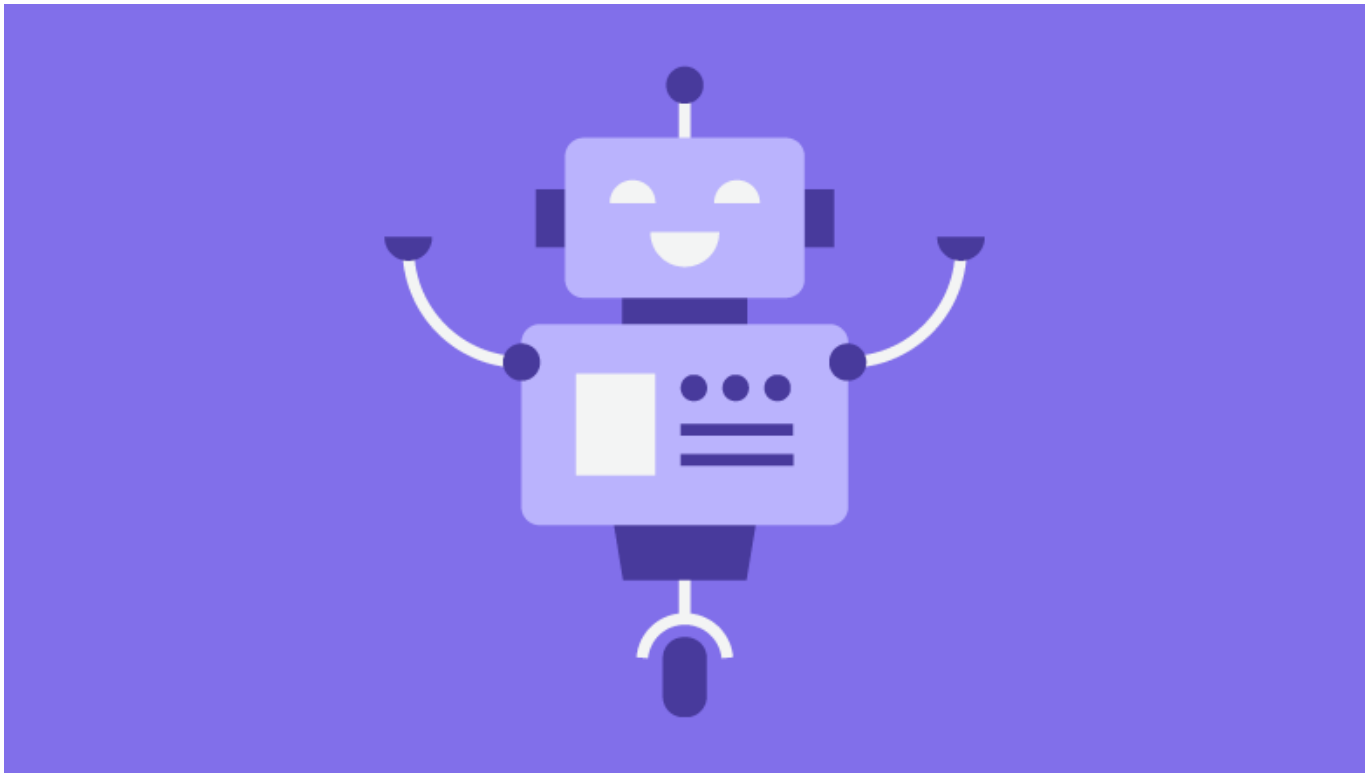
Make a Chat Bot with TensorFlow NLP and Anaconda Navigator..

Mathan Jeya Follow
Apr 3, 2019 · 6 min read

Keep On Practicing guys Python looks easy….

Nowadays, there is great demand of Machine learning and Artificial Intelligence Engineers. If you are looking for a good job then you must have great learning and great understanding of Machine Learning and Artificial Intelligence algorithms. You should also work with few good projects which will give you an end-to-end experience of logic that we use in Machine learning. Today, I am going to share with you the implementation of ChatBot which you can use for your personal question and answer play.



