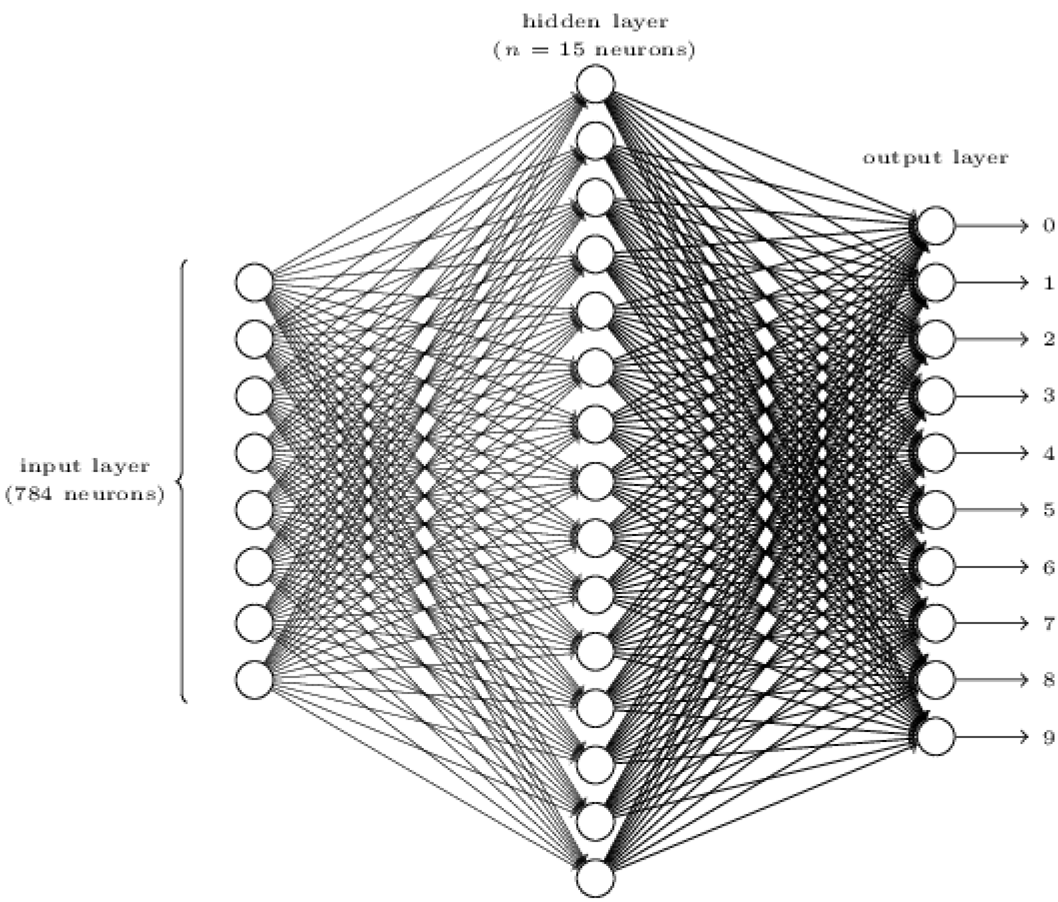
Text Classification using Neural Networks

gk_ Follow

Jan 26, 2017 · 6 min read

Understanding how chatbots work is important. A fundamental piece of machinery inside a chat-bot is the *text classifier*. Let's look at the inner workings of an artificial neural network (ANN) for text classification.



multi-layer ANN

We'll use 2 layers of neurons (1 hidden layer) and a "bag of words" approach to organizing our training data. Text classification comes in 3 flavors: *pattern matching*, *algorithms*, *neural nets*. While the algorithmic approach using Multinomial Naive Bayes is surprisingly effective, it suffers from 3 fundamental flaws:

- **the algorithm produces a *score*** rather than a probability. We want a probability to ignore predictions below some threshold. This is akin to a 'squelch' dial on a VHF radio.
- the algorithm 'learns' from examples of what *is* in a class, but **not what isn't**. This learning of patterns of what does *not* belong to a class is often very important.
- classes with disproportionately large training sets can create distorted classification scores, forcing the algorithm to **adjust scores relative to class size**. This is not ideal.

As with its 'Naive' counterpart, this classifier isn't attempting to understand *the meaning of a sentence*, it's trying to classify it. In fact so called "AI chat-bots" do not understand language, but that's another story.

If you are new to artificial neural networks, here is how they work.

To understand an algorithm approach to classification, see here.

