

[Open in app](#)[Get started](#)

Published in Chatbots Magazine



gk_

[Follow](#)May 7, 2017 · 9 min read · [Listen](#)

Save



Contextual Chatbots with Tensorflow

In conversations, context is king! We'll build a chatbot framework using Tensorflow and add some context handling to show how this can be approached.



[Open in app](#)[Get started](#)

“Whole World in your Hand” — Betty Newman-Maguire (<http://www.bettynewmanmaguire.ie/>)

Ever wonder why most chatbots lack conversational context?

How is this possible given the importance of context in nearly all conversations?

We’re going to create a chatbot framework and build a conversational model for an **island moped rental shop**. The chatbot for this small business needs to handle simple questions about hours of operation, reservation options and so on. We also want it to handle contextual responses such as inquiries about same-day rentals. Getting this right could save a vacation!

We’ll be working through 3 steps:

- We’ll transform conversational intent definitions to a Tensorflow model
- Next, we will build a chatbot framework to process responses
- Lastly, we’ll show how basic context can be incorporated into our response processor

We’ll be using tflearn, a layer above tensorflow, and of course Python. As always we’ll use iPython notebook as a tool to facilitate our work.

Transform Conversational Intent Definitions to a Tensorflow Model

The complete notebook for our first step is here.

A chatbot framework needs a structure in which conversational intents are defined. One clean way to do this is with a JSON file, like this.




[Open in app](#)
[Get started](#)

```

1  {"intents": [
2      {"tag": "greeting",
3       "patterns": ["Hi", "How are you", "Is anyone there?", "Hello", "Good day"],
4       "responses": ["Hello, thanks for visiting", "Good to see you again", "Hi there, how can I help?"],
5       "context_set": ""
6      },
7      {"tag": "goodbye",
8       "patterns": ["Bye", "See you later", "Goodbye"],
9       "responses": ["See you later, thanks for visiting", "Have a nice day", "Bye! Come back again soon."]
10     },
11     {"tag": "thanks",
12      "patterns": ["Thanks", "Thank you", "That's helpful"],
13      "responses": ["Happy to help!", "Any time!", "My pleasure"]
14     },
15     {"tag": "hours",
16      "patterns": ["What hours are you open?", "What are your hours?", "When are you open?" ],
17      "responses": ["We're open every day 9am-9pm", "Our hours are 9am-9pm every day"]
18     },

```

chatbot intents

Each conversational intent contains:

- a **tag** (a unique name)
- **patterns** (sentence patterns for our neural network text classifier)
- **responses** (one will be used as a response)

And later on we'll add *some basic contextual elements*.

First we take care of our imports:

```

1  # things we need for NLP
2  import nltk
3  from nltk.stem.lancaster import LancasterStemmer
4  stemmer = LancasterStemmer()
5
6  # things we need for Tensorflow
7  import numpy as np
8  import tflearn
9  import tensorflow as tf
10 import random

```

Tensorflow chat-bot model part_1 hosted with ❤️ by GitHub

[view raw](#)

Have a look at “Deep Learning in 7 lines of code” for a primer or [here](#) if you need to




[Open in app](#)
[Get started](#)

```
3 with open('intents.json') as json_data:
4     intents = json.load(json_data)
```

Tensorflow chat-bot model part_2 hosted with ❤️ by GitHub

[view raw](#)

With our intents JSON file loaded, we can now begin to organize our documents, words and classification classes.

```
1 words = []
2 classes = []
3 documents = []
4 ignore_words = ['?']
5 # loop through each sentence in our intents patterns
6 for intent in intents['intents']:
7     for pattern in intent['patterns']:
8         # tokenize each word in the sentence
9         w = nltk.word_tokenize(pattern)
10        # add to our words list
11        words.extend(w)
12        # add to documents in our corpus
13        documents.append((w, intent['tag']))
14        # add to our classes list
15        if intent['tag'] not in classes:
16            classes.append(intent['tag'])
17
18 # stem and lower each word and remove duplicates
19 words = [stemmer.stem(w.lower()) for w in words if w not in ignore_words]
20 words = sorted(list(set(words)))
21
22 # remove duplicates
23 classes = sorted(list(set(classes)))
24
25 print (len(documents), "documents")
26 print (len(classes), "classes", classes)
27 print (len(words), "unique stemmed words", words)
```

We create a list of documents (sentences), each sentence is a list of *stemmed words* and each document is associated with an intent (a class).




