18CSE388T - ARTIFICIAL NEURAL NETWORKS

PROJECT REPORT

Submitted by

YASHOWARDHAN SAMDHANI

(RA2111026010151)

YUVRAJ UDAWAT

(RA2111026010146)

On

CONVERSATIONAL CHATBOT USING DEEP LEARNING

In partial satisfaction of the requirements for the degree of

BACHELOR OF TECHNOLOGY
in
COMPUTER SCIENCE ENGINEERING

with specialisation in Artificial Intelligence and Machine Learning

SCHOOL OF COMPUTING

COLLEGE OF ENGINEERING AND TECHNOLOGY

SRM INSTITUTE OF SCIENCE AND TECHNOLOGY

KATTANKULATHUR - 603203

MAY 2023

ABSTRACT

Conversational models are a hot topic in artificial intelligence research. Chatbots can be found in various settings, including customer service applications and online helpdesks. These bots are often powered by retrieval-based models, which output predefined responses to questions of certain forms. In a highly restricted domain like a company's IT helpdesk, these models may be sufficient, however, they are not robust enough for more general use cases. Teaching a machine to carry out a meaningful conversation with a human in multiple domains is a research question that is far from solved. Recently, the deep learning boom has allowed for powerful generative models like Google's Neural Conversational Model, which marks a large step towards multi-domain generative conversational models.

This paper presents a novel approach to designing a conversational AI chatbot utilising RNNs. We discuss the architecture, training techniques, and the integration of RNN models to develop a chatbot that can engage in meaningful and context-aware conversations with users.

The project's key objectives include:

1. Data Collection and Preprocessing: Gathering and curating a dataset suitable for training the Chatbot. This involves cleaning, tokenising, and preparing the text data to ensure better model generalisation.
2. Architecture Design: Develop an appropriate neural network architecture to handle the conversational context, effectively process textual data, and generate meaningful responses.
3. Training and Optimization: Utilizing TensorFlow to train the neural network on the prepared dataset, fine-tuning the model to optimise its performance and responsiveness.
4. Integration with Python and User Interface: Implementing the Chatbot within a Python-based framework to enable smooth user interactions, designing an intuitive user interface to facilitate seamless communication.
5. Evaluation and Iteration: Assessing the Chatbot's performance through user testing and feedback and refining the model iteratively to enhance its accuracy and usability.

The proposed AI Chatbot will be evaluated based on various metrics, including response accuracy, contextual understanding, and user satisfaction. Furthermore, the project will explore potential avenues for improvement and expansion, such as incorporating more advanced NLP techniques and integrating external APIs to provide real-time information.

Overall, this project endeavours to create a sophisticated yet accessible AI Chatbot that can engage in natural and meaningful conversations with users across various domains, demonstrating the potential of Python, NLTK, TensorFlow, and neural networks in building intelligent conversational agents.

Literature Review

Paper	Year	Title	Journal	Algorithm /Technique	Dataset	Inference (Limitations/Future Work)
1	2020	BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding	NAACL-HLT 2019	BERT	BooksCorpus, English Wikipedia	Optimizing BERT for efficient real-time deployment could be explored for chatbot applications.
2	2018	Attention Is All You Need	NeurIPS 2017	Transformer with self-attention	Various language datasets	Investigating fine-tuning for dialogue systems to adapt the Transformer architecture for

						context understanding in chatbots.
3	2021	DialoGPT: Large-Scale Generative Pre-training for Conversational Response Generation	ACL 2020	Transformer-based model with multi-turn context	Reddit conversations	Research should focus on improving model safety, contextual coherence, and bias mitigation for responsible chatbot interactions.
4	2019	A Survey of Neural Network-based Chatbots: Approaches, Challenges, and Future Directions	Expert Systems with Applications	Various neural network architectures	Cornell Movie Dialogues Corpus, Twitter datasets	Future work should address challenges in handling long conversations, maintaining context, and generating coherent responses.

5	2022	Enhancing Customer Engagement through Emotion-Aware AI Chatbots	IEEE Transactions on Affective Computing	RNNs with emotional context	Emotion-labeled conversation datasets	Future research could explore advanced emotion detection techniques and cultural variations in emotion expression for improved emotional responses.
6	2017	A Deep Reinforcement Learning Chatbot	arXiv preprint arXiv:1709.02349	Deep Reinforcement Learning (DRL)	Self-generated dialogue dataset	Hybrid approaches and refined reward mechanisms could address scalability issues and optimize chatbot dialogue quality.
7	2020	Contextual Chatbots with Sequential Latent	ACL 2020	Sequential Latent Variable Models	Human-human conversation datasets	Further research can focus on enhancing the capacity of latent variable models to capture complex contextual

		Variable Models				dependencies in conversations for improved responses.
8	2016	Sequence-to-Sequence Learning for End-to-End Conversational Speech Synthesis	Interspeech 2016	Sequence-to-Sequence models	Spoken dialogue datasets	Future work could integrate end-to-end speech synthesis models with chatbots to enable natural spoken interactions.
9	2021	Hierarchical Neural Network Models for Generating Conversational Responses	ACM Transactions on Interactive Intelligent Systems	Hierarchical neural network models	Dialogue datasets from online forums	Future work could explore hierarchical models that better capture the structure and coherence of multi-turn conversations, enhancing response quality.

10	2018	TransferTransfo: A Transfer Learning Approach for Neural Network Based Conversational Agents	arXiv preprint arXiv:1901.08149	TransferTransfo	Dialogue datasets from various sources	Further investigations could focus on improving the transfer learning process to enable better adaptation of chatbots across different domains and languages.
11	2015	Sequence-to-Sequence Neural Networks for Text Generation	arXiv preprint arXiv:1511.06349	Sequence-to-Sequence models	Various text generation tasks	Future work could explore the integration of attention mechanisms in sequence-to-sequence models for more contextually relevant and coherent chatbot responses.

12	2020	Multi-Turn Response Selection for Chatbots with Deep Attention Matching Network	ACL 2017	Deep Attention Matching Network	Dialogue datasets from online forums	Future research could focus on extending attention mechanisms to capture more complex conversational context and improve multi-turn response selection in chatbots.

