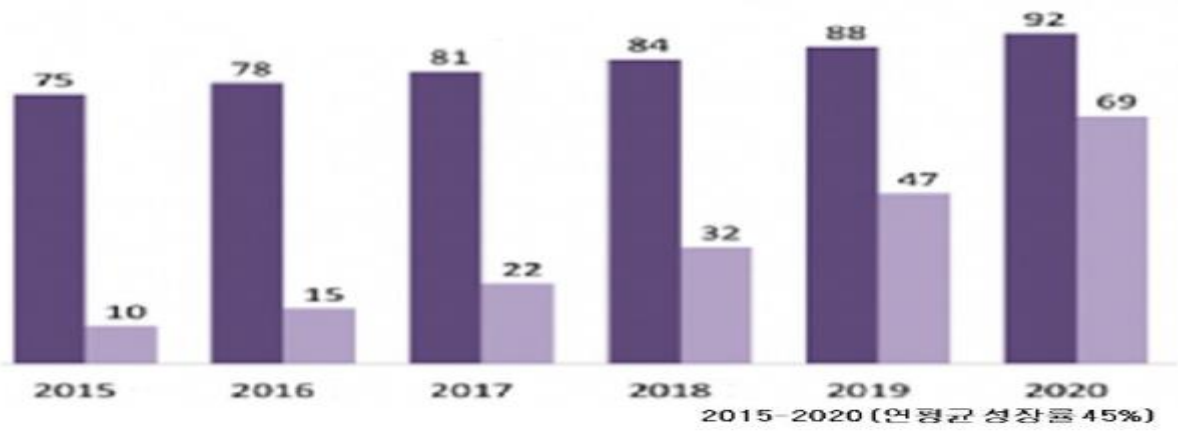
시나리오3: 자율주행 봇

2016 빅데이터 교육 콘텐츠

Contents

I 비즈니스 문제 정의

1. 스마트 카



[출처] BI Intelligence, 전 세계 자동차 판매량 대비 스마트 카 연간 판매량 전망

최근에 자동차 업계에서는 지능 정보 기술이 결합된 스마트 카에 대한 관심이 높아지고 이를 적용하기 위한 기술 테스트가 많이 이뤄지고 있다. 위 그래프와 같이 스마트 카에 대한 수요는 점차적으로 증가하여 **2020년까지 연평균 45%의 성장률**을 기록할 것으로 전망된다.

자동차 시장의 핫 이슈 스마트 카란 무엇일까?

스마트 카란 자동차기술에 차세대 전기전자, 정보통신, 기능제어 기술을 접목하여 자동차의 내 외부 상황을 실시간으로 인식하고 이를 기반으로 **고안전, 고편의 기능을 제공할 수 있는 인간 친화적 자동차**이다.

[출처] 한국산업기술진흥원(KIAT), 스마트 카 정의

따라서 각종 센서, 통신망 등을 이용하여 차량 이용자에게 **고안전, 고편의 기능을 제공**하는 것이 스마트 카의 주요 기술이 되고 해당 기술의 범위는 다음과 같다.

[출처] 한국산업기술진흥원(KIAT), 스마트 카 기술 범위

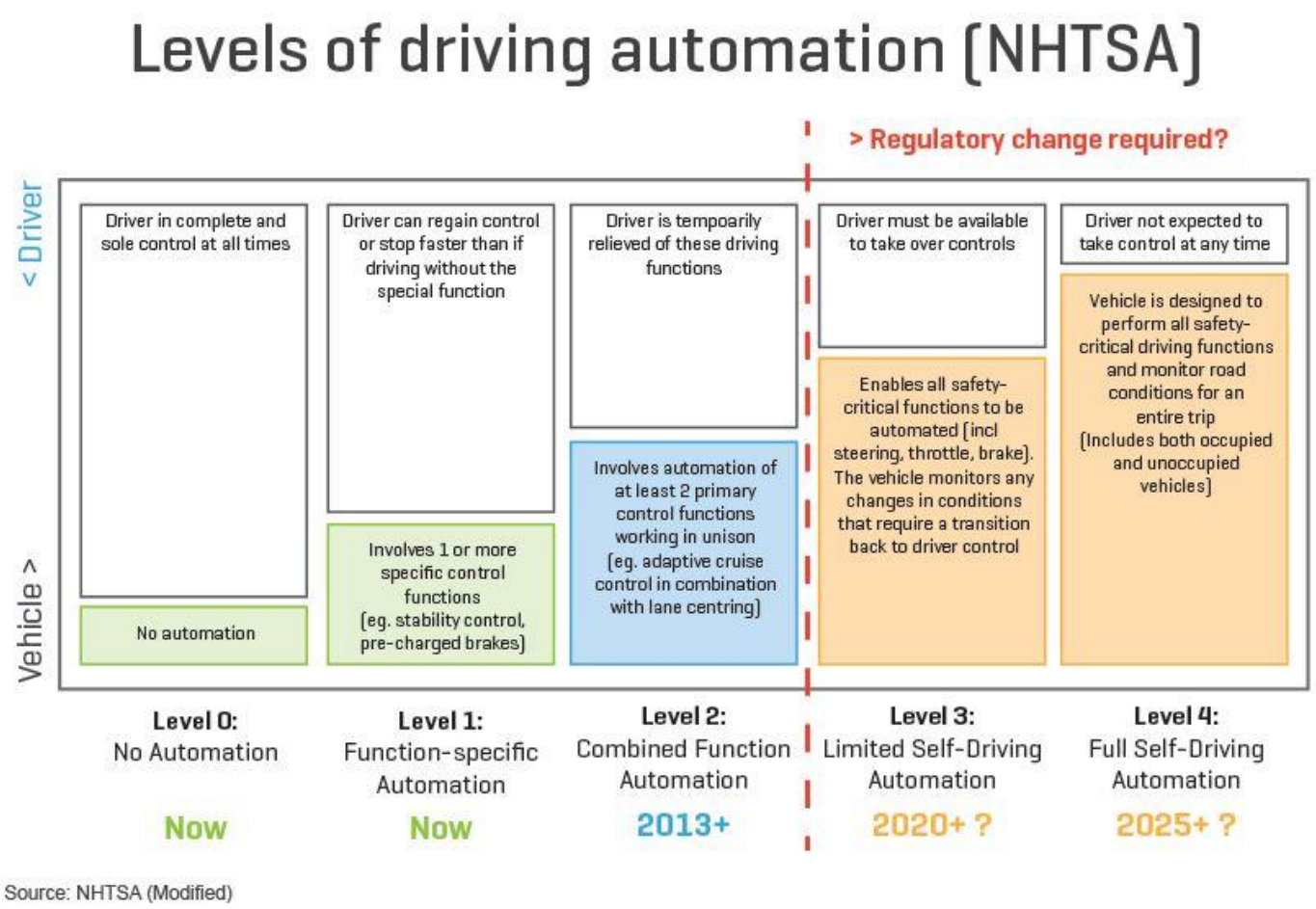
위 그림에서는 스마트 카를 구성하는 수 많은 기술들을 소개하고 있다. 그 중에서도 자동차 업계에서 가장 주목하고 있는 기술은 바로 **AEB(Automatic Emergency Braking)과 무인 자동차 기술**이다. 실제 IT 업계의 최대 플랫폼 사업자인 구글과 애플 역시 스마트 카에 주목하고 있으며 특히 구글의 경우 **완전 자율 주행 자동차를 위한 기술 개발**에 박차를 가하고 있다.

차세대 자동차 산업을 이끌어 갈 스마트 카, 그리고 이러한 스마트 카 시장에서 가장 주목받는 기술 중 하나인 **완전 자율 주행 자동차**에 대해 조금 더 상세히 살펴보자

2. 자율 주행 자동차

자동차 산업의 차세대 먹거리 자율 주행 자동차

자율 주행 자동차란 운전자가 차량을 조작하지 않아도 스스로 주행하는 자동차로 자동차 산업의 차세대 먹거리로 주목 받고 있는 기술이다. 아래의 그림과 같이 미국도로교통안전국(NHTSA)에 따르면 자율주행 자동차 기술은 크게 4단계로 분류되며 현재는 2-3단계에 속한다고 한다.



[그림 1] 자율주행자동차 기술 단계 흐름도

이러한 자율 주행 자동차 개발을 위한 요구 사항 및 기대효과는 다음과 같다.

[요구 사항]

데이터 측면

지도 데이터, 내비게이션 데이터 등을 바탕으로 자율 주행 자동차를 지속적으로 개선한다.

기술적 측면

진보된 고성능 카메라, 충돌 방지 장치 등이 필요하다. 또한 주행 상황 정보를 종합 판단하여 처리할 수 있는 플랫폼 기반의 데이터 처리 기술을 바탕으로 한 **주행 상황 인지, 대응 기술 등 필요**하다

환경 측면

자율 주행 자동차 관련 R&D 활성화를 위해 국가적인 차원에서 적극적인 투자와 실효성을 갖춘 기술 로드맵을 수집 및 검토하여 정책 전략이 제공돼야 한다.

[기대 효과]

자율 주행 자동차는 2020년에 100% 자율 주행 차량이 도입됨에 따라 2030년에 950억달러 규모가 될 것으로 예상 된다. 아래는 이에 따른 기대효과를 경제적 편익이 큰 순으로 나타내었다.

기대 효과	경제적 편익
자유시간 발생에 따른 생산성 향상	5070억 달러
교통사고 발생 감소에 따른 비용 감소	4880억 달러
대체에너지원 활용에 따른 연료 절감	1580억 달러
교통체증 감소에 따른 생산성 향상	1280억 달러
교통체증 감소에 따른 연료 절감	110억 달러

[표 1] 자율 주행 자동차 도입에 따른 경제적 편익

위와 같이 충분한 기대 효과가 예상되는 자율 주행 자동차, 직접 구현해 볼 수 없을까?

정답은 '가능하다'이다. 위에서 언급한 자율 주행 자동차의 요구 사항을 살펴보면 데이터 측면은 이미 갖춰져 있고 환경 측면의 경우는 정부의 정책과 관련된 사항이다. 즉 기술적 측면만 해결한다면 실제 자율 주행이 가능하다는 것을 뜻한다.

