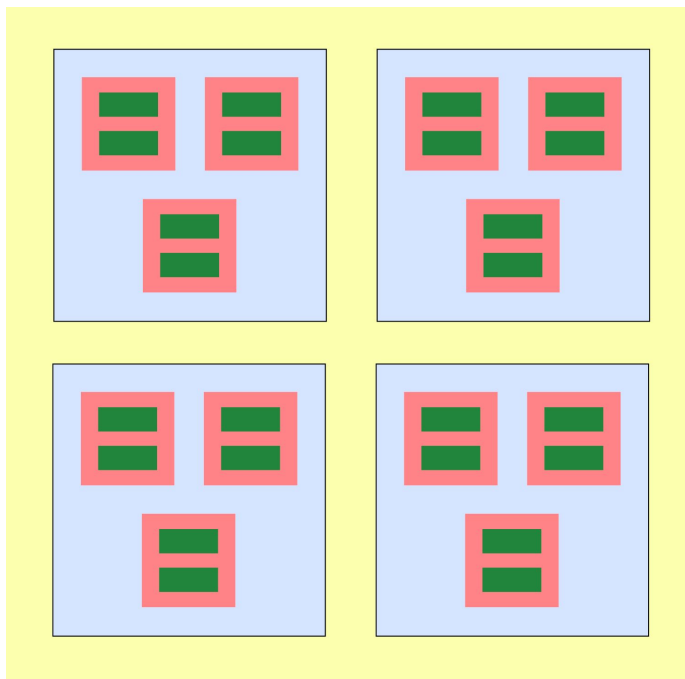


PyTorBot

Have you ever felt isolated amid COVID-19? Do you want to create a virtual friend to share your thoughts? Don't worry! We're here to guide you to create a PyTorch chatbot. Prerequisites for this tutorial are Python, Jupyter Notebook, NLTK, and Neural network. Closely follow our steps, and you'll be able to build your PyTorBot friend.

What is Pytorch? PyTorch is a library that can manipulate tensor and provide tools to work with deep learning. Tensor is an object of some multidimensional data (18). For example, if you have a tensor (4, 3, 2), your tensor will have $4 * 3 * 2 = 24$ numerical values in total.

To understand this easier, let's imagine your whole tensor (4, 3, 2) is a big yellow square. The yellow square contains 4 gray smaller squares which have 3 other light-pink (salmon) squares inside. Finally, each salmon square includes 2 other green squares.



Now, let's create the tensor in PyTorch. If you haven't installed Pytorch, check out <https://pytorch.org/> and find the following table.

PyTorch Build	Stable (1.6.0)		Preview (Nightly)	
Your OS	Linux	Mac	Windows	
Package	Conda	Pip	LibTorch	Source
Language	Python		C++ / Java	
CUDA	9.2	10.1	10.2	None
Run this Command:	conda install pytorch torchvision cpuonly -c pytorch			

Check all the boxes that are suitable for your computer.

In our case, we choose

“Stable (1.6.0)”,

“Windows”,

“Conda”,

“Python”,

and “None” (CUDA).

Then we run the command generated on the last row.

Run this Command:	conda install pytorch torchvision cpuonly -c pytorch
-------------------	--

Let's open your local Jupyter Notebook, “import torch”, and create a torch that has 24 numbers from 0 to 23 by “arange(24)”. Store your tensor in x. If you want to display x as (4, 3, 2) then use “view(4, 3, 2)” as below.

```
import torch

x = torch.arange(24)
x = x.view(4,3,2)
```

Tada, you successfully create your first tensor.

```

tensor([[[ 0,  1],
          [ 2,  3],
          [ 4,  5]],

        [[ 6,  7],
          [ 8,  9],
          [10, 11]],

        [[12, 13],
          [14, 15],
          [16, 17]],

        [[18, 19],
          [20, 21],
          [22, 23]]])

```


Now you know some basic concepts of PyTorch. Let's work on our PyTorBot. Don't forget to "import torch" before getting started.

