Personal Chatbot

Bezawit Woldegebriel
Department of Computer Science
Georgia State University
Atlanta, GA 30303
mogesbezawit@gmail.com

1. Problem and goal

Chatbots are computer programs that conversate via audio or text. They are artificially intelligent systems that use natural language processing to simulate human interaction. Chatbots have been around since the 1950's when the Turing test was first developed. The Turing test is like an imitation game where on one side of a computer there's a human judge who chats with an anonymous interlocutor. On the other side, most of the conversationalists are humans and one is a chatbot. If the human judge is tricked into thinking they're talking with a human when it's actually a chatbot, then the bot is considered 'intelligent'.

Today, some of the more advanced and widely used chatbots include Apple's Siri, Microsoft's Cortana, Google Assistant and Amazon's Alexa. Besides acting as virtual assistants on personal devices, large organizations are increasingly adopting chatbots for different purposes like entertainment, business to customer marketing and sales and to provide better customer service. Companies like Dominos and Pizza Hut have their own chatbots to increase customer engagement. The possibilities here are endless.

My goal for this project is to train a recurrent neural network (RNN) based personal chatbot purely for entertainment purposes. It'll also help me understand and apply machine learning algorithms. The chatbot will be able to give one-line responses at a time.

2. Project

2.1 Data

The data I used to train the chatbot consists of my own personal texts complied from several different sources including backed up text messages, as well as messages from Facebook, WhatsApp and Viber. Although I was aiming to accomplish this project by solely using my personal data, I didn't think the total size of data I finally got was large enough to train a decent chatbot. Therefore, I've included some more data from external sources. [1] is a collection of fictional conversations taken from movie scripts. It contains over 220,000 conversations between 10,292 pairs of characters. It's about 35 MB in size. The total size of training data was approximately 100 MB. After all data was collected, it was re-organized into two separate files corresponding to 'questions' and 'responses'.

2.2 Model

From the files created, I used Python to train a new chatbot model (applying supervised learning) using TensorFlow's Neural Machine Translator (NMT) (seq2seq) [2], [3], [4]. This model involves a text input and a text output. It's made up of two recurrent neural networks– an encoder and a decoder. The encoder RNN will get a sentence and convert it into a feature vector representation – a sequence of numbers that represents the sentence's meaning. As the encoder goes from one word to the other, it will only capture the most important information from the sentence and will get rid of the useless ones. Therefore, it'll finally be left with a summary of the sentence, called a context or thought vector. The decoder RNN will then take this thought vector, look for words in its vocabulary and come up with words one by one that make up the best response. Each generated word is influenced by the previous one. The following figure is an example of an encoder-decoder architecture for an NMT that translates English phrases to French.

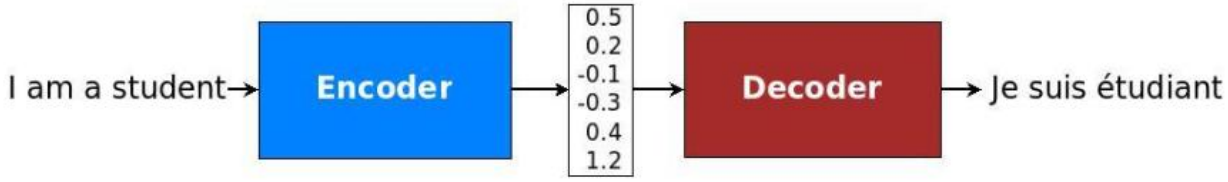


Figure 1 Encoder-decoder architecture – An encoder converts a source sentence into a "meaning" vector which is passed through a decoder to produce a translation [4].

Efficiently converting words to number vectors is one of the main tasks of this project. One way to do this is to create a matrix of 1s and 0s where each row would have a single 1 representing the word and the rest is set to 0s (one-hot method). However, using this method to train a model is impractical. Models typically learn from vocabularies made up of thousands of words and therefore, the input layer would have to consist of thousands of nodes. Moreover, this technique would remove any local context of the words to be represented, which for natural language is a major issue. A good solution for this and a key concept in seq2seq model is word embedding.

