chatbot_using_RNN

December 8, 2017

1 Chatbot using RNN

1.0.1 Northwestern University - Fall 2017

1.0.2 Student: Danilo Neves Ribeiro

1.0.3 E-mail: daniloribeiro2021@u.northwestern.edu

2 Introduction

The idea of this project is to create a simple chatbot by training a Recurrent Neural Networks.

2.1 Chatbots

There are many ways one can go about creating a chat-bot. For example, many chatbots rely on pre-defined rules to answer questions. Those can work well but requires intese human work to create as many rules as possible.

Machine learning greately simplify this task by enableing to learn from pre-existing conversation corpus. The two main types of ML chatbots are:

- Retrieval-based: answer questions by choosing from one of the answers available in the data-set.
- Generative: generates the conversation dialog word by word based on the query. The generated sentense is normally not included in the original data-set.

For this project, I decided to create a chatbot using the generative approch, which normally makes more mistakes, such as grammar mistakes, but can respond a broader set of questions and contexts.

2.2 Dataset

The model was trained using the Cornell Movie Dialog Corpus, that contains a collection of fictional conversations extracted from raw movie scripts. The data was split in 108136 conversation pairs for training, and 30000 conversation pairs for testing.

2.3 Implementation Architecture

Here I use a Recurrent Neural Network to train on the data set. More specifically I use a seq2seq model with bucketing and attention mechanism, which is described in more details below:

2.3.1 Seq2Seq:

Sequence to Sequence RNN models are composed of two main components: encoder and decoder. The encoder is responsible for reading the input, word by word, and generating a hidden state that "represents" the input. The decoder outputs words according to the hidden states generated by the encoder. The following image gives a general idea of this architecture:

2.3.2 Padding and Bucketing:

One of the limitations of the simple Seq2Seq arquitectures is that it has fixed size input and output. Therefore we need to use padding and special symbols to deal with the fact that both input and output sentences can have different length (the ones used here are: EOS = "End of sentence", PAD = "Filler", GO = "Start decoding", plus a special symbol for unknown words: UNK).

To efficiently handle sentenses with different lengths the bucketing method is used. This model uses buckets = [(5, 10), (10, 15), (20, 25), (40, 50)]. This means that if the input is a sentence with 3 tokens, and the corresponding output is a sentence with 6 tokens, then they will be put in the first bucket and padded to length 5 for encoder inputs, and length 10 for decoder inputs.

2.3.3 Attention mechanism:

The attention mechanism tries to address the following limitations: - The decoder is not aware of which parts of the encoding are relevant at each step of the generation. - The encoder has limited memory and can't "remember" more than a single fixed size vector.

The attention model comes between the encoder and the decoder and helps the decoder to pick only the encoded inputs that ar important for each step of the decoding process.

3 Code

The code will be split between: - Preprocessing data (tokenizing, creating vobabulary, transforming input from words to word ids) - Training - Testing

Software requirements

- Python 3.6.2
- Numpy
- TensorFlow

Note: the following code is largely based on the code from the chatbot tutorial by Suriyadeepan Ram and uses the more general seq2seq model provided by the Google Tensorflow tutorial on NMT, which is imported from a separate code file.

3.0.4 Preprocessing

```
In [1]: # IMPORTS

        import os
        import numpy as np
        import re
```

```
        import tensorflow as tf
        from seq2seq_model import Seq2SeqModel
```

In [2]:
```
        # GLOBAL VARIABLES AND PARAMS

        # encoding and decoding paths
        TRAIN_END_PATH = os.path.join('data', 'train.enc')
        TRAIN_DEC_PATH = os.path.join('data', 'train.dec')
        TEST_END_PATH = os.path.join('data', 'test.enc')
        TEST_DEC_PATH = os.path.join('data', 'test.dec')

        TRAIN_END_ID_PATH = os.path.join('data', 'train.enc.id')
        TRAIN_DEC_ID_PATH = os.path.join('data', 'train.dec.id')
        TEST_END_ID_PATH = os.path.join('data', 'test.enc.id')
        TEST_DEC_ID_PATH = os.path.join('data', 'test.dec.id')

        # vocabulary paths
        VOCAB_ENC_PATH = os.path.join('data', 'vocab.enc')
        VOCAB_DEC_PATH = os.path.join('data', 'vocab.dec')
        MAX_VOCAB_SIZE = 20000

        # data utils
        SPLIT_REGEX = re.compile("([.,!?\"':;)(])")
        PAD_TOKEN = "_PAD"
        START_TOKEN = "_GO"
        END_TOKEN = "_EOS"
        UNKNOWEN_TOKEN = "_UNK"
        INIT_VOCAB = [PAD_TOKEN, START_TOKEN, END_TOKEN, UNKNOWEN_TOKEN]

        # args
        BUCKETS = [(5, 10), (10, 15), (20, 25), (40, 50)]
        LSTM_LAYES = 3
        LAYER_SIZE = 256
        BATCH_SIZE = 64
        LEARNING_RATE = 0.5
        LEARNING_RATE_DECAY_FACTOR = 0.99
        MAX_GRADIENT_NORM = 5.0
        STEP_CHECKPOINTS = 5
        MAX_ITERATIONS = 1000

        # pre training
        TRAINED_MODEL_PATH = 'pre_trained'
        TRAINED_VOCAB_ENC = os.path.join('pre_trained', 'vocab.enc')
        TRAINED_VOCAB_DEC = os.path.join('pre_trained', 'vocab.dec')
```