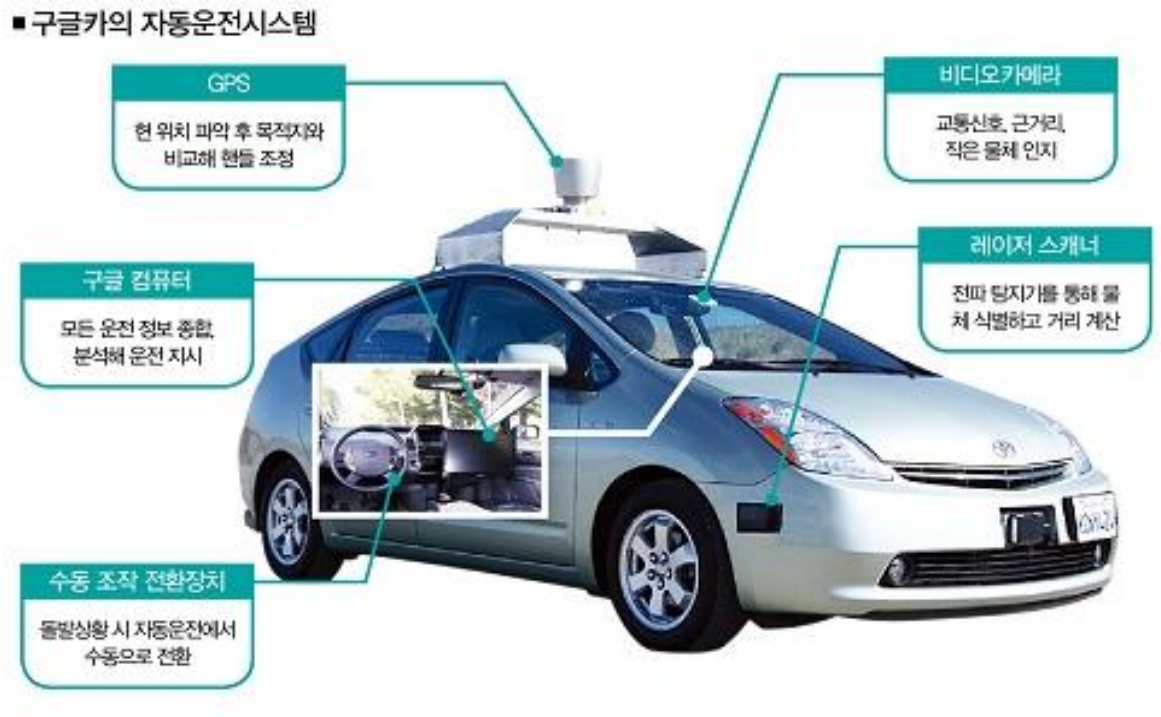
여기서 기술적 측면이란 데이터 처리 기술을 바탕으로 한 **'주행 사항 인지 및 대응 기술 등'**으로 **딥 러닝으로 충분히 구현 가능한 부분**이다. 따라서 본 실습에서는 자율 주행 자동차를 위한 핵심 요소 기술들을 직접 구현하고 학습하는 것을 그 목적으로 한다.

II 비즈니스 문제 해결 방안

1. 자율 주행을 위한 요구 기술

자율 주행 자동차 구현을 위해 요구되는 기술들은 무엇일까?

자율 주행 자동차를 구현하기 위해선 다음과 같은 기술들이 요구된다.



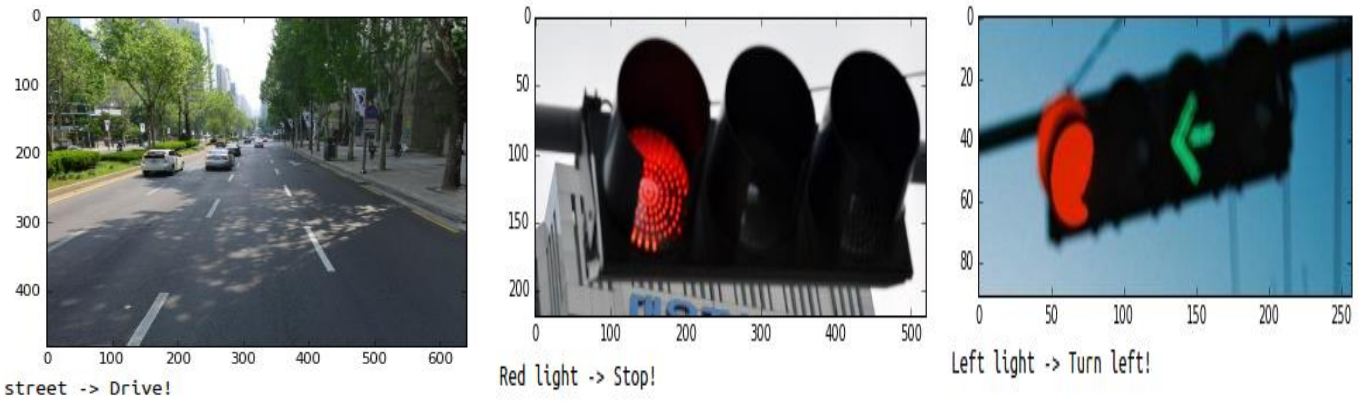
[그림 2] 자율 주행에 필요한 기술 요소의 예

위의 기술들을 충족하기 위해서는 **지능 정보 기반의 Vision 및 Sound 인식** 그리고 GPS 기반의 위치정보 인식 등 많은 기술들이 필요하다. 하지만 본 실습에서는 가장 핵심이 되는 기술인 **지능 정보 기반의 Vision 및 Sound 인식**에 초점을 맞추고자 한다. 즉 카메라와 마이크를 통해 입력되는 **주행 영상(Vision) 및 사용자의 목소리(Sound)를 딥 러닝을 통해 인지**하고 이를 기반으로 자율 주행이 진행되는 '자율 주행 시나리오'를 구현하도록 하겠다.

2. 문제 해결 과정

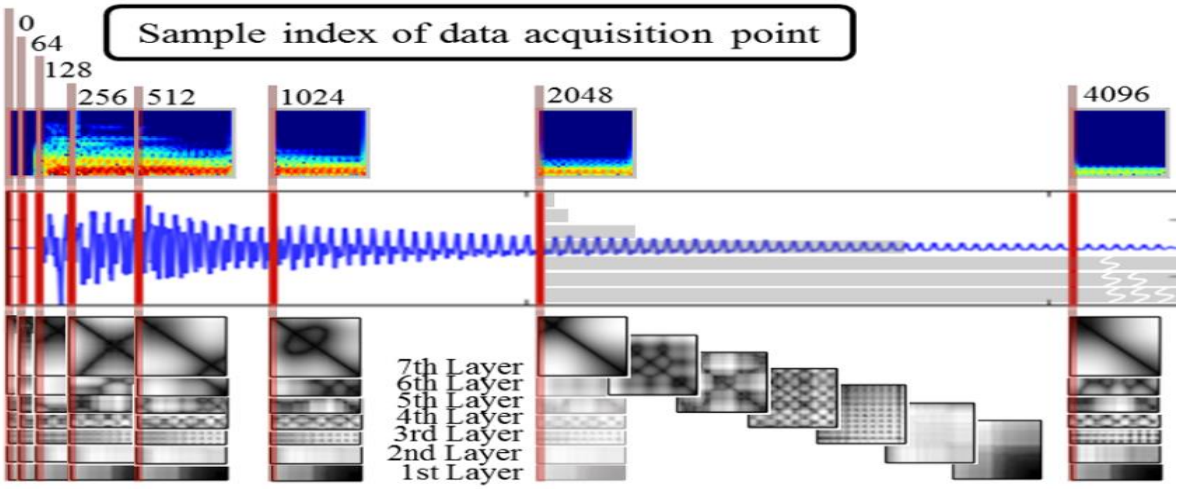
딥 러닝을 통한 Vision(주행 영상) 및 Sound(사용자 목소리) 인식은 어떻게 이루어질까?

데이터 종류	그 목적과 방법
Vision	**[목적]** 자율 주행을 위한 영상 정보 인식 **[방법]** 영상 데이터를 이미지 프레임으로 분할하고 CNN을 적용하여 타겟 이미지를 판별하는 인식 모델 구현



[그림 1] 주행 영상(이미지) 인식 모델 예

Sound	**[목적]** 자율 주행을 위한 음성 정보 인식 **[방법]** 음성 데이터를 프레임 단위로 분할하고 MFCC 및 CNN을 적용하여 타겟 음성을 판별하는 인식 모델 구현



[그림 4] 음성 처리의 예

[분석 과정]

[분석 환경]

필요 환경

- 운영체제: Ubuntu 14.04 LTS 이상 (아래 코드는 Windows에서도 테스트 완료)
- 텐서플로우 버전: r0.11 이상 (r.0.12 에서도 테스트 완료)
- 파이썬 버전: 3.5.x

[분석 기법]

필수 이론

- 딥러닝 알고리즘 CNN
- 음성 처리 알고리즘 Mel Frequency Cepstrum Coefficient
- 위의 이론들은 해당 컨텐츠를 수행하기 앞서 '딥 러닝 기본 이론' 교재를 통해 선행되어야 함.

III 지능 정보를 통한 문제 해결

본 실습에서는 자율주행 자동차의 기초 기술이 되는 음성 및 영상 인식 모델을 구현하는 것을 목표로 한다. 자동차에 해당 모델이 탑재되었다는 가정 하에 자율 주행 시나리오 상에서 음성 및 영상 인식이 적용되는 과정을 상세히 설명한다.

본 실습에서 가정하는 자율 주행 시나리오는 다음과 같다.

