Incorporating Memory into Deep Generative Dialogue Models using a Scalable Attention Mechanism

by

Kira A. Selby

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computational Mathematics

Waterloo, Ontario, Canada, 2018

Abstract

Deep generative dialogue models have been a site of considerable research in recent years. The current state of the art is the "sequence to sequence" model, which uses a recurrent encoder-decoder architecture to construct a response to a given input message. These models are powerful, but condition their responses only on the previous utterance, rather than the entire conversation history. This paper explores the use of an attention mechanism over encoded conversation history to provide the model with context to improve its responses. We propose a computationally efficient variation of the memory network developed by Sukhbaatar et al. [14] which reduces the training time and requires fewer encoder evaluations during training and inference.

Acknowledgements

I would like to acknowledge the contributions of my supervisor, Dr. Pascal Poupart, who I am deeply grateful to for his unfailing support and generosity.

This paper also owes an enormous debt of thanks to Nabiha Asghar, who has been an incredible mentor to me throughout my graduate education, and contributed both research and editing.

Great thanks go as well to Lili Mou, Mike Rudd, Kevin Pantasdo, and Pranav Subramani, all of whom contributed research.

I would also like to thank Dr. Jeff Orchard, Amanda Guderian, and the rest of the Computational Mathematics staff and students for creating a fantastic learning environment and a wonderful community in which to pursue my Master's Degree.

I would also like to thank Dr. Pascal Poupart and Dr. Jeff Orchard again for serving as my committee members.

Code was used from Sean Robertson's Practical Pytorch repository.

Models were trained using the University of Waterloo Math Faculty Computing Facility's research GPU servers.

Dedication

This report is dedicated to my wonderful fiancee Nicole Dumont.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Conversational agents are an extremely important application of artificial intelligence. They have applications in customer service, mobile applications, computer assistants (such as Alexa or Siri), and many other regimes. Most conversation models fall into one of two categories: retrieval-based models and generative models. Retrieval-based models use algorithms to match the user input or current conversation context to a pregenerated response from the model's database. As a result, these models cannot innovate, but are very reliable and thus widely used in industry. Generative models forego pregenerated responses and instead generate their own responses from scratch. This makes these models a site of great theoretical interest due to their capacity to innovate, but also means that these models must solve a much more difficult task. Generative models are thus less reliable, and still an area of active development. The majority of generative conversational agents rely on a "sequence to sequence" architecture using recurrent neural networks. These architectures are powerful, but also extremely limited in that they construct a response to a query based solely on the query itself, and no other context. This thesis will explore the basics of sequence to sequence models as generative conversational agents, and explore ways to augment these models with a capacity to understand and use conversation history when constructing responses.

Chapter 2

Background

2.1 Recurrent Neural Networks

The current gold standard for generative conversational models is the Recurrent Neural Network (RNN) architecture. RNNs are a generalization of the neural network paradigm that has recently engulfed the machine learning community at large, and produced revolutionary results in a wide swath of domains. A neural network, as a black box, is simply an approximation to some unknown function $\mathbf{y} = f(\mathbf{x})$. It parameterizes this function by layers of "neurons", with weights connecting each neuron in a layer to each neuron in the previous layer. Each neuron within a layer has a real-numbered output, or "activation", and each connection from a neuron in one layer to a neuron in the next layer has a real-numbered weight. Given the activations of a single layer, $\mathbf{a}_t$, the next layer's activations are computed by

$$\mathbf{a}_{t+1} = \sigma(\mathbf{W}_t \mathbf{a}_t + \mathbf{b}_t) \tag{2.1}$$

where $\mathbf{W}_t$ is the matrix of weights, $\mathbf{b}_t$ is a vector of constant "biases", and σ is some nonlinear "activation function" - common choices include the logistic function, the tanh function and the "rectified linear unit", or ReLU. The first "layer" of the network is said to be the input layer, and its activations are simply set to be the values of the input vector to the network. The final layer is the "output" layer, and its activations are taken to be the output vector of the function. See fig. 2.1 for an example.

Neural network models are commonly used to solve supervised learning problems - i.e. problems where the goal is to learn a function $\mathbf{y} = f(\mathbf{x})$ to approximate some given labelled dataset $(\mathbf{Y}, \mathbf{X})$ (often called the "training set"), and generalize the results to

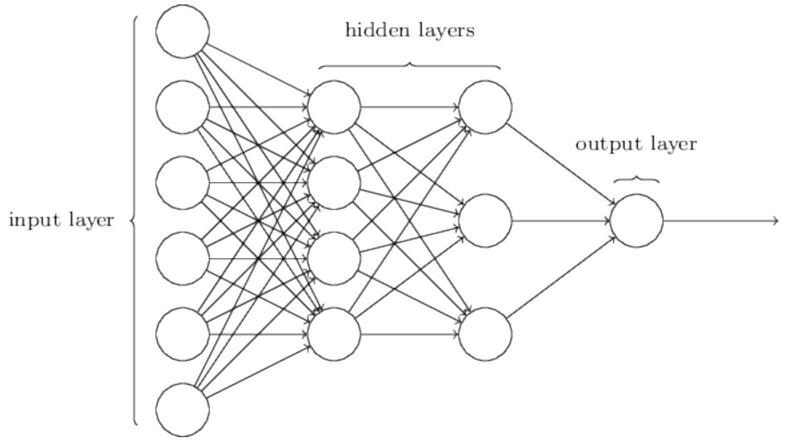


Figure 2.1: A simple feedforward neural network with two hidden layers.

unlabelled data (referred to as the "test set"). Often a third "validation" set is included for hyperparameter tuning and intermediate testing. The network is initialized randomly, then exposed to data from the training set. The network computes

$$\hat{\mathbf{y}} = f(\mathbf{x}) \tag{2.2}$$

and these predictions $\hat{\mathbf{y}}$, alongside the true labels $\mathbf{y}$, are fed to a "loss function". This "loss function" is a function $\ell(\hat{\mathbf{y}}, \mathbf{y})$ which computes a real-valued "loss" for each output (commonly used loss functions include squared error and cross-entropy loss). This is now formulated as an optimization problem, with the goal of finding the weights and biases for the network which minimize the loss summed over all training examples. The backpropagation algorithm is used to compute the gradients of the loss with respect to all weights and biases in the network. It does so by using the multivariate chain rule to compute the gradient at each step of the network, conditioned on the gradients at the next layer after it. This then "propagates" the gradients backward through the network, hence the name. Once these gradients are computed, a simple gradient descent pass is used to update the network's parameters - i.e.

$$\theta_{t+1} = \theta_t - \eta \frac{\partial \ell}{\partial \theta} \tag{2.3}$$

In general, rather than performing full gradient descent on the entire training set (since the training set can be quite large in principle), mini-batch Stochastic Gradient Descent is used. A fixed number of samples are randomly selected and assembled into a "mini-batch", then the gradients with respect to the loss calculated over only those samples are used to update the network's parameters. There is a great deal more that could be said about the architecture and design of such neural network models, but the only case that is relevant to this project is the case of recurrent neural networks.

Recurrent neural networks deviate from the vanilla architecture presented above in that they are specifically designed to process sequential data. To do this, the network's evaluation of a given element of the sequence must depend on the previous elements of the sequence. As such, the network stores a "hidden state" - a vector summarizing the current state of the computation - and updates this state as it passes through the sequence. Information about the previous elements can then be stored in this state vector, allowing the network to make decisions in a temporal fashion.