In [3]:
```
        # SIMPLE TOKENIZER

        def tokenize(sentense):
```

```
    tokens = []
    for token in sentense.strip().split():
        tokens.extend([x for x in re.split(SPLIT_REGEX, token) if x])
    return tokens
```

In [4]:
```
# CREATING VOCABULARY

def create_vocab(data_path, vocab_path):
    vocab = {}
    # only creates new file if file doesn't exist
    if os.path.exists(vocab_path):
        print("file ", vocab_path, " already exists")
    else:
        with open(data_path, 'r') as data_file:
            for line in data_file:
                tokens = tokenize(line)
                for token in tokens:
                    if token not in vocab:
                        vocab[token] = 1
                    else:
                        vocab[token] += 1
        # use the default tokens as initial vocabulity words
        vocab_list = INIT_VOCAB + sorted(vocab, key=vocab.get, reverse=True)
        # trim vocabulary
        vocab_list = vocab_list[:MAX_VOCAB_SIZE]
        print("final vacabulary size for ", data_path, " = ", len(vocab_list))
        # save to file
        with open(vocab_path, 'w') as vocab_file:
            for word in vocab_list:
                vocab_file.write(word + "\n")
        # update vocab with new order
        vocab = dict([(y, x) for (x, y) in enumerate(vocab_list)])
        return vocab
```

In [5]:
```
# TRANSFORM WORDS IN DATA TO IDS

def from_text_data_to_id_list(data_path, ouput_path, vocab):
    # only creates new file is file doesn't exist
    if os.path.exists(ouput_path):
        print("file ", ouput_path, " already exists")
    else:
        with open(data_path, 'r') as data_file:
            with open(ouput_path, 'w') as ouput_file:
                for line in data_file:
                    tokens = tokenize(line)
                    id_list = [str(vocab.get(word, vocab.get(UNKNOWEN_TOKEN))) for word
                    ouput_file.write(" ".join(id_list) + "\n")
```

```
In [6]: # DATA PREPROCESSING

        def preprocess_data():
            encoding_vocab = create_vocab(TRAIN_END_PATH, VOCAB_ENC_PATH)
            decoding_vocab = create_vocab(TRAIN_DEC_PATH, VOCAB_DEC_PATH)
            from_text_data_to_id_list(TRAIN_END_PATH, TRAIN_END_ID_PATH, encoding_vocab)
            from_text_data_to_id_list(TRAIN_DEC_PATH, TRAIN_DEC_ID_PATH, decoding_vocab)
            from_text_data_to_id_list(TEST_END_PATH, TEST_END_ID_PATH, encoding_vocab)
            from_text_data_to_id_list(TEST_DEC_PATH, TEST_DEC_ID_PATH, decoding_vocab)
            print("Data preprocessing complete.")

        preprocess_data()
```

```
file  data\vocab.enc  already exists
file  data\vocab.dec  already exists
file  data\train.enc.id  already exists
file  data\train.dec.id  already exists
file  data\test.enc.id  already exists
file  data\test.dec.id  already exists
Data preprocessing complete.
```

3.0.5 Training

```
In [7]: def read_data(source_path, target_path):
            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()
                    while source and target:
                        source_ids = [int(x) for x in source.split()]
                        target_ids = [int(x) for x in target.split()]
                        target_ids.append(INIT_VOCAB.index(END_TOKEN))
                        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

In [8]: # CREATE MODEL

        def create_model(forward_only):
            # TODO: remove
            return Seq2SeqModel(
                MAX_VOCAB_SIZE, MAX_VOCAB_SIZE, BUCKETS, LAYER_SIZE, LSTM_LAYES, MAX_GRADIENT_NO
                BATCH_SIZE, LEARNING_RATE, LEARNING_RATE_DECAY_FACTOR, forward_only)
```