Firstly, we read a JSON. For our demo, we call it "myjson" and insert directly into the code cell. "myjson" includes a "list" of "tag" (containing the overall meaning of a sentence), possible "input" of users, and PyTorBot's "responses". For example,

```

myjson =
{
  "list": [
    {
      "tag": "greeting",
      "input": ["Hi", "Hey", "High five", "Hello"],
      "responses": ["Hello", "*fist bump*", "Hi there"],
    },
    {
      "tag": "goodbye",
      "input": ["Bye", "Goodbye", "See you again"],
      "responses": ["See you", "Bye Bye", "Bye! See you soon"]
    }
  ]
}

```

Whenever you want to run your code cell, simply click  or concurrently press "shift" + "enter".

"myjson" might have long sentences. Let's break ("tokenize") them into smaller parts to easily handle. Before doing so, we install "nltk" (Natural Language Toolkit) then "import TweetTokenizer".

```
!pip install nltk
```

```

from nltk.tokenize import TweetTokenizer
tokenizer = TweetTokenizer()

```

And you might notice that, each word can be a variation of another word. Thus, we use "stemming" to return all words to their original state.

```
!pip install stemming
from stemming.porter2 import stem
```

Great! Handy tools are installed. Let's tokenize all possible user inputs from "myjson". Before doing so, make sure every word is lowercase by "inputs.lower()".

Every time we run "tokenizer.tokenize(inputs.lower())", we'll receive a list "w" of tokenized lowercase words.

We store all words in a list named "allword" by "extending" "allword" with "w": allword.extend(w). Finally, store "w" with its corresponding "tag" in the "xy" list. And gather all "tag" in a "tags" list for future training process.

```
allword = [] # list of all tokenized words
tags = [] # list of tags
xy = [] # list of tuples containing (tokenized words, tag)

for element in myjson['list']:
    tags.append(element['tag']) # greeting, goodbye, ...
    for inputs in element['input']:
        w = tokenizer.tokenize(inputs.lower()) # break sentence smaller
        allword.extend(w) # gather all words
        xy.append((w, stem(element['tag']))) # words + tags

tags = [stem(word) for word in tags]
```

Since our PyTorBot just need to understand the meaning of words, we exclude punctuation symbols like '?', '!', etc from "allword".

```
# exclude the following
exclude = ['?', '!', ',', '.', ':', ';']
for x in allword:
    if x in exclude:
        allword.remove(x)
```

Without these symbols, we can now "stem" all words.

```
allword = [stem(word) for word in allword]
```

When users input a word, if that word is in your PyTorBot's "allword", mark it "1.0". Otherwise, mark it "0.0". We can do that with a function called "one_or_none", which receives 2 arguments "tokenize_sentence" and "allword". Inside the function, we firstly mark all words in "allword" as "0.0". Then, if user input words is in "allword", we mark it as "1.0".

```
import numpy as np
def one_or_none(tokenized_sentence, allword):
    mark = np.zeros(len(allword), dtype = np.float32) # mark everything 0.0
    for indx, w in enumerate(allword):
        if w in tokenized_sentence: # if input word is in allword
            mark[indx] = 1.0 # mark 1.0
    return mark
```

Good job! We finish the first step of making PyTorBot. Now we create training sets (“x_train” and “y_train”). Let us remind you of the “xy” list - a list of tuples containing “tokenized sentences” and “their tags”.

For each tokenized sentence, compare it with “allword” list by using the function “one_or_none” to mark 1.0 or 0.0. “x_train” list will have all the corresponding 1 and 0. Meanwhile, “y_train” list finds the index of “xy”’s tag in “tags” list. After all, turning x_train and y_train to numpy arrays to make them run faster.

```
x_train = []
y_train = []
for tokenize_sentence, tag in xy:
    tokenize_sentence = [stem(word) for word in tokenize_sentence]
    mark = one_or_none(tokenize_sentence, allword) # mark 0 or 1
    x_train.append(mark) # list of all the 0s and 1s
    find = tags.index(tag) # find tag of "xy" in "tags" list
    y_train.append(find)

x_train = np.array(x_train) # run faster with np.array
y_train = np.array(y_train)
```

You might try to manually create training datasets with random shuffle. However, PyTorch will help you do that. We import Dataset, DataLoader from torch.utils.data, then customize a PyTorch Dataset class to fit into your project.

```
from torch.utils.data import Dataset, DataLoader
```

In our case, let’s call our DataSet class as “PyTorBotDataset”.

In such a class, we define 3 functions __init__, __getitem__, and __len__.