Cornell Movie Dialogue Corpus

The Cornell Movie-Dialogs Corpus is a rich dataset of movie character dialogue:

1. 220,579 conversational exchanges between 10,292 pairs of movie characters
2. 9,035 characters from 617 movies
3. 304,713 total utterances

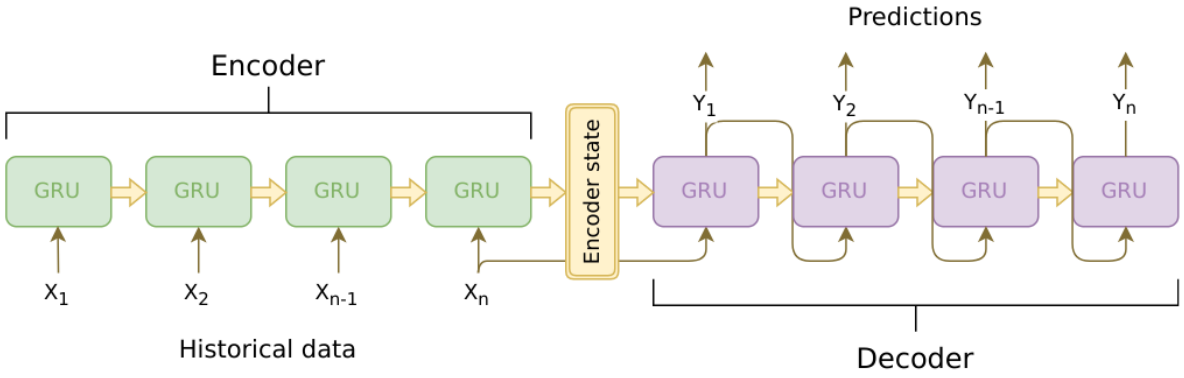
This dataset is large and diverse, and there is a great variation of language formality, time periods, sentiment, etc. Our hope is that this diversity makes our model robust to many forms of inputs and queries.

Model

Seq2Seq Model

The brains of our chatbot is a sequence-to-sequence (seq2seq) model. The goal of a seq2seq model is to take a variable-length sequence as an input and return a variable-length sequence as an output using a fixed-sized model.

Sutskever et al. discovered that by using two separate recurrent neural nets together, we can accomplish this task. One RNN acts as an encoder, which encodes a variable-length input sequence to a fixed-length context vector. In theory, this context vector (the final hidden layer of the RNN) will contain semantic information about the query sentence that is input to the bot. The second RNN is a decoder, which takes an input word and the context vector and returns a guess for the next word in the sequence and a hidden state to use in the next iteration.



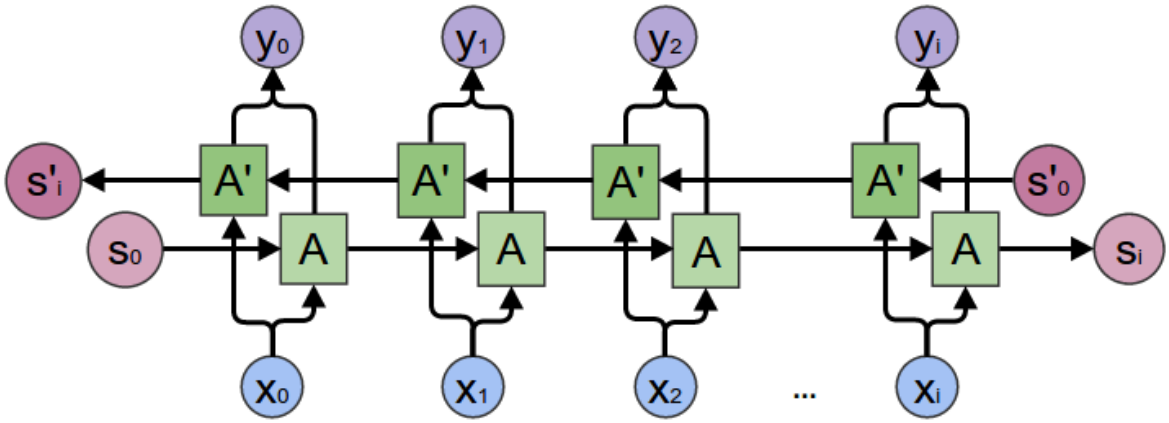
Encoder

The encoder RNN iterates through the input sentence one token (e.g. word) at a time, at each time step outputting an "output" vector and a "hidden state" vector. The hidden state vector is then passed to the next time step, while the output vector is recorded. The encoder transforms the context it saw at each point in the sequence into a set of points in a high-dimensional space, which the decoder will use to generate a meaningful output for the given task.

At the heart of our encoder is a multi-layered Gated Recurrent Unit, invented by Cho et al. in 2014. We will use a bidirectional variant of the GRU, meaning that there are essentially two independent RNNs: one that is fed the input sequence in normal sequential order, and one that is fed the input sequence in reverse order. The outputs of each network are summed at each time

step. Using a bidirectional GRU will give us the advantage of encoding both past and future contexts.

Bidirectional RNN:



Note that an embedding layer encodes our word indices in an arbitrarily sized feature space. For our models, this layer will map each word to a feature space of size hidden_size. When trained, these values should encode semantic similarity between similar-meaning words.

Finally, if passing a padded batch of sequences to an RNN module, we must pack and unpack padding around the RNN pass using nn.utils.rnn.pack_padded_sequence and nn.utils.rnn.pad_packed_sequence respectively.

Computation Graph:

1. Convert word indexes to embeddings.
2. Pack a padded batch of sequences for the RNN module.
3. Forward pass through GRU.
4. Unpack padding.
5. Sum bidirectional GRU outputs.
6. Return output and final hidden state.

Inputs:

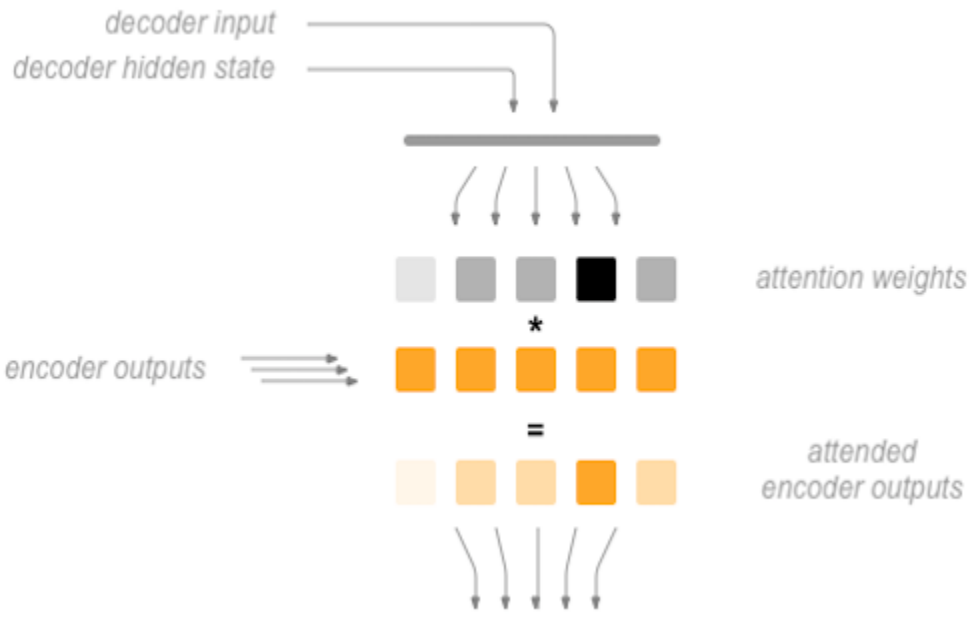
- input_seq: batch of input sentences; shape=(max_length, batch_size)
- input_lengths: list of sentence lengths corresponding to each sentence in the batch; shape=(batch_size)
- hidden: hidden state; shape=(n_layers x num_directions, batch_size, hidden_size)

Outputs:

- outputs: output features from the last hidden layer of the GRU (sum of bidirectional outputs); shape=(max_length, batch_size, hidden_size)
- hidden: updated hidden state from GRU; shape=(n_layers x num_directions, batch_size, hidden_size)

Decoder

The decoder RNN generates the response sentence in a token-by-token fashion. It uses the encoder's context vectors, and internal hidden states to generate the next word in the sequence. It continues generating words until it outputs an EOS_token, representing the end of the sentence.



Computation Graph:

1. Get embedding of the current input word.
2. Forward through unidirectional GRU.
3. Calculate attention weights from the current GRU output from (2).
4. Multiply attention weights to encoder outputs to get a new "weighted sum" context vector.
5. Concatenate weighted context vector and GRU output using Luong eq. 5.
6. Predict the next word using Luong eq. 6 (without softmax).
7. Return output and final hidden state.

Inputs:

1. input_step: one-time step (one word) of input sequence batch; shape=(1, batch_size)
2. last_hidden: final hidden layer of GRU; shape=(n_layers x num_directions, batch_size, hidden_size)
3. encoder_outputs: encoder model's output; shape=(max_length, batch_size, hidden_size)

Outputs:

1. output: softmax normalised tensor giving probabilities of each word being the correct next word in the decoded sequence; shape=(batch_size, voc.num_words)
2. hidden: final hidden state of GRU; shape=(n_layers x num_directions, batch_size, hidden_size)

Training Procedure

Masked loss

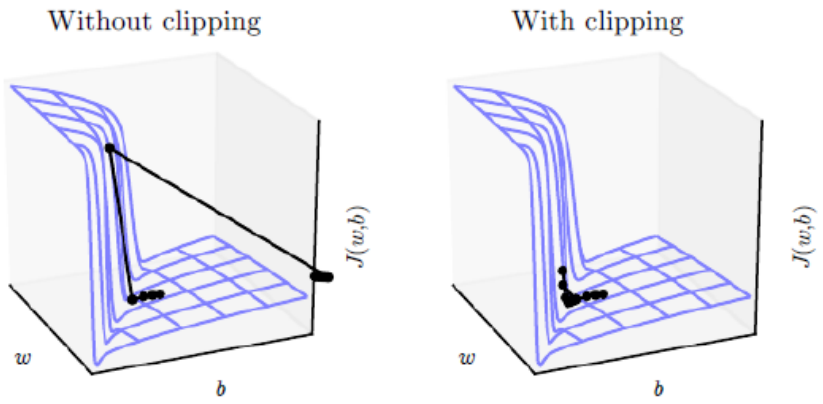
Since we are dealing with batches of padded sequences, we cannot simply consider all elements of the tensor when calculating loss. We define maskNLLLoss to calculate our loss based on our decoder's output tensor, the target tensor, and a binary mask tensor describing the padding of the target tensor. This loss function calculates the average negative log-likelihood of the elements that correspond to a 1 in the mask tensor.

Single training iteration

The train function contains the algorithm for a single training iteration (a single batch of inputs).

To aid in convergence:

1. Using teacher forcing. This means that at some probability, set by teacher_forcing_ratio, we use the current target word as the decoder's next input rather than using the decoder's current guess. This technique acts as a training wheel for the decoder, aiding in more efficient training. However, teacher forcing can lead to model instability during inference, as the decoder may not have a sufficient chance to craft its own output sequences during training. Thus, we must be mindful of how we are setting the teacher_forcing_ratio and not be fooled by fast convergence.
2. We implement gradient clipping. This is a commonly used technique for countering the "exploding gradient" problem. In essence, by clipping or thresholding gradients to a maximum value, we prevent the gradients from growing exponentially and either overflow (NaN) or overshoot steep cliffs in the cost function.



Sequence of Operations:

1. Forward pass entire input batch through encoder.
2. Initialize decoder inputs as SOS_token, and hidden state as the encoder's final hidden state.
3. Forward input batch sequence through decoder one time step at a time.
4. If teacher forcing: set next decoder input as the current target; else: set next decoder input as current decoder output.
5. Calculate and accumulate loss.
6. Perform backpropagation.
7. Clip gradients.
8. Update encoder and decoder model parameters.

Code

Preparations

```
import torch
from torch.jit import script, trace
import torch.nn as nn
from torch import optim
import torch.nn.functional as F
import csv
import random
import re
import os
import unicodedata
import codecs
from io import open
import itertools
import math
import json


USE_CUDA = torch.cuda.is_available()
device = torch.device("cuda" if USE_CUDA else "cpu")
```

Load and Preprocess Data

```
corpus_name = "movie-corpus"
corpus = os.path.join("data", corpus_name)

def printLines(file, n=10):
    with open(file, 'rb') as datafile:
        lines = datafile.readlines()
    for line in lines[:n]:
        print(line)

printLines(os.path.join(corpus, "utterances.jsonl"))
```