So let's start building ChatBot using movies dialogues dataset. Follow each and every steps to get the desired output.

Installation:

For windows user please install Anaconda using following link.

For Mac user please follow this link.

For Linux user please follow this link.

You can easily install anaconda and after that we will create a virtual environment for our ChatBot. After the installation of anaconda open Anaconda prompt (Windows user) and Terminal(Mac & Linux user) and then type the following command-

```
conda create -n chatbot python=3.5 anaconda
```

Click on enter button and then let the installation start. After the installation of this virtual environment write a command-

```
conda activate chatbot
```

Press enter after writing the above command and then you will enter in your virtual environment. After you enter in your virtual environment now you have to install TensorFlow in your environment. So write the following command to install TensorFlow-

```
pip install tensorflow==0.12.1
```

After the installation of tensorflow in your virtual environment you are ready to go and let's start creating our chatbot. You can download the dataset from the following link.

Import these libraries-

```
# Importing the libraries
import numpy as np
import tensorflow as tf
import re
import time
```

Data Preprocessing-

This involves a series of steps to preprocess the imported dataset to the required format needed for it to work with the Seq2seq model in Tensorflow.

```
# Importing the dataset
lines = open('movie_lines.txt', encoding = 'utf-8', errors = 'ignore').read().split('\n')
conversations = open('movie_conversations.txt', encoding = 'utf-8', errors = 'ignore').read().split('\n')
```

```
+ They do not!
++ They do to!
 I hope so.
+ She okay?
 Let's go.
+ Wow
 Okay -- you're gonna need to learn how to lie.
+ No
 I'm kidding.  You know how sometimes you just become this "persona"?  And you don't know how to quit?
 Like my fear of wearing pastels?
+ The "real you".
 What good stuff?
+ I figured you'd get to the good stuff eventually.
+ Thank God!  If I had to hear one more story about your coiffure...
 Me.  This endless ...blonde babble. I'm like, boring myself.
+ What crap?
 do you listen to this crap?
+ No...
 Then Guillermo says, "If you go any lighter, you're gonna look like an extra on 90210."
+ You always been this selfish?
 But
+ Then that's all you had to say.
 Well, no...
+ You never wanted to go out with 'me, did you?
 I was?
+ I looked for you back at the party, but you always seemed to be "occupied".
 Tons
+ Have fun tonight?
+ I believe we share an art instructor
 You know Chastity?
+ Looks like things worked out tonight, huh?
 Hi.
 Who knows?  All I've ever heard her say is that she'd dip before dating a guy that smokes.
+ So that's the kind of guy she likes? Pretty ones?
 Lesbian?  No. I found a picture of Jared Leto in one of her drawers, so I'm pretty sure she's not harboring same-se
+ She's not a...
+ I'm workin' on it. But she doesn't seem to be goin' for him.
 I really, really, really wanna go, but I can't.  Not unless my sister goes.
+ Sure have.
 Eber's Deep Conditioner every two days. And I never, ever use a blowdryer without the diffuser attachment.
+ How do you get your hair to look like that?
```

Creating dictionary so that mapping of each line and id can be done.

```
# Creating a dictionary that maps each line and its id
id2line = {}
for line in lines:
    _line = line.split(' +++$+++ ')
    if len(_line) == 5:
        id2line[_line[0]] = _line[4]

# Creating a list of all of the conversations
conversations_ids = []
for conversation in conversations[:-1]:
    _conversation = conversation.split(' +++$+++ ')[-1][1:-1].replace("'", "").replace(" ", "")
    conversations_ids.append(_conversation.split(','))
```

If the length of list in the dictionary is even, the first conversation is taken as the question and the second one as the answer. This is repeated for its entire length. For odd number of conversations, the last one is neglected for convenience.

```
# Getting separately the questions and the answers
questions = []
answers = []
for conversation in conversations_ids:
    for i in range(len(conversation) - 1):
        questions.append(id2line[conversation[i]])
        answers.append(id2line[conversation[i+1]])
```