[Open in app](#)
[Get started](#)

```
'hello', 'help', 'hi', 'hour', 'how', 'i', 'is', 'kind', 'lat',
'lik', 'mastercard', 'mop', 'of', 'on', 'op', 'rent', 'see', 'tak',
'thank', 'that', 'ther', 'thi', 'to', 'today', 'we', 'what', 'when',
'which', 'work', 'you']
```

The stem ‘tak’ will match ‘take’, ‘taking’, ‘takers’, etc. We could clean the words list and remove useless entries but this will suffice for now.

Unfortunately this data structure won’t work with Tensorflow, we need to transform it further: *from documents of words into tensors of numbers*.

```
1  # create our training data
2  training = []
3  output = []
4  # create an empty array for our output
5  output_empty = [0] * len(classes)
6
7  # training set, bag of words for each sentence
8  for doc in documents:
9      # initialize our bag of words
10     bag = []
11     # list of tokenized words for the pattern
12     pattern_words = doc[0]
13     # stem each word
14     pattern_words = [stemmer.stem(word.lower()) for word in pattern_words]
15     # create our bag of words array
16     for w in words:
17         bag.append(1) if w in pattern_words else bag.append(0)
18
19     # output is a '0' for each tag and '1' for current tag
20     output_row = list(output_empty)
21     output_row[classes.index(doc[1])] = 1
22
23     training.append([bag, output_row])
24
25 # shuffle our features and turn into np.array
26 random.shuffle(training)
27 training = np.array(training)
28
29 # create train and test lists
```






[Open in app](#)
[Get started](#)

```

8  net = tflearn.regression(net)
9
10 # Define model and setup tensorboard
11 model = tflearn.DNN(net, tensorboard_dir='tflearn_logs')
12 # Start training (apply gradient descent algorithm)
13 model.fit(train_x, train_y, n_epoch=1000, batch_size=8, show_metric=True)
14 model.save('model.tflearn')

```

Tensorflow chat-bot model part_5 hosted with ❤️ by GitHub

[view raw](#)

This is the same tensor structure as we used in our 2-layer neural network in [our 'toy' example](#). Watching the model fit our training data never gets old...

```

# Define model
model = tflearn.DNN(net)
# Start training (apply gradient descent algorithm)
model.fit(train_x, train_y, n_epoch=1000, batch_size=16, show_metric=True)

Training Step: 47 | total loss: 0.61682 | time: 0.005s
| Adam | epoch: 024 | loss: 0.61682 - acc: 0.6751 -- iter: 16/22
Training Step: 48 | total loss: 0.60077 | time: 0.008s
| Adam | epoch: 024 | loss: 0.60077 - acc: 0.7006 -- iter: 22/22
--

In [33]: for i,t in enumerate(test x):

```

interactive build of a model in tflearn

To complete this section of work, we'll save ('pickle') our model and documents so the next notebook can use them.

```

1  # save all of our data structures
2  import pickle
3  pickle.dump( {'words':words, 'classes':classes, 'train_x':train_x, 'train_y':train_y}, open( "tr

```



[Open in app](#)[Get started](#)

Building Our Chatbot Framework

The complete notebook for our second step is [here](#).

We'll build a simple state-machine to handle responses, using our intents model (from the previous step) as our classifier. That's [how chatbots work](#).

A contextual chatbot framework is a classifier within a state-machine.

After loading the same imports, we'll *un-pickle* our model and documents as well as reload our intents file. Remember our chatbot framework is separate from our model build — you don't need to rebuild your model unless the intent patterns change. With several hundred intents and thousands of patterns the model could take several minutes to build.

```
1 # restore all of our data structures
```




[Open in app](#)
[Get started](#)

```

7 train_y = data['train_y']
8
9 # import our chat-bot intents file
10 import json
11 with open('intents.json') as json_data:
12     intents = json.load(json_data)

```

Tensorflow chat-bot response part_1 hosted with ❤️ by GitHub

[view raw](#)

Next we will load our saved Tensorflow (tflearn framework) model. Notice you first need to define the Tensorflow model structure just as we did in the previous section.

```

1 # load our saved model
2 model.load('./model.tflearn')

```

Tensorflow chat-bot response part_2 hosted with ❤️ by GitHub

[view raw](#)