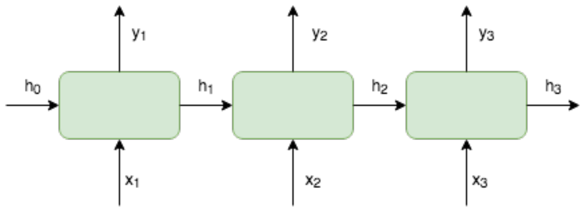


Figure 2.2: A recurrent neural network "unrolled" through time.

Recurrent neural networks are (in their simplest form) parameterized by three matrices: $\mathbf{W}_{xh}$, $\mathbf{W}_{hh}$, and $\mathbf{W}_{hy}$. They begin by receiving an initial state $\mathbf{h}_0$ and initial input $\mathbf{x}_1$, then predicting $\mathbf{h}_1$ and $\mathbf{y}_1$ from these. $\mathbf{h}_1$ is then passed to the next step of the RNN, along with the second input $\mathbf{x}_2$, and this repeats until the end of the sequence. At each step t, the RNN computes:

$$\mathbf{h}_t = f(\mathbf{W}_{xh}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1}) \tag{2.4}$$

$$\mathbf{y}_t = g(\mathbf{W}_{hy}\mathbf{h}_t) \tag{2.5}$$

Where f and g are generally some form of nonlinearity - commonly a tanh function. The three weight matrices $\mathbf{W}_{xh}$, $\mathbf{W}_{hh}$, and $\mathbf{W}_{hy}$ are trained via an algorithm called *backpropagation through time*, where the network is "unrolled" backwards through time (see fig. 2.2), and then trained via the standard NN backpropagation algorithm. Sometimes this unrolling is truncated early after a certain number of steps to maintain computational efficiency.

These "vanilla" RNNs are interesting and effective, but suffer from a number of problems. First, these RNNs often have difficulty maintaining causal relationships over long periods of time. For very long sequences, they can sometimes lose the ability to learn relationships between sequence entries. Second, they are prone to problematic issues with gradients - namely vanishing gradients and exploding gradients. These refer to scenarios where the gradients either converge rapidly to zero or diverge rapidly to infinity as the RNN is unrolled backward through time. These problems can be mitigated through the use of activation functions such as ReLU, gradient clipping (in the case of exploding gradients), or the use of more complicated update functions.

Long short-term memory units, (or LSTMs) were introduced to solve the second problem - and as a result of addressing the vanishing gradient issue, also significantly improve the ability of the network to learn long-term sequential relationships [7]. LSTMs introduce a more complicated update structure, and maintain two hidden states - the outward-facing hidden state at a given time step, as per usual (often denoted s or h), as well as an additional internal "memory" vector (often denoted c). The LSTM computation consists of applying a number of "gates" of the following form (note the similarity to the vanilla RNN update):

$$\mathbf{v}_t^{\phi} = g(\mathbf{U}^{\phi}\mathbf{x}_t + \mathbf{W}^{\phi}\mathbf{h}_{t-1} + \mathbf{b}^{\phi}) \tag{2.6}$$

Here g is again a nonlinearity such as a sigmoid or tanh, and ϕ is an indexing variable that runs over all the gates. In the LSTM formulation, there are three gates: i, o, and f, referred to as "input", "output", and "forget", as well as a fourth gate-like structure g. The output of a the "forget" gate is a vector of weights between 0 and 1 to be multiplied element-wise (this operation is denoted below by $\circ$) by the cell state c_{t-1}. This has the effect of "forgetting" information from the old cell state. g creates a candidate state vector through the same basic structure as the gate computation. The "input" gate then constructs a vector of weights similar to the "forget" gate which are multiplied by the candidate state vector. Intuitively, this corresponds to choosing which parts of the state we want to update. This vector is then added to the cell state c_{t-1} to produce c_t. Once c_t is computed, h_t is then simply computed via the same mechanism of using a gate (this time the "output" gate) to create a vector of weights, then pointwise multiplying the cell state. Thus, the final update is:

$$\mathbf{f}_t = \sigma(\mathbf{U}^f \mathbf{x}_t + \mathbf{W}^f \mathbf{h}_{t-1} + \mathbf{b}^f) \tag{2.7}$$
$$\mathbf{i}_t = \sigma(\mathbf{U}^i \mathbf{x}_t + \mathbf{W}^i \mathbf{h}_{t-1} + \mathbf{b}^i) \tag{2.8}$$
$$\mathbf{g}_t = \tanh(\mathbf{U}^g \mathbf{x}_t + \mathbf{W}^g \mathbf{h}_{t-1} + \mathbf{b}^g) \tag{2.9}$$
$$\mathbf{c}_t = \mathbf{c}_{t-1} \circ \mathbf{f}_t + \mathbf{g}_t \circ \mathbf{i}_t \tag{2.10}$$
$$\mathbf{o}_t = f(\mathbf{U}^o \mathbf{x}_t + \mathbf{W}^o \mathbf{h}_{t-1} + \mathbf{b}^o) \tag{2.11}$$
$$\mathbf{h}_t = \tanh(\mathbf{c}_t) \circ \mathbf{o}_t \tag{2.12}$$

Note that the activations for the three gates are generally sigmoids, whereas the activations for the state vectors are always tanh units.

A "Gated Recurrent Unit", or GRU, is similar in concept to an LSTM, but uses a slightly simplified architecture. It computes only two intermediate gates, and does not store the cell state in memory. The GRU update is as follows:

$$\mathbf{z}_t = \sigma(\mathbf{U}^z \mathbf{x}_t + \mathbf{W}^z \mathbf{h}_{t-1} + \mathbf{b}^z) \tag{2.13}$$
$$\mathbf{r}_t = \sigma(\mathbf{U}^r \mathbf{x}_t + \mathbf{W}^r \mathbf{h}_{t-1} + \mathbf{b}^r) \tag{2.14}$$
$$\mathbf{g}_t = \tanh(\mathbf{U}^g \mathbf{x}_t + \mathbf{W}^g (\mathbf{h}_{t-1} \circ \mathbf{r}_t) + \mathbf{b}^g) \tag{2.15}$$
$$\mathbf{h}_t = (1 - \mathbf{z}_t) \circ \mathbf{g}_t + \mathbf{z}_t \circ \mathbf{h}_{t-1} \tag{2.16}$$

The GRU was introduced by Cho et al in 2014 [3], and has since become the main competitor to the LSTM unit. Notably, the paper which introduced the GRU was also one of the primary works leading up to the invention of the sequence to sequence architecture, and first popularized the RNN encoder-decoder structure.

2.2 Sequence to Sequence

Sequence to sequence models were pioneered by Ilya Sutskever in 2014 [15] and have since become the dominant architecture for neural machine translation and neural conversational models. These models consist of two recurrent neural networks: an "encoder" and a "decoder". The input sequence is fed into the encoder RNN, which learns to "encode" a fixed-size vector representation of the input in the form of its final hidden state. This hidden state then becomes the initial hidden state for the decoder RNN. The decoder's initial input is simply a predefined token referred to as the "start of sequence" token, or SOS. Using the information from the encoder, the decoder then produces a probability distribution over all the words in its vocabulary. The top word is then selected as the output, and this output becomes the input to the next iteration of the decoder. This process continues until the decoder outputs the "End of Sequence" token, or EOS, or until a fixed maximum length is reached.

