

1. NLTK Chatbot

A retrieval based chatbot using NLTK, Keras, Tensorflow (Keras requires Tensorflow as a dependency) and Python is used to code the English version of the chatbot. A generative based chatbot wouldn't be required in this scenario as the questions are **domain specific** and the users will only be asking questions mostly related to the dataset that we have gathered. Moreover, a generative based chatbot will be much harder to train and implement compared to a retrieval based chatbot.

The chatbot will be trained on the dataset which contains categories (intents), pattern and responses. A special recurrent neural network (**LSTM**) is used to classify which category the user's message belongs to and then will give a random response from the list of responses.

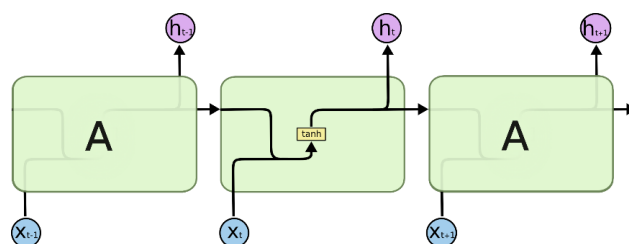
In addition to the above dependencies, we will also be using some helper modules.

1.1. Long Short Term Memory (LSTM) Networks

Long Short Term Memory networks are a special kind of Recurrent Neural Network (RNN), capable of learning long-term dependencies. They were introduced by **Hochreiter & Schmidhuber** (1997), and were refined and popularized by many. They work tremendously well on a large variety of problems, and are now widely used.

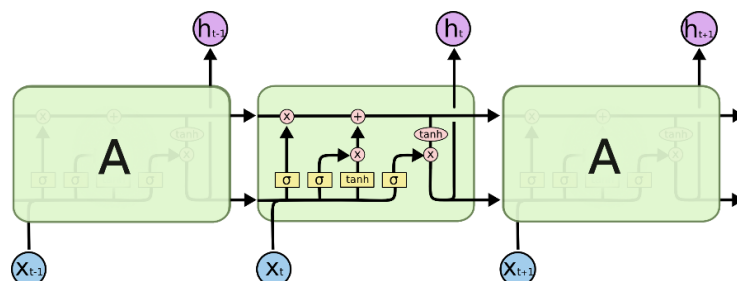
LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn.

All recurrent neural networks have the form of a chain of repeating modules of neural network. In standard RNNs, this repeating module will have a very **simple structure**, such as a single tanh layer.



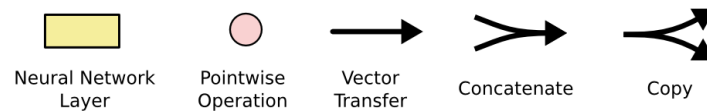
The repeating module in a standard RNN contains a single layer

LSTMs also have this chain like structure, but the repeating module has a different structure. Instead of having a single neural network layer, there are **four**, interacting in a very special way.



The repeating module in an LSTM contains four interacting layers

In the above diagram, each line carries an entire vector, from the output of one node to the inputs of others. The pink circles represent pointwise operations, like vector addition, while the yellow boxes are learned neural network layers. Lines merging denote concatenation, while a line forking denotes its content being copied and the copies going to different locations.



1.2. Natural Language Processing Toolkit (NLTK)

NLTK (Natural Language Toolkit) is a leading platform for building Python programs to work with **human language data**. NLTK has been called “a wonderful tool for teaching and working in, computational linguistics using Python,” and “an amazing library to play with natural language.”

a) Text Pre-Processing with NLTK

The main issue with text data is that it is all in text format (strings). However, the Machine learning algorithms need some sort of numerical feature vector in order to perform the task. So, before we start with any NLP project, we need to pre-process it to make it ideal for working. Basic **text pre-processing** includes:

- Converting the entire text into **uppercase or lowercase**, so that the algorithm does not treat the same words in different cases as different
- **Tokenization:** Tokenization is just the term used to describe the process of converting the normal text strings into a list of tokens i.e., words that we actually want. Sentence tokenizer can be used to find the list of sentences and Word tokenizer can be used to find the list of words in strings.
(Note: The NLTK data package includes a pre-trained Punkt tokenizer for English)
- Removing **Noise** i.e., everything that isn't in a standard number or letter.
- Removing **Stop words**. Sometimes, some extremely common words which would appear to be of little value in helping select documents matching a user need are excluded from the vocabulary entirely. These words are called *stop words*.
- **Stemming:** Stemming is the process of reducing inflected (or sometimes derived) words to their stem, base or root form — generally a written word form. Example if we were to stem the following words: “Stems”, “Stemming”, “Stemmed”, “and Stemtization”, the result would be a single word “stem”.

- **Lemmatization:** A slight variant of stemming is lemmatization. The major difference between these is, that, stemming can often create **non-existent words**, whereas lemmas are actual words. So, your root stem, meaning the word you end up with, is not something you can just look up in a dictionary, but you can look up a lemma. Examples of Lemmatization are that “run” is a base form for words like “running” or “ran” or that the word “better” and “good” are in the same lemma so they are considered the same.

b) Bag of Words

After the initial preprocessing phase, we need to transform text into a meaningful vector (or array) of numbers. The bag-of-words is a representation of text that describes the occurrence of words within a document. It involves two things:

- A vocabulary of known words.
- A measure of the presence of known words.

Why is it called a “bag” of words? That is because any information about the order or structure of words in the document is discarded and the model is only concerned with **whether the known words occur in the document, not where they occur in the document**.

The intuition behind the Bag of Words is that documents are similar if they have similar content. Also, we can learn something about the meaning of the document from its content alone.

For example, if our dictionary contains the words {Learning, is, the, not, great}, and we want to vectorize the text “Learning is great”, we would have the following vector: (1, 1, 0, 0, 1).

c) TF-IDF Approach

A problem with the Bag of Words approach is that **highly frequent words** start to dominate in the document (e.g., larger score), but may not contain as much “informational content”. Also, it will give more weight to longer documents than shorter documents.

One approach is to rescale the frequency of words by how often they appear in all documents so that the scores for frequent words like “the” that are also frequent across all documents are penalized. This approach to scoring is called **Term Frequency-Inverse Document Frequency**, or **TF-IDF** for short, where:

Term Frequency: is a scoring of the frequency of the word in the current document.

$$TF = (\text{Number of times term } t \text{ appears in a document}) / (\text{Number of terms in the document})$$

Inverse Document Frequency: is a scoring of how rare the word is across documents.

$$IDF = 1 + \log(N/n)$$
, where, N is the number of documents and n is the number of documents a term t has appeared in.

TF-IDF weight is a weight often used in **information retrieval** and **text mining**. This weight is a statistical measure used to evaluate how important a word is to a document in a collection or corpus.

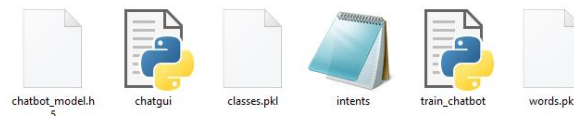
d) Cosine Similarity

TF-IDF is a transformation applied to texts to get two real-valued vectors in vector space. We can then obtain the **Cosine** similarity of any pair of vectors by taking their dot product and dividing that by the product of their norms. That yields the cosine of the angle between the vectors. **Cosine similarity** is a measure of similarity between two non-zero vectors. Using this formula, we can find out the similarity between any two documents d1 and d2.

$$\text{Cosine Similarity (d1, d2)} = \text{Dot product(d1, d2)} / ||\text{d1}|| * ||\text{d2}||$$

1.3. Creating the Chatbot

The type of files used are as follows:



- *Intents.json* – The data file which has predefined patterns and responses.
- *train_chatbot.py* – This python file contains the script to build the model and train the chatbot.
- *Words.pkl* – This is the pickle file in which the words Python object is stored that contains a list of our vocabulary.
- *Classes.pkl* – The classes pickle file contains the list of categories.
- *Chatbot_model.h5* – This is the trained model that contains information about the model and has weights of the neurons.
- *Chatgui.py* – This is the Python script in which GUI is implemented for the chatbot. Users can easily interact with the bot.

