmlabels", "decoder_input_ids", "decoder_attention_mask", "cls_index", "clslabel", "per_input_ids", "per_attention_mask"]
  trainset = []
  valset = []
  for key in KEYS:
    if key == "clslabel":
      train_tensor = persona_train[key].view(-1)
      trainset.append(train_tensor)
      val_tensor = persona_val[key].view(-1)
      valset.append(val_tensor)
    else:
      train_tensor = persona_train[key].view(-1, args.cand, persona_train[key].size(-1))
      trainset.append(train_tensor)
      val_tensor = persona_val[key].view(-1, 20, persona_val[key].size(-1))
      valset.append(val_tensor)
  return trainset, valset
```

Figure 27. Function for preparing processed persona data

Moving on, for the infer data processing, the process is simpler which involve two processing function and reusing the function to get the ids for special token in **Figure 23**. The first function for processing the inference dataset, which is also the main function for processing inference dataset is shown in **Figure 28**. Since the inference dataset is only used to learn entailment relation, any sequence pairs that is not positive will be skipped and if label is positive the sequence pair will be split into premise and hypothesis as independent and dependent variable. These items will then be processed similarly to the persona dataset, which being converted into numeric encoding by the pre-trained Blenderbot tokenizer making them in format compatible with the model inputs' format. **Figure 29** is the function for re-arranging the order of the processed inference data and changing their dimension to fit the model's input. With these, concludes the basic pre-processing steps required for this experiment.

```
def load_data_infer(file_path, tokenizer):
  bos_id, eos_id, pad_id, sep_id, query_id, res_id, latent_id, persona_id = get_token_id(tokenizer)
  positive_set = defaultdict(list)

  with open(file_path, "r") as f:
      row_data = json.load(f)
      for obj in tqdm(row_data, desc='Generate infer data'):
          if obj['label'] == "neutral" or obj['label'] == "negative":
              continue
          pre = obj["sentence1"].lower()
          hyp = obj["sentence2"].lower()
          pre_ids = tokenizer.convert_tokens_to_ids(tokenizer.tokenize(pre, add_prefix_space=True))
          hyp_ids = tokenizer.convert_tokens_to_ids(tokenizer.tokenize(hyp, add_prefix_space=True))
          encoder_input_ids = [latent_id] + [persona_id] + pre_ids + [eos_id]
          attention_mask = [1] * len(encoder_input_ids)

          decoder_input_ids = [res_id] + hyp_ids
          decoder_attention_mask = [1] * len(decoder_input_ids)

          if obj['label'] == 'positive':
              positive_set["encoder_input_ids"].append(encoder_input_ids)
              positive_set["decoder_input_ids"].append(decoder_input_ids)
              positive_set["attention_mask"].append(attention_mask)
              positive_set["decoder_attention_mask"].append(decoder_attention_mask)
              decoder_lmlabel = hyp_ids + [eos_id]
              positive_set["lmlabels"].append(decoder_lmlabel)

      for item_name, item in positive_set.items():
          if item_name == "encoder_input_ids":
              item = pad_sequence([torch.from_numpy(np.array(x)) for x in item],
                                  batch_first=True, padding_value=pad_id)

              positive_set[item_name] = item
          elif item_name == "lmlabels":
              item = pad_sequence([torch.from_numpy(np.array(x)) for x in item],
                                  batch_first=True, padding_value=-100)
              positive_set[item_name] = item
          elif item_name == "attention_mask" or item_name == "decoder_attention_mask":
              item = pad_sequence([torch.from_numpy(np.array(x)) for x in item],
                                  batch_first=True, padding_value=0)
              positive_set[item_name] = item
          elif item_name == "decoder_input_ids":
              item = pad_sequence([torch.from_numpy(np.array(x)) for x in item],
                                  batch_first=True, padding_value=pad_id)
              positive_set[item_name] = item
  return positive_set
```

Figure 28. Main processing function for encoding inference dataset

```
def prepare_data_infer(infer_data):
  KEYS = ["encoder_input_ids", "decoder_input_ids", "attention_mask", "decoder_attention_mask", "lmlabels"]
  inferset = []
  for key in KEYS:
    infer_tensor = infer_data[key].view(-1, 1, infer_data[key].size(-1))
    logger.info("{}: {}".format(key, infer_tensor.size()))
    inferset.append(infer_tensor)
  return inferset
```

Figure 29. Function for preparing processed inference data

6.2 Data Loader

In order to divide the data into batches for training and validation, the data will need to be loaded into a data loader, but before that these data need to be converted into tensor because pytorch uses tensors data structure to help keep track of the gradients during the training processes. In order to do so, the data will first be converted into TensorDataset, here I created two dataset class extending the TensorDataset superclass. The PersonaData in **Figure 30** and

the InferDataset in **Figure 31**. After creating the TensorDataset they will be loaded into the DataLoader to split them into batches.

```
class PersonaDataset(TensorDataset):
  def __init__(
      self,
      input_ids,
      attention_mask,
      lmlabels,
      decoder_input_ids,
      decoder_attention_mask,
      cls_index,
      clslabel,
      per_input_ids,
      per_attention_mask
      ):
    self.input_ids = input_ids
    self.attention_mask = attention_mask
    self.lmlabels = lmlabels
    self.decoder_input_ids = decoder_input_ids
    self.decoder_attention_mask = decoder_attention_mask
    self.cls_index = cls_index
    self.clslabel = clslabel
    self.per_input_ids = per_input_ids
    self.per_attention_mask = per_attention_mask
```

Figure 30. PersonaDataset

```
class InferDataset(TensorDataset):
  def __init__(
      self,
      encoder_input_ids,
      decoder_input_ids,
      attention_mask,
      decoder_attention_mask,
      lmlabels
      ):
    self.encoder_input_ids = encoder_input_ids
    self.decoder_input_ids = decoder_input_ids
    self.attention_mask = attention_mask
    self.decoder_attention_mask = decoder_attention_mask
    self.lmlabels = lmlabels
```

Figure 31. InferDataset

```
# Initialize Data Loaders
logger.info("***Initiate Dataloader***")
train_sampler = torch.utils.data.distributed.DistributedSampler(train_dataset) if args.distributed else None
infer_sampler = torch.utils.data.distributed.DistributedSampler(infer_dataset) if args.distributed else None
val_sampler = torch.utils.data.distributed.DistributedSampler(val_dataset) if args.distributed else None
train_loader = DataLoader(train_dataset, sampler=train_sampler, batch_size=args.train_batch_size, shuffle=(not args.distributed), num_workers=0)
infer_loader = DataLoader(infer_dataset, sampler=infer_sampler, batch_size=args.infer_batch_size, shuffle=(not args.distributed), num_workers=0)
val_loader = DataLoader(val_dataset, sampler=val_sampler, batch_size=args.eval_batch_size, shuffle=False)
```

Figure 32. Data Loader

6.3 Model Building

For this section, the modification on the original BlenderBot architecture will first be discussed, then the main model of this experiment will be discussed in the latter part of this section.

Building the model is the most challenging aspect of this experiment, because this experiment involves changing the internal architecture of the pre-trained model which consist of many layers and configuration into a custom architecture to include the latent memory. Hence, a deep understanding on the architecture of the pre-trained model is required to amend the code correctly. First of all, in order to avoid modifying the pre-trained model's parts that does not require modification the layers that require modification were overridden using the same class name. For instance, in order to includes latent memory for control generation, the original BlenderBotDecoder layer was overridden by adding a latent_memory parameter in its forward method as shown in **Figure 33**. The forward function is what will be called automatically when the main model pass parameter to the layer.