Let's examine our text classifier one section at a time. We will take the following steps:

1. refer to **libraries** we need
2. provide **training data**
3. **organize** our data
4. **iterate**: code + test the results + tune the model
5. **abstract**

The code is here, we're using iPython notebook which is a super productive way of working on data science projects. The code syntax is Python.

We begin by importing our natural language toolkit. We need a way to reliably tokenize sentences into words and a way to stem words.

```
# use natural language toolkit
import nltk
from nltk.stem.lancaster import LancasterStemmer
import os
import json
import datetime
stemmer = LancasterStemmer()
```

text_ANN_part1 hosted with ❤ by GitHub view raw

And our training data, 12 sentences belonging to 3 classes ('intents').

```
# 3 classes of training data
training_data = []
training_data.append({"class":"greeting", "sentence":"how are you?"})
training_data.append({"class":"greeting", "sentence":"how is your day?"})
training_data.append({"class":"greeting", "sentence":"good day"})
training_data.append({"class":"greeting", "sentence":"how is it going today?"})

training_data.append({"class":"goodbye", "sentence":"have a nice day"})
training_data.append({"class":"goodbye", "sentence":"see you later"})
training_data.append({"class":"goodbye", "sentence":"have a nice day"})
training_data.append({"class":"goodbye", "sentence":"talk to you soon"})

training_data.append({"class":"sandwich", "sentence":"make me a sandwich"})
training_data.append({"class":"sandwich", "sentence":"can you make a sandwich?"})
training_data.append({"class":"sandwich", "sentence":"having a sandwich today?"})
training_data.append({"class":"sandwich", "sentence":"what's for lunch?"})
print ("%s sentences in training data" % len(training_data))
```

text_ANN_part2 hosted with ❤ by GitHub view raw

```
12 sentences in training data
```

We can now organize our data structures for *documents*, *classes* and *words*.

```
words = []
```

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

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

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

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

text_ANN_part3 hosted with ❤ by GitHub view raw

```
12 documents
3 classes ['greeting', 'goodbye', 'sandwich']
26 unique stemmed words ['sandwich', 'hav', 'a', 'how', 'for', 'ar',
'good', 'mak', 'me', 'it', 'day', 'soon', 'nic', 'lat', 'going',
'you', 'today', 'can', 'lunch', 'is', "'s", 'see', 'to', 'talk',
'yo', 'what']
```

Notice that each word is stemmed and lower-cased. Stemming helps the machine equate words like “have” and “having”. We don’t care about case.

the dog is on the table

Our training data is transformed into “bag of words” for each sentence.

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

    training.append(bag)
    # output is a '0' for each tag and '1' for current tag
    output_row = list(output_empty)
    output_row[classes.index(doc[1])] = 1
    output.append(output_row)

# sample training/output
i = 0
w = documents[i][0]
print ([stemmer.stem(word.lower()) for word in w])
print (training[i])
print (output[i])
```

text_ANN_part4 hosted with ❤ by GitHub view raw

```
['how', 'ar', 'you', '?']
[0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
```

```
0, 0, 0]
[1, 0, 0]
```

The above step is a classic in text classification: each training sentence is reduced to an array of 0's and 1's against the array of unique words in the corpus.

```
['how', 'are', 'you', '?']
```

is stemmed:

```
['how', 'ar', 'you', '?']
```

then transformed to input: *a 1 for each word in the bag (the ? is ignored)*