Word embedding enables words or phrases that have the same meaning or are used in the same way have similar vector representation. It also serves as an efficient encoding technique. Word2vec is a word to vector representation model that makes use of word embedding; it's also the one I used [5]. Let's say we have a vocabulary of 15,000 words and we want to decrease this to a 500-length embedding. If we take the word 'chair', which is one of 15,000-words in the vocabulary, word2vec would represent it as a 15,000 length one-hot vector. We would then feed this vector to a 500-node hidden layer. The weights connecting this layer (15,000 x 500 matrix) would produce a new word vector which is then an input to a softmax output layer. While we train the model, we would change the weights of this network so that words around 'chair' would have higher probability in the softmax layer. So, for example, if the training dataset has sentences like 'She sat on the chair.', words like 'sat' and 'on' would be associated with each other. Once training is done, we wouldn't use the softmax layer since the weight matrix would basically serve as an encoding table.

One thing to consider while dealing with a sequence to sequence model for a chatbot, like the one I'm making, is the variable lengths of each sequences. Each input phrase is given a corresponding token

using word vectors [5] to create training samples. The issue is that one-word questions could lead to long worded responses while long worded questions could only require a single-word response. Moreover, each input is going to be different from the previous regarding characters, words etc. One solution to handle the varying word lengths is to pad the training dataset before training. For this to work, we would set all statements to 15 words, for instance, and if the number of words in a statement is less than that we'd pad it with other symbols till it gets to 15. On the other hand, if a statement is more than 15 words long, we would either cut off the extra words or just not use it.

However, padding becomes an issue when the longest sentence in our training dataset is 100 words. Now all the input statements would have to be padded to 100 words. For short worded questions, the padding would dominate over the actual important information. This is where bucketing comes in. Bucketing categorizes input statements into buckets based on the size of words. For example, if we have a bucket list of [(5,15), (15,25), (25,40), (40,60)] and the statement to be encoded has 7 words and its response is 10 words, it'll be put into the (5,15) bucket. Both the input and output statements would be padded to a length of 15. When the model is being run, separate models are used for each bucket, consistent with the question – response pair.

3. Schedule

Task	**Duration**
Acquiring data from several sources	2 weeks
Formatting data into useable training data	2 weeks
Training different models and compare them	3 weeks
Writing final report	1 week

4. Implementation

The first thing that must be done to build a RNN based chatbot is to train a model. To make the job easier, I started with TensorFlow's RNN implementation of a seq2seq model used to train a translation model from English to French [6]. As mentioned above, this model consists of an encoder and decoder. The encoder represents English words as numerical vectors and the decoder processes these vectors to come up with equivalent French words that serves as translation. The idea here is that instead of giving it English to French words and phrases to be trained upon, I'm going to be giving it English to English question-response pairs. This way we can have one lined conversations that are both in English i.e. a chatbot.

The following are code snippets for creating the model and the encoder and decoder portions that are used in the project:

```
# Create the internal multi-layer cell for our RNN.
single_cell = tf.nn.rnn_cell.GRUCell(size)
if use_lstm:
  single_cell = tf.nn.rnn_cell.BasicLSTMCell(size)
cell = single_cell
if num_layers > 1:
  cell = tf.nn.rnn_cell.MultiRNNCell([single_cell] * num_layers)
  tf.nn.seq2seq to "tf.contrib.legacy_seq2seq"
```

Figure 2 Seq2Seq model creation [6]

```
# Feeds for inputs.
self.encoder_inputs = []
self.decoder_inputs = []
self.target_weights = []
for i in xrange(buckets[-1][0]):  # Last bucket is the biggest one.
  self.encoder_inputs.append(tf.placeholder(tf.int32, shape=[None], name="encoder{0}".format(i)))
for i in xrange(buckets[-1][1] + 1):
  self.decoder_inputs.append(tf.placeholder(tf.int32, shape=[None], name="decoder{0}".format(i)))
  self.target_weights.append(tf.placeholder(dtype, shape=[None], name="weight{0}".format(i)))
```

Figure 3 Encoder, decoder and buckets [6]

Some data is required to train this chatbot model. For testing and comparison purposes, I've trained a few models each one using more training data than the previous. As mentioned above, the aim of this project is to build my own personal chatbot. Therefore, I'm mostly going to be using my personal messages. To train the first model, the first source I turned to was Facebook. I requested for my data to be compiled and after a couple days I was able to get all my messages since my account was created back in 2009. The next step I took was appropriately formatting all the messages into question-response pairs for the seq2seq model to use. The messages came in several 'htm' format files which I converted to a single JSON text file using Facebook Chat Archive Parser [7]. This file came out to be a little over 10MB (I expected more, I don't think Facebook gave me all my messages). Since this text file included other unwanted information, I wrote a simple Python script to extract only the messages and remove the unnecessary things like emojis, blank spaces or any unique characters that aren't words or numbers. Once I got a text file with just the messages I separated every odd lined message to another file. These two are the question and response data that'll be used to train the model.