```
class BlenderbotDecoder(BlenderbotPreTrainedModel):
    """
    Transformer decoder consisting of *config.decoder_layers* layers. Each layer is a [`BlenderbotDecoderLayer`]

    Args:
        config: BlenderbotConfig
        embed_tokens (nn.Embedding): output embedding

    Taken from, modifying forward to include latent memory -
    https://github.com/huggingface/transformers/blob/main/src/transformers/models/blenderbot_small/modeling_blenderbot_small.py
    """

    def __init__(self, config: BlenderbotConfig, embed_tokens: Optional[nn.Embedding] = None):
        super().__init__(config)
        self.dropout = config.dropout
        self.layerdrop = config.decoder_layerdrop
        self.padding_idx = config.pad_token_id
        self.max_target_positions = config.max_position_embeddings
        self.embed_scale = math.sqrt(config.d_model) if config.scale_embedding else 1.0

        if embed_tokens is not None:
            self.embed_tokens = embed_tokens
        else:
            self.embed_tokens = nn.Embedding(config.vocab_size, config.d_model, self.padding_idx)

        self.embed_positions = BlenderbotLearnedPositionalEmbedding(
            config.max_position_embeddings,
            config.d_model,
        )
        self.layers = nn.ModuleList([BlenderbotDecoderLayer(config) for _ in range(config.decoder_layers)])
        self.layer_norm = nn.LayerNorm(config.d_model)

        self.gradient_checkpointing = False
        # Initialize weights and apply final processing
        self.post_init()
```

```
def forward(
    self,
    input_ids=None,
    attention_mask=None,
    encoder_hidden_states=None,
    encoder_attention_mask=None,
    head_mask=None,
    cross_attn_head_mask=None,
    past_key_values=None,
    inputs_embeds=None,
    use_cache=None,
    output_attentions=None,
    output_hidden_states=None,
    return_dict=None,
    latent_memory=None
):
```

Figure 33. BlenderBotDecoder modification

Next, the latent_memory will be added into the hidden state produce by the decoder as hypothesis (persona information) as shown in Figure 31. For transformers-based architecture the hidden state three dimensions generally followed the conventions of shape (batch size, sequence length, features dimension) so the reason that latent_memory was added every first element within the second dimension (every first element in the input sequence) was because

the first element is a special token that mark the start of the sequence, hence here it was used to represent the hypothesis. This method followed the work of Chen et al., (2023), **Figure 34** shows the illustrate the concept of adding the latent memory as hypothesis (Chen et al., 2023).

```
# embed positions
positions = self.embed_positions(input_shape, past_key_values_length)

hidden_states = inputs_embeds + positions

# Add latent memory
if latent_memory != None:
  assert latent_memory.size(-1) == inputs_embeds.size(-1)
  hidden_states[:,0,:] = hidden_states[:,0,:] + latent_memory

hidden_states = nn.functional.dropout(hidden_states, p=self.dropout, training=self.training)
```

Figure 34. Implementing Latent Memory (persona hypothesis)

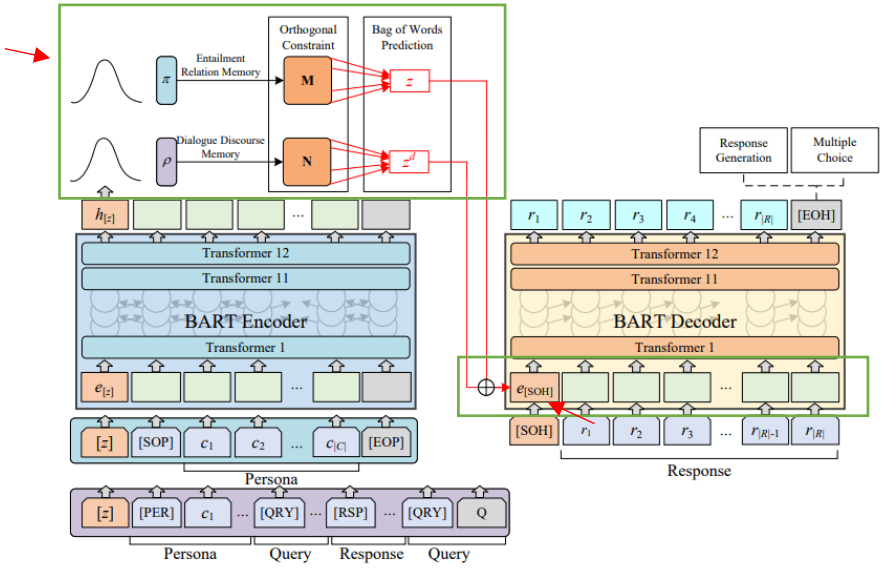

Figure 35. Latent memory as the start of hypothesis (SOH)

Moving forward, unlike Bart the BlenderBotModel does not have a classification head layer for sequence classification task. So, I introduce a classification head to my custom model building on top of the BlenderBotModel. **Figure 36** shows the implementation for the classification head. This class followed the Bart's classification head architecture consisting of dense layer and linear layer.

```python
class BlenderBotClassificationHead(nn.Module):
    """
    https://github.com/huggingface/transformers/blob/v4.37.2/src/tran
    """

    """Head for sentence-level classification tasks."""

    def __init__(
        self,
        input_dim: int,
        inner_dim: int,
        num_classes: int,
        pooler_dropout: float,
    ):
        super().__init__()
        self.dense = nn.Linear(input_dim, inner_dim)
        self.dropout = nn.Dropout(p=pooler_dropout)
        self.out_proj = nn.Linear(inner_dim, num_classes)

    def forward(self, hidden_states: torch.Tensor) -> torch.Tensor:
        hidden_states = self.dropout(hidden_states)
        hidden_states = self.dense(hidden_states)
        hidden_states = torch.tanh(hidden_states)
        hidden_states = self.dropout(hidden_states)
        hidden_states = self.out_proj(hidden_states)
        return hidden_states
```

Figure 36. BlenderBot classification head

Lastly, the main architecture for this model BlenderBotSpaceModel extends the pre-trained BlenderBot Model utilizing its' encoder, the overridden BlenderBotDecoder and the BlenderBotClassificationHead as a full model that fits the requirement of this experiment. First the configuration of this model includes several components shown in **Figure 37 and 38**.