As discussed earlier, the following six steps will be used to develop the chatbot in python:

1.3.1 Importing and Loading the Data File

First, make a file called *train_chatbot.py*. We import the necessary packages for our chatbot and initialize the variables that we will use in our Python project.

The data file is in JSON format so we use the json package to parse the JSON file into Python.

```
import nltk
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
import json
```

```

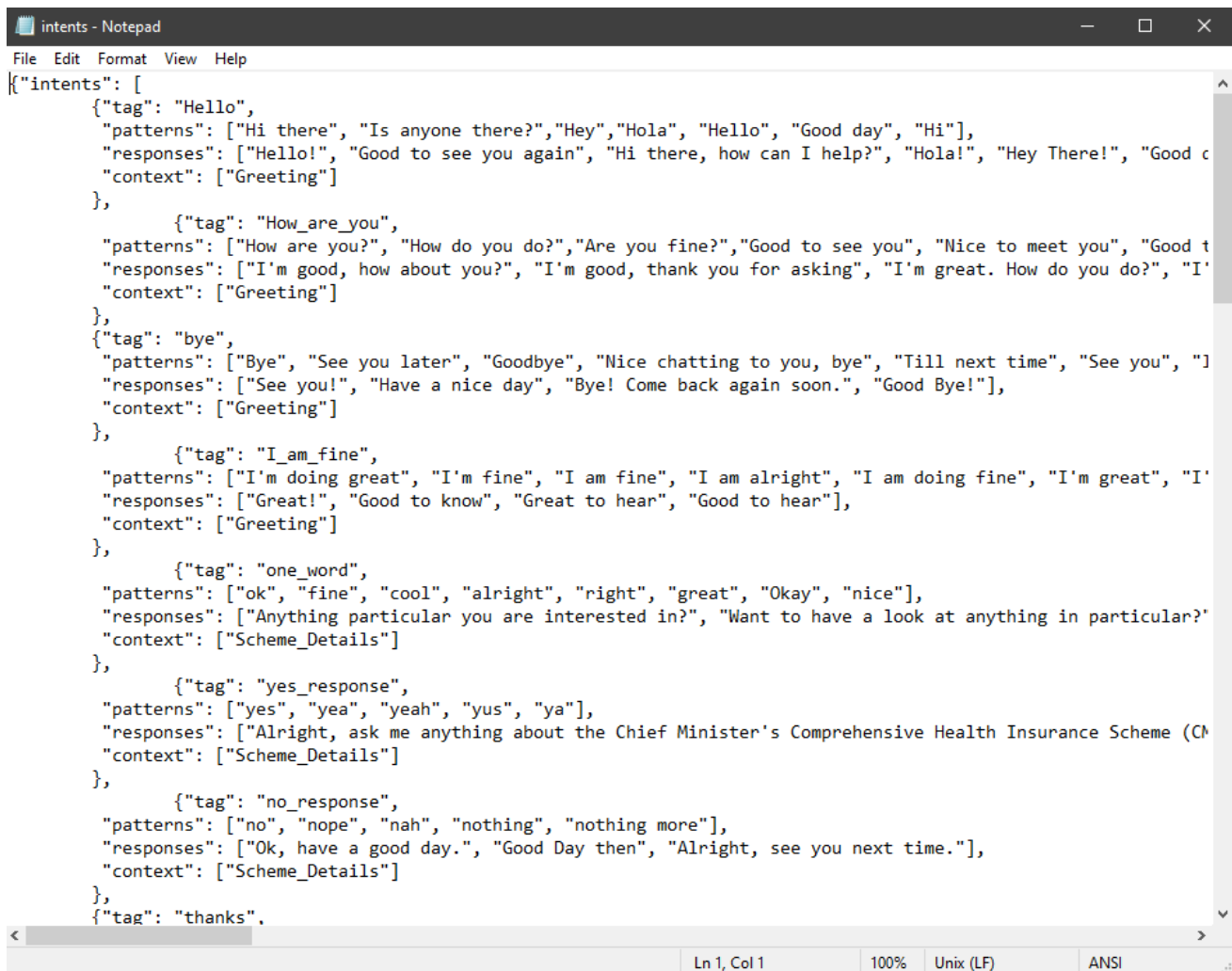
import pickle

import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Activation, Dropout
from keras.optimizers import SGD
import random

words=[]
classes = []
documents = []
ignore_words = ['?', '!']
data_file = open('intents.json').read()
intents = json.loads(data_file)