Sequence to sequence (or Seq2Seq) models are appealing for multiple reasons. The initial motivation for sequence to sequence models was that they avert the need for a fixed-length RNN input and output. A sequence of any length can be mapped to a fixed-size encoding by the encoder, and the decoder simply starts with SOS and continues until it detects EOS. This averts the need to "pad" inputs and outputs to bring them all to equal fixed length. While this did serve as one of the initial impetuses for the Seq2Seq architecture, it is not one of the major draws in practice, as using a minibatch training approach requires padding and masking anyways. Once these models were implemented, however, it was found that they performed exceptionally well for Neural Machine Translation tasks, and exceeded the state of the art for generative conversational agents as well.

2.3 Attention

A major advance in the use of sequence to sequence models was the idea of "attention", introduced first by Bahdanau in 2014 [1] and then refined by Luong [9] in 2015. The motivation behind the attention model is to allow the network to "attend" to particular sections of the input sequence more than others. For example, consider the task of translation. In translating "I love my cat" to "J'aime mon chat", the words "I love" and "J'aime", "my" and "mon", and "cat" and "chat" are direct translations of each other; it would be helpful, for example, for the network to learn to pay attention primarily to "cat" when producing the last word in the sequence. More generally though, the concepts behind attention can be used to compute alignments between any candidate object and a set of reference objects.

Viewed in this general fashion, the attention mechanism is essentially a mapping from an input vector $\mathbf{x}$ and a set of reference vectors $\mathbf{Q} = \{\mathbf{q}_i\}$ to a context vector $\mathbf{c}$. The mapping is computed as follows:

1. A scoring function $s(\mathbf{x}, \mathbf{y})$ computes a vector of attention "energies" $\mathbf{e}$, measuring the similarity between $\mathbf{x}$ and each vector $\mathbf{q}_i$:

$$e_i = s(\mathbf{x}, \mathbf{q}_i) \tag{2.17}$$

Common scoring functions used include [9]:

$$s(x, y) = \mathbf{y}^T\mathbf{x} \tag{dot}$$

$$s(x, y) = \mathbf{y}^T\mathbf{A}\mathbf{x} \tag{general}$$

$$s(x, y) = \mathbf{v}^T \tanh(\mathbf{A}_1\mathbf{x} + \mathbf{A}_2\mathbf{y}) \tag{concat}$$

2. These "energies" are then normalized to produce the attention weights $\mathbf{w}$:

$$w_i = \frac{\exp(e_i)}{\sum_j \exp(e_j)} \tag{2.18}$$

3. From the weights, the context vector c is then constructed as a weighted sum:

$$\mathbf{c} = \sum_i w_i\mathbf{q}_i = \mathbf{Q}\mathbf{w} \tag{2.19}$$

In a sequence-to-sequence model, as mentioned previously, the attention mechanism is generally introduced to allow the decoder to focus on certain parts of the input more than others. Thus, the encoded representations of each token in the input (i.e. the encoder hidden states after the RNN acts on that token) are used as the reference vectors, and the decoder's hidden state is used as the input vector. In the Bahdanau attention model, the input hidden state to the decoder at the current stage of the computation is used as the candidate, and the context vector is simply concatenated with the input to the decoder. In the Luong attention model, the output hidden state of the decoder is used as the candidate vector, and the context vector is concatenated with the decoder RNN's output, then passed together into further linear layers, eventually leading to the output.

Chapter 3

Related Work

Sequence to sequence models (and variations thereof) have been studied extensively in recent years, and many different attempts have been made to incorporate "context" or "history" into these models. One approach to incorporating history into a generative language model was pioneered by Sordoni et al in 2015[13]. They used a hierarchical encoder-decoder structure with two encoders: one to compute the phrase representations at the level of individual tokens in a sentence (as is standard for a sequence to sequence model), and another which acts over a sequence of phrase representations (encoded by the first encoder) over a corpus. The decoder is then conditioned on a history consisting of multiple phrases, rather than just the previous phrase.

Another class of approaches has been to use memory networks, first suggested by Weston, Chopra and Bordes [16]. This framework introduces a memory $\mathbf{M} = \{\mathbf{m}_i\}$, along with feature maps $\{I, G, O, R\}$. Given an input $\mathbf{x}$, the model acts as follows:

1. Convert $\mathbf{x}$ to internal feature representation $I(\mathbf{x})$
2. Update the memory based on $I(\mathbf{x})$:
$$\mathbf{m}_i' = G(\mathbf{m}_i, I(\mathbf{x}), \mathbf{M}) \tag{3.1}$$
3. Compute output features based on $\mathbf{x}$ and the updated memory:
$$\mathbf{o} = O(I(\mathbf{x}), \mathbf{M}') \tag{3.2}$$
4. Decode output features to produce the response:
$$\mathbf{r} = R(\mathbf{o}) \tag{3.3}$$

The authors give a sample framework for text that they call "MemNN". This network uses very simple input and generalization maps: I reads in a textual input and G simply stores the input in the first available slot - it does not update old memories based on new information. O then finds the top-k most relevant memories using a scoring function, and R generates a response based on these top-k memories. The R function here employs some sort of argmax over a scoring function that scores the alignment between the output word and the input and memories. The training is performed with supervision at every step - i.e. there is a known correct choice for each max function for every training iteration.

This model was later extended by Sukhbaatar et al [14] to a fully end-to-end approach, without relying on supervision during intermediate training steps. They propose an architecture in which memory inputs v_i are stored as embedded vectors $\mathbf{v}_i$ of dimension d (using some sort of embedding map - in the simplest case a matrix of dimension $d \times V$, where V is the vocabulary size). The query x is similarly embedded as some vector $\mathbf{x}$ of dimension d; the query can either use the same embedding map as the memory or a distinct map, but they should have the same dimension. Alignments are then computed between the query vector $\mathbf{x}$ and the memory vectors $\mathbf{v}_i$ through a dot product, and normalized to produce weights w_i through a softmax.

$$\mathbf{w} = \text{softmax}(\mathbf{V}^T \mathbf{x}) \tag{3.4}$$

The memory's output vector is then computed using embedded output vectors $\mathbf{c}_i$ corresponding to each memory $\mathbf{v}_i$:

$$\mathbf{o} = \sum_i w_i \mathbf{c}_i \tag{3.5}$$

The embedded input state $\mathbf{x}$ and the memory output $\mathbf{o}$ can then be summed and passed into some further model to produce an output. This approach is fully differentiable, and can thus be trained end-to-end with backpropagation without requiring intermediate supervision. See fig. 3.1 for details.

Ghazvininejad et al then use this model as the basis for their knowledge-based conversation model [5]. They adapt this memory architecture to use a sequence to sequence structure with a recurrent decoder, along with a memory of "facts" which are encoded through a bag-of-words model into continuous embedded representations. They then use the model in Sukhbaatar et al to compute a memory context vector, which is passed into the RNN decoder to generate output tokens. See fig. 3.2 for details.