```python
class BlenderBotSpaceModel(BlenderbotPreTrainedModel):
  """
  _prepare_encoder_decoder_kwargs_for_generation and generate from official github repo
  https://github.com/huggingface/transformers/blob/main/src/transformers/generation/utils.py override both of this method
  to fit requirement when doing generation
  """
  def __init__(self, config: BlenderbotConfig, num_token=None, num_entailment_latent=10, num_dialog_latent=10, num_labels=1):
    super().__init__(config)
    self.model = BlenderbotModel(config)
    self.register_buffer("final_logits_bias", torch.zeros((1, self.model.shared.num_embeddings)))
    self.lm_head = nn.Linear(config.d_model, self.model.shared.num_embeddings, bias=False)

    # BOW Prediction Layer
    if num_token != None:
      self.bow_head = nn.Linear(config.d_model, num_token)
    else:
      self.bow_head = nn.Linear(config.d_model, self.model.config.vocab_size)

    # Following Bart Configuration For Classification Head
    self.num_labels = num_labels
    self.classifier_dropout = 0.0

    # Sentence Level Classification
    self.classification_head = BlenderBotClassificationHead(
        config.d_model,
        config.d_model,
        self.num_labels,
        self.classifier_dropout,
    )

    self.model._init_weights(self.classification_head.dense)
    self.model._init_weights(self.classification_head.out_proj)

    # Latent Factors
    self.num_entailment_latent = num_entailment_latent
    self.num_dialog_latent = num_dialog_latent
```

Figure 37. BlenderBot Space Model Configuration Part 1

```
self.entailment_latent_head = BlenderBotClassificationHead(
    config.d_model,
    config.d_model,
    self.num_entailment_latent,
    self.classifier_dropout,
)

self.model._init_weights(self.entailment_latent_head.dense)
self.model._init_weights(self.entailment_latent_head.out_proj)

self.dialog_latent_head = BlenderBotClassificationHead(
    config.d_model,
    config.d_model,
    self.num_dialog_latent,
    self.classifier_dropout,
)

self.model._init_weights(self.dialog_latent_head.dense)
self.model._init_weights(self.dialog_latent_head.out_proj)

self.latent_memory_entailment = nn.Parameter(torch.randn(self.num_entailment_latent, config.d_model))
self.latent_memory_dialog = nn.Parameter(torch.randn(self.num_dialog_latent, config.d_model))

# Initialize weights and apply final processing
self.post_init()
```

Figure 38. BlenderBot Space Model Configuration Part 2

The BlenderBotSpaceModel is made up of a language BlenderBotEncoder, BlenderBotDecoder, model head (lm_head), which is a linear layer to transform hidden state into language model output logits for computing lm loss, similarly the bag of word head (bow_head) is used for outputting the bow logits for computing bow loss. The classification head is for sequence classification task, whereas the entail latent head and dialog latent head is for learning the entailment relation and memorizing the dialog discourse respectively.

```
if infer_lmlabels is not None:
  infer_encoder_outputs = self.model.encoder(
      input_ids=infer_input_ids,
      attention_mask=infer_attention_mask,
      head_mask=head_mask,
      inputs_embeds=inputs_embeds,
      output_attentions=output_attentions,
      output_hidden_states=output_hidden_states,
      return_dict=return_dict,
  )
  infer_latent_hidden_state = infer_encoder_outputs.last_hidden_state[:, 0, :]
  infer_latent_logits = self.entailment_latent_head(infer_latent_hidden_state)

  weight_memory = torch.mm(torch.softmax(infer_latent_logits, dim=-1), self.latent_memory_entailment)

  infer_decoder_outputs = self.model.decoder(
      input_ids=infer_decoder_input_ids,
      attention_mask=infer_decoder_attention_mask,
      encoder_hidden_states=infer_encoder_outputs[0],
      encoder_attention_mask=infer_attention_mask,
      head_mask=decoder_head_mask,
      cross_attn_head_mask=cross_attn_head_mask,
      past_key_values=past_key_values,
      inputs_embeds=decoder_inputs_embeds,
      use_cache=use_cache,
      output_attentions=output_attentions,
      output_hidden_states=output_hidden_states,
      return_dict=return_dict,
      latent_memory=weight_memory
  )

  infer_lm_logits = self.lm_head(infer_decoder_outputs[0]) + self.final_logits_bias
  loss_fnc = CrossEntropyLoss()
  infer_masked_lm_loss = loss_fnc(infer_lm_logits.view(-1, self.config.vocab_size), infer_lmlabels.view(-1).long())

  return infer_lm_logits, infer_masked_lm_loss

return None, None
```

Figure 39. Model's Architecture for Entailment Learning

Figure 39 shows the model sub architecture for entailment learning. For entailment learning the pre-trained BlenderBotEncoder will be used to generate embeddings (last hidden state) for the premises (contain only entailed premises), the embeddings will then pass through the entailment latent head to map the premises into latent space and then multiplied with the learnt weight. The weighted latent will be passed into the overridden BlenderBotDecoder to output the last hidden state which will be used to compute the logits and compute loss with respect to the hypothesis sentence.

```
dialog_latent_variable = None
entail_latent_variable = None
if input_ids is not None:
  if infer:
    per_encoder_outputs = self.model.encoder(
        input_ids=per_input_ids,
        attention_mask=per_attention_mask,
        head_mask=head_mask,
        inputs_embeds=inputs_embeds,
        output_attentions=output_attentions,
        output_hidden_states=output_hidden_states,
        return_dict=return_dict,)
    latent_hidden_state_entailment = per_encoder_outputs.last_hidden_state[:, 0, :]
    latent_logits_entailment = self.entailment_latent_head(latent_hidden_state_entailment)
    entail_latent_variable = torch.mm(torch.softmax(latent_logits_entailment, dim=-1), self.latent_memory_entailment)

  encoder_outputs = self.model.encoder(
      input_ids=input_ids,
      attention_mask=attention_mask,
      head_mask=head_mask,
      inputs_embeds=inputs_embeds,
      output_attentions=output_attentions,
      output_hidden_states=output_hidden_states,
      return_dict=return_dict,
  )

  latent_hidden_state_dialog = encoder_outputs.last_hidden_state[:, 0, :]

  if dialog:
    latent_logits_dialog = self.dialog_latent_head(latent_hidden_state_dialog)
    dialog_latent_variable = torch.mm(torch.softmax(latent_logits_dialog, dim=-1), self.latent_memory_dialog)
```

Figure 40. Model Architecture for Dialogue Generation Part 1

```
if decoder_input_ids is not None:
  if latent_variable is not None:
    input_latent = latent_variable
  elif dialog_latent_variable is not None or entail_latent_variable is not None:
    dialog_latent_variable = dialog_latent_variable if dialog_latent_variable is not None else torch.zeros_like(entail_latent_variable)
    entail_latent_variable = entail_latent_variable if entail_latent_variable is not None else torch.zeros_like(dialog_latent_variable)
    input_latent = dialog_latent_variable + entail_latent_variable
  else:
      input_latent = None

  decoder_outputs = self.model.decoder(
      input_ids=decoder_input_ids,
      attention_mask=decoder_attention_mask,
      encoder_hidden_states=encoder_outputs[0],
      encoder_attention_mask=attention_mask,
      head_mask=decoder_head_mask,
      cross_attn_head_mask=cross_attn_head_mask,
      past_key_values=past_key_values,
      inputs_embeds=decoder_inputs_embeds,
      use_cache=use_cache,
      output_attentions=output_attentions,
      output_hidden_states=output_hidden_states,
      return_dict=return_dict,
      latent_memory=input_latent,)
```

Figure 41. Model Architecture for Dialogue Generation Part 2

Figure 40 and Figure 41 shows the model sub architecture for dialogue generation learning. Here the model will first map the persona into entailment latent space and the dialog history into discourse memory. The discourse latent and entailment latent is then summed together to produce a single latent memory that will be fed into the decoder. Finally, the decoder will output the last hidden states. The last hidden states can then be used to compute loss during training or used along with a search algorithm (beam search, greedy search, and others) to guide the text generation during inference.