```

This is how our “intents.json” file looks like:



```

{"intents": [
  {
    "tag": "Hello",
    "patterns": ["Hi there", "Is anyone there?", "Hey", "Hola", "Hello", "Good day", "Hi"],
    "responses": ["Hello!", "Good to see you again", "Hi there, how can I help?", "Hola!", "Hey There!", "Good c",
    "context": ["Greeting"]
  },
  {
    "tag": "How_are_you",
    "patterns": ["How are you?", "How do you do?", "Are you fine?", "Good to see you", "Nice to meet you", "Good t",
    "responses": ["I'm good, how about you?", "I'm good, thank you for asking", "I'm great. How do you do?", "I'
    "context": ["Greeting"]
  },
  {
    "tag": "bye",
    "patterns": ["Bye", "See you later", "Goodbye", "Nice chatting to you, bye", "Till next time", "See you", "I
    "responses": ["See you!", "Have a nice day", "Bye! Come back again soon.", "Good Bye!"],
    "context": ["Greeting"]
  },
  {
    "tag": "I_am_fine",
    "patterns": ["I'm doing great", "I'm fine", "I am fine", "I am alright", "I am doing fine", "I'm great", "I'
    "responses": ["Great!", "Good to know", "Great to hear", "Good to hear"],
    "context": ["Greeting"]
  },
  {
    "tag": "one_word",
    "patterns": ["ok", "fine", "cool", "alright", "right", "great", "Okay", "nice"],
    "responses": ["Anything particular you are interested in?", "Want to have a look at anything in particular?"
    "context": ["Scheme_Details"]
  },
  {
    "tag": "yes_response",
    "patterns": ["yes", "yea", "yeah", "yus", "ya"],
    "responses": ["Alright, ask me anything about the Chief Minister's Comprehensive Health Insurance Scheme (CH
    "context": ["Scheme_Details"]
  },
  {
    "tag": "no_response",
    "patterns": ["no", "nope", "nah", "nothing", "nothing more"],
    "responses": ["Ok, have a good day.", "Good Day then", "Alright, see you next time."],
    "context": ["Scheme_Details"]
  },
  {
    "tag": "thanks",

```

1.3.2 Preprocessing Data

As discussed earlier, tokenizing is the most basic and first thing you can do on text data. To recap, tokenizing is the process of breaking the whole text into small parts like words.

Here we iterate through the patterns and tokenize the sentence using `nltk.word_tokenize()` function and append each word in the words list. We also create a list of classes for our tags.

```
for intent in intents['intents']:
    for pattern in intent['patterns']:

        #tokenize each word
        w = nltk.word_tokenize(pattern)
        words.extend(w)
        #add documents in the corpus
        documents.append((w, intent['tag']))

    # add to our classes list
    if intent['tag'] not in classes:
        classes.append(intent['tag'])
```