Before we can begin processing intents, we need a way to produce a bag-of-words *from user input*. This is the same technique as we used earlier to create our training documents.

```

1 def clean_up_sentence(sentence):
2     # tokenize the pattern
3     sentence_words = nltk.word_tokenize(sentence)
4     # stem each word
5     sentence_words = [stemmer.stem(word.lower()) for word in sentence_words]
6     return sentence_words
7
8 # return bag of words array: 0 or 1 for each word in the bag that exists in the sentence
9 def bow(sentence, words, show_details=False):
10    # tokenize the pattern
11    sentence_words = clean_up_sentence(sentence)
12    # bag of words
13    bag = [0]*len(words)
14    for s in sentence_words:
15        for i,w in enumerate(words):
16            if w == s:
17                bag[i] = 1
18            if show_details:
19                print ("found in bag: %s" % w)
20

```



[Open in app](#)[Get started](#)

Let's look at a classification example, the most likely tag and its probability are returned.

```
classify('is your shop open today?')  
[('opentoday', 0.9264171123504639)]
```

Notice that 'is your shop open today?' is not one of the patterns for this intent: *"patterns": ["Are you open today?", "When do you open today?", "What are your hours today?"]* however the terms 'open' and 'today' proved irresistible to our model (they are prominent in the chosen intent).

We can now generate a chatbot response from user-input:

```
response('is your shop open today?')  
Our hours are 9am-9pm every day
```

And other context-free responses...

```
response('do you take cash?')  
We accept VISA, Mastercard and AMEX
```

```
response('what kind of mopeds do you rent?')  
We rent Yamaha, Piaggio and Vespa mopeds
```

```
response('Goodbye, see you later')  
Bye! Come back again soon.
```



[Open in app](#)[Get started](#)

Let's work in some basic context into our moped rental chatbot conversation.

Contextualization

We want to handle a question about renting a moped and ask if the rental is for today. That clarification question is a simple contextual response. If the user responds 'today' *and the context is the rental timeframe* then it's best they call the rental company's 1-800 #. No time to waste.

To achieve this we will add the notion of 'state' to our framework. This is comprised of a data-structure to maintain state and specific code to manipulate it while processing intents.

Because the state of our state-machine needs to be easily persisted, restored, copied, etc. it's important to keep it all in a data structure such as a dictionary.

Here's our response process with basic contextualization:




[Open in app](#)
[Get started](#)

```

6     # generate probabilities from the model
7     results = model.predict([bow(sentence, words))][0]
8     # filter out predictions below a threshold
9     results = [[i,r] for i,r in enumerate(results) if r>ERROR_THRESHOLD]
10    # sort by strength of probability
11    results.sort(key=lambda x: x[1], reverse=True)
12    return_list = []
13    for r in results:
14        return_list.append((classes[r[0]], r[1]))
15    # return tuple of intent and probability
16    return return_list
17
18    def response(sentence, userID='123', show_details=False):
19        results = classify(sentence)
20        # if we have a classification then find the matching intent tag
21        if results:
22            # loop as long as there are matches to process
23            while results:
24                for i in intents['intents']:
25                    # find a tag matching the first result
26                    if i['tag'] == results[0][0]:
27                        # set context for this intent if necessary
28                        if 'context_set' in i:
29                            if show_details: print ('context:', i['context_set'])
30                            context[userID] = i['context_set']
31
32                        # check if this intent is contextual and applies to this user's conversation
33                        if not 'context_filter' in i or \
34                            (userID in context and 'context_filter' in i and i['context_filter'] ==
35                             userID):
36                            if show_details: print ('tag:', i['tag'])
37                            # a random response from the intent
38                            return print(random.choice(i['responses']))
39
40            results.pop(0)

```

Our context state is a dictionary, it will contain state for each user. We'll use some unique identified for each user (eg. cell #). This allows our framework and state-machine to *maintain state for multiple users simultaneously*.

```
# create a data structure to hold user context
```




[Open in app](#)
[Get started](#)

```

2         # set context for this intent if necessary
3         if 'context_set' in i:
4             if show_details: print ('context:', i['context_set'])
5             context[userID] = i['context_set']
6
7         # check if this intent is contextual and applies to this user's conversation
8         if not 'context_filter' in i or \
9             (userID in context and 'context_filter' in i and i['context_filter'] == context[userID]):
10            if show_details: print ('tag:', i['tag'])
11            # a random response from the intent
12            return print(random.choice(i['responses']))