```
  lm_logits = self.lm_head(decoder_outputs[0]) + self.final_logits_bias
  if input_ids is None:
      lm_logits = lm_logits.view(bs, -1, lm_logits.size(-1))
      return Seq2SeqLMOutput(
          loss=None,
          logits=lm_logits,
          past_key_values=decoder_outputs.past_key_values,
          decoder_hidden_states=decoder_outputs.last_hidden_state,
          decoder_attentions=decoder_outputs.attentions,
          cross_attentions=decoder_outputs.cross_attentions,
          encoder_last_hidden_state=encoder_outputs.last_hidden_state,
          encoder_hidden_states=encoder_outputs.hidden_states,
          encoder_attentions=encoder_outputs.attentions,
      )
  seq_len = decoder_input_ids.size(-1)
  bow_logits = self.bow_head(input_latent).repeat(seq_len, 1, 1).transpose(0,1).contiguous()
  hidden_states = decoder_outputs[0]

masked_lm_loss = None
bow_loss = None
if lmlabels is not None:
  loss_fnc = CrossEntropyLoss()
  masked_lm_loss = loss_fnc(lm_logits.view(-1, self.config.vocab_size), lmlabels.view(-1))
  bow_loss = loss_fnc(bow_logits.view(-1, self.config.vocab_size), lmlabels.view(-1))
  lm_logits = lm_logits.view(bs, -1, lm_logits.size(-1))
```

Figure 42. Computing Model Loss Part 1

```
cls_logits = None
if cls_index != None:
  cls_mask = cls_index.eq(self.config.eos_token_id)
  if len(torch.unique_consecutive(cls_mask.sum(1))) > 1:
      raise ValueError("All examples must have the same number of <eos> tokens.")
  sentence_representation = hidden_states[cls_mask, :].view(hidden_states.size(0), -1,
                                                            hidden_states.size(-1))[:, -1, :] # last hidden states (masked [pad not included])
  cls_logits = self.classification_head(sentence_representation)

cls_loss = None
if clslabel is not None:
  loss_fnc = CrossEntropyLoss()
  cls_loss = loss_fnc(cls_logits.view(bs, -1), clslabel.view(-1))
  cls_logits = cls_logits.view(bs, -1)

m_loss = None
if input_ids is not None and infer and dialog:
    m_fnc = MemoryLoss()
    m_loss = m_fnc(self.latent_memory_entailment, self.latent_memory_dialog)
```

Figure 43. Computing Model Loss Part 2

Figure 42 and Figure 43 shows the loss computation to be used for gradient optimization. The total loss for dialogue generation learning is the combination of 4 losses, namely, the language

model loss, the bag of word loss, the sequence classification loss, and the memory loss (Chen et al., 2023). The language model loss is the was computed from the decoder last hidden state with respect to the target response. The classification loss is the sequence classification on whether positive response is ranked as the top candidate. The bag of word loss is used to facilitate the latent variable (Chen et al., 2023). Lastly, the memory loss (Chen et al., 2023) is computed as **Figure 44** is primarily to act as orthogonal constraint between the entailment relation memory and dialogue discourse memory. As stated by Chen et al., (2023), the dialog context and the entailment relation should not be dependent with each other so the orthogonal constraint was used to reduce their correlation (Chen et al., 2023). The idea is to minimize the cosine similarity between them.

```
class MemoryLoss(nn.Module):
    def __init__(self, eps=1e-8):
        super(MemoryLoss, self).__init__()
        self.eps = eps
    def forward(self, memory1, memory2):
        memory1 = F.normalize(memory1)
        memory2 = F.normalize(memory2)
        dot_production = torch.mm(memory1, memory2.T)
        dot_size = dot_production.size(0)
        loss = dot_production.norm()

        return loss
```

Figure 44. Memory Loss (Orthogonal Constraint)

6.4 Training

The default training epochs for this experiment 3 provides the best result, following Chen et al., (2023) BlenderBotSpace also run two training stage, first stage for learning entailment and second stage for dialogue generation learning. **Figure 45 and 46** shows the overall model training fitting processes for each epoch. For every epoch, the entailment learning will first be done by looping through the mini batches, which the batch size in this experiment was defaulted with 68. Once the entailment learning is done the training will proceed with the dialogue generation training, similarly by looping through its' mini batches, which was defaulted to 4 for this experiment. After that, at the end of each epoch, validation will be conducted on the CONVAI2 development set (Dinan et al., 2019) to evaluate and observe the performance of the model, to determine whether the model needed to be tuned and whether there are any room for improvement. The detail of the evaluation process will be discussed in **section 6.5** below.

```python
def _run_epoch(self, epoch):
  if self.distributed:
    if self.infer_loader: self.infer_loader.sampler.set_epoch(epoch)
    self.train_loader.sampler.set_epoch(epoch)
    self.infer_loader.sampler.set_epoch(epoch)

  # ---Infer Training---
  if self.infer_loader:
    b_sz_infer = self.infer_loader.batch_size
    print(f"[Device{self.device}] Epoch: {epoch} | Batchsize: {b_sz_infer} | Steps: {len(self.infer_loader)} | Loader: Infer Loader")
    for idx, batch in enumerate(tqdm(self.infer_loader, desc="Training Infer")):
      # scheduler step
      # self.scheduler_infer.step()
      # current_lr = self.scheduler_infer.get_param()
      # for param_group in self.optimizer_infer.param_groups:
      #   param_group['lr'] = current_lr
      self.scheduler_infer(idx)
      # print("LR: ", self.optimizer_infer.param_groups[0]['lr'])
      batch = tuple(tensor.to(self.device) for tensor in batch)
      infer_loss = self._run_batch_infer(batch)
      step = epoch * len(self.infer_loader) + idx
      logger.info(f"infer loss:{infer_loss['loss']} step: {step}")
      self.tb_writer.add_scalar("infer loss", infer_loss['loss'], global_step=step)
```

Figure 45. Training processes for every epoch part 1

```python
# ---Training---
b_sz_train = self.train_loader.batch_size
print(f"[Device{self.device}] Epoch: {epoch} | Batchsize: {b_sz_train} | Steps: {len(self.train_loader)} | Loader: Train Loader")
for idx, batch in enumerate(tqdm(self.train_loader, desc="Training")):
  self.iter_state += 1 # update iteration counter
  # scheduler step
  # self.scheduler.step()
  # current_lr = self.scheduler.get_param()
  # for param_group in self.optimizer.param_groups:
  #   param_group['lr'] = current_lr
  self.scheduler(idx)
  # print("LR: ", self.optimizer.param_groups[0]['lr'])
  batch = tuple(tensor.to(self.device) for tensor in batch)
  train_loss = self._run_batch_train(batch) # loss, cls, lm, mem, bow
  step = epoch* len(self.train_loader) + idx
  logger.info(f"train loss:{train_loss['loss']} step: {step}")
  self.tb_writer.add_scalar("train loss", train_loss['loss'], global_step=step) # concat loss
  self.tb_writer.add_scalar("cls loss", train_loss['cls'], global_step=step) # classification
  self.tb_writer.add_scalar("lm loss", train_loss['lm'], global_step=step) # language model
  self.tb_writer.add_scalar("mem loss", train_loss['mem'], global_step=step)
  self.tb_writer.add_scalar("bow loss", train_loss['bow'], global_step=step) # bag of word

