A Generative Chatbot with NLP

David Liebman david.c.liebman@gmail.com

January 18, 2020

Coordinator 1 Coordinator 2

Department of Computer Science

SUNY New Paltz

Abstract

We are interested in making a chat-bot. We want a computer program that can answer questions that might come up in a simple conversation.

We experiment with the Transformer Neural Network model and we try to explain in this paper how one works.

We chronicle a few experiments in Natural Language Processing. We try a GRU based chatbot. We try a transformer based chatbot. We also try a GPT2 based chatbot.

We are also interested in installing the chatbot code on a small computer like a Raspberry Pi with speech recognition and speech-to-text software. In this way we might be able to create a device which speaks and which you can speak to. For the GRU based model we can use a Raspberry Pi 3B. In the case of GPT2 the running chatbot model uses too much ram, so we may try to install it on a Raspberry Pi 4B. GPT2 does not fit on a Raspberry Pi 3B.

Contents

List of Figures

List of Tables

Chapter 1

Background/History of the Study

1.1 Background

In their paper Vinyals et al (2015)[1] discuss making a chatbot using a neural network configured for sequence to sequence neural machine translation. We code our own sequence to sequence chatbot, though our results are less than spectacular. As our hand coded model does not run sufficiently well, we use code authored by Matthew Inkawhich (2018)[2].

In their paper Vaswani et al (2017)[3] discuss using the Transformer architecture for solving machine learning tasks. We train a transformer model as a chatbot.

Also Radford et al (2019)[4] discuss the GPT2 neural network for NLP tasks. The GPT2 model is based largely on the Transformer architecture. GPT2 stands for 'Generative Pre-training Transformer 2'.

We implement a chatbot with a GPT2 model. We use a program library from Wolf et al (2019)[5] to run our model.

It is worth noting that with the appearance of the Transformer architecture and WordPiece vocabulary scheme, some technologies have become redundant or obsolete. This may be true of any model that uses RNN components and also the traditional word vector embeddings.

1.2 Recurrent Neural Network Components

Overview (RNN)

The goal behind RNN components is to detect patterns. Here we explore a simple RNN.

The simplest RNN has two inputs and two outputs. They can be arranged in chains. In our example the input will be a sequence of data and the RNN chain will be a line of components of the same length.

One input from each component is the hidden state output from the RNN to the left. Another input is the current input from the sequence that the chain is monitoring. One output is the generated hidden state, meant for the component to the right. The last output is the value that the RNN surmises.

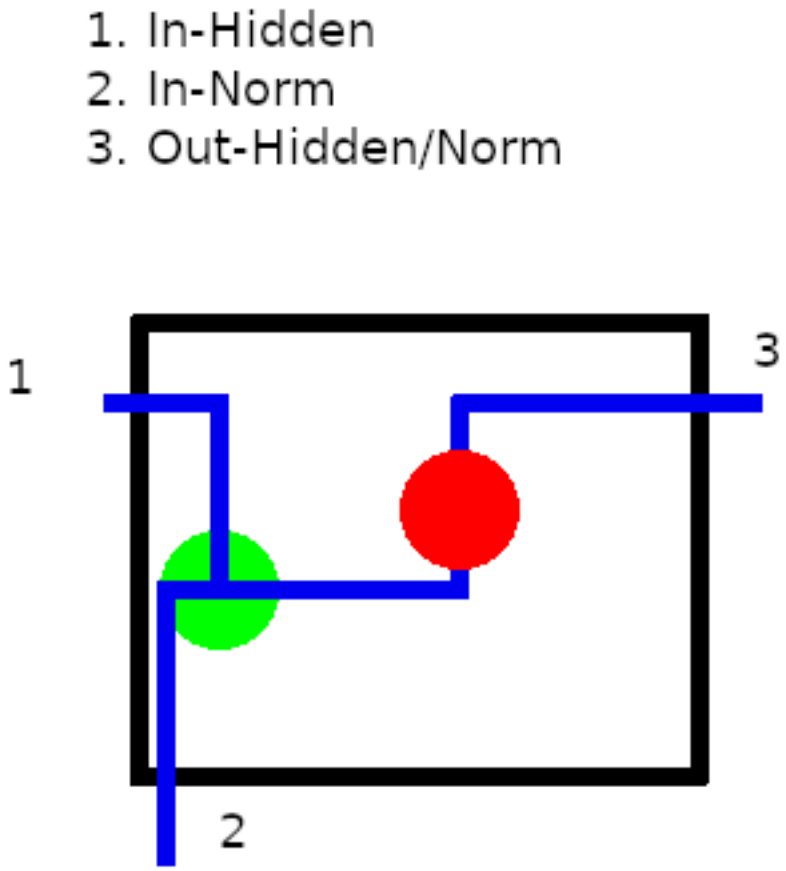

Figure 1.1: RNN - 1 and 2 are inputs, 3 is the output.

In our diagram the two inputs are labeled on the left side, and the single output does double duty as both the hidden state output and the value that the RNN surmises.

There are several designs for an RNN component. The inner workings of these components are what makes them different. In the example in the diagram the inner workings are very simple. Two paths, labeled as inputs, take data into the RNN. Their data is combined in the green circle. This combination is done with concatination and simple feed forward neural network components. The output from the concatination is passed through the red circle. This is a tanh activation operation that limits the output to values from -1 through 1. This tanh activation keeps the output within reasonable values. Finally the output is delivered outside the component to the program employing the RNN. In this diagram there is only one output. The single output would serve as both the

hidden state output for that position in the chain, and also the data output component for that position in the chain.

