Chat Bot

Deep Learning Application

Kalash Jain (*Author*)
Computer Science Department
Wayne State University
Detroit, Michigan, USA
kalash.jain@wayne.edu

***Abstract*—Chatbot are the most popular and powerful application of Machine Learning. The task is to make the bot more human like and pass the Turing test. This paper is about designing of a chatbot using Deep Learning and python's Tensorflow library.**

Keywords—Tensorflow, Sequence to Sequence Model, Recurrent Neural Network, Chatbot.

I. BACKGROUND INTRODUCTION

Chat bots are a very powerful application of Artificial Intelligence and Machine Learning. Now a day, the reach of machine learning is infinite and the new concepts and advancements in the technology have eased the process of making the model and designing the chatbot. The hot topic is Deep Learning which deals with the challenges of Machine Learning and propose models and methods that are efficient in generating output and reducing the errors.

II. METHODS

A. Model

Sequence to Sequence model by Tensorflow is used in the designing of the chat bot. A basic sequence to sequence model uses two recurrent neural networks, one out of that is encoder and other is decoder. Encoder is used for processing the input and the decoder is used for generating the output. In the sequence to sequence model multi-layer cells can be successfully implemented and the use of LSTM (Long Short Term Memory) have been used as the bot will be used for conversations and it will require to remember the sentences and the previous some conversations.

B. Technology and Language

To build the chatbot Tensorflow library in python is used. Python is a very powerful and efficient language for machine learning and with the advantage of having the tensorflow library, the process of designing the bot becomes even easy. There are already made modules and functions for using. Along with tensorflow, numpy, tflearn is also used in the project.

C. Dataset

The dataset that is used for the implementation is named as "Cornell movie--Dialogs Corpus". This movie data contains:

This corpus contains a large metadata-rich collection of fictional conversations extracted from raw movie scripts:

- 220,579 conversational exchanges between 10,292 pairs of movie characters
- involves 9,035 characters from 617 movies
- in total 304,713 utterances
- movie metadata included:
  - genres
  - release year
  - IMDB rating
  - number of IMDB votes
  - IMDB rating
- character metadata included:
  - gender (for 3,774 characters)
  - position on movie credits (3,321 characters)

III. RESULTS AND DISCUSSION

There are two stages of the chatbot, the train phase and the test phase. In the train phase the bot is set to train on the input provided which is the Cornell movie dataset. With a lot of training that initially started with the gibberish output, the chatbot finally completed its round of training and was ready to test or say chat. As the bot is trained on the movie data then it is more than obvious to find the artificial agent give some filmy answers to the questions. I am giving an example of the conversation with the bot.

Me: hi..

Bot: hi! how are you?

Me: i am good.

Bot: i am okay

Me: what's your name?

Bot: let’s talk about something else.

Me: how old are you?

Bot: can you guess?

Me: you are boring!

Bot: i am not.

Me: do you drink alcohol?

Bot: in time.