 # 자율 주행 시나리오

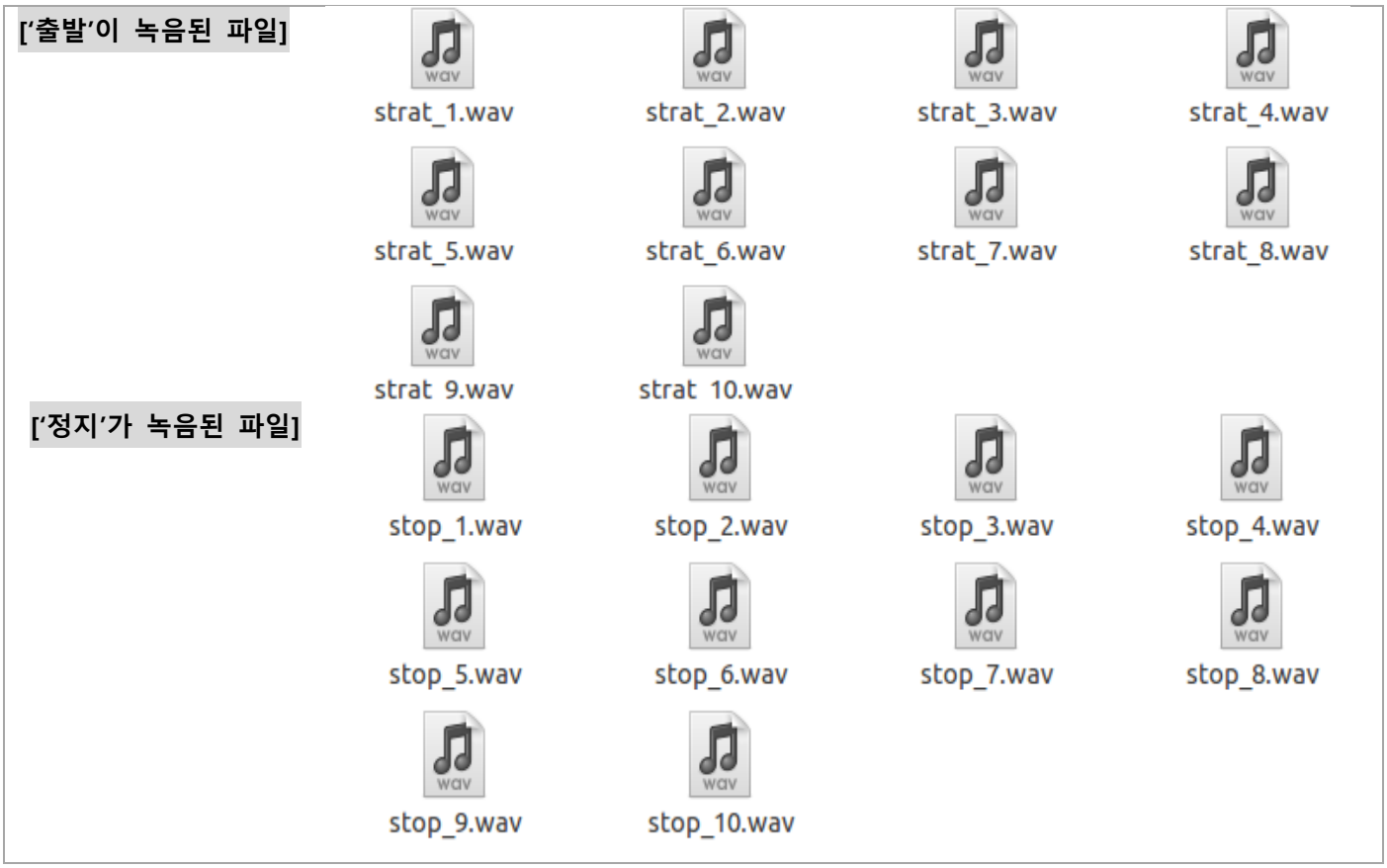
① 시동어 '출발'을 인식하여 자동차의 시동이 켜진다.
② 시동이 켜진 후, 카메라로 입력되는 영상을 Frame 단위 이미지로 인식하여 'Go straight', 'Stop', 'Turn left' 'Destination' 여부를 판단한다.
③ 목적지에 도착 후 시동어 '정지'를 인식하여 자동차의 시동을 끈다.

따라서 본 실습은 음성 인식과 영상 인식을 나누어서 구현하도록 하겠다.

1. 음성 인식

음성 인식을 위한 데이터 셋은 다음과 같이 시동어 '출발'과 '정지' 그리고 '잡음' 및 '테스트 음성'으로 구분하여 직접 녹음하고 WAV파일로 저장한다.

시동어 '출발'과 '정지'는 아래와 같이 10개의 WAV 파일로 각 파일은 약 1초의 길이를 가진다. 또한 '잡음'와 '테스트 음성'의 경우 각 하나의 WAV 파일로 구성되어 있으며 약 60초의 길이를 가진다.



[그림 5] 음성 인식을 위한 WAV 파일

실행 코드

```
# 라이브러리를 생성합니다.
from pylab import*
import pyaudio
import numpy as np
import scipy.signal
import scipy.fftpack
```

```
from scipy.io import wavfile as wvf
from scipy.fftpack import dct
import os
import decimal
```

```
# 데이터 처리를 위한 기본 라이브러리인 numpy 그리고 신경망 분석을 위한 tensorflow를 불러온다.
# 음성 데이터 전 처리를 위한 라이브러리들(pyaudio, scipy.signal, scipy.io)을 불러온다.
# MFCC 적용을 위한 라이브러리(scipy.fftpack) 역시 불러온다.
# 지정한 경로 안에 있는 파일의 목록을 불러오기 위한 라이브러리인 os를 불러온다.
# 값을 반올림하는데 필요한 라이브러리인 decimal을 불러온다.
```

위와 같이 필요 라이브러리를 불러온 후 **각 WAV파일을 Array형식으로 변환** 후 저장하도록 한다.

실행 코드

```
# 음성 데이터를 불러온 후 Array형식으로 변환합니다.
wav_dir = '/home/eduuser/NN/Data/Voice/'
wav_labels = os.listdir(wav_dir)
x_start = []
x_stop = []
x_noise = []
x_test = []
time_cut = 0.5

for wav_label in wav_labels:
   for wav_f in os.listdir(wav_dir+wav_label):
            if wav_label == 'start':
                SampFreq, f = wvf.read(wav_dir+wav_label+'/' + wav_f)
                N_cut = int(SampFreq*time_cut)
                audio_data = f[:N_cut]
                audio_data = audio_data[:,0]
                x_start.append(audio_data)
            elif wav_label == 'stop':
                SampFreq, f = wvf.read(wav_dir+wav_label+'/' + wav_f)
                N_cut = int(SampFreq*time_cut)
                audio_data = f[:N_cut]
                audio_data = audio_data[:,0]
                x_stop.append(audio_data)
            elif wav_label == 'noise':
                SampFreq, f = wvf.read(wav_dir+wav_label+'/' + wav_f)
                audio_data = f
                audio_data = audio_data[:,0]
                x_noise.append(audio_data)
            else:
                SampFreq, f = wvf.read(wav_dir+wav_label+'/' + wav_f)
                audio_data = f
                audio_data = audio_data[:,0]
                x_test.append(audio_data)
```

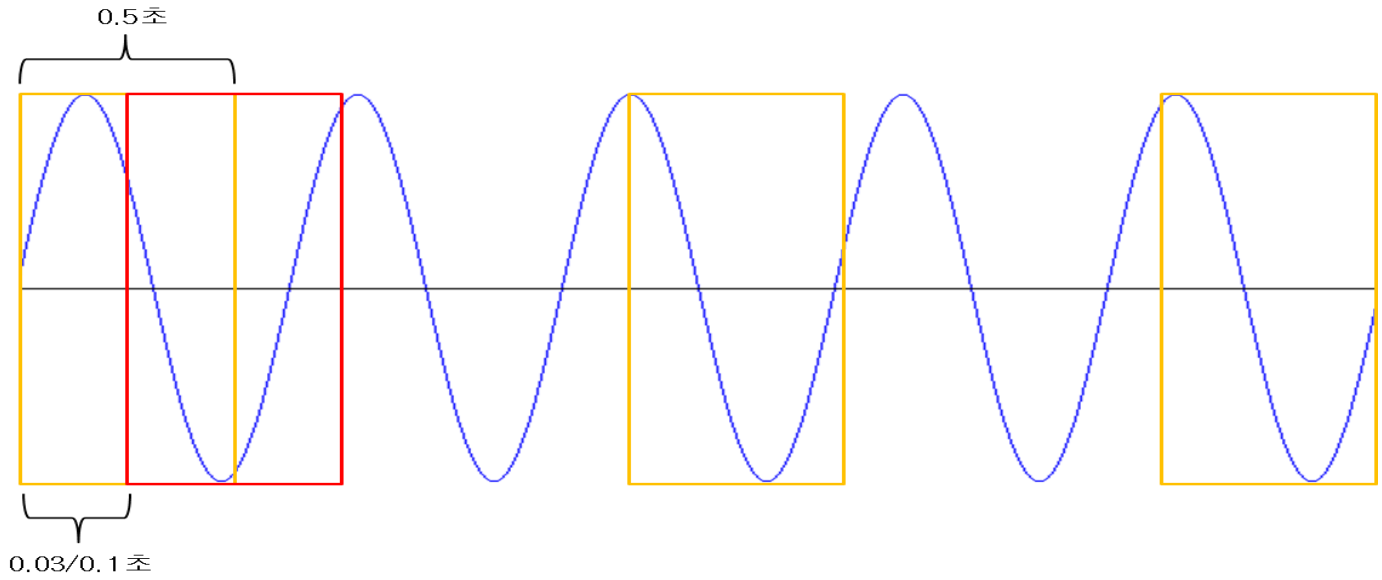
```
# 보통 '출발' 및 '정지'와 같은 두 글자로 이루어진 단어는 음성의 길이가 약 0.5초 이하이다. 따라서 시동어 '출발' 및 '정지'로 이루어진 WAV파일을 Array형태로 변환 시 그 길이(Time_cut)를 0.5로 지정하였다.
```

[음성 프레임 생성]

실행 코드

```
# 노이즈 음성을 프레임 단위로 나눕니다.
frame_len = N_cut
step_len = SampFreq*0.03
num_frame = int(ceil((len(x_noise[0]) - frame_len)/step_len))
x_noise_frame = []

for i in arange(num_frame):
    tmp = np.array(x_noise)
    sta = int(i*step_len)
    end = sta + int(frame_len)
    frame = tmp[0][sta:end]
    x_noise_frame.append(frame)

# 테스트 음성을 프레임 단위로 나눕니다.
frame_len = N_cut
step_len = SampFreq*0.1
num_frame = int(ceil((len(x_test[0]) - frame_len)/step_len))
x_test_frame = []

for i in arange(num_frame):
    tmp = np.array(x_test)
    sta = int(i*step_len)
    end = sta + int(frame_len)
    frame = tmp[0][sta:end]
    x_test_frame.append(frame)
```

WAV 파일 'noise'와 'test'는 약 60 초 정도로 이루어진 음성 파일이다. 이를 모델에 적용하기 위해서는 해당 음성을 **0.5 초로 이루어진 프레임 단위**로 나누어야 한다. 따라서 본 실습에서는 'noise'의 경우 0.03 초, 그리고 'test'의 경우 0.1 초 단위로 윈도(Window)를 움직여 0.5 초의 길이를 갖는 음성 프레임을 생성하였다.