Specific (GRU)

A second type of RNN is the GRU. GRU stands for 'Gated Recurrent Unit'. A GRU has two inputs and two outputs. The formulas for a GRU, as outlined by Denny Britz in the website WILDML (Britz 2015)[6], are as follows.

$$z = \sigma(x_t U^z + s_{t-1} W^z)$$

$$r = \sigma(x_t U^r + s_{t-1} W^r)$$

$$h = tanh(x_t U^h + (s_{t-1} \circ r) W^h)$$

$$s_t = (1 - z) \circ h + z \circ s_{t-1}$$

The GRU has two inputs and two outputs. It also has two internal gates. One internal gate is the 'reset' gate. This one determines how much of the previous input is combined with the new value calculated by the mechanism of the GRU. It is denoted as 'r' above. Another internal gate is the 'update' gate. The update gate decides how much new information is to be included in gate computation. It is denoted as 'z'.

Here 's_t' is the symbol for the output. 'h' is the symbol for the 'hidden output'. The two inputs are 'x_t' and 's_{t-1}' . 'x_t' is the hidden state input. 's_{t-1}' is the regular input for the RNN or GRU. Sigmoid activation is used on the two gates, while tanh activation is used to compute the hidden output.

In the last line, the regular output is determined using the 'dot' product which is denoted with a circle, along with an addition operation. In the two gate formulas (the first and second) the output is determined as the sum of two matrix multiplication operations passed through sigmoid activation. This produces values in the range of 0 to 1.

Under most programming circumstances the GRU is not implemented by the average programmer. The programmer employs a language like python and a programming library like Pytorch or Tensorflow. The library then implements the GRU and makes it easy for the programmer to use that implementation.

1.3 Sequence to Sequence

Translating text from one language to another has become a common task for computers. The Sequence to Sequence architecture is often used today for this purpose.

A naive approach to translation involves using a dictionary. You would encode each key as a word from one language and the value for that key would be the translated word in the target language. Of course this doesn't work, because different languages not only have different words for the same thing, but they also have different sentence structures for what might be similar concepts.

A better approach is sequence to sequence translation. A description follows with a section at the end for how training works.

In this approach we use recurrent neural networks to obtain our translation. Two recurrent neural network components are employed. One is responsible for the input language and the other for the output. Recurrent elements can remember sequences.

Also employed are two vocabulary sets. One vocabulary is for the source language and another is for the target language. A table of word vectors the size of the input vocabulary is created and a maximum sentence length is picked. There is also a 'hidden size', which is an important dimension in the model. In practice the hidden size could be 300 units and more for this kind of application.

First text is prepared for training. A text corpus with source and target pairs is chosen. Sentences in the source corpus are paired with sentences with the same meaning in the target corpus. Sentence length is observed and for all sentences shorter than that length a special 'end-of-sequence' token is appended to all sequences in both languages.

Word vectors are created composed of floating point numbers. Each word in the vocabulary is translated to an integer which remains the same throughout the use of the translator. The word vectors are arranged in a table of floating point numbers with one dimension being the size of the input vocabulary and one dimension being the hidden size for the vector.

Recurrent Elements

The recurrent unit in this case is a GRU. This stands for Gated Recurrent Unit. The GRU takes as input a single vector. It processes the vector and returns another vector. This could be exactly the same as the input but is usually somehow changed. The input vector and the output vector have the dimension of the 'hidden size' mentioned above. Throughout the discussion of this model the

hidden size will remain the same. The GRU also operates on two hidden states. One hidden state, a vector, is taken in and another hidden state, also a vector, is produced for output.

So far the model takes a word, translates it to an integer, and finds the vector in the word embedding table that represents that word. It does this for the first word and all subsequent words one by one. Then it gives the entire vector for a word to the GRU. The GRU takes the word and passes it to some inner components. It decides weather to return as output just the input or the input modified. This is what the GRU does internally.

The input segments composed of GRUs take two input vectors and return two output vectors. One input is the vector from the embedding table. Another input vector is the hidden state. The hidden state is the same size as the input from the embedding table, but typically it comes from the previous input GRU. One output vector is like the word vector that's input to the GRU. One output vector is a hidden value.

The first word in the input sentence is converted to a vector and passed to the first GRU. The GRU has two inputs and two outputs. One input is for new data and one is for the hidden state of the segment in question.

The very first GRU in the input chain ignores the fact that the first word has no hidden value. It consumes the first word vector. Then it passes it's output to the next GRU in the chain. This GRU uses the output of the previous GRU as the hidden value. It also uses the vector for the second word. It passes it's important information to the GRU to it's right. Then the last GRU in the chain passes it's hidden state to the output chain.

Later we will use GRUs to construct output segments. Here we want to stress that the inputs cascade from GRU to GRU in the input segment.

The first GRU consumes the first word. The second GRU takes the hidden state from the first GRU along with the second word and consumes them. The third GRU takes the hidden output from the second GRU and the third word and consumes them. This pattern is repeated for the length of the input sentence. One at a time all the words from the input sentence are passed to the input GRU. The hidden state from the last GRU is passed to the output segment. The output segment is also GRU based.

A complicating detail is that although many GRUs are called for in the input segment they all use the same set of weights and biases. For this reason only a single input GRU is used for all of the words in practice. Outputs are calculated and then cycled around and fed with the next word

from the sentence in vector form to the input of the GRU.

The output chain is in charge of generating tokens that represent the input in the output language. The output uses GRU segments also. The first hidden input for the first output is taken from the last hidden output of the last recursive unit of the input chain. It is important because it is the spot where a great amount of data is passed from the input section to the output section. The connection at this point is said to carry the 'thought vector'. Most of the information responsible for translating one language to another is passed at this point.

The hidden values from the input section are passed to the first output GRU. It outputs the values that are later converted to the first word of the output. The first output GRU also has a hidden state. It passes the first word and the hidden state on to the second GRU.

The second GRU generates the second word and also its own hidden state. The second word is recorded and the word and the hidden state are passed on the the next GRU. This is repeated until a special 'end-of-sequence' token is found or until the number of tokens equals the maximum number allowed.