Now we will lemmatize each word and remove duplicate words from the list. Lemmatizing is the process of converting a word into its lemma form and then creating a pickle file to store the Python objects which we will use while predicting.

```
# lemmatize, lower each word and remove duplicates
words = [lemmatizer.lemmatize(w.lower()) for w in words if w not in ignore_words]
words = sorted(list(set(words)))
# sort classes
classes = sorted(list(set(classes)))
# documents = combination between patterns and intents
print (len(documents), "documents")
# classes = intents
print (len(classes), "classes", classes)
# words = all words, vocabulary
print (len(words), "unique lemmatized words", words)

pickle.dump(words,open('words.pkl','wb'))
pickle.dump(classes,open('classes.pkl','wb'))
```

1.3.4 Creating Training and Testing Data

Now, we will create the training data in which we will provide the input and the output. Our input will be the pattern and output will be the class our input pattern belongs to. But the computer doesn't understand text so we will convert text into numbers.

```
# create our training data
training = []
# create an empty array for our output
output_empty = [0] * len(classes)
# training set, bag of words for each sentence
for doc in documents:
    # initialize our bag of words
    bag = []
```

```

# list of tokenized words for the pattern
pattern_words = doc[0]
# lemmatize each word - create base word, in attempt to represent related words
pattern_words = [lemmatizer.lemmatize(word.lower()) for word in pattern_words]
# create our bag of words array with 1, if word match found in current pattern
for w in words:
    bag.append(1) if w in pattern_words else bag.append(0)

# output is a '0' for each tag and '1' for current tag (for each pattern)
output_row = list(output_empty)
output_row[classes.index(doc[1])] = 1

training.append([bag, output_row])
# shuffle our features and turn into np.array
random.shuffle(training)
training = np.array(training)
# create train and test lists. X - patterns, Y - intents
train_x = list(training[:,0])
train_y = list(training[:,1])
print("Training data created")

```

1.3.5 Building the model

We have our training data ready, now we will build a deep neural network that has 3 layers. We use the Keras sequential API for this. We will train the model for 200 epochs. Let us save the model as 'chatbot_model.h5'.

```

# Create model - 3 layers. First layer 128 neurons, second layer 64 neurons and 3rd
output layer contains number of neurons
# equal to number of intents to predict output intent with softmax
model = Sequential()
model.add(Dense(128, input_shape=(len(train_x[0]),), activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(len(train_y[0]), activation='softmax'))

# Compile model. Stochastic gradient descent with Nesterov accelerated gradient
gives good results for this model
sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy', optimizer=sgd, metrics=['accuracy'])

#fitting and saving the model
hist = model.fit(np.array(train_x), np.array(train_y), epochs=200, batch_size=5,
verbose=1)
model.save('chatbot_model.h5', hist)

print("model created")