# ---Eval---
batch_acc = []
batch_nll = []
b_sz_eval = self.eval_loader.batch_size
print(f"[Device{self.device}] Epoch: {epoch} | Batchsize: {b_sz_eval} | Steps: {len(self.eval_loader)} | Loader: Eval Loader")
for idx, batch in enumerate(tqdm(self.eval_loader, desc="Evaluating")):
  batch = tuple(tensor.to(self.device) for tensor in batch)
  (_logits_lm, _logits_cls), (_labels_lm, _labels_cls) = self._run_batch_eval(batch)
  metrics = Metrics(_logits_lm, _labels_lm, _logits_cls, _labels_cls)
  accuracy, nnl = metrics.accuracy, metrics.nnl
  batch_acc.append(accuracy)
  batch_nll.append(nnl)
return batch_acc, batch_nll
```

Figure 46. Training processes for every epoch part 2

For every mini-batches, the evaluation results will be stored in the arrays to compute its average performance based for the respective metrics by summing the metrics and dividing them by the number of batches at the end of the epoch as shown in **Figure 47**, the results will also be recorded into tensorboard.

```python
def fit(self, max_epochs: int):
  for epoch in range(self.epochs_run, max_epochs):
    batch_acc, batch_nll = self._run_epoch(epoch)
    avg_acc = self._average_distributed_scalar(sum(batch_acc) / len(batch_acc))
    avg_nnl = self._average_distributed_scalar(sum(batch_nll) / len(batch_nll))
    avg_ppl = torch.exp(avg_nnl) # https://huggingface.co/docs/transformers/en/perplexity
    # Logging ppl in tensorboard
    logger.info(f"average accuracy:{avg_acc} epoch: {epoch}")
    logger.info(f"average perplexity:{avg_ppl} epoch: {epoch}")
    self.tb_writer.add_scalar("average accuracy", avg_acc, epoch)
    self.tb_writer.add_scalar("average perplexity", avg_ppl, epoch)
  # Save model
  model_checkpoint_path = os.path.join(self.args.save_dir, "model/")
  self.model.save_pretrained(model_checkpoint_path)
  # Save training arguments
  args_path = os.path.join(self.args.save_dir, 'training_args.bin')
  torch.save(self.args, args_path)
  # Save model configurations
  config_path = os.path.join(self.args.save_dir, CONFIG_NAME)
  model_config = getattr(self.model, 'module', self.model).config.to_json_file(config_path)
  # Close tensorboard writer
  self.tb_writer.close()
```

Figure 47. Model Fitting and performance logging

6.5 Evaluation

```python
class Metrics(object):
  def __init__(self, lm_pred_logits, lm_true, cls_pred_logits, cls_true):
    """
    cls_true: tensor[0,0] first element (of first item in current batch), first element (of second item in current batch)
    cls_pred_logits: tensor[batch_size, candidate_seq_len] (batch size is 2 for eval)
    """
    # Move tensors to CPU
    lm_pred_logits, lm_true, cls_pred_logits, cls_true = (
        lm_pred_logits.cpu(),
        lm_true.cpu(),
        cls_pred_logits.cpu(),
        cls_true.cpu()
    )

    # classification metrics
    self.accuracy = sk_metrics.accuracy_score(cls_true.view(-1), torch.argmax(cls_pred_logits, dim=1))

    # language model metrics
    flat_lm_logits = lm_pred_logits.view(-1, lm_pred_logits.size(-1))
    flat_lm_true = lm_true.view(-1)
    self.loss_fnc = CrossEntropyLoss(ignore_index=-100)
    self.nnl = self.loss_fnc(flat_lm_logits, flat_lm_true)
```

Figure 48. Evaluation Metrics during training

In order to evaluate the model during training time, perplexity and accuracy metrics were used on the validation set. **Figure 48** shows the method for computing the accuracy and perplexity. The accuracy was computed using the sklearn library to determine whether the model was able to rank the positive candidate as the top candidate. On the other hand, the cross-entropy loss function was used to compute the negative log likelihood between the language model prediction with the actual expected response. This negative log likelihood value will then be used to compute the perplexity, which is defined as the exponential of the negative log likelihood.

7.0 Tuning & Validation

7.1 Hyperparameter Tuning & Processing Power

```
# hyperparam
parser.add_argument("--lr", type=float, default=5e-6, help="learning rate") # 5e-6
parser.add_argument('--max_grad_norm', type=float, default=1.0)
parser.add_argument("--num_entailment_latent", type=int, default=10, help="number of entailment latent")
parser.add_argument("--num_dialog_latent", type=int, default=10, help="number of dialog latent")
parser.add_argument("--gradient_accumulation_steps", type=int, default=8, help="Accumulate gradients on several steps")
# training setting
parser.add_argument('--seed', type=int, default=783435, metavar='S', help='random seed')
parser.add_argument("--train_batch_size", type=int, default=4, help="batch size for dialogue training")
parser.add_argument("--infer_batch_size", type=int, default=128, help="batch size for infer training") # default 128 / 2
parser.add_argument("--eval_batch_size", type=int, default=2, help="batch size for evaluation")
parser.add_argument("--epochs", type=int, default=3, help="number of training epochs") #
parser.add_argument('--revised', action='store_true', default=False, help='use revised dataset (for PersonaChat dataset)')
parser.add_argument('--eval', action='store_true', default=False, help='eval model')
parser.add_argument("--eval_before_start", action='store_true', help="if true start with a first evaluation before training")
parser.add_argument('--cand', type=int, default=5, help='number of candidate')
parser.add_argument("--max_history", type=int, default=4, help="number of dialogue context")
```

Figure 49. Hyperparameter

For this experiment there are several tuneable parameters as shown in **Figure 49**, first is the learning rate which was set to start with a value small value 5e-6 (0.000005) referencing multiple works that used pre-trained model (Chen et al., 2023) (Lin et al., 2021) (Liu et al., 2020), it was determined that since the models was initially trained on large amount of data the fine-tuning of the pre-trained model on downstream task should start with a relatively small value. The gradient accumulation step is the amount of mini-batches iterated before performing parameter update to help simulating a larger batch size due to limited processing power. In this experiment, the original batch size for the entailment learning (infer batch size) was set to 64 due to the limited processing resources in Google Colab. One thing to note is that in order to replicate this experiment in the same settings that provide the best result, Google Colab Pro is needed to access the best provided GPU, namely, the A100 GPU which have 40GB GPUs RAM. If not available, the infer batch size will need to be set to 64 and the epochs will need to be set to be lower. In the minimal setting, which was setting the infer batch size to 64 and epoch to 1 the performance has only slight reduction tested in this experiment. The free version T4 GPU provided by Google Colab should not have problem running the minimal setting training. Continuing with the tuneable parameter, the candidate can also be tune, which is the number of false responses that wanted to add for training the model sequence classification. Lastly, is the max conversation history that the model should remember, which in this experiment the dialogue context was set to 4. Since the initial hyperparameter setting have resulted in a good performance, this experiment settled with these settings. Another factor is the size of the proposed model in this experiment, which is expensive to train continuously and each run will take about 4 hours, so future work can be extended on this experiment to test several other combinations of hyperparameters if computation resources are available.

7.2 Validation

After that training is done, the model would need to be validated. For this experiment, three evaluation metrics will be used on the CONVAI2 test set (Dinan et al., 2019) which contains data unseen during the training process. The evaluation metrics are Hits@1 for measuring how many time the model accurately rank the positive response as top candidate, Perplexity to measure how well the prediction sequence are with respect to the actual response, and lastly the F1 score in word level to evaluate the precision and recall between the predicted words and response words.