```

Tensorflow chat-bot response part 6 hosted with ❤ by GitHub

[view raw](#)

If an intent wants to **set** context, it can do so:

```

{
  "tag": "rental",
  "patterns": ["Can we rent a moped?", "I'd like to rent a moped", ... ],
  "responses": ["Are you looking to rent today or later this week?"],
  "context_set": "rentalday"
}

```

If another intent wants to be contextually linked to a context, it can do that:

```

{
  "tag": "today",
  "patterns": ["today"],
  "responses": ["For rentals today please call 1-800-MYMOPED", ...],
  "context_filter": "rentalday"
}

```

In this way, if a user just typed 'today' out of the blue (no context), our 'today' intent won't be processed. If they enter 'today' *as a response to our clarification question* (intent tag:'rental') then the intent is processed.

```

response('we want to rent a moped')
Are you looking to rent today or later this week?

```



[Open in app](#)[Get started](#)**context**

```
{'123': 'rentalday'}
```

We defined our ‘greeting’ intent to clear context, as is often the case with small-talk. We add a ‘show_details’ parameter to help us see inside.

```
response("Hi there!", show_details=True)
```

```
context: ''
```

```
tag: greeting
```

```
Good to see you again
```

Let’s try the ‘today’ input once again, a few notable things here...

```
response('today')
```

```
We're open every day from 9am-9pm
```

```
classify('today')
```

```
[('today', 0.5322513580322266), ('opentoday', 0.2611265480518341)]
```

First, our response to the context-free ‘today’ was different. Our classification produced 2 suitable intents, and the ‘opentoday’ was selected because the ‘today’ intent, while higher probability, was bound to *a context that no longer applied*. Context matters!

```
response("thanks, your great")
```

```
Happy to help!
```



[Open in app](#)[Get started](#)

A few things to consider now that contextualization is happening...