This function converts all characters to lower-cases and converts commonly used short texts to their full version. Now lets pass our *questions* and *answers* through this function.

```
# Doing a first cleaning of the texts
def clean_text(text):
    text = text.lower()
    text = re.sub(r"i'm", "i am", text)
    text = re.sub(r"he's", "he is", text)
    text = re.sub(r"she's", "she is", text)
    text = re.sub(r"that's", "that is", text)
    text = re.sub(r"what's", "what is", text)
    text = re.sub(r"where's", "where is", text)
    text = re.sub(r"how's", "how is", text)
    text = re.sub(r"\'ll", " will", text)
    text = re.sub(r"\'ve", " have", text)
    text = re.sub(r"\'re", " are", text)
    text = re.sub(r"\'d", " would", text)
    text = re.sub(r"n't", " not", text)
    text = re.sub(r"won't", "will not", text)
    text = re.sub(r"can't", "cannot", text)
    text = re.sub(r"[-()\"#/@;:<>{}`+=~|.!?,]", "", text)
    return text
```

This code is self explanatory. It removes very short and very long questions. Makes it easier for our model to train.

```
# Cleaning the questions
clean_questions = []
for question in questions:
    clean_questions.append(clean_text(question))

# Cleaning the answers
clean_answers = []
for answer in answers:
    clean_answers.append(clean_text(answer))
```

```
# Filtering out the questions and answers that are too short or too long
short_questions = []
short_answers = []
i = 0
for question in clean_questions:
    if 2 <= len(question.split()) <= 25:
        short_questions.append(question)
        short_answers.append(clean_answers[i])
    i += 1
clean_questions = []
clean_answers = []
i = 0
for answer in short_answers:
    if 2 <= len(answer.split()) <= 25:
        clean_answers.append(answer)
        clean_questions.append(short_questions[i])
    i += 1



# Creating a dictionary that maps each word to its number of occurrences
word2count = {}
for question in clean_questions:
    for word in question.split():
        if word not in word2count:
            word2count[word] = 1
        else:
            word2count[word] += 1
for answer in clean_answers:
    for word in answer.split():
        if word not in word2count:
            word2count[word] = 1
        else:
            word2count[word] += 1
```

This is to get the frequency of each word in our dataset. Will help in filtering out words that are rare (usually names,nouns that don't help the model learn). The less frequent words are removed with the below code.

```
# Creating two dictionaries that map the questions words and the answers words to a unique integer
threshold_questions = 15
questionswords2int = {}
word_number = 0
for word, count in word2count.items():
    if count >= threshold_questions:
        questionswords2int[word] = word_number
        word_number += 1
threshold_answers = 15
answerswords2int = {}
word_number = 0
for word, count in word2count.items():
    if count >= threshold_answers:
        answerswords2int[word] = word_number
        word_number += 1
```

```
# Adding the last tokens to these two dictionaries
tokens = ['<PAD>', '<EOS>', '<OUT>', '<SOS>']
for token in tokens:
    questionswords2int[token] = len(questionswords2int) + 1
for token in tokens:
    answerswords2int[token] = len(answerswords2int) + 1
```

These tokens help the model understand the start and the end of a sentence, rare words etc.

```
# Creating the inverse dictionary of the answerswords2int dictionary
answersints2word = {w_i: w for w, w_i in answerswords2int.items()}
```

The *answersints2word* helps form a sentence when the model predicts an output with integers.

```
# Adding the End Of String token to the end of every answer
for i in range(len(clean_answers)):
    clean_answers[i] += ' <EOS>'
```

Helps the model understand the end of each answer in the model.

```
# Translating all the questions and the answers into integers
# and Replacing all the words that were filtered out by <OUT>
questions_into_int = []
for question in clean_questions:
    ints = []
    for word in question.split():
        if word not in questionswords2int:
            ints.append(questionswords2int['<OUT>'])
        else:
            ints.append(questionswords2int[word])
    questions_into_int.append(ints)
answers_into_int = []
for answer in clean_answers:
    ints = []
    for word in answer.split():
        if word not in answerswords2int:
            ints.append(answerswords2int['<OUT>'])
        else:
            ints.append(answerswords2int[word])
    answers_into_int.append(ints)
```