Other sources I included to train more models are my What's App, Viber, text messages and movie dialogues [1]. For What's App, I have a chat history backup that's locally saved to the device (which I then upload to Google Drive) and is regularly updated. The backup files are encrypted but with a rooted phone, the key can be easily located. I used WhatsApp crypt12 decryption tool [8] to decrypt the backup messages and save it as an sqlite3 database file. After some data formatting, I ended up with question-response pairs similar to Facebook files created.

Acquiring Viber messages was relatively easier. The app itself provides a way to export a zip file that contains several csv files corresponding to contacts I've chatted with. I combined these files, extracted only the useful information and added the output to the question-response files I have compiled so far.

To get backups of my text messages from my phone, I installed SMS Backup+ [9]. This app saved my messages as emails in a separate label. I then used Google Takeout [10] to downloaded them all into one big MBOX format file. From here I used [11] to convert messages to csv file. Just like the

rest, I reorganized the file so I would only have the contents of the messages. Finally, I combined the output to the other question-response files.

The last source I added was movie dialogues [1]. Movie transcripts came in one long text file that contained other unnecessary formatting. I used a dialogue converter [12] to convert it to an appropriate format that can be used for training (question-response). The total size of the two question-response files came out to be around 100 MB. The following is approximate amount of contributions from each source: Facebook – 10MB, What's App – 15MB, Viber – 15MB, SMS – 25MB and movie dialogues – 35MB.

To improve quality of training and the overall outcome of the final chatbot model, I've separated the training files into train and test. The last few models I trained had an input of four files: train.a, train.b, test.a and test.b. The *.a files are 'questions' while the *.b files are 'responses'.

Now that the training data is ready, it's time for the actual training. To preserve my local machine's processing resources, I used Google's Cloud Machine Learning (ML) Engine from Google Cloud Platform (GCP). The Cloud ML Engine:

- Enables you to train machine learning models at scale by running TensorFlow training applications in the cloud. [8]
- Hosts those trained models for you in the cloud so that you can use them to get predictions about new data. [8]

These tasks are essential for this project as I trained all the models using ML Engine. GCP also allows for the models host my model in the cloud so anyone can access it. Therefore, this engine is prefect for the job. The Cloud ML Engine also comes with a Google Cloud Shell, a command-like environment for managing cloud resources hosted on GCP. Another service provided by GCP is Cloud Storage Buckets. Buckets are essentially containers that hold data. For this project, I used a bucket to store my training data and the model itself. I was originally planning to use some sort of database to store my training data, however these buckets serve the purpose.

5. Results

I began training from the data I had prepared (see section 4). As mentioned above, I've trained several models with different sets of training data, each with more data than the previous. I noticed that the more training data I gave it, the longer time it took to get to checkpoints. The following is one of the first models I've trained. During the process, some checkpoints were created, which allowed for some testing. After overnight training and 32,000 iterations, this was the outcome of the chatbot:

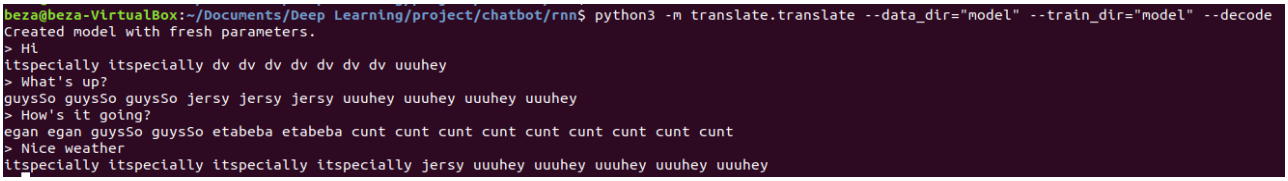


Figure 4 Chatbot after 32,000 iterations

After 82,000 iterations:

```
beza@beza-VirtualBox:~/Documents/Deep Learning/project/chatbot/rnn$ python3 -m translate.translate --data_dir="model2" --train_dir="model2" --decode
Created model with fresh parameters.
> Hi
yeswa yeswa yeswa yeswa yeswa Education Education alkush alkush alkush
> What's up?
Due Gebe Bezayeee Bezayeee Bezayeee Bezayeee Idkminew Idkminew Idkminew Idkminew
```

