Chatbots with Seq2Seq

Learn to build a chatbot using TensorFlow

Posted on June 28, 2016

Update 01.01.2017 Part II of Sequence to Sequence Learning is available - Practical seq2seq

Last year, Telegram released its bot API, providing an easy way for developers, to create bots by interacting with a bot, the Bot Father. Immediately people started creating abstractions in nodejs, ruby and python, for building bots. We (Free Software Community) created a group for interacting with the bots we built. I created Myshkin in nodejs that answers any query with a quote. The program uses the linux utility fortune, a pseudorandom message generator. It was dumb. But it was fun to see people willingly interact with a program that I've created. Someone made a **Hodor bot**. You probably figured out what it does. Then I encountered another bot, Mitsuku which seemed quite intelligent. It is written in **AIML** (Artificial Intelligence Markup Language); an XML based "language" that lets developers write rules for the bot to follow. Basically, you write a PATTERN and a TEMPLATE, such that when the bot encounters that pattern in a sentence from user, it replies with one of the templates. Let us call this model of bots, **Rule based model**.

Rule based models make it easy for anyone to create a bot. But it is incredibly difficult to create a bot that answers complex queries. The pattern matching is kind of weak and hence, AIML based bots suffer when they encounter a sentence that doesn't contain any known patterns. Also, it is time consuming and takes a lot of effort to write the rules manually. What if we can build a bot that learns from existing conversations (between humans). This is where *Machine Learning* comes in.

Let us call these models that automatically learn from data, **Intelligent models**. The Intelligent models can be further classified into:

1. **Retrieval-based** models
2. **Generative** models

The Retrieval-based models pick a response from a collection of responses based on the query. It does not generate any new sentences, hence we don't need to worry about grammar. The Generative models are quite intelligent. They generate a response, word by word based on the query. Due to this, the responses generated are prone to grammatical errors. These models are difficult to train, as they need to learn the proper sentence structure by themselves. However, once trained, the generative models outperform the retrieval-based models in terms of handling previously unseen queries and create an impression of talking with a human (a toddler may be) for the user.

Read Deep Learning For Chatbots by Denny Britz where he talks about the length of conversations, open vs closed domain dialogs, challenges in generative models like Context based responses, Coherent Personality, understanding the Intention of user and how to evaluate these models.

Seq2Seq

Sequence To Sequence model introduced in Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation has since then, become the Go-To model for Dialogue Systems and Machine Translation. It consists of two RNNs (Recurrent Neural Network) : An Encoder and a Decoder. The encoder takes a sequence(sentence) as input and processes one symbol(word) at each timestep. Its objective is to convert a sequence of symbols into a fixed size feature vector that encodes only the important information in the sequence while losing the unnecessary information. You can visualize data flow in the encoder along the time axis, as the flow of local information from one end of the sequence to another.