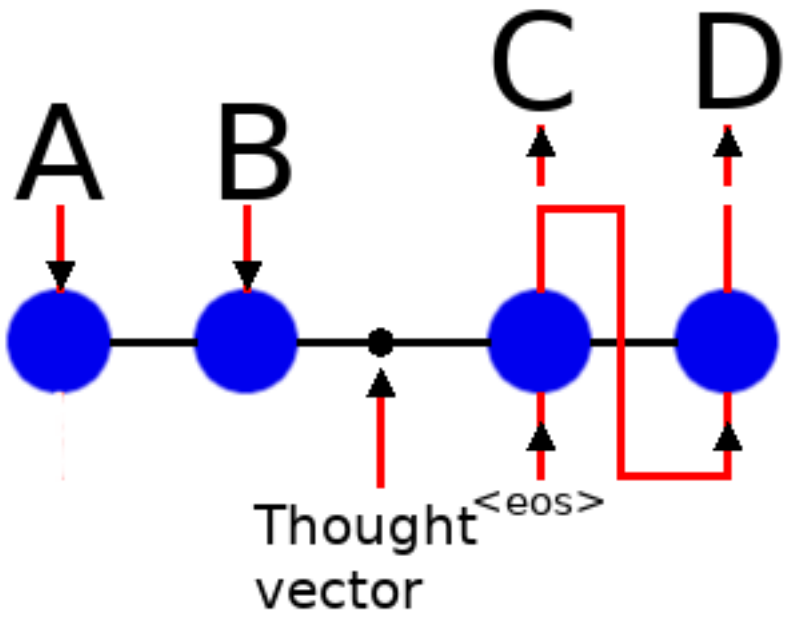

Figure 1.2: Seq2seq: A and B represent an input sequence and C and D represent the corresponding output.

Output Tokens

Each output we have is currently in the form of a vector. These vectors are a long string of floating point numbers, each one the dimensions of the 'hidden size' mentioned above. What's done with them is they are converted to the dimensions of the output vocabulary. Then they are processed in what is called an 'arg-max' function. This processing determines the index of the maximum value in the new vocabulary sized vector. This index allows the program to look up the corresponding word in the output vocabulary. This word is then used as the model output at that point in the output chain.

How does the model know what target vocabulary indexes go at what point in the output? For this we refer back to the corpus that we already have. One part of the corpus is the source language. The other part is the target language. A sentence is applied to the input and for that sentence an output is generated. The output is called a prediction. If the target and the prediction are not the same, the model must modify itself so that in the future they are.

There are some inherent problems. Because the output of a GRU is constantly being reused as the input, lots of data that might be useful to have is lost when the GRU churns through internal operations where several matrix multiplication operations are performed together on input. With every iteration more data is lost and so, for example, the effective length of the input and output sentences must be short.

Another problem is that all input to be translated to target output has to be boiled down and passed to the output section through a small corridor the size of a typical word vector. This channel is sometimes referred to as the 'thought vector.' Somehow all necessary information must be there. This also limits the length of the input and output vectors. It does help, when setting up a model for later training, to make the hidden size larger, but it only helps so much. There is a point at which the benefit of increasing the hidden size is lost.

That ends our discussion of Sequence to Sequence Translation. This is how some computer models do language translation. Using arg-max is an example of a greedy approach. Another approach might use something like 'Beam Selection' but we're not going to get into that here.

1.4 Loss and Accuracy

At first the prediction from a model is not very close to the desired output. The output is compared to the prediction and a 'loss' is generated. 'Loss' measures the difference between the predicted output and the target. A larger loss represents two values, prediction and target, that are further apart. The loss function must be chosen.

Another metric is Accuracy. 'Accuracy' is a numerical representation of the difference between the desired output and the generated prediction. It is a percentage of the time that the output is exactly correct.

Getting a prediction, running input data through a neural network, is forward propagation. Training, then, is a mathematical process involving backpropagagtion. Backpropagation identifies areas of the model weights that need to be modified in order to get the proper prediction in the future.

In practice we take the derivative of the loss function in order to backpropagate. The derivative is multiplied by the learning rate and subtracted from the original weight value. The result is a set of adjusted weight matrices. When these matrices are used later they allow for better predictions.

This is done over and over with every source/target sentence pair. Slowly the model is changed and predictions start to match the target. That's training. The loss should decrease over time and the accuracy should increase.

There are several numerical metrics that we can record during training that tell us how our model is training. The loss, a mathematical calculation of the difference between the model's output and the predicted value, is mentioned above. Loss is an important number. Also accuracy is important. Accuracy is a mathematical calculation of the difference between the model's output and the value that output should be, but it focuses on the number of times the model comes out with a correct prediction verses how many output values there are in total.

1.5 Attention Mechanism

Here we consider a simple attention mechanism that is used in the Sequence to Sequence model by Inkawhich (2018)[2]. The concept for this attention comes from Luong et al (2015)[7].

Luong et al (2015)[7] are interested in three kinds of calculation for their attention mechanism. The three methods use slowly increasing levels of complication. First they propose a method that

just uses the dot product. Then they propose a method that just uses a field of weights. Finally they use a method that uses concatination, along with a field of weights and a pass through a tanh activation layer.

$$\boldsymbol{score}(\boldsymbol{h_t}, \bar{\boldsymbol{h}}_{\boldsymbol{s}}) = \begin{cases} \boldsymbol{h_t^\intercal} \bar{\boldsymbol{h}}_{\boldsymbol{s}} & \text{dot} \\ \boldsymbol{h_t^\intercal} \boldsymbol{W_a} \bar{\boldsymbol{h}}_{\boldsymbol{s}} & \text{general} \\ \boldsymbol{v_a^\intercal} \tanh(\boldsymbol{W_a}[\boldsymbol{h_t}; \bar{\boldsymbol{h}}_{\boldsymbol{s}}]) & \text{concat} \end{cases}$$

Here h_t is the symbol for the output of the current decoder and $\bar{h}_s$ is the symbol for another output taken from the input encoder. This one is the entire set of encoder states. Inkawhich (2018)[2] uses the 'dot' variety.