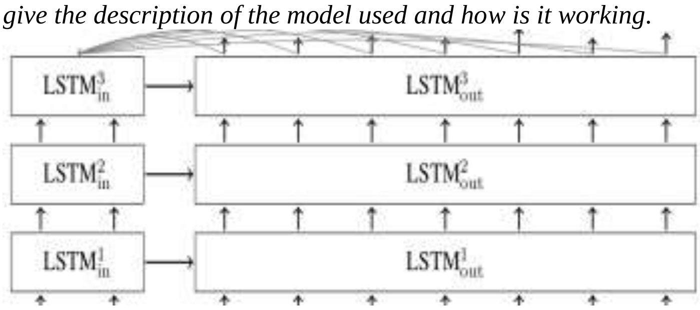
IV. CONCLUSION

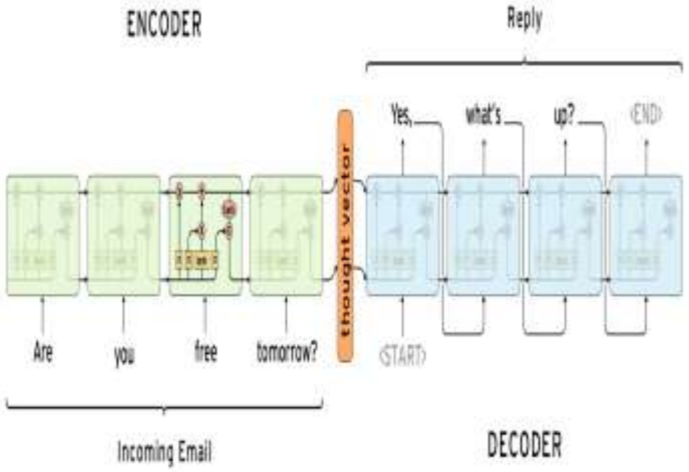
After the training and the testing stages of the chatbot, the conclusion is that the bot is doing well but not so well. For better results and conversations, the idea is to train the bot with more data that is of different variety, something like twitter conversations or blogs written by people, something like this. Although there lies the risk of the bot getting confused and performing even worse or talking more nonsense.

In future, I am to give the bot some kind of a back story with which the bot will be having its own personality and view towards the world and the people that are talking to it and in that way, the conversations will be affected by that back story. I am working on this idea and hopeful to implement it.

A. *Figures and Tables*

1) Sequence to Sequence Model: The following images give the description of the model used and how is it working.


LSTM^{3}_{in}
LSTM^{3}_{out}
LSTM^{2}_{in}
LSTM^{2}_{out}
LSTM^{1}_{in}
LSTM^{1}_{out}



BIBLIOGRAPHY

[1] https://www.tensorflow.org/tutorials/seq2seq

[2] https://chatbotsmagazine.com/the-complete-beginner-s-guide-to-chatbots-8280b7b906ca

[3] https://github.com/codedak/chatbot

[4] http://www.wildml.com/2016/04/deep-learning-for-chatbots-part-1-introduction/

[5] https://chatbotslife.com/ultimate-guide-to-leveraging-nlp-machine-learning-for-you-chatbot-531ff2dd870c

[6] http://web.stanford.edu/class/cs20si/lectures/slides_13.pdf

[7] http://lauragelston.ghost.io/speakeasy-pt2/

[8] https://github.com/chiphuyen/tf-stanford-tutorials

[9] https://www.youtube.com/watch?v=SJDEOWLHYVo&t=322s

[10] https://github.com/llSourcell/tensorflow_chatbot

[11] http://web.stanford.edu/class/cs124/lec/chatbot.pdf

[12] https://venturebeat.com/2016/08/01/how-deep-reinforcement-learning-can-help-chatbots/