```
[0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0]
```

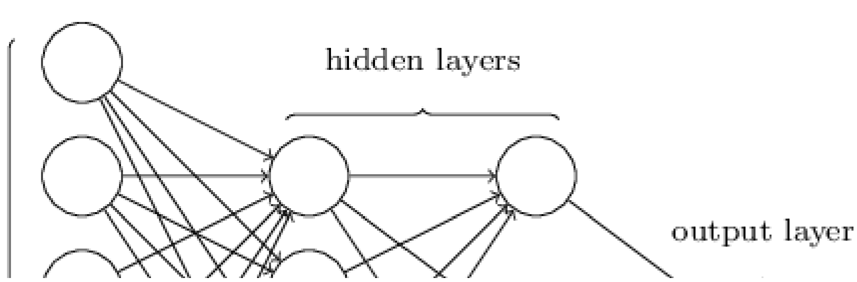
and output: *the first class*

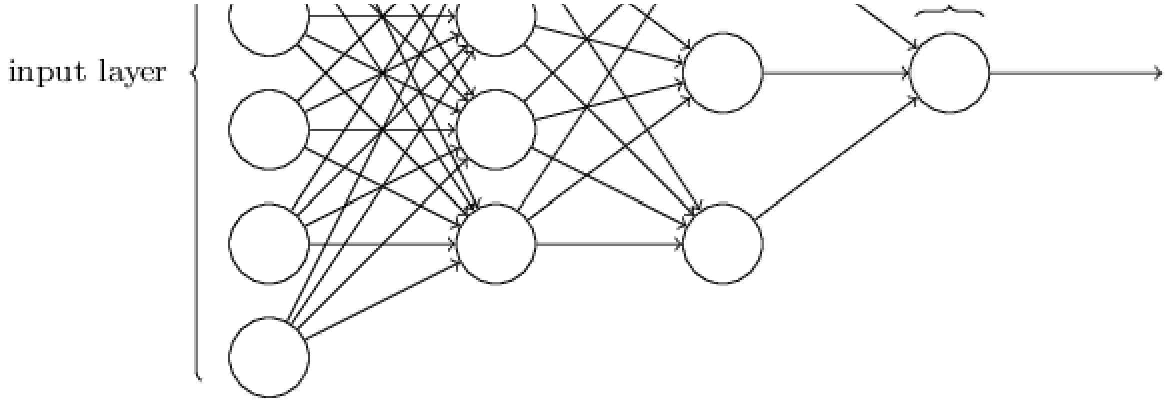
```
[1, 0, 0]
```

Note that a sentence could be given multiple classes, *or none*.

Make sure the above makes sense and play with the code until you grok it.

Your first step in machine learning is to have clean data.

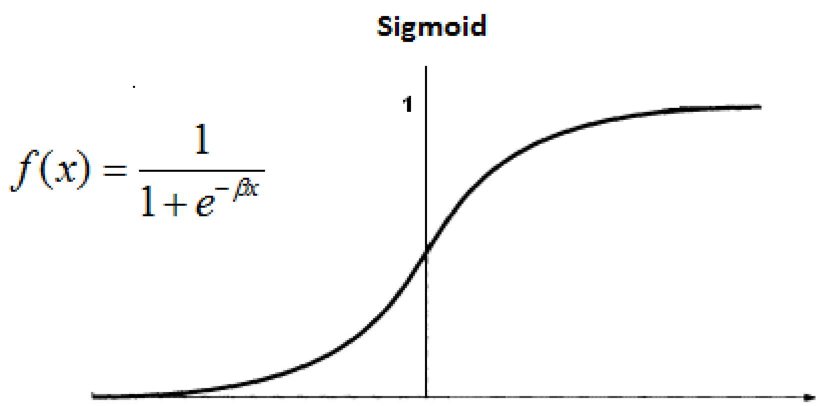




Next we have our core functions for our 2-layer neural network.

If you are new to artificial neural networks, here is how they work.

We use numpy because we want our matrix multiplication to be fast.



We use a sigmoid function to normalize values and its derivative to measure the error rate. Iterating and adjusting until our error rate is acceptably low.

Also below we implement our bag-of-words function, transforming an input sentence into an array of 0's and 1's. This matches precisely with our transform for training data, always crucial to get this right.

```
import numpy as np
import time

# compute sigmoid nonlinearity
def sigmoid(x):
```

```python
    output = 1/(1+np.exp(-x))
    return output

# convert output of sigmoid function to its derivative
def sigmoid_output_to_derivative(output):
    return output*(1-output)

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))

def think(sentence, show_details=False):
    x = bow(sentence.lower(), words, show_details)
    if show_details:
        print ("sentence:", sentence, "\n bow:", x)
    # input layer is our bag of words
    l0 = x
    # matrix multiplication of input and hidden layer
    l1 = sigmoid(np.dot(l0, synapse_0))
    # output layer
    l2 = sigmoid(np.dot(l1, synapse_1))
    return l2
```

text_ANN_part5 hosted with ❤ by **GitHub** view raw

And now we code our neural network training function to create synaptic weights. Don’t get too excited, this is mostly matrix multiplication — from middle-school math class.