```
class LogitsMetrics(object):
  def __init__(self, lm_pred_logits, lm_true, cls_pred_logits, cls_true):
    """
    cls_true: tensor[0,0] first element (of first item in current batch), first element (of second item in current batch)
    cls_pred_logits: tensor[batch_size, candidate_seq_len] (batch size is 2 for eval)
    """
    # Move tensors to CPU
    lm_pred_logits, lm_true, cls_pred_logits, cls_true = (
        lm_pred_logits.cpu(),
        lm_true.cpu(),
        cls_pred_logits.cpu(),
        cls_true.cpu()
    )

    # classification metrics
    self.accuracy = sk_metrics.accuracy_score(cls_true.view(-1), torch.argmax(cls_pred_logits, dim=1))

    # language model metrics
    flat_lm_logits = lm_pred_logits.view(-1, lm_pred_logits.size(-1))
    flat_lm_true = lm_true.view(-1)
    self.loss_fnc = CrossEntropyLoss(ignore_index=-100)
    self.nnl = self.loss_fnc(flat_lm_logits, flat_lm_true)

    # hits@1 metrics
    self.hits_at_1 = self.hits_at_1(cls_pred_logits, cls_true)

  def hits_at_1(self, cls_pred_logits, cls_true):
    predicted_labels = torch.argmax(cls_pred_logits, 1)
    correct_predictions = torch.eq(predicted_labels, cls_true)
    hits_at_1 = correct_predictions.sum().item() / len(cls_true)
    return hits_at_1
```

Figure 50. Perplexity and Hits@1 metrics

Figure 50 shows the computation methods for perplexity and hits@1, the perplexity was computed similarly to how it was during training, specifically, by computing the negative log likelihood and then using it to compute the exponential of the negative log likelihood which is the perplexity score. The hits@1 was computed by getting the predicted probability distribution that provide max classification score and then comparing the predicted sequence with the true sequence to check if they are equal. Then sum of the correct prediction divided by the total comparison done is the hits@1 score.

```python
class F1Metrics(object):
  def __init__(self, reference, generated):
    self.reference = self.remove_punc(reference)
    self.generated = self.remove_punc(generated)
    common_tokens = Counter(self.reference) & Counter(self.generated)
    precision = sum(common_tokens.values()) / len(self.generated) if len(self.generated) > 0 else 0
    recall = sum(common_tokens.values()) / len(self.reference) if len(self.reference) > 0 else 0
    self.f1_score = (2 * precision * recall) / (precision + recall) if (precision + recall) > 0 else 0

  def remove_punc(self, text):
    translator = str.maketrans("", "", s.punctuation)
    out = text.translate(translator)
    out = out.split()
    return out
```

Figure 51. F1 metric

```python
def _run_batch_test_f1(self, batch):
  self.model.eval()
  with torch.no_grad():
    gen_input_ids, gen_attention_mask, gen_per_input_ids, gen_per_attention_mask, history, response = batch

    # generation (f1)
    out_ids = self.model.generate(
        inputs=gen_input_ids,
        attention_mask=gen_attention_mask,
        per_input_ids=gen_per_input_ids,
        per_attention_mask=gen_per_attention_mask,
        max_length=self.args.max_length,
        num_beams=self.args.beam
        )

    out_text = self.tokenizer.batch_decode(
        out_ids,
        skip_special_tokens=True,
        spaces_between_special_tokens=False,
        clean_up_tokenization_spaces=True
        )

    ans = out_text[0].strip()
    print(f"History: '{history[0]}' | Generated: '{ans}' | Target: '{response[0]}'")