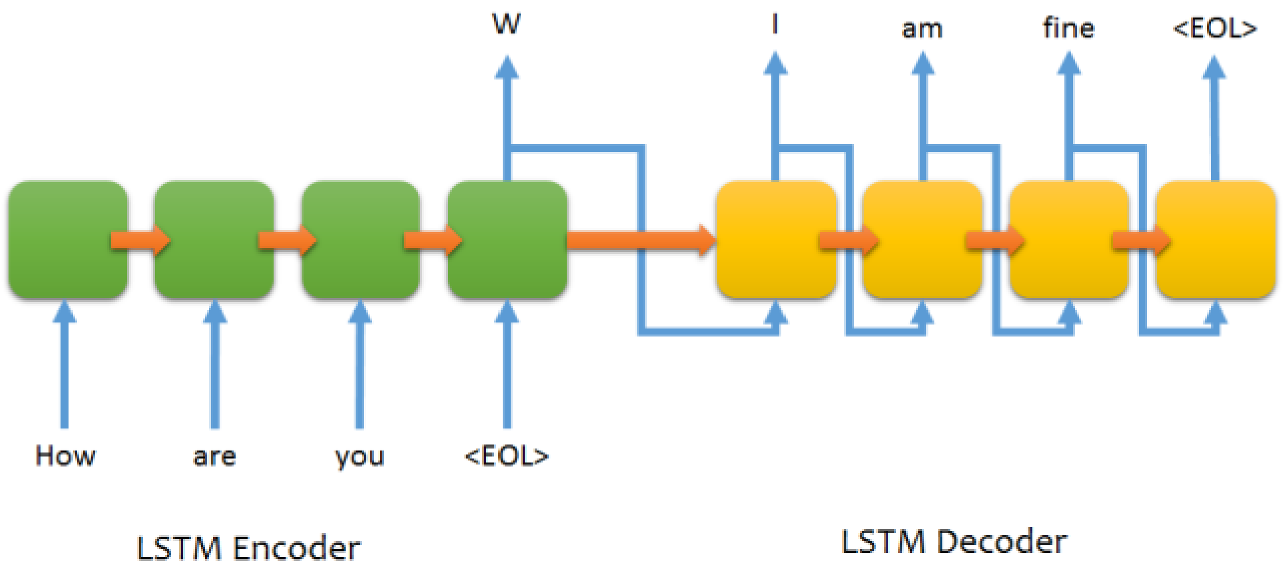


Image borrowed from farizrahman4u/seq2seq

Each hidden state influences the next hidden state and the final hidden state can be seen as the summary of the sequence. This state is called the context or thought vector, as it represents the intention of the sequence. From the context, the decoder generates another sequence, one symbol(word) at a time. Here, at each time step, the decoder is influenced by the context and the previously generated symbols.

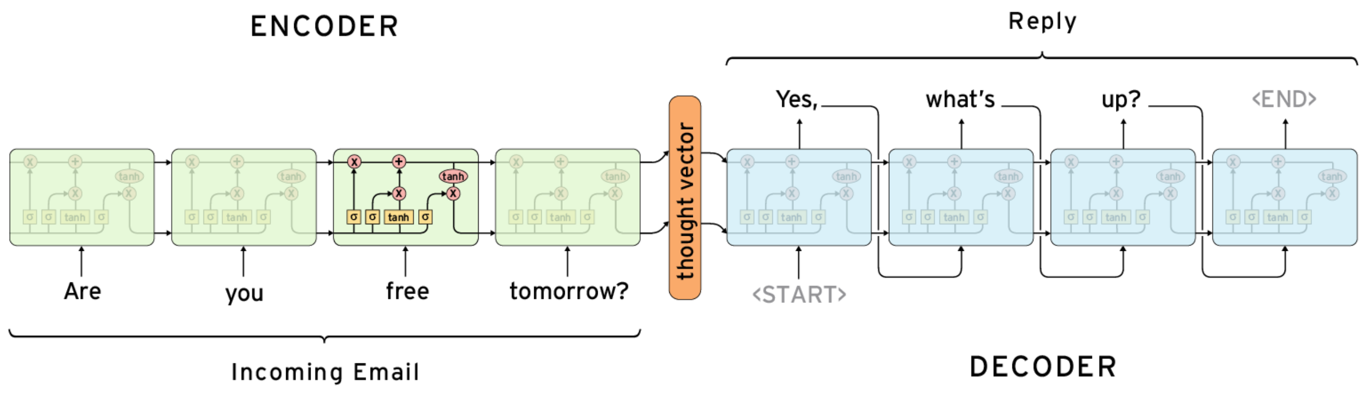


Image borrowed from Deep Learning for Chatbots : Part 1

There are a few challenges in using this model. The most disturbing one is that the model cannot handle variable length sequences. It is disturbing because almost all the sequence-to-sequence applications, involve variable length sequences. The next one is the vocabulary size. The decoder has to run softmax over a large vocabulary of say 20,000 words, for each word in the output. That is going to slow down the training process, even if your hardware is capable of handling it.