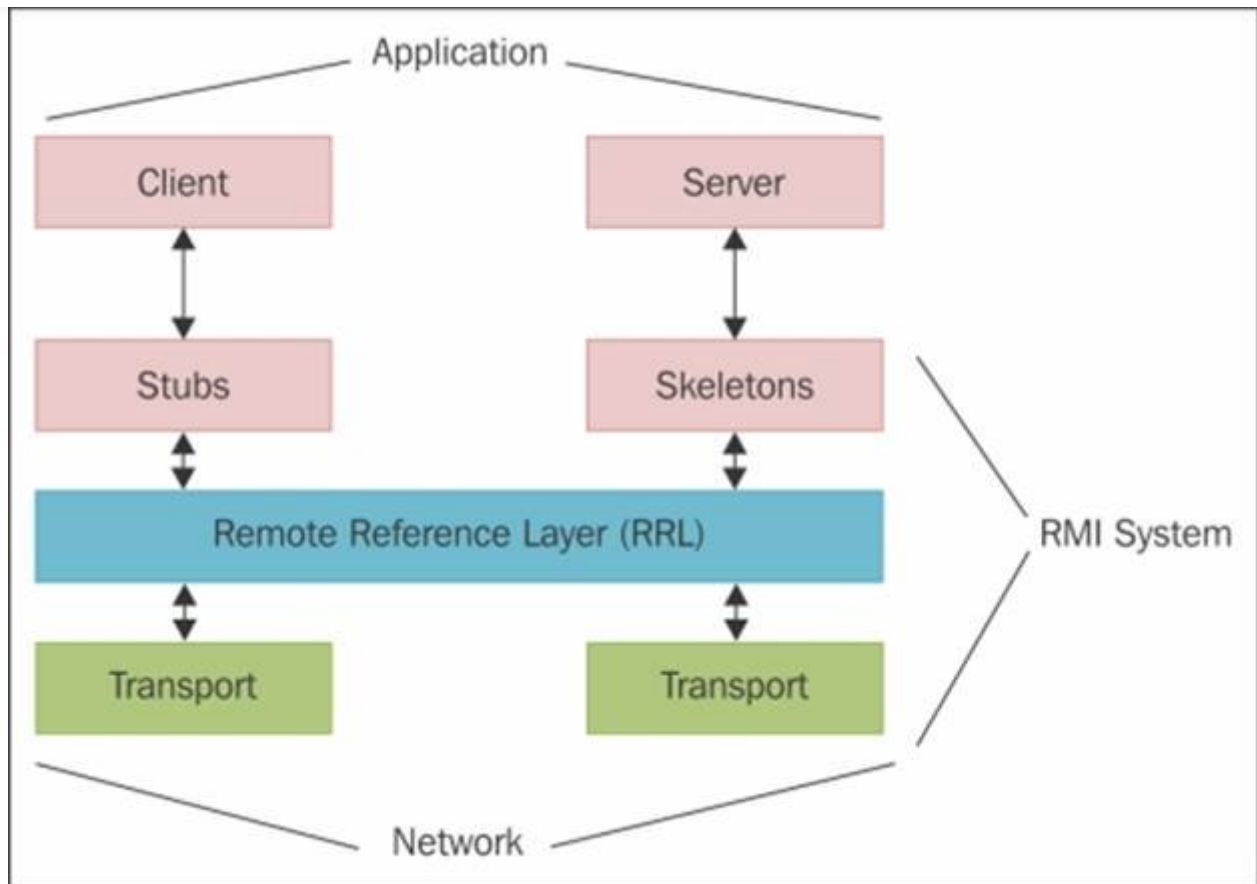
With State Comes Statefulness

That's right, your chatbot will no longer be happy *as a stateless service*.

Unless you want to reconstitute state, reload your model and documents — with every call to your chatbot framework, you'll need to make it *stateful*.

This isn't that difficult. You can run a stateful chatbot framework in its own process and call it using an RPC (remote procedure call) or RMI (remote method invocation), I recommend [Pyro](#).




[Open in app](#)
[Get started](#)


RMI client and server setup

The user-interface (client) is typically stateless, eg. HTTP or SMS.

Your chatbot *client* will make a Pyro function call, which your stateful service will handle. Voila!

Here's a step-by-step guide to build a Twilio SMS chatbot client, and here's one for FB Messenger.

Thou Shalt Not Store State in Local Variables

All state information must be placed in a data structure such as a dictionary, easily persisted, reloaded, or copied atomically.

Each user's conversation will carry context which will be carried statefully for that user. The user ID can be their cell #, a Facebook user ID, or some other unique identifier.

There are scenarios where a user's conversational state needs to be copied (by value)



[Open in app](#)[Get started](#)

Python dictionaries are your friend.

So now you have a chatbot framework, a recipe for making it a stateful service, and a starting-point for adding context. Most chatbot frameworks in the future will treat context seamlessly.

Think of creative ways for intents to impact and react to different context settings. Your users' context dictionary can contain a wide-variety of conversation context.

Enjoy!




credit: <https://wickedgoodweb.com>

Sign up for the Chatbots Magazine newsletter!

Get the essential briefing of the top chatbot stories, case studies and announcements






Open in app

Get started


Sign up


☐


I agree to leave chatbotsmagazine.com and submit this information, which will be collected and used according to [Upscribe's privacy policy](#).





34





 1













Clap below to recommend this article to others






Join
The Community




Apply
To Be A Writer





Follow
Chatbots Magazine

More from Chatbots Magazine

Follow








[Open in app](#)
[Get started](#)

Chatbots 2 min read



Nikita Tank · May 5, 2017

Online Courses For Bot Development

Bots 5 min read



Arte Merritt · May 5, 2017

Let's Chat.

Bots 4 min read



Abhishek Anand · May 5, 2017

The Celebrity Chatbot I Almost Brought To Life

Chatbots 11 min read



Adam Bastock · May 5, 2017

How Will Chatbots Transform Traditional Retail?

Chatbots 6 min read


[Read more from Chatbots Magazine](#)

Recommended from Medium



Guang

Use Multidimensional LSTM network to learn linear and non-linear mapping of matrices

near Mapping Matrix




[Open in app](#)[Get started](#) AI Technology & Systems in AITS Journal

Disaster Tweet Prediction on Cainvas

 Narasimha Karthik Jwalapuram in MLearning.ai

Approaching Data-centric AI using Fast.ai

 Mansimransinghanand


My 2 cents on “Deep Learning Fundamentals with Keras” course

 MRINAL WALIA in Artificial Intelligence in Plain English

Top 5 Open-Source Image Super-Resolution Projects To Boost Your Image Processing Tasks

 Anil Chandra Naidu Matcha

Summary for Multitask Learning* by Rich Caruana

 Grace Tenorio in Drop Engineering

Building a Recommender System Using Embeddings

[About](#) [Help](#) [Terms](#) [Privacy](#)

Get the Medium app