[그림 6] 음성 Frame 생성

[노이즈 추가]

실행 코드

```
# 시동어 음성에 노이즈를 추가합니다.
start_noise = []
stop_noise = []
num_rand = 100
max_noise = 500

for i in arange(len(x_start)):
    true = x_start[i]
    for j in arange(num_rand):
        add = true + np.random.randn(N_cut)*max_noise
        start_noise.append(add)

for i in arange(len(x_stop)):
    true = x_stop[i]
    for j in arange(num_rand):
        add = true + np.random.randn(N_cut)*max_noise
        stop_noise.append(add)
```

```
# 각 시동어에 num_rand 만큼의 Random Noise를 생성하여 추가함.
```

시동어인 '출발'과 '정지'는 각 10개의 wav파일로 이루어져있다. 이는 모델을 학습시키기에는 매우 부족한 양이다. 또한 '출발'과 '정지'라는 단어가 실제로 주변 소음이 거의 없는 상태로 녹음이 된다는 보장이 없으므로 본 실습에서는 정규분포를 기반으로 Random Noise를 생성하여 **각 시동어 데이터에 노이즈를 추가**하였다.

이렇게 Random Noise 가 추가된 데이터와 Frame 단위로 나누어진 noise 를 결합하여 학습데이터를 생성한다.

실행 코드

```
# 모델링을 위한 입력 데이터 및 라벨 데이터를 생성합니다.
num_start = len(start_noise) + len(x_start)
num_stop = len(stop_noise) + len(x_stop)
num_noise = len(x_noise_frame)

x_train = []
x_train.extend(start_noise)
x_train.extend(x_start)
x_train.extend(stop_noise)
x_train.extend(x_stop)
x_train.extend(x_noise_frame)

y_start = [[1,0,0]]*num_start
y_stop = [[0,1,0]]*num_stop
y_noise = [[0,0,1]]*num_noise

y_train = []
y_train.extend(y_start)
y_train.extend(y_stop)
y_train.extend(y_noise)

voice = np.array(x_train) / (2.**15)
label = np.array(y_train)
```

```
# 모델 구축을 위해 '출발', '정지', 'Noise'에 각각 [1,0,0] [0,1,0] [0,0,1]의 라벨을 지정한다.
```

[Mel-Frequency Cepstrum Coefficient]

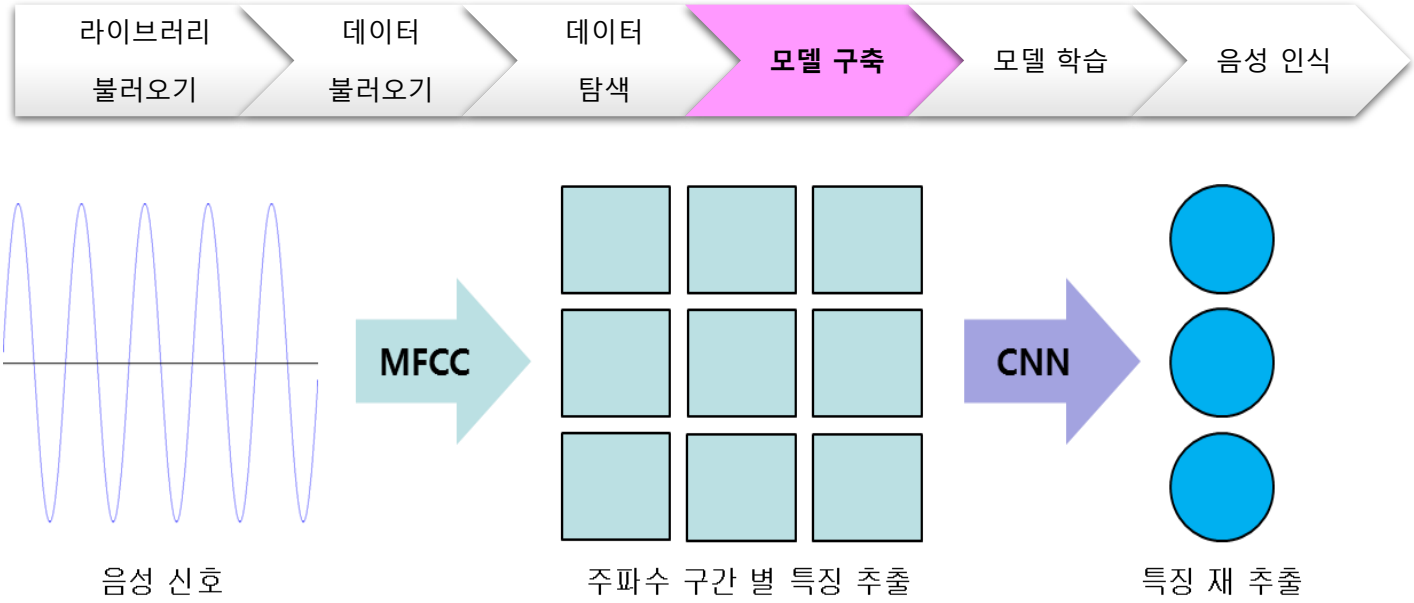
음성 인식을 위해서는 기본적으로 Time(시간) Domain으로 되어있는 각 데이터를 Frequency(주파수) Domain으로 변환 후 그 특징 값을 추출하는 것이 필요하다. 이렇게 추출 된 특징 값을 MFCC이라 부른다. 해당 값을 추출하는 코드는 아래와 같다.

실행 코드

```
# MFCC 값을 추출합니다.
from python_speech_features import mfcc
from python_speech_features import logfbank

def round_half_up(number):
    return int(decimal.Decimal(number).quantize(decimal.Decimal('1'), rounding=decimal.ROUND_HALF_UP))

samplerate = SampFreq
winlen = 0.02
winstep = 0.01
numcep = 15
nfilt = 23
frame_len = int(round_half_up(winlen * samplerate))
nfft = frame_len
lowfreq = 0
highfreq = None
preemph = 0.97
ceplifter = 22
appendEnergy = True
winfunc= lambda x:np.hamming(x)
num_channel = 1
cnn_feat = []

def normalize(a):
    b = (a-np.mean(a))/np.std(a)
    return b

for i in arange(len(voice)):
    mfcc_feat = mfcc(voice[i],samplerate, winlen, winstep,
                     numcep, nfilt, nfft, lowfreq, highfreq, preemph, ceplifter, appendEnergy,
                     winfunc)
    print("step %d"%(i))
    mfcc_feat = np.apply_along_axis(normalize,1,mfcc_feat)
    feat_flat = mfcc_feat.flatten()

    N = len(feat_flat)
    N_root = np.sqrt(N)
    N_root = int(np.ceil(N_root))
    N_square = N_root**2
    N_add = int(N_square - N)
    zeros = np.zeros(N_add)
    feat_zeros = np.append(feat_flat, zeros)
    feat_zeros = np.reshape(feat_zeros, (N_root,N_root,num_channel))
    cnn_feat.append(feat_zeros)

voice_feat = np.array(cnn_feat)
```

winlen, winstep, numcep 등은 MFCC를 추출하기 위한 파라미터 값이다. 해당 파라미터 및 MFCC에 대한 상세한 설명은 'DNN 이론 교재의 음성인식' 부분을 참고하기 바란다.
일반적으로 CNN 모델 구축 시 데이터의 개수가 정수의 제곱 값이 되는 것이 기술적인 측면에서 편리하다. 따라서 np.zeros 와 np.append 메서드를 활용하여 정수의 제곱 값이 되도록 0값을 추가한다.

해당 과정을 거치게 되면 각 음성 데이터 당 49개의 구간 그리고 구간 당 15개의 특징, 총 735개의 특징 값이 추출된다.



[그림 7] 음성 인식 모델 기본 구조

MFCC를 통해 추출된 음성데이터를 기반으로 시동어 '출발' 및 '정지'를 인식하는 모델을 구축한다. 일반적으로 각 음성 데이터의 MFCC 값 자체로도 모델 구축이 가능하지만(이는 딥러닝 이론 교재에 적용된 바이다.), 본 실습에서는 **사람의 목소리를 구분해야 하는 정교한 모델이 요구**되므로 추가 작업이 필요하다.

따라서 MFCC 값에서 **영역별로 특징을 재 추출하는 작업**이 필요하다. 이는 CNN을 통해 이루어진다.

[모델 파라미터 설정 및 배치 데이터 생성]

실행 코드

```
# CNN 파라미터 값을 지정합니다.
import tensorflow as tf

num_filter_1 = 16
num_filter_2 = 32
num_neuron_1 = 1024
num_neuron_2 = 256
num_classes = len(label[0,:])

x = tf.placeholder(tf.float32, [None, N_root, N_root, num_channel])
y_ = tf.placeholder(tf.float32, [None, num_classes])

# 배치데이터를 생성합니다.
batch_size = 100

def random_batch():
    num_voice = len(voice_feat)
    idx = np.random.choice(num_voice,
                           size=batch_size,
                           replace=False)
    x_batch = voice_feat[idx, :, :, :]
    y_batch = label[idx, :]
    return x_batch, y_batch
```

본 실습에서는 2개의 합성곱 계층과 2개의 은닉 계층(Fully-Connected Net)으로 이루어진 네트워크를 구성한다. 따라서 합성곱 계층에 필요한 필터의 수는 각 16개 그리고 32개로 지정하고 은닉 계층에 필요한 뉴런의 수는 1024개 그리고 256개로 지정한다. 또한 배치 데이터 생성을 위한 배치 사이즈는 100으로 지정한다.