Representation of words is of great importance. How do you represent the words in the sequence? Use of one-hot vectors means we need to deal with large sparse vectors due to large vocabulary and there is no semantic meaning to words encoded into one-hot vectors. Lets look into how we can face these challenges, one by one.

Padding

Before training, we work on the dataset to convert the variable length sequences into fixed length sequences, by **padding**. We use a few special symbols to fill in the sequence.

1. **EOS** : End of sentence
2. **PAD** : Filler
3. **GO** : Start decoding
4. **UNK** : Unknown; word not in vocabulary

Consider the following query-response pair.

> **Q** : *How are you?*
> **A** : *I am fine.*

Assuming that we would like our sentences (queries and responses) to be of fixed length, **10**, this pair will be converted to:

> **Q** : **[** PAD, PAD, PAD, PAD, PAD, PAD, "?", "*you*", "*are*", "*How*" **]**
> **A** : **[** GO, "I", "*am*", "*fine*", ".", EOS, PAD, PAD, PAD, PAD **]**

Bucketing

Introduction of padding did solve the problem of variable length sequences, but consider the case of large sentences. If the largest sentence in our dataset is of length 100, we need to encode all our sentences to be of length 100, in order to not lose any words. Now, what happens to “How are you?” ? There will be 97 PAD symbols in the encoded version of the sentence. This will overshadow the actual information in the sentence.

Bucketing kind of solves this problem, by putting sentences into buckets of different sizes. Consider this list of buckets : **[** (5,10), (10,15), (20,25), (40,50) **]**. If the length of a query is 4 and the length of its response is 4 (as in our previous example), we put this sentence in the bucket (5,10). The query will be padded to length 5 and the response will be padded to length 10. While running the model (training or predicting), we use a different model for each bucket, compatible with the lengths of query and response. All these models, share the same parameters and hence function exactly the same way.

If we are using the bucket (5,10), our sentences will be encoded to :

> **Q** : **[** PAD, “?”, “*you*”, “*are*”, “*How*” **]**
> **A** : **[** GO, “*I*”, “*am*”, “*fine*”, “*.*”, EOS, PAD, PAD, PAD, PAD **]**

Word Embedding

Word Embedding is a technique for learning dense representation of words in a low dimensional vector space. Each word can be seen as a point in this space, represented by a fixed length vector. Semantic relations between words are captured by this technique. The **word vectors** have some interesting properties.

> *paris – france + poland = warsaw.*

The vector difference between paris and france captures the concept of capital city.