Figure 5 Chatbot after 82,000 iterations

As it can be inferred from figures 4 and 5, the chatbot isn't too smart. The responses are repetitive and out of context. I believe the reason that the chatbot acts this way is because of the size of training data it was given. Given that this is one of the first few models trained, it's not too surprising. The responses might not completely make sense to an average reader (see section 6). For example, the word 'beyalawe' translates to 'I've said' and 'yeswa' means 'hers'. You can also see my name (Bezayeee) in the last line.

Looking at the logs, the training process consisted of four buckets (see section2.2) – each with their own perplexity. Perplexity measures how well a probability model predicts a sample. Lower perplexity means better predictions. Figures 6 and 7 show the decreasing overall perplexity during the training process. Another important thing to mention here is the decreasing learning rate. The learning rate describes how quick the model learns new stuff. A higher learning rate means it understands faster. However, this isn't always a good thing. We generally want the learning rate to decrease overtime to produce a better model.

The training log presented in the following is for one of the models I think is better than the rest. It also used all the training data I've compiled. The following is what the log looked like at 28,000, 46,000 iterations respectively. The steps per checkpoint was set to 500. It was trained for over eight days in GCP's ML Engine.

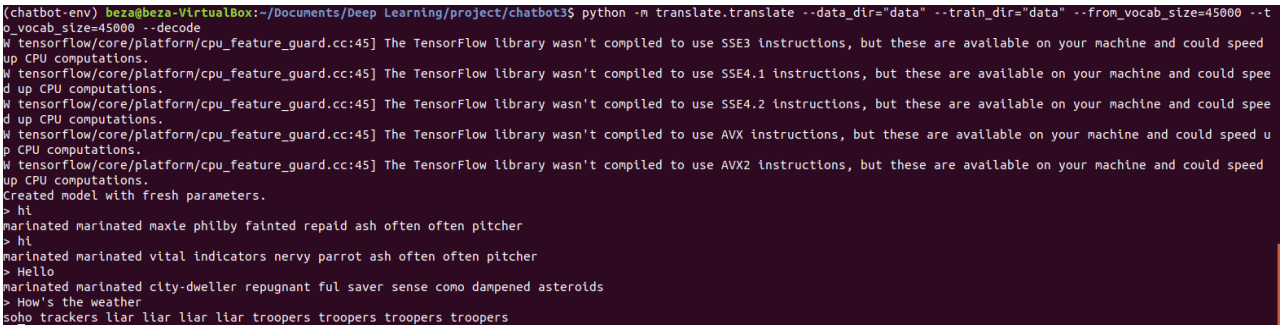
```
2018-04-24 04:40:01.978 EDT master-replica-0 global step 28000 learning rate 0.5000 step-time 15.05 perplexity 2.54
2018-04-24 04:40:01.980 EDT master-replica-0 eval: bucket 0 perplexity 65.69
2018-04-24 04:40:01.980 EDT master-replica-0 eval: bucket 1 perplexity 220.43
2018-04-24 04:40:01.981 EDT master-replica-0 eval: bucket 2 perplexity 122.24
2018-04-24 04:40:01.981 EDT master-replica-0 eval: bucket 3 perplexity 172.09
```

Figure 6 Training log after 28,000 iterations

```
2018-04-27 11:14:03.321 EDT master-replica-0 global step 46000 learning rate 0.4950 step-time 15.52 perplexity 1.40
2018-04-27 11:14:03.321 EDT master-replica-0 eval: bucket 0 perplexity 608.11
2018-04-27 11:14:03.322 EDT master-replica-0 eval: bucket 1 perplexity 2215.07
2018-04-27 11:14:03.322 EDT master-replica-0 eval: bucket 2 perplexity 897.93
2018-04-27 11:14:03.322 EDT master-replica-0 eval: bucket 3 perplexity 1813.47
```

Figure 7 Training log after 46,000 iterations

Here is how the final model I trained interacts after 46,000 iterations:



In order to get better responses from the chatbot, the model should keep on training until the learning rate goes down to about 0.2 or 0.1. Due to time and GCP credit constraints, this is the best I could get it to conversate.

