

# Building a Dead Simple Speech Recognition Engine using ConvNet in Keras



Manash Kumar Mandal

Follow

Nov 22, 2017 · 7 min read

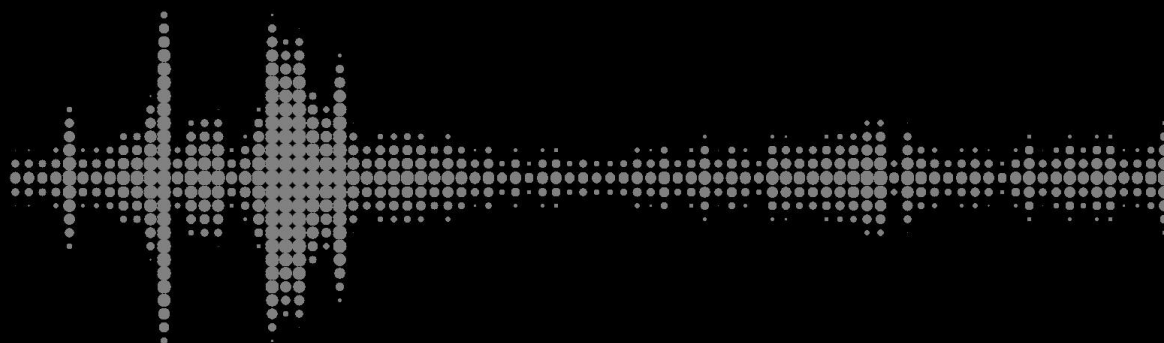


Image Courtesy (<https://forums.creativecow.net/thread/2/1067998>)

So you've classified MNIST dataset using Deep Learning libraries and want to do the same with speech recognition! Well continuous speech recognition is a bit tricky so to keep everything simple I am going to start with a simpler problem instead. Which is **word** recognition. I've seen a competition going on at [Kaggle](#) and couldn't help but downloading the dataset.

If you think this blog post will make you an expert in Speech Recognition field please feel free to skip it. I am going to show you some quick techniques to be up and running in speech recognition area rather going deeper.

## Before Beginning

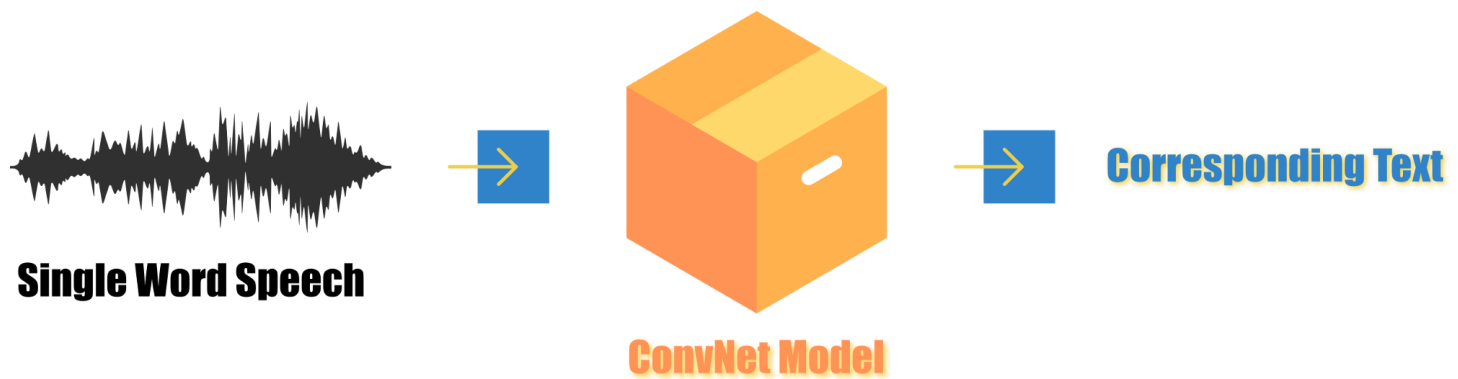
- Python 3.5 should be installed
- Following libraries will be used throughout the article, make sure you've installed it before trying out the codes.
- `librosa, keras, tensorflow, scikit-learn, numpy, scipy`