The formula is used after the decoder GRU calculates it's hidden state. It is below.

$$score = h_t^\intercal \bar{h}_s$$

Basically the output of the current decoder is transposed. Then it is multiplied by the hidden value from the entire set of encoder states. The result is multiplied by the GRU decoder output, and then passed through a tanh activation layer.

1.6 Sequence to Sequence Chatbot

Vinyals et al (2015)[1] make an interesting proposition. They say that it's possible to make what they call a Neural Conversational Model by constructing a Sequence to Sequence model, but instead of using two corpus from different languages a single language is used for both source and target.

Chatbots have for a long time been constructed using AIML. AIML, (Artificial Intelligence Markup Language) requires hand coding individual rules for every input and response. A Neural Conversational Model would not use those sorts of rules.

To create a model like this more is required than just a single input and output language. There must be a relationship between the source and the target. We want there to be a question-and-answer-like relation. Finding a corpus like this can be difficult.

It would be easy to duplicate the input source material in the target corpus. This would produce auto-encoding. The model would learn to repeat everything that was given to it in it's output. Though the model learns a task, it is not the dynamic task we want. Conversations, on the other hand, supply the relationship we are looking for. Starting with almost any question sentence in a conversation, the sentence following it answers the question posed. The latter fills a space that is opened up by the former.

What would be a good candidate for this kind of verbal play? Vinyals et al (2015)[1] use a movie transcript dataset. Essentially they take movie dialogue and break it into sentences. Even numbered sentence are the source material, and odd numbered sentences are the target. Using this method there are times when the source and target are not from the same conversation, like times in a movie dialog when the scene switches from one locale to another. Comparing this, though, to the number of times that the two sentences are from the same dialogue, the movie transcript database serves very well.

Training for this model is relatively straight forward. There is a problem though. We can follow the loss and make sure it is decreasing, but the accuracy tells us nothing and in this case does not increase as we would like it to. The loss goes down but the accuracy does not go up.

Figure 1.3: Loss and Accuracy: Red is accuracy and blue is loss.

This is because the source and target do not have the same meaning. The model does learn the task at hand but during training we have to ignore the accuracy. Success is usually measured by the accuracy of the holdout test set. Here we must measure the success with a subjective examination of the trained model.

We have to interactively give the model questions that we might ask someone that we are having a conversation with, and then see how it answers. Over and over we have to test the model. If we are satisfied with the answers then the training was a success.

Chapter 2

GPT2 and Transformers

Chapter 3

Experiments

3.1 Approach to the Study

Several tasks are necessary for the project. One task is to implement the algorithms for different models, one the sequence to sequence model and the other the transformer model for a generative chatbot and finally the GPT2 model.

We will not try to rewrite the transformer or GPT2 model ourselves.

3.2 Setup

We use linux computers, sometimes with gpu hardware for parallel processing. We also use the Python programming language. Code from this project can be run with the 3.x version of Python.

When the project was started we did some programming with Keras using Tensorflow as a back-end. Keras was later discarded in favor of Pytorch. Pytorch as a library is still under development at the time of this writing.

Some of the GPT2 code uses Pytorch. Some of the Transformer and GPT2 code uses Tensorflow. There is a repository on Github that has the GPT2 trained model using Pytorch instead of Tensorflow.

We use github as a code repository. Code corresponding with this paper can be found at: https://github.com/radiodee1/awesome-chatbot .

As a coding experiment we rewrite the code for the sequence-to-sequence model. We have varying amounts of success with these experiments. We do not rewrite the GPT2 code from the Tensorflow or Pytorch repository.

3.3 ARMv7 Build/Compile

Pytorch 'torch' Library 1.1.0 For ARMv7

We compile the Pytorch library for Raspberry Pi. We use several virtualization techniques to do this compilation. The result of those efforts is a Pytorch python 3.7 library for the Raspberry Pi.

Explanation here.

Docker Container 'tensorflow-model-server' For ARMv7

Docker has to be installed on the Raspberry Pi and on the development laptop computer.

Then we use someone's pre-compiled Docker Container for ARMv7. We write our own Docker script to interact with the pre-compiled container.

Explanation here.

3.4 Experiments

We have several basic neural network models. One is the basic sequence to sequence model typically used for neural machine translation. We also have the transformer and the GPT2.

The GPT2 code is versatile. We use it for the chatbot problem. The GPT2 model is pretrained. We experiment with transfer learning and further training of the GPT2 model, but in our case it does not improve the model's performance.

When we talk about the chatbot problem on GPT2 we are talking about a PyTorch version of the GPT2 code.

We found that the chatbot with the hand coded Neural Machine Translation did not work very well. We found that the GPT2 chatbot worked well enough with the 'zero-shot' setup so that fine-tuning was not necessary.

Fine tuning on the GPT2 problem actually had a negative effect. To fine-tune GPT2 also involved training the model on Tensorflow and then translating the model to PyTorch when that was done.

We use speech recognition and speech to text libraries to allow a user to give the chatbot model auditory input.

We also train the transformer to do the chatbot task. This works better than the sequence to sequence but not as well as the GPT2.

Finally we describe installing the chatbot in small computing platforms, the Raspberry Pi and the O-Droid, in an attempt to create a smart speaker.

All of the three mentioned models work on a laptop computer.

It should be noted that at some point we would like to describe some of our subjective results with the different models. It is difficult to measure our results objectively because we are using code most closely associated with a translation task.

If we were measuring progress with an actual translation task accuracy could be measured as a score that improves when the output matches the meaning of the input, albeit in a different language. We don't use an actual translation task, we use something close to one. The problem is that though the input and the output are in the same language, the input and output have different meanings. We do not dictate what the correct output would be for a given input. The output just has to make sense as a reply in the English language. We would actually prefer that the output not have the same meaning as the input. This makes it difficult to calculate an objective accuracy score. This is touched on in the paper from Vinyals et al(2015)[1].

