 Speech Recognition Engine with Keras and Python

# Approach

Speech recognition is the ability of a machine or program to identify words and phrases in spoken language and convert them to a machine-readable format. Usually, simple implementation of these algorithms has a limited vocabulary, and it may only identify words/phrases if they are spoken very clearly.

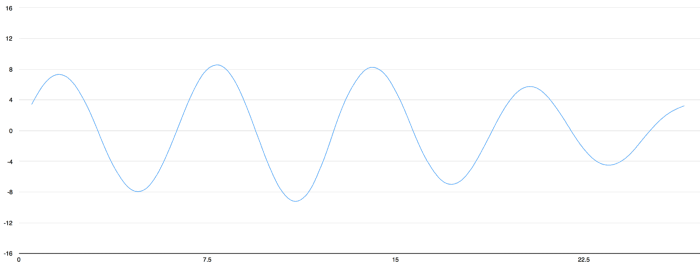
In this context, I will report how to build a speech recognition model to recognize short commands. Best of all, developing and including speech recognition in a Python project using Keras

* How speech to text works
* How to process audio to be transcribed
* A deep learning model using Keras to solve this challenge
* One way to evaluate this model
* A script to integrate the predictive model in your project

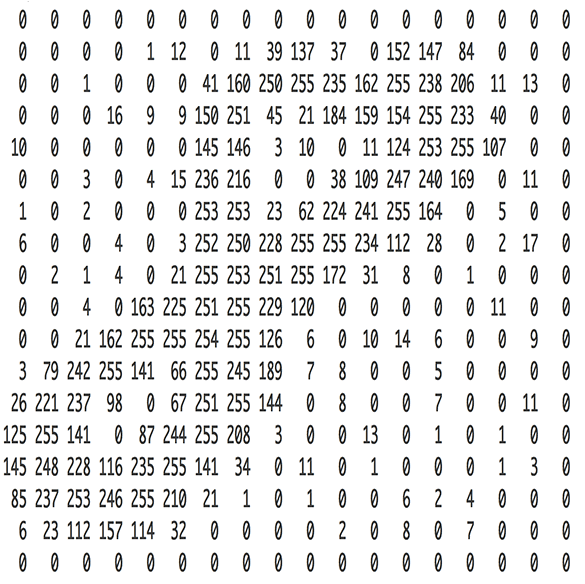
Speech recognition works using algorithms through acoustic and language modeling. Acoustic modeling represents the relationship between linguistic units of speech and audio signals; language modeling matches sounds with word sequences to help distinguish between words that sound similar. Often, deep learning models based on recurrent layers are used to recognize temporal patterns in speech to improve accuracy within the system

# Signal Processing:

There are many ways to transform an audio wave into elements that an algorithm can process, one of these ways, and the one that we will use in this Model,

[](file:///C:\Users\Hamza%20pc\Downloads\0_Vd_5HdXrCO8Asppw.gif)

We are reading thousands of times per second and recording a number that represents the height of the sound wave at that moment. Basically, it is an uncompressed .wav audio file. The “CD Quality” audio is sampled at 44.1 kHz (44,100 readings per second). But for speech recognition, a sampling rate of 16 kHz (16,000 samples per second) is sufficient to cover the frequency range of human speech.



# 1. Dataset

We used in our experiments the Speech Commands Datasets provided by TensorFlow. It includes 65,000 one-second long utterances of 30 short words, by thousands of different people. We’ll build a speech recognition system that understands simple spoken commands. You can download the Dataset from here: https://www.kaggle.com/c/tensorflow-speech-recognition-challenge/data

# 2. Preprocessing the audio waves

In the used dataset, the duration of a few recordings is less than 1 second and the sampling rate is too high. So, let us read the audio waves and use the preprocessing steps to deal with this.

* Resampling
* Removing shorter commands of less than 1 second

 We can understand that the sampling rate of the signal is 16000 Hz. Let us resample it to 8000 Hz since most of the speech related frequencies are present in 8000 Hz.