APPENDIX

The source code for the bot is as follows:

```
# Copyright 2015 The TensorFlow Authors. All Rights Reserved.

#

# Licensed under the Apache License, Version 2.0 (the "License");

# you may not use this file except in compliance with the License.

# You may obtain a copy of the License at

#

#     http://www.apache.org/licenses/LICENSE-2.0

#

# Unless required by applicable law or agreed to in writing, software

# distributed under the License is distributed on an "AS IS" BASIS,

# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

# See the License for the specific language governing permissions and

# limitations under the License.

# ==============================================================================


from __future__ import absolute_import

from __future__ import division

from __future__ import print_function
```

```
import math
import os
import random
import sys
import time

import numpy as np
from six.moves import xrange  # pylint: disable=redefined-builtin
import tensorflow as tf

import data_utils
import seq2seq_model

# python2 and python3 support
try:
    reload
except NameError:
    # py3k has unicode by default
    pass
else:
    reload(sys).setdefaultencoding('utf-8')

try:
    from ConfigParser import ConfigParser
except:
    from configparser import ConfigParser # In Python 3, ConfigParser has been renamed to configparser for PEP 8 compliance.

gConfig = {}

def get_config(config_file='seq2seq.ini'):
    parser = ConfigParser()
    parser.read(config_file)
    # get the ints, floats and strings
    _conf_ints = [ (key, int(value)) for key,value in parser.items('ints') ]
    _conf_floats = [ (key, float(value)) for key,value in parser.items('floats') ]
    _conf_strings = [ (key, str(value)) for key,value in parser.items('strings') ]
    return dict(_conf_ints + _conf_floats + _conf_strings)

# We use a number of buckets and pad to the closest one for efficiency.
# See seq2seq_model.Seq2SeqModel for details of how they work.
_buckets = [(5, 10), (10, 15), (20, 25), (40, 50)]


def read_data(source_path, target_path, max_size=None):
  """Read data from source and target files and put into buckets.

  Args:
    source_path: path to the files with token-ids for the source language.
    target_path: path to the file with token-ids for the target language;
      it must be aligned with the source file: n-th line contains the desired
      output for n-th line from the source_path.
    max_size: maximum number of lines to read, all other will be ignored;
      if 0 or None, data files will be read completely (no limit).

  Returns:
    data_set: a list of length len(_buckets); data_set[n] contains a list of
      (source, target) pairs read from the provided data files that fit
      into the n-th bucket, i.e., such that len(source) < _buckets[n][0] and
      len(target) < _buckets[n][1]; source and target are lists of token-ids.
  """
  data_set = [[] for _ in _buckets]
  with tf.gfile.GFile(source_path, mode="r") as source_file:
    with tf.gfile.GFile(target_path, mode="r") as target_file:
      source, target = source_file.readline(), target_file.readline()
      counter = 0
      while source and target and (not max_size or counter <
max_size):
        counter += 1
        if counter % 100000 == 0:
          print("  reading data line %d" % counter)
          sys.stdout.flush()
        source_ids = [int(x) for x in source.split()]
        target_ids = [int(x) for x in target.split()]
        target_ids.append(data_utils.EOS_ID)
        for bucket_id, (source_size, target_size) in
enumerate(_buckets):
          if len(source_ids) < source_size and len(target_ids) <
target_size:
            data_set[bucket_id].append([source_ids, target_ids])
            break
        source, target = source_file.readline(),
target_file.readline()
  return data_set



def create_model(session, forward_only):


  """Create model and initialize or load parameters"""
  model = seq2seq_model.Seq2SeqModel(
gConfig['enc_vocab_size'], gConfig['dec_vocab_size'],
_buckets, gConfig['layer_size'], gConfig['num_layers'],
gConfig['max_gradient_norm'], gConfig['batch_size'],
gConfig['learning_rate'],
gConfig['learning_rate_decay_factor'],
forward_only=forward_only)


  if 'pretrained_model' in gConfig:

model.saver.restore(session,gConfig['pretrained_model'])
      return model


  ckpt =
tf.train.get_checkpoint_state(gConfig['working_directory'])
  # the checkpoint filename has changed in recent versions
of tensorflow
  checkpoint_suffix = ""
  if tf.__version__ > "0.12":
      checkpoint_suffix = ".index"
```

```
  if ckpt and tf.gfile.Exists(ckpt.model_checkpoint_path +
checkpoint_suffix):
    print("Reading model parameters from %s" %
ckpt.model_checkpoint_path)
    model.saver.restore(session,
ckpt.model_checkpoint_path)
  else:
    print("Created model with fresh parameters.")
    session.run(tf.initialize_all_variables())