Loss can still be monitored during training. When the loss stops decreasing you know you should stop training as the model is probably overfitting.

3.5 Chatbot - GRU Model

We trained the sequence to sequence model on a large english corpus in an attempt to produce a chatbot.

For the sequence to sequence model we want to use text found in a movie dialog corpus (Danescu-Niculescu-Mizil et al, 2011)[8].

In our experience with coding our own chatbot we find that the model learns a single english sentence that can be used for most replies.

Subjectively the GRU chatbot is unusable.

For example our chatbot frequently replies to input with the phrase "I don't know". This is the most common output from our model. In this respect our original chatbot responds poorly.

A large portion of the time the bot's response to a question is "I don't know". This is interesting in that the chatbot has identified that this phrase is a suitable answer to many questions, but it is disappointing that there isn't more variety to the output.

Occasionally the sequence-to-sequence model will reply with the phrase "I'm sorry." This happens very infrequently and it is not clear why the model chooses to reply this way.

3.6 Smart Speaker - GRU Model

The GRU model was installed on a Raspberry Pi. This allowed us to test out speech-to-text and text-to-speech libraries.

It also allowed us to compile the Pytorch library for Raspberry Pi.

3.7 Chatbot - Transformer Model

Using the Persona corpus we trained a transformer model to use as a chatbot. This transformer was not pre-trained with any large corpus, so this example did not use transfer learning.

Subjectively this model did not perform as well as the GPT2 model, but it did perform better than the GRU model.

The memory footprint of the model while it was running was below 1 Gigabyte. It is conceivable that the model could be installed on a Raspberry Pi board but it requires a python package called 'tensorflow-model-server' and this package would have to be built from source for the Raspberry Pi.

Training of the model followed a certain pattern. First the model was trained on the persona corpus until a familiar pattern emerged. When the model began to answer all questions with the phrase "I don't know" training was stopped.

At that time the corpus was modified to include no sentences that have the word "don't" in them. Training was started again until the output contained nothing but the phrase "I'm sorry."

At that time the corpus was modified to include no sentences that have the word "sorry" in them. Training was started again and was continued for some period. Training was stopped. A further segment of training was not attempted.

At this point, after looking at the change in loss, further training was not thought of as helpful. Loss stopped improving at some point in this process, and this lack of improvement was taken as a sign that progress was not likely.

Subjectively the transformer model is better than the GRU model. It can respond to something like four sentences. When it comes upon a question that it doesn't expect it defaults to a certain sentence. It can answer questions that you might ask in a rudimentary conversation. It has answers to prompts like 'hi', 'How are you?' and 'What do you do?'. If you tell it your name it will tell you that its name is 'Sarah'. It doesn't answer arbitrary questions. It cannot answer 'What is your

favorite color?'. It cannot tell you the time. The default reply sentence for unknown prompts is 'Hi, how are you today?'

3.8 Smart Speaker - Transformer Model

Later we look to install the transformer model on the Raspberry Pi. This model is more dynamic than the sequence-to-sequence model.

The transformer model takes about two minutes to boot on the Raspberry Pi. After that the time between responses is slow. The time between the first two or three responses is uncomfortably slow. After those first responses the time between answers gets to be more natural.

3.9 Chatbot - GPT2 Model

We used a pre-trained GPT2 model with the large english corpus to produce a chatbot and ascertain if this model works better than the sequence-to-sequence model. In our tests this worked well. The corpus is called 'WebText'.

For our experiments GPT2 was used for the chatbot model in 'zero-shot' mode. This means we did no special fine-tuning of the GPT2 model in the application.

We did do some special coding for the input and output of the GPT2 code in order to operate it as a chatbot. Input and output was limited to about 25 tokens.

Input to the model was prepended with the character string "Q:" by our code. Output was observed to have the character string "A:" prepended to it. We assume therefore that the GPT2 model was at some point exposed to the "Question/Answer" paradigm in written passages during its training. This was helpful.

Output from the GPT2 model was usually larger in size than we needed. Also, output had the character of having some sensible utterance followed by some output that was only a partial sentence.

It was necessary to 'scrape' the output. First the output was checked for the "A:" character string at the start. If it was there it was removed. Then the first complete sentence was used as output, while words and phrases after that were discarded.

Lastly we decided that we would attempt to give the model some details that it could draw

on during normal execution. We had two choices here. One choice was to train the model using fine-tuning and transfer learning to recognize certain questions and to supply answers. The other choice was to simply show the model the list of facts that we thought were important before every input sequence. This information would be summarized with each reply.

The second choice was more interesting. The text that the model was shown always included the name of the model (picked somewhat arbitrarily) along with information about the location of the model and the occupation. The time was also included.

Subjectively the model was the best of the three tested. The model would answer questions about it's location, it's name, and the time, faithfully most of the time. Interestingly there where times when it did not do so. Some times it used alternative answers. For example, it would answer with the time but not the correct time. This was odd.

Under almost all circumstances the output was sensible English. There were few if any times where the model replied with gibberish.

The subject matter of the prompts did not need to be the same as the simple introductory conversation of the transformer model. In fact any subject matter could be chosen and the model would answer. The model did not remember its own answers but it was consistent. Questions it answered include 'What is your favorite color?' and 'Do you like lollipops?'.

3.10 Smart Speaker - GPT2 Model

Tests showed that the GPT2 chatbot worked well. We wanted to continue and allow the chatbot to have more of the abilities of a smart speaker. We constructed a simple corpus that contained key phrases that we wanted the chatbot to recognize and act upon. We did some transfer learning with this new corpus.

We found that one of two things would happen. The chatbot would either learn the new phrases and forget all it's pre-training, or it would not learn the new phrases and it would retain all it's pre-training. For our examples there seemed to be no middle ground. Comparisons were made with all available models and a version without the transfer learning was settled on.

Code was added that uses Text To Speech and Speech To Text libraries. In this way the model could interact with a subject using auditory cues and commands.