Generally speaking,

__init__ function is used to read data;

__getitem__ is used to return an item with its index;

__len__ is used to return data length.

```

class PyTorBotDataset(Dataset):
    def __init__(self): # read data

    def __getitem__(self, index): # return item with its index

    def __len__(self): # return length of data

```

Now, based on the purposes of each function, we write its body as below.

```

class PyTorBotDataset(Dataset):
    def __init__(self):
        self.len = len(x_train)
        self.x_data = x_train
        self.y_data = y_train

    def __getitem__(self, index):
        return self.x_data[index], self.y_data[index]

    def __len__(self):
        return self.len

```

It's time to put your PyTorBotDataset() into a "train_loader"

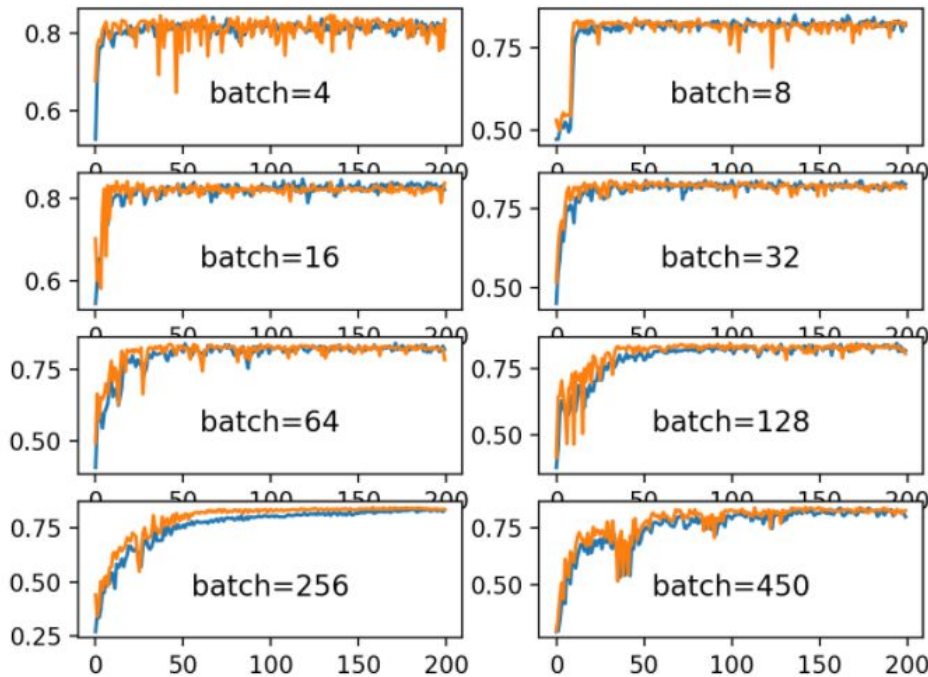
```

dataset = PyTorBotDataset()
train_loader = DataLoader(dataset = dataset, batch_size = 32, shuffle = True, num_workers = 0)

```

You might change features like "batch_size" and "num_workers", but in our case, we choose batch_size = 100 (a pretty good default) and num_workers = 0 (to avoid errors). Hmm... As a beginner, we might wonder what "batch_size" and "num_workers" are. So, here's your information.

- batch_size is the number of training examples (samples) in each pass (forward or backward). Bigger batch_size uses more memory. A too large batch_size can reduce the accuracy, while a small batch_size can make the program run very slow. Thus, a suitable batch_size will ease your training process.
- Suppose batch_size = 100 and you have 1000 examples. It will take $1000/100 = 10$ iterations to complete 1 epoch, with epoch being a complete pass (forward and backward) of all samples.
- Check out the following picture comparing how test sets fit into train sets. (Test set is orange, and the train set is blue).



source: [ML Mastery](#)

- “num_workers” is the number of processes that produce batches in parallel. You might set a positive num_workers. However, we tried, and it ran so slow and had errors. Thus, we decided to turn it into 0. It ran faster after all. You could try num_workers = 2, and if you get errors, change it back to 0. Making errors is one step of our learning process.

Next step, we create neural networks (i.e. perception classifier). Neural networks contain multiple perceptions used in supervised learning. Supervised learning is when your data is labeled separately. You use your labeled data, put them through some layers, and finally have your output classified. To do this, we need torch.nn

```
import torch.nn as nn
```

Our NeuralNetwork class receives an argument “nn.Module”. Just like PyTorchDataset class, the outline NeuralNetwork class has 2 specific functions: “__init__” and “forward”, and they don’t change the order.

```
class NeuralNetwork(nn.Module):
    def __init__(self, inputsize, hiddensize, outputsize):

    def forward(self, x):
```

With the above outline, we start to fill in the blank.

