Chatbot Lesson

**Intro:**

Increased computing power and access to massive data sets has made applying deep learning techniques (a class of machine learning algorithms) feasible in practice. Deep learning is an extremely powerful tool that can be leveraged to do incredible things but it’s not without vulnerabilities.

This two-part lesson will first walk you through the process of building an NLP chatbot designed to give a diagnosis based on symptoms reported by the user, and the second part will focus on demonstrating ways the system you designed could be vulnerable/exploitable.

Let’s get started!

First, open a blank Jupyter Notebook and install the necessary packages. In the example below, %pip install is being used, but any equivalent shell command will work.

%pip install nltk

%pip install numpy as np

%pip install pickle

%pip install tensorflow

%pip install keras

* The Natural Language Tool Kit (NLTK) comes with a host of different Python modules for developing Natural Language Processing applications.
	+ Documentation: https://www.nltk.org/
* NumPy is a library that aids in performing computations on large, multi-dimensional mathematical objects like arrays and matrices.
	+ Documentation: https://numpy.org/doc/
* Pickle is a module for serializing and de-serializing python object structures.
	+ Documentation: https://docs.python.org/3/library/pickle.html
* TensorFlow is the most popular library for building deep learning models and Keras is a high-level API that is built on top of TensorFlow. Keras allows developers to create neural networks without the need for a deep mathematical understanding of tensor algebra and optimization.
	+ Documentation: https://www.tensorflow.org/api\_docs/python/tf/keras

After we’ve downloaded the requisite packages, we can begin importing what we’ll need to build our chatbot:

import json

import pickle

import numpy as np

import nltk

from nltk.stem import WordNetLemmatizer

from keras.models import Sequential

from keras.optimizers import SGD

from keras.layers import Dense, Activation, Dropout

import random

Python has a few built-in packages we’ll be importing here (you may notice we didn’t need to install the packages json or random). Json is a built-in package used to work with JSON data and random will be used to shuffle the training data before it’s converted into a numpy array.

--

Before we get too far into the code, let’s look at a few concepts that are critical to Natural Language Processing:

First, what is Natural Language Processing? Natural Language Processing (or just ‘NLP’) is a subfield in linguistics, computer science, and artificial intelligence concerned with how computers can be programmed to process and analyze language data.

A word you will hear a lot when working with NLP is *tokenize*. Tokenizing (in linguistics) refers to the process of breaking text up into individual linguistic units. So, tokenizing, within the context of NLP, could mean breaking down a paragraph into individual sentences or breaking down sentences into individual words.

*Lemmatization,* in linguistics, is the process of grouping together the inflected forms of a word so they can be analyzed as a single item. For example, the word ‘drive’ has many inflected forms including ‘driving’, ‘drove’, ‘drives’, etc. The base form ‘drive’ is called the lemma. We will import the WordNetLemmatizer from the nltk.stem module for later lemmatizations.

----

Now that we’ve gotten all the necessary packages imported, we can get to work preparing the data for the model. Data preprocessing is an essential component of any successful ML model and our chatbot powered by deep learning is no exception.

words = []

classes = []

documents = []

ignore\_words = ['?', '!',',','.']

data\_file = open('intents.json').read()

intents = json.loads(data\_file)

Here’s a brief description of the lists we are creating:

classes = type of response (e.g. arthritis\_diagnosis)

words = individual unique lemmatized words used to train model

documents = objects containing tokenized versions of each pattern and their corresponding tag (e.g. (['How', 'are', 'you'], 'greeting'))

After we’ve created empty lists to hold our different objects (classes, words, and clusters), we will create a list that simply contains strings or characters we don’t want to be included in our list of words used to train the model (ignore\_words), and we load the json intents file into the *intents* variable.

The intents file, located here <provide link here>, provides the logical and lexical backbone for our chatbot. It contains JavaScript objects containing different word patterns that will be used to train our bot.