We did some programming that allowed the GPT2 model to launch programs when directed to

by the user. In this way we have tried to move our project closer to the smart-speakers that are produced commercially. The programming did not rely on the neural-network aspects of the model. Instead the code used heuristics and simple word recognition. This code can be disabled when the model is run from the command line.

Appendix A

Terminology

A.1 Gated Recurrent Unit

A Gated Recurrent Unit is a relatively simple recurrent network component. It is a component where the output of the model is fed, after some modification, to the input of the model.

At each time step the model sees the new input and a version of previous inputs. In this way it remembers previous inputs. Several inputs in a series are fed to a GRU in sequence and the GRU is responsible for deciding which of the inputs is most important.

GRU elements, like all recurrent network components, suffer from information loss when the input segments are very long.

A.2 Sequence to Sequence - Chatbot

The sequence to sequence model we use is based on the 'Neural Machine Translation' model, or NMT. Originally designed for translating one language to another, NMT can be used to create a chatbot.

In NMT two languages are given to the sequence to sequence model. For a chatbot NMT is used with two identical language sets. The input language is the same as the output language.

After establishing this language relationship all that is left is to find a suitable training set. Vinyals et al (2015)[1] use a set of movie subtitles. For each sentence in the data set the network is trained to reply with the contents of the next sentence in the set.

It is pointed out in the paper (Vinyals et al, 2015)[1] that the NMT network is typically designed to answer questions directed at the computer with an answer that has the same meaning - though in a different language. Training for the chatbot consists of sentence pairs that are not necessarily related in that way.

During training accuracy improvements are not necessarily reflective of progress, where loss improvements are.

We use a GRU arrangement for a Sequence to Sequence chatbot.

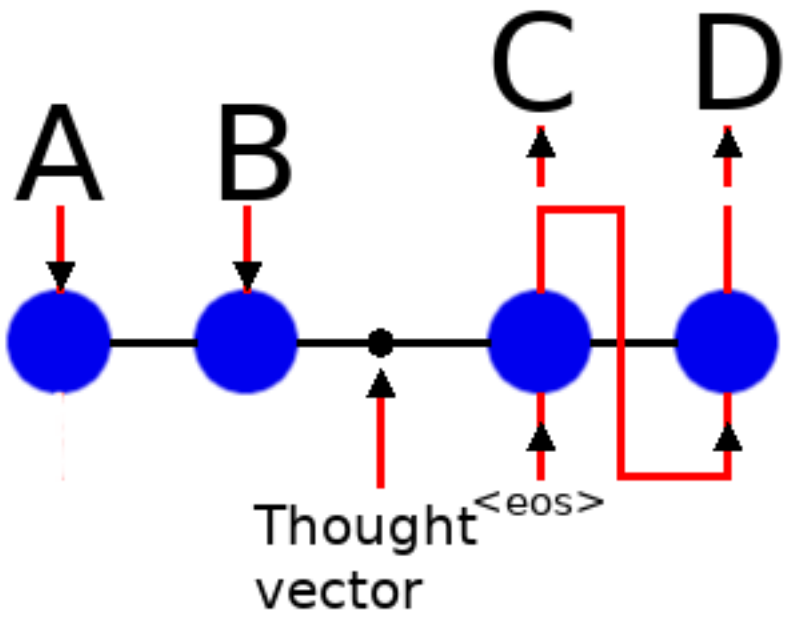

Figure A.1: Seq2seq: A and B represent an input sequence and C and D represent the corresponding output.

A.3 Sequence to Sequence - Architecture

In Fig. A.1 we generalize a sequence to sequence model. The idea is that A and B, on the left side of the diagram, deal with the encoding of sentences. A and B would be consecutive words in a sentence, and the round blue nodes below A and B are RNN units. They are Recurrent Neural Network elements. C and D are outputs and in the right side of the diagram the blue nodes represent the output RNN units. Between the input and the output there is a corridor of information exactly the size of the RNN hidden vector.

All of the information that the decoder uses for it's output is present in this corridor and is passed along the corridor from the encoder. For this reason we refer to it as the thought vector.

Making this vector larger by increasing the size of the hidden unit size allows for more information in the thought vector. Size also increases the time to train the network. The size must also match the dimension of the vectors in the GloVe or Word2Vec download if one of those is used.

Ultimately exceedingly large hidden dimension does not improve the sequence to sequence model.

A.4 Corpus Considerations

We have collected several data sets for the training of a chatbot model. Firstly we have a corpus of movie subtitles. Secondly we have the 'JSON' dump from Reddit that is downloadable. Finally we have the corpus described by Mazaré et al(2018)[9]. This final corpus is designed for training the chatbot task specifically. This is referred to as the Persona corpus.

The movie corpus is medium sized and the Reddit 'JSON' download is large and filled with hyperlinks and sentence fragments.

At the time of this writing we are using the movie subtitles corpus and the Persona corpus. We use the movie corpus because it is smaller. Both the movie corpus and the Reddit corpus are described as noise filled, so it is likely that neither one is perfect for the training. The movie corpus is easier to deal with if we are training on a single processor. In the future if we can train in a faster environment the Reddit corpus might be superior.

For the Persona corpus the text is organized into 'JSON' objects. There are several different repeated labels. Some of the text is meant to be used in question and answer pairs. There is also some very specific information there that is not organized in this kind of pattern. When we take apart the Persona corpus we find that the sentences labeled with the 'history' tag are most suited to our task. We record these values only and discard other labels.

A.5 Word Embeddings

There are several basic building blocks of sequence to sequence models. They are regular neural network cells, recurrent network components, and word embedding components.

Regular neural network cells, are arranged in layers and are pretty straight forward in most

programming environments. To use them you define their dimensions in width and height. They have weights and biases that must be initialized before use.

Recurrent networks have internal parts that are constructed of regular network cells, so they have weights and biases too. They have several internal dimensions that need to be set. One of these is the RNN hidden unit. This hidden unit is a dimension.

