AI Project - GRU based Seq2Seq Chatbot

Aakash Kaushik - RA1911032010001
Parikshit Kumar - RA1911032010002
Aryan Kargwal - RA1911032010023
Aditya Choudhury - RA1911032010021

Problem Statement

- To create a versatile, deployable non rule based chatbot.
- Used to answer user queries in bulk to handle multiple users at the same time.
- The chatbot can be further used for tasks like medical help.
- Integrating the chatbot into large scale services for quick and easy help to people using the service.

Features

- A simple UI around user queries.
- Versatility to handle different type of natural sentences.
- Modularity in the components to make easy changes without interfering with other parts.
- Simple solution to all immediate assistance needs.

Module Design

Module 1

- This is the Data Scraping and preparation Module.
- This module scrapes data needed to train the NLP model.
- And prepares the data to be fed in the model for training.

Dataset - cornell movie dialogs corpus

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

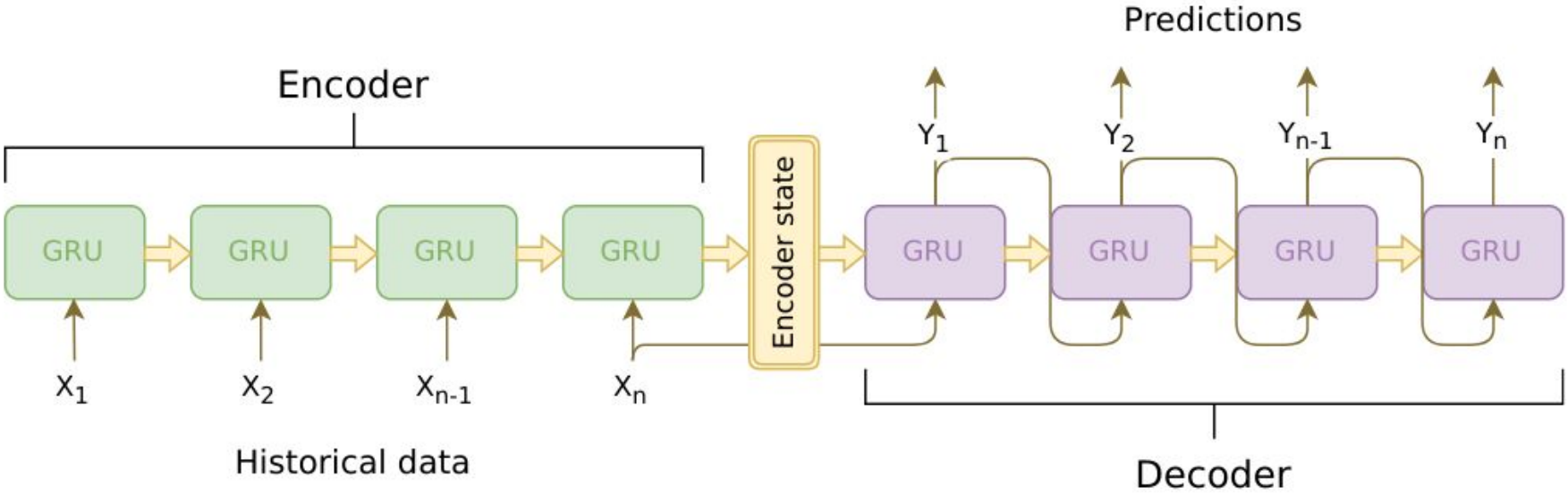
- 220,579 conversational exchanges between 10,292 pairs of movie characters
- involves 9,035 characters from 617 movies
- in total 304,713 utterances
- movie metadata included:
  - Genres
  - release year
  - IMDB rating
  - number of IMDB votes
  - IMDB rating
- character metadata included:
  - gender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

```python
def print_lines(file,n=10):
    """Shows some lines ftom the text file"""
    with open(file,'rb') as datafile:
        lines=datafile.readlines()
    for line in lines[:n]:
        print(line)

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

def load_lines(file,fields):
    """loads lines and splits then into fields and return a dict"""
    lines={}
    with open(file,"r",encoding="iso-8859-1") as f:
        for line in f:
            values=line.split(" +++$+++ ")
            line_obj={}
            for idx,field in enumerate(fields):
                line_obj[field]=values[idx]
            lines[line_obj[fields[0]]]=line_obj
    return lines

def load_conv(file,lines,fields):
    convs=[]
    with open(file,"r",encoding="iso-8859-1") as f:
        for line in f:
            values=line.split(" +++$+++ ")
            conv_obj={}
            for idx,field in enumerate(fields):
                conv_obj[field]=values[idx]
            line_id_pattern=re.compile('L[0-9]+')
            line_ids=line_id_pattern.findall(conv_obj[fields[-1]])
            conv_obj["lines"]=[]
            for line_id in line_ids:
                conv_obj["lines"].append(lines[line_id])
            convs.append(conv_obj)
    return convs

def sentence_pair_extract(convs):
    qa_pairs=[]
    for conv in convs:
        for i in range(len(conv["lines"])-1):
            input_line=conv["lines"][i]["text"].strip()
            target_line=conv["lines"][i+1]["text"].strip()

            if input_line and target_line:
                qa_pairs.append([input_line,target_line])
    return qa_pairs
```