  return model



def train():
  # prepare dataset
  print("Preparing data in %s" %
gConfig['working_directory'])
  enc_train, dec_train, enc_dev, dec_dev, _, _ =
data_utils.prepare_custom_data(gConfig['working_directory'],
gConfig['train_enc'],gConfig['train_dec'],gConfig['test_enc'],g
Config['test_dec'],gConfig['enc_vocab_size'],gConfig['dec_voc
ab_size'])


  # Only allocate 2/3 of the gpu memory to allow for
running gpu-based predictions while training:
  gpu_options =
tf.GPUOptions(per_process_gpu_memory_fraction=0.666)
  config = tf.ConfigProto(gpu_options=gpu_options)
  config.gpu_options.allocator_type = 'BFC'


  with tf.Session(config=config) as sess:
    # Create model.
    print("Creating %d layers of %d units." %
(gConfig['num_layers'], gConfig['layer_size']))
    model = create_model(sess, False)


    # Read data into buckets and compute their sizes.
    print ("Reading development and training data (limit:
%d)."
           % gConfig['max_train_data_size'])
    dev_set = read_data(enc_dev, dec_dev)
    train_set = read_data(enc_train, dec_train,
gConfig['max_train_data_size'])
    train_bucket_sizes = [len(train_set[b]) for b in
xrange(len(_buckets))]
```

```
    train_total_size = float(sum(train_bucket_sizes))


    # A bucket scale is a list of increasing numbers from 0 to 1 that we'll use
    # to select a bucket. Length of [scale[i], scale[i+1]] is proportional to
    # the size if i-th training bucket, as used later.
    train_buckets_scale = [sum(train_bucket_sizes[:i + 1]) / train_total_size
                           for i in xrange(len(train_bucket_sizes))]


    # This is the training loop.
    step_time, loss = 0.0, 0.0
    current_step = 0
    previous_losses = []
    while True:
      # Choose a bucket according to data distribution. We pick a random number
      # in [0, 1] and use the corresponding interval in train_buckets_scale.
      random_number_01 = np.random.random_sample()
      bucket_id = min([i for i in xrange(len(train_buckets_scale))
                       if train_buckets_scale[i] > random_number_01])


      # Get a batch and make a step.
      start_time = time.time()
      encoder_inputs, decoder_inputs, target_weights = model.get_batch(
          train_set, bucket_id)
      _, step_loss, _ = model.step(sess, encoder_inputs, decoder_inputs,
                                   target_weights, bucket_id, False)
      step_time += (time.time() - start_time) / gConfig['steps_per_checkpoint']
      loss += step_loss / gConfig['steps_per_checkpoint']
      current_step += 1


      # Once in a while, we save checkpoint, print statistics, and run evals.
      if current_step % gConfig['steps_per_checkpoint'] == 0:
        # Print statistics for the previous epoch.
        perplexity = math.exp(loss) if loss < 300 else float('inf')
        print ("global step %d learning rate %.4f step-time %.2f perplexity "
               "%.2f" % (model.global_step.eval(), model.learning_rate.eval(),
                         step_time, perplexity))
        # Decrease learning rate if no improvement was seen over last 3 times.
        if len(previous_losses) > 2 and loss > max(previous_losses[-3:]):
          sess.run(model.learning_rate_decay_op)
        previous_losses.append(loss)
        # Save checkpoint and zero timer and loss.
        checkpoint_path = os.path.join(gConfig['working_directory'], "seq2seq.ckpt")
        model.saver.save(sess, checkpoint_path, global_step=model.global_step)
        step_time, loss = 0.0, 0.0
        # Run evals on development set and print their perplexity.
        for bucket_id in xrange(len(_buckets)):
          if len(dev_set[bucket_id]) == 0:
            print("  eval: empty bucket %d" % (bucket_id))
            continue
          encoder_inputs, decoder_inputs, target_weights = model.get_batch(
              dev_set, bucket_id)
          _, eval_loss, _ = model.step(sess, encoder_inputs, decoder_inputs,
                                       target_weights, bucket_id, True)
          eval_ppx = math.exp(eval_loss) if eval_loss < 300 else float('inf')
          print("  eval: bucket %d perplexity %.2f" % (bucket_id, eval_ppx))
        sys.stdout.flush()



def decode():


  # Only allocate part of the gpu memory when predicting.
  gpu_options = tf.GPUOptions(per_process_gpu_memory_fraction=0.2)
  config = tf.ConfigProto(gpu_options=gpu_options)
  with tf.Session(config=config) as sess:
    # Create model and load parameters.
    model = create_model(sess, True)
    model.batch_size = 1  # We decode one sentence at a time.

    # Load vocabularies.
    enc_vocab_path = os.path.join(gConfig['working_directory'],"vocab%d.enc" % gConfig['enc_vocab_size'])
    dec_vocab_path = os.path.join(gConfig['working_directory'],"vocab%d.dec" % gConfig['dec_vocab_size'])

    enc_vocab, _ = data_utils.initialize_vocabulary(enc_vocab_path)