```
def train(X, y, hidden_neurons=10, alpha=1, epochs=50000, dropout=False, dropout_percent=0.5):

    print ("Training with %s neurons, alpha:%s, dropout:%s %s" % (hidden_neurons, str(alpha), dro
    print ("Input matrix: %sx%s    Output matrix: %sx%s" % (len(X),len(X[0]),1, len(classes)) )
    np.random.seed(1)

    last_mean_error = 1
    # randomly initialize our weights with mean 0
    synapse_0 = 2*np.random.random((len(X[0]), hidden_neurons)) - 1
    synapse_1 = 2*np.random.random((hidden_neurons, len(classes))) - 1

    prev_synapse_0_weight_update = np.zeros_like(synapse_0)
    prev_synapse_1_weight_update = np.zeros_like(synapse_1)

    synapse_0_direction_count = np.zeros_like(synapse_0)
    synapse_1_direction_count = np.zeros_like(synapse_1)

    for j in iter(range(epochs+1)):

        # Feed forward through layers 0, 1, and 2
        layer_0 = X
        layer_1 = sigmoid(np.dot(layer_0, synapse_0))

        if(dropout):
            layer_1 *= np.random.binomial([np.ones((len(X),hidden_neurons))],1-dropout_percent)[(

        layer_2 = sigmoid(np.dot(layer_1, synapse_1))

        # how much did we miss the target value?
        layer_2_error = y - layer_2

        if (j% 10000) == 0 and j > 5000:
            # if this 10k iteration's error is greater than the last iteration, break out
            if np.mean(np.abs(layer_2_error)) < last_mean_error:
                print ("delta after "+str(j)+" iterations:" + str(np.mean(np.abs(layer_2_error))
                last_mean_error = np.mean(np.abs(layer_2_error))
            else:
                print ("break:", np.mean(np.abs(layer_2_error)), ">", last_mean_error )
                break

        # in what direction is the target value?
```

```
        # in what direction is the target value?
        # were we really sure? if so, don't change too much.
        layer_2_delta = layer_2_error * sigmoid_output_to_derivative(layer_2)

        # how much did each l1 value contribute to the l2 error (according to the weights)?
        layer_1_error = layer_2_delta.dot(synapse_1.T)

        # in what direction is the target l1?
        # were we really sure? if so, don't change too much.
        layer_1_delta = layer_1_error * sigmoid_output_to_derivative(layer_1)

        synapse_1_weight_update = (layer_1.T.dot(layer_2_delta))
        synapse_0_weight_update = (layer_0.T.dot(layer_1_delta))

        if(j > 0):
            synapse_0_direction_count += np.abs(((synapse_0_weight_update > 0)+0) - ((prev_synap
            synapse_1_direction_count += np.abs(((synapse_1_weight_update > 0)+0) - ((prev_synap

        synapse_1 += alpha * synapse_1_weight_update
        synapse_0 += alpha * synapse_0_weight_update

        prev_synapse_0_weight_update = synapse_0_weight_update
        prev_synapse_1_weight_update = synapse_1_weight_update

    now = datetime.datetime.now()

    # persist synapses
    synapse = {'synapse0': synapse_0.tolist(), 'synapse1': synapse_1.tolist(),
               'datetime': now.strftime("%Y-%m-%d %H:%M"),
               'words': words,
               'classes': classes
              }
    synapse_file = "synapses.json"

    with open(synapse_file, 'w') as outfile:
        json.dump(synapse, outfile, indent=4, sort_keys=True)
    print ("saved synapses to:", synapse_file)
```

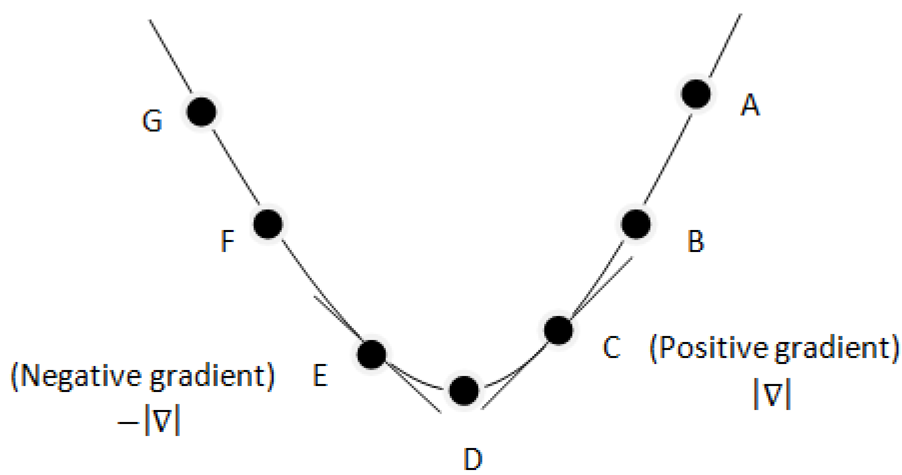
text_ANN_part6 hosted with ❤ by **GitHub** view raw

credit Andrew Trask https://iamtrask.github.io//2015/07/12/basic-python-network/

We are now ready to build our neural network *model*, we will save this as a json structure to represent our synaptic weights.

You should experiment with different 'alpha' (gradient descent parameter) and see how it affects the error rate. This parameter helps our error adjustment find the lowest error rate:

synapse_0 += **alpha** * synapse_0_weight_update



We use 20 neurons in our hidden layer, you can adjust this easily. These parameters will vary depending on the dimensions and shape of your training data, tune them down to ~10^-3 as a reasonable error rate.