6. Difficulties and limitations

Regular human conversation, especially in text, isn't usually one lined. This model doesn't take that into account. It's been trained on one liners therefore it'll spit out one liners. However, in my text messages and in the movie dialogues, there are situations where a single question has several responses or there's only one response for several questions or both. When I split the training data into question-response pairs, I randomly categorized the messages into each pair (even lines were questions, odd lines were responses). This is a problem because the model might not display a meaningful response to a given question.

Another issue I didn't foresee is that a considerable amount of my text messages isn't in English. Since my native language is Amharic, that's the language I use to communicate with some people. Although it has its own completely unique set of alphabets, we usually use English alphabets to write Amharic words. To an English-speaking person, these words will not make sense. However, the effect of this is lowered because the movie dialogues are in plain English.

Familiarizing myself to GCP was another obstacle. To start training my chatbot models, I started with a pre-trained English to French language translation model that used seq2seq. Getting used to this model wasn't easy since I didn't write the code. Moreover, the repo hasn't been updated for over a year and some TensorFlow functions were outdated. Another thing is the time ML Engine took to train these models. I hadn't anticipated it would take as long as it did.

Once the models started training and some checkpoints were produced, I attempted to test them out. Running the model on my local machine was quite difficult (the GCP console kept on running out of memory). The problems I ran into range from missing simple Python packages, to index and directory errors, to unknown errors that cased Python to crash. I also found that things run smoother in Linux than Windows. However, the more I worked on the project, the better I understood the whole concept.

7. Conclusion

As can be seen from the results (see section 5), I was successfully able to train a chatbot by mostly using my personal data. Although it can't possibly fool a human into thinking they're speaking with another person to pass the Turing test, it shows that we can train a bot using data of our choosing. This means that a bot could be trained for a specific purpose. For example, companies can use their frequently asked questions and responses data to train a chatbot so that when other customers have similar questions, they can ask the bot and it'll display a relevant response.

8. Future Works

The final chatbot model is trained with the most data, and we see some responses. The responses, however, don't really make sense. Therefore, the next step would be to get more and better training data from other sources. As mentioned in section 4, I've split the training data into train and test so the quality of the output would be better. One other thing to try is to further split the data into training, testing and validation for a better model.

Once a satisfactory model is trained, it could be deployed using GCP. Deploying this model will allow users to interact with it in website form. Although I planned to do this for my chatbot, I was facing difficulties mostly because of the way the language translation decoder was set up and the total size of my final model. I was also constrained by time.

Another feature that could be implemented is to train a retrieval based chatbot model as opposed to a generative one. A retrieval based a model would require a repository of pre-defined responses it could use. Moreover, the input to this model would be a context or the conversation until that point and the output would be a response to the context. This makes it smarter. Generative models on the other hand do not need to have repositories and can generate words they haven't seen before. In practice, retrieval based models work better and tend to make less grammatical errors since they're limited in how they can respond.

9. References

1. https://www.cs.cornell.edu/~cristian/Cornell_Movie-Dialogs_Corpus.html
2. https://github.com/tensorflow/nmt
3. arXiv:1406.1078v3 [cs.CL]
4. https://www.tensorflow.org/tutorials/seq2seq
5. https://github.com/tensorflow/tensorflow/tree/r1.2/tensorflow/examples/tutorials/word2vec
6. https://github.com/tensorflow/tensorflow/tree/r0.11/tensorflow/models/rnn/translate
7. https://github.com/ownaginatious/fbchat-archive-parser
8. https://gitlab.com/stackpointer/whatsapp-crypt12
9. https://play.google.com/store/apps/details?id=com.zegoggles.smssync&hl=en_US
10. https://takeout.google.com/settings/takeout
11. https://github.com/jarrodparkes/mbox-to-csv
12. https://github.com/b0noI/dialog_converter
13. https://cloud.google.com/ml-engine/docs/technical-overview

10. Other resources

1. http://adventuresinmachinelearning.com/word2vec-tutorial-tensorflow/
2. http://suriyadeepan.github.io/2016-06-28-easy-seq2seq/