    return ans, response
```

Figure 52. Beam Search generation

Figure 51 shows the computation of the F1 score, where the generated text will be tokenized and compared with reference text which will also be tokenized. The precision and recall will be computed by evaluating the common token between them and then used to compute the F1 score. Since the output of the original model was the last hidden state which is in a format that human cannot understand, hence, transformer library provides a generate function for generative model to use various search algorithm that utilize their last hidden state to guide text generation. For this experiment the beam search with a beam size of 2 was used for generating the sequence texts. **Figure 52** shows the computation of beam search.

8.0 Visualization & Critical Analysis

8.1 Training Analysis

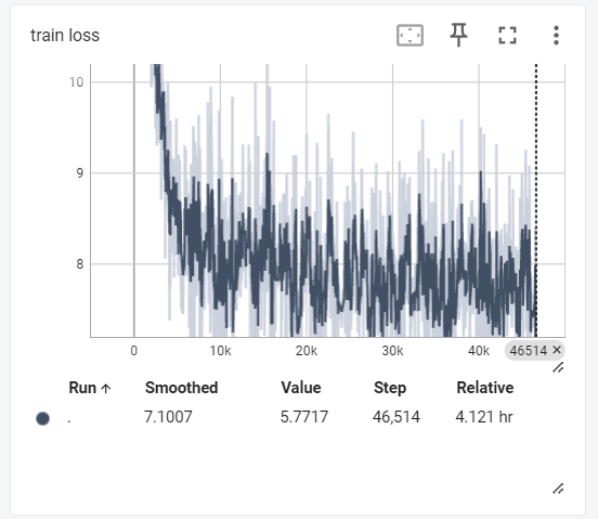

Figure 53. Training Loss

Figure 53 shows the visualization for the loss during the training process. Based on the graph it can be seen that the loss slowly converges from about 14 to 5 as the training end. As the loss is computed by the sum of 4 types of losses as mentioned in previous sections the graph may look unsmooth, but overall, the loss is reducing as the training goes on. It was also shown that the training took around 4.121 hours.

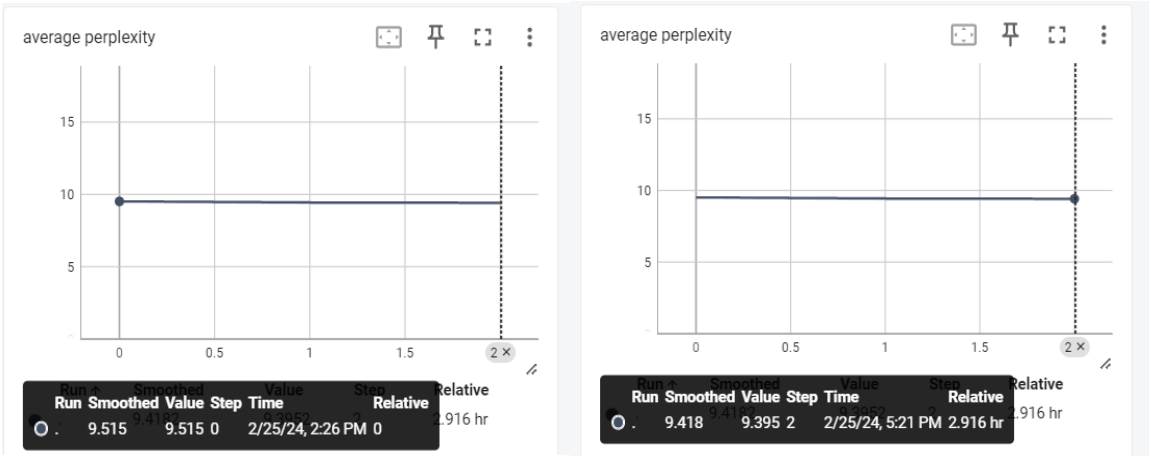

Figure 54. Perplexity Score during training

Figure 54 shows the perplexity score for each epoch during training. For each epoch the perplexity was reduced by slight from the first epoch having 9.515 perplexity to the last epoch having 9.395 perplexity showing 0.15 improvement after 3 epochs.

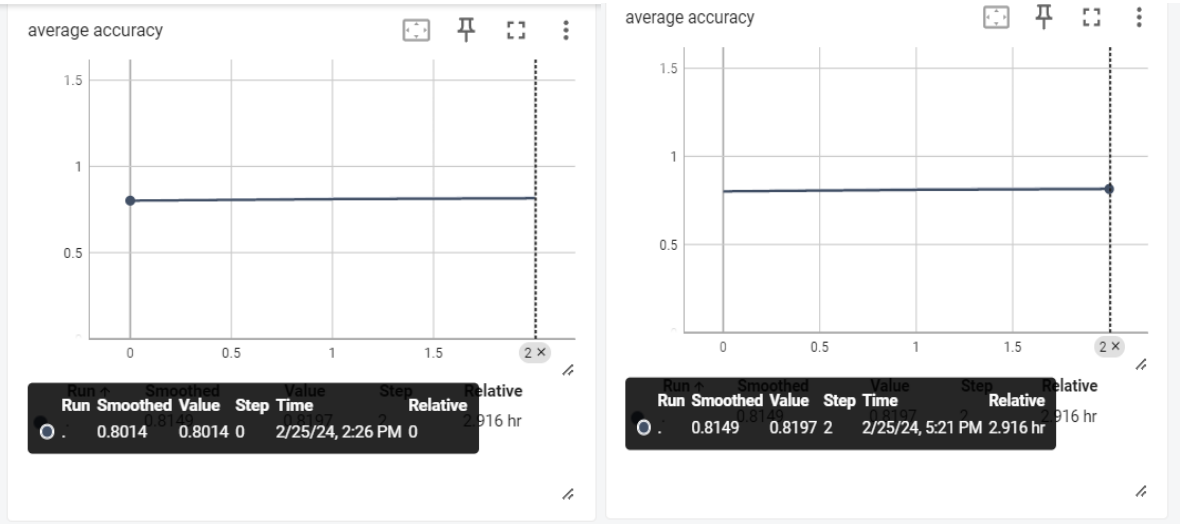

Figure 55. Accuracy Score during training

Figure 55 shows the accuracy scores during the training process. Similarly, to the perplexity the accuracy increase as the training goes on for each epoch. The accuracy was increased by about 2% at the end of the training. Based on the training analysis, it can be seen that the increase of the accuracy which is computed from the sequence classification task also result in the decrement of the perplexity indicating that the sequence classification helps in some way to improve the text generation performance.

8.2 Experimental Result

Table 2. Hits@1, Perplexity, and F1 benchmark on CONVAI2 test set

Model	Hits@1	Perplexity	F1
Bart with entailment and discourse memory (Chen et al., 2023)	**89.5**	10.99	**21.99**
TransferTransfo (Wolf et al., 2019)	82.1	17.51	19.09
p^2 bot (Liu et al., 2020)	81.9	15.21	19.77
Adapter-Bot (Lin et al., 2021)	-	11.08	15.0
BoB (Song et al., 2021)	-	**7.8**	-
GPT-2 (Lin et al., 2021)	-	13.13	-
BART (Lin et al., 2021)	-	11.9	20.7
LIC (Chen et al., 2023)	17.3	-	17.79
BlenderBot Space (Ours)	81.49	9.36	21.47

Table 2 shows the results of validation results of this experiment comparing with other benchmark models. Based on the result it can be seen that our model achieved the second highest in Perplexity and F1 score among the other model having (1.56) for perplexity and (.52) for F1 score differences with the best model by Song et al., (2021) and Chen et al., (2023) respectively. On the other hand, our model achieved a Hits@1 score of 81.49 which is not much differences with the other models. Another thing to note is that, most of the models above uses pre-trained architecture that had a classification head trained for sequence classification, but our model initially does not have such mechanism which was later introduced in our architecture. This indicate that our model is training the classification head from scratch, which was why our model have a lower Hits@1 score compared to the others. Nevertheless, our model had shown a good performance even when compared with the current state-of-the-art models.

8.3 Critical Analysis

Overall, there are several findings within this experiment, firstly based on the training analysis the sequence classification task helps the model improve performance by reducing the perplexity, the trend shown that when the accuracy increase (sequence classification) the perplexity decrease meaning the model are generating responses relating to the target response. Next, through my own observation on the generated text by model, I have identified that the generated texts take into account of the persona information, meaning the model was able to generate creative responses by incorporating personality to the generated statement, additionally the generated texts are often consistent to the conversational context as well. This shown that the method proposed by Chen et al., (2023) which incorporate latent space variables to control the text generation is effectively. However, do note that this is only my own speculation throughout the experiment as to actually evaluate such properties accurately require multiple human experts and can be expensive. Hence, unfortunately it can only be speculated as my own observation for this experiment.

Even though the proposed model in this experiment shows great performance, there are several limitations involved. First, the model can only receive input sequence up to length of 128 as it is what the model settings was for the original pre-trained model. This also indicate that the total words from context of the conversation (4 sequences) cannot be more than 128 words. This should not cause an issue for casual conversation as typically the sequences are

not lengthy, but for knowledge specific conversations alternative pre-trained model such as Bart may serve as a better model. Nevertheless, the experiment in this paper have shown that the latent memory for consistent dialogue generation by Chen et al., (2023) is effective and transferable to various pre-trained model's architectures. Additionally, this method can also be applied to other aspect instead of only for persona, for instance it may be extended to QnA chatbot by memorizing the domain specific knowledge to determine whether the generated responses entail with the domain knowledge.

References

Chen, R., Wang, J., Yu, L.-C., & Zhang, X. (2023). Learning to Memorize Entailment and Discourse Relations for Persona-Consistent Dialogues. *Proceedings of the ... AAAI Conference on Artificial Intelligence*, *37*(11), 12653–12661. https://doi.org/10.1609/aaai.v37i11.26489

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2018, October 11). *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. ArXiv.org. https://arxiv.org/abs/1810.04805