In [ ]:
```
# TRAIN MODEL

        def train():
            # setup config to use BFC allocator
            config = tf.ConfigProto()
            config.gpu_options.allocator_type = 'BFC'
            with tf.Session(config=config) as sess:
                print("creating model...")
                model = create_model(forward_only = False)
                sess.run(tf.global_variables_initializer())

                # Read data into buckets and compute their sizes.
                dev_set = read_data(TEST_END_ID_PATH, TEST_DEC_ID_PATH)
                train_set = read_data(TRAIN_END_ID_PATH, TRAIN_DEC_ID_PATH)
                train_bucket_sizes = [len(train_set[b]) for b in range(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 range(len(train_bucket_sizes))]

                print("Running main loop...")
                for current_step in range(MAX_ITERATIONS):
                    # 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 = np.random.random_sample()
                    bucket_id = min([i for i in range(len(train_buckets_scale))
                                     if train_buckets_scale[i] > random_number])

                    # Get a batch and make a step.
                    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)

                    # Print statistics.
                    perplexity = math.exp(step_loss) if step_loss < 300 else float('inf')
                    print ("global step %d perplexity %.2f" % (model.global_step.eval(), perplex

        # train model
        train()
```

```
creating model...
```

3.0.6 Testing

In [ ]:
```
# LOAD PRE-TRAINED MODEL

    def load_vocabulary_list(vocabulary_path):
        with open(vocabulary_path, mode="r") as vocab_file:
            return [line.strip() for line in vocab_file.readlines()]

    def load_pre_trained_model(session):
        print("Loading vocab...")
        enc_vocab_list = load_vocabulary_list(TRAINED_VOCAB_ENC)
        dec_vocab_list = load_vocabulary_list(TRAINED_VOCAB_DEC)
        enc_vocab = dict([(x, y) for (y, x) in enumerate(dec_vocab_list)])
        rev_dec_vocab = dict(enumerate(dec_vocab_list))

        print("Creting model...")
        model = create_model(forward_only = True)

        print("Loading saved model...")
        ckpt = tf.train.get_checkpoint_state(TRAINED_MODEL_PATH)
        model.saver.restore(session, ckpt.model_checkpoint_path)
        return (model, enc_vocab, rev_dec_vocab)
```

In [ ]:
```
# DECODING

    def decode(sentence, model, session, enc_vocab):
        # Get token-ids for the input sentence.
        token_ids = [enc_vocab.get(w, INIT_VOCAB.index(UNKNOWEN_TOKEN)) for w in tokenize(se
        bucket_id = min([b for b in range(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(session, 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]
        return outputs
```

In [ ]:
```
# CHATBOT MAIN APP

    def run_chatbot():
        print("Starting chatbot...")
        with tf.Session() as sess:
            model, enc_vocab, rev_dec_vocab = load_pre_trained_model(sess)
            model.batch_size = 1  # We decode one sentence at a time.
            # Decode from standard input.
            sentence = input("Chatbot started, ask anything!\n> ")
            while sentence:
```

```
                outputs = decode(sentence, model, sess, enc_vocab)
                # If there is an EOS symbol in outputs, cut them at that point.
                if INIT_VOCAB.index(END_TOKEN) in outputs:
                    outputs = outputs[:outputs.index(INIT_VOCAB.index(END_TOKEN))]
                    print(" ".join([tf.compat.as_str(rev_dec_vocab[output]) for output in ou
                    sentence = input("> ")

    run_chatbot()
```

4 Evaluation

In the last stop of training the model reported global perplexity around 8.3

Follows an image with a sample conversation, to help evaluate the qualitative side of the model:

5 Conclusion

This project showed how a simple generative chatbot can be created using a Recurrent Neural Net. The final results indicates that the model can perform reasonably well for a open conversation chatbot, even though it still makes grammar mistakes and sometimes gives very vague or unrelated answers.

5.1 Project Challenges

Here I present some of the challenges I faced when tring to train the model for this project:

- Initially I tried to use the ubuntu-dialog corpus. The dataset proved to be very large (a few Gb) and it took several hours just to preprocess the data. I decided that this corpus would be to complex to train on and decided to use the The Cornell Movie Dialog Corpus.
- The Cornell Movie Dialog Corpus is a smaller and more manageble dataset, but training the model still took several hours (almost two days), while consuming a big par of my computer's resources.