## So what am I going to cook?



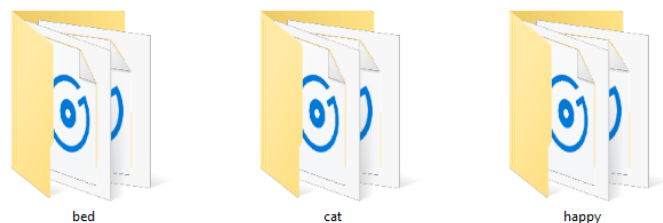
I'm the one who cooks

Nothing special, basically this one.



Yes I know directly feeding a speech signal to a ConvNet model doesn't seem right. There are some preprocessing steps which I'll introduce soon enough.

## Let's take a look at the data



There are basically three classes to recognize from. You can add as much as you may wish. Just change the number of output at the softmax layer and you'll be good to go. Don't forget to download data from Kaggle if you are willing to classify more than these three.

Each folder contains approximately 1700 audio files. The name of the folder is actually the **label** of those audio files. You can play some audio files randomly to get an overall idea.

The task will be to classify an audio between **bed**, **cat** and **happy**—these three classes.

## First things first

You can't just feed an audio file to a ConvNet, that's outrageous! We need to prepare a **fixed size vector** for each audio file for classification.


What can we do here? Represent the audio files into one-hot encoding?  
That doesn't make any sense, right?

## Embedding

An **embedding** is a mapping from discrete objects, such as words, to vectors of real numbers. [From TensorFlow]

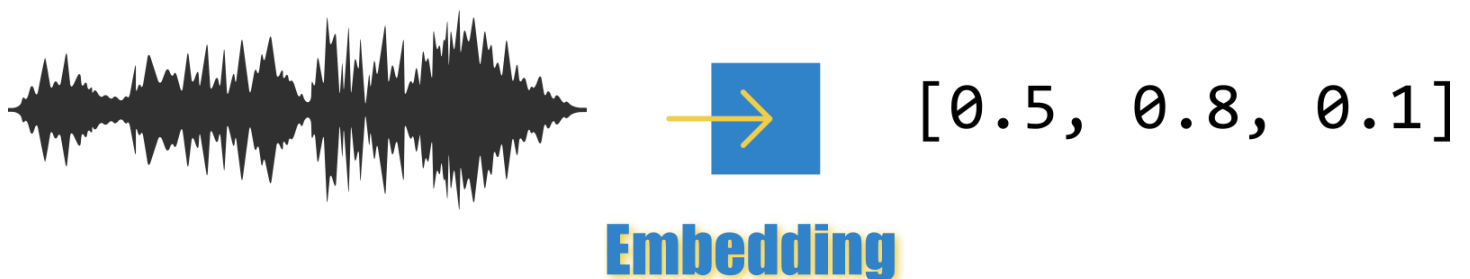
TensorFlow guide contains a very good guide on embedding. If you have no idea what I am talking about, check it out.

<b>Embeddings   TensorFlow</b>  This document introduces the concept of embeddings, gives a simple example of how to... <a href="http://www.tensorflow.org">www.tensorflow.org</a>	
---	--

<b>Vector Representations of Words   TensorFlow</b>  The goal is to make an update to the embedding parameters $\theta$ to improve (in this case,... <a href="http://www.tensorflow.org">www.tensorflow.org</a>	<p>IMAGES</p>  <p>Image pixels DENSE</p>
--	---

But we are dealing with a specific domain of embedding. Which is audio embedding.

## Audio Embedding



So here is the problem, what technique we should actually use to embed the audio into vector space? There are a lot of techniques you can find online, even there are some python packages solely for audio

feature extraction. But to keep everything simple. I am going to use the most obvious and simple (!) one which is called **MFCC** encoding, which is super effective for working with speech signals.