The last step of pre-processing step is reshape the 2D array to 3D since the input to the conv1d must be a 3D array:

# 3. Create train and validation set

In order to perform our deep learning model, we will need generate two sets (train and validate). For this experiment, I train the model with 80% of the data and validate on the remaining 20%:

# 4. Model Architecture

I used Conv1d and GRU layers to model the network that is used for speech recognition. Conv1d is a convolutional neural network which performs the convolution along only one dimension and GRU (Gated Recurrent Unit) aims to solve the vanishing gradient problem which comes with a standard recurrent neural network. GRU can also be considered as a variation on the LSTM because both are designed similarly and, in some cases, produce equally excellent results.

Batch normalization is a type of layer (BatchNormalization in Keras) introduced

in 2015 by Ioffe and Szegedy;7 it can adaptively normalize data even as the mean and

Variance change over time during training. It works by internally maintaining an exponential

moving average of the batch-wise mean and variance of the data seen during

training. The main effect of batch normalization is that it helps with gradient propagation—much like residual connections—and thus allows for deeper networks

The BatchNormalization layer takes an axis argument, which specifies the feature

Axis that should be normalized.

## Implementing a 1D convnet

In Keras, you use a 1D convnet via the Conv1D layer, which has an interface similar to

Conv2D. It takes as input 3D tensors with shape (samples, time, features) and

Returns similarly shaped 3D tensors. The convolution window is a 1D window on the

temporal axis: axis 1 in the input tensor.

Conv1D and MaxPooling1D layers, ending in

either a global pooling layer or a Flatten layer, that turns the 3D outputs into 2D outputs,

allowing you to add one or more Dense layers to the model for classification or

Regression.

### Arguments

**Pool\_size** Integer, size of the max pooling window.

**strides** Integer, or None. Specifies how much the pooling window moves for each pooling step. If None, it will default to pool\_size.

**Padding** One of "valid" or "same" (case-insensitive). "Valid" adds no padding. "Same" adds padding such that if the stride is 1, the output shape is the same as the input shape.

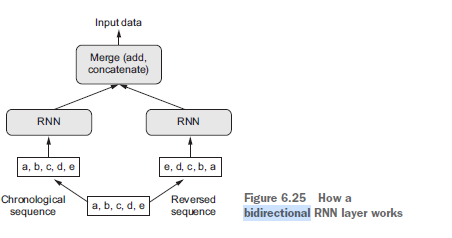
**Data\_format** A string, one of channels\_last (default) or channels\_first. The ordering of the dimensions in the inputs. Channels\_last corresponds to inputs with shape (batch, steps, features) while channels\_first corresponds to inputs with shape (batch, features, steps).

**Dropout** is a technique where randomly selected neurons are ignored during training. They are “dropped-out” randomly. This means that their contribution to the activation of downstream neurons is temporally removed on the forward pass and any weight updates are not applied to the neuron on the backward pass.

A bidirectional RNN exploits this idea to improve on the performance of chronologicalorder

RNNs. It looks at its input sequence both ways (see figure 6.25), obtaining potentially

richer representations and capturing



**Bidirectional** creates a second, separate

instance of this recurrent layer and uses one instance for processing the input sequences

in chronological order and the other instance for processing the input sequences in

reversed order.

## GRU layers

SimpleRNN isn’t the only recurrent layer available in Keras. There are two others: LSTM

and GRU. In practice, you’ll always use one of these, because SimpleRNN is generally too simplistic to be of real use. SimpleRNN has a major issue: although it should theoretically be able to retain at time t information about inputs seen many timesteps before, in practice, such long-term dependencies are impossible to learn. This is due to the *vanishing* *gradient problem*, an effect that is similar to what is observed with non-recurrent networks (feedforward networks) that are many layers deep: as you keep adding layers to a network, the network eventually becomes untrainable. The theoretical reasons for this effect were studied by Hochreiter, Schmidhuber, and Bengio in the early 1990s.2 The LSTM and GRU layers are designed to solve this problem.

## Flatten layer

Flattening a tensor means to remove all of the dimensions except for one. This is exactly what the Flatten layer do.

Dense layer are connected to every other unit. The layer attempts to map relationships between any two input features; this is unlike a 2D convolution layer, for instance, which only looks at *local* relationships. Densely connected networks are most commonly used for categorical data.