Converts every single word in questions and answers to integers and assigns the ‘OUT’ tag for the words that were filtered out in one of the above steps.

```
# Sorting questions and answers by the length of questions
sorted_clean_questions = []
sorted_clean_answers = []
for length in range(1, 25 + 1):
    for i in enumerate(questions_into_int):
        if len(i[1]) == length:
            sorted_clean_questions.append(questions_into_int[i[0]])
            sorted_clean_answers.append(answers_into_int[i[0]])
```

Sorts questions and answers by their length. From 1 to 25.

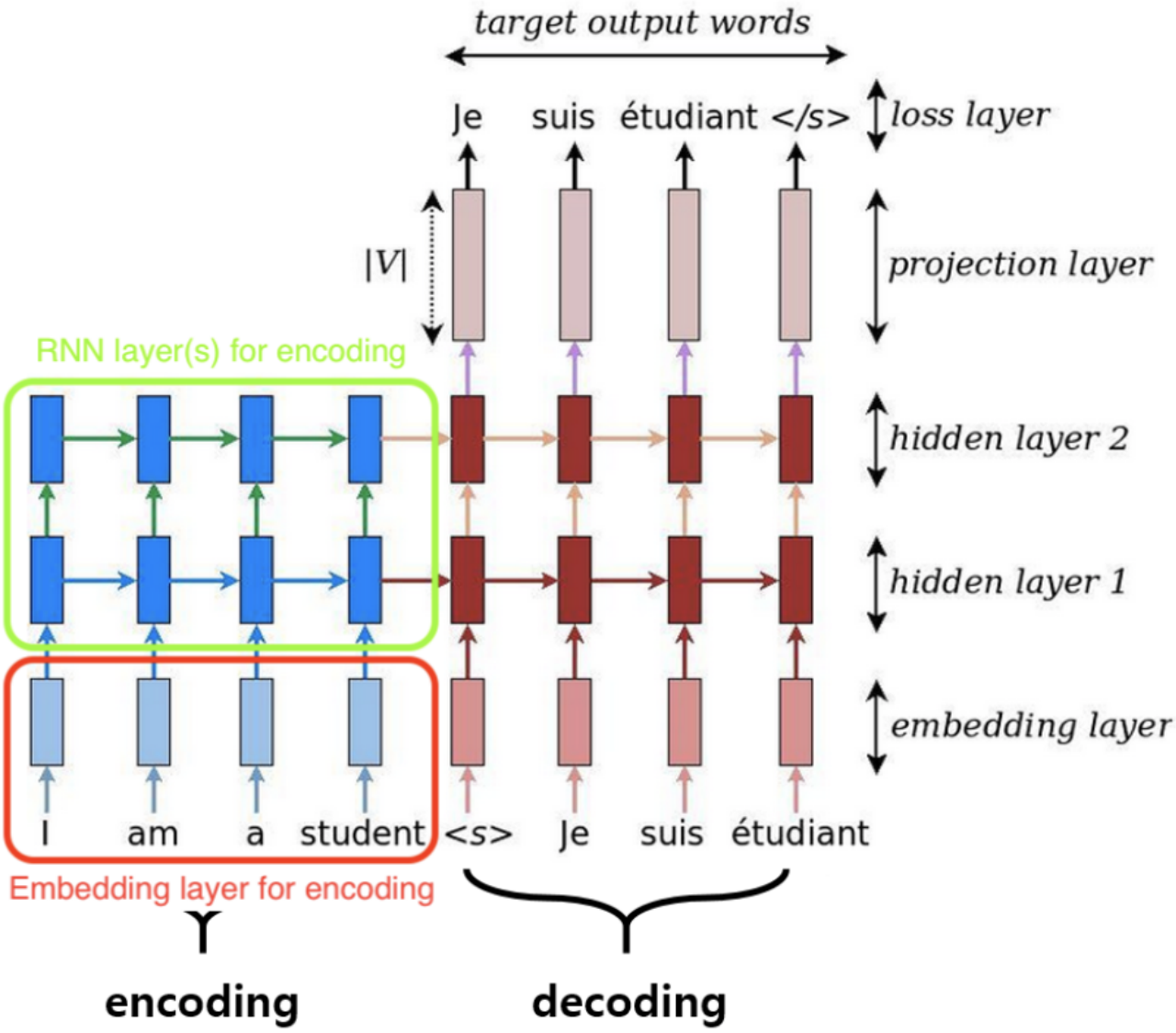
. . .