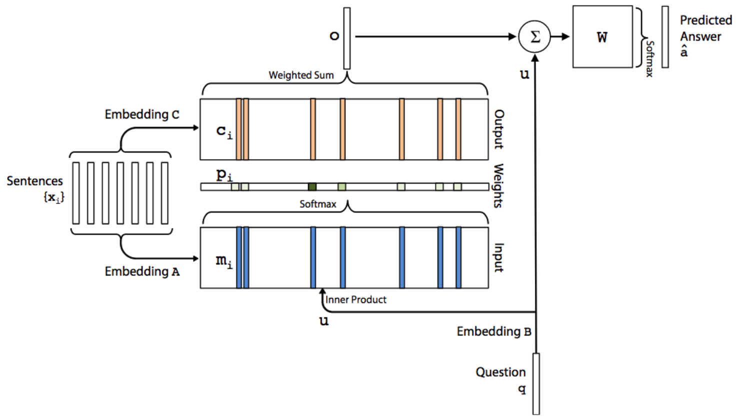
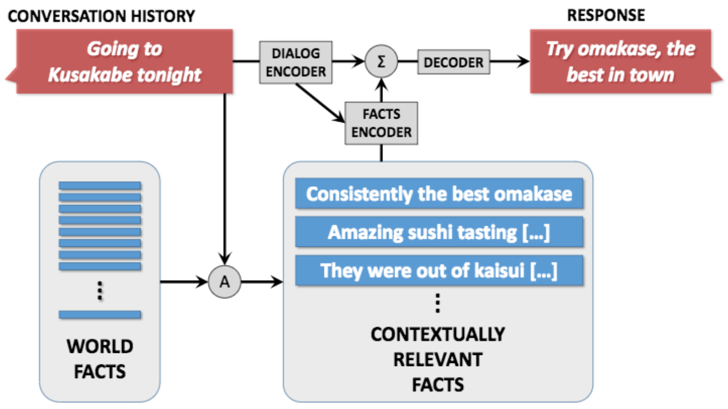


Figure 3.1: The model from Sukhbaatar et al.



CONVERSATION HISTORY
Going to Kusakabe tonight
DIALOG ENCODER
Σ
DECODER
RESPONSE
Try omakase, the best in town
FACTS ENCODER
Consistently the best omakase
Amazing sushi tasting [...]
They were out of kaisui [...]
CONTEXTUALLY RELEVANT FACTS
A
WORLD FACTS

Figure 3.2: The model from Ghazvininejad et al.

Chapter 4

Methodology

4.1 A Scalable Memory Architecture

4.1.1 Motivation

We use a model very similar to the memory network suggested by Ghazvininejad et al [5], save for certain key modifications. Our memory is defined over previous conversation entries, using the same encoder for the memory as for the input sequence. Our goal is to train this model in a way that is both effective and scalable, even when the memory is potentially very large. As such, it is not feasible to train the model while fully back-propagating over the encodings of all entries in the memory pad each step, as this requires $O(n^2)$ passes through the encoder (both forward and backward) for a conversation with n entries. Instead, the memory vectors are encoded using a fixed encoder which is simply updated using the current version of the trained encoder at the beginning of each epoch (and update the encoded memories accordingly). These embedded representations are pre-computed every epoch of training by passing the raw messages and responses through the same encoder that is used to encode the inputs. The input encoder already learns to map the input sequence to an embedding space that encodes the semantics of the sentence. As a result, the memory alignments using these encodings should still be sensible, even without explicitly training the encoder for this task.

This simplification saves tremendously on computation time compared to computing the embeddings anew with each training pass, as we do not need to re-compute the encoded memory vectors for each new conversation entry. This also means that gradients are not propagated from the memory pad to the encoder. With large memory sizes ($\sim$ 10,000 - 100,000), such a training regime could become prohibitively expensive in both memory and computation time (see 5.2 for details).

4.1.2 Architecture

We begin with a naive sequence to sequence model, using one-layer GRUs for both the encoder and decoder, with a bidirectional encoder and a uni-directional decoder. Experiments were performed using each of:

(a) a "Luong-style" attention mechanism over the input tokens [9]

(b) a "Bahdanau-style" attention mechanism over the input tokens [1]

(c) no base attention mechanism over the input tokens

We use a vocabulary size of approximately 25,000, with hidden size 1024. This will be referred to henceforth as the "base model", or "naive model".

We then augment this model with a simplified form of the memory network architecture proposed in Sukhbaatar et al [14]. Given a conversation history of pairs $\{m_i, r_i\}$, where m_i and r_i are the i-th message and response respectively, the model creates a memory pad containing the embedded representations $\mathbf{m}_i$ and $\mathbf{r}_i$.

During the forward pass of the model, the final hidden state of the encoder, $\mathbf{h}$, is passed to the memory module, which computes the attention weights

$$\mathbf{w} = \text{softmax}(\mathbf{M}^T \mathbf{A} \mathbf{h}) \tag{4.1}$$

where M is the matrix of embedded memory vectors, and A is a learned transformation. The matrix '$\mathbf{A}$' can be included or omitted. The models where $\mathbf{A}$ is omitted are referred to as using 'dot' attention over the memory (i.e. the dot product directly between the memory vectors and the hidden state), and the models where $\mathbf{A}$ is included as using 'general' attention (i.e. including a generalized transform). From these calculated weights, a context vector is computed through a weighted sum of the encoded response vectors:

$$\mathbf{c} = \sum_i w_i \mathbf{r}_i = \mathbf{R}\mathbf{w} \tag{4.2}$$

In the case of Bahdanau attention or no attention, the memory context vector is concatenated to the input of the decoder at each step. In the case of Luong attention, the memory context vector is concatenated to the output of the decoder, then passed into further linear layers at the end of the network. From there, the remainder of the decoding proceeds in the same way as the naive case.

This computation is linear in memory size for the forward pass, and does not have any dependence at all on memory size during the backward pass of training. Since the encoder is updated only at the end of each completed training conversation, the memory pairs can be stored in encoded form, and thus the encodings for a given pair need only be computed once for the entire conversation. As a result, the model requires only $O(n)$ encoder evaluations per N conversation entries, rather than $O(n^2)$.

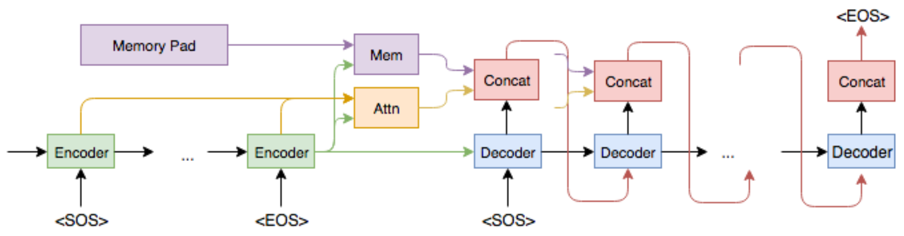


Figure 4.1: The architecture of the model.

4.2 Incorporating Confidence Scores

This mechanism allows alignments to be computed between the current input phrase and previous inputs in memory, and uses these alignments to construct a "context vector" summarizing the relevant information in memory. One difficulty with this procedure, however, is that it does not permit the model to express a "confidence" in the alignments it computes. Consider the following two hypothetical scenarios. In the first, the memory computes a very high degree of alignment between the current input message and a message in memory. In the second case, the memory finds no particularly strong alignments. Ideally, the model should attend closely to the high-scoring message in the first case, and base its output heavily on the corresponding response to that message. With the given model, however, that may not be the case, as there is no mechanism for the model to convey a "confidence" in the relevance of the context vector. With that in mind, we propose a second variant of the above algorithm.