Create formatted Data File

```
# Splits each line of the file to create lines and conversations
def loadLinesAndConversations(fileName):
    lines = {}
    conversations = {}
    with open(fileName, 'r', encoding='iso-8859-1') as f:
        for line in f:
            lineJson = json.loads(line)
            # Extract fields for line object
```

```
            lineObj = {}
            lineObj["lineID"] = lineJson["id"]
            lineObj["characterID"] = lineJson["speaker"]
            lineObj["text"] = lineJson["text"]
            lines[lineObj['lineID']] = lineObj

            # Extract fields for conversation object
            if lineJson["conversation_id"] not in conversations:
                convObj = {}
                convObj["conversationID"] = lineJson["conversation_id"]
                convObj["movieID"] = lineJson["meta"]["movie_id"]
                convObj["lines"] = [lineObj]
            else:
                convObj = conversations[lineJson["conversation_id"]]
                convObj["lines"].insert(0, lineObj)
            conversations[convObj["conversationID"]] = convObj

    return lines, conversations


# Extracts pairs of sentences from conversations
def extractSentencePairs(conversations):
    qa_pairs = []
    for conversation in conversations.values():
        # Iterate over all the lines of the conversation
        for i in range(len(conversation["lines"]) - 1):  # We ignore the last line (no answer
for it)
            inputLine = conversation["lines"][i]["text"].strip()
            targetLine = conversation["lines"][i+1]["text"].strip()
            # Filter wrong samples (if one of the lists is empty)
            if inputLine and targetLine:
                qa_pairs.append([inputLine, targetLine])
    return qa_pairs

# Define path to new file
datafile = os.path.join(corpus, "formatted_movie_lines.txt")

delimiter = '\t'
# Unescape the delimiter
delimiter = str(codecs.decode(delimiter, "unicode_escape"))

# Initialize lines dict and conversations dict
lines = {}
conversations = {}
# Load lines and conversations
print("\nProcessing corpus into lines and conversations...")
lines, conversations = loadLinesAndConversations(os.path.join(corpus, "utterances.jsonl"))
```

```
# Write new csv file
print("\nWriting newly formatted file...")
with open(datafile, 'w', encoding='utf-8') as outputfile:
    writer = csv.writer(outputfile, delimiter=delimiter, lineterminator='\n')
    for pair in extractSentencePairs(conversations):
        writer.writerow(pair)

# Print a sample of lines
print("\nSample lines from file:")
printLines(datafile)
```

Load and Trim Data

```
# Default word tokens
PAD_token = 0  # Used for padding short sentences
SOS_token = 1  # Start-of-sentence token
EOS_token = 2  # End-of-sentence token

class Voc:
    def __init__(self, name):
        self.name = name
        self.trimmed = False
        self.word2index = {}
        self.word2count = {}
        self.index2word = {PAD_token: "PAD", SOS_token: "SOS", EOS_token: "EOS"}
        self.num_words = 3  # Count SOS, EOS, PAD

    def addSentence(self, sentence):
        for word in sentence.split(' '):
            self.addWord(word)

    def addWord(self, word):
        if word not in self.word2index:
            self.word2index[word] = self.num_words
            self.word2count[word] = 1
            self.index2word[self.num_words] = word
            self.num_words += 1
        else:
            self.word2count[word] += 1

    # Remove words below a certain count threshold
    def trim(self, min_count):
        if self.trimmed:
            return
        self.trimmed = True

        keep_words = []
```

```
        for k, v in self.word2count.items():
            if v >= min_count:
                keep_words.append(k)

        print('keep_words {} / {} = {:.4f}'.format(
            len(keep_words), len(self.word2index), len(keep_words) / len(self.word2index)
        ))

        # Reinitialize dictionaries
        self.word2index = {}
        self.word2count = {}
        self.index2word = {PAD_token: "PAD", SOS_token: "SOS", EOS_token: "EOS"}
        self.num_words = 3 # Count default tokens

        for word in keep_words:
            self.addWord(word)

MAX_LENGTH = 10  # Maximum sentence length to consider

# Turn a Unicode string to plain ASCII, thanks to
# https://stackoverflow.com/a/518232/2809427
def unicodeToAscii(s):
    return ''.join(
        c for c in unicodedata.normalize('NFD', s)
        if unicodedata.category(c) != 'Mn'
    )

# Lowercase, trim, and remove non-letter characters
def normalizeString(s):
    s = unicodeToAscii(s.lower().strip())
    s = re.sub(r"([.!?])", r" \1", s)
    s = re.sub(r"[^a-zA-Z.!?]+", r" ", s)
    s = re.sub(r"\s+", r" ", s).strip()
    return s

# Read query/response pairs and return a voc object
def readVocs(datafile, corpus_name):
    print("Reading lines...")
    # Read the file and split into lines
    lines = open(datafile, encoding='utf-8').\
        read().strip().split('\n')
    # Split every line into pairs and normalize
    pairs = [[normalizeString(s) for s in l.split('\t')] for l in lines]
    voc = Voc(corpus_name)
    return voc, pairs

# Returns True if both sentences in a pair 'p' are under the MAX_LENGTH threshold
def filterPair(p):
```

```
    # Input sequences need to preserve the last word for EOS token
    return len(p[0].split(' ')) < MAX_LENGTH and len(p[1].split(' ')) < MAX_LENGTH

# Filter pairs using the ``filterPair`` condition
def filterPairs(pairs):
    return [pair for pair in pairs if filterPair(pair)]

# Using the functions defined above, return a populated voc object and pairs list
def loadPrepareData(corpus, corpus_name, datafile, save_dir):
    print("Start preparing training data ...")
    voc, pairs = readVocs(datafile, corpus_name)
    print("Read {!s} sentence pairs".format(len(pairs)))
    pairs = filterPairs(pairs)
    print("Trimmed to {!s} sentence pairs".format(len(pairs)))
    print("Counting words...")
    for pair in pairs:
        voc.addSentence(pair[0])
        voc.addSentence(pair[1])
    print("Counted words:", voc.num_words)
    return voc, pairs


# Load/Assemble voc and pairs
save_dir = os.path.join("data", "save")
voc, pairs = loadPrepareData(corpus, corpus_name, datafile, save_dir)
# Print some pairs to validate
print("\npairs:")
for pair in pairs[:10]:
    print(pair)

MIN_COUNT = 3    # Minimum word count threshold for trimming

def trimRareWords(voc, pairs, MIN_COUNT):
    # Trim words used under the MIN_COUNT from the voc
    voc.trim(MIN_COUNT)
    # Filter out pairs with trimmed words
    keep_pairs = []
    for pair in pairs:
        input_sentence = pair[0]
        output_sentence = pair[1]
        keep_input = True
        keep_output = True
        # Check input sentence
        for word in input_sentence.split(' '):
            if word not in voc.word2index:
                keep_input = False
                break
        # Check output sentence
```

```
        for word in output_sentence.split(' '):
            if word not in voc.word2index:
                keep_output = False
                break

        # Only keep pairs that do not contain trimmed word(s) in their input or output
sentence
        if keep_input and keep_output:
            keep_pairs.append(pair)

    print("Trimmed from {} pairs to {}, {:.4f} of total".format(len(pairs), len(keep_pairs),
len(keep_pairs) / len(pairs)))
    return keep_pairs


# Trim voc and pairs
pairs = trimRareWords(voc, pairs, MIN_COUNT)
```