## MFCC (Mel Frequency Cepstral Coefficients)

You can know a lot from internet by searching a bit. Explaining MFCC is not the target of this article. In short,

*In sound processing, the mel-frequency cepstrum (MFC) is a representation of the short-term power spectrum of a sound, based on a linear cosine transform of a log power spectrum on a nonlinear mel scale of frequency.*

Yeah a lot to process, you can get an overview how this is computed from an audio signal. Which is clearly written in Wikipedia's wiki.

### Mel-frequency cepstrum - Wikipedia

Mel-frequency cepstral coefficients ( MFCCs) are coefficients that collectively make up an MFC....  
en.wikipedia.org

For the time being, just assume that MFCC can create useful vectors from audio signal for our deep learning model to work with.

## Preprocessing Audio Files

The whole code can [be found here](#). So don't worry about it. I am going to explain my workflow and explain some piece of codes.

1. First step will be to read the audio file from given path
2. I am going to take audio input from one channel only
3. Then perform downsample operation [You can skip it if you have enough computational power]
4. Then compute **MFCC** using `librosa` library
5. MFCC vectors might vary in size for different audio input, remember ConvNets can't handle sequence data so we need to prepare a fixed size vector for all of the audio files.

6. To overcome this problem all we need to do is to pad the output vectors with constant value (the one I used here is `0` ).

Here is my function which reads an audio file and computes MFCC.

### wav2mfcc

```
1 import numpy as np
2 import librosa
3
4 def wav2mfcc(file_path, max_pad_len=11):
5     wave, sr = librosa.load(file_path, mono=True, sr=None)
6     wave = wave[:3]
7     mfcc = librosa.feature.mfcc(wave, sr=16000)
8     pad_width = (max_pad_len - mfcc.shape[1],
```

## Preparing MFCC for all available audio files and save the computed vectors in a NumPy file

In `get_labels` function there are some unnecessary codes. I'll fix those later if I get enough time. Let's ignore it for now.

```
1 import numpy as np
2 import os
3
4 DATA_PATH = "./data/"
5
6 # Input: Folder Path
7 # Output: Tuple (Label, Indices of the labels, one-hot)
8 def get_labels(path=DATA_PATH):
9     labels = os.listdir(path)
10    label_indices = np.arange(0, len(labels))
11    return labels, label_indices, to_categorical(label
```

get\_labels.py hosted with ❤ by GitHub

[view raw](#)

```
1 import numpy as np
2
3 def save_data_to_array(path=DATA_PATH, max_pad_len=11):
4     labels, _, _ = get_labels(path)
5
```

`save_data_to_array` reads all audio files from each directory and save the vectors in a `.npy` file which is named after the name of the label.

Since computing MFCC is time consuming, we will do it only once. And for later times we will just load it from the saved files, which will buy us some time to do other things ;)

## Prepare Train set and Test set

Using the previous function we can compute MFCC but now we need to prepare train set and test set based on the data we have. To do this, I'll take advantage of `sklearn`'s nice function `train_test_split` which will automatically split the whole dataset.

```
1  import numpy as np
2  from sklearn.model_selection import train_test_split
3
4  def get_train_test(split_ratio=0.6, random_state=42):
5      # Get available labels
6      labels, indices, _ = get_labels(DATA_PATH)
7
8      # Getting first arrays
9      X = np.load(labels[0] + '.npy')
10     y = np.zeros(X.shape[0])
11
12     # Append all of the dataset into one single array,
13     for i, label in enumerate(labels[1:]):
14         x = np.load(label + '.npy')
```

## Building the CNN model!

Preprocessing takes away most of the time and it's a pretty tedious job. But when everything works out, it feels good, right?