In addition to computing a context vector, the model will also compute a scalar 'confidence' value according to the following algorithm:

$$\mathbf{c}_m = \sum_i w_i \mathbf{m}_i = \mathbf{M}\mathbf{w}$$
$$\alpha = \sigma(\mathbf{c}_m^T \mathbf{B}\mathbf{h})$$

This computes a message context $\mathbf{c}_m$ with the calculated weights and the encoded messages in the same way the final context vector is computed using the encoded responses. This "message context" is then multiplied by the input hidden state (along with a learned transformation $\mathbf{B}$), then passed into a sigmoid to compute the confidence on the interval $[0, 1]$. Once the confidence is computed, the confidence value is used to modify how the memory context is integrated into the original model. No matter which attention model is under consideration, the integration of the memory context into the base model is performed through a concatenation and a following linear layer, like so:

$$\mathbf{v}_{\text{out}} = \begin{bmatrix} \mathbf{L}_{\text{naive}} & \mathbf{L}_{\text{mem}} \end{bmatrix} \begin{bmatrix} \mathbf{v}_{\text{naive}} \\ \mathbf{v}_{\text{mem}} \end{bmatrix} \tag{4.3}$$
$$= \mathbf{L}_{\text{naive}} \mathbf{v}_{\text{naive}} + \mathbf{L}_{\text{mem}} \mathbf{v}_{\text{mem}} \tag{4.4}$$

To introduce the confidence α, this is simply modified to become:

$$\mathbf{v}_{\text{out}} = (1 - \alpha) \cdot \mathbf{L}_{\text{naive}} \mathbf{v}_{\text{naive}} + \alpha \cdot \mathbf{L}_{\text{mem}} \mathbf{v}_{\text{mem}} \tag{4.5}$$

Chapter 5

Experiments

5.1 Experimental Design

All models were trained and evaluated using the PyTorch framework (version 0.4.0), with training examples drawn from the Cornell Movie Dialogue Corpus [4]. This corpus is a set of lines of dialogue drawn from transcriptions of 617 movies (see addendum at the end of this report for some important notes about this dataset). Each movie was sorted into tuples of (message, response), then organized into training and validation sets and fed into the model. 10% (or 61) movies were withheld as validation data. For the "naive" model, training was performed upon randomly selected batches of lines drawn from anywhere in the data-set. The memory model required a more sophisticated training setup. Given pair number N from movie M, training (and evaluation) was performed with pairs 0-(N-1) from movie M in the memory bank, and no pairs from any other movie. The model was thus able to see any previous dialogue from the same movie in its memory, but no pairs from further on in the movie than the current line of dialogue, and no pairs from any other movie, thus simulating real conversation. The memory model's encoder was initialized using the encoder from the trained naive model, so as to give the memory a headstart in learning meaningful representations to select good memory alignments.

5.2 Scalability

The most critical test for the model is whether it indeed scales better with a large memory pad than previous proposed architectures. To demonstrate this, consider the toy task of training for a single epoch on a dataset containing the first n pairs from one movie. The model presented in this thesis will be referred to as the "simplified" model, whereas the model which propagates gradients through the memory pad to the encoder will be referred to as the "full" model. Experiments were run using a batch size of 25. The time to train all batches was recorded, as was the maximum memory used (excluding the memory used to store the model itself or the training data). As shown in Table 5.1, the "simplified" model requires a relatively flat amount of memory, with the bulk of the memory taken up by the constant amount necessary to store gradients for the model parameters. The "full" model on the other hand, requires a large amount of memory to store the gradients with respect to the memory entries - and this scales rapidly with the size of the memory pad. Similarly, the computation time for the "full" model is similar to the "simplified" model for a small memory pad, but grows far more steeply as the memory size increases.

	Simplified		Full	
n	Time (s)	Memory (MB)	Time (s)	Memory (MB)
100	1.293	1530	1.488	1516
200	2.427	1554	2.947	1860
400	5.156	1562	6.777	1880
800	10.477	1566	16.536	2535

Table 5.1: Time and memory usage for each of the "Simplified" and "Full" models on a toy task with n training pairs

5.3 Evaluation Metrics

5.3.1 BLEU

The primary quantitive metrics used to assess the model's results were BLEU scores [11]. BLEU scores are calculated using a weighted geometric average of modified n-gram precisions. The modified n-gram precisions are calculated as the number of n-grams in the hypothesis which appear in the reference, clipped to the maximum number of occurrences in the reference - i.e. if $N_n = \{g\}$ is the set of all *unique* n-grams in the hypothesis, and M is the *total* number of n-grams in the hypothesis then

$$p_n = \frac{1}{M} \sum_{g \in N_n} \text{Count}(g) \tag{5.1}$$

where Count(g) refers to the number of times n-gram g occurs in the reference. More sophisticated definitions exist for the case of multiple candidates and multiple references, but they are not needed in our case. Given a set of weights $\mathbf{w}$, the BLEU score is computed as:

$$\text{BLEU} = BP \cdot \exp\left(\sum_{w_n \in \mathbf{w}} w_n \log p_n \right) \tag{5.2}$$

where BP is a brevity penalty which serves to penalize overly short sentences. The standard N-th order BLEU scores use weights $w_n = \frac{1}{N}\ \forall n \in \{1...N\}$. We use BLEU-2 scores for our model, as they show the highest correlation with human evaluations of the BLEU-N metrics [8]. BLEU scores were originally developed for Neural Machine Translation, and have often been criticized as a metric both in that domain and in the domain of dialogue agents [2] [8]. Unfortunately, as even many of these papers admit [8], few if any other good quantitative metrics exist for this domain, and BLEU scores are widely used in the literature.

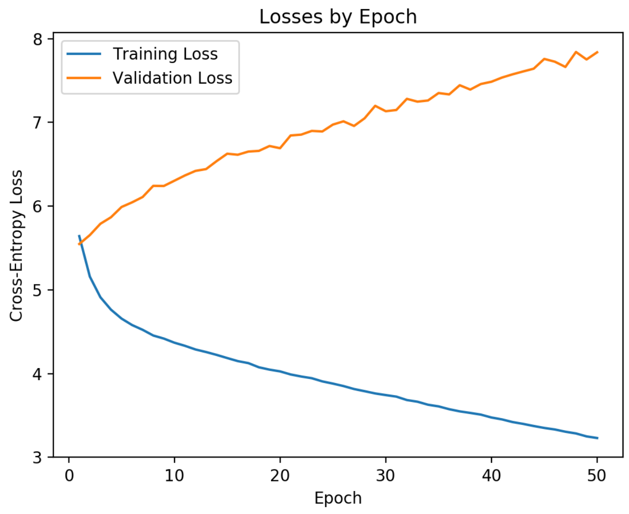
5.3.2 Word2Vec Embedding Similarity

The second quantitative metric we used to assess the model was a cosine similarity metric using word embeddings from Google's pretrained Word2Vec model [10]. The embedding vector for each word in the output sentence is computed using the Word2Vec model, then the vectors for each word in the sentence are summed to obtain the embedding for the sentence as a whole. The same procedure is performed to obtain the embedding for the ground truth output, and the score is computed by the cosine similarity between the two embedding vectors.