```

```
(cmch-bot) C:\Users\lenovo\Desktop\Chatbot\python train_chatbot.py
2021-11-06 10:38:45.287086: W tensorflow/stream_executor/platform/default/dso_loader.cc:55] Could not load dynamic library 'cudart64_101.dll'; dlderror: cudart64_101.dll not found
2021-11-06 10:38:45.287452: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Ignore above cudart dlderror if you do not have a GPU set up on your machine.
[(['Hi', 'there'], 'Hello'), (['Is', 'anyone', 'there', '?'], 'Hello'), (['Hey', 'Hello'], (['Hola', 'Hello'], (['Hello', 'Hello'], (['Good', 'day'], 'Hello'), (['Hi', 'Hello'], (['How', 'are', 'you', '?'], 'How_are_you'), (['How', 'do', 'you', 'do', '?'], 'How_are_you'), (['Are', 'you', 'fine', '?'], 'How_are_you'), (['Good', 'to', 'see', 'you'], 'How_are_you'), (['Nice', 'to', 'meet', 'you'], 'How_are_you'), (['Good', 'to', 'meet', 'you'], 'How_are_you'), (['How', 'are', 'you'], 'How_are_you'), (['How', 'are', 'you', 'doing', '?'], 'How_are_you'), (['Bye', 'bye'], (['See', 'you', 'later', 'bye'], (['Goodbye', 'bye'], (['Nice', 'chatting', 'to', 'you', 'bye'], 'bye'), (['Till', 'next', 'time'], 'bye'), (['See', 'you', 'bye'], (['later', 'bye'], (['I', 'have', 'to', 'go', 'now'], 'bye'), (['it', 'was', 'nice', 'talking', 'to', 'you', 'bye'], (['I', 'need', 'to', 'leave'], 'bye'), (['I', 'm', 'doing', 'great'], 'I_am_fine'), (['I', 'm', 'fine'], 'I_am_fine'), (['I', 'am', 'fine'], 'I_am_fine'), (['I', 'am', 'alright'], 'I_am_fine'), (['I', 'am', 'doing', 'fine'], 'I_am_fine'), (['I', 'm', 'great'], 'I_am_fine'), (['I', 'm', 'good'], 'I_am_fine'), (['I', 'am', 'good'], 'I_am_fine'), (['good'], 'I_am_fine'), (['Amazing'], 'I_am_fine'), (['Fantastic'], 'I_am_fine'), (['ok'], 'one_word'), (['fine'], 'one_word'), (['cool'], 'one_word'), (['alright'], 'one_word'), (['right'], 'one_word'), (['great'], 'one_word'), (['Okay'], 'one_word'), (['nice'], 'one_word'), (['yes'], 'yes_response'), (['yea'], 'yes_response'), (['yeah'], 'yes_response'), (['yus'], 'yes_response'), (['ya'], 'yes_response'), (['no'], 'no_response'), (['nope'], 'no_response'), (['nah'], 'no_response'), (['nothing'], 'no_response'), (['nothing', 'more'], 'no_response'), (['Thanks'], 'thanks'), (['Thank', 'you'], 'thanks'), (['That', 's', 'helpful'], 'thanks'), (['Awesome', 'thanks'], 'thanks'), (['Thanks', 'for', 'helping', 'me'], 'thanks'), (['good'], 'thanks'), (['That', 's', 'amazing', 'thanks'], 'thanks'), (['fantastic'], 'thanks'), (['wassup', '?'], 'blabber'), (['sup'], 'blabber'), (['tell', 'me', 'bro'], 'blabber'), (['That', 's', 'lit'], 'blabber'), (['12232321'], 'blabber'), (['who', 'are', 'you', '?'], 'identity'), (['what', 'are', 'you', '?'], 'identity'), (['Do', 'I', 'know', 'you', '?'], 'identity'), (['How', 'you', 'could', 'help', 'me', '?'], 'options'), (['What', 'you', 'can', 'do', '?'], 'options'), (['What', 'help', 'you', 