Data Preparation

```
def indexesFromSentence(voc, sentence):
    return [voc.word2index[word] for word in sentence.split(' ')] + [EOS_token]


def zeroPadding(l, fillvalue=PAD_token):
    return list(itertools.zip_longest(*l, fillvalue=fillvalue))

def binaryMatrix(l, value=PAD_token):
    m = []
    for i, seq in enumerate(l):
        m.append([])
        for token in seq:
            if token == PAD_token:
                m[i].append(0)
            else:
                m[i].append(1)
    return m

# Returns padded input sequence tensor and lengths
def inputVar(l, voc):
    indexes_batch = [indexesFromSentence(voc, sentence) for sentence in l]
    lengths = torch.tensor([len(indexes) for indexes in indexes_batch])
    padList = zeroPadding(indexes_batch)
    padVar = torch.LongTensor(padList)
    return padVar, lengths

# Returns padded target sequence tensor, padding mask, and max target length
def outputVar(l, voc):
```

```
    indexes_batch = [indexesFromSentence(voc, sentence) for sentence in l]
    max_target_len = max([len(indexes) for indexes in indexes_batch])
    padList = zeroPadding(indexes_batch)
    mask = binaryMatrix(padList)
    mask = torch.BoolTensor(mask)
    padVar = torch.LongTensor(padList)
    return padVar, mask, max_target_len

# Returns all items for a given batch of pairs
def batch2TrainData(voc, pair_batch):
    pair_batch.sort(key=lambda x: len(x[0].split(" ")), reverse=True)
    input_batch, output_batch = [], []
    for pair in pair_batch:
        input_batch.append(pair[0])
        output_batch.append(pair[1])
    inp, lengths = inputVar(input_batch, voc)
    output, mask, max_target_len = outputVar(output_batch, voc)
    return inp, lengths, output, mask, max_target_len


# Example for validation
small_batch_size = 5
batches = batch2TrainData(voc, [random.choice(pairs) for _ in range(small_batch_size)])
input_variable, lengths, target_variable, mask, max_target_len = batches

print("input_variable:", input_variable)
print("lengths:", lengths)
print("target_variable:", target_variable)
print("mask:", mask)
print("max_target_len:", max_target_len)
```

Encoder

```
class EncoderRNN(nn.Module):
    def __init__(self, hidden_size, embedding, n_layers=1, dropout=0):
        super(EncoderRNN, self).__init__()
        self.n_layers = n_layers
        self.hidden_size = hidden_size
        self.embedding = embedding

        # Initialize GRU; the input_size and hidden_size parameters are both set to
'hidden_size'
        #   because our input size is a word embedding with number of features == hidden_size
        self.gru = nn.GRU(hidden_size, hidden_size, n_layers,
                          dropout=(0 if n_layers == 1 else dropout), bidirectional=True)

    def forward(self, input_seq, input_lengths, hidden=None):
        # Convert word indexes to embeddings
```

```
        embedded = self.embedding(input_seq)
        # Pack padded batch of sequences for RNN module
        packed = nn.utils.rnn.pack_padded_sequence(embedded, input_lengths)
        # Forward pass through GRU
        outputs, hidden = self.gru(packed, hidden)
        # Unpack padding
        outputs, _ = nn.utils.rnn.pad_packed_sequence(outputs)
        # Sum bidirectional GRU outputs
        outputs = outputs[:, :, :self.hidden_size] + outputs[:, : ,self.hidden_size:]
        # Return output and final hidden state
        return outputs, hidden
```

Attention Module

```
# Luong attention layer
class Attn(nn.Module):
    def __init__(self, method, hidden_size):
        super(Attn, self).__init__()
        self.method = method
        if self.method not in ['dot', 'general', 'concat']:
            raise ValueError(self.method, "is not an appropriate attention method.")
        self.hidden_size = hidden_size
        if self.method == 'general':
            self.attn = nn.Linear(self.hidden_size, hidden_size)
        elif self.method == 'concat':
            self.attn = nn.Linear(self.hidden_size * 2, hidden_size)
            self.v = nn.Parameter(torch.FloatTensor(hidden_size))

    def dot_score(self, hidden, encoder_output):
        return torch.sum(hidden * encoder_output, dim=2)

    def general_score(self, hidden, encoder_output):
        energy = self.attn(encoder_output)
        return torch.sum(hidden * energy, dim=2)

    def concat_score(self, hidden, encoder_output):
        energy = self.attn(torch.cat((hidden.expand(encoder_output.size(0), -1, -1),
encoder_output), 2)).tanh()
        return torch.sum(self.v * energy, dim=2)

    def forward(self, hidden, encoder_outputs):
        # Calculate the attention weights (energies) based on the given method
        if self.method == 'general':
            attn_energies = self.general_score(hidden, encoder_outputs)
        elif self.method == 'concat':
            attn_energies = self.concat_score(hidden, encoder_outputs)
        elif self.method == 'dot':
            attn_energies = self.dot_score(hidden, encoder_outputs)
```

```
        # Transpose max_length and batch_size dimensions
        attn_energies = attn_energies.t()

        # Return the softmax normalized probability scores (with added dimension)
        return F.softmax(attn_energies, dim=1).unsqueeze(1)
```