```
X = np.array(training)
y = np.array(output)

start_time = time.time()

train(X, y, hidden_neurons=20, alpha=0.1, epochs=100000, dropout=False, dropout_percent=0.2)

elapsed_time = time.time() - start_time
print ("processing time:", elapsed_time, "seconds")
```

text_ANN_part7 hosted with ❤ by GitHub view raw

```
Training with 20 neurons, alpha:0.1, dropout:False
Input matrix: 12x26    Output matrix: 1x3
delta after 10000 iterations:0.0062613597435
```

```
delta after 20000 iterations:0.00428296074919
delta after 30000 iterations:0.00343930779307
delta after 40000 iterations:0.00294648034566
delta after 50000 iterations:0.00261467859609
delta after 60000 iterations:0.00237219554105
delta after 70000 iterations:0.00218521899378
delta after 80000 iterations:0.00203547284581
delta after 90000 iterations:0.00191211022401
delta after 100000 iterations:0.00180823798397
saved synapses to: synapses.json
processing time: 6.501226902008057 seconds
```

The synapse.json file contains all of our synaptic weights, **this is our model.**

This **classify()** function is all that's needed for the classification once synapse weights have been calculated: ~15 lines of code.

The catch: if there's a change to the training data our model will need to be re-calculated. For a very large dataset this could take a non-insignificant amount of time.

We can now generate the probability of a sentence belonging to one (or more) of our classes. This is super fast because it's dot-product calculation in our previously defined **think()** function.

```
# probability threshold
ERROR_THRESHOLD = 0.2
# load our calculated synapse values
synapse_file = 'synapses.json'
with open(synapse_file) as data_file:
    synapse = json.load(data_file)
    synapse_0 = np.asarray(synapse['synapse0'])
    synapse_1 = np.asarray(synapse['synapse1'])

def classify(sentence, show_details=False):
    results = think(sentence, show_details)

    results = [[i,r] for i,r in enumerate(results) if r>ERROR_THRESHOLD ]
    results.sort(key=lambda x: x[1], reverse=True)
    return_results =[[classes[r[0]],r[1]] for r in results]
    print ("%s \n classification: %s" % (sentence, return_results))
    return return_results

classify("sudo make me a sandwich")
classify("how are you today?")
classify("talk to you tomorrow")
classify("who are you?")
classify("make me some lunch")
classify("how was your lunch today?")
print()
classify("good day", show_details=True)
```

text_ANN_part7 hosted with ❤ by GitHub view raw

sudo make me a sandwich
 [['sandwich', 0.99917711814437993]]
how are you today?
 [['greeting', 0.99864563257858363]]
talk to you tomorrow
 [['goodbye', 0.95647479275905511]]
who are you?
 [['greeting', 0.8964283843977312]]
make me some lunch
 [['sandwich', 0.95371924052636048]]
how was your lunch today?
 [['greeting', 0.99120883810944971], ['sandwich', 0.31626066870883057]]

Experiment with other sentences and different probabilities, you can then add training data and improve/expand the model. Notice the solid predictions with scant training data.

Some sentences will produce multiple predictions (above a threshold). You will need to establish the right threshold level for your application. Not all text classification scenarios are the same: *some predictive situations require more confidence than others.*

The last classification shows some internal details:

```
found in bag: good
found in bag: day
sentence: good day
 bow: [0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
good day
 [['greeting', 0.99664077655648697]]
```

Notice the bag-of-words (bow) for the sentence, 2 words matched our corpus. The neural-net also learns from the 0's, the non-matching words.

A low-probability classification is easily shown by providing a sentence where 'a' (common word) is the only match, for example:

```
found in bag: a
sentence: a burrito!
 bow: [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]
a burrito!
 [['sandwich', 0.61776860634647834]]
```

Here you have a fundamental piece of machinery for building a chat-bot, capable of handling a large # of classes ('intents') and suitable for classes with limited or extensive training data ('patterns'). Adding one or more responses to an intent is trivial.

Thanks to Alexander Pinto.

Machine Learning   Neural Networks   Artificial Intelligence   Chatbots   Python