[CNN 구축]

실행 코드

```
# CNN 모델을 정의합니다.
def weight_variable(shape):
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial)

def bias_variable(shape):
    initial = tf.constant(0,1, shape=shape)
    return tf.Variable(initial)

def conv2d(x,W):
    return tf.nn.conv2d(x,W, strides=[1,1,1,1], padding='SAME')

def max_pool_2X2(x):
    return tf.nn.max_pool(x, ksize=[1,2,2,1], strides=[1,2,2,1], padding='SAME')

# 첫번째 합성곱 계층
W_conv1 = weight_variable([5,5,1,num_filter_1])
b_conv1 = bias_variable([num_filter_1])

# ReLU 및 풀링 계층
h_conv1 = tf.nn.relu(conv2d(x, W_conv1) + b_conv1)
h_pool1 = max_pool_2X2(h_conv1)

# 두번째 합성곱 계층
W_conv2 = weight_variable([5,5,num_filter_1,num_filter_2])
b_conv2 = bias_variable([num_filter_2])

h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
h_pool2 = max_pool_2X2(h_conv2)

# Fully connected net 첫번째 계층
W_fc1 = weight_variable([7*7*num_filter_2,num_neuron_1])
b_fc1 = bias_variable([num_neuron_1])

h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*num_filter_2])
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)

# 첫번째 드랍아웃
keep_prob = tf.placeholder("float")
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

# Fully connected net 두번째 계층
W_fc2 = weight_variable([num_neuron_1,num_neuron_2])
b_fc2 = bias_variable([num_neuron_2])

h_fc2 = tf.nn.relu(tf.matmul(h_fc1_drop, W_fc2) + b_fc2)

# 두번째 드랍아웃
h_fc2_drop = tf.nn.dropout(h_fc2, keep_prob)

# Softmax 계층
W_fc3 = weight_variable([num_neuron_2,num_classes])
b_fc3 = bias_variable([num_classes])

y_conv=tf.nn.softmax(tf.matmul(h_fc2_drop, W_fc3) + b_fc3)
```

위의 CNN 구축 코딩의 경우 'CNN 이론 교재의 MNIST, Cifar-10 실습'과 유사하다. 따라서 별도의 설명은 생략하도록 하겠다.

실행 코드

```
# 학습을 위한 매개변수를 지정하고 모델을 학습합니다.
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y_conv), reduction_indices=[1]))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(y_conv,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
num_iter = 1000

sess = tf.Session()
sess.run(tf.initialize_all_variables())

for i in range(num_iter):
    x_batch, y_batch = random_batch()
    if i%100 == 0:
        train_accuracy = sess.run(accuracy, feed_dict={x: x_batch, y_: y_batch, keep_prob: 1.0})
        train_cost = np.exp(sess.run(cross_entropy, feed_dict={x: x_batch, y_: y_batch, keep_prob: 1.0}))
        print("step %d, training cost %g"%(i, train_cost))
    sess.run(train_step, feed_dict={x: x_batch, y_: y_batch, keep_prob: 0.5})
```

```
# 총 num_iter만큼 반복하여 학습을 진행한다.
```

출력 결과

```
step 0, training cost 5.74901
step 100, training cost 1.0107
step 200, training cost 1.01064
….
Step 900, training cost 1.00001
```

학습된 모델을 기반으로 실제 자율 주행 시나리오 기반 테스트 음성을 인식해 본다. 테스트 데이터는 약 60초의 길이를 가지는 음성 데이터로 중간에 본 실습의 시동어인 '출발'과 '정지'를 포함하고 있다. 따라서 이미 앞장에서 0.5초 단위로 나누어 놓은 음성 Frame 중 시동어 '출발'과 '정지'를 포함하는 Frame이 반드시 존재한다. 만약 모델이 잘 구축되었다면 두가지 **시동어를 포함한 음성 Frame을 인식**하여야 한다.

실행 코드

```
###### model assessment ######
test = np.array(x_test_frame) / (2.**15)
threshold = 0.7
for i in arange(len(test)):
    mfcc_feat = mfcc(test[i],samplerate, winlen, winstep,
                     numcep, nfilt, nfft, lowfreq, highfreq, preemph, ceplifter, appendEnergy, winfunc)
    mfcc_feat = np.apply_along_axis(normalize,1,mfcc_feat)
    feat_flat = mfcc_feat.flatten()

    N = len(feat_flat)
    N_root = np.sqrt(N)
    N_root = int(np.ceil(N_root))
    N_square = N_root**2
    N_add = int(N_square - N)
    zeros = np.zeros(N_add)
    feat_zeros = np.append(feat_flat, zeros)
    feat_zeros = np.reshape(feat_zeros, (N_root,N_root,num_channel))
    test_feat = np.array(feat_zeros)
    shape = [1]
    tmp = test_feat.shape
    shape.extend(tmp)
    test_feat = np.reshape(test_feat,shape)
    result = sess.run(y_conv, feed_dict={x: test_feat, keep_prob: 1.0})
    result_start = round(result[0][0],2)
    result_stop = round(result[0][1],2)
    result_none = round(result[0][2],2)
    if result_start > threshold:
        print("%d th sound, result %g, Start"%(i, result_start))
    elif result_stop > threshold:
        print("%d th sound, result %g, Stop"%(i, result_stop))
    else:
        print("%d th sound, None"%(i))

test_result = []
batchidx  = 0

for i in arange(num_batches_test):
    x_test = xbatches_test[batchidx]
    y_test = ybatches_test[batchidx]
    batchidx      = batchidx + 1
    tmp = sess.run(accuracy, feed_dict={input_data: x_test, targets: y_test})
    test_result.append(tmp)

Accuracy = mean(test_result) * 100
print ("Test Accuracy: %.4f" % (Accuracy))
```

```
# threshold를 기준으로 해당 값보다 '출발' 클래스의 확률 값이 큰 경우 '출발'로 인식한다('정지"에도 똑같이 적용).
```

출력 결과

```
142 th sound, None
143 th sound, None
144 th sound, None
145 th sound, None
146 th sound, None
147 th sound, None
148 th sound, None
149 th sound, result 0.91, Start
150 th sound, None
151 th sound, None
152 th sound, None
153 th sound, None
154 th sound, None
….
312 th sound, None
313 th sound, None
314 th sound, None
315 th sound, None
316 th sound, result 0.98, Stop
317 th sound, None
318 th sound, None
319 th sound, None
320 th sound, None
```

테스트 결과 **149번째 Frame과 316번째 Frame을 각각 '출발'과 '정지'로 인식**하였다. 해당 프레임이 정말 '출발'과 '정지'를 포함한 음성 Frame인지를 확인하기 위해 각각을 WAV파일로 변환하여 들어보도록 한다.

실행 코드

```
# 평가된 음성 데이터를 확인합니다.
sample = test[149]
out_f = '/home/eduuser/NN/Data/Voice/start.wav'
wvf.write(out_f, SampFreq, sample)


sample = test[316]
out_f = '/home/eduuser/NN/Data/Voice/stop.wav'
wvf.write(out_f, SampFreq, sample)
```

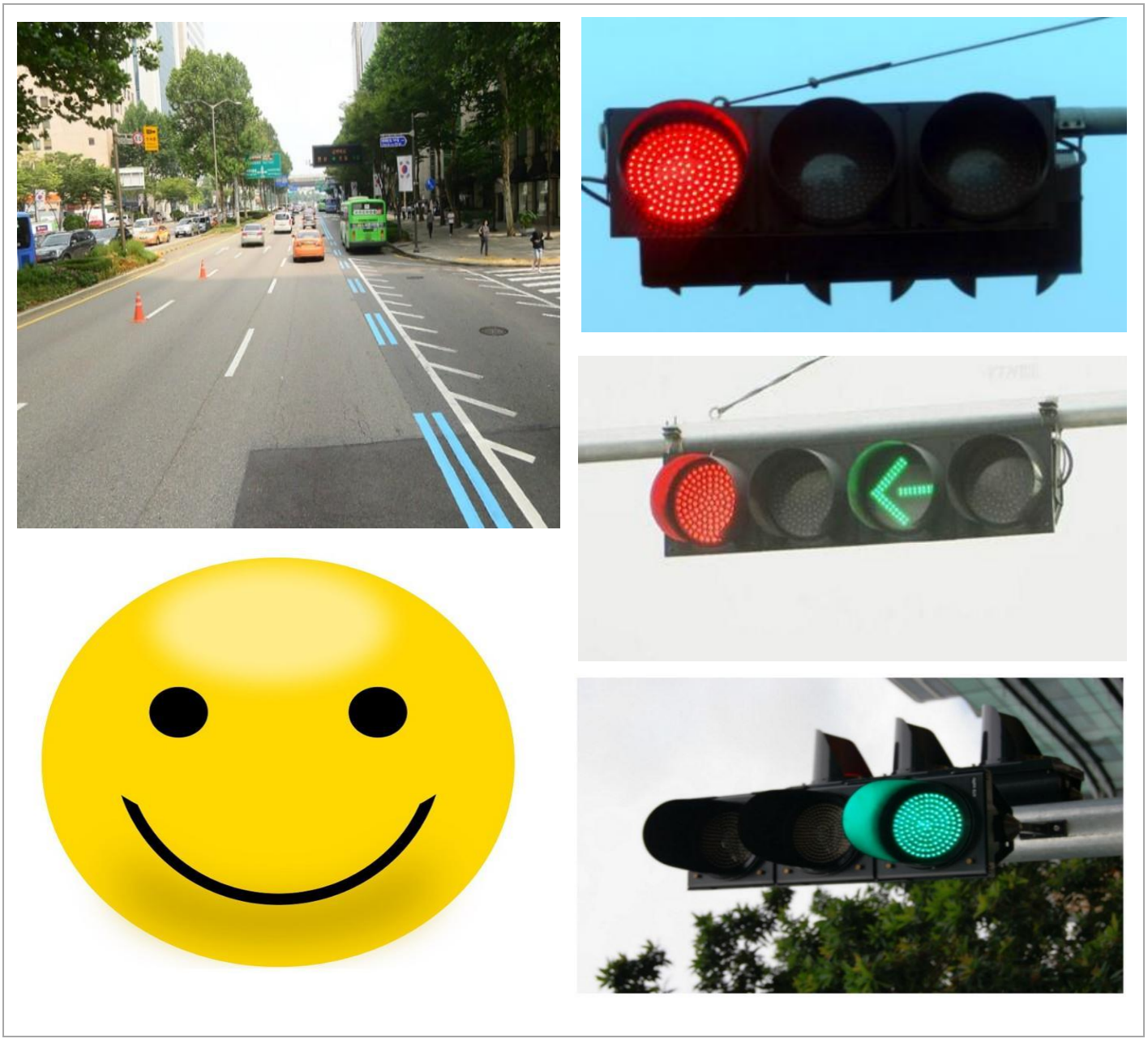
실제 해당 과정을 약 100번 정도 반복하면 약 95번은 정확히 음성을 인식하고 나머지 5번은 잘못 인식하는 경우가 나타난다. 이것이 바로 우리가 흔히 말하는 Test Accuracy와 동일한 의미를 가지게 되는 것으로 본 실습의 모델은 약 95%의 Test Accuracy를 가진다고 판단할 수 있다.

2. 영상 인식

이제 실제 주행 영상을 판별하는 영상 인식 모델을 구현한다.