'provide', '?'], 'options'), (['How', 'you', 'can', 'be', 'helpful', '?'], 'options'), (['What', 'support', 'is', 'offered'], 'options'), (['What', 'can', 'I', 'ask', 'you', '?'], 'options'), (['What', 'can', 'you', 'do', 'for', 'me', '?'], 'options'), (['What', 'do', 'you', 'do', '?'], 'options'), (['When', 'was', 'this', 'scheme', 'launched', '?'], 'year_of_launch'), (['When', 'was', 'CMCHIS', 'started', '?'], 'year_of_launch'), (['When', 'did', 'it', 'launch', '?'], 'year_of_launch'), (['When', 'did', 'it', 'start', 'year_of_launch')]]]

2021-11-06 10:39:07.277476: I tensorflow/compiler/xla/service/service.cc:176] StreamExecutor device (0): Host, Default Version
2021-11-06 10:39:07.279106: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1102] Device interconnect StreamExecutor with strength 1 edge matrix:
2021-11-06 10:39:07.279228: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1108]
Epoch 1/200
38/38 [=====] - 0s 818us/step - loss: 3.3086 - accuracy: 0.0529
Epoch 2/200
38/38 [=====] - 0s 822us/step - loss: 3.1874 - accuracy: 0.0741
Epoch 3/200
38/38 [=====] - 0s 822us/step - loss: 3.0648 - accuracy: 0.1429
Epoch 4/200
38/38 [=====] - 0s 822us/step - loss: 2.9129 - accuracy: 0.1905
Epoch 5/200
38/38 [=====] - 0s 822us/step - loss: 2.7335 - accuracy: 0.2381
Epoch 6/200
38/38 [=====] - 0s 1ms/step - loss: 2.4792 - accuracy: 0.3016
Epoch 7/200
38/38 [=====] - 0s 822us/step - loss: 2.2973 - accuracy: 0.3704
Epoch 8/200
38/38 [=====] - 0s 822us/step - loss: 2.0245 - accuracy: 0.4286
Epoch 9/200
38/38 [=====] - 0s 822us/step - loss: 1.8507 - accuracy: 0.4762
Epoch 10/200
38/38 [=====] - 0s 822us/step - loss: 1.6225 - accuracy: 0.5503
Epoch 11/200
38/38 [=====] - 0s 1ms/step - loss: 1.4095 - accuracy: 0.6138
Epoch 12/200
38/38 [=====] - 0s 822us/step - loss: 1.5232 - accuracy: 0.5238

Epoch 188/200
38/38 [=====] - 0s 822us/step - loss: 0.1399 - accuracy: 0.9524
Epoch 189/200
38/38 [=====] - 0s 822us/step - loss: 0.1014 - accuracy: 0.9683
Epoch 190/200
38/38 [=====] - 0s 822us/step - loss: 0.1486 - accuracy: 0.9471
Epoch 191/200
38/38 [=====] - 0s 822us/step - loss: 0.1103 - accuracy: 0.9630
Epoch 192/200
38/38 [=====] - 0s 822us/step - loss: 0.1943 - accuracy: 0.9259
Epoch 193/200
38/38 [=====] - 0s 822us/step - loss: 0.0826 - accuracy: 0.9577
Epoch 194/200
38/38 [=====] - 0s 822us/step - loss: 0.0754 - accuracy: 0.9735
Epoch 195/200
38/38 [=====] - 0s 822us/step - loss: 0.0620 - accuracy: 0.9735
Epoch 196/200
38/38 [=====] - 0s 822us/step - loss: 0.0875 - accuracy: 0.9630
Epoch 197/200
38/38 [=====] - 0s 822us/step - loss: 0.1032 - accuracy: 0.9577
Epoch 198/200
38/38 [=====] - 0s 822us/step - loss: 0.0571 - accuracy: 0.9788
Epoch 199/200
38/38 [=====] - 0s 822us/step - loss: 0.0851 - accuracy: 0.9577
Epoch 200/200
38/38 [=====] - 0s 822us/step - loss: 0.1440 - accuracy: 0.9471
model created
```