    _, rev_dec_vocab = data_utils.initialize_vocabulary(dec_vocab_path)

    # Decode from standard input.
    sys.stdout.write("> ")
    sys.stdout.flush()
    sentence = sys.stdin.readline()
    while sentence:
      # Get token-ids for the input sentence.
      token_ids = data_utils.sentence_to_token_ids(tf.compat.as_bytes(sentence), enc_vocab)
      # Which bucket does it belong to?
      bucket_id = min([b for b in xrange(len(_buckets))
                       if _buckets[b][0] > len(token_ids)])
      # Get a 1-element batch to feed the sentence to the model.
      encoder_inputs, decoder_inputs, target_weights = model.get_batch(
          {bucket_id: [(token_ids, [])]}, bucket_id)
      # Get output logits for the sentence.
      _, _, output_logits = model.step(sess, encoder_inputs, decoder_inputs,
                                       target_weights, bucket_id, True)
      # This is a greedy decoder - outputs are just argmaxes of output_logits.
      outputs = [int(np.argmax(logit, axis=1)) for logit in output_logits]
      # If there is an EOS symbol in outputs, cut them at that point.
      if data_utils.EOS_ID in outputs:
        outputs = outputs[:outputs.index(data_utils.EOS_ID)]
      # Print out French sentence corresponding to outputs.
      print(" ".join([tf.compat.as_str(rev_dec_vocab[output]) for output in outputs]))
      print("> ", end="")
      sys.stdout.flush()
      sentence = sys.stdin.readline()



def self_test():
  """Test the translation model."""
  with tf.Session() as sess:
    print("Self-test for neural translation model.")
    # Create model with vocabularies of 10, 2 small buckets, 2 layers of 32.
    model = seq2seq_model.Seq2SeqModel(10, 10, [(3, 3), (6, 6)], 32, 2,
                                       5.0, 32, 0.3, 0.99, num_samples=8)
    sess.run(tf.initialize_all_variables())

    # Fake data set for both the (3, 3) and (6, 6) bucket.
    data_set = ([([1, 1], [2, 2]), ([3, 3], [4]), ([5], [6])],
                [([1, 1, 1, 1, 1], [2, 2, 2, 2, 2]), ([3, 3, 3], [5, 6])])
    for _ in xrange(5):  # Train the fake model for 5 steps.
      bucket_id = random.choice([0, 1])
      encoder_inputs, decoder_inputs, target_weights = model.get_batch(
          data_set, bucket_id)
      model.step(sess, encoder_inputs, decoder_inputs, target_weights,
                 bucket_id, False)



def init_session(sess, conf='seq2seq.ini'):
    global gConfig
    gConfig = get_config(conf)
```

```
        # Create model and load parameters.
        model = create_model(sess, True)
        model.batch_size = 1  # We decode one sentence at a time.


        # Load vocabularies.
        enc_vocab_path = os.path.join(gConfig['working_directory'],"vocab%d.enc" % gConfig['enc_vocab_size'])
        dec_vocab_path = os.path.join(gConfig['working_directory'],"vocab%d.dec" % gConfig['dec_vocab_size'])


        enc_vocab, _ = data_utils.initialize_vocabulary(enc_vocab_path)
        _, rev_dec_vocab = data_utils.initialize_vocabulary(dec_vocab_path)


        return sess, model, enc_vocab, rev_dec_vocab


    def decode_line(sess, model, enc_vocab, rev_dec_vocab, sentence):
        # Get token-ids for the input sentence.
        token_ids = data_utils.sentence_to_token_ids(tf.compat.as_bytes(sentence), enc_vocab)


        # Which bucket does it belong to?
        bucket_id = min([b for b in xrange(len(_buckets)) if _buckets[b][0] > len(token_ids)])


        # Get a 1-element batch to feed the sentence to the model.
        encoder_inputs, decoder_inputs, target_weights = model.get_batch({bucket_id: [(token_ids, [])]}, bucket_id)


        # Get output logits for the sentence.
        _, _, output_logits = model.step(sess, encoder_inputs, decoder_inputs, target_weights, bucket_id, True)


        # This is a greedy decoder - outputs are just argmaxes of output_logits.
        outputs = [int(np.argmax(logit, axis=1)) for logit in output_logits]
        # If there is an EOS symbol in outputs, cut them at that point.
        if data_utils.EOS_ID in outputs:
            outputs = outputs[:outputs.index(data_utils.EOS_ID)]


        return " ".join([tf.compat.as_str(rev_dec_vocab[output]) for output in outputs])


    if __name__ == '__main__':
        if len(sys.argv) - 1:
            gConfig = get_config(sys.argv[1])
        else:
            # get configuration from seq2seq.ini
            gConfig = get_config()


        print('\n>> Mode : %s\n' %(gConfig['mode']))


        if gConfig['mode'] == 'train':
            # start training
            train()
        elif gConfig['mode'] == 'test':
            # interactive decode
            decode()
        else:
            # wrong way to execute "serve"
            #   Use : >> python ui/app.py
            #          uses seq2seq_serve.ini as conf file
            print('Serve Usage : >> python ui/app.py')
            print('# uses seq2seq_serve.ini as conf file')
```