Module 2

- Module 2 which is the main architecture comprises Gated recurrent units with attention.
- This is a encoder-decoder based architecture in which a user query or simply a sentence is passed to the encoder and that converts into into a tensor which is passed to the decoder and which thus gives an answer.

Architecture



Model

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.

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.

Decode

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. A common problem with a vanilla seq2seq decoder is that if we rely solely on the context vector to encode the entire input sequence’s meaning, it is likely that we will have information loss. This is especially the case when dealing with long input sequences, greatly limiting the capability of our decoder.

```python
#model part
class encoder_rnn(nn.Module):
    def __init__(self,hidden_size,embedding,n_layers=1,dropout=0):
        super(encoder_rnn,self).__init__()
        self.n_layers=n_layers
        self.hidden_size=hidden_size
        self.embedding=embedding

        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):
        embedded=self.embedding(input_seq)
        packed=nn.utils.rnn.pack_padded_sequence(embedded,input_lengths)
        outputs,hidden=self.gru(packed,hidden)
        outputs,_=nn.utils.rnn.pad_packed_sequence(outputs)
        outputs=outputs[:,:,:self.hidden_size]+outputs[:,:,self.hidden_size:]
        return outputs,hidden

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(energy*hidden,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):
        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":
```

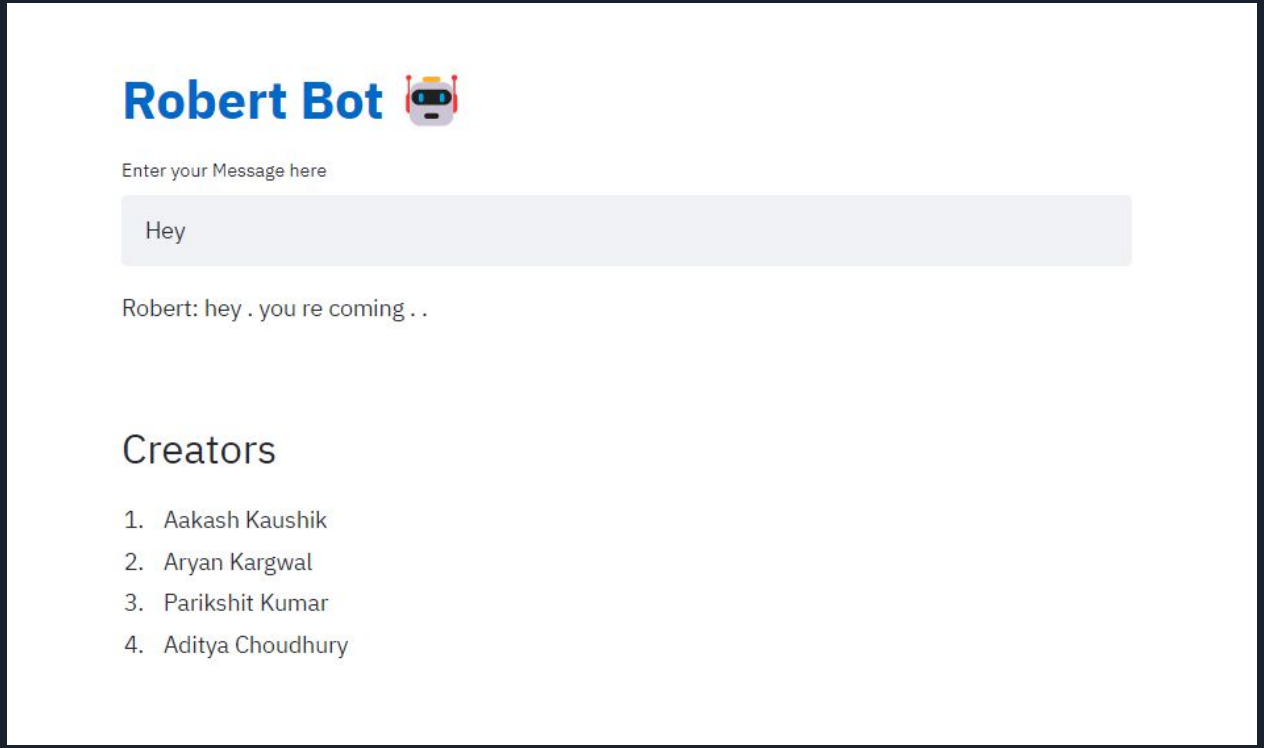
```python
class attn_decoder_rnn(nn.Module):
    def __init__(self,attn_model,embedding,hidden_size,output_size,n_layers=1,dropout=0.1):
        super(attn_decoder_rnn,self).__init__()
        self.attn_model=attn_model
        self.hidden_size=hidden_size
