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.

gk_ Follow
May 7, 2017 · 9 min read

“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.

```
{"intents": [
        {"tag": "greeting",
         "patterns": ["Hi", "How are you", "Is anyone there?", "Hello", "Good day"],
         "responses": ["Hello, thanks for visiting", "Good to see you again", "Hi there, how can I help?"],
         "context_set": ""
        },
        {"tag": "goodbye",
         "patterns": ["Bye", "See you later", "Goodbye"],
         "responses": ["See you later, thanks for visiting", "Have a nice day", "Bye! Come back again soon."]
        },
        {"tag": "thanks",
         "patterns": ["Thanks", "Thank you", "That's helpful"],
         "responses": ["Happy to help!", "Any time!", "My pleasure"]
        },
        {"tag": "hours",
         "patterns": ["What hours are you open?", "What are your hours?", "When are you open?" ],
         "responses": ["We're open every day 9am-9pm", "Our hours are 9am-9pm every day"]
        },
```

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:

```python
# things we need for NLP
import nltk
from nltk.stem.lancaster import LancasterStemmer
stemmer = LancasterStemmer()

# things we need for Tensorflow
import numpy as np
import tflearn
import tensorflow as tf
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 demystify Tensorflow.

```python
# import our chat-bot intents file
import json
with open('intents.json') as json_data:
    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.

```python
words = []
classes = []
documents = []
ignore_words = ['?']
# loop through each sentence in our intents patterns
for intent in intents['intents']:
    for pattern in intent['patterns']:
        # tokenize each word in the sentence
        w = nltk.word_tokenize(pattern)
        # add to our words list
        words.extend(w)
        # add to documents in our corpus
        documents.append((w, intent['tag']))
        # add to our classes list
        if intent['tag'] not in classes:
            classes.append(intent['tag'])

# stem and lower each word and remove duplicates
words = [stemmer.stem(w.lower()) for w in words if w not in ignore_words]
words = sorted(list(set(words)))

# remove duplicates
classes = sorted(list(set(classes)))

print (len(documents), "documents")
print (len(classes), "classes", classes)
print (len(words), "unique stemmed words", words)
```

Tensorflow chat-bot model part_3 hosted with ❤ by **GitHub** **view raw**

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).

```
27 documents
9 classes ['goodbye', 'greeting', 'hours', 'mopeds', 'opentoday',
'payments', 'rental', 'thanks', 'today']
44 unique stemmed words ["'d", 'a', 'ar', 'bye', 'can', 'card',
'cash', 'credit', 'day', 'do', 'doe', 'good', 'goodby', 'hav',
'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*.

```
# create our training data
training = []
```

```python
output = []
# 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]
    # stem each word
    pattern_words = [stemmer.stem(word.lower()) for word in pattern_words]
    # create our bag of words array
    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
    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
train_x = list(training[:,0])
train_y = list(training[:,1])
```

Tensorflow chat-bot model part_4 hosted with ❤ by **GitHub** **view raw**

Notice that our data is shuffled. Tensorflow will take some of this and use it as test data *to gauge accuracy for a newly fitted model*.

If we look at a single x and y list element, we see 'bag of words' arrays, one for the intent pattern, the other for the intent class.

```
train_x example: [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 1, 0, 0, 0, 1]
train_y example: [0, 0, 1, 0, 0, 0, 0, 0, 0]
```

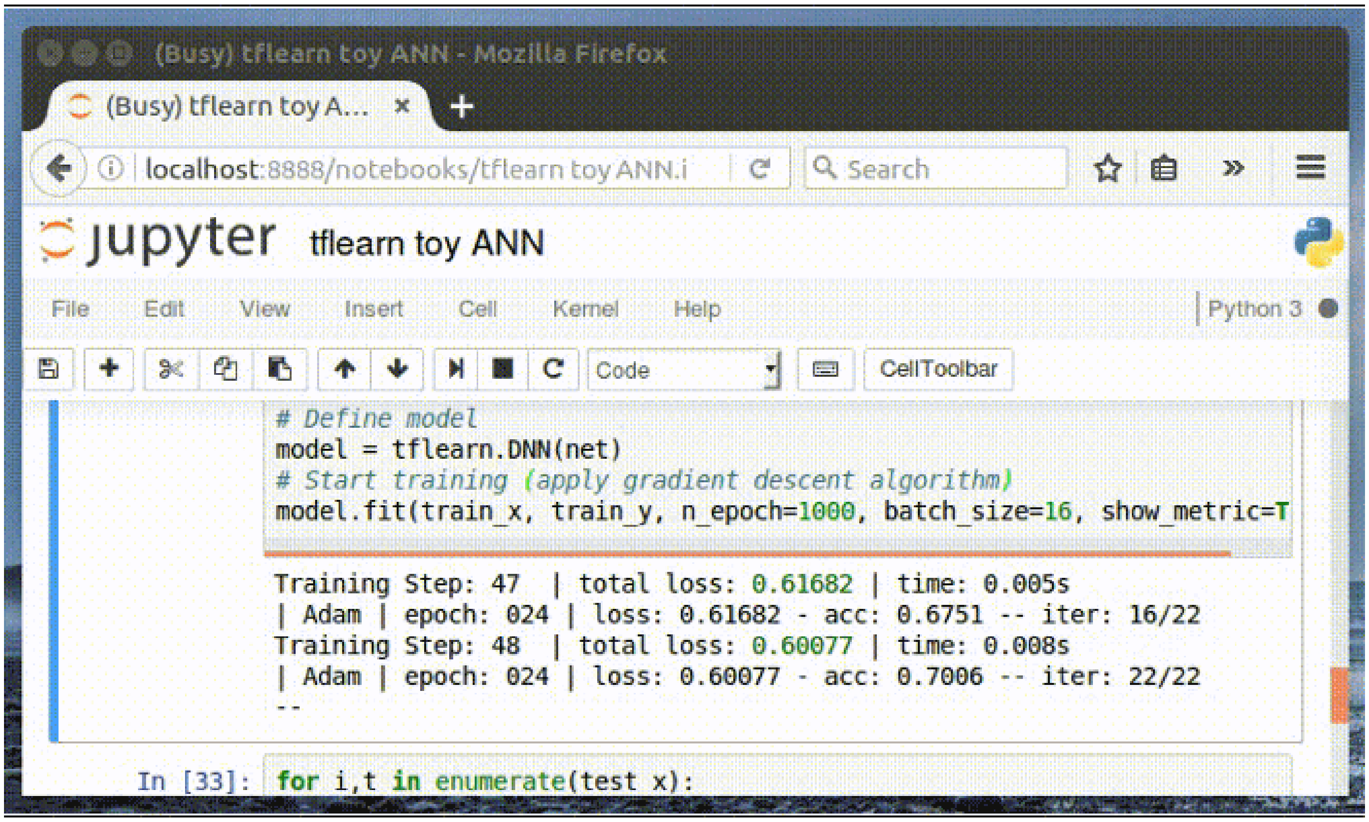
We’re ready to build our model.

