my=open('/content/FinaldataV1.json','r')

jsondata=my.read()

intents=json.loads(jsondata)

print(intents)

import json

import numpy as np

import nltk

# nltk.download('punkt')

from nltk.stem.porter import PorterStemmer

import torch

import torch.nn as nn

from torch.utils.data import Dataset,DataLoader

my=open('/content/FinaldataV1.json','r')

jsondata=my.read()

intents=json.loads(jsondata)

stemmer = PorterStemmer()

def tokenize(sentence):

"""

split sentence into array of words/tokens

a token can be a word or punctuation character, or number

"""

return nltk.word\_tokenize(sentence)

def stem(word):

"""

stemming = find the root form of the word

examples:

words = ["organize", "organizes", "organizing"]

words = [stem(w) for w in words]

-> ["organ", "organ", "organ"]

"""

return stemmer.stem(word.lower())

def bag\_of\_words(tokenized\_sentence, words):

"""

return bag of words array:

1 for each known word that exists in the sentence, 0 otherwise

example:

sentence = ["hello", "how", "are", "you"]

words = ["hi", "hello", "I", "you", "bye", "thank", "cool"]

bog = [ 0 , 1 , 0 , 1 , 0 , 0 , 0]

"""

# stem each word

sentence\_words = [stem(word) for word in tokenized\_sentence]

# initialize bag with 0 for each word

bag = np.zeros(len(words), dtype=np.float32)

for idx, w in enumerate(words):

if w in sentence\_words:

bag[idx] = 1

return bag

all\_words = []

tags = []

xy = []

for intent in intents['intents']:

tag = intent['tag']

# add to tag list

tags.append(tag)

for pattern in intent['patterns']:

# tokenize each word in the sentence

w = tokenize(pattern)

# add to our words list

all\_words.extend(w)

# add to xy pair

xy.append((w, tag))

# stem and lower each word

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

all\_words = [stem(w) for w in all\_words if w not in ignore\_words]

# remove duplicates and sort

all\_words = sorted(set(all\_words))

tags = sorted(set(tags))

print(len(xy), "patterns")

print(len(tags), "tags:", tags)

print(len(all\_words), "unique stemmed words:", all\_words)

# create training data

X\_train = []

y\_train = []

for (pattern\_sentence, tag) in xy:

# X: bag of words for each pattern\_sentence

bag = bag\_of\_words(pattern\_sentence, all\_words)

X\_train.append(bag)

# y: PyTorch CrossEntropyLoss needs only class labels, not one-hot

label = tags.index(tag)

y\_train.append(label)

X\_train = np.array(X\_train)

y\_train = np.array(y\_train)

# Hyper-parameters

num\_epochs = 1000

batch\_size = 8

learning\_rate = 0.001

input\_size = len(X\_train[0])

hidden\_size = 8

output\_size = len(tags)

print(input\_size, output\_size)

class ChatDataset(Dataset):

def \_\_init\_\_(self):

self.n\_samples = len(X\_train)

self.x\_data = X\_train

self.y\_data = y\_train

# support indexing such that dataset[i] can be used to get i-th sample

def \_\_getitem\_\_(self, index):

return self.x\_data[index], self.y\_data[index]

# we can call len(dataset) to return the size

def \_\_len\_\_(self):

return self.n\_samples

dataset = ChatDataset()

train\_loader = DataLoader(dataset=dataset,

batch\_size=batch\_size,

shuffle=True,

num\_workers=0)

device = torch.device('cuda' if torch.cuda.is\_available() else 'cpu')

model = NeuralNet(input\_size, hidden\_size, output\_size).to(device)

# Loss and optimizer

criterion = nn.CrossEntropyLoss()

optimizer = torch.optim.Adam(model.parameters(), lr=learning\_rate)

# Train the model

for epoch in range(num\_epochs):

for (words, labels) in train\_loader:

words = words.to(device)

labels = labels.to(dtype=torch.long).to(device)

# Forward pass

outputs = model(words)

# if y would be one-hot, we must apply

# labels = torch.max(labels, 1)[1]

loss = criterion(outputs, labels)

# Backward and optimize

optimizer.zero\_grad()

loss.backward()

optimizer.step()

if (epoch+1) % 100 == 0:

print (f'Epoch [{epoch+1}/{num\_epochs}], Loss: {loss.item():.4f}')

print(f'final loss: {loss.item():.4f}')

data = {

"model\_state": model.state\_dict(),

"input\_size": input\_size,

"hidden\_size": hidden\_size,

"output\_size": output\_size,

"all\_words": all\_words,

"tags": tags

}

FILE = "s.pth"

torch.save(data, FILE)

print(f'training complete. file saved to {FILE}')

import torch

import torch.nn as nn

class NeuralNet(nn.Module):

def \_\_init\_\_(self, input\_size, hidden\_size, num\_classes):

super(NeuralNet, self).\_\_init\_\_()

self.l1 = nn.Linear(input\_size, hidden\_size)

self.l2 = nn.Linear(hidden\_size, hidden\_size)

self.l3 = nn.Linear(hidden\_size, num\_classes)

self.relu = nn.ReLU()

def forward(self, x):

out = self.l1(x)

out = self.relu(out)

out = self.l2(out)

out = self.relu(out)

out = self.l3(out)

# no activation and no softmax at the end

return out

import random

import json

FILE = "s.pth"

data = torch.load(FILE)

input\_size = data["input\_size"]

hidden\_size = data["hidden\_size"]

output\_size = data["output\_size"]

all\_words = data['all\_words']

tags = data['tags']

model\_state = data["model\_state"]

model = NeuralNet(input\_size, hidden\_size, output\_size).to(device)

model.load\_state\_dict(model\_state)

model.eval()

bot\_name = "ChatBOT: "

print("Hello! Let's chat! (type 'quit' to exit)")

while True:

# sentence = "do you use credit cards?"

sentence = input("You: ")

if sentence == "Thankyou":

break

sentence = tokenize(sentence)

X = bag\_of\_words(sentence, all\_words)

X = X.reshape(1, X.shape[0])

X = torch.from\_numpy(X).to(device)

output = model(X)

\_, predicted = torch.max(output, dim=1)

tag = tags[predicted.item()]

probs = torch.softmax(output, dim=1)

prob = probs[0][predicted.item()]

if prob.item() > 0.75:

for intent in intents['intents']:

if tag == intent["tag"]:

reply=random.choice(intent['responses'])

print(f"{bot\_name}",reply)

else:

print(f"{bot\_name}: I do not understand...")