Build better voice apps. Get more articles & interviews from voice technology experts at voicetechpodcast.com

. . .

Building the seq2seq model -

A seq2seq model is a type of many-to-many RNN model. They are most commonly used for chatbots and translation models. It has 2 components, the encoder and the decoder.



```
# Creating placeholders for the inputs and the targets
def model_inputs():
    inputs = tf.placeholder(tf.int32, [None, None], name = 'input')
    targets = tf.placeholder(tf.int32, [None, None], name = 'target')
    lr = tf.placeholder(tf.float32, name = 'learning_rate')
    keep_prob = tf.placeholder(tf.float32, name = 'keep_prob')
    return inputs, targets, lr, keep_prob
```

A placeholder is simply a variable that we will assign data to at a later date. It allows us to create our operations and build our computation graph, without needing the data.

```
# Preprocessing the targets
def preprocess_targets(targets, word2int, batch_size):
    left_side = tf.fill([batch_size, 1], word2int['<SOS>'])
    right_side = tf.strided_slice(targets, [0,0], [batch_size, -1], [1,1])
    preprocessed_targets = tf.concat([left_side, right_side], 1)
    return preprocessed_targets
```

Assigns SOS tag to the start of every target(answers).

```
# Creating the Encoder RNN
def encoder_rnn(rnn_inputs, rnn_size, num_layers, keep_prob, sequence_length):
    lstm = tf.contrib.rnn.BasicLSTMCell(rnn_size)
    lstm_dropout = tf.contrib.rnn.DropoutWrapper(lstm, input_keep_prob = keep_prob)
    encoder_cell = tf.contrib.rnn.MultiRNNCell([lstm_dropout] * num_layers)
    encoder_output, encoder_state = tf.nn.bidirectional_dynamic_rnn(cell_fw = encoder_cell,
                                                                    cell_bw = encoder_cell,
                                                                    sequence_length = sequence_length,
                                                                    inputs = rnn_inputs,
                                                                    dtype = tf.float32)
    return encoder_state
```