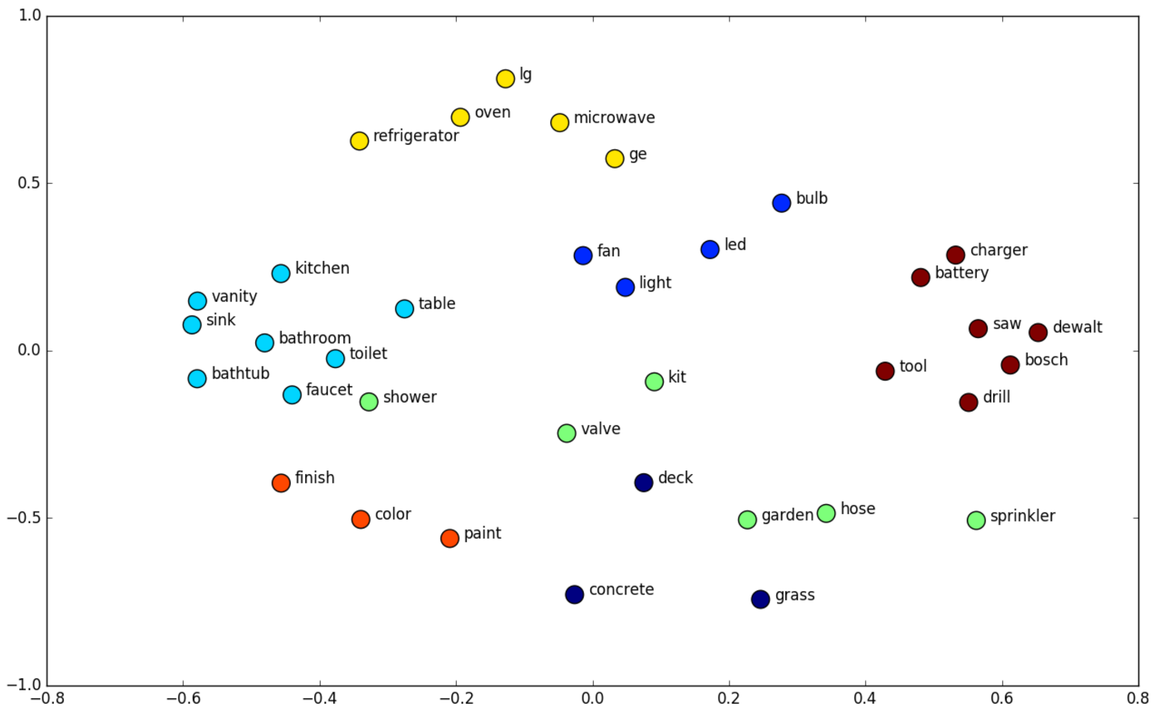


Image borrowed from Home Depot Product Search Relevance, Winners' Interview

Word Embedding is typically done in the first layer of the network : Embedding layer, that maps a word (index to word in vocabulary) from vocabulary to a dense vector of given size. In the seq2seq model, the weights of the embedding layer are jointly trained with the other parameters of the model. Follow this tutorial by Sebastian Ruder to learn about different models used for word embedding and its importance in NLP.

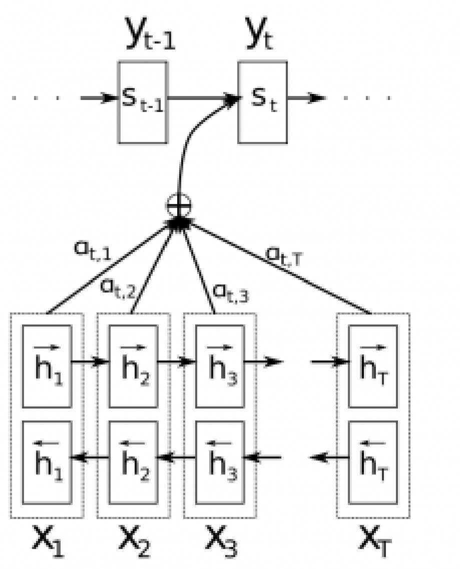
Papers on Sequence to Sequence

1. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation
2. Sequence to Sequence Learning with Neural Networks
3. Neural Machine Translation by Jointly Learning to Align and Translate
4. A Neural Conversational Model

Attention Mechanism

One of the limitations of seq2seq framework is that the entire information in the input sentence should be encoded into a fixed length vector, **context**. As the length of the sequence gets larger, we start losing considerable amount of information.

This is why the basic seq2seq model doesn't work well in decoding large sequences. The attention mechanism, introduced in this paper, Neural Machine Translation by Jointly Learning to Align and Translate, allows the decoder to selectively look at the input sequence while decoding. This takes the pressure off the encoder to encode every useful information from the input.



y_{t-1}
y_t
s_{t-1}
s_t
$a_{t,1}$
$a_{t,2}$
$a_{t,3}$
$a_{t,T}$
$\overrightarrow{h}_1$ $\overrightarrow{h}_2$ $\overrightarrow{h}_3$ $\overrightarrow{h}_T$
$\overleftarrow{h}_1$ $\overleftarrow{h}_2$ $\overleftarrow{h}_3$ $\overleftarrow{h}_T$
x_1 x_2 x_3 x_T