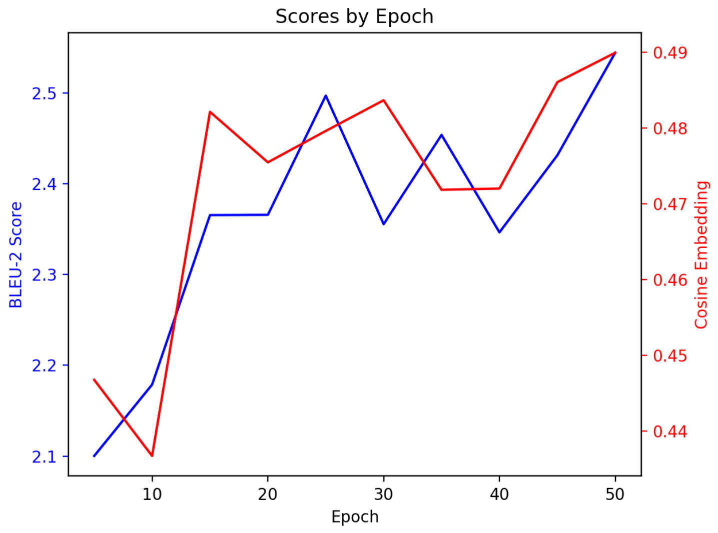
5.3.3 Perplexity/Validation Loss

Another common metric used in language modelling is the *perplexity* - the model's estimated probability of producing the ground truth output. Using a cross-entropy loss, this is effectively equal to the exponential of the validation loss. We omit this measure, as it is not a particularly strong measure of the model's success; during training, the model's validation loss often rises significantly (from approx. 5 to approx. 8 or sometimes higher) without becoming noticeably worse in output quality or BLEU score. In fact, the model often learns to perform better across other metrics during periods where the validation loss rises. See fig. 5.1, for instance, in which the BLEU-2 score and embedding score rise during training while validation loss increases.

Figure 5.1: Losses and evaluation metrics as a function of epoch for the base memory model with 'dot' attention. Evaluation metric scores were averaged over all validation set movies using model checkpoints saved every 5 epochs.



(a) Training and validation losses



(b) BLEU-2 and Word2Vec embedding scores

5.4 Results

5.4.1 Scores

When training the models, checkpoints were saved every 5 epochs, and the models were evaluated using the above metrics averaged over all movies from the validation set. The memory models were initialized using the weights from the encoder of the top naive model.

	BLEU-2			W2V		
Epoch	Luong	Bahdanau	No-Attn	Luong	Bahdanau	No-Attn
5	0.428	0.457	0.469	0.420	0.383	0.400
10	0.524	0.521	0.666	0.390	0.379	0.397
15	0.733	0.571	0.963	0.409	0.395	0.420
20	1.09	0.689	1.74	0.425	0.400	0.451
25	1.60	1.24	1.99	0.446	0.420	0.447
30	1.60	1.85	2.68	0.438	0.436	0.460
35	2.04	1.95	2.39	0.461	0.432	0.454
40	2.21	2.46	2.55	0.464	0.454	0.452
45	2.17	2.57	**2.74**	0.459	0.455	**0.476**
50	2.35	**2.85**	2.63	0.474	0.475	0.464
55	**2.54**	2.76	2.61	**0.479**	**0.479**	0.468
60	2.48	2.51	2.49	**0.479**	0.473	0.462

Table 5.2: BLEU-2 and W2V embedding scores for the naive models by training epoch. Top results are bolded.

	BLEU-2				W2V			
Epoch	Luong	Bahdanau	No-Attn	Confidence	Luong	Bahdanau	No-Attn	Confidence
5	2.20	1.90	1.23	2.25	0.453	0.443	0.434	0.419
10	1.89	2.03	1.74	2.33	0.417	0.435	0.442	0.458
15	2.32	2.22	1.69	2.35	0.456	0.449	0.406	0.483
20	2.45	2.47	**2.54**	2.33	0.483	0.466	**0.498**	0.468
25	2.41	2.58	1.97	2.35	0.480	0.473	0.441	0.486
30	**2.53**	2.58	2.48	**2.71**	0.479	0.467	0.444	**0.503**
35	2.28	**2.61**	2.14	2.33	0.489	**0.485**	0.446	0.483
40	2.34	2.43	2.48	2.28	0.479	0.478	0.465	0.481
45	2.48	2.41	2.38	2.40	**0.494**	0.472	0.474	0.488
50	2.37	2.60	2.42	2.40	0.489	0.479	0.457	0.483

Table 5.3: BLEU-2 and W2V embedding scores for the memory models using 'general' attention by training epoch. Top results are bolded.

	BLEU-2				W2V			
Epoch	Luong	Bahdanau	No-Attn	Confidence	Luong	Bahdanau	No-Attn	Confidence
5	2.10	1.86	1.76	1.72	0.447	0.401	0.426	0.441
10	2.18	**2.75**	2.05	2.13	0.437	0.473	0.443	0.453
15	2.37	2.45	2.02	2.23	0.482	0.462	0.390	0.469
20	2.37	2.44	2.38	2.18	0.475	0.447	**0.475**	0.436
25	2.50	2.54	2.50	2.21	0.480	0.461	0.442	0.444
30	2.36	2.40	2.57	2.15	0.483	0.469	0.452	0.455
35	2.45	2.46	2.42	2.32	0.471	0.474	0.468	0.458
40	2.35	2.48	2.68	2.42	0.472	0.478	0.468	0.474
45	2.43	2.45	**2.82**	2.31	0.486	**0.492**	0.445	0.466
50	**2.54**	2.31	2.58	**2.47**	**0.490**	0.477	0.461	**0.475**

Table 5.4: BLEU-2 and W2V embedding scores for the memory models using 'dot' attention by training epoch. Top results are bolded.

5.4.2 Scores by Memory Size

Given that the memory models can utilize the conversation history in constructing responses, we would expect that these models would perform better as the number of entries in memory increases. See plots in figs. 5.2 and 5.3 for average BLEU-2 and Word2Vec embedding similarity scores over the validation set for one of the top-scoring models as a function of number of entries in memory.

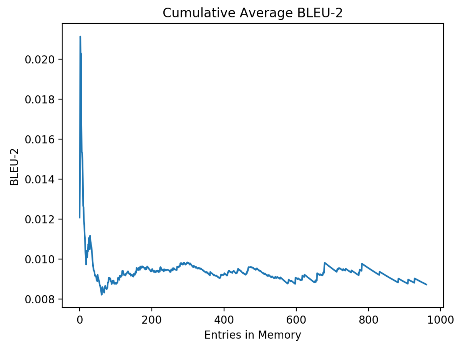


Figure 5.2: Validation BLEU-2 scores for the base memory model with 'dot' attention as a function of the number of entries in memory. Scores were computed separately for each movie in the dataset, then summed together and averaged.

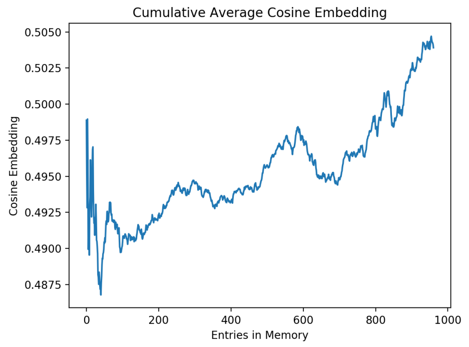


Figure 5.3: Validation Word2Vec embedding similarity scores for the base memory model with 'dot' attention as a function of the number of entries in memory. Scores were computed separately for each movie in the dataset, then summed together and averaged.

5.4.3 Memory Alignments

This section shows a selection of example alignments computed by the model between sample queries and phrases in memory. In each case, the alignments were computed by adding all lines from the first 20 movies in the validation set to the memory pad, then taking the memory entries with the 5 highest weights. The tables show the user’s message and model’s response, followed by the top 5 (message, response) tuples in the memory pad, shown with their corresponding weights.