실제 데이터 수집은 자동차에 부착된 카메라를 통해 영상을 촬영하고 이를 프레임 단위로 처리한 이미지를 활용하여야 한다. 그러나 이는 실습 환경 상 제약 사항이 존재한다. 따라서 본 실습에서는 주행 시 획득되는 여러 이미지들을 기반으로 주행 영상 인식 모델을 구축한다.

데이터 셋은 총 5개의 클래스로 이루어진 이미지 데이터로 **해당 클래스는 'Street', 'Red', 'Green' 'Left' 그리고 'Home'으로 구성**된다. 각 클래스는 15개로 이루어져 있으며 다음은 해당 이미지다.



[그림 8] 영상 인식을 위한 기본 데이터 셋

스마일 이미지는 클래스 'Home'을 의미한다.

실행 코드

```
# 라이브러리를 불러옵니다.
from pylab import*
import numpy as np
import cv2
import matplotlib.pyplot as plt
import tensorflow as tf
import time
```

데이터 처리를 위한 기본 라이브러리인 **numpy** 그리고 신경망 분석을 위한 **tensorflow**를 불러온다.
영상(이미지) 데이터 전 처리를 위한 라이브러리 **cv2**를 불러온다.
Image Plot을 위한 라이브러리 **matplotlib.pyplot** 역시 불러온다.
시간 측정을 위한 **time** 라이브러리를 불러온다.

영상 인식을 위한 라이브러리를 생성 후 데이터 셋을 불러온다.

데이터를 불러오기 앞서 이미지의 width 및 height 그리고 channel의 수를 정하고 이미지 Inflate를 위한 함수를 정의한다.

실행 코드

```
# 모델링에 사용될 이미지의 크기를 지정합니다.
width = 160
height = 120
cropped_width = 100
cropped_height = 100
num_channels = 3
save_dir = '/home/eduuser/NN/Data/AutoDrive_Image/imgs/'

sess = tf.Session()
init = tf.initialize_all_variables()
sess.run(init)

off_height = int((height - cropped_height)/2)
off_width = int((width - cropped_width)/2)

imgX = tf.placeholder(tf.uint8, [None,width,height,num_channels], name='x_input')

# 학습 데이터를 부풀리기 위한 함수를 정의합니다.
def pre_process_image(image, training=None):
    if training:
        image = tf.image.crop_to_bounding_box(image,offset_height=off_width,
```

```
                                                    offset_width=off_height,
                                                    target_height=cropped_height,
                                                    target_width=cropped_width
                                                    )
        image = tf.image.random_hue(image, max_delta=0.05)
        image = tf.image.random_contrast(image, lower=0.3, upper=1.0)
        image = tf.image.random_brightness(image, max_delta=0.2)
    else:
        image = tf.image.crop_to_bounding_box(image,offset_height=off_width,
                                                    offset_width=off_height,
                                                    target_height=cropped_height,
                                                    target_width=cropped_width
                                                    )
    return image

def pre_process(images, training):
    images = tf.map_fn(lambda image: pre_process_image(image, training), images)
    return images

distorted_images = pre_process(images=imgX, training=True)

def duplicate_image(image,num):     # Repeat the input image 9 times.
    image_duplicates = np.repeat(image[np.newaxis, :, :, :], num, axis=0)
    feed_dict = {imgX: image_duplicates}
    result = sess.run(distorted_images, feed_dict=feed_dict)
    return result
```

```
# 이미지 데이터 Inflate와 관련된 용어 및 함수들은 'CNN 이론교재, Cifar-10' 예제를 참조하기 바란다.
# 원본 이미지의 Width 및 Height는 각각 160, 120으로 정하고 Crop을 적용한 이미지의 Width 및 Height는 모두 100으로 지정한다.
# 원본 이미지의 색상이 컬러이기 때문에 채널의 수(num_channels)를 3으로 정한다.
```

이미지 Inflate를 위한 함수 지정 후 cv2 라이브러리를 사용하여 **이미지 데이터를 Array형태로 변환**하여 저장한다.

실행 코드

```
# 데이터를 불러옵니다.
left_data=[]
home_data=[]
green_data=[]
red_data=[]
street_data=[]

import os
for img in os.listdir(save_dir):

    tmp = cv2.imread(save_dir+img)

    tmp_resize = cv2.resize(tmp,(height,width))
    tmp_color = cv2.cvtColor(tmp_resize, cv2.COLOR_BGR2RGB)
    if 'left' in img:
        left_data.append(tmp_color)
    elif 'home' in img:
        home_data.append(tmp_color)
    elif 'green' in img:
        green_data.append(tmp_color)
    elif 'red' in img:
        red_data.append(tmp_color)
    elif 'street' in img:
        street_data.append(tmp_color)

left_data = np.asarray(left_data)
home_data = np.asarray(home_data)
green_data = np.asarray(green_data)
red_data = np.asarray(red_data)
```

```
street_data = np.asarray(street_data)
```

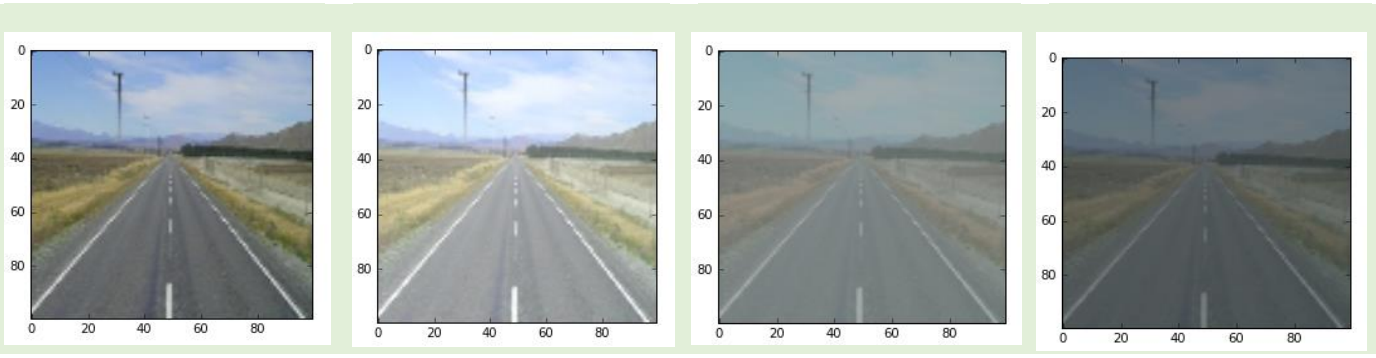

[이미지 데이터 Inflate]

총 5개의 클래스로 이루어진 학습데이터는 각 15개의 jpg파일로 이루어져있다. 이는 모델을 학습시키기에는 매우 부족한 양이다. 또한 각 이미지가 실제로 색상이나 명암이 깨끗한 상태로 촬영된다는 보장이 없다. 따라서 본 실습에서는 이미지의 Hue, Contrast 그리고 Brightness 등을 랜덤 하게 변형시켜 이미지를 **인위적으로 Inflate하는 작업을 진행**하도록 한다. 또한 추가적으로 각 이미지의 가장 자리를 제거하는 Crop 역시 적용하도록 한다.

실행 코드

```
# 실제 데이터에 이미지 부풀리기 함수를 적용합니다.
left_img=[]
home_img=[]
green_img=[]
red_img=[]
street_img=[]

N = len(left_data)

for i in range(N):
    left_img.extend(duplicate_image(left_data[i],100))
    home_img.extend(duplicate_image(home_data[i],100))
    green_img.extend(duplicate_image(green_data[i],100))
    red_img.extend(duplicate_image(red_data[i],100))
    street_img.extend(duplicate_image(street_data[i],100))

for i in arange(20,25):
    plt.imshow(street_img[i])
    plt.show()
```

출력 결과



위 결과는 'Street' 클래스에 대해 Inflate를 적용한 결과이다. 위와 같이 다양한 Hue, Contrast 그리고 Brightness가 적용된 것을 확인 할 수 있다.

[학습 데이터 생성]

Inflate로 생성된 데이터를 기반으로 학습 데이터를 생성하도록 한다.

실행 코드

```
image = []
image.extend(street_img)
image.extend(green_img)
image.extend(left_img)
image.extend(red_img)
image.extend(home_img)
image = np.array(image)

N_left = len(left_img)
N_home = len(home_img)
N_green = len(green_img)
N_red = len(red_img)
N_street = len(street_img)

# 이미지 라벨 데이터를 생성합니다.
y_street = [[1,0,0,0,0]] * N_street
y_green = [[0,1,0,0,0]] * N_green
y_left = [[0,0,1,0,0]] * N_left
y_red = [[0,0,0,1,0]] * N_red
y_home = [[0,0,0,0,1]] * N_home

label = []
label.extend(y_street)
label.extend(y_green)
label.extend(y_left)
label.extend(y_red)
label.extend(y_home)
label = np.array(label)

def scale(imgs):
    im=imgs/float(256)
    return im

train_x = scale(image)
train_y = label
```

```
# 컬러 이미지의 경우 각 Pixel의 최대값이 256이다. 따라서 전체 값을 256으로 나누어주는 scale 함수를 정의한다.
# 총 5개의 클래스에 대한 라벨은 각 [1,0,0,0,0], [0,1,0,0,0], [0,0,1,0,0], [0,0,0,1,0], [0,0,0,0,1]로 지정한다.
```

이제 Inflate된 이미지들의 Pixel값을 기반으로 총 5개의 클래스를 인식하는 모델을 구축하여야 한다. 영상(이미지) 인식의 경우 Convolutional Neural Network, CNN이 가장 효과적이다. 따라서 본 장에서는 CNN을 기반으로 모형을 구축하도록 한다.

[모델 파라미터 설정]

실행 코드

```
# CNN 파라미터 값을 지정합니다.
window_size = 5
num_filter_1 = 32
num_filter_2 = 64
num_neuron_1 = 512
num_neuron_2 = 256
num_classes = 5
```

```
pool_width_1 = 4
pool_heigth_1 = 4
pool_width_2 = 2
pool_heigth_2 = 2
```

본 실습에서는 2개의 Convolution Layer와 2개의 Hidden Layer(Fully-Connected Net)로 이루어진 Network를 구현한다. 따라서Convolution Layer에 필요한 Filter의 수는 각 32개 그리고 64개로 지정하고 Hidden Layer에 필요한 Neuron의 수는 512개 그리고 128개로 지정한다. 또한 모델에 입력되는 이미지의 사이즈가 100X100으로 큰 편에 속하기 때문에 Pooling 사이즈는 2X2뿐만 아니라 4X4도 추가한다.