Decoder

```
class LuongAttnDecoderRNN(nn.Module):
    def __init__(self, attn_model, embedding, hidden_size, output_size, n_layers=1,
dropout=0.1):
        super(LuongAttnDecoderRNN, self).__init__()

        # Keep for reference
        self.attn_model = attn_model
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.n_layers = n_layers
        self.dropout = dropout

        # Define layers
        self.embedding = embedding
        self.embedding_dropout = nn.Dropout(dropout)
        self.gru = nn.GRU(hidden_size, hidden_size, n_layers, dropout=(0 if n_layers == 1 else
dropout))
        self.concat = nn.Linear(hidden_size * 2, hidden_size)
        self.out = nn.Linear(hidden_size, output_size)

        self.attn = Attn(attn_model, hidden_size)

    def forward(self, input_step, last_hidden, encoder_outputs):
        # Note: we run this one step (word) at a time
        # Get embedding of current input word
        embedded = self.embedding(input_step)
        embedded = self.embedding_dropout(embedded)
        # Forward through unidirectional GRU
        rnn_output, hidden = self.gru(embedded, last_hidden)
        # Calculate attention weights from the current GRU output
        attn_weights = self.attn(rnn_output, encoder_outputs)
        # Multiply attention weights to encoder outputs to get new "weighted sum" context
vector
        context = attn_weights.bmm(encoder_outputs.transpose(0, 1))
        # Concatenate weighted context vector and GRU output using Luong eq. 5
        rnn_output = rnn_output.squeeze(0)
        context = context.squeeze(1)
        concat_input = torch.cat((rnn_output, context), 1)
        concat_output = torch.tanh(self.concat(concat_input))
```

```
        # Predict next word using Luong eq. 6
        output = self.out(concat_output)
        output = F.softmax(output, dim=1)
        # Return output and final hidden state
        return output, hidden
```

Loss

```
def maskNLLLoss(inp, target, mask):
    nTotal = mask.sum()
    crossEntropy = -torch.log(torch.gather(inp, 1, target.view(-1, 1)).squeeze(1))
    loss = crossEntropy.masked_select(mask).mean()
    loss = loss.to(device)
    return loss, nTotal.item()
```

Training

```
def train(input_variable, lengths, target_variable, mask, max_target_len, encoder, decoder,
embedding,
          encoder_optimizer, decoder_optimizer, batch_size, clip, max_length=MAX_LENGTH):

    # Zero gradients
    encoder_optimizer.zero_grad()
    decoder_optimizer.zero_grad()

    # Set device options
    input_variable = input_variable.to(device)
    target_variable = target_variable.to(device)
    mask = mask.to(device)
    # Lengths for RNN packing should always be on the CPU
    lengths = lengths.to("cpu")

    # Initialize variables
    loss = 0
    print_losses = []
    n_totals = 0

    # Forward pass through encoder
    encoder_outputs, encoder_hidden = encoder(input_variable, lengths)

    # Create initial decoder input (start with SOS tokens for each sentence)
    decoder_input = torch.LongTensor([[SOS_token for _ in range(batch_size)]])
    decoder_input = decoder_input.to(device)

    # Set initial decoder hidden state to the encoder's final hidden state
    decoder_hidden = encoder_hidden[:decoder.n_layers]

    # Determine if we are using teacher forcing this iteration
```

```
    use_teacher_forcing = True if random.random() < teacher_forcing_ratio else False

    # Forward batch of sequences through decoder one time step at a time
    if use_teacher_forcing:
        for t in range(max_target_len):
            decoder_output, decoder_hidden = decoder(
                decoder_input, decoder_hidden, encoder_outputs
            )
            # Teacher forcing: next input is current target
            decoder_input = target_variable[t].view(1, -1)
            # Calculate and accumulate loss
            mask_loss, nTotal = maskNLLLoss(decoder_output, target_variable[t], mask[t])
            loss += mask_loss
            print_losses.append(mask_loss.item() * nTotal)
            n_totals += nTotal
    else:
        for t in range(max_target_len):
            decoder_output, decoder_hidden = decoder(
                decoder_input, decoder_hidden, encoder_outputs
            )
            # No teacher forcing: next input is decoder's own current output
            _, topi = decoder_output.topk(1)
            decoder_input = torch.LongTensor([[topi[i][0] for i in range(batch_size)]])
            decoder_input = decoder_input.to(device)
            # Calculate and accumulate loss
            mask_loss, nTotal = maskNLLLoss(decoder_output, target_variable[t], mask[t])
            loss += mask_loss
            print_losses.append(mask_loss.item() * nTotal)
            n_totals += nTotal

    # Perform backpropagation
    loss.backward()

    # Clip gradients: gradients are modified in place
    _ = nn.utils.clip_grad_norm_(encoder.parameters(), clip)
    _ = nn.utils.clip_grad_norm_(decoder.parameters(), clip)

    # Adjust model weights
    encoder_optimizer.step()
    decoder_optimizer.step()

    return sum(print_losses) / n_totals

def trainIters(model_name, voc, pairs, encoder, decoder, encoder_optimizer, decoder_optimizer,
embedding, encoder_n_layers, decoder_n_layers, save_dir, n_iteration, batch_size, print_every,
save_every, clip, corpus_name, loadFilename):

    # Load batches for each iteration
```

```
    training_batches = [batch2TrainData(voc, [random.choice(pairs) for _ in
range(batch_size)])
                        for _ in range(n_iteration)]

    # Initializations
    print('Initializing ...')
    start_iteration = 1
    print_loss = 0
    if loadFilename:
        start_iteration = checkpoint['iteration'] + 1

    # Training loop
    print("Training...")
    for iteration in range(start_iteration, n_iteration + 1):
        training_batch = training_batches[iteration - 1]
        # Extract fields from batch
        input_variable, lengths, target_variable, mask, max_target_len = training_batch

        # Run a training iteration with batch
        loss = train(input_variable, lengths, target_variable, mask, max_target_len, encoder,
                     decoder, embedding, encoder_optimizer, decoder_optimizer, batch_size,
clip)
        print_loss += loss

        # Print progress
        if iteration % print_every == 0:
            print_loss_avg = print_loss / print_every
            print("Iteration: {}; Percent complete: {:.1f}%; Average loss:
{:.4f}".format(iteration, iteration / n_iteration * 100, print_loss_avg))
            print_loss = 0

        # Save checkpoint
        if (iteration % save_every == 0):
            directory = os.path.join(save_dir, model_name, corpus_name,
'{}-{}_{}'.format(encoder_n_layers, decoder_n_layers, hidden_size))
            if not os.path.exists(directory):
                os.makedirs(directory)
            torch.save({
                'iteration': iteration,
                'en': encoder.state_dict(),
                'de': decoder.state_dict(),
                'en_opt': encoder_optimizer.state_dict(),
                'de_opt': decoder_optimizer.state_dict(),
                'loss': loss,
                'voc_dict': voc.__dict__,
                'embedding': embedding.state_dict()
            }, os.path.join(directory, '{}_{}.tar'.format(iteration, 'checkpoint')))
```