        self.output_size=output_size
      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

        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):
        embedded=self.embedding(input_step)
        embedded=self.embedding_dropout(embedded)
        rnn_output,hidden=self.gru(embedded,last_hidden)
        attn_weights=self.attn(rnn_output,encoder_outputs)
        context=attn_weights.bmm(encoder_outputs.transpose(0,1))
        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))
        output=self.out(concat_output)
        output=nn.functional.softmax(output,dim=1)

        return output,hidden
```

Streamlit - Frontend

app.py

```python
import streamlit as st
from bot import *

title = """
        <h1><a href="https://github.com/Aakash-kaushik/robert_bot">Robert Bot 🤖</a></h1>
        """

st.write("\n")
st.write("\n")

st.markdown(title, unsafe_allow_html=True)
user_input = st.text_input("Enter your Message here","Hey")
bot_output_list = eval_input(encoder, decoder, searcher, voc, user_input)
if bot_output_list != -1:
  bot_output_str=""
  for bot_output_word in bot_output_list:
      bot_output_str += bot_output_word
      bot_output_str += " "
  st.write("Robert: ", bot_output_str)
else:
  st.write("Robert: ", "Try something else human. 🕯 ")

st.write("\n")
st.write("\n")

html_string = """
  <h2> Creators </h2>
  <p align="center">
  <ol>
  <li> Aakash Kaushik </li>
  <li> Aryan Kargwal </li>
  <li> Parikshit Kumar </li>
  <li> Aditya Choudhury </li>
  </ol>
</p>
"""
st.markdown(html_string, unsafe_allow_html=True)

st.write("\n")
st.write("\n")
```

Project Screenshot



Deployment - Heroku

We have used the Heroku free tier for deployment:

- They offer a python container to run your python-based web apps which we have here.
- We have the backend which is our bot’s mind and training code.
- and the front which is written with the help of streamlit which helps build front-end apps all in pure python.

Project Demo