[CNN 구축]

실행 코드

```
# CNN 모델을 정의합니다.
def weight_variable(shape,name):
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial, name=name)

def bias_variable(shape,name):
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial, name=name)

def conv2d(x,W):
    return tf.nn.conv2d(x,W,strides=[1,1,1,1], padding='SAME')

def max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize=[1,2,2,1], strides=[1,2,2,1], padding='SAME')

def max_pool_4x4(x):
    return tf.nn.max_pool(x, ksize=[1,4,4,1], strides=[1,4,4,1], padding='SAME')


x = tf.placeholder('float', [None,cropped_width,cropped_height,num_channels])
y_ = tf.placeholder('float', [None,num_classes], name='y_')

# 첫번째 합성곱 계층
W_conv1 = weight_variable([window_size,window_size,num_channels,num_filter_1],'weight_1')
b_conv1 = bias_variable([num_filter_1],'bias_1')

# ReLU 계층
with tf.name_scope('first_relu') as scope:
    h_conv1 = tf.nn.relu(conv2d(x,W_conv1) + b_conv1)

# 풀링 계층
with tf.name_scope('first_pool') as scope:
    h_pool1 = max_pool_4x4(h_conv1)

# 두번째 합성곱 계층 두
W_conv2 = weight_variable([window_size,window_size,num_filter_1,num_filter_2],'weight_2')
b_conv2 = bias_variable([num_filter_2],'bias_2')

# ReLU 계층
with tf.name_scope('second_relu') as scope:
    h_conv2 = tf.nn.relu(conv2d(h_pool1,W_conv2) + b_conv2)

# 풀링 계층
with tf.name_scope('second_pool') as scope:
    h_pool2 = max_pool_2x2(h_conv2)

flat_num = int(ceil(cropped_width/pool_width_1/pool_width_2) *
            ceil(cropped_height/pool_heigth_1/pool_heigth_2) *
            num_filter_2)
```

```
# Fully connected net 첫번째 계층
W_fc1 = weight_variable([flat_num, num_neuron_1],'weight_fc1')
b_fc1 = bias_variable([num_neuron_1],'bias_fc1')

h_pool2_flat = tf.reshape(h_pool2,[-1,flat_num])

h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1)+ b_fc1)

# 첫번째 드랍아웃
keep_prob = tf.placeholder('float')
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

# Fully connected net 두번째 계층
W_fc2 = weight_variable([num_neuron_1,num_neuron_2],'weight_fc2')
b_fc2 = bias_variable([num_neuron_2],'bias_fc2')
h_fc2 = tf.nn.relu(tf.matmul(h_fc1_drop, W_fc2) + b_fc2)

# 두번째 드랍아웃
h_fc2_drop = tf.nn.dropout(h_fc2, keep_prob)

# Softmax 계층
W_fc3 = weight_variable([num_neuron_2,num_classes],'weight_fc3')
b_fc3 = bias_variable([num_classes],'bias_fc3')

y_conv=tf.nn.softmax(tf.matmul(h_fc2_drop, W_fc3) + b_fc3)
clipped_y_conv = tf.clip_by_value(y_conv, 1e-5, 1.0)
```

위의 CNN 구축 코딩도 'CNN 이론 교재의 MNIST, Cifar-10'과 흡사하다. 다만 차이가 있다면 **tf.clip_by_value 메서드를 사용**하였다는 것이다. 이는 추정하는 변수의 최소값과 최대값을 지정해 줌으로써 추정치가 해당 범위 밖으로 나가지 않도록 제한하는 메서드이다.

기본적으로 Softmax Layer를 통해 출력되는 Output은 확률 값으로 그 값이 0이 될 경우 Loss 계산 시 기술적이 오류가 발생한다. 따라서 **tf.clip_by_value 메서드를 통해 확률의 최소값을 1e-5로 제한**한다.

실행 코드

```
# 손실 함수 및 추정 방법을 정의합니다.
with tf.name_scope('cross-entropy') as scope:
    cross_entropy = -tf.reduce_sum(y_* tf.log(clipped_y_conv))
    ce_summ = tf.scalar_summary('Entropy', cross_entropy)

with tf.name_scope('train') as scope:
    train_step = tf.train.AdamOptimizer(0.00003).minimize(cross_entropy)

with tf.name_scope('accuracy'):
    with tf.name_scope('correct_prediction'):
        correct_prediction = tf.equal(tf.argmax(y_conv, 1), tf.argmax(y_, 1))

    with tf.name_scope('accuracy'):
        accuracy = tf.reduce_mean(tf.cast(correct_prediction, 'float'))
        tf.scalar_summary('accuracy', accuracy)

# 세션을 시작하여 학습을 진행합니다.
sess = tf.Session()
```

```
sess.run(tf.initialize_all_variables())

train_batch_size = 100

def random_batch(arr_x,arr_y):
    # Number of images in the training-set.
    num_images = len(arr_x)
    idx = np.random.choice(num_images,
                           size=train_batch_size,
                           replace=False)
    x_batch = arr_x[idx]
    y_batch = arr_y[idx]
    return x_batch, y_batch

stime = time.time()


for i in range(2000):
    batchX, batchY = random_batch(train_x,train_y)
    sess.run(train_step, feed_dict={x:batchX, y_:batchY, keep_prob:0.5})
    if i%100==0:
        cost = sess.run(cross_entropy, feed_dict={x:batchX, y_:batchY, keep_prob:1.0})
        etime = time.time()
        print ('step %d, train cost %.4f, avg_time %.4f sec' %(i, cost, (etime-stime)/100.0))
        stime = time.time()
        if cost < 0.0001:
            break
```

```
# 배치 사이즈(train_batch_size)는 100으로 정한다.

# 총 2000만큼 반복하여 학습을 진행한다.
```

출력 결과

```
step 300, train cost 26.9835, avg_time 0.1540 sec
step 400, train cost 20.2478, avg_time 0.1535 sec
step 500, train cost 6.4818, avg_time 0.1535 sec
step 600, train cost 4.5960, avg_time 0.1521 sec
step 700, train cost 4.8395, avg_time 0.1545 sec
step 800, train cost 3.2914, avg_time 0.1533 sec
step 900, train cost 1.5011, avg_time 0.1542 sec
step 1000, train cost 1.0574, avg_time 0.1537 sec
step 1100, train cost 0.6011, avg_time 0.1519 sec
step 1200, train cost 1.0531, avg_time 0.1508 sec
step 1300, train cost 0.2793, avg_time 0.1508 sec
step 1400, train cost 0.2284, avg_time 0.1508 sec
….
Step 1900, train cost 0.1548, avg_time 0.1503 sec
```

학습된 모델을 기반으로 실제 자율 주행 시나리오 기반 테스트 이미지를 인식해 본다.

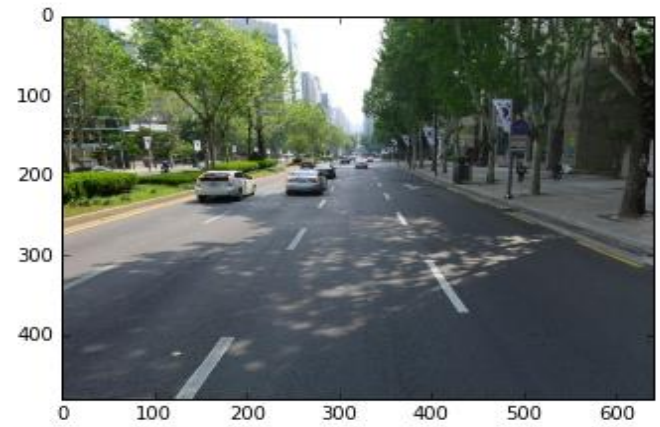
테스트 이미지는 총 16장으로 실제 자율 주행 시나리오 상에서 촬영된 영상이 16개의 이미지 프레임으로 나눠졌다고 가정하고 테스트를 진행한다. 만약 모델이 잘 구축되었다면 16개의 테스트 이미지의 클래스를 정확히 판별할 것이다.

실행 코드

```
# 테스트 이미지를 불러온 후 테스트를 진행하고 그 결과값을 출력합니다.
test_dir = '/home/eduuser/NN/Data/AutoDrive_Image/test_imgs/'
test_img = []

for img in sort(os.listdir(test_dir)):

    tmp = cv2.imread(test_dir+img)
    plt.imshow(cv2.cvtColor(tmp,cv2.COLOR_BGR2RGB))

    tmp_resize = cv2.resize(tmp,(height,width))
    tmp_color = cv2.cvtColor(tmp_resize, cv2.COLOR_BGR2RGB)

    picture = sess.run(tf.image.crop_to_bounding_box(
                                            tmp_color,
                                            offset_height=off_width,
                                            offset_width=off_height,
                                            target_height=cropped_height,
                                            target_width=cropped_width
                                            )
                       )

    plt.show()

    picture_reshape = np.reshape(picture,[-1,cropped_width,cropped_height,num_channels])
    scaled_picture = scale(picture_reshape)
    test_img.append(scaled_picture)

    result = sess.run(tf.argmax(y_conv, 1), feed_dict={x:scaled_picture, keep_prob: 1.0})
    if result == 0:
        print('street -> Drive!')
    elif result == 1:
        print('Green light -> Go Straight!')
    elif result == 2:
        print('Left light -> Turn left!')
    elif result == 3:
        print('Red light -> Stop!')
    elif result == 4:
        print('Destination!!')
```