```
# reset underlying graph data
tf.reset_default_graph()
# Build neural network
net = tflearn.input_data(shape=[None, len(train_x[0])])
net = tflearn.fully_connected(net, 8)
net = tflearn.fully_connected(net, 8)
net = tflearn.fully_connected(net, len(train_y[0]), activation='softmax')
net = tflearn.regression(net)

# Define model and setup tensorboard
model = tflearn.DNN(net, tensorboard_dir='tflearn_logs')
# Start training (apply gradient descent algorithm)
model.fit(train_x, train_y, n_epoch=1000, batch_size=8, show_metric=True)
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=T
```

```
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
--
```

```
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.

```
# save all of our data structures
import pickle
pickle.dump( {'words':words, 'classes':classes, 'train_x':train_x, 'train_y':train_y}, op
```

Tensorflow chat-bot model part_7 hosted with ❤ by GitHub view raw

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.

```
# restore all of our data structures
import pickle
data = pickle.load( open( "training_data", "rb" ) )
```

```python
words = data['words']
classes = data['classes']
train_x = data['train_x']
train_y = data['train_y']

# import our chat-bot intents file
import json
with open('intents.json') as json_data:
    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.

```python
# load our saved model
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.

```python
def clean_up_sentence(sentence):
    # tokenize the pattern
    sentence_words = nltk.word_tokenize(sentence)
    # stem each word
    sentence_words = [stemmer.stem(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=False):
    # tokenize the pattern
    sentence_words = clean_up_sentence(sentence)
    # bag of words
    bag = [0]*len(words)
    for s in sentence_words:
        for i,w in enumerate(words):
            if w == s:
                bag[i] = 1
                if show_details:
                    print ("found in bag: %s" % w)

    return(np.array(bag))
```

Tensorflow chat-bot response part_3 hosted with ❤ by GitHub view raw

```
p = bow("is your shop open today?", words)
print (p)
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 1 0 0 0 0 0 1 0]
```

We are now ready to build our response processor.

```
ERROR_THRESHOLD = 0.25
def classify(sentence):
    # generate probabilities from the model
    results = model.predict([bow(sentence, words)])[0]
    # filter out predictions below a threshold
    results = [[i,r] for i,r in enumerate(results) 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((classes[r[0]], r[1]))
    # return tuple of intent and probability
    return return_list

def response(sentence, userID='123', show_details=False):
    results = classify(sentence)
    # if we have a classification then find the matching intent tag
    if results:
        # loop as long as there are matches to process
        while results:
            for i in intents['intents']:
                # find a tag matching the first result
                if i['tag'] == results[0][0]:
                    # a random response from the intent
                    return print(random.choice(i['responses']))

            results.pop(0)
```

Tensorflow chat-bot response part_4 hosted with ❤ by GitHub view raw

Each sentence passed to response() is classified. Our classifier uses **model.predict()** and is lighting fast. The probabilities returned by the model are lined-up with our intents definitions to produce a list of potential responses.

If one or more classifications are above a threshold, we see if a tag matches an intent and then process that. We'll treat our classification list as a stack and pop off the stack looking for a suitable match until we find one, or it's empty.

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.
```

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:

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*
> *context = {}*

The context handlers are added within the intent processing flow, shown again below:

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?

response('today')
Same-day rentals please call 1-800-MYMOPED
```

Our context state changed:

```
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!
```

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.

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) and then restored as a result of intent processing. If your state machine carries state across variables within your framework you will have a difficult time making this work in real life scenarios.

> 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

👏👏Clap below to recommend this article to others👏👏

🙌 **Join** The Community

🤖 **Apply** To Be A Writer

❤️ **Follow** Chatbots Magazine

Chatbots TensorFlow Artificial Intelligence Python Tech

About Write Help Legal

Get the Medium app

Download on the App Store

GET IT ON Google Play