Word embedding components are the third item we want to describe. What happens is words are translated from strings to individual numbers from a vocabulary dictionary. The dictionary only contains a single unique number for every word. Then the number is passed through an embedding structure. This turns the single number into a vector of numbers that is the same size as the RNN hidden dimension. Then, from that time on the model uses the vector instead of words.

The contents of the embedding unit is a table of numbers, all of the size of the RNN hidden dimension. The vectors are usually, but don't have to be, unique values. There is one complete hidden dimension sized vector for each word in the vocabulary.

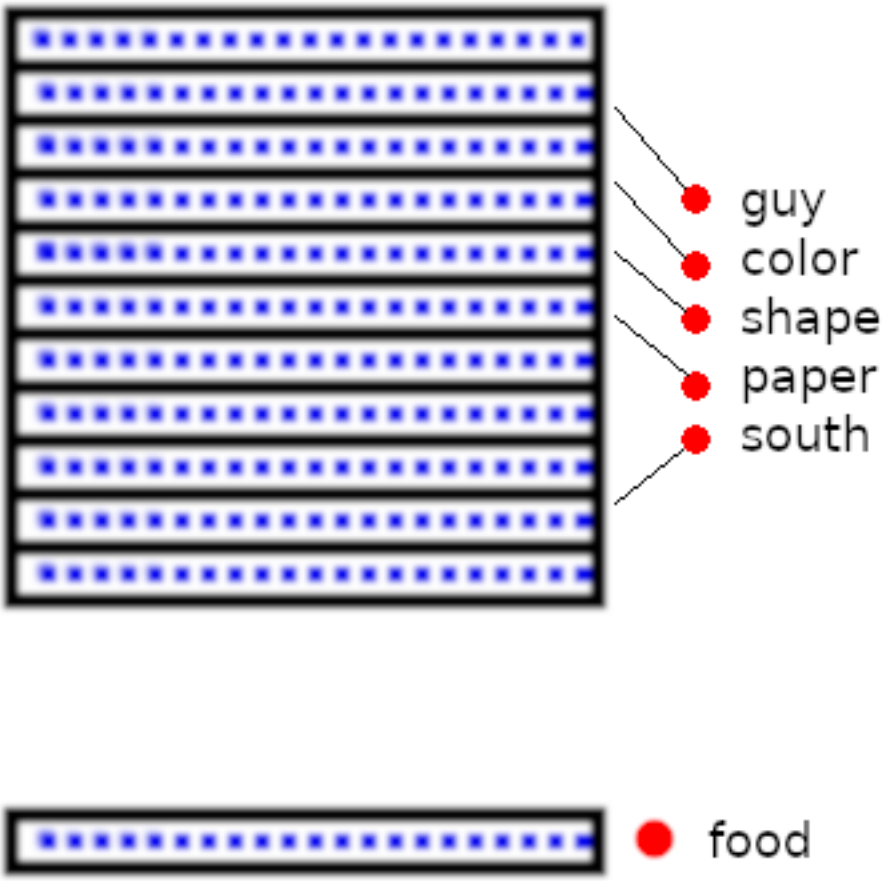

Figure A.2: Embeddings: Each word from a dictionary is converted to a vector of numbers.

The vectors can be initialized randomly or they can be filled with predetermined values. As the network trains the embedding values can either be modified or frozen in place. Typically if the contents were initialized randomly the values would be trained. If the contents were filled with predetermined values you don't want to train them or change them in any way.

There are at this writing two main types of pretrained word embeddings. One is called 'Word2Vec' and one is called 'GloVe'.

Word2Vec is short for 'Word to Vector.' (Mikolov et al, 2013)[10] GloVe is short for 'Global Vector.' (Pennington et al, 2014)[11] .

A.6 WordPiece - BPE

BPE stands for 'Byte Pair Encoding.' WordPiece is a particular implementation of Bite Pair Encoding.

WordPiece is used by the BERT system to encode words much the way that Word2Vec does. Like Word2Vec, WordPiece has a vocabulary list and a table of embeddings that maps one word or token to a vector of a given size.

WordPiece, though, handles Out Of Vocabulary (OOV) words gracefully. It breaks large words into smaller pieces that are in the vocabulary, and has a special notation so that these parts can easily be recombined in order to create the input word again.

A.7 Transformer

In this section we discuss the 'Transformer' model. In their paper, Vaswani et al (2018)[12] describe use of the python project Tensorflow and the Transformer model which is part of it.

The Transformer uses no recurrent elements. It is in a sense a group of attention mechanisms. The Transformer in this case is a single structure that can be used to solve many machine learning problems. It is used for Neural Machine Translation, Sentiment Analysis, and others. It is a single model capable of solving many machine learning questions.

We use the translation models for the chatbot problem by feeding the model the English language on both input and output.

The Transformer itself can be configured for sentence long output but it is not a pre-trained model. There are pre-trained versions of the transformer, one of which is called BERT. BERT is described in Devlin et al (2018)[13] and the acronym stands for Bidirectional Encoder Representations from Transformers. Unfortunately BERT output is as a classifier. Full sentence-length output is not supported.

A.8 GPT2

Another pre-trained model that uses the Transformer is GPT2. GPT2 stands for 'Generative Pre-training Transformer 2.'

GPT2 is a model that takes as input a seed sentence or topic and returns as output text in the

same language that is auto-generated. GPT2 also is very capable at summarizing the input or seed statement. We use both of these capabilities in our experiments.

There are two GPT2 models. One is larger than the other. The smaller model has been released and the larger model has not.

GPT2 is discussed in Radford et al (2019)[4] and on the blog associated with the parent company, OpenAI.com. (https://openai.com/blog/better-language-models/)

GPT2 comes with its own vocabulary encoding and decoding functionality. This system is closer to WordPiece and BPE than it is to Glove or Word2Vec.

It is pre-trained on a corpus that is developed from Reddit posts called WebText. WebText is a 40 Gigabyte corpus that takes high powered computers to train.