Image borrowed from Neural Machine Translation by Jointly Learning to Align and Translate

How does it work? During each timestep in the decoder, instead of using a fixed context (last hidden state of encoder), a distinct context vector \(c_i\) is used for generating word \(y_i\). This context vector \(c_i\) is basically the weighted sum of hidden states of the encoder.

\[c_i = \sum_{j=1}^{n} \alpha_{ij}h_{j}\]

where *n* is the length of input sequence, \(h_j\) is the hidden state at timestep *j*.

\[\alpha_{ij} = \exp(e_{ij}) / \sum_{k=1}^{n} \exp(e_{ik})\]

\(e_{ij}\) is the alignment model which is function of decoder's previous hidden state \(s_{i-1}\) and the jth hidden state of the encoder. This alignment model is parameterized as a feedforward neural network which is jointly trained with the rest of model.

Each hidden state in the encoder encodes information about the local context in that part of the sentence. As data flows from word 0 to word n, this local context information gets diluted. This makes it necessary for the decoder to peak through the encoder, to know the local contexts. Different parts of input sequence contain information necessary for generating different parts of the output sequence. In other words, each word in the output sequence is aligned to different parts of the input sequence. The alignment model gives us a measure of how well the output at position *i* match with inputs at around postion *j*. Based on which, we take a weighted sum of the input contexts (hidden states) to generate each word in the ouput sequence.

Code

The code is available in this repository, suriyadeepan/easy_seq2seq. It is based on Tensorflow's English-French translation example. I borrowed the files *data_utils.py*, *seq2seq_model.py* and *translate.py* from it. *data_utils.py* consists of functions for preprocessing the dataset. We will be working with Cornell Movie Dialog Corpus. The function *prepare_custom_data( )* takes as input, files containing questions and responses, separated into training and test data. Read the comments for a detailed description.

```
'''
        converts raw sentences into token id's and returns
                the path to vocabulary and tokenzied sentences

working_directory : directory where vocabulary files will be stored
train_enc : encoder input for training (X_train)
train_dec : decoder input for training (Y_train)
test_enc : encoder input for evaluation (X_test)
test_dec : decoder input for evaluation (Y_test)
enc_vocabulary_size : size of vocabulary on encoder side (I choose 200
00)
dec_vocabulary_size : size of vocabulary on decoder side (same here)
tokenizer : None - uses basic_tokenizer function in data_utils.py
'''
def prepare_custom_data(working_directory, train_enc, train_dec, test_
enc, test_dec, enc_vocabulary_size, dec_vocabulary_size, tokenizer=Non
e):
        # Create vocabularies of the appropriate sizes.
        enc_vocab_path = os.path.join(working_directory,
                "vocab%d.enc" % enc_vocabulary_size)
        dec_vocab_path = os.path.join(working_directory,
                "vocab%d.dec" % dec_vocabulary_size)
        create_vocabulary(enc_vocab_path, train_enc,
                 enc_vocabulary_size, tokenizer)
        create_vocabulary(dec_vocab_path, train_dec,
                dec_vocabulary_size, tokenizer)

        # Create token ids for the training data.
        enc_train_ids_path = train_enc + (".ids%d" % enc_vocabulary_si
ze)
        dec_train_ids_path = train_dec + (".ids%d" % dec_vocabulary_si
ze)
        data_to_token_ids(train_enc, enc_train_ids_path,
                 enc_vocab_path, tokenizer)
        data_to_token_ids(train_dec, dec_train_ids_path,
                 dec_vocab_path, tokenizer)

        # Create token ids for the development data.
        enc_dev_ids_path = test_enc + (".ids%d" % enc_vocabulary_size)
        dec_dev_ids_path = test_dec + (".ids%d" % dec_vocabulary_size)
        data_to_token_ids(test_enc, enc_dev_ids_path,
                enc_vocab_path, tokenizer)
        data_to_token_ids(test_dec, dec_dev_ids_path,
                dec_vocab_path, tokenizer)

        return (enc_train_ids_path, dec_train_ids_path, enc_dev_ids_pa
th,
                dec_dev_ids_path, enc_vocab_path, dec_vocab_path)
```