After we’ve prepared our variables, we use ‘for loops’ to iteratively tokenize the sentence patterns into their individual words and then add them to a variable *w*, which we then add to the **words** list. The clusters list is prepared by appending each tokenized pattern to its corresponding tag (e.g. [(['Hi', 'there'], 'greeting'), (['How', 'are', 'you'], 'greeting'), ([ ….) as specified above, and then each intent is added to the **classes** list.

for intent in intents['intents']:

 for pattern in intent['patterns']:

 # tokenizing sentence patterns into individual words

 w = nltk.word\_tokenize(pattern)

 # adding w to words list

 words.extend(w)

 # populationg clusters list

 clusters.append((w, intent['tag']))

 # populating classes list

 if intent['tag'] not in classes:

 classes.append(intent['tag'])

We then take the **words** list and lemmatize each word and convert each character to lowercase to prepare the data to be used for training our model. Before pickling our **words** and **classes** lists, we are going to include a few lines of code that will output the contents of the lists that we’ve generated so far to get a look at the data we will be using to train the model.

lemmatizer = WordNetLemmatizer()

words = [lemmatizer.lemmatize(w.lower())

 for w in words if w not in ignore\_words]

print(len(clusters), "clusters")

print(len(classes), "classes", classes)

print(len(words), "unique lemmatized words", words)

Next, we will write pickled representations of the **words** and **classes** lists to their corresponding words.pkl and classes.pkl files. These files will be used later when we build the GUI.

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

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

We will be using something called the “bag-of-words” method to work with the raw text that will be used to train the model. The bag-of-words (BoW) method is a relatively simple data preprocessing technique that converts text into vectors of numbers (a data format that the computer can interpret). Before we build the for-loop that will format the training data for the neural network, we are going to create an empty list that will eventually hold our formatted training data and another list variable (**output\_empty**) that will eventually be used to create the **output\_row** variable that will serve as the binary representation of the class corresponding to the **pattern\_words** variable that contains all the words in the aforementioned class.

training = []

output\_empty = [0] \* len(classes)

Now we are ready to dig in to the for-loop that will prepare our data for the neural network. The loop looks like this:

for doc in documents:

 pattern\_words = doc[0]

 pattern\_words = [lemmatizer.lemmatize(word.lower()) for word in pattern\_words]

 bag = []

 for w in words:

 bag.append(1) if w in pattern\_words else bag.append(0)

 output\_row = list(output\_empty)

 output\_row[classes.index(doc[1])] = 1

 training.append([bag, output\_row])

Don’t worry if that isn’t all that meaningful to you. We are going to break down the code line by line, so you understand exactly what’s going on.

First, **documents** is a nested list composed of a list of tuple objects (**doc**) that contain within them a list of the words in any given pattern and the class that’s associated with that pattern. Here is a closer look at the data stored in **documents**. Looking now at one of the **doc** tuples: ([‘Hi’, ‘there’], ‘greeting’) – you can see that ‘greeting’ is the class (found in index position [1]), and there is a nested list of words found in the ‘greeting’ pattern located at index position [0].

We will call a for-loop that will iteratively append lemmatized and lowercased versions of the words contained in the pattern part of the **doc** variable to new **pattern\_words** variable.

and then create another empty list for our bag of words.

for doc in documents:

 pattern\_words = doc[0]

 pattern\_words = [lemmatizer.lemmatize(word.lower()) for word in pattern\_words]

 bag = []

Then, within our current for-loop, we will create another loop that iteratively appends a 1 to the bag of words list when the word found in the pattern corresponds with the word in our **words** variable. This will create a new list of ones and zeros that’s exactly the same length as our sequential list of words in our **words** variable. If ‘Hi’ is the 35th word in **words** and ‘there’ is the 60th word in the list, there will be zeros for every value in the list except for a 1 in the 35th and 60th position of the new list. Then we will take our **output\_empty** variable and create a binary representation of the class corresponding to the word pattern we just encoded into a different binary list. So, say we are working with a list of 10 classes and ‘greeting’ is the class in the 5th position of the list, our **output\_row** variable will be created by adding a 1 to the 5th spot of the **output\_empty** variable, thus creating a list that looks like this: [0, 0, 0, 0, 1, 0, 0, 0, 0, 0].