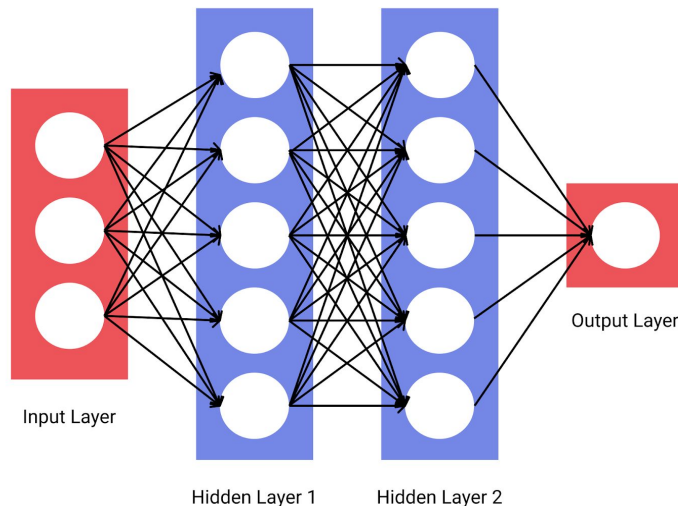
__init__ has 4 arguments:

```
self,
inputsize (input vector's size),
```

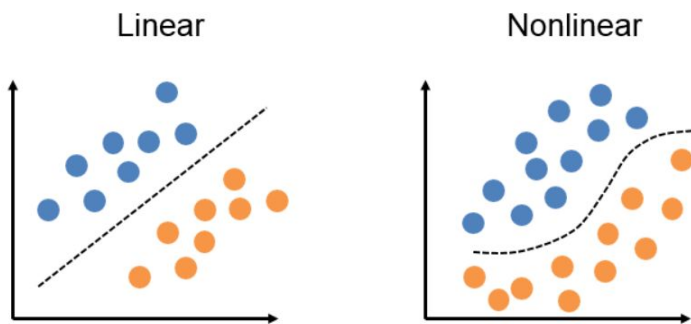
hiddensize (a linear layer's output size),
and outputsize (another linear layer's output size)

Remember to call "super(NeuralNetwork, self).__init__()" before writing the body of __init__ to prevent errors. After "super().__init__", "nn.Linear" will help us linearly transform incoming data:

$$y = xA^T + b \quad (\text{PyTorch documentation}).$$



The input layer receives data from our JSON and does some calculation before generating an output, which will be used for hidden layer 1's input. Standing between the input layer and the final output layer are 2 hidden layers. Hidden layers are not visible to us, but they are visible to neural networks and are powerful tools. We use hidden layers whenever our data is not separated linearly. Linearly separable data can be divided by a straight line, and non-linear separated data is not.



In short, we use hidden layers due to our non-linear data, and our layers are linear.

Therefore, we have 3 similar lines of code:

"self.linear = nn.Linear(input_size, output_size)".

From "input layer to hidden layer 1", we write: "self.l1 = nn.Linear(inputsize, hiddensize)".

From "hidden layer 1 to hidden layer 2", we write: "self.l2 = nn.Linear(hiddensize, hiddensize)".

From "hidden layer 2 to output layer", write: "self.l3 = nn.Linear(hiddensize, outputsize)".

Also, we use a Rectified Linear Unit ("ReLU") for one Linear layer's output before turning this output to the next Linear layer's input.


```
def __init__(self, inputsizes, hiddensizes, outputsizes):
    super(NeuralNetwork, self).__init__()
    self.l1 = nn.Linear(inputsizes, hiddensizes)
    self.l2 = nn.Linear(hiddensizes, hiddensizes)
    self.l3 = nn.Linear(hiddensizes, outputsizes)
    self.relu = nn.ReLU()
```

“forward” function has 2 arguments: self and input data tensor (x). Each input tensor will go through “relu” before it goes to the next layer. It will return output data (also called predicted-y or resulting tensor).

```
def forward(self, x):
    out = self.l1(x)
    out = self.relu(out)

    out = self.l2(out)
    out = self.relu(out)

    out = self.l3(out)
    return out
```