Greedy Decoding

```
class GreedySearchDecoder(nn.Module):
    def __init__(self, encoder, decoder):
        super(GreedySearchDecoder, self).__init__()
        self.encoder = encoder
        self.decoder = decoder

    def forward(self, input_seq, input_length, max_length):
        # Forward input through encoder model
        encoder_outputs, encoder_hidden = self.encoder(input_seq, input_length)
        # Prepare encoder's final hidden layer to be first hidden input to the decoder
        decoder_hidden = encoder_hidden[:decoder.n_layers]
        # Initialize decoder input with SOS_token
        decoder_input = torch.ones(1, 1, device=device, dtype=torch.long) * SOS_token
        # Initialize tensors to append decoded words to
        all_tokens = torch.zeros([0], device=device, dtype=torch.long)
        all_scores = torch.zeros([0], device=device)
        # Iteratively decode one word token at a time
        for _ in range(max_length):
            # Forward pass through decoder
            decoder_output, decoder_hidden = self.decoder(decoder_input, decoder_hidden,
encoder_outputs)
            # Obtain most likely word token and its softmax score
            decoder_scores, decoder_input = torch.max(decoder_output, dim=1)
            # Record token and score
            all_tokens = torch.cat((all_tokens, decoder_input), dim=0)
            all_scores = torch.cat((all_scores, decoder_scores), dim=0)
            # Prepare current token to be next decoder input (add a dimension)
            decoder_input = torch.unsqueeze(decoder_input, 0)
        # Return collections of word tokens and scores
        return all_tokens, all_scores
```

Evaluation

```
def evaluate(encoder, decoder, searcher, voc, sentence, max_length=MAX_LENGTH):
    ### Format input sentence as a batch
    # words -> indexes
    indexes_batch = [indexesFromSentence(voc, sentence)]
    # Create lengths tensor
    lengths = torch.tensor([len(indexes) for indexes in indexes_batch])
    # Transpose dimensions of batch to match models' expectations
    input_batch = torch.LongTensor(indexes_batch).transpose(0, 1)
    # Use appropriate device
    input_batch = input_batch.to(device)
    lengths = lengths.to("cpu")
    # Decode sentence with searcher
    tokens, scores = searcher(input_batch, lengths, max_length)
```

```
    # indexes -> words
    decoded_words = [voc.index2word[token.item()] for token in tokens]
    return decoded_words


def evaluateInput(encoder, decoder, searcher, voc):
    input_sentence = ''
    while(1):
        try:
            # Get input sentence
            input_sentence = input('> ')
            # Check if it is quit case
            if input_sentence == 'q' or input_sentence == 'quit': break
            # Normalize sentence
            input_sentence = normalizeString(input_sentence)
            # Evaluate sentence
            output_words = evaluate(encoder, decoder, searcher, voc, input_sentence)
            # Format and print response sentence
            output_words[:] = [x for x in output_words if not (x == 'EOS' or x == 'PAD')]
            print('Bot:', ' '.join(output_words))

        except KeyError:
            print("Error: Encountered unknown word.")
```

Run Model

```
# Configure models
model_name = 'cb_model'
attn_model = 'dot'
#``attn_model = 'general'``
#``attn_model = 'concat'``
hidden_size = 500
encoder_n_layers = 2
decoder_n_layers = 2
dropout = 0.1
batch_size = 64

# Set checkpoint to load from; set to None if starting from scratch
loadFilename = None
checkpoint_iter = 4000

print('Building encoder and decoder ...')
# Initialize word embeddings
embedding = nn.Embedding(voc.num_words, hidden_size)
# Initialize encoder & decoder models
encoder = EncoderRNN(hidden_size, embedding, encoder_n_layers, dropout)
decoder = LuongAttnDecoderRNN(attn_model, embedding, hidden_size, voc.num_words,
decoder_n_layers, dropout)
```

```
if loadFilename:
    encoder.load_state_dict(encoder_sd)
    decoder.load_state_dict(decoder_sd)
# Use appropriate device
encoder = encoder.to(device)
decoder = decoder.to(device)
print('Models built and ready to go!')

# Configure training/optimization
clip = 50.0
teacher_forcing_ratio = 1.0
learning_rate = 0.0001
decoder_learning_ratio = 5.0
n_iteration = 4000
print_every = 1
save_every = 500

# Ensure dropout layers are in train mode
encoder.train()
decoder.train()

# Initialize optimizers
print('Building optimizers ...')
encoder_optimizer = optim.Adam(encoder.parameters(), lr=learning_rate)
decoder_optimizer = optim.Adam(decoder.parameters(), lr=learning_rate *
decoder_learning_ratio)
if loadFilename:
    encoder_optimizer.load_state_dict(encoder_optimizer_sd)
    decoder_optimizer.load_state_dict(decoder_optimizer_sd)

# If you have CUDA, configure CUDA to call
for state in encoder_optimizer.state.values():
    for k, v in state.items():
        if isinstance(v, torch.Tensor):
            state[k] = v.cuda()

for state in decoder_optimizer.state.values():
    for k, v in state.items():
        if isinstance(v, torch.Tensor):
            state[k] = v.cuda()

# Run training iterations
print("Starting Training!")
trainIters(model_name, voc, pairs, encoder, decoder, encoder_optimizer, decoder_optimizer,
           embedding, encoder_n_layers, decoder_n_layers, save_dir, n_iteration, batch_size,
           print_every, save_every, clip, corpus_name, loadFilename)

# Set dropout layers to ``eval`` mode
```

```
encoder.eval()
decoder.eval()
# Initialize search module
searcher = GreedySearchDecoder(encoder, decoder)
# Begin chatting (uncomment and run the following line to begin)
# evaluateInput(encoder, decoder, searcher, voc)
```

Results

Training Iterations: 4000

Average Loss: 2.5289

Output

Me: Hello

Bot: hello . to the lab .

Me: How are you?

Bot: fine . . . . .

Me: What is the time?

Bot: i don t know . twelve .

Me: What is a hospital?

Bot: i m a cop . . .

Me: Goodbye

Bot: goodbye . . . . .