I have renamed *translate.py* to *execute.py* and modified the *train( )* to use our *prepare_custom_data( )* function. Instead of passing arguments as flags, I've used **ConfigParser** to read from *seq2seq.ini*, which contains various options to configure the model, like the size of vocabulary, number of layers of LSTM, etc,. The next section explains all the configurations in detail. The file *seq2seq_model.py* remains unchanged.

Bootstrapping easy_seq2seq

We are using Cornell Movie Dialog Corpus for training our model. The preprocessed dataset is available here, which you can get by running the script *pull_data.sh* available at the *data/* folder. But you might want to preprocess it yourself in order to modify the number of sentences in training and test set. You can make use of *prepare_data.py* script, available here for preprocessing the raw corpus. It generates 4 files containing queries and replies, for training and testing.

Configuration

Switch	Purpose	Default
mode	train, test (interactive sessin), serve (Flask app)	test
train_enc	encoder inputs file for training (X_train)	data/train.enc
train_dec	decoder inputs file for training (Y_train)	data/train.dec
test_enc	encoder inputs file for testing (X_test)	data/test.enc
test_dec	decoder inputs file for testing (Y_test)	data/test.dec
working_directory	folder where checkpoints, vocabulary, temporary data will be stored	data/
pretrained_model	previously trained model saved to file	-
enc_vocab_size	encoder vocabulary size	20000
dec_vocab_size	decoder vocabulary size	20000
num_layers	number of layers	2
layer_size	number of units in a layer	512
max_train_data_size	limit count of training data	0 (no limit)