The version of GPT2 which has been released is similar in size to the largest currently released BERT transformer model. GPT2 has been released in Tensorflow format, and more recently in a converted PyTorch format. It is possible to download the trained model and then fine-tune the model for your own task. This kind of fine-tuning is called 'transfer learning'.

GPT2 works so well in certain conditions that it is appropriate to use without fine tuning. This type of implementation is called 'zero-shot' implementation. We use GPT2 in a zero-shot implementation with the chatbot problem.

A.9 Raspberry Pi

A Raspberry Pi is a small single board computer with an 'arm' processor. There are several versions on the market, the most recent of which sports built-in wifi and on-board graphics and sound. The memory for a Raspberry Pi 3B computer is 1Gig of RAM. Recently available, the Raspberry Pi 4B computer can sport 4Gig of RAM.

It has always been the intention that at some time some chatbot of those examined will be seen as superior and will be installed and operated on a Raspberry Pi computer. If more than one model is available then possibly several models could be installed on Pi computers.

For this to work several resources need to be made available. Pytorch needs to be compiled for the Pi. Speech Recognition (SR) and Text To Speech (TTS) need to work on the Pi.

For the transformer model to work Tensorflow needs to work on the Pi.

All the files that are trained in the chosen model need to be small enough in terms of their file

size to fit on the Pi. Also it must be determined that the memory footprint of the running model is small enough to run on the Pi.

In the github repository files and scripts for the Raspberry Pi are to be found in the 'bot' folder.

Early tests using Google's SR and TTS services show that the Pi can support that type of functionality.

Google's SR service costs money to operate. Details for setting up Google's SR and TTS functions is beyond the scope of this document. Some info about setting this up can be found in the README file of this project's github repository.

The pytorch model that is chosen as best will be trained on the desktop computer and then the saved weights and biases will be transferred to the Raspberry Pi platform. The Pi will not need to do any training, only inference.

A.10 Speech

Google has python packages that translate text to speech and speech to text. In the case of text to speech the library is called 'gTTS'. In the case of speech to text the library is called 'google-cloud-speech'.

The gTTS package is simple to use and can be run locally without connection to the internet. The google-cloud-speech package uses a google cloud server to take input from the microphone and return text. For this reason it requires an internet connection and an account with Google that enables google cloud api use. Google charges the user a small amount for every word that they translate into text.

Appendix B

Tables

B.1 Model Overview

Model Name	File Size	RAM Train	RAM Interactive	Pretrained Weights	Hand Coded
Seq-2-Seq/GRU	230 M	1.1 G	324 M	NO	YES
Transformer	25 M	556 M	360 M	NO	NO
GPT2 small*	523 M	5 G	1.5 G	YES	NO

* a large GPT2 model exists, but it has not been released to the public.

B.2 Question Answering – babi

	DMN plus	hand coded	GPT2 - small
QA1: Single Supporting Fact	100	100	100
QA2: Two Supporting Facts	99.7	xx	96.0
QA3: Three Supporting Facts	98.2	xx	38.18
QA4: Two Argument Relations	100	100	100
QA5: Three Argument Relations	99.5	99.4	97.8
QA6: Yes/No Questions	100	100	98.4
QA7: Counting	97.6	97.8	98.6
QA8: Lists/Sets	100	99.4	98.8
QA9: Simple Negation	100	98.2	97.0
QA10: Indefinite Knowledge	100	99.4	96.6
QA11: Basic Coreference	100	100	97.6
QA12: Conjunction	100	100	99.4
QA13: Compound Coreference	100	99.8	95.8
QA14: Time Reasoning	99.8	97.2	87.0
QA15: Basic Deduction	100	100	64.4
QA16: Basic Induction	54.7	48.2	96.39
QA17: Positional Reasoning	95.8	59.2	99.0
QA18: Size Reasoning	97.9	91.6	100
QA19: Path Finding	100	xx	97.3
QA20: Agents Motivation	100	100	100

Bibliography

[1] O. Vinyals and Q. V. Le, "A neural conversational model," *CoRR*, vol. abs/1506.05869, 2015.

[2] Matthew Inkawhich, "pytorch-chatbot," 2018. https://github.com/pytorch/tutorials/blob/master/beginner_source/chatbot_tutorial.py.

[3] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *NIPS*, 2017.

[4] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," 2019.

[5] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, and J. Brew, "Huggingface's transformers: State-of-the-art natural language processing," *ArXiv*, vol. abs/1910.03771, 2019.

[6] Denny Britz, "Recurrent neural network tutorial, part 4 – implementing a gru/lstm rnn with python and theano," 2015. http://www.wildml.com/2015/10/recurrent-neural-network-tutorial-part-4-implementing-a-grulstm-rnn-with-python-and-theano/.

[7] M. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," *CoRR*, vol. abs/1508.04025, 2015.

[8] C. Danescu-Niculescu-Mizil and L. Lee, "Chameleons in imagined conversations: A new approach to understanding coordination of linguistic style in dialogs.," in *Proceedings of the Workshop on Cognitive Modeling and Computational Linguistics, ACL 2011*, 2011.

[9] P. Mazaré, S. Humeau, M. Raison, and A. Bordes, "Training millions of personalized dialogue agents," *CoRR*, vol. abs/1809.01984, 2018.

[10] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[11] J. Pennington, R. Socher, and C. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, (Doha, Qatar), pp. 1532–1543, Association for Computational Linguistics, Oct. 2014.

[12] A. Vaswani, S. Bengio, E. Brevdo, F. Chollet, A. N. Gomez, S. Gouws, L. Jones, L. Kaiser, N. Kalchbrenner, N. Parmar, R. Sepassi, N. Shazeer, and J. Uszkoreit, "Tensor2tensor for neural machine translation," *CoRR*, vol. abs/1803.07416, 2018.

[13] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *CoRR*, vol. abs/1810.04805, 2018.