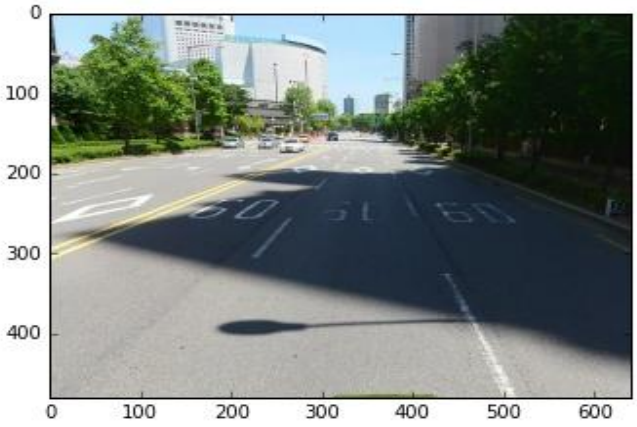
출력 결과

첫번째 프레임 판별 결과



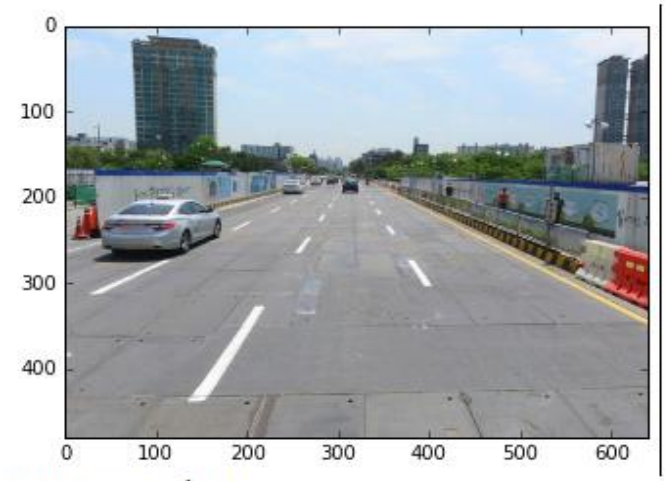

```
street -> Drive!
```

두번째 프레임 판별 결과



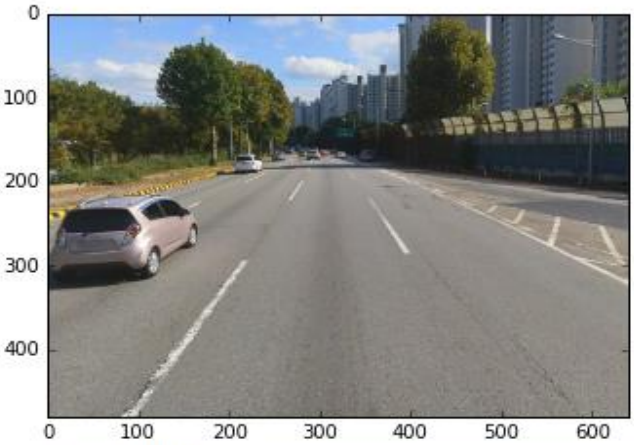

```
street -> Drive!
```

네번째 프레임 판별 결과



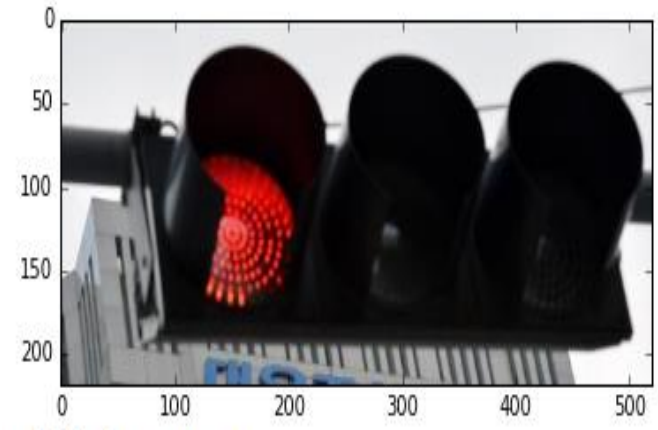

```
street -> Drive!
```

세번째 프레임 판별 결과




```
street -> Drive!
```

다섯번째 프레임 판별 결과




```
Red light -> Stop!
```

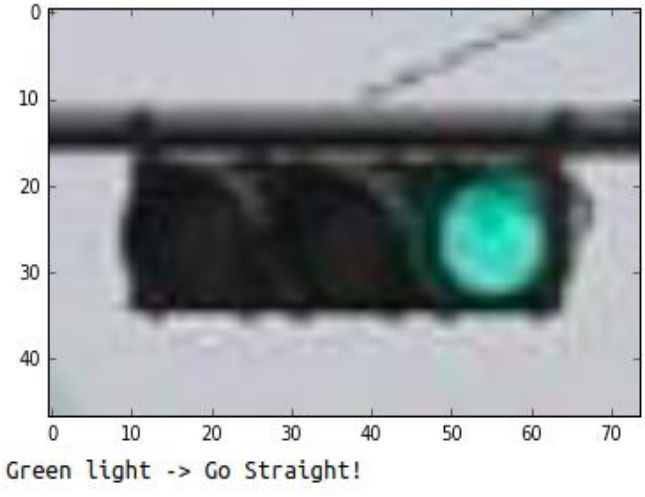
여섯번째 프레임 판별 결과


```
street -> Drive!
```

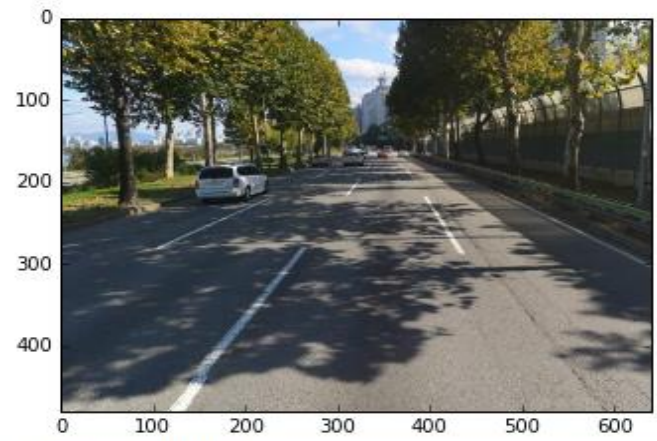
여덟번째 프레임 판별 결과

일곱번째 프레임 판별 결과

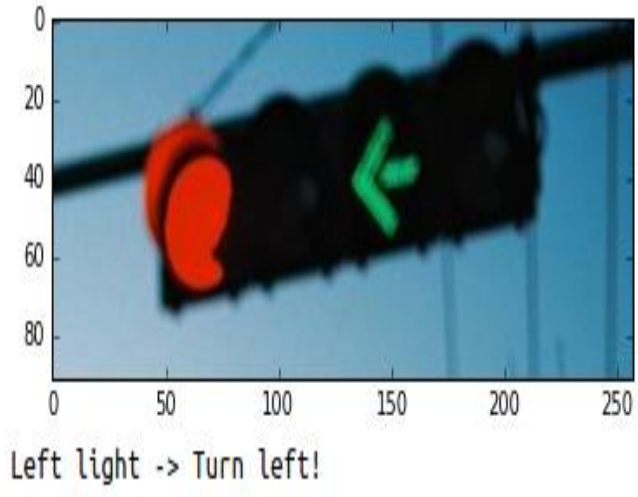
아홉번째 프레임 판별 결과



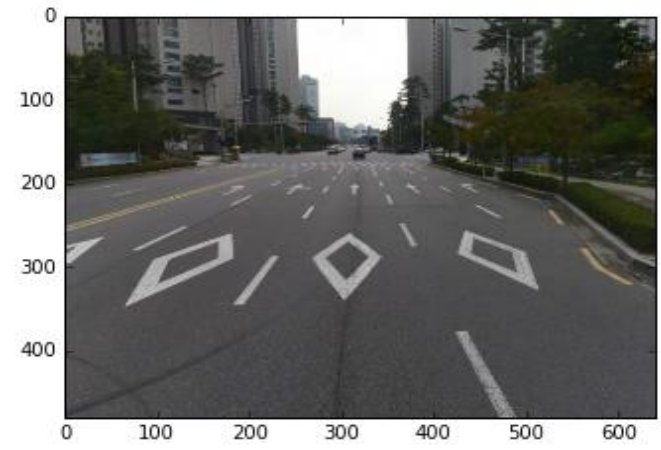
열번째 프레임 판별 결과



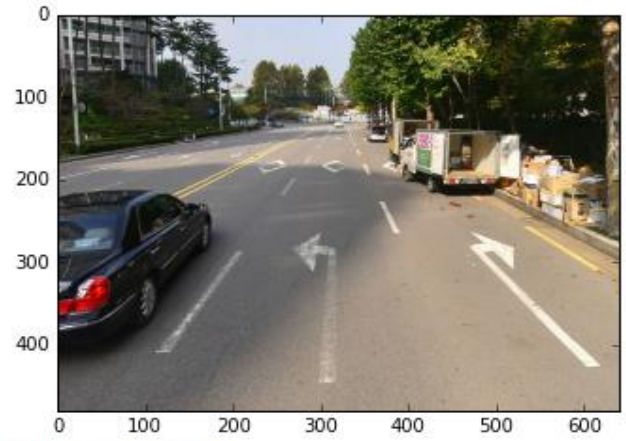
열 두번째 프레임 판별 결과



열 한번째 프레임 판별 결과



열 세번째 프레임 판별 결과

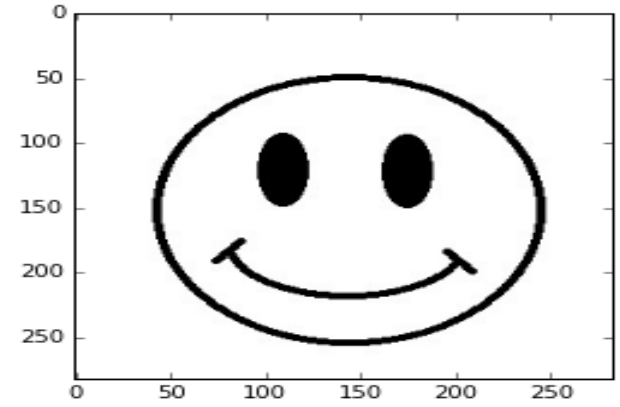



```
street -> Drive!
```

열 네번째 프레임 판별 결과


```
street -> Drive!
```

열 여섯번째 프레임 판별 결과




```
Destination!!
```

열 다섯번째 프레임 판별 결과


```
street -> Drive!
```

위 이미지들은 자율 주행 시나리오 기반 영상인식 테스트 이미지이다. 실제 자동차가 도로 및 신호등을 촬영하고 촬영된 영상을 이미지 Frame 단위로 인식하여 각 클래스를 판별한 결과이다. 위의 결과를 살펴보면 그 값이 정확하다는 것을 알 수 있다.

실제로 위 모델 추정을 시작으로 테스트까지의 과정을 약 100번 정도 실행하게 되면 93번은 이미지를 정확히 판별하고 나머지 7번은 잘못 판별한다. 이는 음성 인식에서도 언급했듯이 흔히 말하는 Test Accuracy와 같은 의미를 가진다. 즉 해당 모델의 Test Accuracy는 약 93%이다.