Dharma, E., Lumban Gaol, F., Leslie, H., Warnars, H., & Soewito, B. (2022). THE ACCURACY COMPARISON AMONG WORD2VEC, GLOVE, AND FASTTEXT TOWARDS CONVOLUTION NEURAL NETWORK (CNN) TEXT CLASSIFICATION. *Journal of Theoretical and Applied Information Technology*, *31*(2). https://www.jatit.org/volumes/Vol100No2/5Vol100No2.pdf

Dinan, E., Varvara Logacheva, Malykh, V., Miller, A. L., Shuster, K., Urbanek, J., Douwe Kiela, Szlam, A., Iulian Vlad Serban, Lowe, R. J., Shrimai Prabhumoye, Black, A. W., Rudnicky, A. I., Williams, J., Pineau, J., Mikhail Burtsev, & Weston, J. (2019). The Second Conversational Intelligence Challenge (ConvAI2). *ArXiv (Cornell University)*. https://doi.org/10.48550/arxiv.1902.00098

Ethayarajh, K. (2019). How Contextual are Contextualized Word Representations? Comparing the Geometry of BERT, ELMo, and GPT-2 Embeddings. *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. https://doi.org/10.18653/v1/d19-1006

Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, *9*(8), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

Houlsby, N., Giurgiu, A., Jastrze¸bski, S., Morrone, B., De Laroussilhe, Q., Gesmundo, A., Attariyan, M., & Gelly, S. (2019). *Parameter-Efficient Transfer Learning for NLP*. https://arxiv.org/pdf/1902.00751.pdf

Khurana, D., Koli, A., Khatter, K., & Singh, S. (2022). Natural language processing: state of the art, current trends and challenges. Multimedia Tools and Applications, 82, 3713–3744. https://doi.org/10.1007/s11042-022-13428-4

Lee, J., Oh, M., & Lee, D. (2023). *P5: Plug-and-Play Persona Prompting for Personalized Response Selection*. https://doi.org/10.18653/v1/2023.emnlp-main.1031

Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., Stoyanov, V., Zettlemoyer, L., & Ai, F. (2019). *BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension*. https://arxiv.org/pdf/1910.13461.pdf

Lewis, P., Perez, E., Aleksandara Piktus, Petroni, F., Vladimir Karpukhin, Goyal, N., Heinrich Küttler, Lewis, M., Yih, W., Rocktäschel, T., Riedel, S., & Douwe Kiela. (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. *ArXiv (Cornell University)*.

Liu, Q., Chen, Y., Chen, B., Lou, J.-G., Chen, Z., Zhou, B., & Zhang, D. (2020). You Impress Me: Dialogue Generation via Mutual Persona Perception. *ArXiv (Cornell University)*. https://doi.org/10.18653/v1/2020.acl-main.131

Liu, S., Cho, H., Freedman, M. R., Ma, X., & May, J. (2023). *RECAP: Retrieval-Enhanced Context-Aware Prefix Encoder for Personalized Dialogue Response Generation*. https://doi.org/10.18653/v1/2023.acl-long.468

Lin, Z., Madotto, A., Bang, Y., & Fung, P. (2021). The Adapter-Bot: All-In-One Controllable Conversational Model. *Proceedings of the AAAI Conference on Artificial Intelligence*, *35*(18), 16081–16083. https://doi.org/10.1609/aaai.v35i18.18018

Liu, X., Zheng, Y., Du, Z., Ding, M., Qian, Y., Yang, Z., & Tang, J. (2023). GPT understands, too. *AI Open*. https://doi.org/10.1016/j.aiopen.2023.08.012

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, *5*(4), 115–133. https://doi.org/10.1007/bf02478259

Otter, D. W., Medina, J. R., & Kalita, J. K. (2020). A Survey of the Usages of Deep Learning for Natural Language Processing. *IEEE Transactions on Neural Networks and Learning Systems*, *32*(2), 1–21. https://doi.org/10.1109/TNNLS.2020.2979670

Pandey, S., & Sharma, S. (2023). *A comparative study of retrieval-based and generative-based chatbots using Deep Learning and Machine Learning*. *3*, 100198–100198. https://doi.org/10.1016/j.health.2023.100198

Roller, S., Dinan, E., Goyal, N., Ju, D., Williamson, M., Liu, Y., Xu, J., Ott, M., Shuster, K., Smith, E. M., Boureau, Y.-L., & Weston, J. (2020). Recipes for building an open-domain chatbot. *ArXiv:2004.13637 [Cs]*. https://arxiv.org/abs/2004.13637

Sarker, I. H. (2021). Deep Learning: A Comprehensive Overview on Techniques, Taxonomy, Applications and Research Directions. *SN Computer Science*, *2*(6). Springer. https://doi.org/10.1007/s42979-021-00815-1

Shickel, B., & Rashidi, P. (2020). Sequential Interpretability: Methods, Applications, and Future Direction for Understanding Deep Learning Models in the Context of Sequential Data. *ArXiv (Cornell University)*.

Song, H., Wang, Y., Zhang, K., Zhang, W., & Liu, T. (2021). BoB: BERT Over BERT for Training Persona-based Dialogue Models from Limited Personalized Data. *ArXiv (Cornell University)*. https://doi.org/10.18653/v1/2021.acl-long.14

Staudemeyer, Ralf C, & Morris, E. R. (2019). Understanding LSTM -- a tutorial into Long Short-Term Memory Recurrent Neural Networks. *ArXiv (Cornell University)*. https://doi.org/10.48550/arxiv.1909.09586

Schmidt, R. M. (2019). Recurrent Neural Networks (RNNs): A gentle Introduction and Overview. *ArXiv (Cornell University)*. https://doi.org/10.48550/arXiv.1912.05911

Williams, A., Nangia, N., & Bowman, S. R. (2017). A Broad-Coverage Challenge Corpus for Sentence Understanding through Inference. *ArXiv (Cornell University)*. https://doi.org/10.48550/arxiv.1704.05426

Wolf, T., Sanh, V., Julien Chaumond, & Delangue, C. (2019). TransferTransfo: A Transfer Learning Approach for Neural Network Based Conversational Agents. *ArXiv (Cornell University)*.

Van Houdt, G., Mosquera, C., & Nápoles, G. (2020). A review on the long short-term memory model. *Artificial Intelligence Review, 53*(8). https://doi.org/10.1007/s10462-020-09838-1

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A., Kaiser, Ł., & Polosukhin, I. (2017). *Attention Is All You Need*. https://arxiv.org/pdf/1706.03762.pdf

Xiao, J., & Zhou, Z. (2020, June 1). *Research Progress of RNN Language Model*. IEEE Xplore. https://doi.org/10.1109/ICAICA50127.2020.9182390

Welleck, S., Kulikov, I., Roller, S., Dinan, E., Cho, K., & Weston, J. (2019, September 26). *Neural Text Generation with Unlikelihood Training*. ArXiv.org. https://doi.org/10.48550/arXiv.1908.04319

Welleck, S., Weston, J., Szlam, A., & Cho, K. (2019). *Dialogue Natural Language Inference*. Retrieved February 27, 2024, from https://arxiv.org/pdf/1811.00671.pdf

Zhu, S., Ma, T., Rong, H., & Najla Al-Nabhan. (2023). A Personalized Multi-Turn Generation-Based Chatbot with Various-Persona-Distribution Data. *Applied Sciences, 13*(5), 3122–3122. https://doi.org/10.3390/app13053122

Zhang, S., Dinan, E., Urbanek, J., Szlam, A., Kiela, D., & Weston, J. (2018). *Personalizing Dialogue Agents: I have a dog, do you have pets too?* https://arxiv.org/pdf/1801.07243.pdf