Finally, we will add a new object to our previously empty training list that is composed of our bag of words (remember this is just a binary list with all zeros except for ones in place of the words present in the pattern of interest) and our **output\_row** list (the binary representation of the class of interest)

 for w in words:

 bag.append(1) if w in pattern\_words else bag.append(0)

 output\_row = list(output\_empty)

 output\_row[classes.index(doc[1])] = 1

 training.append([bag, output\_row])

--

We then shuffle our data, convert it to a numpy array, and then split up the patterns and intents to create the dependent and independent variable. The patterns serve as the independent variable (X) and the intents are the dependent variable (Y).

random.shuffle(training)

training = np.array(training)

train\_x = list(training[:, 0])

train\_y = list(training[:, 1])

--

Now that the data has been prepared, we can begin constructing the model. We are only dealing with one input tensor and one output tensor so a sequential model will serve us well. Keras makes constructing neural networks simple. First, we declare the model as sequential, and then use the add() method to incrementally add layers.

Our model will use 4 dense layers, with the first layer containing 256 neurons, followed by 128, 64, and then finally the output later. The shape of the input layer is specified as the shape of the first vector in the train\_x array containing our binary-encoded patterns, and the output layer is designated as the shape of first vector in the train\_y array (our intents).

model = Sequential()
model.add(Dense(256, input\_shape=(len(train\_x[0]),), activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.1))
model.add(Dense(len(train\_y[0]), activation='softmax'))

The softmax activation function is an extension of the sigmoid function designed to perform multi-class classification so we will use that in our output layer.

Finally, we configure and then train the model:

sgd = SGD(learning\_rate=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical\_crossentropy', optimizer=sgd, metrics=['accuracy'])
hist = model.fit(np.array(train\_x), np.array(train\_y), epochs=1000, batch\_size=5, verbose=1)
model.save('chatbot\_model.h5', hist)

*At this point, the first part of the program is done. This might be a good time to ask learners how they suspect the system being built could be nefariously manipulated.*

With our model saved, we can begin building the GUI.

Here’s a look at the packages we will need to create the GUI

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
from tkinter import \*

After importing the packages, we will load our intents file and un-pickle our words and classes:

intents = json.loads(open('intents.json').read())
words = pickle.load(open('words.pkl','rb'))
classes = pickle.load(open('classes.pkl','rb'))

Next, we are going to build the functions responsible for generating the chatbot responses:

The first function will take the text entered by the user and lemmatize and lowercase all words. The idea here is to prepare the user text in the same way that we prepared the text used for training the model.

def clean\_up\_sentence(sentence):
 sentence\_words = nltk.word\_tokenize(sentence)
 sentence\_words = [lemmatizer.lemmatize(word.lower()) for word in sentence\_words]
 return sentence\_words

Next, we will design a function to create a bag of words from text prepared by our clean\_up\_sentence function in an analogous manner to how we created bags of words for training the model.

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))

After our user input is prepared, we will create another function that generates a list of potential user intents and corresponding probabilities. With the list of probabilities for each intent calculated, our chatbot will select the most likely intent and then another function will randomly select a response that corresponds with the user’s predicted intent, and, finally, another simple function will return the response selected by the getResponse function.

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

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(msg):
 ints = predict\_class(msg, model)
 res = getResponse(ints, intents)
 return res

The only thing left to do now is create the GUI for interacting with our chatbot. For this example, we will be using tkinter. Tkinter is a built-in GUI framework that is widely considered to be the de facto choice for building interfaces in Python because of its relative ease of use.

#Creating GUI

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, "Bob: " + res + '\n\n')

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

base = Tk()
base.title("Diagnostic Bob")
base.geometry("400x500")
base.resizable(width=TRUE, height=TRUE)

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

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=("Cambria",10,'italic'), text="Ask Bob", width="12", height=5,
 bd=5, bg="DarkOliveGreen4", activebackground="black",fg='DarkOliveGreen4',
 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()

Here’s what your GUI should look like if you followed along with the code suggested in this lesson:



Cool, huh?

\*Now might be a suitable time to brainstorm potential vulnerabilities and then have the student experiment with manipulating the json file to create a chatbot that delivers bad / incorrect / offensive information\*