General Model

Message	hello.
Response	[’hello’, ’.’]
Weight	Message
0.0309214	[’so’, ’there’, ’it’, ’is’, ’.’, ’you’, ’got’, ’that’, ’big’, ’itch’, ’<eos>’]
0.0170202	[’if’, ’stage’, ’two’, ’was’, ’completed’, ’it’, ’was’, ’underground’, ’she’, ’said’, ’<eos>’]
0.0168152	[’say’, ’baxter’, ’the’, ’way’, "you’re", ’<unk>’, ’that’, ’stuff’, ’you’, ’must’, ’<eos>’]
0.0145271	[’chin’, ’you’, ’have’, ’been’, ’ordered’, ’by’, ’the’, ’ministry’, ’of’, ’public’, ’<eos>’]
0.0141344	[’i’, ’had’, ’an’, ’unfortunate’, ’little’, ’run’, ’in’, ’with’, ’him’, ’today’, ’<eos>’]
Weight	Response
0.0309214	[’no’, ’i’, ’want’, ’my’, ’cut’, ’!’, ’<eos>’]
0.0170202	[’stage’, ’two’, ’of’, ’what’, ’?’, ’<eos>’]
0.0168152	[’oh’, "that’s", ’not’, ’me’, ’.’, "it’s", ’just’, ’that’, ’once’, ’in’, ’<eos>’]
0.0145271	[’this’, ’is’, ’still’, ’hong’, ’kong’, ’.’, ’<eos>’]
0.0141344	[’you’, ’better’, ’not’, ’mess’, ’with’, ’me’, ’rhodes’, ’.’, "i’d", ’love’, ’<eos>’]

Message	what’s your name?
Response	[’i’, ’.’]

Weight	Message
0.0282931	[’now’, "let’s", ’talk’, ’about’, ’you’, ’chaps’, ’.’, ’<eos>’]
0.0214106	[’i’, ’saw’, ’you’, ’go’, ’up’, ’from’, ’the’, ’<unk>’, ’working’, ’that’, ’<eos>’]
0.0177163	[’ronald’, ’here’, ’likes’, ’telephones’, ’.’, ’used’, ’to’, ’tape’, ’wooden’, ’matches’, ’<eos>’]
0.0147374	[’so’, ’much’, ’for’, ’honor’, ’among’, ’thieves’, ’.’, ’you’, ’would’, ’have’, ’<eos>’]
0.0127931	[’then’, ’there’, ’it’, ’is’, ’.’, ’ashby’, ’gets’, ’the’, ’itch’, ’.’, ’<eos>’]

Weight	Message
0.0282931	["we’d", ’rather’, ’stay’, ’.’, ’<eos>’]
0.0214106	["can’t", ’say’, ’.’, ’<eos>’]
0.0177163	[’did’, ’he’, ’tell’, ’you’, ’how’, ’we’, ’finally’, ’met’, ’?’, ’<eos>’]
0.0147374	[’i’, ’was’, ’doing’, ’it’, ’for’, ’them’, ’.’, ’<eos>’]
0.0127931	[’the’, ’standard’, ’ten’, ’.’, ’<eos>’]

’Dot’ Model

Message	hello.
Response	[’hi’, ’.’]
Weight	**Message**
0.312987	[’hello’, ’.’, ’<eos>’]
0.312987	[’hello’, ’.’, ’<eos>’]
0.312987	[’hello’, ’.’, ’<eos>’]
0.00864437	[’hello’, ’lloyd’, ’.’, ’<eos>’]
0.00569477	[’hello’, ’thomas’, ’.’, ’<eos>’]
Weight	**Response**
0.312987	[’hi’, ’.’, ’<eos>’]
0.312987	[’uh’, ’oh’, ’hi’, ’!’, ’my’, ’name’, ’is’, ’brad’, ’majors’, ’.’, ’<eos>’]
0.312987	[’how’, ’you’, "doin’", ’?’, ’<eos>’]
0.00864437	[’hello’, ’.’, ’heard’, ’about’, ’that’, ’graduation’, ’present’, ’.’, ’really’, ’quite’, ’<eos>’]
0.00569477	[’i’, ’just’, ’this’, ’minute’, ’got’, ’the’, ’baby’, ’to’, ’sleep’, ’.’, ’<eos>’]

Message	what’s your name?
Response	[’jacob’, ’.’]
Weight	**Message**
0.672749	["what’s", ’your’, "friend’s", ’name’, ’?’, ’<eos>’]
0.121251	[’name’, ’?’, ’<eos>’]
0.105954	[’tell’, ’me’, ’again’, "what’s", ’my’, ’name’, ’?’, ’<eos>’]
0.0286244	["what’s", ’her’, ’name’, ’?’, ’<eos>’]
0.0126811	[’this’, ’is’, ’travis’, ’and’, ’bob’, ’<elp>’, "what’s", ’your’, ’last’, ’name’, ’<eos>’]
Weight	**Response**
0.672749	[’baxter’, ’.’, ’<eos>’]
0.121251	[’gus’, ’gorman’, ’.’, ’<eos>’]
0.105954	[’dr’, ’.’, ’dreyfuss’, ’.’, ’<eos>’]
0.0286244	[’miss’, ’kubelik’, ’fran’, ’.’, ’<eos>’]
0.0126811	[’uh’, ’<elp>’, ’head’, ’?’, ’huh’, ’huh’, ’.’, ’my’, ’first’, "name’s", ’<eos>’]

5.4.4 Sample Conversations

This conversation was carried out with the base memory model using 'dot' attention. The memory was empty to begin, and each message-response pair in the conversation was added to the memory as the conversation unfolded. After each message save the first (since the memory is empty for the first response), the weights of the top 5 entries will again be listed.

Message	hello there.
Response	['where', 'are', 'you', '?']

Message	i am at home.
Response	['where', 'is', 'this', '?']
Weight	**Message**
1.0	['hello', 'there', '.', '<eos>']
Weight	**Response**
1.0	['<sos>', 'where', 'are', 'you', '?', '<eos>']

Message	my apartment.
Response	['i', 'told', 'you', 'i', 'had', 'to', '.']
Weight	**Message**
0.756646	['i', 'am', 'at', 'home', '.', '<eos>']
0.240464	['hello', 'there', '.', '<eos>']
0.00289034	['<sos>', 'where', 'are', 'you', '?', '<eos>']
Weight	**Response**
0.756646	['<sos>', 'where', 'is', 'this', '?', '<eos>']
0.240464	['<sos>', 'where', 'are', 'you', '?', '<eos>']
0.00289034	['i', 'am', 'at', 'home', '.', '<eos>']

Message	what is your name?
Response	['my', 'name', '.']
Weight	**Message**
0.886455	['<sos>', 'where', 'is', 'this', '?', '<eos>']
0.0902068	['hello', 'there', '.', '<eos>']
0.0165442	['<sos>', 'where', 'are', 'you', '?', '<eos>']
0.00545995	['my', 'apartment', '.', '<eos>']
0.00133363	['i', 'am', 'at', 'home', '.', '<eos>']
Weight	**Response**
0.886455	['my', 'apartment', '.', '<eos>']
0.0902068	['<sos>', 'where', 'are', 'you', '?', '<eos>']
0.0165442	['i', 'am', 'at', 'home', '.', '<eos>']
0.00545995	['<sos>', 'i', 'told', 'you', 'i', 'had', 'to', '.', '<eos>']
0.00133363	['<sos>', 'where', 'is', 'this', '?', '<eos>']