## But Wait! This audio is not three dimensional! How can I possibly apply CNN on it?

Good question, but when we computed MFCC we actually converted the audio into sort of image like data.

```
X_train, X_test, y_train, y_test = get_train_test()

print(X_train.shape)
#(3112, 20, 11)
```

First number in the array shows the number of rows or audio samples to be precise. For each audio file the vector we get are 20 by 11 dimensional matrix. Yeah you guessed right! We can treat it just like an image. When we embed things it actually don't matter what kind of data we are dealing with. Since everything gets converted into vector.

### Wait again!

Yes we need to reshape it to give a depth. Just imagine you are working with an image which has a single channel only. Reshaping yields.

```
X_train = X_train.reshape(X_train.shape[0], 20, 11, 1)
print(X_train.shape)
# (3112, 20, 11, 1)
```

Now we are ready to feed it into CNN.

### One last thing!

So much hindrance! Sorry for that, train test split output actually gives the output tag like this [0, 1, 2] which each number means separate class. If you remember your machine learning theories, we need to encode this output vector into one-hot-encoded one to perform softmax .

Thanks to Keras we can just use the built in function,

```
y_train_hot = to_categorical(y_train)
y_test_hot = to_categorical(y_test)
```

That's it!

## Finally Building the CNN model



Instead of putting too much thought in model architecture, I am going to snatch an established one [I know I am lazy] and change a few things. This is the model I am going to use,

fchollet/keras

keras - Deep Learning library for Python. Runs on TensorFlow, Theano, or CNTK.

[github.com](https://github.com/fchollet/keras)



```
from preprocess import *
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten,
Conv2D, MaxPooling2D
from keras.utils import to_categorical

X_train, X_test, y_train, y_test = get_train_test()

X_train = X_train.reshape(X_train.shape[0], 20, 11, 1)
X_test = X_test.reshape(X_test.shape[0], 20, 11, 1)
y_train_hot = to_categorical(y_train)
y_test_hot = to_categorical(y_test)

model = Sequential()
model.add(Conv2D(32, kernel_size=(2, 2),
activation='relu', input_shape=(20, 11, 1)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.25))
model.add(Dense(3, activation='softmax'))
model.compile(loss=keras.losses.categorical_crossentropy,
optimizer=keras.optimizers.Adadelta(),
metrics=['accuracy'])

model.fit(X_train, y_train_hot, batch_size=100,
epochs=200, verbose=1, validation_data=(X_test,
y_test_hot))
```

## Output

```

....
Epoch 197/200
3112/3112 [=====] - 0s
100us/step - loss: 0.0598 - acc: 0.9788 - val_loss:
0.2071 - val_acc: 0.9398
Epoch 198/200
3112/3112 [=====] - 0s
98us/step - loss: 0.0619 - acc: 0.9762 - val_loss:
0.2212 - val_acc: 0.9398
Epoch 199/200
3112/3112 [=====] - 0s
97us/step - loss: 0.0596 - acc: 0.9759 - val_loss:
0.2102 - val_acc: 0.9422
Epoch 200/200
3112/3112 [=====] - 0s
98us/step - loss: 0.0599 - acc: 0.9785 - val_loss:
0.2158 - val_acc: 0.9441

```

Nice accuracy!

## How to infer using the model?

1. Create MFCC from an audio file
2. Feed the vector to the model using the function `predict`
3. Perform `argmax` to find the index of the label

Here

```

# Getting the MFCC
sample =
wav2mfcc('./data/happy/012c8314_nohash_0.wav')

# We need to reshape it remember?
sample_resaped = sample.reshape(1, 20, 11, 1)

# Perform forward pass
print(get_labels()[0][
    np.argmax(model.predict(sample_resaped))
])
# Output: 'happy'

```

I know it looks ugly, but works though!

## What's next?

- Apply CNN-LSTM to see how it works
- Increase dimension of MFCC vectors
- Don't downsample
- Include Batch Normalization Layer
- Add more classes!

There are a lot of things to do. Here is the project [repository](#). Thanks for reading!