After all, the perception classifier we obtain:

```
class NeuralNetwork(nn.Module):
    def __init__(self, inputsizes, hiddensizes, outputsizes):
        super(NeuralNetwork, self).__init__()
        self.l1 = nn.Linear(inputsizes, hiddensizes)
        self.l2 = nn.Linear(hiddensizes, hiddensizes)
        self.l3 = nn.Linear(hiddensizes, outputsizes)
        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)
        return out
```

As mentioned above, perception is used in supervised learning, which eventually helps give us output as labeled data. Our labels are “tags” (like “greeting” tag, “goodbye” tag); therefore, outsize = len(tags). Our input are all tokenized words in “allword”; thus, insize = len(allword). Plus, we set hiddensize = 8 (another default). Our model will be trained by this code:

```
model = NeuralNetwork(insize, hiddensize, outsize).to(device)
```

“device” is either “cuda” or “cpu”. In our case, we didn’t download ‘cuda’, and the device would be ‘cpu’. To check what type of your device, we ask PyTorch whether or not cuda is available:

```

if torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')
print(device)

```

Remember to check device before making the model:

```

outsize = len(tags)
insize = len(allword)
hiddensize = 8
model = NeuralNetwork(insize, hiddensize, outsize).to(device)

```

Now, our model is moved to the designated device.

Next, we create a supervised training loop containing 6 steps. This loop is a “for” loop iterating over the number of “epochs”. For each iteration, we go over elements in “train_loader” from the DatasetLoader part above. Since “train_loader” contains our words and corresponding labels, we move those words and labels to our designated device. This first step is to get input data.

```

# step 1: get inputs
words = words.to(device)
labels = labels.to(device)

```

Plus, in the calculation of our model, there exists some computation of derivatives, which is done with the help of “gradient”. Noticing PyTorch continues adding all gradients of subsequent backward passes, we need to clear gradients in each iteration. In step #2, we “zero gradients” by writing optimizer.zero_grad() where optimizer is

```
optimizer = torch.optim.Adam(model.parameters(), lr = 0.001)
```

; lr = 0.001

is a default learning rate, and Adam stands for A Method for Stochastic Optimization.

```

# step 2: zero gradients (cleared gradients stored inside the model object)
optimizer.zero_grad()

```

Clearing these gradients will help the model update accurately.

Great! We have our inputs and clean gradients. Now, it’s time for the forward pass. Step #3: forward pass is used to compute predicted-y by putting the “words” of “train_loader” into our “model”.

```

# step 3: forward pass: compute predicted y
y_pred = model(words)

```

Simply write

Of course, when training, we want every optimization for our model, but loss still happens.

That’s why we compute the loss value in step #4 `loss = CEL(y_pred, labels)`,

where `CEL = nn.CrossEntropyLoss()`. CEL stands for Cross-Entropy-Loss, which is a criteria to check our neural network’s performance. Then we move “loss” to a backward pass in step #5:

```

# step 5: propagate backward the loss
loss.backward()

```

. Finally, we update weights by applying optimizer.

```
# step 6: update weights by using optimizer
optimizer.step()
```

. Putting everything together, we have

the following code cell:

```
epochs = 500
CEL = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr = 0.001)

for epoch in range(epochs):
    for (words, labels) in train_loader:

        # step 1: get inputs
        words = words.to(device)
        labels = labels.to(device)

        # step 2: zero gradients
        optimizer.zero_grad()

        # step 3: forward pass: compute predicted y
        y_pred = model(words)

        # step 4: compute the loss value that we wish to optimize
        # --> expected "labels" have type Long but found Int
        # cast the labels to Long
        labels = labels.to(dtype = torch.long)
        loss = CEL(y_pred, labels)

        # step 5: propagate backward the loss
        loss.backward()

        # step 6: update weights by using optimizer
        optimizer.step()

    # print loss
    if ((epoch + 1) % 100 == 0):
        print(f'epoch {epoch+1}/{epochs}, loss = {loss.item():.4f}')

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

Wow! We're almost there. Let's save all data to a python file (.pth) and randomly load possible responses of PyTorBot.