Message	my name is kira
Response	['what', '?', '.']
Weight	**Message**
0.345566	['what', 'is', 'your', 'name', '?', '<eos>']
0.273278	['<sos>', 'where', 'is', 'this', '?', '<eos>']
0.163733	['my', 'apartment', '.', '<eos>']
0.0940137	['i', 'am', 'at', 'home', '.', '<eos>']
0.0865199	['hello', 'there', '.', '<eos>']
Weight	**Response**
0.345566	['<sos>', 'my', 'name', '.', '<eos>']
0.273278	['my', 'apartment', '.', '<eos>']
0.163733	['<sos>', 'i', 'told', 'you', 'i', 'had', 'to', '.', '<eos>']
0.0940137	['<sos>', 'where', 'is', 'this', '?', '<eos>']
0.0865199	['<sos>', 'where', 'are', 'you', '?', '<eos>']

Message	what is my name?
Response	['my', 'name', 'is', '.', '.']
Weight	**Message**
0.997059	['what', 'is', 'your', 'name', '?', '<eos>']
0.00202852	['<sos>', 'my', 'name', '.', '<eos>']
0.00040756	['<sos>', 'where', 'is', 'this', '?', '<eos>']
0.000350248	['my', 'name', 'is', '<unk>', '<eos>']
0.000107462	['my', 'apartment', '.', '<eos>']
Weight	**Response**
0.997059	['<sos>', 'my', 'name', '.', '<eos>']
0.00202852	['my', 'name', 'is', '<unk>', '<eos>']
0.00040756	['my', 'apartment', '.', '<eos>']
0.000350248	['<sos>', 'what', '?', '<eos>']
0.000107462	['<sos>', 'i', 'told', 'you', 'i', 'had', 'to', '.', '<eos>']

Message	where is this?
Response	['i', "don't", 'know', '.']
Weight	**Message**
0.999923	['<sos>', 'where', 'is', 'this', '?', '<eos>']
2.95377e-05	['what', 'is', 'your', 'name', '?', '<eos>']
2.06299e-05	['what', 'is', 'my', 'name', '?', '<eos>']
1.73101e-05	['<sos>', 'where', 'are', 'you', '?', '<eos>']
7.58477e-06	['<sos>', 'what', '?', '<eos>']
Weight	**Response**
0.999923	['my', 'apartment', '.', '<eos>']
2.95377e-05	['<sos>', 'my', 'name', '.', '<eos>']
2.06299e-05	['<sos>', 'my', 'name', 'is', '.', '.', '<eos>']
1.73101e-05	['i', 'am', 'at', 'home', '.', '<eos>']
7.58477e-06	['what', 'is', 'my', 'name', '?', '<eos>']

Chapter 6

Conclusion

The memory models performed poorly according to the BLEU-2 scores, but well according to the embedding cosine similarity scores. The BLEU scores also did not noticeably increase as the memory size did, but the embedding scores increased quite noticeably. Unfortunately, since metrics in the field of dialogue models are so poor compared to human assessment, it is difficult to say exactly what this means. It is possible that this simply means that the memory models are not a significant improvement over the naive model. It is also possible, however, that the lower BLEU score represents a low ability to exactly reproduce the (often questionable) responses from the script, and the higher embedding scores signify that the memory models instead reproduce phrases that are conceptually similar to the desired response, even if they are not necessarily using the exact same words.

The alignments computed by the 'general' attention models were not reasonable, but the 'dot' attention models provided much better alignments. The 'dot' model also provided generally good responses to queries, and did attend to the correct statements in short conversations - though it did not utilize them enough to create the best responses. When asked "where is this?", for instance, the model correctly attended to the pair ("where is this?", "my apartment.") but replied "i don't know" instead of "my apartment". This is still a perfectly valid response, but ideally the model should incorporate the contextual information from the previous pair in memory.

Many of the issues with the results may be caused by problems with the data. The Cornell Movie Dialogue Corpus is a relatively standard dataset for dialogue models, but it has a critical flaw: while training this model, it was discovered that the corpus contains duplicates of several movies. This creates the possibility for contamination between the training and validation sets, and led to massively inflated results until the issue was found and corrected. This also calls into question the results of all other models and papers trained on this corpus, as they will also benefit from this inflation. Even once this issue is corrected, however, there are difficulties associated with using this data. Since the Cornell dataset is composed of movie transcriptions, the dialogue is grounded in visual context that is not available to the model. As a result of this missing context, it is difficult to accurately predict the expected response to a query. This can easily be seen simply from the sample data above - many of the responses from the dataset are not intuitive or obvious, even to a human.

6.1 Future Work

In order to try to improve the model, there are a number of further extensions to consider. First, one mechanism that might improve the model's computed alignments would be the use of autoencoders. Since the functionality of an autoencoder is to learn a dense, information-rich encoding of a given input, encoded states computed using an autoencoder might produce more accurate alignments than the standard sequence to sequence encodings. In the simplest case, the model's encoder could simply be initialized using an autoencoder. The model could also make heavier use of autoencoders, however. If the model used an autoencoder to generate both the message and response encodings in memory, then it could use the autoencoder's decoder to essentially produce some sort of decoded combination of the aligned responses. This might be an interesting avenue of study to pursue with this framework.

Another approach to extending the model would be to try to solve the "out of vocabulary" (or OOV) issue. Currently, the model relies on a fixed dictionary, which cannot be expanded. As a result, it cannot handle simple examples such as retrieving a name from memory - as the name might not appear enough times in the training corpus to be worth including in the model's vocabulary. One solution to this issue would be to build on the "Copynet" proposal of Gu et al [6]. This paper uses intermediate encoded states to not only compute alignments, but also "copy" portions of the input sequence. This could be extremely useful in the context of memory, and could allow the model to avoid the OOV issue by copying relevant portions of memory phrases directly.

References

[1] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *ICLR*, 2015.

[2] Chris Callison-Burch, Miles Osborne, and Philipp Koehn. Re-evaluating the role of bleu in machine translation research. In *In EACL*, 2006.

[3] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *EMNLP*, 2015.

[4] Cristian Danescu-Niculescu-Mizil and Lillian Lee. Chameleons in imagined conversations: A new approach to understanding coordination of linguistic style in dialogs. In *Workshop on Cognitive Modeling and Computational Linguistics, ACL*, 2011.

[5] Marjan Ghazvininejad, Chris Brockett, Ming-Wei Chang, Bill Dolan, Jianfeng Gao, Wen-tau Yih, and Michel Galley. A knowledge-grounded neural conversation model. *CoRR*, abs/1702.01932, 2017.

[6] Jiatao Gu, Zhengdong Lu, Hang Li, and Victor O. K. Li. Incorporating copying mechanism in sequence-to-sequence learning. *ACL*, 2016.

[7] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, November 1997.

[8] Chia-Wei Liu, Ryan Lowe, Iulian V Serban, Michael Noseworthy, Laurent Charlin, and Joelle Pineau. How not to evaluate your dialogue system: An empirical study of unsupervised evaluation metrics for dialogue response generation. *EMNLP*, 2016.

[9] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. *EMNLP*, 2015.

[10] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *NIPS*, 2013.

[11] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: A method for automatic evaluation of machine translation. In *ACL*, 2002.

[12] Matt Post. A call for clarity in reporting BLEU scores. *CoRR*, abs/1804.08771, 2018.

[13] Alessandro Sordoni, Yoshua Bengio, Hossein Vahabi, Christina Lioma, Jakob Grue Simonsen, and Jian-Yun Nie. A hierarchical recurrent encoder-decoder for generative context-aware query suggestion. *CIKM*, 2015.

[14] Sainbayar Sukhbaatar, Arthur Szlam, Jason Weston, and Rob Fergus. End-to-end memory networks. *NIPS*, 2015.

[15] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *NIPS*, 2014.

[16] Jason Weston, Sumit Chopra, and Antoine Bordes. Memory networks. *ICLR*, 2015.