The Decoder RNN is built after decoding the training and test sets using the encoder state obtained from the above function.

```
# Decoding the test/validation set
def decode_test_set(encoder_state, decoder_cell, decoder_embeddings_matrix, sos_id, eos_id, maximum_length, num_words, decoding_scope, output_function, keep_prob, batch_size):
    attention_states = tf.zeros([batch_size, 1, decoder_cell.output_size])
    attention_keys, attention_values, attention_score_function, attention_construct_function = tf.contrib.seq2seq.prepare_attention(attention_states,
                                                                                                                                     attention_option = "bahdanau",
                                                                                                                                     num_units = decoder_cell.output_size)
    test_decoder_function = tf.contrib.seq2seq.attention_decoder_fn_inference(output_function,
                                                                              encoder_state[0],
                                                                              attention_keys,
                                                                              attention_values,
                                                                              attention_score_function,
                                                                              attention_construct_function,
                                                                              decoder_embeddings_matrix,
                                                                              sos_id,
                                                                              eos_id,
                                                                              maximum_length,
                                                                              num_words,
                                                                              name = "attn_dec_inf")
    test_predictions, decoder_final_state, decoder_final_context_state = tf.contrib.seq2seq.dynamic_rnn_decoder(decoder_cell,
                                                                                                                test_decoder_function,
                                                                                                                scope = decoding_scope)
    return test_predictions
```

```
# Creating the Decoder RNN
def decoder_rnn(decoder_embedded_input, decoder_embeddings_matrix, encoder_state, num_words, sequence_length, rnn_size, num_layers, word2int, keep_prob, batch_size):
    with tf.variable_scope("decoding") as decoding_scope:
        lstm = tf.contrib.rnn.BasicLSTMCell(rnn_size)
        lstm_dropout = tf.contrib.rnn.DropoutWrapper(lstm, input_keep_prob = keep_prob)
        decoder_cell = tf.contrib.rnn.MultiRNNCell([lstm_dropout] * num_layers)
        weights = tf.truncated_normal_initializer(stddev = 0.1)
        biases = tf.zeros_initializer()
        output_function = lambda x: tf.contrib.layers.fully_connected(x,
                                                                      num_words,
                                                                      None,
                                                                      scope = decoding_scope,
                                                                      weights_initializer = weights,
                                                                      biases_initializer = biases)
        training_predictions = decode_training_set(encoder_state,
                                                   decoder_cell,
                                                   decoder_embedded_input,
                                                   sequence_length,
                                                   decoding_scope,
                                                   output_function,
                                                   keep_prob,
                                                   batch_size)
        decoding_scope.reuse_variables()
        test_predictions = decode_test_set(encoder_state,
                                           decoder_cell,
                                           decoder_embeddings_matrix,
                                           word2int['<SOS>'],
                                           word2int['<EOS>'],
                                           sequence_length - 1,
                                           num_words,
                                           decoding_scope,
                                           output_function,
                                           keep_prob,
                                           batch_size)
    return training_predictions, test_predictions
```

Now that we have both the encoder RNN and the decoder RNN we'll use them to build our seq2seq model.

```
# Building the seq2seq model
def seq2seq_model(inputs, targets, keep_prob, batch_size, sequence_length, answers_num_words, questions_num_words, encoder_embedding_size, decoder_embedding_size,
                  rnn_size, num_layers, questionswords2int):
    encoder_embedded_input = tf.contrib.layers.embed_sequence(inputs,
                                                              answers_num_words + 1,
                                                              encoder_embedding_size,
                                                              initializer = tf.random_uniform_initializer(0, 1))
    encoder_state = encoder_rnn(encoder_embedded_input, rnn_size, num_layers, keep_prob, sequence_length)
    preprocessed_targets = preprocess_targets(targets, questionswords2int, batch_size)
    decoder_embeddings_matrix = tf.Variable(tf.random_uniform([questions_num_words + 1, decoder_embedding_size], 0, 1))
    decoder_embedded_input = tf.nn.embedding_lookup(decoder_embeddings_matrix, preprocessed_targets)
    training_predictions, test_predictions = decoder_rnn(decoder_embedded_input,
                                                         decoder_embeddings_matrix,
                                                         encoder_state,
                                                         questions_num_words,
                                                         sequence_length,
                                                         rnn_size,
                                                         num_layers,
                                                         questionswords2int,
                                                         keep_prob,
                                                         batch_size)
    return training_predictions, test_predictions
```

Training the seq2seq model-

We have all our functions defined and are getting closer to training the model. Let's setup the hyperparameters needed.

```
# Setting the Hyperparameters
epochs = 100
batch_size = 32
rnn_size = 1024
num_layers = 3
encoding_embedding_size = 1024
decoding_embedding_size = 1024
learning_rate = 0.001
learning_rate_decay = 0.9
min_learning_rate = 0.0001
keep_probability = 0.5
```

This is totally modifiable depending on your system spec. 25 epochs is usually too small for a building a good chatbot. You can choose to increase it if you want better results.

```
# Defining a session
tf.reset_default_graph()
session = tf.InteractiveSession()

# Loading the model inputs
inputs, targets, lr, keep_prob = model_inputs()

# Setting the sequence length
sequence_length = tf.placeholder_with_default(25, None, name = 'sequence_length')

# Getting the shape of the inputs tensor
input_shape = tf.shape(inputs)
```