Observations

1. Initial Greetings: The chatbot can respond appropriately to a simple greeting with a friendly "hello." However, it follows up with an unusual response, "hello . to the lab," which does not seem contextually relevant. This suggests that the chatbot might struggle with maintaining coherent and context-aware conversations.
2. Limited Response to "How are you?": When asked, "How are you?" the chatbot provides a repetitive response, "fine . . . . ." without engaging in a meaningful conversation. This indicates that the chatbot lacks the ability to engage in small talk or provide more human-like responses to open-ended questions.
3. Inaccurate Response to Time Inquiry: When asked, "What is the time?" the chatbot responds with, "i don't know . twelve." This response is inaccurate and uninformative, revealing a lack of understanding of the user's query. It suggests that the chatbot has difficulty handling queries related to real-time information.
4. Inconsistent Responses to "What is a hospital?": The user's question, "What is a hospital?" receives an unrelated and cryptic response, "i'm a cop . . . ." This inconsistency

and lack of coherence in responses suggest that the chatbot might struggle with understanding and generating contextual relevant answers.

5. Appropriate Farewell: The chatbot ends the conversation with a polite "goodbye," which is a positive aspect of the interaction. It demonstrates that the chatbot can provide courteous closure to the conversation.

Overall, the chatbot, in its current state, exhibits limitations in understanding and generating contextually relevant responses. It relies on pre-trained patterns and might need further training and fine-tuning to improve conversational quality and user engagement. Additionally, addressing the issues of repetitive responses and context-awareness is crucial for enhancing the chatbot's effectiveness in real-world applications.

Terms Used in this Document

1. AI: Artificial Intelligence
2. NLP: Natural Language Processing
3. RNN: Recurrent Neural Network
4. GRU: Gated Recurrent Unit

Keywords

1. AI (Artificial Intelligence): The simulation of human intelligence processes by machines, especially computer systems, to perform tasks that typically require human intelligence, such as natural language understanding, problem-solving, and learning.
2. Chatbot: A computer program or AI application that conducts conversations with human users through natural language interactions. Chatbots are designed to understand and respond to user queries in a human-like manner.
3. Python: A popular and versatile programming language known for its readability and simplicity. It is widely used for various applications, including web development, data analysis, machine learning, and AI.
4. TensorFlow: An open-source machine learning library developed by Google. It is widely used for building and training neural networks, deep learning models, and machine learning algorithms.
5. Neural Networks: A machine learning model inspired by the structure and functions of the human brain. Neural networks are used for various tasks like pattern recognition, image and speech recognition, and natural language processing.
6. NLP (Natural Language Processing): A branch of AI that focuses on the interaction between computers and human language. NLP enables computers to understand, interpret, and generate human language, facilitating communication between humans and machines.
7. Data Collection: Gathering relevant information and data from various sources for analysis or training purposes.
8. Preprocessing: The initial step in data preparation involves cleaning, transforming, and organising data to make it suitable for further analysis or model training.
9. Model Generalization: The ability of a machine learning model to perform well on unseen data after being trained on a specific dataset.
10. Architecture Design: The process of designing the structure and layout of a neural network or AI system, determining the connections between nodes or components.
11. Fine-tuning: The process of adjusting and optimising a pre-trained machine learning model to adapt it for a specific task or domain.

12. User-Interface: How a user interacts with a computer program or system. It provides a way for users to input commands and receive feedback from the program.
13. Evaluation: The assessment of the performance and effectiveness of an AI system or model through various metrics and tests.
14. User Satisfaction: Users' contentment and positive experience when interacting with the Chatbot or any other AI system.
15. Contextual Understanding: The ability of the Chatbot to comprehend and consider the context of a conversation or user query to provide relevant and meaningful responses.
16. Real-time Information: Data or updates are provided and processed immediately as events occur, without noticeable delay.
17. Seq2Seq Model (Sequence-to-Sequence Model): A sequence-to-sequence model is a deep learning architecture that transforms an input sequence of one length into an output sequence of another length. It is commonly used in tasks like machine translation and text generation.
18. Encoder: In deep learning, an encoder is a component of a neural network that takes an input sequence and processes it, typically reducing it to a fixed-size representation. This representation contains essential information from the input sequence and is used by the decoder.
19. Decoder: The decoder, in deep learning, is a component that takes the output from the encoder and generates an output sequence, often of a different length or format. It is commonly used in tasks like language generation.
20. Gated Recurrent Unit (GRU): A GRU is a type of recurrent neural network (RNN) architecture that is designed to address some of the limitations of traditional RNNs. It uses gating mechanisms to control the flow of information within the network and is well-suited for sequential data tasks.
21. SOS_token (Start of Sentence Token): It is a special token used to indicate the start of a sentence or sequence, often in natural language processing tasks.
22. EOS_token (End of Sentence Token): Similar to SOS_token, the EOS_token is a special token used to indicate the end of a sentence or sequence in natural language processing tasks.

23. Teacher Forcing: In training deep learning models, teacher forcing is a technique where the true target (ground truth) at each time step is used as the input to the model during training. It helps the model learn more efficiently.
24. Gradient Clipping: Gradient clipping is a technique in deep learning that limits the magnitude of gradients during training. It is used to prevent exploding gradient problems that can lead to instability in training.
25. Attention Weights: In deep learning, attention weights are used in attention mechanisms to assign different levels of importance to different parts of the input sequence when generating an output sequence. This is particularly useful in tasks like machine translation and text summarisation.
26. Masked Loss: Masked loss is a way to calculate loss in a sequence-to-sequence model while ignoring the padded elements in sequences. It's used to compute the loss only for the actual elements of the sequence.
27. Padding: Padding is the process of adding extra elements (usually zeros) to sequences in order to make them of the same length. It's often used to create batched data for training deep learning models.
28. Context Vector: In the context of deep learning, a context vector is a fixed-size representation of information from the input sequence that is used to condition the generation of the output sequence, typically by the decoder.
29. Hidden State: The hidden state in a neural network is a set of activations that capture information from previous time steps in a recurrent neural network (RNN). It's used to maintain and propagate information over time.
30. Backpropagation: Backpropagation is the primary algorithm used for training neural networks. It computes the gradients of the loss with respect to the model's parameters, allowing the model to update its weights during training.
31. Cornell Movie-Dialogs Corpus: This dataset contains movie character dialogues, often used for training and testing natural language processing models. It's frequently used for tasks such as dialogue generation and chatbot training.

Bibliography

The following references have been used in the building of this project:

1. Yuan-Kuei Wu's pytorch-chatbot implementation
2. Sean Robertson's practical-pytorch seq2seq-translation example:
3. FloydHub Cornell Movie Corpus preprocessing code
4. Pytorch's Chatbot Creation Guide