```
# save data
alldata = {
    "modelstate": model.state_dict(),
    "allwords": allword,
    "tags": tags,
    "inputsize": insize,
    "hiddensize": hiddensize,
    "outputsize": outsize
}

file = "alldata.pth"
torch.save(alldata, file)
print("saved")
```

```
# import random: random choice of possible responses
import random
data = torch.load(file)

allwords = data["allwords"]
thistag = data["tags"]
model_state = data["modelstate"]

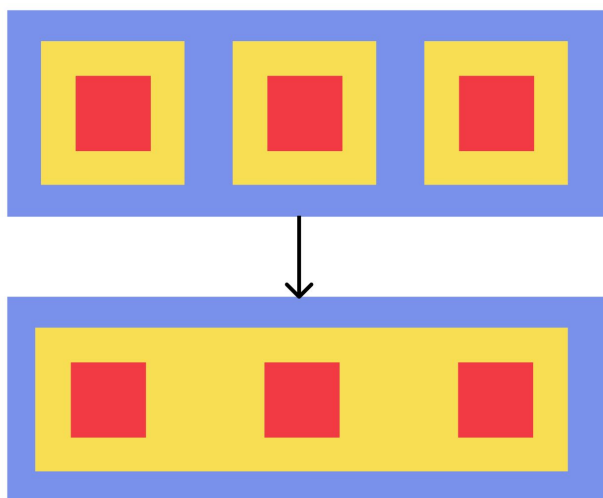
model = NeuralNetwork(data["inputsize"], data["hiddensize"], data["outputsize"]).to(device)
model.load_state_dict(model_state)
model.eval()
```

“load_state_dict” is new to us. Let’s figure it out. “state_dict” - an object in Python dictionary - connects layers with corresponding tensors ([PyTorch documentation](#)). We then call .eval() - a function that is opposite to .train(), and we use .eval() to prevent our model from re-computing loss or propagate backward loss, and avoid unnecessary changing in validation data.

All done now. Let’s call our bot’s name “PyTorBot”, receive user inputs, and randomly generate answers. To stop chatting, just simply enter ‘quit’, and PyTorBot will “break” the conversation.

```
botname = 'PyTorBot: '
print("Hello Hello. PyTorBot is here! How can I help you? Enter 'quit' to exit")
while True:
    userinput = input("PyTorBot's dear friend: ")
    if userinput == 'quit':
        break
    userinput = tokenizer.tokenize(userinput)
    userinput = [stem(word) for word in userinput]
```

Again, when getting user inputs, we compare them with a list of all words by the function “one_or_none” (mark 1.0 if “allwords” has “userinput”, and 0.0 if not available). We’ll receive a list of 1.0 or 0.0, which is equivalent to a tensor (named “tensorx”) having a shape of [len(allwords), 1]. We want to reshape it as [1, len(allword)] by writing “tensorx.reshape(1, tensorx.shape[0])”. For example,



Then we create a non-resizable tensor from numpy.ndarray.


```

tensorx = one_or_none(userinput, allwords)
tensorx = tensor_x.reshape(1, tensorx.shape[0])
tensorx = torch.from_numpy(tensorx)

```

We have an input tensor. Let's obtain our "output" by modelling "tensorx" and have our "tag" matched with "predicted.item()"

```

output = model(tensorx)
_, predicted = torch.max(output, dim = 1)
tag = thistag[predicted.item()]

```

"torch.max" will return as many elements from input tensor as possible, and it reduces its dimension to 1.

Finally, we apply torch.softmax() to make elements of tensor output be in [0,1] and add up to 1. ([PyTorch documentation](#)). In other words, torch.softmax() will return a tensor having the same dimension [1, len(allwords)] as the input and its values are in [0, 1]. Also, we check the probability of finding the items in the package of words. If probability is too small, PyTorBot doesn't understand and needs more clarification from user input.

```

probability = torch.softmax(output, dim = 1)
prob = probability[0][predicted.item()]

if prob.item() > 0:
    # check the tag in json list
    for element in myjson["list"]:
        if tag == stem(element["tag"]):
            print(botname, random.choice(element["responses"]))
else:
    print(botname, "Sorry. I don't get it. Can you help me clarify that?")

```

Ok, are you ready to run your PyTorBot? We're so exited!
Check out our demo to see how it works.

Thank you for reading our tutorial! As beginners, we are new to PyTorch. Our PyTorBot might take quite a lot of time to finish its training process, so we hope to keep practicing our skills to optimize the training process afterward.