Starting a Tensorflow session and defining few other parameter needed for training.

```
# Getting the training and test predictions
training_predictions, test_predictions = seq2seq_model(tf.reverse(inputs, [-1]),
                                                       targets,
                                                       keep_prob,
                                                       batch_size,
                                                       sequence_length,
                                                       len(answerswords2int),
                                                       len(questionswords2int),
                                                       encoding_embedding_size,
                                                       decoding_embedding_size,
                                                       rnn_size,
                                                       num_layers,
                                                       questionswords2int)
```

Defining the predictions, which we know is the output of the seq2seq function defined above.

```
# Setting up the Loss Error, the Optimizer and Gradient Clipping
with tf.name_scope("optimization"):
    loss_error = tf.contrib.seq2seq.sequence_loss(training_predictions,
                                                  targets,
                                                  tf.ones([input_shape[0], sequence_length]))
    optimizer = tf.train.AdamOptimizer(learning_rate)
    gradients = optimizer.compute_gradients(loss_error)
    clipped_gradients = [(tf.clip_by_value(grad_tensor, -5., 5.), grad_variable) for grad_tensor, grad_variable in gradients if grad_tensor is not None]
    optimizer_gradient_clipping = optimizer.apply_gradients(clipped_gradients)
```

Like any Deep Learning model, we have to define the loss error and optimizer. And gradient clipping is done to prevent exploding or vanishing of errors in the earlier layers of the neural network.

```
# Padding the sequences with the <PAD> token
def apply_padding(batch_of_sequences, word2int):
    max_sequence_length = max([len(sequence) for sequence in batch_of_sequences])
    return [sequence + [word2int['<PAD>']] * (max_sequence_length - len(sequence)) for sequence in batch_of_sequences]
```

Padding are dummy values added to the sequence making the questions and answers to have the same length.

```
# Splitting the data into batches of questions and answers
def split_into_batches(questions, answers, batch_size):
    for batch_index in range(0, len(questions) // batch_size):
        start_index = batch_index * batch_size
        questions_in_batch = questions[start_index : start_index + batch_size]
        answers_in_batch = answers[start_index : start_index + batch_size]
        padded_questions_in_batch = np.array(apply_padding(questions_in_batch, questionswords2int))
        padded_answers_in_batch = np.array(apply_padding(answers_in_batch, answerswords2int))
        yield padded_questions_in_batch, padded_answers_in_batch
```

The RNN model accepts inputs only in batches.

```
# Splitting the questions and answers into training and validation sets
training_validation_split = int(len(sorted_clean_questions) * 0.15)
training_questions = sorted_clean_questions[training_validation_split:]
training_answers = sorted_clean_answers[training_validation_split:]
validation_questions = sorted_clean_questions[:training_validation_split]
validation_answers = sorted_clean_answers[:training_validation_split]
```

Splitting the data to training set and test set. The model trains of 85% the data and tests its prediction on the 15% data to compute the loss error and improve in further epochs.

```
# Training
batch_index_check_training_loss = 100
batch_index_check_validation_loss = ((len(training_questions)) // batch_size // 2) - 1
total_training_loss_error = 0
list_validation_loss_error = []
early_stopping_check = 0
early_stopping_stop = 100
checkpoint = "chatbot_weights.ckpt"
session.run(tf.global_variables_initializer())
for epoch in range(1, epochs + 1):
    for batch_index, (padded_questions_in_batch, padded_answers_in_batch) in enumerate(split_into_batches(training_questions, training_answers, batch_size)):
        starting_time = time.time()
        _, batch_training_loss_error = session.run([optimizer_gradient_clipping, loss_error], {inputs: padded_questions_in_batch,
                                                                                               targets: padded_answers_in_batch,
                                                                                               lr: learning_rate,
                                                                                               sequence_length: padded_answers_in_batch.shape[1],
                                                                                               keep_prob: keep_probability})
        total_training_loss_error += batch_training_loss_error
        ending_time = time.time()
        batch_time = ending_time - starting_time
        if batch_index % batch_index_check_training_loss == 0:
            print('Epoch: {:>3}/{}, Batch: {:>4}/{}, Training Loss Error: {:>6.3f}, Training Time on 100 Batches: {:d} seconds'.format(epoch,
                                                                                                                                      epochs,
                                                                                                                                      batch_index,
                                                                                                                                      len(training_questions) // batch_size,
                                                                                                                                      total_training_loss_error / batch_index_check_training_loss,
                                                                                                                                      int(batch_time * batch_index_check_training_loss)))
            total_training_loss_error = 0
        if batch_index % batch_index_check_validation_loss == 0 and batch_index > 0:
            total_validation_loss_error = 0
            starting_time = time.time()
            for batch_index_validation, (padded_questions_in_batch, padded_answers_in_batch) in enumerate(split_into_batches(validation_questions, validation_answers, batch_size)):
                batch_validation_loss_error = session.run(loss_error, {inputs: padded_questions_in_batch,
                                                                       targets: padded_answers_in_batch,
                                                                       lr: learning_rate,
                                                                       sequence_length: padded_answers_in_batch.shape[1],
                                                                       keep_prob: 1})
                total_validation_loss_error += batch_validation_loss_error
            ending_time = time.time()
            batch_time = ending_time - starting_time
            average_validation_loss_error = total_validation_loss_error / (len(validation_questions) / batch_size)
            print('Validation Loss Error: {:>6.3f}, Batch Validation Time: {:d} seconds'.format(average_validation_loss_error, int(batch_time)))
            learning_rate *= learning_rate_decay
            if learning_rate < min_learning_rate:
                learning_rate = min_learning_rate
            list_validation_loss_error.append(average_validation_loss_error)
            if average_validation_loss_error <= min(list_validation_loss_error):
                print('I speak better now!!')
                early_stopping_check = 0
                saver = tf.train.Saver()
                saver.save(session, checkpoint)
            else:
                print("Sorry I do not speak better, I need to practice more.")
                early_stopping_check += 1
                if early_stopping_check == early_stopping_stop:
                    break
    if early_stopping_check == early_stopping_stop:
        print("My apologies, I cannot speak better anymore. This is the best I can do.")
        break
print("Game Over")
```

Testing the seq2seq model-

```
# Loading the weights and Running the session
checkpoint = "./chatbot_weights.ckpt"
session = tf.InteractiveSession()
session.run(tf.global_variables_initializer())
saver = tf.train.Saver()
saver.restore(session, checkpoint)
```

During training, the weights are saved at every checkpoint. This can be loaded to chat with the chatbot.

```
# Converting the questions from strings to lists of encoding integers
def convert_string2int(question, word2int):
    question = clean_text(question)
    return [word2int.get(word, word2int['<OUT>']) for word in question.split()]
```

This function converts our input question to integers. Our model was trained with integers and not words.

Setting up our chatbot-

```
# Setting up the chat
while(True):
    question = input("You: ")
    if question == 'Goodbye':
        break
    question = convert_string2int(question, questionswords2int)
    question = question + [questionswords2int['<PAD>']] * (25 - len(question))
    fake_batch = np.zeros((batch_size, 25))
    fake_batch[0] = question
    predicted_answer = session.run(test_predictions, {inputs: fake_batch, keep_prob: 0.5})[0]
    answer = ''
    for i in np.argmax(predicted_answer, 1):
        if answersints2word[i] == 'i':
            token = ' I'
        elif answersints2word[i] == '<EOS>':
            token = '.'
        elif answersints2word[i] == '<OUT>':
            token = 'out'
        else:
            token = ' ' + answersints2word[i]
        answer += token
        if token == '.':
            break
    print('ChatBot: ' + answer)
```

> *Interacting with our chatbot-*

```
################Start Chatting Below########################

You: what are you doing?
ChatBot: just been a good time

You: how is the weather?
ChatBot: is that a thing

You: is it cold there?
ChatBot: its a good one

You: great! then have fun
ChatBot: thanks for the retweet

You: have a good day
ChatBot: you are a good guy

You: thank you so much
ChatBot: youre welcome
```

Machine Learning Chatbots Opencv Dataset Conversational UI

About Help Legal