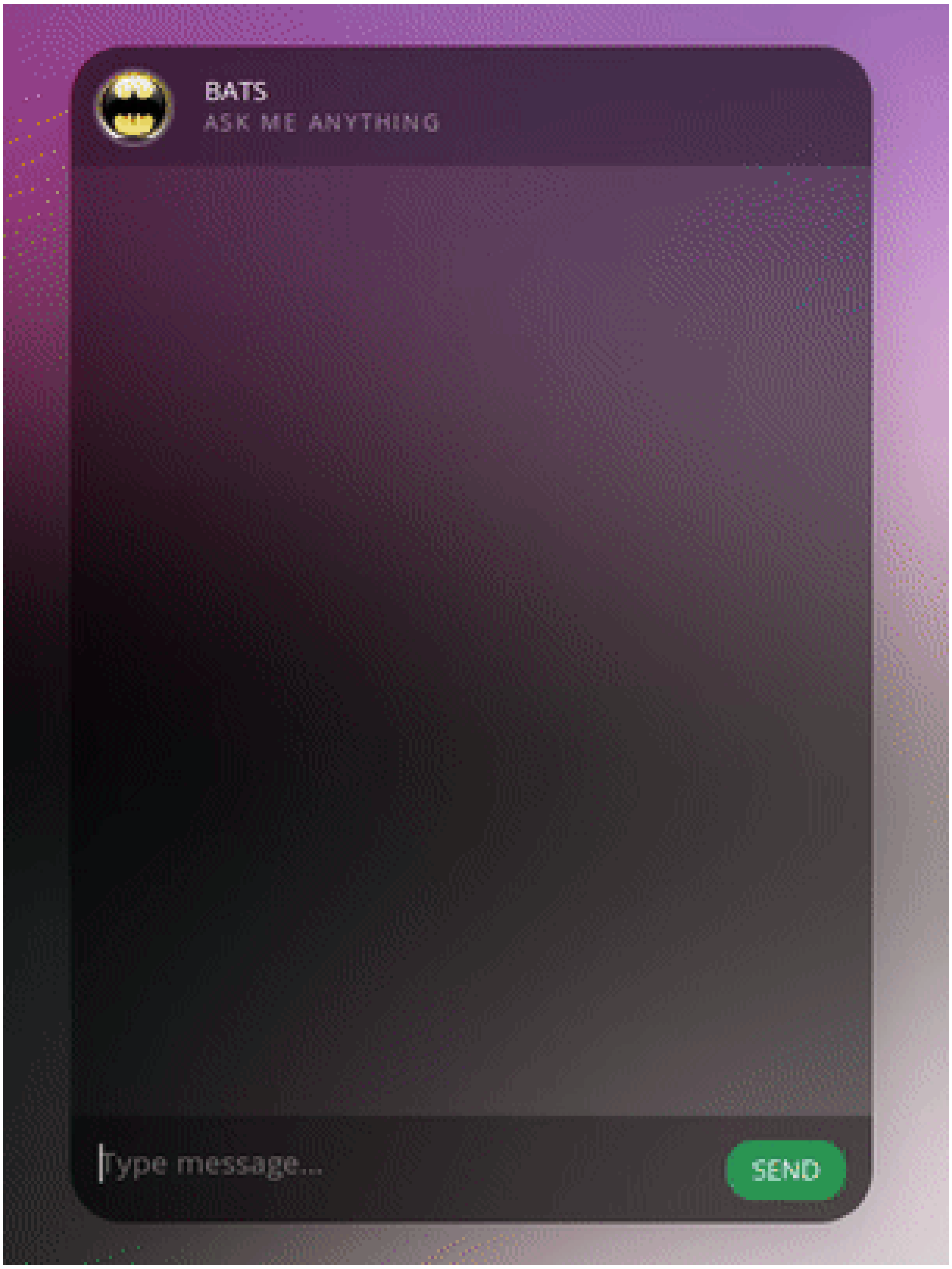
Switch	Purpose	Default
batch_size	batch size for training; modify this based on your hardware specs	64
steps_per_checkpoint	At a checkpoint, parameters are saved, model is evaluated	200
learning_rate	Learning rate	0.5
learning_rate_decay_factor	Learning rate decay factor	0.99
max_gradient_norm	Gradient clipping threshold	5.0

Web Interface

Flask : Quick Start should help you setup flask in virtual environment. I've built a tiny Flask app that provides a chat interface to the user, to interact with our *seq2seq* model. It is so tiny, that I'm putting it here as a code snippet. The reply method in the code is called via an AJAX request from index.js. It sends the text from user to our seq2seq model via the decode_line method, which returns a reply. The reply is passed to *index.js* and rendered as text in the chatbox.

```
from flask import Flask, render_template, request
from flask import jsonify

app = Flask(__name__,static_url_path="/static")

# Routing
@app.route('/message', methods=['POST'])
def reply():
    return jsonify( { 'text': execute.decode_line(sess, model, enc_voc
ab, rev_dec_vocab, request.form['msg'] ) } )

@app.route("/")
def index():
    return render_template("index.html")

# Init seq2seq model
import tensorflow as tf
import execute
```

```
sess = tf.Session()
sess, model, enc_vocab, rev_dec_vocab = execute.init_session(sess)

# start app
if (__name__ == "__main__"):
    app.run(port = 5000)
```



Reference

- Telegram Bots : An introduction for developers
- Botfather
- Mitsuku
- Deep Learning for Chatbots : Introduction
- Understanding LSTMS
- Padding and Bucketing
- On word embeddings - Part 1
- On word embeddings - Part 2 (sampled softmax)
- Attention and Memory in Deep Learning and NLP
- easy_seq2seq
- English-Frnch Translation in Tensorflow
- Cornell Movie Dialog Corpus
- Cornell Movie Dialog Corpus - Preprocessed
- Flask : Quick Start

Part II of Sequence to Sequence Learning is available - Practical seq2seq

• tensorflow • machine learning • seq2seq • NLP

← PREVIOUS POST

NEXT POST →