As observed from the screenshots in training, we have reached a model accuracy of 94.71%.

3.2.3.5 Predicting the Response (Graphical User Interface)

To predict the sentences and get a response from the user to let us create a new file 'chatgui.py'.

We will load the trained model and then use a graphical user interface that will predict the response from the bot. The model will only tell us the class it belongs to, so we will implement some functions which will identify the class and then retrieve a random response from the list of responses inputted.

Again, we import the necessary packages and load the 'words.pkl' and 'classes.pkl' pickle files which we have created when we trained our model:

```
import nltk
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
import pickle
import numpy as np

from keras.models import load_model
model = load_model('chatbot_model.h5')
import json
import random
intents = json.loads(open('intents.json').read())
words = pickle.load(open('words.pkl', 'rb'))
classes = pickle.load(open('classes.pkl', 'rb'))
```

To predict the class, we will need to provide input in the same way as we did while training. So, we will create some functions that will perform text preprocessing and then predict the class.

```
def clean_up_sentence(sentence):
    # tokenize the pattern - split words into array
    sentence_words = nltk.word_tokenize(sentence)
    # stem each word - create short form for word
    sentence_words = [lemmatizer.lemmatize(word.lower()) for word in sentence_words]
    return sentence_words

# return bag of words array: 0 or 1 for each word in the bag that exists in the sentence

def bow(sentence, words, show_details=True):
    # tokenize the pattern
    sentence_words = clean_up_sentence(sentence)
    # bag of words - matrix of N words, vocabulary matrix
    bag = [0]*len(words)
    for s in sentence_words:
        for i,w in enumerate(words):
            if w == s:
                # assign 1 if current word is in the vocabulary position
                bag[i] = 1
                if show_details:
                    print ("found in bag: %s" % w)
    return(np.array(bag))

def predict_class(sentence, model):
    # filter out predictions below a threshold
    p = bow(sentence, words, show_details=False)
```

```

res = model.predict(np.array([p]))[0]
ERROR_THRESHOLD = 0.25
results = [[i,r] for i,r in enumerate(res) if r>ERROR_THRESHOLD]
# sort by strength of probability
results.sort(key=lambda x: x[1], reverse=True)
return_list = []
for r in results:
    return_list.append({"intent": classes[r[0]], "probability":
str(r[1])})
return return_list

```

After predicting the class, we will get a random response from the list of intents.

```

def getResponse(ints, intents_json):
    tag = ints[0]['intent']
    list_of_intents = intents_json['intents']
    for i in list_of_intents:
        if(i['tag']== tag):
            result = random.choice(i['responses'])
            break
    return result

def chatbot_response(text):
    ints = predict_class(text, model)
    res = getResponse(ints, intents)
    return res

```

Now we will develop a graphical user interface. We use the Tkinter library which is shipped with tons of useful libraries for GUI. We will take the input message from the user and then use the helper functions we have created to get the response from the bot and display it on the GUI. The source code for the GUI is shown below.

```

#Creating GUI with tkinter
import tkinter
from tkinter import *

def send():
    msg = EntryBox.get("1.0",'end-1c').strip()
    EntryBox.delete("0.0",END)

    if msg != '':
        ChatLog.config(state=NORMAL)
        ChatLog.insert(END, "You: " + msg + '\n\n')
        ChatLog.config(foreground="#442265", font=("Verdana", 12 ))

        res = chatbot_response(msg)
        ChatLog.insert(END, "Bot: " + res + '\n\n')

        ChatLog.config(state=DISABLED)
        ChatLog.yview(END)

```

```

base = Tk()
base.title("Hello")
base.geometry("400x500")
base.resizable(width=FALSE, height=FALSE)

#Create Chat window
ChatLog = Text(base, bd=0, bg="white", height="8", width="50", font="Arial",)

ChatLog.config(state=DISABLED)

#Bind scrollbar to Chat window
scrollbar = Scrollbar(base, command=ChatLog.yview, cursor="heart")
ChatLog['yscrollcommand'] = scrollbar.set

#Create Button to send message
SendButton = Button(base, font=("Verdana",12,'bold'), text="Send", width="12",
height=5,
                        bd=0, bg="#32de97", activebackground="#3c9d9b",fg='ffffff',
                        command= send )

#Create the box to enter message
EntryBox = Text(base, bd=0, bg="white",width="29", height="5", font="Arial")
#EntryBox.bind("<Return>", send)

#Place all components on the screen
scrollbar.place(x=376,y=6, height=386)
ChatLog.place(x=6,y=6, height=386, width=370)
EntryBox.place(x=128, y=401, height=90, width=265)
SendButton.place(x=6, y=401, height=90)

base.mainloop()

```

3.2.3.6 Running the Chatbot

To run the chatbot, we have two main files: **train_chatbot.py** and **chatgui.py**

First, we train the model by running the following code in the terminal:

```
python train_chatbot.py
```

If we don't see any error during training, we have successfully created the model. Then to run the app, we run the second file.

```
python chatgui.py
```

The following screenshots show the chatbot in action:

