

# **Deep Learning 이론**

2016 빅데이터 교육 콘텐츠

나 어제 택배 부치려 우체국에 갔는데, 직원이 일일이 우편번호를 확인하더라~.

정말? 그래서 택배 부치는데 시간 좀 걸렸겠다!



맞아, 그래서 우편번호 자동 분류기가 있는지 궁금해졌어.

그냥 네가 하나 만들어보지 그래?  
너 딥러닝을 좀 찾아보지 않았어?



맞아! 그렇게 해볼까?  
와~ 메일 스팸 분류도 딥러닝이  
한다던대..

그래 해봐! 숫자로 분류하는 건  
너도 할 수 있을 것 같아.



그래! 그럼 어떻게 시작할 수  
있지? 난 정말로 조금밖에 몰라.

우리 딥러닝을 이용한 실제 사례를  
찾아보자. 여기서 힌트를 얻을 수  
있을거야!



## ★ 딥러닝을 활용한 사례 조사

1. 자율 주행 자동차 및 무인 택배 드론 @@ 차 타고 갈 때 편하고 오히려 사고도 줄 것 같아!!!

- 원리: 자동차와 드론이 스스로 주변 교통 상황과 행인, 지도에 대한 데이터를 능동적으로 수집하여 이를 분석 및 판단한다.

- 필요 기술:

1) 사람인지 아닌지, 사물인지 아닌지 판별하기 -> 영상인식 기술

2) 운전자의 목소리로 시동걸고 정차하기 -> 음성인식 기술

3) 교통 표지판과 주변 교통 상황 읽기 -> 이미지 인식 기술

4) 실시간으로 데이터 수집, 처리, 분석하기 -> 빅데이터 기술

- 필요 데이터 : 교통 표지판 데이터, 지도 데이터, 음성 데이터, 교통 상황 데이터

2. 우편번호 솔루션 인식 @@ 이걸 활용하면 우편번호 말고도 무궁무진할 것 같아!!!

- 원리: 손으로 쓰여진 우편번호를 읽고 자동 분류한다.

- 필요 기술:

1) 사람마다 다른 솔루션 인식하기 -> 이미지 인식 기술

2) 데이터 처리, 분석하기 -> 빅데이터 기술

- 필요 데이터: 숫자로 이루어진 솔루션 데이터 -> 온라인 상에 이런 데이터가 실제로 있어!!

3. 자동 보고서 생성

- 원리: 온라인 상의 수많은 데이터를 학습하여 종합적인 새로운 보고서를 자동으로 생성

- 필요 기술:

1) 언어를 학습할 수 있는 모델 필요

- 필요한 데이터: 특정 주제와 관련된 온라인상의 문서 모음

내가 작성한 “딥러닝을 활용한 사례들”이야. 어때?

우리가 알고 있는 것 보다 훨씬 더 적용되는 곳이 많았네?

응 맞아. 잠깐! 딥러닝 하려면 데이터는 필수인데 어디서 구하지?

내가 누구야~  
내가 이미 정리했어.  
학인해봐.

## ★ 나도 딥러닝을 할 수 있다 (데이터셋 찾기)

### 1. 솔루션 데이터로

- 위에서 우편번호로 솔루션 인식 찾아보다 알게 됨 -> MINIST 데이터라고 있음!

- 솔루션 데이터는 딥러닝을 시작하기 위한 아주 기초단계 -> 나도 할 수 있을 거야!!



### 2. 음원 데이터로

- 찾아보니 음악데이터가 있음 -> GTZAN Genre collection라는 데이터가 있음!

- 나는 음악을 듣는 것이 나의 행복한 취미! 꼭 이걸 활용해서 간단하게 딥러닝을 구현하고 싶어!!

### 3. 이미지 데이터

- 우리가 접할 수 있는 데이터를 모아놓은 것이 있음 -> cifar10라는 데이터!

### 4. 언어 학습 데이터

- 자동으로 보고서를 생성하려면 언어 학습이 필요하다 -> Penn TreeBank 데이터!

- 나만의 보고서 봇을 만들 수 있을까?

우와! 그러면 데이터도 있으니  
이제 방법만 알면 구현할 수  
있겠네?



맞아! 근데 너 어떤 방법이  
있는지 알고 말하는 거야?

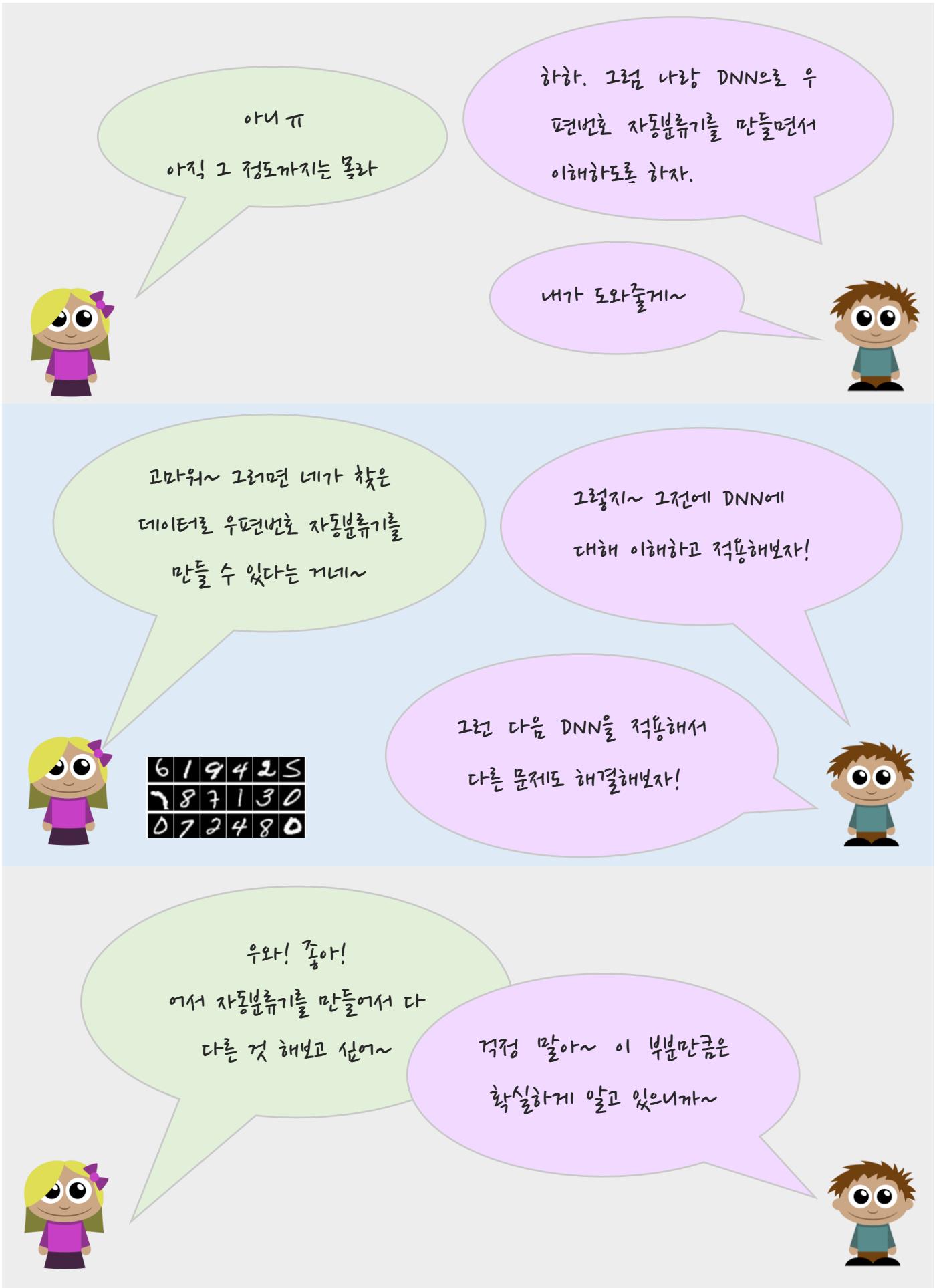


내가 모를 리가?  
친구야. 딥러닝은 DNN이란 알고리  
즘으로 시작하는 거야.



그래? 그럼 네가 만들고자  
하는 우편번호로 자동 분류기  
당장 만들 수 있겠네?





# CONTENTS

## 딥러닝 시작하기(1)

### 1단계

#### [기초]

DNN 이해 ↗ p8



#### [실습]

MNIST로 우편번호 손글씨 자동분류기 만들기 ↗ p14



#### [방법]

① 단일 계층 ↗ p18

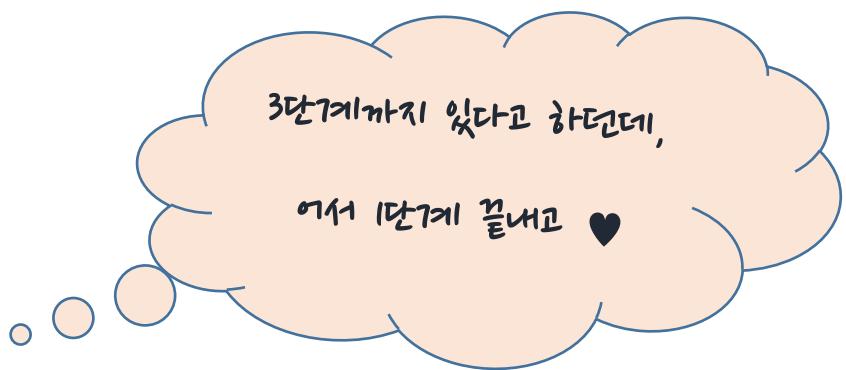
② 복합 계층 ↗ p24



#### [활용]

음악 장르 자동분류기 만들기 ↗ p28

(부제: GTZAN Genre Collection 음원 데이터로)



## [선수 지식 및 시스템 요구 사항 체크리스트]

	★ 독자 여러분은 이것을 이미 알고 있나요?
	파이썬 기초 문법
	파이썬 분석 라이브러리
	텐서플로우 사용법
	지능정보 기술 흐름에 대한 이해
	머신러닝 기법들 중 딥러닝이 가지는 장점
	딥러닝의 대표적인 알고리즘 종류
	딥러닝으로 얻을 수 있는 효과
	★ 독자 여러분의 PC는 다음 사항들을 충족하나요?
	운영체제: Ubuntu 14.04 LTS/CentOS 7 이상
	텐서플로우 버전: r0.11 이상
	파이썬 버전: 3.5.x



## I. DNN은 어떤 알고리즘이죠?

여러분은 지능 정보 입문 문서를 읽었으므로 딥 러닝을 구현하기 위한 방법론 중에서 DNN이 있다는 것을 알 것입니다. 그러면 DNN에 대해서 구체적으로 알아볼까요?

DNN은 Deep Neural Network의 준말로 딥 러닝 구현 알고리즘에서 가장 기본이 되는 심층 신경망입니다. 즉 이것은 학습을 통해서 얻은 모델을 기초로 결과를 추론하기까지의 과정입니다.

### # 딥 러닝에서의 학습과 추론이란?

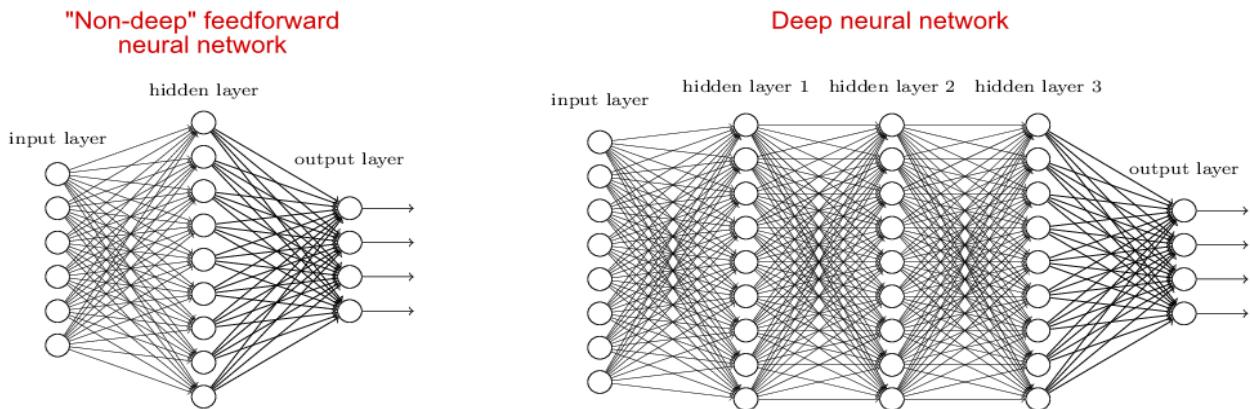
딥 러닝을 어린아이의 학습 방법과 연결시켜 봅시다. 어린아이는 하나의 특정 사물을 인식하기 위해 수 차례에 걸쳐 그 사물의 특징을 보고 듣고 만지는 과정을 반복하게 됩니다. 이러한 과정을 통해 사물을 인식하게 되면 그 모습이 조금 다를지라도 쉽게 사물을 구분할 수 있게 됩니다. 이 과정을 '학습'이라고 합니다.

그러면 추론은 무엇일까요? 추론은 학습된 모델에 대해서 나온 예측 값을 말합니다.

그러면 가장 기본적인 구조라는 DNN에 대해서 알아볼까요?

DNN은 인공신경망에서 진화된 것으로 당연히 [입력 계층(Input Layer)]-[은닉 계층(Hidden Layer)]-[출력 계층(Output Layer)]으로 나눠져 있습니다. 그러면 왜 때문에 DEEP이라는 용어가 붙은 것일까요?

이 점에 대해 아래 그림으로 설명하도록 하겠습니다.



[그림 1] DNN의 구조

위의 그림의 두 가지 모두 뉴럴 네트워크입니다. 왼쪽 그림은 은닉 계층이 단 하나만 들어있고, 오른쪽 그림은 은닉 계층이 3개가 들어있습니다. 두 그림에서 큰 차이가 없지요? 이처럼 일반적으로 은닉 계층이 두 개 이상 들어간 것을 DNN이라고 말합니다.

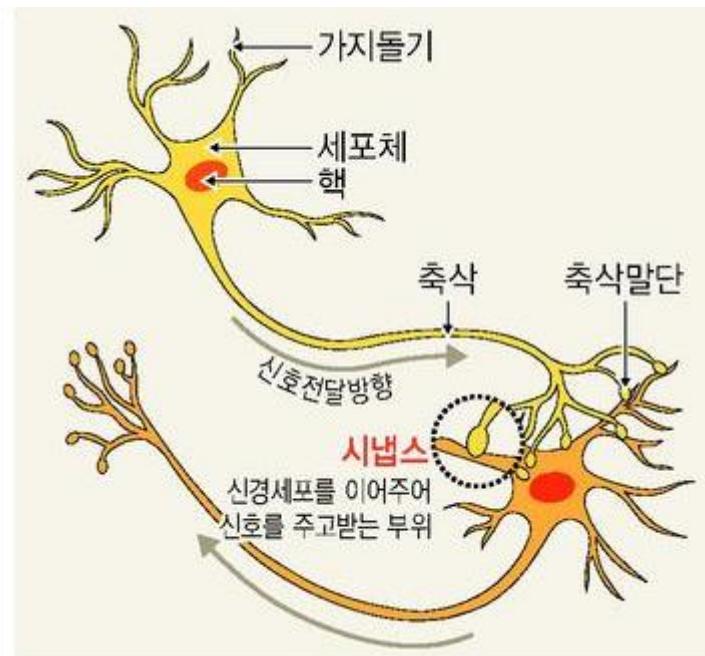
그러나 실제 구현된 DNN의 경우에는 저렇게 간단하지 않으며 은닉 계층이 60-90개 정도로 이뤄져 있다고 합니다.

## II. DNN으로 모델링을 어떻게 하나요?

자 여러분은 이제 DNN이 여러 개의 은닉 계층으로 이루어진 것을 알게 됐습니다. 그럼 DNN이란 어떤 매커니즘으로 구성된 것일까요? 이에 대해 알아봅시다.

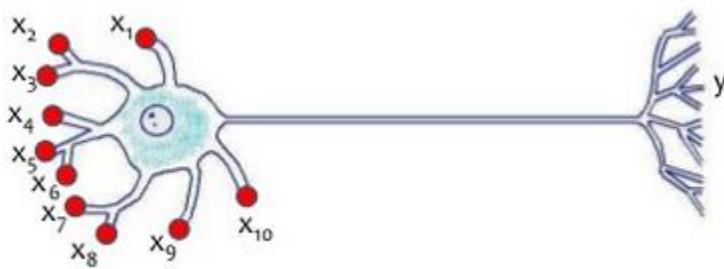
뇌 속의 실제 뉴런을 생각해봅시다. 뉴런이 수 백개가 서로 연결되어 있고 그 뉴런들은 신호를 전달하는 과정을 수 없이 거칩니다. 이때 전달되는 과정에서의 신호는 수상돌기(입력)에서 받은 신호와 축삭돌기( 출력)로 나가는 신호는 같은 것일까요?

분명히 다를 것입니다. 그 이유를 설명하자면 초기에 입력 받은 개별 신호들의 중요도가 서로 다르기 때문에 출력될 때도 동일한 값으로 출력되지 않고 어떤 변수나 환경이 추가되어 변형된 신호가 출력될 것이라고 단순하게 생각하면 될 것입니다.



[그림 2] 신경세포의 신호 흐름

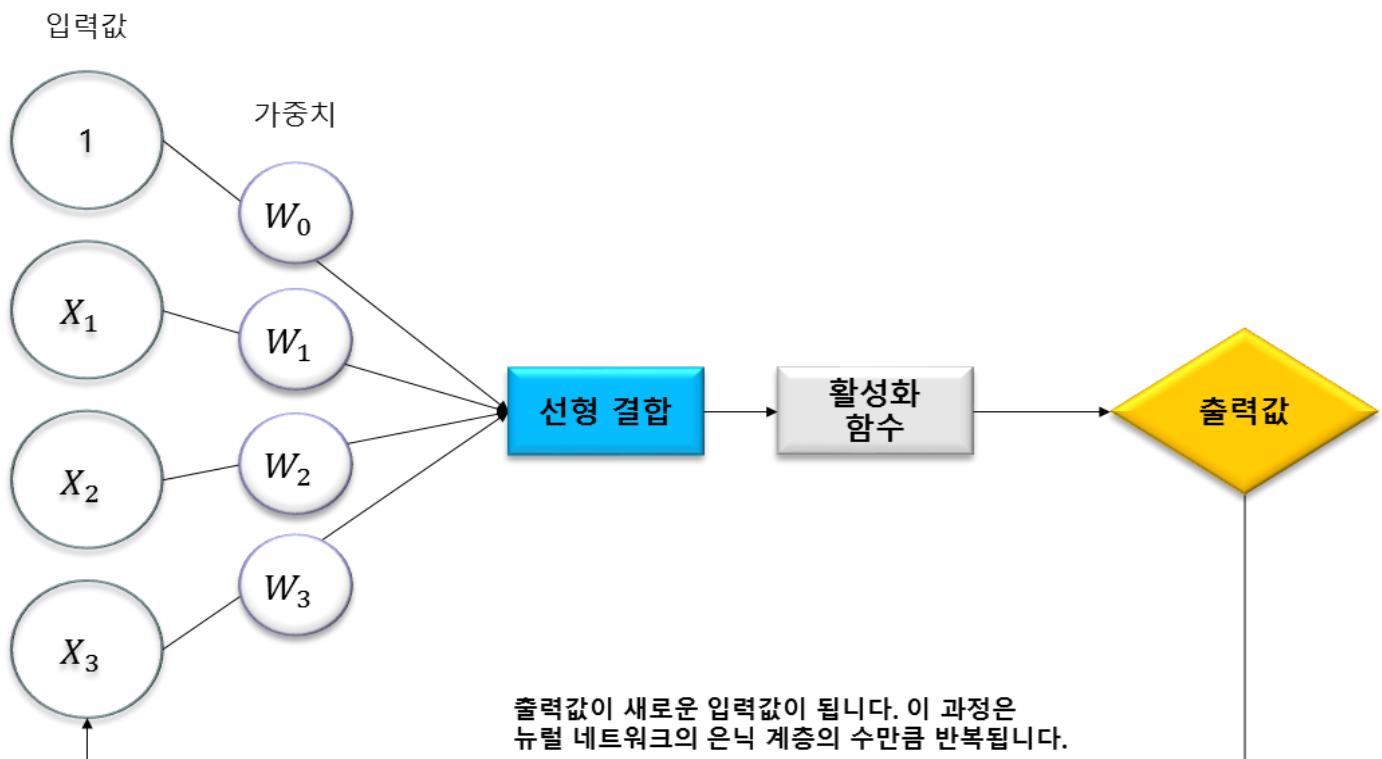
이런 뉴런의 구조에 착안해서 만들어진 인공 신경망에서도 비슷한 매커니즘을 가질 것이라고 생각합시다. 한 예로 수상돌기에서 축삭돌기로 신호가 전달되는 것을 입력 계층에서 출력 계층까지의 과정이라고 볼 수 있습니다.



[그림 3] 신경세포의 신호 전달 과정

여기 그림과 같이 뉴런의 시냅스가 10 개라고 가정합시다. 그리고 각 시냅스에서 받은 입력 신호가 총 10 개로 이뤄진  $x_1 - x_{10}$  이라고 합시다. 이것들은 신호를 처리하기 위해 어떤 과정을 거쳐  $y$ 로 출력될 것입니다. 다시 여기의  $y$  값들은 다른 뉴런의 시냅스의 입력값이 될 것입니다.

우리는 그 어떤 과정을 구체화하는 작업이 필요하는데, 이것을 다시 말하자면 DNN으로 구현한다는 것을 의미합니다. 그러면 DNN을 어떻게 모델링 해야하는 것일까요? 결론적으로는  $x$ 에서  $y$ 가 되는 과정을 모델링 해야합니다. 이 모델은 우리가 일반적으로 아는 **선형 모델을 기초로 합니다.**

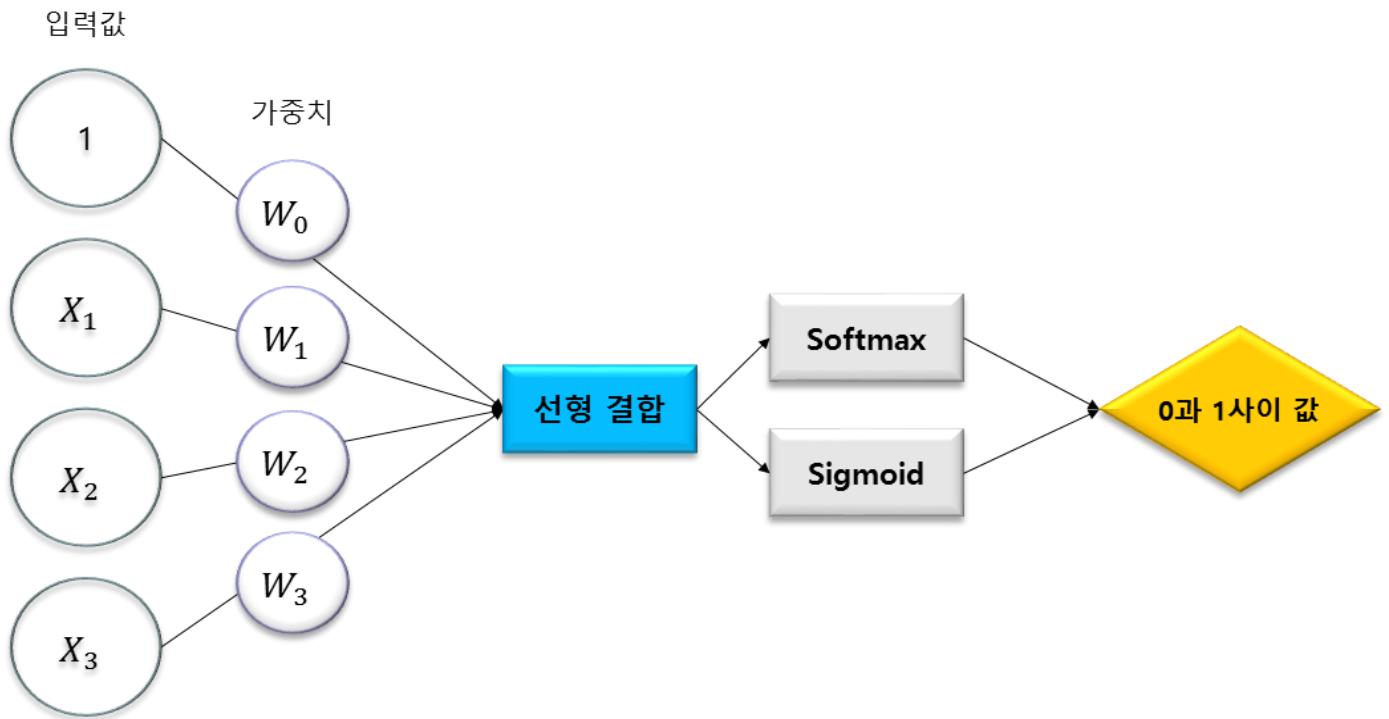


[그림 4] 신경망의 기본 구조

위 그림과 같이 입력값들은 가중치( $w$ )와의 선형 결합 그리고 활성화 함수(Activation Function)를 통해 출력값으로 변환되고 이러한 과정이 은닉 계층의 수 만큼 반복되어 최종적으로 우리가 원하는 결과값이 됩니다. 이해가 되셨나요? 아마 완벽하게 이해가 되진 않았을 겁니다. 이 글을 읽고 있는 독자 여러분들은 두 가지 의문이 머릿속에 떠오를 것입니다.

### 가중치는 어떻게 결정되는가? 그리고 활성화 함수란 무엇인가?

먼저 활성화 함수는 조금 더 좋은 결과값을 출력하기 위해 신경망의 은닉 계층 사이사이에 넣어주는 것입니다. 예를 들어 우리가 원하는 결과값이 0과 1 사이의 확률값이라면 Softmax 또는 Sigmoid를 활성화 함수로 넣어주면 더 좋은 결과값을 얻게 됩니다(Softmax나 Sigmoid 함수는 항상 0과 1 사이의 값을 출력함).

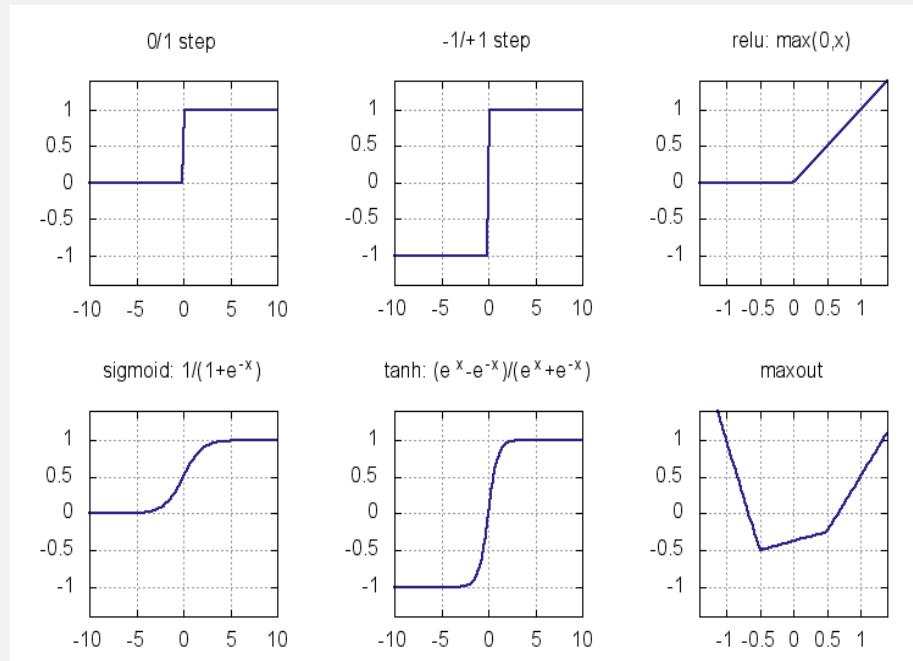


[그림 5] 활성화 함수의 역할

활성화 함수는 위에서 언급한 Softmax 그리고 Sigmoid 외에도 그 종류가 다양합니다.

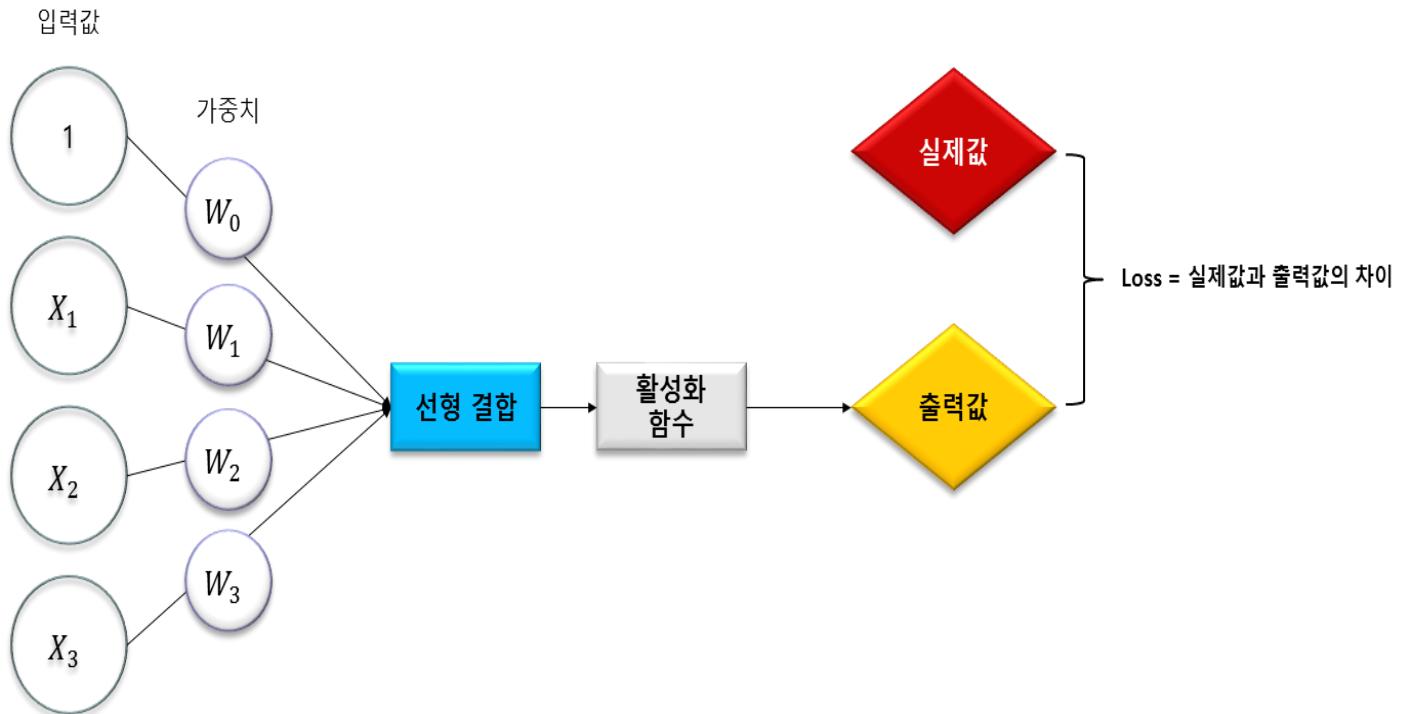
#### # 활성화 함수(Aactivation Function)의 종류

여러 개의 활성화 함수에서 적절한 것을 고르는 것은 어떤 모델로 어떤 결과를 나타낼 것인지를 정하는 것을 의미합니다. 주로 쓰이는 활성화 함수로는 시그모이드, 소프트맥스, 하이퍼탄젠트, ReLU 등이 있으며 이 과정을 거쳐야만 원하는 모델 구성이 완료 됩니다. 이에 대한 사용 방법은 실습을 통해 소개하겠습니다.



[그림 6] 활성화 함수의 종류

그렇다면 가중치는 어떻게 결정될까요? 답을 말하자면 학습을 통해 Loss를 최소화하는 방향으로 추정됩니다. 앞서 말한 바와 같이 심층 신경망은 많은 양의 데이터를 입력 값으로 하여 수 많은 은닉 계층을 거쳐 결과값을 추론하게 됩니다. 이 다음 과정은 무엇일까요? 당연히 추론한 결과값과 실제 결과값을 비교해야 합니다. 즉 우리가 신경망을 통해 추론한 결과값과 실제 결과값과의 차이를 손실(Loss)로 정의하고 이러한 차이를 표현한 함수를 손실 함수(Loss Function)라고 부릅니다.

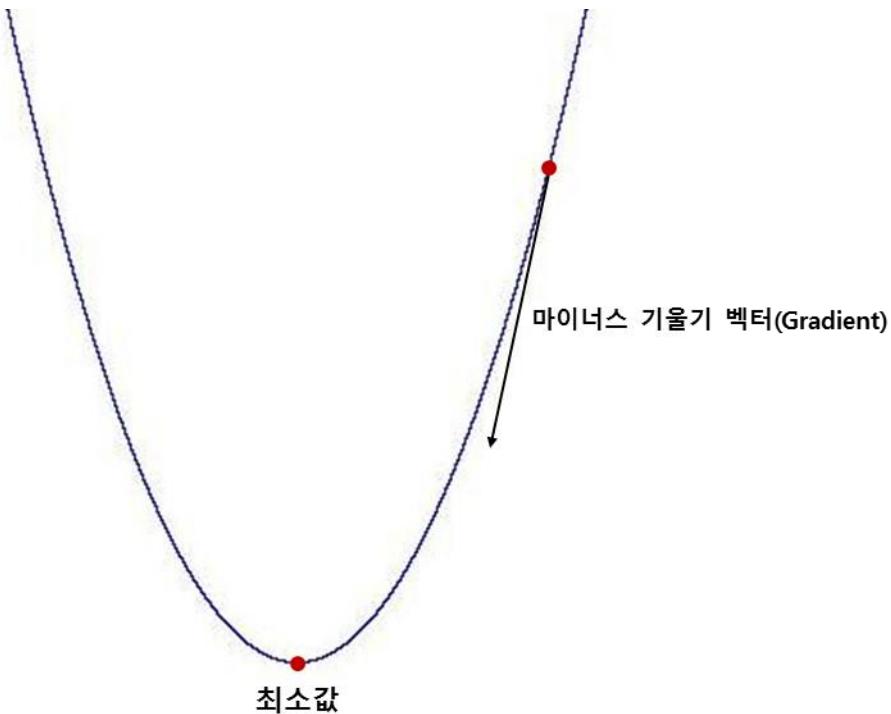


[그림 7] Loss의 의미

우리는 손실의 정의에 대해 살펴보았습니다. 앞에서 언급한 바와 같이 가중치는 이러한 손실을 최소화하는 방향으로 추정이 됩니다. 그렇다면,

### 손실을 최소화하는 방향은 어떻게 결정될까요?

손실을 최소화하는 방향은 기울기 벡터(Gradient)를 통해 정해집니다.



[그림 8] 기울기 벡터(Gradient)의 의미

위 그림과 같이 마이너스 기울기 벡터(Gradient)는 현재 위치에서 함수의 최소값으로 가는 가장 빠른 방향을 정해줍니다. 이와 같은 과정을 반복하여 마이너스 기울기 벡터를 따라가게 되면 결국은 해당 함수의 최소값으로 수렴할 수 있습니다. 이를 경사하강법(Gradient Decent)이라 부릅니다. 뉴럴 네트워크의 관점에서 설명하면, 학습을 통해 현재 가중치 값에 해당하는 손실 함수의 마이너스 기울기 벡터(Gradient)를 구하게 되고 이와 같은 과정을 반복함으로써 손실 함수를 최소화하는 가중치 값을 추정하게 됩니다. 이렇게 추정된 값을 기반으로 최종 모델이 결정됩니다.

## I. 간단한 우편번호 자동 분류기 만들기

우리는 이미지를 자동으로 분류하는 문제를 해결하기 위해서는 다른 머신 러닝 기법보다도 딥 러닝으로 구현하는 것이 훨씬 더 효과적임을 알고 있습니다. 그러면 우리도 간단한 우편번호 자동분류기를 만들어볼까요?

우편번호 자동 분류기처럼 숫자를 분류할 수 있도록 만들기 위해 필요한 것들을 생각해봅시다.

- 딥 러닝 구현 알고리즘
- 우편번호와 같은 손글씨 숫자 데이터

첫 번째의 딥 러닝 구현 알고리즘은 앞에서 설명한 DNN을 사용할 것입니다. 두 번째의 우편번호와 같은 손글씨 숫자 데이터를 생각해봅시다. 어디서 이런 것을 어떻게 구할 수 있을까요?

딥 러닝으로 학습 시키기 위해선 많은 학습 데이터 셋이 필요합니다. 우리가 손으로 일일이 숫자를 쓰는 방법도 가능합니다만 그러기엔 너무 힘들겠죠? 다행히도 이러한 손 글씨 데이터는 온라인 상에서 제공하고 있어 API를 통해 다운로드가 가능합니다.

다음 챕터에서는 이러한 손글씨 데이터에 대해 조금 더 상세히 알아보도록 하겠습니다.

## II. 손글씨 데이터는 어떤 건가요?

이제 우편번호 분류기 만들기를 본격적으로 시작해볼까요? 우리가 찾은 손글씨 데이터는 어떻게 생겼는지를 알아야 딥 러닝을 효과적으로 적용할 수 있겠죠?

여러분이 찾은 손글씨 데이터는 아마도 MNIST 라고 불리는 데이터 셋 일 것입니다. 사실 이 데이터 셋은 딥 러닝을 학습하기 위해 너무나도 많이 쓰이는 유명한 데이터 셋 중 하나입니다.

얼마나 유명한지에 대해 단 하나의 문장으로 여러분을 납득시키도록 하겠습니다.

"MNIST로 딥 러닝을 하는 것은 첫 코딩을 배울 때 "Hello, World!" 를 작성하는 것과 같다."

이처럼 여기서 사용될 손글씨 데이터는 우리의 목표인 우편번호 분류기를 만드는 데 쓰여야 할 데이터라는 것과 동시에 딥 러닝을 시작한다면 당연히 한번은 시도해봐야 하는 좋은 데이터입니다.



[그림 9] MNIST 손글씨 데이터 샘플

위의 그림은 MNIST 데이터의 샘플입니다. 이러한 데이터가 훈련용으로 55,000 개와 테스트용으로 10,000개로 이뤄졌습니다. 또한 딥 러닝의 시작으로 많이 쓰이는 데이터셋 답게 깔끔하게 전처리와 포매팅(Formatting) 되어 있기에 바로 사용하면 됩니다.

### # 데이터 전처리는 왜 필요 한가요?

기본적으로 이미지 데이터는 JPG 파일입니다. DNN을 구현하기 위해서는 이러한 형식의 데이터를 숫자로 이루어진 Array형식으로 바꾸는 것이 꼭 필요합니다.

## III. 이미지를 어떻게 인식하나요?

손글씨 데이터, MNIST는 결국 흑백 이미지 데이터입니다. 이러한 이미지 데이터를 컴퓨터는 어떻게 인식할까요? 컴퓨터가 이미지를 인식하는 원리에 대해서 알아봅시다.

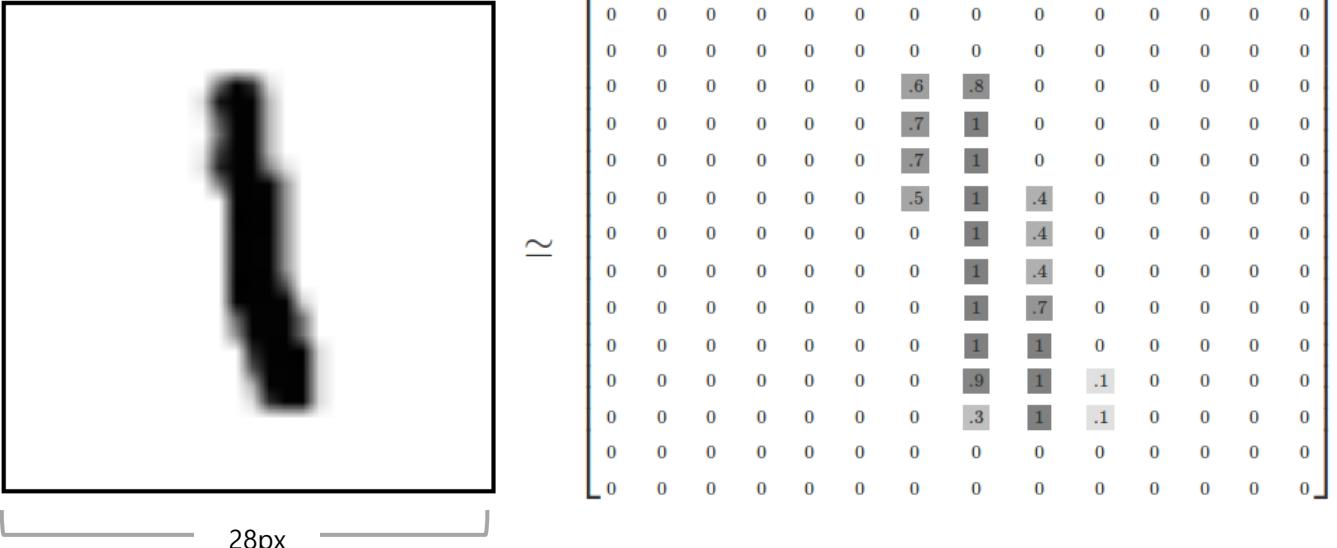
우선 손글씨 데이터인 MNIST를 다운로드 하여 봅시다. 검색 창에 MNIST라고 써서 검색하면 나옵니다. 우리는 여러분의 편의를 위해서 링크를 알려드리겠습니다. (<http://yann.lecun.com/exdb/mnist/>)

화면 중간에 보면 아래와 같이 되어있고, 아래 4개의 데이터를 모두 받고 적절한 위치에 넣어줍니다.

[train-images-idx3-ubyte.gz](#): training set images (9912422 bytes)  
[train-labels-idx1-ubyte.gz](#): training set labels (28881 bytes)  
[t10k-images-idx3-ubyte.gz](#): test set images (1648877 bytes)  
[t10k-labels-idx1-ubyte.gz](#): test set labels (4542 bytes)

[그림 10] MNIST 데이터셋의 구조

다운로드 한 파일의 이름으로도 알 수 있듯이 “훈련용”과 “테스트용”이 구분 되어있습니다. 또한 이것들은 각각 “이미지”와 “라벨”로 구분됩니다. 여기서 라벨이란 해당 이미지에 그려져 있는 숫자 값이 적혀 있는 값을 말합니다. 이제 이미지와 라벨을 어떻게 인식하는지에 대해서 알아볼까요?



[그림 11] 숫자 이미지와 그 벡터

위 그림의 왼쪽은 이미지고 오른쪽은 그 이미지가 벡터로 표현된 것입니다. 우리는 왼쪽 이미지를 보면 “어? 1이다”라는 것을 알 수 있지만 컴퓨터는 그렇지 못합니다. 그저 “이미지”를 받았다는 사실만 인지할 뿐입니다. 따라서 이를 컴퓨터가 이해할 수 있도록 쪼개고 숫자화하여 인식하게끔 만드는 작업이 필요합니다. 그렇다면 이를 어떻게 할까요?

우선 모든 숫자 이미지 파일은 28 pixel로 이뤄진 정사각형입니다. 여기서 숫자화를 어떻게 할까요? 이미지를 보면 희거나 약간 어둡고 그리고 어두운 상태만 있다는 것을 알 수 있습니다. 우리는 이 점을 이용해 각 픽셀마다 얼마나 밝은 지에 대해 숫자 값 표기했습니다.

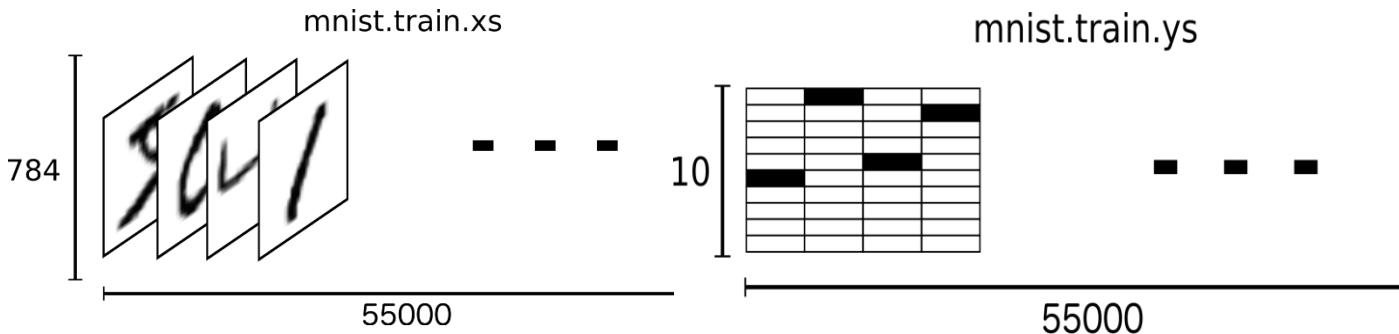
그 숫자 값으로 표현된 것이 바로 오른쪽 그림입니다. 이제 이해할 수 있겠습니까? 그림의 왼쪽 오른쪽은 모두 같은 이미지를 가리키는 것이며 오른쪽은 그 이미지를 숫자화 한 것을 보여준 것입니다. 여기서 이미지의 밝기는 **가장 어두운 것이 1이며 가장 밝은 것이 0입니다.** 그리고 그 중간 값은 0과 1의 실수로 표기돼 있습니다.

다시 말해 컴퓨터가 숫자 이미지를 받게 되면 숫자가 1이라는 것을 인식하기 전에 픽셀의 전체 수만큼 이뤄진 벡터로 우선 인식하게 됩니다. 당연히 숫자 이미지의 픽셀 전체 수는 784개( $=28 \times 28$ )로 이뤄졌음을 알 수 있으며, 동시에 784개의 크기를 가진 1차원 숫자 배열로 인식하게 됩니다.

자 이제 숫자 이미지를 어떻게 인식하는지 알게 됐고, 그러면 라벨은 어떨까요? 결론적으로 말하자면 숫자 이미지와 같은 구조입니다. 숫자 1이라고 쓰여 있는 이미지에 대해서 이미지가 1이라는 것을 알려주기 위해  $[0,1,0,0,0,0,0,0,0]$ 로 구성되어 있는 1차원 배열입니다. (이를 One-Hot Vector라고 합니다.)

## # One-Hot Vectors

벡터로 표현된 값이 있을 때, 그 벡터가 단 하나의 개념을 나타내는 것을 의미합니다.



[그림 12] MNIST 트레이닝 셋 구조

그럼 이제 위의 그림을 이해할 수 있을 것입니다. 우리는 딥러닝을 텐서플로우로 구현할 것이므로 텐서(Tensor)에 적용되는지를 봐야합니다. `Mnist.train.xs`라는 이름의 텐서에 이미지 트레이닝셋을 넣을 수 있으며 그 텐서의 크기는 [55000, 784]라고 볼 수 있습니다. 마찬가지로 `mnist.train.ys`라는 이름의 텐서에 숫자 라벨 트레이닝셋을 넣을 수 있으며 그 텐서의 크기는 [55000, 10]이 되겠습니다.

## IV. 숫자 데이터 어떻게 불러오나요?

지금까지 손글씨 데이터에 대한 전반적인 설명을 진행하였습니다. 이젠 텐서플로우라는 딥 러닝 도구를 통해 MNIST를 불러오도록 하겠습니다.

우리는 텐서플로우라는 딥러닝 라이브러리를 어느 정도 사용할 수 있다는 가정하에 글을 작성했습니다. 따라서 이 과정이 이해되질 않는다면 텐서플로우 부분에 대해 참고하시기 바랍니다.

여러분이 쓰기 편한 파이썬 IDE나 파이썬 콘솔이든 뭐든 좋으니 열어봅시다. 이제 아래처럼 코드를 작성하고 좀 전에 다운로드 한 MNIST 데이터셋의 디렉토리 위치를 파악한 뒤 코드에 적절히 맞춰 넣습니다.

### 실행 코드

```
# 라이브러리 및 데이터 셋을 불러옵니다.
import tensorflow as tf
import numpy as np
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

### 출력 결과

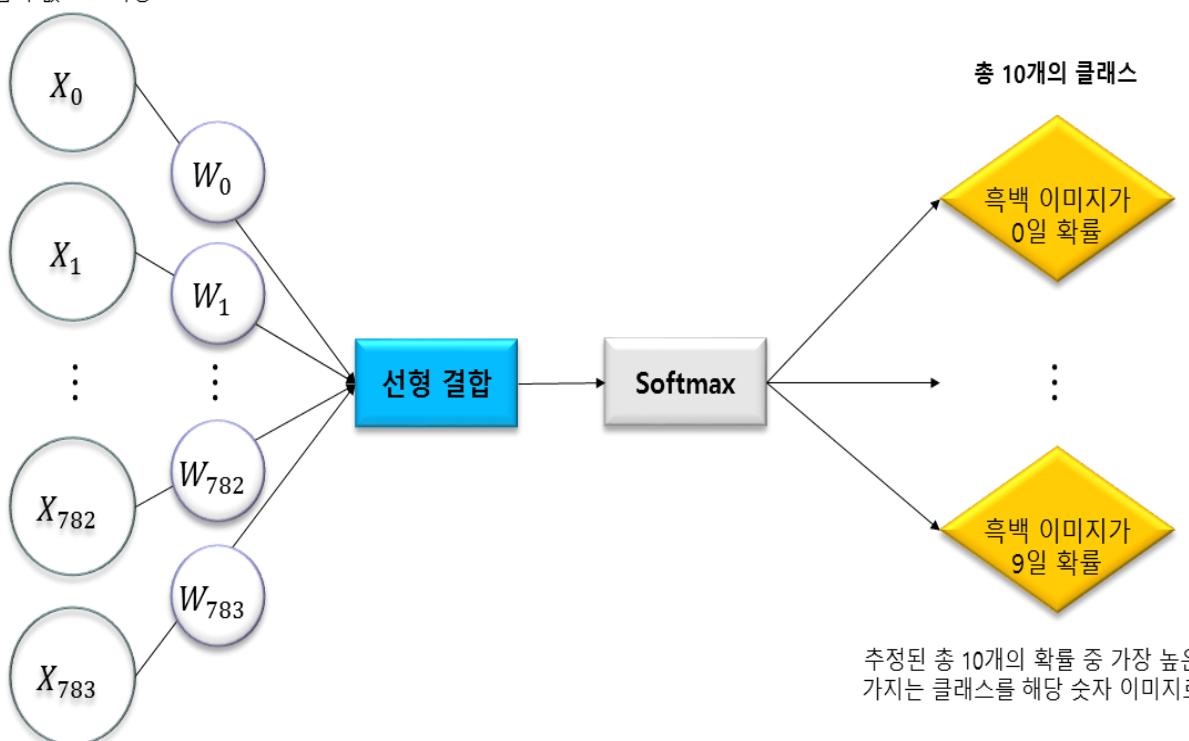
```
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
```

## I. 이제 구현할 모델링 흐름을 설명해주세요.

드디어 딥러닝의 시작입니다. DNN으로 우편번호 자동분류기를 구현해봅시다.

### [DNN 단일 계층 구성도]

숫자 이미지의 Pixel값(총 784개)을  
입력 값으로 지정



### [분석 과정]

라이브러리  
불러오기

데이터  
불러오기

모델 구축

모델 학습

모델 평가

### [분석 환경]

- 운영체제: Ubuntu 14.04 LTS 이상, CentOS 7 이상
- 텐서플로우 버전: r0.11 이상 (CPU & GPU 버전)
- 파이썬 버전: 3.5.x

## II. 직접 DNN으로 작성해서 보여주세요.

이제 DNN으로 간단한 우편번호 손글씨 자동분류기를 만들어봅시다.



우선 사용할 라이브러리와 데이터를 불러옵니다.

## 실행 코드

```
# 라이브러리 및 데이터 셋을 불러옵니다.  
import tensorflow as tf  
import numpy as np  
from tensorflow.examples.tutorials.mnist import input_data  
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

모델 학습을 위해 인풋과 아웃풋을 정의합니다. 아래의 과정은 일반적으로 맨 처음에 함께 작성합니다.

## 실행 코드

```
# 이미지와 라벨 값을 정의합니다.  
x = tf.placeholder(tf.float32, [None, 784])  
y_ = tf.placeholder(tf.float32, [None, 10])
```

MNIST 데이터셋의 이미지는 총 784( $=28 \times 28$ )의 실수로 구성됐기에 데이터타입과 그 형태를 각각 float32, [None, 784]로 정해야 합니다. 여기서의 **None**은 크기를 정하지 않고 어떠한 크기도 가능하다는 것을 말합니다. 정확히 말하자면 맨 처음 MNIST 데이터 소개할 때의 그림처럼 해당 이미지의 총 개수를 의미합니다.



자 이제 텐서플로우와 라이브러리를 불러오고 기본 변수를 정했으니 다시 'DNN 단일 계층 구성도'를 봅시다. 아직 정해지지 않은 가중치  $w$ 와 편향  $b$ 를 저장할 변수를 만듭니다.

## 실행 코드

```
# 가중치와 편향을 정의합니다.  
W = tf.Variable(tf.zeros([784, 10]))  
b = tf.Variable(tf.zeros([10]))
```

여기서 두 변수는 반복된 훈련으로 최적의 값을 찾는 것이 목표이기에 `tf.Variable` 이라고 변수를 생성합니다. 또한 모두 0으로 초기화를 합니다.

가중치와 편향이 [784, 10], [10] 이렇게 돼 있는 것을 앞서 말했던 것처럼 하나의 이미지가 갖는 입력 값이

784(=28x28)이고, 출력 값이 10개의 클래스를 가져야 하므로 이렇게 구성된 것입니다.

## 실행 코드

```
# 출력값을 정의합니다.  
y = tf.nn.softmax(tf.matmul(x,W)+b)
```

이젠 가중치, 편향에 대해 정의했으니 출력 값을 정의해야 합니다. 출력은 10개의 클래스에 대한 확률 값으로 나와야 합니다. 다시 말해서 총 10개의 출력 값이 필요하고, 그 값들은 0에서 1사이의 값을 가져야 합니다. 또한 총 합은 1이 되는 활성화 함수를 작성해야 합니다.

이때 적절한 활성화 함수는 바로 ‘소프트맥스’입니다. 따라서 텐서플로우에서 제공하는 소프트맥스 함수를 가중치 합에 적용시킵니다.

### # 소프트맥스(SoftMax) 함수

대표적인 활성화 함수이며, 주로 DNN의 마지막 출력 노드에서 분류(Classification)을 사용하고자 할 때 사용됩니다. 여러 개의 시그모이드 값들(0에서 1사이의 값들)을 나눠 합치면 1이 되는 확률로 해석하기 위해 사용한 것입니다.

따라서 MNIST 자동분류기는 출력이 0에서 1의 사이 값을 가져야 하며, 그 합을 1이 되는 결과를 나타내야 했기에 이를 적용한 것입니다.

여기까지 잘 따라왔다면 입력 데이터와 매개 변수(가중치와 편향), 출력 데이터에 대한 정의가 다 됐으니, 이제 그 값을 최적화 시키는 과정이 필요합니다.

그 과정을 적용하기 위해선 손실함수(예: 크로스 엔트로피) 정의와 그 함수의 값을 낮출 방법(예: 경사하강법)에 대해서 고려해야 합니다.

## # 가중치와 편향의 값 최적화 하는 방법

### 1) 손실함수 정의

- 정의: 손실함수란 모델을 통해 얻은 출력 값과 실제 값간의 오차를 나타내는 함수입니다.

예를 들어 MNIST 데이터셋의 경우 숫자 2의 라벨 값은 [0,0,1,0,0,0,0,0,0,0]의 형태의 벡터 값으로 표현되지만 실제 모델의 추론 값은 [0.01, 0.01, 0.90, 0.01, 0.02, 0.01, 0.01, 0.01, 0.01, 0.01]으로 나올 것입니다. 따라서 오차를 측정해야 할 필요가 생깁니다.

오차를 측정하는 방법에는 여러 방법이 있는데 대표적인 예로 선형회귀모델에서 사용하는 평균제곱오차, 유clidean 제곱거리 등이 있습니다. 그러나 본 예제에서는 주로 '크로스 엔트로피 함수(Cross Entropy Function)' 이라는 손실함수를 사용합니다. 아래는 크로스 엔트로피에 대한 식입니다.

$$H_{y'}(y) = - \sum_i y'_i \log(y_i)$$

결론적으로 말하자면 크로스 엔트로피 함수는 모델의 예측 값이 실제 값을 설명하는데 얼마나 비효율적인지를 나타내줍니다. 즉 값이 작을수록 손실이 작아지게 되고 잘 만들어진 모델입니다.

위 식에서  $y'$ 은  $y$ 는 각각 실제 값의 분포와 예측된 확률분포를 의미합니다. 위 표현식을 살펴보면 **실제 값과 예측 값의 차이가 크면 클수록 계산 되는 크로스 엔트로피 값(H)**이 커짐을 알 수 있습니다. 반대로 그 차이가 작을수록 **크로스 엔트로피 값(H)** 역시 작아집니다.

따라서 최적화 과정은 이러한 크로스 엔트로피를 최소화하는 가중치  $W$ 와 편향  $b$ 를 찾는 것을 의미합니다.

### 2) 크로스 엔트로피(손실) 낮추기

앞에서 크로스 엔트로피 값을 줄여야 한다고 언급했습니다. 이를 위해서는 경사 하강법을 사용해야 줄일 수 있습니다. 다시 말해 손실함수를 최소화 할 수 있도록 적절한 가중치  $W$ 와 편향  $b$ 를 구하는 것인데 이를 경사하강법으로 구하는 것입니다.

여기서 경사 하강법이란 함수의 한 점이 기울기 벡터(Gradient)의 반대 방향으로 이동할 때 그 함수 값을 가장 빠르게 감소시킨다는 특성을 이용하여 최소값을 찾아나가는 방법입니다.

이러한 과정을 통해 최소값을 찾게 되면 모델 학습이 완료됩니다.

지금까지의 과정을 코드로 작성하면 다음과 같습니다.

우선 손실 함수를 구하기 위해 실제 값을 정의합니다.

손실 함수를 정의합니다.

**실행 코드**

```
# 손실 함수를 정의합니다.  
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
```

손실 함수의 경우 한 이미지당 10개의 값이 나오고, 실제 학습 시 여러 이미지 데이터가 한번에 입력되기 때문에 각 크로스 엔트로피 값을 합산하고 평균 내는 것이 필요합니다.

마지막으로 손실함수를 최소화 하기 위한 최적화 알고리즘(경사하강법)을 정의합니다.

## 실행 코드

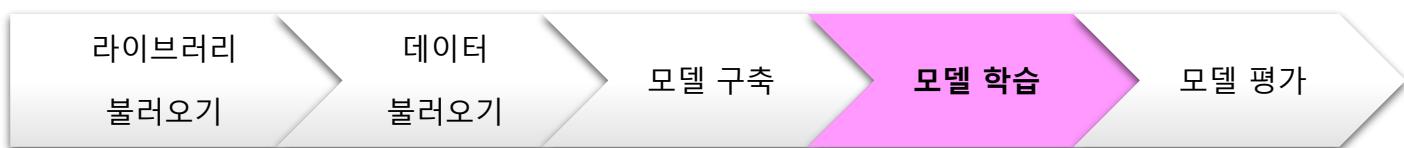
```
# 추정 방법(경사하강법)을 정의합니다.  
train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)
```

여기선 학습 속도가 0.5 인 경사하강법입니다.

이제 모든 알고리즘을 작성했으니 텐서플로우 연산을 실행합니다. 우선 모든 변수를 초기화 해야 합니다.

## 실행 코드

```
# 모든 변수를 초기화 한 후 세션을 시작합니다.  
init = tf.initialize_all_variables()  
sess = tf.Session()  
sess.run(init)
```



이제 훈련데이터를 입력하여 train\_step에서 경사하강법을 통해 매개변수를 산출하는 과정을 충분히 반복하면, 손실 함수를 최소화 하는 매개변수 값을 도출 됩니다. 따라서 모델 학습이 완료됩니다.

## 실행 코드

```
for i in range(1000):  
    batch_xs, batch_ys = mnist.train.next_batch(100)  
    sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
```

루프 내 첫 번째 라인은 훈련 데이터 셋으로부터 무작위로 100개의 데이터를 추출하는 것을 뜻한다. 이를 ‘배치 사이즈가 100인 배치 데이터를 생성’이라고 말합니다.

사실 루프를 반복할 때마다 총 55000개의 전체 데이터 셋을 사용하는 것이 가장 이상적이지만, 그럴 경우 코딩이 너무 무거워져서 속도가 느려지게 되고 컴퓨터에 부담이 가기 때문에 무작위로 적은 수의 데이터를 뽑아내어 학습을 진행합니다.

```
# 이러한 학습을 Stochastic Training이라 합니다.
```

여기서 주목할 점은 이러한 Stochastic Training이 실제 전체데이터 셋을 사용하는 학습과 거의 비슷한 성능을 가지기에 사용한다는 것입니다.



학습이 완료된 모델은 항상 그 성능을 반드시 평가해야 합니다.

MNIST 데이터로 만든 우편번호 자동분류기는 총 10 개의 확률로 출력되는 테스트 데이터 셋의 출력 값에 tf.argmax 함수를 적용하여 가장 높은 확률 값의 라벨을 리턴합니다.

그 후 tf.equal 메서드를 이용해 리턴 된 라벨 값과 실제 추론 값을 비교하여 올바르게 추론 되었는지를 구합니다. 바로 이것이 **모델의 정확도(Accuracy)**를 나타냅니다.

## 실행 코드

```
# 예측값과 실제값을 비교하여 정확도를 출력합니다.  
correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))  
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))  
print(sess.run(accuracy, feed_dict={x: mnist.test.images, y_: mnist.test.labels}))
```

## 출력 결과

```
0.9154
```

두 번째 라인의 tf.cast 는 텐서의 데이터타입을 변경해주는 메서드입니다. 왜냐하면 실제 tf.equal 값은 Boolean(True, False 값만 나타내는)으로 리턴되기 때문입니다.

구체적으로 말하자면 테스트데이터의 총 4 개의 이미지가 있을 때 그 중 3 개만을 정확히 예측했다고 가정합시다. 사용된 tf.equal 의 결과 값은 [True, False, True, True]와 같은 형식으로 나타날 수 있습니다. 여기서 정확도 계산을 위해서 [1, 0, 1, 1]과 같이 변환시켜주는 것이 필요합니다.

이제 feed\_dict 매개변수로 mnist.test.images 와 mnist.test.labels 를 입력하면 정확도가 출력됩니다.

그 정확도는 무려 91%가 됐습니다.

## I. 단일 계층과의 차이점은 뭐예요?

말 그대로 두 개 이상의 은닉 계층이 들어간 것을 의미합니다.

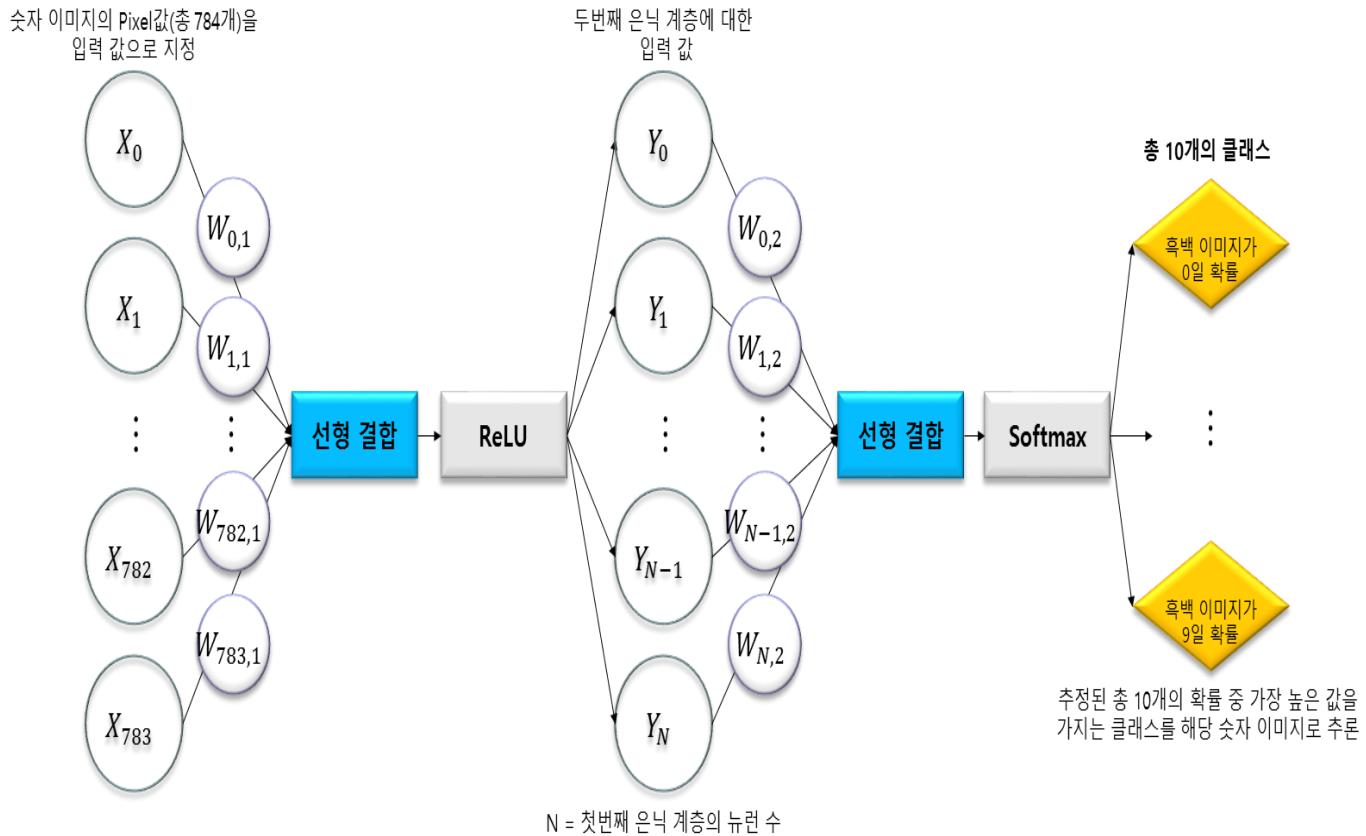
복합계층과 단일 계층과의 차이점이라면 은닉 계층이 더 들어간 것인데, 그렇게 사용하는 목적은 아주 단순합니다. 바로 더 복잡한 모델을 사용하여 정확도를 높이기 위함입니다.

모든 딥러닝 모델이 매우 간단했으면 좋겠지만 사실 그렇지 않습니다. 은닉 계층이 하나만 사용된 경우는 거의 없습니다. 따라서 여기선 은닉 계층에 새로운 계층을 추가하여 단일 계층일 때보다 정확도가 어떻게 향상되는지 알아볼 것입니다.

## II. 이제 구현할 모델링 흐름을 설명해주세요.

단일 계층 모델로 만든 것과 크게 다르지 않습니다. 다만 어떻게 새로운 은닉 계층이 추가됐는지에 대해 중점적으로 파악하시길 바랍니다.

### [DNN 복합 계층 구성도]



## [분석 과정]



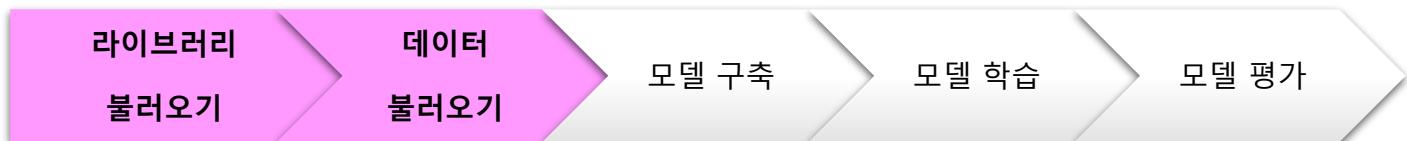
## [분석 환경]

- 운영체제: Ubuntu 14.04 LTS 이상, Centos 7 이상
- 텐서플로우 버전: r0.11 이상 (CPU & GPU 버전)
- 파이썬 버전: 3.5.x

위와 같은 방법으로 구현하겠습니다.

## III. 직접 DNN으로 작성해서 보여주세요.

단일 계층의 DNN은 소프트맥스만 사용했습니다. 그러나 여기서는 은닉 계층 및 ReLu(또 다른 활성화 함수) 계층을 추가하여 정확도를 향상하겠습니다. 얼마나 정확도가 상승했는지를 확인해봅시다.



라이브러리와 데이터를 불러오겠습니다. 여기서 단일계층으로 했을 때와 어떤 차이점이 있는지 보이시나요?

### 실행 코드

```
# 라이브러리 및 데이터를 불러온 후 이미지와 라벨 그리고 가중치의 초기값 설정을 위한 함수를 정의합니다.
import tensorflow as tf
import numpy as np
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)

x = tf.placeholder(tf.float32, [None, 784])
y_ = tf.placeholder(tf.float32, [None, 10])

# weight & bias initialization #####
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)
```

### 출력 결과

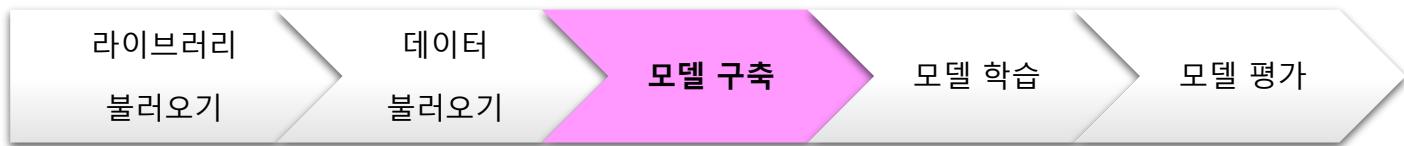
```
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
```

모델 구현에서 그 차이점을 찾아봅시다.

그 첫 번째는 초기값을 설정하는 방법입니다. 앞에서 소개한 것과는 다르게 복합 계층에서는 초기값을 특정 상수로 지정하지 않고 정규분포를 활용하여 무작위로 생성했습니다.

아래는 앞에서 구현한 단일 계층의 가중치와 편향의 초기값 설정 방법입니다.

```
# 단일 계층으로 가중치와 편향을 구했을 때
# 첫번째 은닉 계층의 가중치와 편향을 정의합니다.
W_1 = weight_variable([784,360])
b_1 = bias_variable([360])
```



이제 계층을 하나 더 추가하겠습니다. 이 부분이 단일 계층 DNN과 가장 확연히 차이가 나는 부분으로 은닉 계층은 하나 그리고 ReLU 계층은 두개가 추가되었습니다.

## 실행 코드

```
# 첫번째 은닉 계층의 가중치와 편향을 정의합니다.
W_1 = weight_variable([784,360])
b_1 = bias_variable([360])

# ReLU 함수를 적용합니다.
h_1 = tf.nn.relu(tf.matmul(x,W_1)+b_1)

# 두번째 은닉 계층의 가중치와 편향을 정의합니다.
W_2 = weight_variable([360,180])
b_2 = bias_variable([180])

# ReLU 함수를 적용합니다.
h_2 = tf.nn.relu(tf.matmul(h_1,W_2)+b_2)

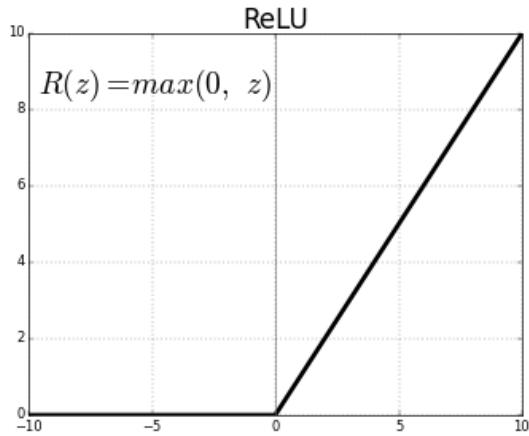
# 세번째 은닉 계층의 가중치와 편향을 정의합니다.
W_3 = weight_variable([180,10])
b_3 = bias_variable([10])

# 소프트 맥스 함수를 적용하여 예측값을 정의합니다.
y = tf.nn.softmax(tf.matmul(h_2,W_3)+b_3)
```

위 코드에서 사용한 활성화 함수 ReLU는 실제 인공신경망 모델링 시 가장 많이 활용됩니다.

### # ReLU 활성화 함수

자 모든 활성화 함수는 출력을 어떻게 할 것인지를 고려한 다음에 정하는 것입니다. 여기서 활성화 함수 ReLU를 왜 썼을까요? 아래 ReLU 그래프로 확인해봅시다.



초기에는 이 함수가 고려된 것은 시그모이드로 구성된 은닉 계층이 여러 개 사용될 경우 Gradient Vanishing 문제가 발생합니다. 이는 Gradient가 여러 번 곱해져서 0.xxxxx 가 (즉, 0에 수렴하게 됩니다.) 되어서 이 문제를 해결하고자 고안됐습니다. 위의 표를 보면 양수는 그 값을 그대로 쓰고 음수는 0으로 표기했습니다. **신경망의 출력 값을 확률(양수 값)로 표현하는 것이 주 목적이기에 ReLU 함수를 사용하는 것이 Gradient Vanishing 문제도 해결하는 등 가장 적절한 함수로 선정됐습니다.**

라이브러리  
불러오기

데이터  
불러오기

모델 구축

모델 학습

모델 평가

이렇게 구현된 복합 계층의 DNN의 모델 학습과 평가는 동일합니다.

## 실행 코드

```
# 손실 함수 및 추정방법(경사하강법) 그리고 정확도 측정을 위한 기타 함수들을 정의합니다.
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

# 모든 변수를 초기화 한 후 세션을 시작합니다.
init = tf.initialize_all_variables()
sess = tf.Session()
sess.run(init)

# 1000 번의 반복 학습 후 최종 예측값 및 정확도를 출력합니다.
for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})

print(sess.run(accuracy, feed_dict={x: mnist.test.images, y_: mnist.test.labels}))
```

## 출력 결과

0.9373

모델 훈련 및 평가 결과 정확도는 약 2% 상승하였습니다.

## I. 만들고자 하는 자동 분류기는 무엇인가요?

여러분은 손글씨 데이터인 MNIST로 우편번호 자동 분류기 구현에 성공했습니다. 이제 비슷한 방법으로 다른 분야에 적용해보겠습니다.

손글씨 데이터로 우편번호 자동분류기가 성공했다면 다른 데이터도 같은 알고리즘을 써서 구현이 가능하지 않을까요? 이번에는 음원 데이터 셋인 GTZAN Genre Collection을 적용하여 구현하도록 하겠습니다.

이 데이터 셋은 음성 처리를 위한 오픈 데이터 셋 중 하나로 총 1000개의 오디오 트랙으로 구성되어 있으며 각 트랙은 30초의 길이를 가집니다. 다시 말해 1000개의 오디오 트랙은 총 10개의 장르로 구성돼 있습니다. 또한 각 트랙은 각 샘플링은 22050HZ의 .wav 형식의 모노 16bit로 구성돼 있습니다.

이 데이터를 다운받기 위해선 링크([http://marsyasweb.appspot.com/download/data\\_sets/](http://marsyasweb.appspot.com/download/data_sets/))에 접속하여 다운로드 합시다.

그래서 받은 데이터 셋의 특징을 살려 우리가 해볼 부분은 이제 DNN으로 음악 장르를 구별하는 것입니다. 여기서 우리는 10개의 장르를 모두 구별하는 알고리즘을 작성하면 좋겠지만 가장 구별이 잘 되는 4개의 장르(Classic, Pop, Metal, Jazz)만을 골라서 자동 분류가 되도록 해보겠습니다.

## II. 음원을 다루기 위한 사전지식이 필요하지 않나요?

네, 맞습니다. 음원 데이터를 알기 위해서는 신호 데이터에 대한 이해가 필요합니다. 딥 러닝 적용을 위해 필요한 만큼 간단히 소개해드리겠습니다.

컴퓨터나 모바일의 음악 플레이어로 음원을 재생할 때를 생각해보세요. 파형(Wave Form)이 그려져 있지 않았나요? 우리는 보통 파형이 보이는 플레이어를 많이 접해봤을 것입니다.



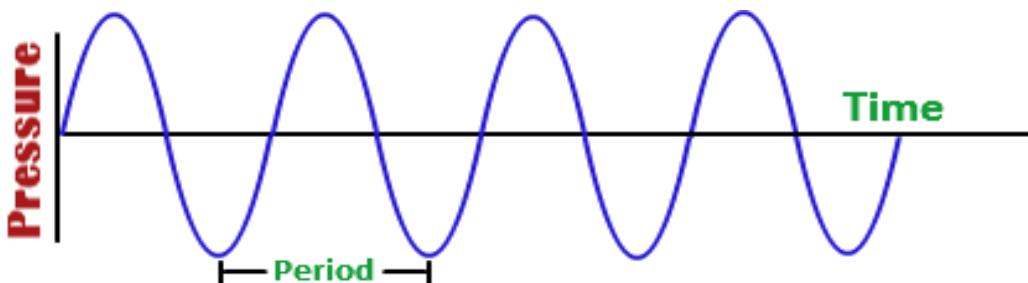
[그림 5] 파형의 예

위 그림과 같은 형태를 나타내는 데이터를 신호 데이터라고 부릅니다. 맞습니다. 우리는 음원 데이터를 이해하기 위해서 신호 데이터에 대한 이해가 필요합니다.

여러분도 학창 시절에 신호 데이터에 대해서 배웠습니다. 그렇지만 기억 안 나시죠? 이제 기억을 상기시켜 드리겠습니다. 관련 용어에 대해서 짚고 넘어가보겠습니다. 주파수, 진폭 등... 이런 것의 정확한 의미는 기억하시나요?

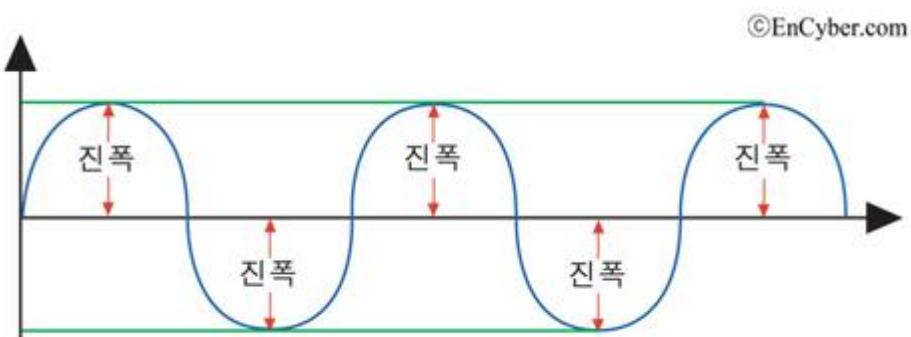
먼저 **주파수(Frequency)**는 무엇일까요? 단위 시간 당 반복 되는 횟수를 말합니다. 단위로는 Hz(헤르츠)라고 표현하고 이는 단위 시간인 1초에 반복되는 횟수를 말합니다. 그럼 이제 사용할 데이터는 22050Hz로 됐다는 말은 1초에 22050번 진동한 샘플 데이터라는 것을 알 수 있겠죠?

다음은 **주기(period)**는 무엇일까요? 이것은 동작이 한 번 이루어질 때까지의 걸리는 시간을 말합니다. 단위는 sec(초)입니다. 당연히 주파수와 연관 있겠죠? 주파수의 역수가 주기입니다. 예를 들어 말하면, 40Hz 주파수의 1회전 진동에 걸리는 시간인 주기를 구하면  $1/40=0.025$  sec 가 되는 겁니다.



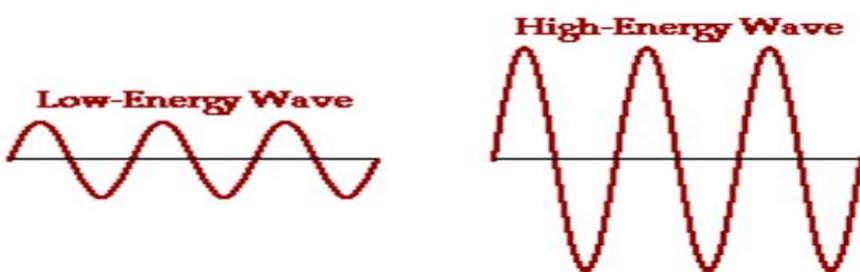
[그림 6] 단순화한 파형의 예

그러면 **진폭(Amplitude)**은 기억하시나요? 신호가 갖는 에너지를 말하며 변위 크기의 극댓값을 말합니다. 이에 대한 설명은 아래 그림이 한 번에 이해시켜줄 것입니다.



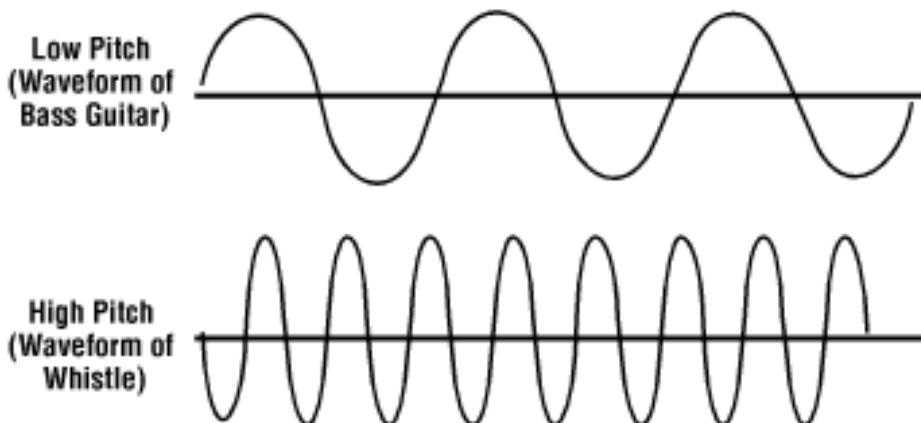
[그림 7] 진폭의 예

그러면 아래 사진을 보고 에너지가 큰 것과 아닌 것은 어떤 차이가 있을까요? 에너지가 큰 파형일수록 진폭의 크기는 무척 커집니다.



[그림 8] 에너지 차이에 따른 파형 차이

여기까지는 신호 데이터에 대한 이해였습니다. 자 음원으로 돌아가서 높은 음을 내는 파형과 낮은 음을 내는 파형은 어떻게 다를까요?



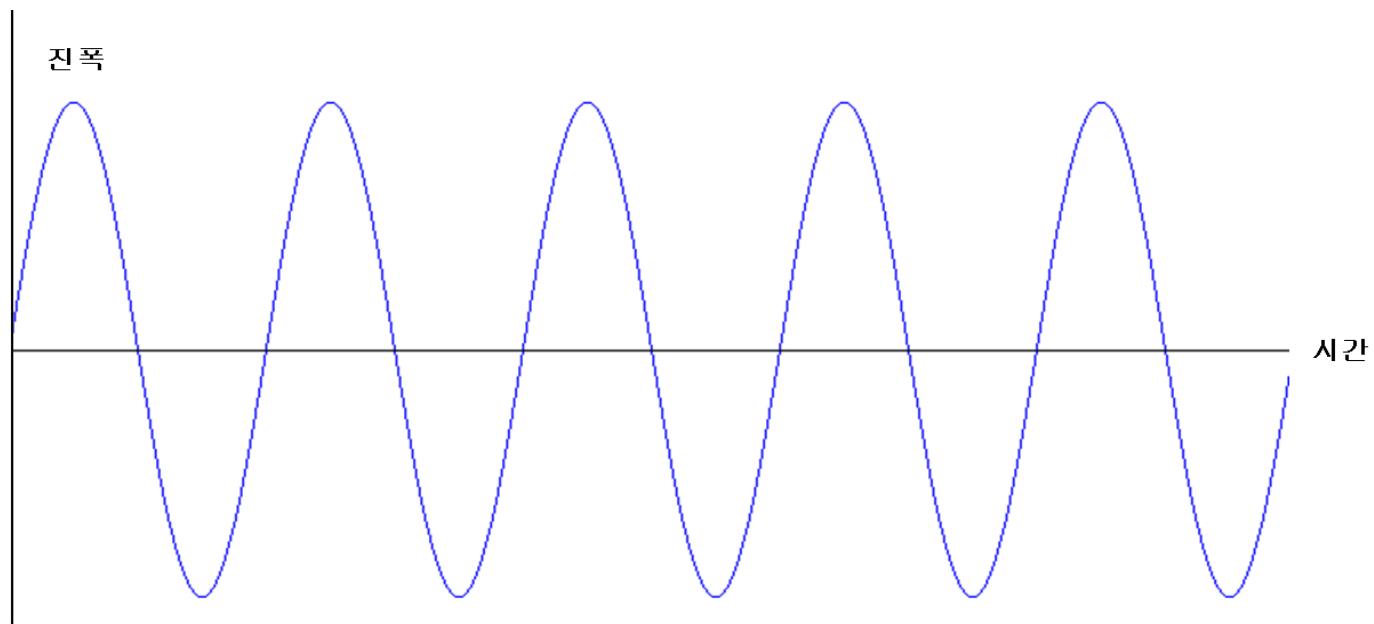
[그림 9] 높은 음과 낮은 음의 파형 차이

높은 음은 단위 시간 동안 반복되는 횟수가 기므로 주기가 짧다고 말할 수 있습니다. 여기서 높은 음과 낮은 음을 통칭하는 말을 '음의 높이'라고 하며 '피치(Pitch)'라고도 부릅니다.

### III. 이제 음원 데이터에 딥 러닝을 적용할 수 있나요?

여러분은 음원 즉 신호 데이터에 대한 가장 기초적인 개념인 주파수, 진폭, 피치 등을 알았습니다. 그럼 이제 DNN을 바로 적용할 수 있을까요? 만약 아니라면 어떠한 추가적인 작업이 더 필요할까요?

결론적으로 말씀 드리자면 바로 적용은 어렵습니다. 그 이유는 음원 즉 신호 데이터의 특성 때문입니다. 신호 데이터는 아래 그림과 같이 "시간-진폭"으로 이루어져 있습니다.



[그림 18] 높은 음과 낮은 음의 파형 차이

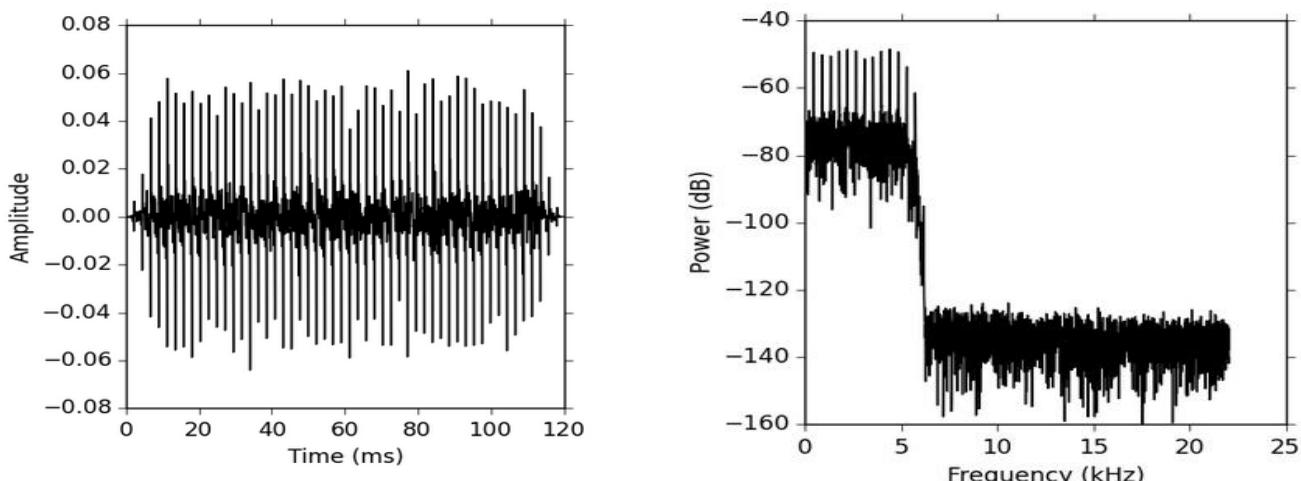
자 이제 그럼 여러분이 “파도 소리”를 녹음하고 있다고 상상해 봅시다. 파도 소리를 녹음하고 있는 도중, 갑자기 소나기가 오면 어떻게 될까요? 당연히 빗방울이 떨어지는 소리까지 함께 녹음될 것입니다. “파도 소리”的 입장에서는 이 “빗방울 소리”는 바로 소음(Noise)입니다. 이렇듯 신호(소리) 데이터에 소음이 추가되면 어떤 일이 발생할까요? “시간-진폭”的 영역에서는 당연히 빗방울이 떨어지는 “시간”에 “진폭”이 크게 흔들릴 것입니다. 또한 여러분들은 이러한 소음이 발생한 시간 구간을 여러분의 녹음 파일에서 제거해야 할 것입니다. 이렇듯 “시간-진폭”的 영역에서의 신호데이터는 외부 소음에 매우 취약함을 알 수 있습니다. 따라서 신호 데이터를 분석하기 위해서는 반드시 ‘전처리’가 이루어져야 합니다.

## IV. 그러면 그 전처리 방법을 알려주세요.

이제껏 음원 데이터에 대한 전처리를 해야 하는 이유에 대해서 알아봤습니다. 이제는 그 전처리 방법인 푸리에 변환(Fourier Transform)과 MFCC에 대해서 알려드리겠습니다.

앞으로 우리가 음원 데이터에 적용할 것은 “시간-진폭”的 영역을 “주파수-에너지”영역으로 바꾸는 변환 작업입니다. 이 변환 작업을 위한 방법론을 “푸리에 변환(Fourier Transform)”이라 부릅니다.

우리는 이 방법으로써 “시간-진폭” 간의 관계를 “주파수-에너지” 간의 관계로 변환할 것입니다. 그럼 어떤 일이 일어날까요? 앞에서 언급한 외부 소음을 적용해 보겠습니다. 이러한 외부 소음이 개입하게 되면 “주파수-에너지” 영역에서는 도메인(Domain) 영역인 “주파수”는 변화가 없습니다. 다만 해당 소음이 가지는 주파수의 에너지 값만 약간 변할 것입니다. 차이가 느껴지시나요? “시간-진폭”的 관계에서는 외부 소음이 추가 되면 그 추가된 만큼 도메인 영역인 “시간”이 바뀌고 또한 해당 시간의 “진폭”이 크게 흔들립니다. 하지만 “주파수-에너지” 영역에선 이러한 일이 발생하지 않습니다.



[그림 19] 푸리에 변환 적용 전/후에 따른 예

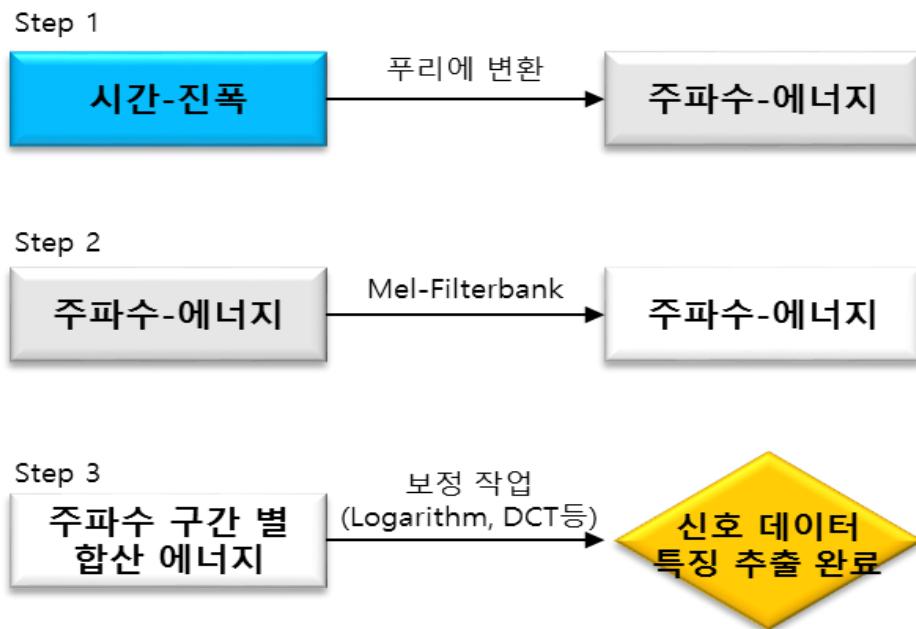
우리는 푸리에 변환을 적용하여 신호데이터를 왜 “주파수-에너지”영역으로 바꾸었습니다. 이제 해당 신호 데이터를 어떻게 분석에 활용하는지에 대해 알아보겠습니다. 일단 신호 데이터가 “주파수-에너지” 영역으로 바뀌게 되면 우리는 많은 것을 할 수 있습니다.

대표적인 예로, 특정 주파수 영역만 선택하여 추출하는 것이 가능해지고 소음 제거와 Onset-Detection 등의 여러 분석이 가능하게 됩니다. 이러한 분석은 “시간-진폭”的 영역에서는 거의 불가능한 것들입니다.

그 외에도 여러가지 분석을 진행 할 수 있지만 그 중에서도 가장 중요한 것은 바로 ‘특징 추출’입니다. 우리는 “주파수-에너지” 영역으로 변환된 신호 데이터에서

**주파수 영역별 합산 에너지를 계산하여 추출할 수 있습니다. 이러한 과정을 우리는 MFCC라 부릅니다.**

이제 우리가 사용할 다른 전처리 방법인 MFCC에 대해서 소개해보겠습니다. MFCC는 Mel Frequency Cepstral Coefficient의 준말로 신호 데이터를 전처리할 때 쓰이는 방법론입니다. 그 과정에 대해 간단히 설명하자면 앞에서 배운 푸리에 변환을 통해 신호 데이터를 “주파수-에너지” 영역으로 변환 후 특정 필터(Mel-FilterBank)를 거쳐 주파수를 구간을 나누고 해당 구간 별 에너지를 합산을 나누게 됩니다. 이제 합산 된 에너지에 대한 보정 작업을 거쳐 최종적으로 “주파수 구간 별 합산 에너지”를 추출하게 됩니다. 다시 말해 MFCC를 통해 신호 데이터의 특징을 추출할 수 있습니다.



[그림 20] MFCC 추출 과정

여기서 질문입니다. 이렇게 추출된 신호 데이터의 특징들은 어디서 사용될까요?

바로 딥 러닝 모델링을 위한 입력값으로 활용이 됩니다. 말 그대로 신호 데이터의 “특징” 이기 때문에 모델링을 위한 아주 좋은 재료가 됩니다. 이제 이러한 기법들을 활용하여 음원 데이터에 대해 딥 러닝을 적용해 보겠습니다.

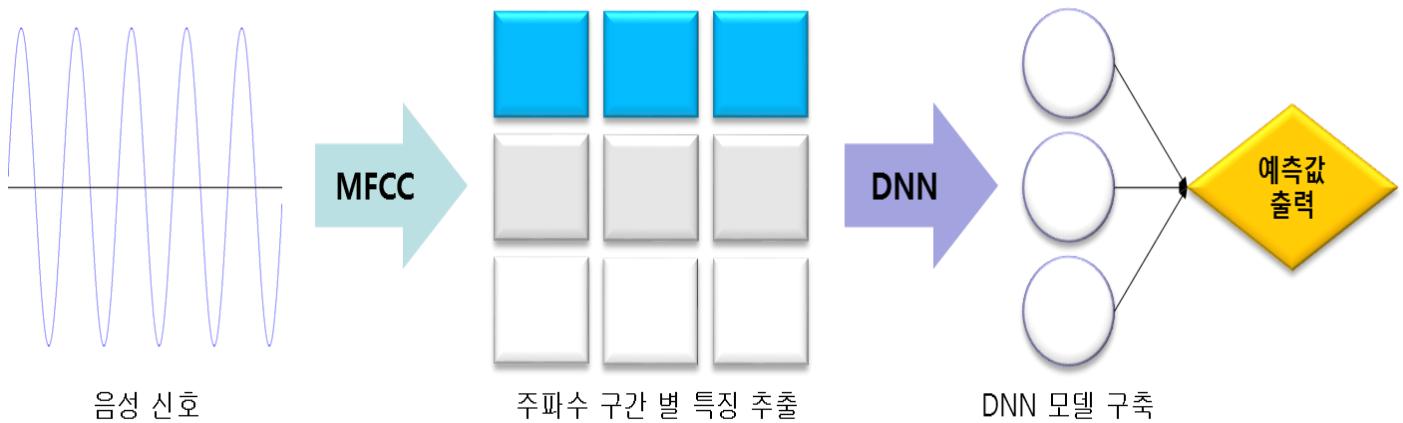
## # Onset-Detection

여러분 놀음할 때를 생각해보세요. 우리는 아무 말도 하지 않았는데도 파형이 말을 시작하기 전에도 있다는 것을 알 수 있죠? 그 파형에서 유의미한 부분의 시작점(실제 말한 부분)을 찾는 것을

## V. 이제 구현할 모델링 흐름을 설명해주세요.

이제는 직접 구현할 모델링의 흐름에 대해 설명하도록 하겠습니다.

### [MFCC를 활용한 DNN 구조]



[그림 10] MFCC 추출 과정

### [분석 과정]



### [분석 환경]

- 운영체제: Ubuntu 14.04 LTS 이상, CentOS 7 이상
- 텐서플로우 버전: r0.11 이상 (CPU & GPU 버전)
- 파이썬 버전: 3.5.x

## VI. 직접 DNN으로 작성해서 보여주세요.

GTZAN Genre Collection 데이터에 대해 DNN 모델을 구현해서 음악 장르를 분류할 수 있도록 만들시다.



이제 400개의 음원 데이터로 음악의 장르를 구분하는 모델을 구현 할 것입니다.

주목할 점은 복잡한 인공지능신경망(DNN, CNN)을 적용하지 않을 것입니다. 그 이유는 이미 푸리에 변환과 MFCC를 통해 유의미한 특징들을 추출하였기 때문에 간단한 DNN으로도 충분히 높은 정확도의 모델 구현이 가능합니다.

## 실행 코드

```
# 라이브러리를 불러옵니다.  
from pylab import*  
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
```

먼저 필요한 라이브러리를 불러옵니다. 여기서 중요한 라이브러리는 푸리에변환을 위한 `scipy.fftpack`과 음원 파일 형식인 `wav` 파일을 읽고 변환하기 위한 `scipy.io.wavfile` 입니다.



해당 라이브러리로 `wav` 파일을 시간\*진폭으로 이루어진 배열(Array)형식으로 변환합니다. 분석을 위해 각 음원 파일에 해당하는 라벨(예: 장르가 'pop'인 경우 [1,0,0,0] 등)도 마찬가지로 생성하도록 합니다.

## 실행 코드

```
# WAV 파일을 읽은 후 Array 형태로 변환합니다.  
wav_dir = '/home/eduuser/NN/Data/genres/'  
  
wav_labels = os.listdir(wav_dir)  
x = []  
y = []  
  
for wav_label in wav_labels:  
    for wav_f in os.listdir(wav_dir+wav_label):  
        SampleFreq, f = wvf.read(wav_dir+wav_label+'/'+ wav_f)  
        N_total = len(f)  
        N_cut = SampleFreq*5  
        audio_data = f[N_cut:N_total-N_cut]  
        N = len(audio_data)  
        x.append(audio_data)  
        if wav_label == 'pop':  
            y.append([1,0,0,0])  
        elif wav_label == 'classical':  
            y.append([0,1,0,0])  
        elif wav_label == 'jazz':  
            y.append([0,0,1,0])  
        else:  
            y.append([0,0,0,1])  
  
music = np.array(x) / (2.**15)  
label = np.array(y)
```

이제 각 음원 파일에 대해 MFCC 를 적용합시다.

## 실행 코드

```
# MFCC 라이브러리를 불러옵니다.
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))

# MFCC 추출을 위한 파라미터 값을 정의합니다.
samplerate = SampleFreq
winlen = 0.02
winstep = 0.01
numcep = 15
nfilt = 26
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)

dnn_feat = []

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

# MFCC 를 추출합니다.
for i in arange(len(music)):
    mfcc_feat = mfcc(music[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_mean = np.mean(mfcc_feat, axis = 0)
    feat_cov = np.cov(mfcc_feat, rowvar=0)
    feat_cov_up = np.triu(feat_cov, k=0)
    feat_flat = feat_cov_up.flatten()
    index = arange(len(feat_flat))[feat_flat==0]
    feat_cov = list(np.delete(feat_flat, index))
    feat = []
    feat.extend(feat_mean)
    feat.extend(feat_cov)
    dnn_feat.append(feat)

dnn_feat = np.array(dnn_feat)
```

MFCC를 구하기 위해 `python_speech_features` 라이브러리의 `mfcc` 메서드를 활용합니다.

**# `python_speech_features` 라이브러리의 `mfcc` 메서드**

해당 메서드는 신호의 MFCC를 구할 수 있는 함수입니다.

주요 파라미터는 `winlen`(프레임의 길이), `winstep`(프레임 간격) 등이 있습니다.

자세한 내용은 다음 주소를 참조 ([https://github.com/jameslyons/python\\_speech\\_features](https://github.com/jameslyons/python_speech_features)).

이렇게 구해진 MFCC는 신호의 구간별 주파수 영역별 특징을 의미합니다.

하지만 이러한 구간별 특징을 모두 활용하기에는 그 수가 너무 많다는 게 문제입니다. 그래서 각 주파수 영역별 특징만 남을 수 있도록 주파수 영역 값을 기준으로 구간에 대한 Mean Vector와 Covariance-Matrix를 구해야 합

니다. 이렇게 구해진 Mean과 Covariance값들은 Signal의 주파수 영역간 특징으로 나타나게 됩니다.



이제 음악 장르 판별을 위한 복합 계층의 DNN을 구현합시다.

먼저 모델 훈련을 위해 훈련데이터와 테스트데이터를 7:3의 비율로 나눈 후 입력 값(x)과 출력 값(y)을 placeholder로 지정해줍니다.

그 후 모델 구현은 비교적 간단하게 이루어집니다.

### 실행 코드

```
# DNN 구축
import tensorflow as tf

# 추출된 MFCC 값을 학습 및 테스트 데이터로 분리합니다.
N_music = len(dnn_feat)
N_train = int(ceil(N_music * 0.70))
idx = np.random.choice(N_music, N_train, replace=False)
x_train = dnn_feat[idx]
y_train = label[idx]

test_idx = np.delete(arange(N_music), idx)
x_test = dnn_feat[test_idx]
y_test = label[test_idx]

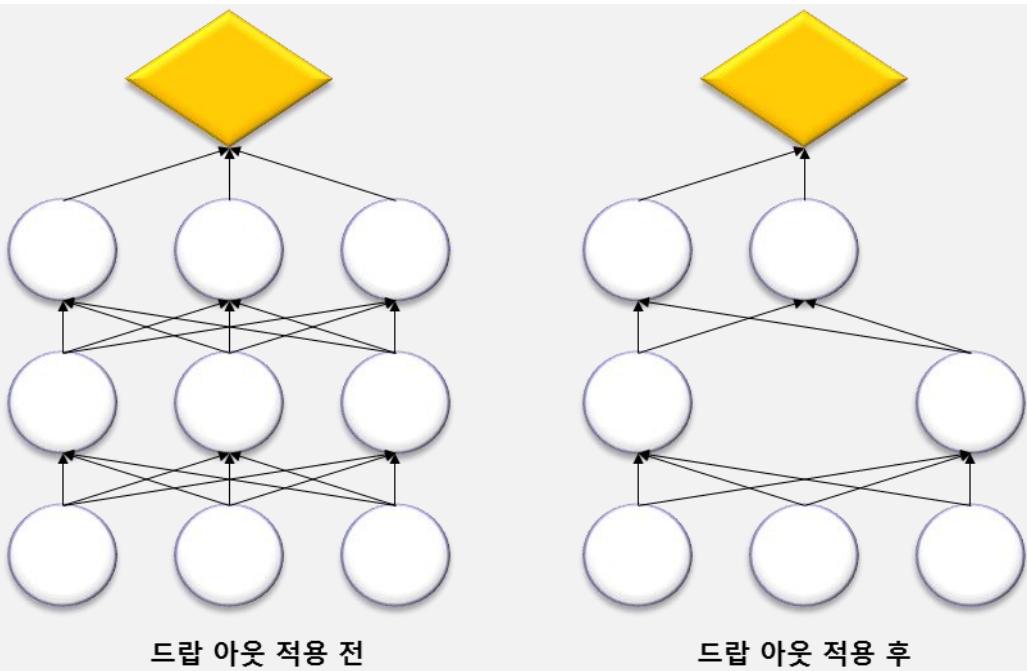
num_feat = len(dnn_feat[0,:])
num_classes = len(label[0,:])

# 입력값 및 라벨값을 정의합니다.
x = tf.placeholder(tf.float32, [None, num_feat])
y_ = tf.placeholder(tf.float32, [None, num_classes])
```

단일 계층의 DNN과 마찬가지로 가중치(W)와 편향(b)의 초기값을 지정하고 첫 번째의 은닉계층, ReLU 계층, 두 번째 은닉 계층, 소프트맥스 계층까지 지정해주면 모델이 완성됩니다. 그러나 여기서 드롭 아웃이 추가됩니다.

#### # 드롭 아웃(DropOut)

과적합(OverFitting: 훈련 데이터 셋에 너무나도 맞게 잘 나오는 현상)을 막기 위해서 고안된 방법입니다.



[그림 11] 드롭아웃 적용 전 후

그림의 왼쪽처럼 뉴런의 수가 많아서 '과적합'이 우려될 때 모델 훈련을 바꿔줍니다. 즉 모델 훈련 시 특정 확률로 가중치를 무작위로 0으로 만들어주는 방법을 택합니다. 이 방법으로 모델 훈련 하는 중 가중치의 수가 줄어들게 되고, 과적합 가능성 또한 줄어들게 됩니다.

## 실행 코드

```
# 가중치와 편향의 초기값 설정을 위한 함수를 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)

# 첫번째 은닉 계층의 가중치와 편향을 정의합니다.
num_neuron = 25

W_1 = weight_variable([num_feat, num_neuron])
b_1 = bias_variable([num_neuron])

# 드롭아웃을 정의합니다.
keep_prob = tf.placeholder("float")
W_1_drop = tf.nn.dropout(W_1, keep_prob)

# ReLU 함수를 정의합니다.
h_1 = tf.nn.relu(tf.matmul(x,W_1_drop)+b_1)

# 두번째 은닉계층의 가중치와 편향을 정의합니다.
W_2 = weight_variable([num_neuron, num_classes])
b_2 = bias_variable([num_classes])

y = tf.nn.softmax(tf.matmul(h_1,W_2)+b_2)
```



이제 드롭아웃까지 적용했으니 모델 훈련과 평가를 실시합시다.

## 실행 코드

```
# 모델 학습 및 테스트를 진행하고 정확도를 출력합니다.
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

sess = tf.Session()

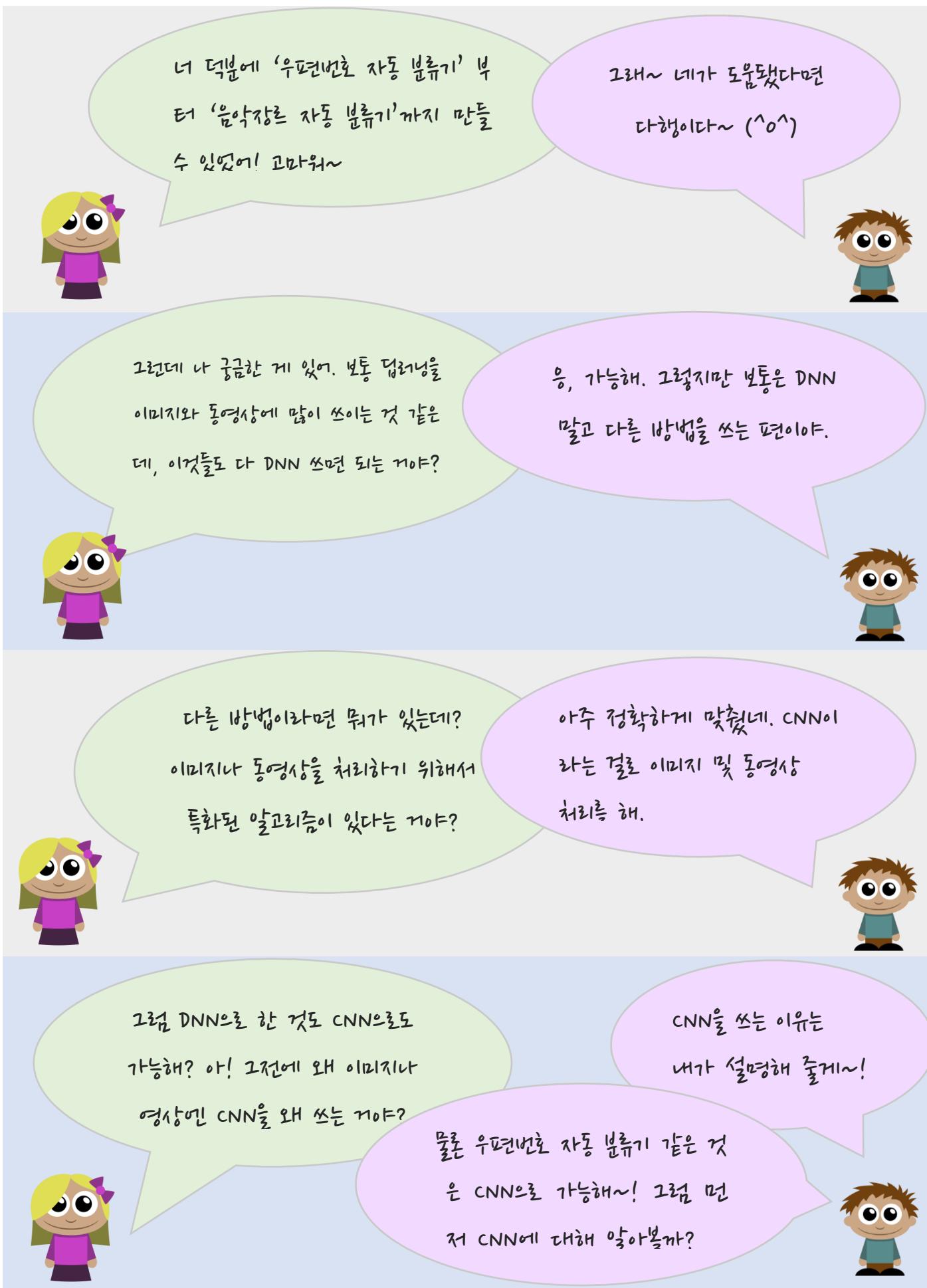
sess.run(tf.initialize_all_variables())
for i in range(3000):
    if i%100 == 0:
        train_accuracy = sess.run(accuracy, feed_dict={x: x_train, y_: y_train, keep_prob: 1.0})
        print("step %d, training accuracy %g"%(i, train_accuracy))
        train_cost = np.exp(sess.run(cross_entropy, feed_dict={x: x_train, y_: y_train, keep_prob: 1.0}))
        print("step %d, training cost %g"%(i, train_cost))
    sess.run(train_step, feed_dict={x: x_train, y_: y_train, keep_prob: 0.5})

print("test accuracy %g"% sess.run(accuracy, feed_dict={x: x_test, y_: y_test, keep_prob: 1.0}))
```

## 출력 결과

```
step 7900, training accuracy 0.964286
step 7900, training cost 2.80408
test accuracy 0.93051
```

위 코드에서 볼 수 있듯이 드롭아웃을 위해 모델 훈련 간 0.5의 확률을 지정하였습니다. 모델 평가 결과 정확도는 약93%로 높은 수치를 기록했음을 알 수 있습니다.



# CONTENTS

## 딥러닝 시작하기(2)

### 2단계

[기초]

CNN 이해 p41



[비교]

왜 CNN으로 만들면 더 효과적인가요? p48



[실습]

CNN을 적용한 우편번호 손글씨 자동분류기 p50



[활용]

컬러 이미지 자동분류기 p55

(부제: Cifar-10 이미지 데이터로)



열어서 2단계까지 왔네요!

CNN과 DNN을 비교해보자

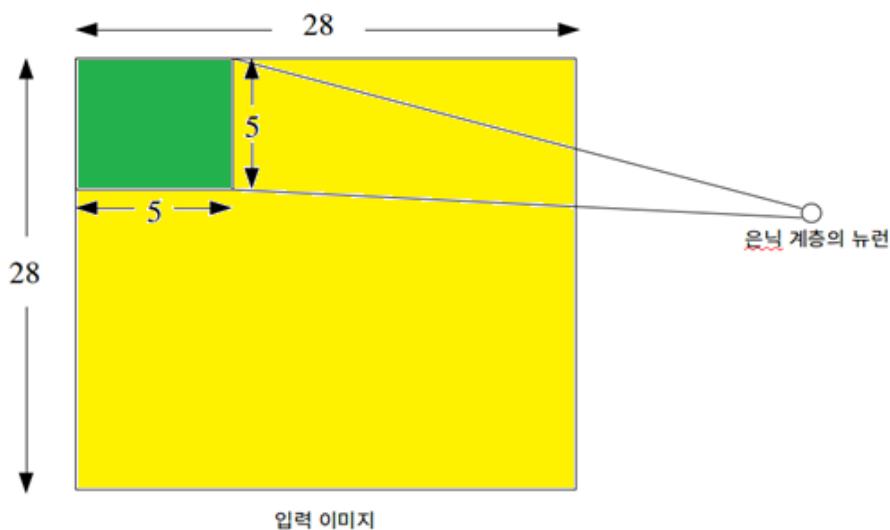
## I. CNN은 어떤 알고리즘이죠?

이제 CNN이 탄생한 배경에 대해서 알아봤으니 CNN(혹은 ConvNet)의 원리에 대해 알아보겠습니다.

우리가 했던 손글씨 데이터인 MNIST로 DNN을 구현했던 것을 떠올려봅시다. 숫자 이미지는 28 pixel로 정규화된 정사각형입니다. 그리고 이것을 784의 크기를 가진 1 차원 벡터로 보는 것으로 시작했습니다. 그런데 꼭 그렇게 봐야하는 것일까요? CNN은 이러한 의문에서부터 시작합니다.

아직은 복잡하게 돼있는 이미지 전체를 한번에 인식하는 일은 어렵습니다. 대신 이러한 방법을 고안했습니다. 바로 CNN을 풀어 쓴 말인 Convolutional Neural Network에서 따온 Convolution이라는 개념입니다. Convolution은 한국어로 합성곱이라고 하며, CNN을 합성곱 신경망이라고 부릅니다. 그런데 합성곱이라는 그 자체의 의미로는 하나의 함수와 또 다른 함수를 반전 이동한 값을 곱한 후에 각 구간을 적분하여 새로운 함수를 만드는 수학 연산자입니다. 이름에서 따온 것처럼 CNN은 어떤 구간(입력 데이터의 구간)에 대해 필터(새롭게 적용할 함수)를 적용하여 새로운 결과(합성곱 된 결과)를 만들어줍니다.

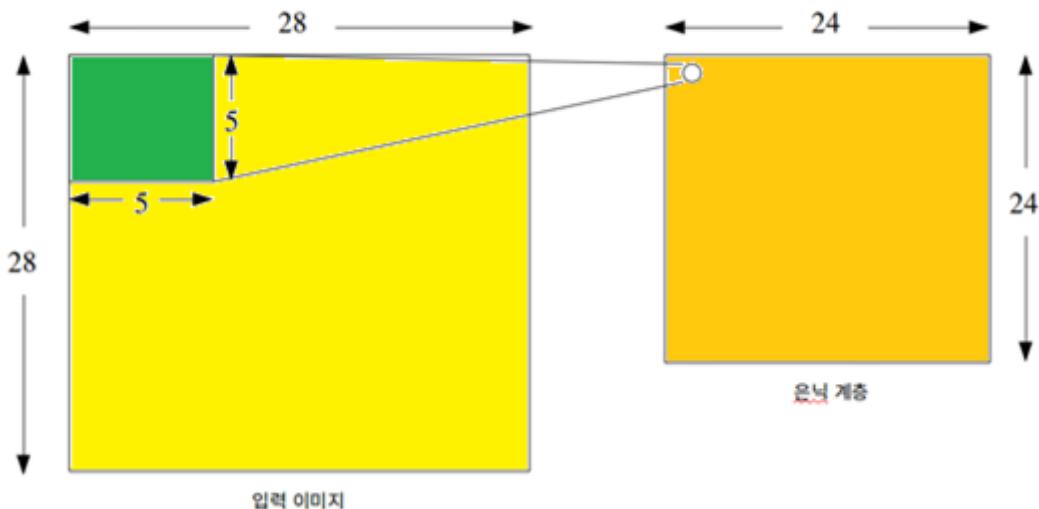
위와 같은 적용을 할 수 있는 이유는 바로 CNN은 입력 이미지의 전체가 은닉 계층의 뉴런에 모두 연결되어 있지 않다는 것입니다. 그게 무슨 말일까요? 자, 여기 모눈종이 위에 색칠된 28X28 pixel 크기의 컬러 이미지가 있습니다. 우리는 이것을 입력 계층에 넣을 것이지만 전체를 한번에 다 넣거나, 이를 1 차원 배열로 생각할 수 없습니다. 대신 입력 이미지를 나누는 작업을 수행할 것입니다.



[그림 12] 입력 이미지를 훑는 윈도

어떻게 하냐고요? 일단 입력 이미지를  $5 \times 5$ 의 크기로 나눈다고 가정합시다. 모눈종이에 색칠된 이미지에서 5 픽셀 크기로 줄을 긋는 것이 아닙니다. 다른 모눈종이에  $5 \times 5$  픽셀만큼 색칠된 초록색 이미지를 따로 만들어서, 입력 계층에 넣을 원본 이미지(노란색)위에 올립니다. 그러면 위의 그림처럼 노란색 종이와 초록색 종이가 포개질 것입니다. 이때 노란색 모눈종이에서 1pixel 크기만큼 초록색 종이를 조금씩 움직여줍니다. 그러면 그 초록색 종이만큼 매칭되는 노란색 이미지가 있을 것입니다. 바로 이런 식으로 입력 이미지를 나누는 것입니다. 그 초록색 종이는 1 pixel 크기만큼 이동하기에 오른쪽으로 23 번을, 아래쪽으로 23 번 움직이는 것이 가능하게 됩니다.

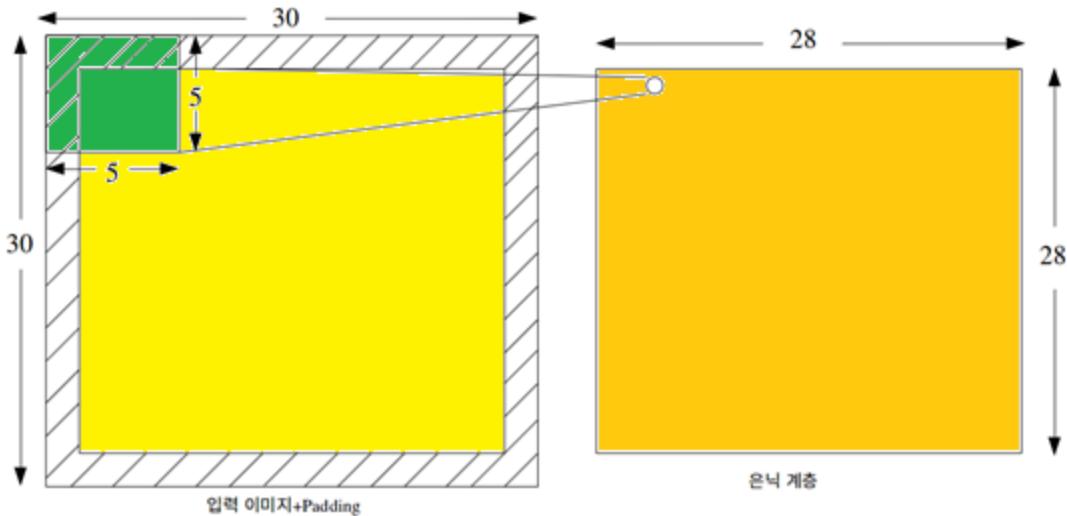
잘 따라오셨나요? 이 방법을 통해 테두리, 선, 색 등의 시각적 특징을 감지하는데 아주 효과적입니다. 다시 위 그림을 봅시다. 노란색 종이는 입력하고자 하는 이미지 원본입니다. 그리고 그것을 초록색 종이에 다시 매칭되는 노란색 부분 여러 개를 만들 수 있습니다. 여기서 초록색 종이는 입력 이미지 전체를 훑고 지나가는 개념이기에 흔히 윈도(Window)라고 부릅니다. 이렇게 분할된 이미지를 데이터를 처리할 수 있는 은닉 계층의 뉴런과 연결해야겠죠? 그 뉴런은 매칭된 노란색 부분인  $5 \times 5$  pixel 크기의 이미지만 연결됩니다. 그래서 은닉 계층은 총  $24 \times 24$  만큼 생성됩니다. 아래 그림을 보면 이해하실 수 있을 겁니다.



[그림 13] 윈도가 훑고 지나간 은닉 계층

여기까지 보면 초록색 종이, 즉 윈도를 1pixel 크기만큼 이동했는데 반드시 그래야만 할까요? 당연히 아니겠죠. 이 크기도 조절할 수 있습니다. 얼만큼 움직일 수 있는지에 대한 단위를 정하는 매개변수를 스트라이드(Stride)라고 부릅니다.

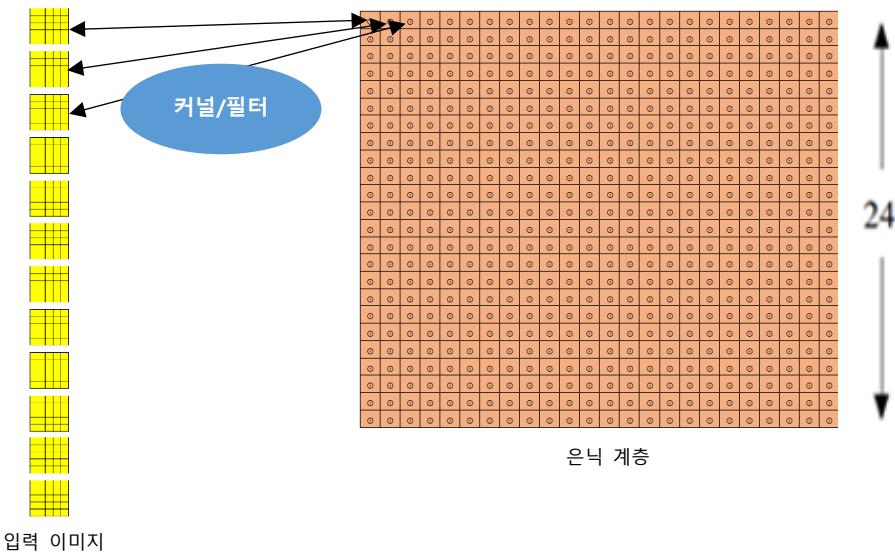
그런데 말이죠. 입력 이미지는  $28 \times 28$ 의 크기였는데 윈도로 훑는 작업을 거치고 나니  $24 \times 24$  만큼 밖에 되질 않았습니다. 더 좋은 성능을 내기 위해선 원래 이미지 입력과 같은 크기가 되는 것이 더 나을 것입니다. 그래서 윈도를 이미지의 맨 끝부터 시작하지 않고 이미지 바깥쪽부터 움직이게 해서 훑게 하는 방법도 있습니다. 아래 그림을 보면 알 것입니다. 원본이  $28 \times 28$ 인데 테두리에 2 pixel이 더 있다고 가정하고 윈도를 훑었습니다.



[그림 14] 패딩을 적용한 결과

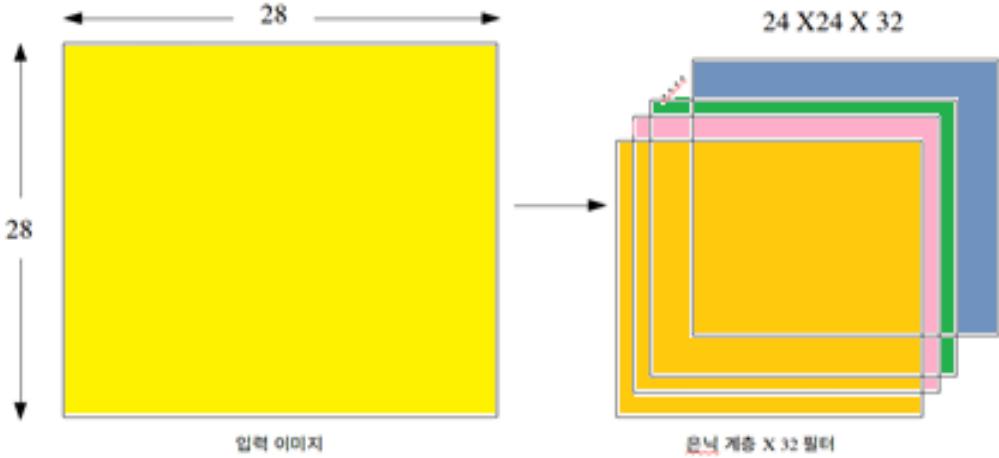
그랬더니 원래의 이미지 크기만큼 은닉 계층도  $28 \times 28$  크기가 되었습니다. 이것을 하는 이유는 정보의 축소를 막기 위함입니다. 정보 축소를 막기 위해 2pixel에 허수를 채워주는데, 보통 0을 채워 테두리를 추가합니다. 또한 스트라이드처럼 테두리의 크기도 정할 수 있는 매개변수가 있는데 이를 패딩(Padding)이라고 부릅니다.

이제 데이터 처리를 위해 이것들을 연결해야 합니다. DNN에서 배웠던 선형 모델을 적용할 때가 됐습니다. 여기서  $5 \times 5$  만큼의 가중치 행렬  $w$ 와 편향  $b$ 가 필요합니다. 여기서 CNN의 특이점이 있습니다. 은닉 계층의 모든 뉴런은 가중치 행렬  $w$ 와 편향  $b$ 를 모두 공유한다는 것입니다.



[그림 15] 은닉 계층의 뉴런과의 관계

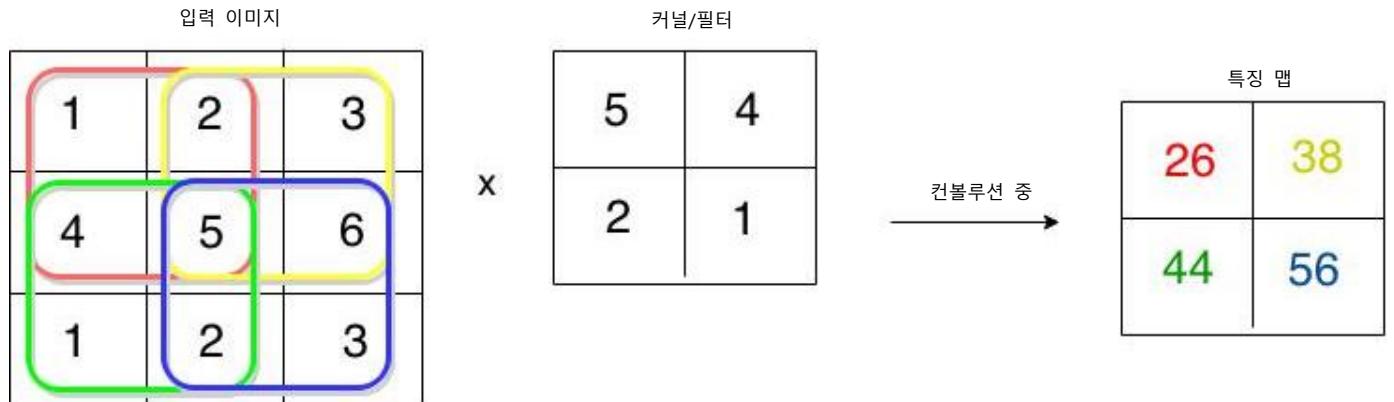
위 그림은 은닉 계층의 뉴런 하나가 입력 이미지의  $5 \times 5$  으로부터 출력되는 관계를 나타냈습니다. 다시 말해 이런 관계가 총  $24 \times 24$  만큼 있다는 것을 의미하며, 그 데이터를 처리하는 관계를 구현하기 위해서는 선형 모델이 필요합니다. 이때 사용되는 가중치  $w$ 와 편향  $b$ 는 은닉 계층의 모든 뉴런의 공통이므로 이 관계를 유지하기 위해 사용되는 **매개변수는  $24 \times 24$  가 아닌  $5 \times 5$**  입니다. (만일 공유하지 않는다면 가중치 행렬의 크기를 포함한  $24 \times 24 \times 5 \times 5$  가 될 것입니다.)



[그림 16] 적용할 필터만큼 생성된 은닉 계층

정확히 말하자면 그 관계를 처리하기 위해서 사용되는 **가중치 w** 와 **편향 b**로 이뤄진 것을 **커널(Kernel)** 혹은 **필터(Filter)**라고 부릅니다. 그리고 각 커널은 이미지에서 고유 특징을 찾는데 사용되며, 그 커널로 얻게 된 한 종류의 특징을 **특징 맵(Feature Map)**이라고 합니다. 보통 감지하고 싶은 특징을 여러 개 만드는 것이 보통입니다. 만약에 32 개의 필터로 특징 맵을 만든다면 위와 같은 그림일 것입니다. 이제 겨우 첫 번째 은닉 계층을 완성했습니다. 이렇게 특징 맵들로 구성된 것을 합성곱 계층이라고 부릅니다.

그렇다면 필터는 어떤 방법을 통해서 특징 맵을 추출할 수 있을까요? 아래 그림으로 설명해드리겠습니다.



[그림 17] 커널/필터 연산

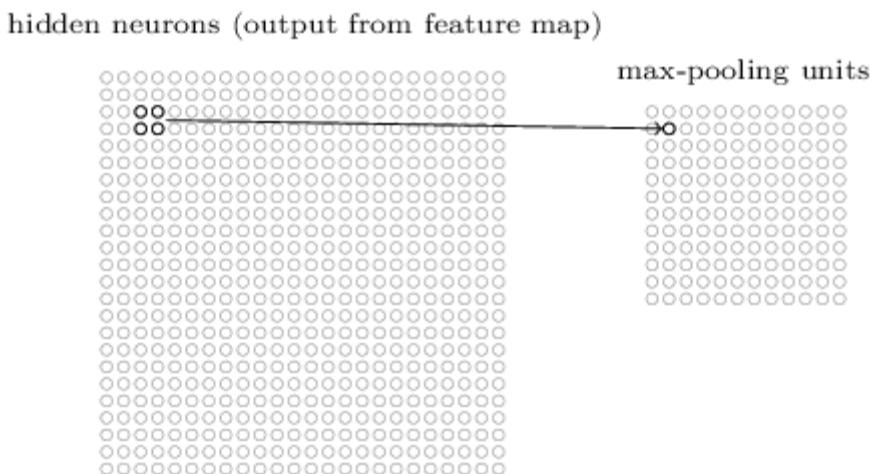
예를 들어 입력 이미지의 값 중에서 우선  $\begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix}$  부분만 필터 연산을 하겠습니다. 필터 연산은 그저 매트릭스 계산 방법과 동일합니다. 따라서  $\begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix} \times \begin{bmatrix} 5 & 4 \\ 2 & 1 \end{bmatrix}$ 을 한 값입니다. (특징 맵의 26을 말합니다.  $26 = 1 \times 5 + 2 \times 4 + 4 \times 2 + 5 \times 1$ ) 적절한 특징 맵을 추출하기 위해서 필터 값을 지정하는 것은 중요한 일입니다. 필터 값의 예로는 아래와 같은 것들이 있습니다.

Operation	Filter	Convolved Image	Operation	Filter	Convolved Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$		Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$			$\frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$			$\frac{1}{16} \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

[그림 18] 필터의 종류 및 그에 따른 연산 결과

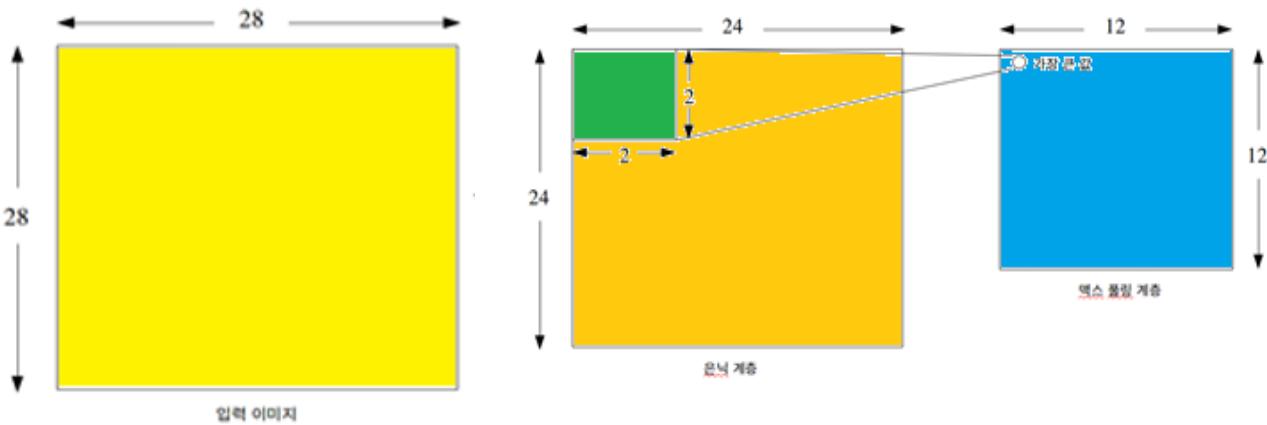
잘 따라오셨나요? 페이지를 4 쪽이 넘도록 설명했지만 아직 CNN의 매커니즘에 대해 다 말씀드리지 못했습니다. 노파심에서 말씀드리는 거지만, 합성곱 계층은 이미지 처리를 위해서 만들 여러 은닉 계층 중에서의 첫번째 은닉 계층의 구체적인 이름입니다. 이제 다음 은닉 계층인 풀링 계층(Pooling Layer)에 대해서 알아봅시다.

일반적으로 풀링 계층은 합성곱 계층 다음에 뒤 따라옵니다. 이것은 합성곱 계층에서의 특징 맵들의 차원을 축소하여 가장 중요한 부분만을 남기고 압축을 합니다. 보통 최대값(Max Pooling)을 남기거나, 평균값(Mean Pooling)을 남기는 방법, 그 합을 구하는 방법(Sum Pooling)이 있습니다.



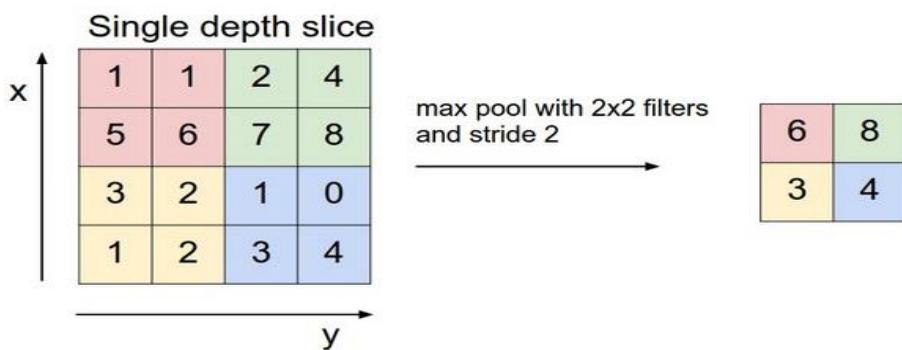
[그림 19] 특징 맵에서 맥스 풀링한 결과

위 그림에서 보면 합성곱 계층의 특징 맵에서 맥스 풀링(Max Pooling)하여 축소되는 과정을 보여줍니다. 이 방법은 특징 맵의 2x2 크기에서 가장 큰 값을 선택하여 정보를 압축합니다. 이 과정은 여러 개의 필터로 만들어진 특징 맵들에 대해 각각 풀링을 적용해주는 작업을 반복합니다. 여기서 잠깐! 이 과정은 일정한 크기로 나눌 때 윈도(Window)처럼 훑고 지나가는 것이 아니라 모눈 종이에 2x2 씩 줄을 긋는다고 생각하면 됩니다.



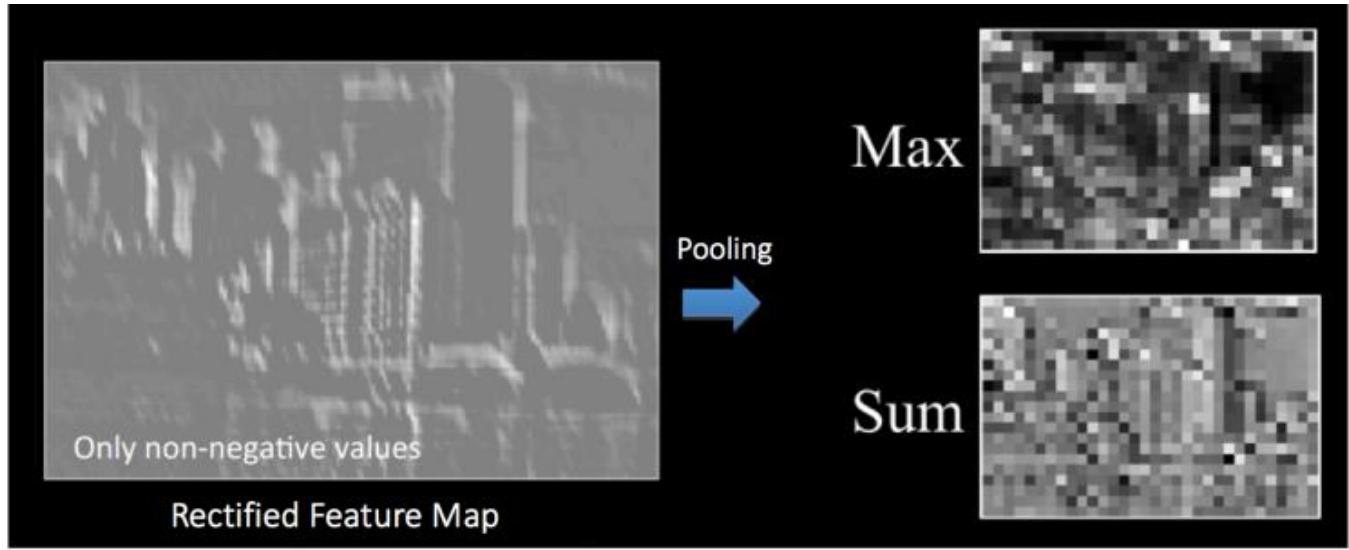
[그림 20] 입력 계층 → 합성곱 계층 → 맥스 풀링 계층에 대한 과정

따라서 위의 그림에 나타나 있는 과정에서 알 수 있듯이, 합성곱 계층에서 맥스 풀링을 적용하면 그 크기가  $12 \times 12$ 로 되는 것을 확인할 수 있습니다. 또한 필터가 적용된 특징 맵 만큼 적용되어야 하므로 풀링 계층의 뉴런은 총  $12 \times 12 \times 32$  가 됨을 확인할 수 있습니다.



[그림 21] 맥스 풀링 되는 과정

구체적으로 풀링되는 과정에 대해 간단히 알아보겠습니다. 위 그림에서 특징맵의 일부가 왼쪽과 같다고 가정합시다. 그리고 풀링으로 압축하기 위해선  $2 \times 2$ 로 나눠야 하므로  $[1,1,5,6]$ ,  $[2,4,7,8]$ ,  $[3,2,1,2]$ ,  $[1,0,3,4]$ 로 나눴습니다. 그 중에서 맨 처음의 구간인  $[1,2,5,6]$ 을 맥스 풀링하면 가장 큰 값은 6이 됨을 알 수 있습니다. 이 과정을 거쳐 풀링된 결과는  $[6,8,3,4]$ 가 됩니다.



[그림 22] 동일한 특징 맵에서 맥스 폴링과 합 폴링에 대해 적용한 결과 비교

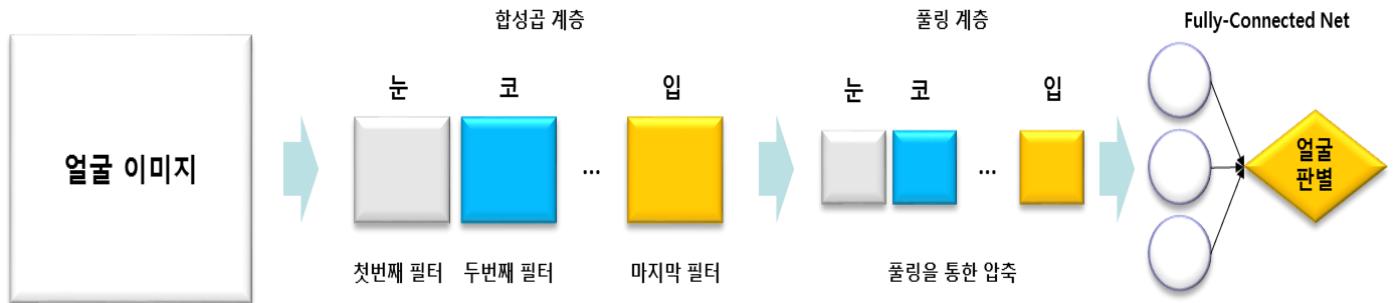
그렇다면 압축을 왜 하는 것일까요? 위 그림으로 간략히 설명 드리겠습니다. 그림의 왼쪽의 특징 맵을 통해 폴링을 각각 다르게 적용한 결과입니다. 맥스 폴링과 합계 폴링을 적용했네요. 맥스 폴링은 지정된 구간에서의 가장 큰 값을 선택해서 압축을 합니다. 이는 특징의 정확한 위치 보다는 그 구간에서의 패턴의 민감성을 나타내 줍니다. 반대로 합계 폴링(Mean Pooling과 비슷합니다.)은 정해진 구역에서의 패턴의 평균 값을 알게 해줍니다.

이렇게 입력된 이미지에 대해서 합성곱 계층과 폴링 계층을 거친 은닉 계층이 완성됐습니다. 이제 DNN 과 마찬가지로 생성된 계층의 뉴런에 완전히 연결(Fully-Connected)한 후 결과값을 얻어낼 수 있게 됐습니다.

## I. Why CNN?

왜 이미지나 영상에 DNN이 아닌 CNN을 적용하면 더 효과적일까요? 그 해답은 이미지의 용량 즉 크기와 관계 있습니다.

이미지 데이터는 그 용량이 매우 큽니다. 우리가 앞에서 배웠던 흑백 숫자 이미지인 Mnist의 경우도 이미지 하나당  $28 \times 28 = 784$ 개의 크기를 가지고 있습니다. 생각해보십시오. 우리가 분석하는 이미지는 대부분 컬러입니다. 그럼 그 용량은 레드, 그린, 블루 계열의 색상 정보만큼, 다시 말해 무려 3배나 증가하게 됩니다. Mnist를 컬러이미지가 되었다고 가정해봅시다. 그럼 이미지 하나당 무려  $28 \times 28 \times 3 = 2352$ 개의 크기를 가지게 됩니다. 어떤 일이 일어날까요? 크기가 2352인 이미지들을 DNN을 통해 모델링하기 위해서는 **정말 어마어마하게 많은 은닉 계층이 필요할 것입니다.** 또한 이렇게 많은 은닉 계층들로 이루어진 DNN은 과적합의 위험이 매우 높습니다. 정말 비효율적이죠? 하지만 CNN은 다릅니다. 왜일까요? 아래 그림에 그 해답이 나와있습니다.

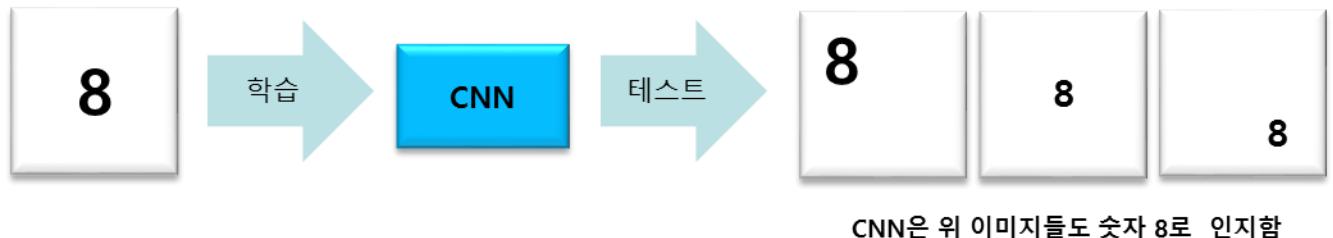
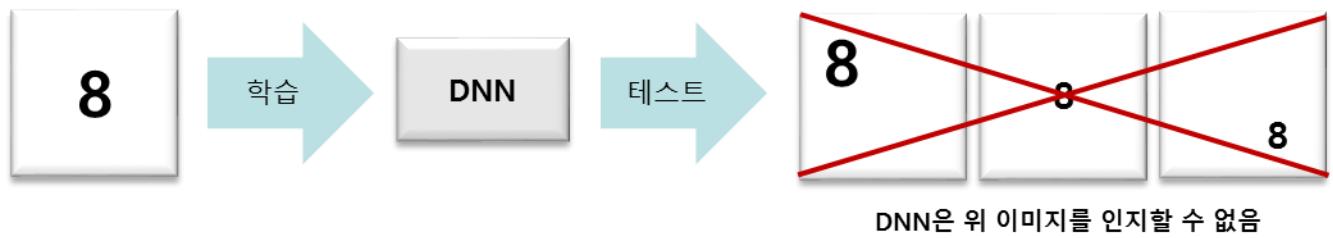


[그림 23] CNN을 통한 얼굴 인식 과정

CNN은 DNN과 다르게 이미지 전체에서 한번에 특징을 추출하지 않습니다. 필터를 활용하여 특징들을 명확히 구분하여 추출합니다. 위 그림에서와 같이 얼굴 이미지를 눈과 관련된 특징들, 코와 관련된 특징들, 그리고 입과 관련된 특징들로 구분하여 추출합니다. 이는 수 많은 은닉 계층을 활용하여 무작위로 특징을 뽑아내는 DNN과 비교하여 훨씬 더 효율적임을 알 수 있습니다.

또한 풀링 계층을 통해 이미 구분되어 추출된 특징들에서 가장 확연히 드러나는 특징들을 다시 재추출하는 압축 과정을 거칩니다. 이렇게 압축 과정까지 거친 특징들을 기반으로 DNN과 마찬가지로 Fully-Connected Net을 구성하게 됩니다. 그러면 당연히 DNN보다 훨씬 과적합 위험이 낮아지겠죠?

이미지에 대한 딥 러닝에서 CNN을 선호하는 이유는 더 있습니다. 바로 Invariant 특성 때문입니다. Invariant란 말 그대로 “불변”입니다. 아래 그림을 보도록 합시다.



[그림 24] CNN의 Invariant 특성

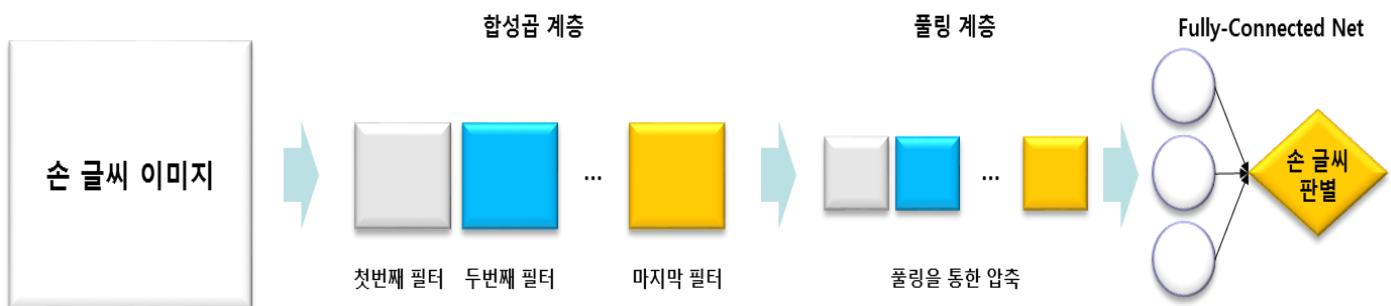
차이가 느껴지시나요? 가장 왼쪽에 있는 숫자8을 학습한 DNN 모델은 오른쪽과 같이 위치가 바뀌거나 그 크기가 바뀔 경우 다른 이미지로 인식하게 됩니다. 하지만 CNN은 인간과 마찬가지로 동일한 숫자8이라고 인식합니다.

이러한 특징을 Translation/Scale Inavariant라고 부릅니다.

## I. 이제 구현할 모델링 흐름을 설명해주세요.

이제 CNN으로 하는 이유에 대해서 알게 됐으니 MNIST 손글씨 데이터를 CNN을 구현해봅시다. 그리고 그 차이가 뭔지 생각해봅시다.

### [CNN 구성도]



### [분석 과정]



### [분석 환경]

- 운영체제: Ubuntu 14.04 LTS 이상, CentOS 7 이상
- 텐서플로우 버전: r0.11 이상 (CPU & GPU 버전)
- 파이썬 버전: 3.5.x

위 환경을 바탕으로 모델을 구축하겠습니다.

## II. 이제 CNN으로 직접 작성해주세요

DNN으로 작성할 때 이미 MNIST 데이터셋에 대해서 소개했으므로 그것에 대한 설명은 넘어가겠습니다. 이제 CNN으로 작성합시다.

라이브러리  
불러오기

데이터  
불러오기

모델 구축

모델 학습

모델 평가

먼저 필요 패키지들을 불러오고 입력 데이터와 출력데이터를 설정합니다. 그리고 reshape 함수를 사용하여 다음과 같이 입력 데이터를 원래 이미지의 구조로 재구성합니다.

### 실행 코드

```
# 라이브러리 및 데이터를 불러옵니다.  
import tensorflow as tf  
import numpy as np  
import matplotlib.pyplot as plt  
import matplotlib.cm as cm  
import matplotlib  
import os  
import glob  
from tensorflow.examples.tutorials.mnist import input_data  
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)  
  
x = tf.placeholder(tf.float32, [None, 784])  
y_ = tf.placeholder(tf.float32, [None, 10])  
  
# CNN 모델에 맞게 입력 이미지의 형태를 변경합니다.  
x_image = tf.reshape(x, [-1, 28, 28, 1])
```

tf.reshape에서 첫 번째 인자(x 부분)는 여러 개의 이미지 값을 가질 수 있음을 의미하고, 나머지 인자들은 각각 28X28 pixel의 1가지 색상(흑백)을 의미합니다.

라이브러리  
불러오기

데이터  
불러오기

모델 구축

모델 학습

모델 평가

이제는 모델 구축하기 위해서 가중치 행렬 W와 편향 b의 초기값 설정을 위한 함수 두 개를 정의합시다.

### 실행 코드

```
# 가중치 및 편향의 초기값 설정을 위한 함수를 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)
```

첫 번째 함수인 weight\_variable는 일종의 random noise로 초기화하였고, 두 번째 함수인 bias-variable는 편향을 적절한 상수 값으로 초기화 하였습니다

## 실행 코드

```
# 합성곱/패딩/풀링 적용을 위한 함수를 정의합니다. #####
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')
```

앞장에서 설명 하였듯이 CNN을 구성하기 위해서는 스트라이드(Stride), 패딩(Padding), 풀링(Pooling) 크기를 설정해야 합니다. 여기서는 스트라이드를 1로, 패딩을 'SAME', 풀링 크기는 2X2로 설정하겠습니다.

### # 패딩의 값이 SAME 이라는 것은?

스트라이드를 1로 주었을 때 생성되는 계층의 크기가 입력 계층과 같아지도록 하는 옵션입니다.

위에서의 각 함수의 스트라이드의 두 번째와 세 번째의 요소를 확인해봅시다. 각각 [1,1]과 [2,2]로 되어있습니다. 이것은 합성곱 계층과 풀링 계층을 만들기 위해 윈도를 오른쪽으로 1, 아래로 1, 오른쪽으로 2, 아래로 2만큼 이동하라는 것을 의미합니다. 여기서 입력 크기는 28X28 이기에 패딩 옵션에 의해 생성되는 합성곱 계층 크기 역시 28X28의 크기가 됩니다.

이제 첫 번째 합성곱 계층에 사용될 가중치 행렬 W와 편향 b를 정의합시다.

## 실행 코드

```
# 첫번째 합성곱 계층의 가중치 및 편향을 정의합니다.
W_conv1 = weight_variable([5,5,1,32])
b_conv1 = bias_variable([32])
```

이 예제에서는 윈도 크기가 5X5인 32개의 필터를 사용합니다. 특히 색상은 흑백이므로 차원의 크기는 1이 됩니다. 따라서 가중치 행렬 W는 구조가 [5,5,1,32]인 텐서로 정의되며, 마찬가지로 32개의 가중치 행렬에 대한 편향은 크기가 [32]인 텐서로 정의됩니다.

다음으로는 활성화 함수 및 풀링 계층에 대해 정의합시다.

## 실행 코드

```
# ReLU 함수 및 풀링 계층을 정의합니다. #####
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
h_pool1 = max_pool_2X2(h_conv1)
```

활성화 함수는 앞에서 설명한 DNN에서 가장 많이 쓰이는 ReLU를 사용합니다. 또한 Max Pooling을 적용합시다. 이제는 DNN처럼 지금까지 작성한 구조를 바탕으로 여러 계층이 되도록 쌓아봅시다.

5X5 윈도에 필터 수를 늘려 64개의 필터를 갖는 두 번째 합성곱 계층을 만들어봅시다. 바로 앞 단계인 풀링 계층이 총 32개의 필터를 가지고 있으므로 가중치 행렬의 구조는 [5,5,32,64]가 됩니다.

## 실행 코드

```
# 두번째 합성곱/ReLU/풀링 계층을 정의합니다.  
W_conv2 = weight_variable([5,5,32,64])  
b_conv2 = bias_variable([64])  
  
h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)  
h_pool2 = max_pool_2X2(h_conv2)
```

활성화 함수도 ReLU를 사용하고 풀링 역시 Max Pooling으로 동일하게 합니다.

이렇게 완성된 2번째 풀링 계층의 크기는 7X7X64가 됩니다. 다음 단계는 이전 장에서 했던 것과 비슷하게 Fully Connected Network를 통해 소프트맥스를 구성하고 출력합니다.

먼저 입력 값을 처리하기 위해 1024개의 뉴런을 사용하도록 하겠습니다. 이때 가중치와 편향은 다음과 같습니다.

## 실행 코드

```
# Fully connected net을 정의합니다.  
W_fc1 = weight_variable([7*7*64,1024])  
b_fc1 = bias_variable([1024])
```

Fully-Connected Net을 구성하는 은닉 계층의 입력 값은 직렬화된 벡터 형태여야 합니다. 이를 위해 두 번째 풀링 계층의 결과 값을 1차원 벡터 형태로 변형시킵니다.

활성화 함수는 마찬가지로 ReLU를 적용하고 마지막에는 드롭아웃을 적용하여 총 10개의 classes로 구성된 소프트맥스를 만듭니다.

## 실행 코드

```
h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])  
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)  
  
# 드랍아웃을 정의합니다.  
keep_prob = tf.placeholder("float")  
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)  
  
# Softmax를 적용하여 최종 결과값을 정의합니다.  
W_fc2 = weight_variable([1024,10])  
b_fc2 = bias_variable([10])  
  
y_conv=tf.nn.softmax(tf.matmul(h_fc1_drop, W_fc2) + b_fc2)
```

라이브러리  
불러오기

데이터  
불러오기

모델 구축

모델 학습

모델 평가

이제는 MNIST 데이터 각각의 이미지가 어떤 클래스(0~9)에 속하는지에 대해 판별하는 모델을 학습시키며 모델 평가를 통해 성능을 측정하겠습니다.

## 실행 코드

```

# 모델 학습 및 테스트를 통해 정확도를 출력합니다.
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))

sess = tf.Session()

sess.run(tf.initialize_all_variables())
for i in range(2000):
    batch = mnist.train.next_batch(100)
    if i%10 == 0:
        train_accuracy = sess.run(accuracy, feed_dict={
            x:batch[0], y_: batch[1], keep_prob: 1.0})
        print("step %d, train accuracy %g"%(i, train_accuracy))
    sess.run(train_step,
             feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})

print("test accuracy %g"% sess.run(accuracy, feed_dict={
    x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0}))

```

CNN 모델 훈련 및 평가를 구현은 DNN과 거의 흡사합니다. 다만 다른 점이 있다면 최적화 알고리즘으로 gradient decent가 아닌 ADAM 알고리즘을 사용했다는 것과 드롭아웃의 확률을 조절하는 매개 변수가 추가 된다는 점입니다.

### 출력 결과

```

step 0, train accuracy 0.13
step 10, train accuracy 0.23
step 20, train accuracy 0.51
step 30, train accuracy 0.76
step 40, train accuracy 0.75
step 50, train accuracy 0.81
step 60, train accuracy 0.87
step 70, train accuracy 0.86
.... (생략) ....
step 1950, train accuracy 1
step 1960, train accuracy 0.98
step 1970, train accuracy 0.97
step 1980, train accuracy 0.95
step 1990, train accuracy 0.98
test accuracy 0.9862

```

CNN으로 학습한 모델의 최종 Test Accuracy는 약 98%로 DNN 모형 보다 약 6%가 증가함을 알 수 있습니다.

## I. 만들고자 하는 자동분류기는 무엇인가요?

컬러 이미지를 이용해서 자동분류기를 만들 수 있을지 확인합시다.

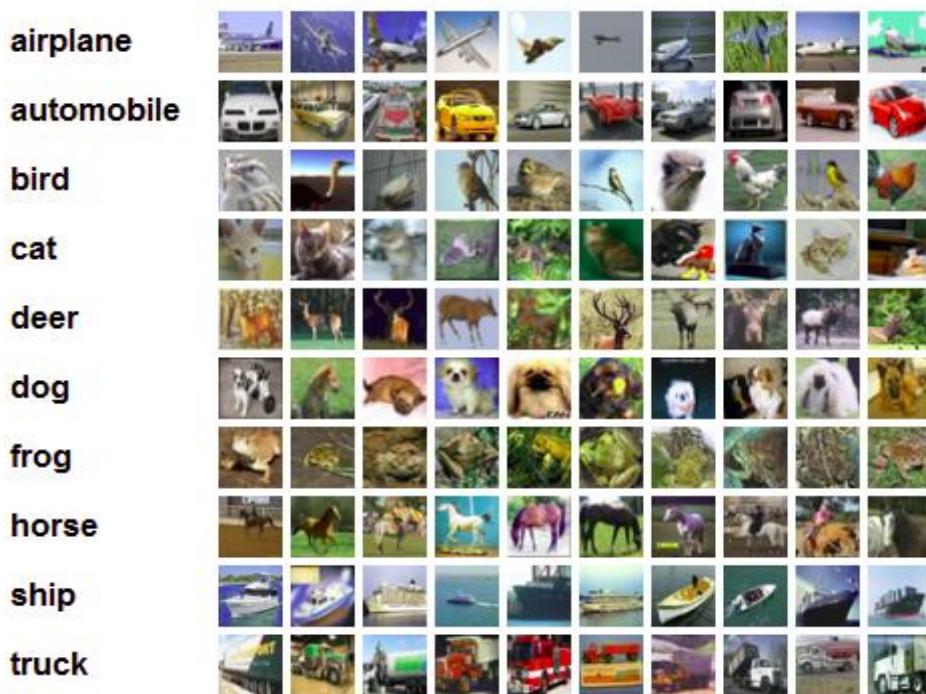
지금까지는 흑백 이미지로 자동분류기를 만들었습니다. 그러면 컬러 이미지로 자동분류기를 만드는 것도 가능할 것 같지 않나요? 우선 흑백 이미지와 컬러 이미지의 차이를 생각해봅시다. 컬러 이미지는 흑백보다 훨씬 더 많은 색이 있으므로 전처리 작업이 반드시 요구될 것입니다.

우리는 손글씨 데이터처럼 딥러닝 학습에 많이 사용되는 데이터인 Cifar-10을 사용할 것입니다. 이 데이터는 Cifar-10 사이트인 <https://www.cs.toronto.edu/~kriz/cifar.html>에 들어가서 직접 다운로드 할 수 있습니다.

## II. Cifar-10 데이터는 어떤 것인가요?

그러면 사용할 데이터셋인 Cifar-10에 대해서 알아보고 자동분류기를 만들 준비를 합시다.

이 데이터셋은 총 6만 개의 32x32 pixel 크기의 컬러 이미지로 구성되어 있습니다. 이미지는 총 10개의 클래스로 나눠져 있고 각 클래스는 6000개의 이미지가 들어있습니다. 각 클래스에 대한 정보는 아래 그림과 같습니다.



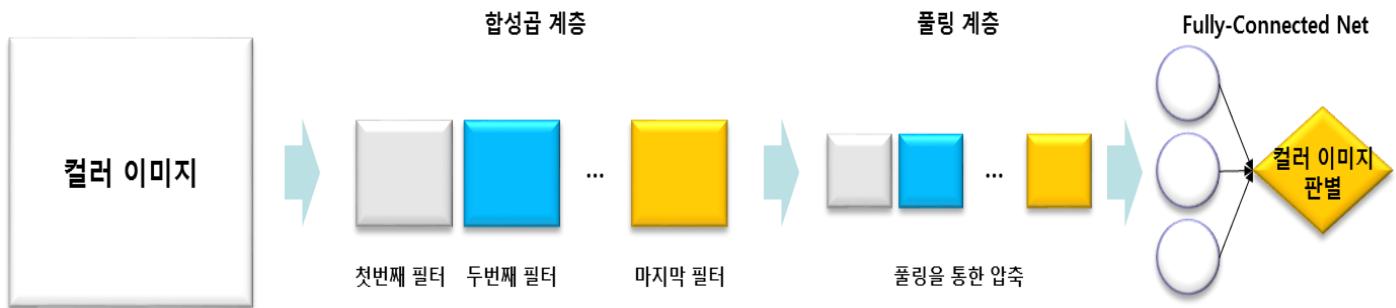
[그림 25] Cifar-10 데이터셋 샘플

앞에서 다룬 MNIST 데이터와 다른 점은 아무래도 색이 있는 것이 큰 차이겠죠? 따라서 컬러 채널(Channel)의 사이즈가 3인 것을 기억합시다.

### III. 이제 구현할 모델링 흐름을 설명해주세요.

컬러 이미지인 CiFar-10으로 어떻게 자동 분류기를 만들 수 있는지에 대해서 흐름을 생각해봅시다.

#### [CNN 구성도]



#### [분석 과정]



#### [분석 환경]

- 운영체제: Ubuntu 14.04 LTS 이상, CentOS 7 이상
- 텐서플로우 버전: r0.11 이상 (CPU & GPU 버전)
- 파이썬 버전: 3.5.x

위 환경을 바탕으로 모델을 구축하겠습니다.

## IV. 직접 CNN으로 작성해주세요.

MNIST로 만드는 것과 어떤 차이가 있는지 생각하면서 분석 흐름을 따라가봅시다.

데이터 다운로드 &  
메모리 로딩하기

데이터  
전처리

모델 구축

모델 학습

모델 평가

여기서는 파이썬 모듈을 따로 만들어 Cifar-10 데이터셋 다운로드를 수행하겠습니다. 그 모듈에는 download.py 와 cifar10.py 가 있는데, 먼저 Download.py 라는 모듈은 다운로드 과정이 들어 있습니다. 그리고 cifar10.py 는 다운로드 후 메모리에 로딩하기까지의 과정도 포함되어 있습니다.

해당 모듈에 대한 자세한 설명은 생각하고 곧바로 코드 작성은 하도록 하겠습니다.

[download.py]

실행 코드

```
import sys
import os
import urllib.request
import tarfile

def _print_download_progress(count, block_size, total_size):
    # Percentage completion.
    pct_complete = float(count * block_size) / total_size
    # Status-message. Note the \r which means the line should overwrite itself.
    msg = "\r- Download progress: {0:.1%}".format(pct_complete)
    # Print it.
    sys.stdout.write(msg)
    sys.stdout.flush()

def maybe_download_and_extract(url, download_dir):
    # Filename for saving the file downloaded from the internet.
    # Use the filename from the URL and add it to the download_dir.
    filename = url.split('/')[-1]
    file_path = os.path.join(download_dir, filename)
    # Check if the file already exists.
    # If it exists then we assume it has also been extracted,
    # otherwise we need to download and extract it now.
    if not os.path.exists(file_path):
        # Check if the download directory exists, otherwise create it.
        if not os.path.exists(download_dir):
            os.makedirs(download_dir)
        # Download the file from the internet.
        file_path, _ = urllib.request.urlretrieve(url=url,
                                                   filename=file_path,
                                                   reporthook=_print_download_progress)
        print()
        print("Download finished. Extracting files.")
        # Unpack the tar-ball.
        tarfile.open(name=file_path, mode="r:gz").extractall(download_dir)
        print("Done.")
    else:
        print("Data has apparently already been downloaded and unpacked.")
```

## [cifar10.py]

### 실행 코드

```
import numpy as np
import pickle
import download

data_path = "/home/eduuser/NN/Data/CIFAR-10/"

data_url = "https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz"

img_size = 32
num_channels = 3
img_size_flat = img_size * img_size * num_channels
num_classes = 10
_num_files_train = 5
_images_per_file = 10000
_num_images_train = _num_files_train * _images_per_file

def _get_file_path(filename=""):
    return data_path + "cifar-10-batches-py/" + filename

def _unpickle(filename):
    file_path = _get_file_path(filename)
    print("Loading data: " + file_path)
    with open(file_path, mode='rb') as file:
        data = pickle.load(file, encoding='bytes')
    return data

def _convert_images(raw):
    raw_float = np.array(raw, dtype=float) / 255.0
    images = raw_float.reshape([-1, num_channels, img_size, img_size])
    images = images.transpose([0, 2, 3, 1])
    return images

def _load_data(filename):
    data = _unpickle(filename)
    raw_images = data[b'data']
    cls = np.array(data[b'labels'])
    images = _convert_images(raw_images)
    return images, cls

def _one_hot_encoded(cls):
    return np.eye(num_classes)[cls]

def maybe_download_and_extract():
    download.maybe_download_and_extract(url=data_url, download_dir=data_path)

def load_class_names():
    raw = _unpickle(filename="batches.meta")[b'label_names']
    names = [x.decode('utf-8') for x in raw]
    return names

def load_training_data():
    images = np.zeros(shape=[_num_images_train, img_size, img_size, num_channels], dtype=float)
    cls = np.zeros(shape=[_num_images_train], dtype=int)
    begin = 0
    for i in range(_num_files_train):
        images_batch, cls_batch = _load_data(filename="data_batch_" + str(i + 1))
```

```

num_images = len(images_batch)
end = begin + num_images
images[begin:end, :] = images_batch
cls[begin:end] = cls_batch
begin = end
return images, cls, _one_hot_encoded(cls)

def load_test_data():
    images, cls = _load_data(filename="test_batch")
    return images, cls, _one_hot_encoded(cls)

```



Cifar-10 데이터셋의 이미지 판별을 위한 CNN 구현은 앞에서 이미 학습한 MNIST 데이터셋과 거의 흡사합니다. 다만 Cifar-10의 경우 컬러 이미지이기 때문에 모델 구현에 앞서 데이터 전 처리가 필요합니다.

# 원본이미지와 전처리 후의 이미지의 차이는 본 실습 마지막에 볼 수 있는 코드를 첨부해 두었습니다.

## 실행 코드

```

# 라이브러리 및 데이터를 불러옵니다.
import numpy as np
import tensorflow as tf
import cifar10

cifar10.maybe_download_and_extract()

class_names = cifar10.load_class_names()
images_train, cls_train, labels_train = cifar10.load_training_data()
images_test, cls_test, labels_test = cifar10.load_test_data()

# 인위적인 이미지 부풀리기를 위한 함수를 정의합니다.
img_size_cropped = 24
from cifar10 import img_size, num_channels, num_classes

imgX = tf.placeholder('float', [None, img_size, img_size, num_channels], name='x_input')

def pre_process_image(image, training):
    if training:
        image = tf.random_crop(image, size=[img_size_cropped, img_size_cropped, num_channels])
        image = tf.image.random_flip_left_right(image)
        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.resize_image_with_crop_or_pad(image,
                                                       target_height=img_size_cropped,
                                                       target_width=img_size_cropped)
    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):

```

```

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

```

## 실행 코드

```

# 이미지 부풀리기를 위한 함수를 실제 데이터에 적용합니다.
sess = tf.Session()
init = tf.initialize_all_variables()
sess.run(init)

train_imgs = []
train_label = []

duplicate_num = 2
for i in range(len(images_train)):
    train_imgs.extend(duplicate_image(images_train[i,:,:,:], duplicate_num))
    labels = labels_train[i,:]
    train_label.extend(np.repeat(labels[np.newaxis,:], duplicate_num, axis=0))

sess.close()
train_imgs = np.asarray(train_imgs)
train_label = np.asarray(train_label)

```

위 코드는 이미지 데이터를 로드 후 각각의 이미지 파일을 원하는 개수만큼 복제하고 복제된 각각의 이미지를 'Crop', 'flip', 'hue', 'contrast', 그리고 'brightness' 등을 적용하는 과정입니다.

### # 이미지를 왜곡 시키고 학습 데이터를 늘리기 위한 작업

- ① 'Crop'은 이미지의 랜덤한 부분을 원하는 사이즈로 자르게 합니다.
- ② 'flip'은 이미지를 좌우 반전시킬 수 있습니다.
- ③ 'hue'는 색상을 조절합니다.
- ④ 'contrast'는 채도를 조절합니다.
- ⑤ 'brightness'는 명암을 조절합니다. 간단히 설명하면,

본 예제의 경우 기존의 32X32 pixel의 이미지를 'Crop'을 통해 24X24로 줄였고, duplicate를 통해 원하는 'duplicate\_num' 만큼 학습데이터를 늘립니다. 여기서는 'duplicate\_num'의 수를 2로 지정하였습니다. 전처리 과정을 거치기 때문에 원본 이미지에 대해 여러 번 왜곡 과정을 거친 데이터가 생성됩니다.

여기서 img\_size, num\_channels, num\_classes는 각각 32, 3 그리고 10으로 cifar10.py 모듈에서 미리 설정하였습니다. 이것으로 데이터 전 처리는 완료됩니다.

이제 배치 데이터를 생성합시다. 배치 데이터는 np.random.choice 메서드를 활용하여 배치 사이즈만큼 랜덤하게 뽑아내는 방식을 취하도록 합시다.

## 실행 코드

```

# 배치데이터를 생성합니다.
train_batch_size = 100
def random_batch(arr_x,arr_y):
    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

```

데이터 다운로드 &  
메모리 로딩하기

데이터  
전처리

모델 구축

모델 학습

모델 평가

이제 전처리 작업은 끝났으니 모델을 만들어보겠습니다. MNIST로 만든 예제와 아주 흡사합니다. 다만 추가되는 사항이 조금 더 있습니다. 해당 모델을 구현한 코드는 아래와 같습니다.

- ① Fully-Connected Net에 은닉 계층 추가
- ② 각 계층(합성곱 계층과 풀링 계층 등)을 구성하는 뉴런들의 수 증가

## 실행 코드

```
# CNN 모델 구축
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)

# 합성곱/패딩/풀링
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')

window_size = 5
filter_1 = 32
filters_2 = 64

full_layer_1 = 256
full_layer_2 = 128

pool_width_1 = 2
pool_height_1 = 2
pool_width_2 = 2
pool_height_2 = 2

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

# 첫번째 합성곱 계층 및 풀링
W_conv1 = weight_variable([window_size,window_size,num_channels,filter_1], 'weight_1')
b_conv1 = bias_variable([filter_1], 'bias_1')

h_conv1 = tf.nn.relu(conv2d(x,W_conv1) + b_conv1)
h_pool1 = max_pool_2x2(h_conv1)

# 두번째 합성곱 계층 및 풀링
W_conv2 = weight_variable([window_size,window_size,filter_1,filters_2], 'weight_2')
b_conv2 = bias_variable([filters_2], 'bias_2')

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

```

h_pool2 = max_pool_2x2(h_conv2)

flat_num = int(ceil(img_size_cropped/pool_width_1/pool_width_2) *
    ceil(img_size_cropped/pool_height_1/pool_height_2) *
    filters_2)

# Fully-Connected Net
W_fc1 = weight_variable([flat_num, full_layer_1], 'weight_fc1')
b_fc1 = bias_variable([full_layer_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)

# 드랍 아웃
keep_prob = tf.placeholder('float')
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

W_fc2 = weight_variable([full_layer_1, full_layer_2], 'weight_fc2')
b_fc2 = bias_variable([full_layer_2], 'bias_fc2')

h_fc2 = tf.nn.relu(tf.matmul(h_fc1_drop,W_fc2)+b_fc2)
h_fc2_drop = tf.nn.dropout(h_fc2, keep_prob)

W_fc3 = weight_variable([full_layer_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)

# 최종 예측값 범위 지정합니다.
clipped_y_conv = tf.clip_by_value(y_conv, 1e-5, 1.0)
cross_entropy = -tf.reduce_sum(y_* tf.log(clipped_y_conv))

train_step = tf.train.AdamOptimizer(0.0001).minimize(cross_entropy)

correct_prediction = tf.equal(tf.argmax(y_conv, 1), tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, 'float'))

# 모델 학습 및 테스트를 통해 정확도를 출력합니다.
with tf.device('/cpu:0'):
    sess = tf.Session()
    sess.run(tf.initialize_all_variables())

for i in range(25000):
    x_batch, y_batch = random_batch(train_imgs, train_label)

    if i%100 == 0:
        train_accuracy = sess.run(accuracy, feed_dict={x: x_batch, y_: y_batch, keep_prob: 1.0})
        print("step %d, training accuracy %g"%(i, train_accuracy))
        train_cost = 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})

x_test = sess.run(pre_process(images_test, training=False))
print("test accuracy %g"% sess.run(accuracy, feed_dict={x: x_test, y_: labels_test, keep_prob: 1.0}))

```

## 출력 결과

```

step 24800, training accuracy 0.952
step 24800, training cost 183.028
test accuracy 0.7463

```

해당 모델의 최종 Test Accuracy는 약 75%로 측정되었습니다.

## [추가]

이미 모든 데이터를 3차원 Array 형태로 변환하였기 때문에 해당 예제를 수행하면서 학습데이터를 볼 수 있는 방법이 없습니다. 따라서 원본 이미지 및 전처리 과정을 거친 왜곡된 이미지 등을 Plotting 할 수 있는 부분을 아래에 추가합니다.

### 실행 코드

```
# 부풀려진 이미지를 출력하여 확인하겠습니다.
import matplotlib.pyplot as plt
def plot_images(images, cls_true, cls_pred=None, smooth=True, show_row=None, show_col=None):

    # 이미지를 출력하기 위한 sub-plots를 생성합니다.
    if show_row is None and show_col is None:
        fig, axes = plt.subplots(3, 3)
    else :
        fig, axes = plt.subplots(show_row,show_col)

    # 이미지 간의 간격을 조정합니다.
    if cls_pred is None:
        hspace = 0.3
    else:
        hspace = 0.6
    fig.subplots_adjust(hspace=hspace, wspace=0.3)

    if show_row ==1 and show_col == 1:
        cls_true_name = class_names[cls_true]
        xlabel = "True: {}".format(cls_true_name)
        plt.xlabel(xlabel)
        plt.imshow(images)
    else:
        for i, ax in enumerate(axes.flat):
            # 이미지의 화질을 조정합니다.
            if smooth:
                interpolation = 'spline16'
            else:
                interpolation = 'nearest'

            # 이미지를 생성합니다.
            ax.imshow(images[i, :, :, :], interpolation=interpolation)

            # 이미지가 속한 실제 클래스의 이름을 저장 및 지정합니다.
            cls_true_name = class_names[cls_true[i]]

            if cls_pred is None:
                xlabel = "True: {}".format(cls_true_name)
            else:
                # 예측된 이미지의 클래스를 지정합니다.
                cls_pred_name = class_names[cls_pred[i]]

                xlabel = "True: {}\nPred: {}".format(cls_true_name, cls_pred_name)

            ax.set_xlabel(xlabel)

            ax.set_xticks([])
            ax.set_yticks([])
```

```

plt.show()

# 출력을 원하는 이미지를 지정합니다.
trainImg = images_train[5]
train_cls_true = cls_train[5]

# 이미지 부풀리기를 적용 후 실제 이미지와 왜곡된 이미지를 출력합니다.
distortedImg = duplicate_image(trainImg, 9)
distorted_cls_true = np.repeat(train_cls_true, 9, axis=0)

plot_images(images=trainImg, cls_true=train_cls_true, smooth=True, show_row=1, show_col=1)
plot_images(images=distortedImg, cls_true=distorted_cls_true, smooth=True, show_row=3, show_col=3)

```

## 출력 결과



왼쪽의 이미지는 원본이미지입니다. 오른쪽의 이미지는 해당 결과에 'Crop'과 'flip'을 적용하고 'Hue', 'Contrast', 'Brightness'를 무작위로 조절하여 Inflate시킨 이미지들입니다.

이번 기회를 통해 컴퓨터가 '인식'하기까지 많은 과정이 요구된다는 것을 깨달았어.

맞아. 그런데 말  
이야,  
나 예전부터 딥러닝으로



응? 그게 뭐였어? 어서 말해봐! 우리  
가 악힌 DNN과 CNN으로 구현할  
수 있을거야!

난 대신 과제 작성해주는 모델을  
딥러닝으로 만들어보고 싶어.



그런 것도 딥러닝으로 만들 수  
있어? DNN이나 CNN으로?

아니, 내가 찾아봤는데 언어로  
문장을 만들어 주기 위해선  
다른 방법을 써야한다.



그게 뭔지 설명해줄 수 있어? 나도  
요즘 과제가 많아서 누가 대신 작  
성해줬을 줄겠다는 생각 많이 해.

응, 찾아보니까 RNN이라는  
알고리즘을 적용하면 된다.  
우리 같이 해보자!



# CONTENTS

## 딥러닝 시작하기(3)

### 3단계

[기초]

RNN 이해 ↗ p67



[실습]

RNN으로 ‘반갑습니다’가 자동 완성되도록 학습 ↗ p71



[발전]

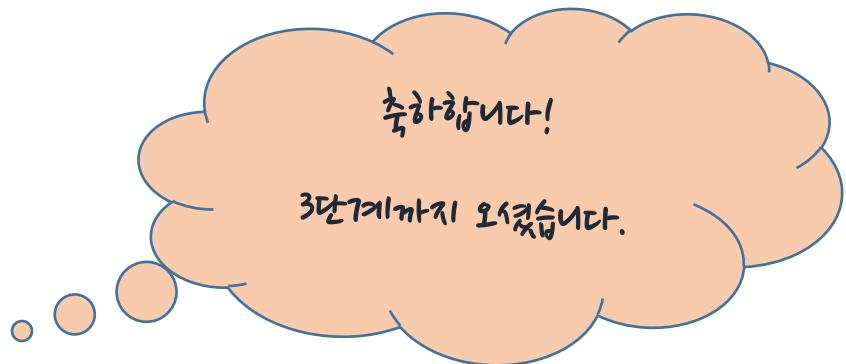
LSTM의 등장 ↗ p75



[활용]

언어 학습 모델 만들기 ↗ p79

(부제: Penn Treebank 언어 학습 데이터로)



## I. 먼저 자연어 처리(NLP) 이해가 필요합니다.

여러분이 과제를 작성해주는 모델을 만든다고 생각해봅시다. 그러기 위해선 우리가 사용하는 자연어처리에 대한 이해가 필요합니다. 지금까지 자연어 처리를 어떻게 해왔으며, 어떤 방식으로 해왔을까요?

'과제를 작성한다.'를 아주 단순하게 표현하자면 '글을 작성한다.'로 말할 수 있겠죠? 그러면 '글을 작성한다'는 어떤 것인가요? 언어를 학습한 다음에 논리적인 구조로 문장 하나 하나를 연결하는 것일 겁니다. 그리고 문장을 하나씩 연결한다는 것은 앞의 문장을 이해하고, 이에 대한 연관 관계를 찾은 후에 다음 문장을 만드는 것일 겁니다. 그런데 이것을 어떻게 구현할까요?

그 질문에 답하기에 앞서 자연어 처리(NLP, Natural Language Processing)를 어떻게 해왔는지에 대해 생각해봅시다. 처리할 텍스트 데이터가 있다면, 이를 각 단어별로 매칭시켜서 치환해주는 '기계번역'이 있고, 처리할 텍스트에 대한 내용을 요약해주는 '자동 요약' 기능이 있으며 마지막으로 문법적으로 안 맞는 것에 대해 알려주는 '자동 오류 수정' 기능이 있어 왔습니다. 이 모든 기능들은 딥러닝 없이도 구현이 가능한 하위 레벨의 기능입니다.

그렇지만 이 기능들은 문제점이 있습니다. 예를 들면 몇 년 전의 포털 사이트의 번역기를 생각해봅시다. 지금은 더할 나위없이 많이 나아진 것을 확인할 수 있지만, 예전의 번역기는 말도 안되는 문장을 출력해주었습니다. 그 때는 번역기보다 사전 검색해서 문장을 이해하는 것이 더 빨랐습니다.

그 당시에는 왜 그랬을까요? 자연어는 단어의 이중적 의미나 모호성 때문에 일대일 매칭이 온전히 되질 않으며, 작성자의 상황과 환경에 따라 표현이 달라질 수 있는 특징이 있습니다. 이 특징들이 있기에 통계적 규칙 기반으로 효과적인 결과를 얻어내기가 매우 어려웠습니다.

그 점을 획기적으로 극복하게 된 것은 바로 1957년에 Harris가 발표한 '분포가설'을 통해서입니다. 분포 가설이란 한마디로 비슷한 문맥을 가진 단어는 비슷한 의미를 갖는다는 내용입니다. 이 가설에 기반하여 크게 두 가지 종류가 있습니다. 하나는 단어와 문맥의 출현 횟수(Count-Based Methods)를 세는 방법이며 다른 하나는 단어에서의 문맥을 혹은 문맥에서의 단어를 예측하는 방법(Predictive Methods)입니다. 이 방법을 가장 효과적으로 나타낼 수 있는 것은 단어가 갖은 의미를 벡터화 시키는 것입니다.

단어가 벡터가 된다는 것만으로도 할 수 있는 것들이 많아집니다. 단어 사이의 유사도 측정, 단어들 간의 평균을 수치적으로 나타낼 수 있어서 예측도 가능하게 됩니다. 이제는 단어를 벡터화해서 자연어 처리를 하는 것이 아주 당연한 일이 되었습니다. 따라서 이와 관련된 모델을 만드는데 앞으로 단어를 벡터화해서 구현해보도록 하겠습니다.

## II. RNN은 어떤 이유로 생겨났나요?

이제 단어를 벡터화 하는 이유에 대해서는 알게 됐습니다. 그러면 이를 신경망에 어떻게 접근할 수 있었을까요? 그 배경에 대해 알아보고 DNN과 CNN으로는 구현하지 않고, 왜 RNN으로 하는지에 대해 알아봅시다.

자 이제 단어를 벡터화 하면 효과적인 방법으로 자연어 처리를 구현할 수 있다는 것을 알게 됐습니다. 그러면 질문이 하나 생길 것입니다. “그러면 어떻게 딥러닝으로 구현할 수 있다는 거야?”

이 질문은 보다 세부적으로 나눠야 할 것입니다. 첫째, 딥러닝으로 자연어 처리를 할 수 있는가? 둘째, 자연어처리가 효과적으로 되기 위해선 분산표현이 가능해야 하는데 이것을 인공 신경망으로 구현할 수 있는가? 셋째, 딥러닝으로 다 가능하다면 어떤 알고리즘을 사용해야하는가?

위 세가지 질문에 대해 이렇게 답변 드리겠습니다. 딥러닝 기술로 분산 표현이 가능하므로 풍부한 정보를 얻어낼 수 있습니다. 또한 이것은 당연히 단어를 벡터화해서 사용하므로 가능하며, 언어처리를 위해선 대표적으로 RNN이라는 알고리즘을 사용합니다.

그러면 왜 RNN이라는 알고리즘을 써야할까요? 이에 대한 대답은 다음 절에 소개 드리겠습니다. 그리고 DNN과 CNN으로는 자연어처리를 위해 사용하지 않는지에 대한 이유를 먼저 소개하겠습니다.

여러분이 알게 된 DNN과 CNN은 은닉 계층의 내용의 차이가 있을 뿐 [입력 계층]-[은닉 계층]-[출력 계층] 이렇게 구조가 동일합니다. 그런데 문맥과 문맥을 이해해서 연결을 해줘야 하는 것이 주 목적인 자연어 처리 기법을 기준의 알고리즘에 적용할 수 있었을까요?

**문맥을 이해한다는 것은 앞의 내용과 뒤의 내용의 연관성을 찾는 것입니다.** 컴퓨터가 연관성을 찾는 것을 어떻게 할 수 있을까요? 문장을 이루는 연속된 단어들이 있다면 그 단어들을 컴퓨터가 기억할 수 있어야합니다. 그리고 그 기억된 것을 바탕으로 모델 학습하고 추론된 결과를 나타낼 수 있어야 합니다.

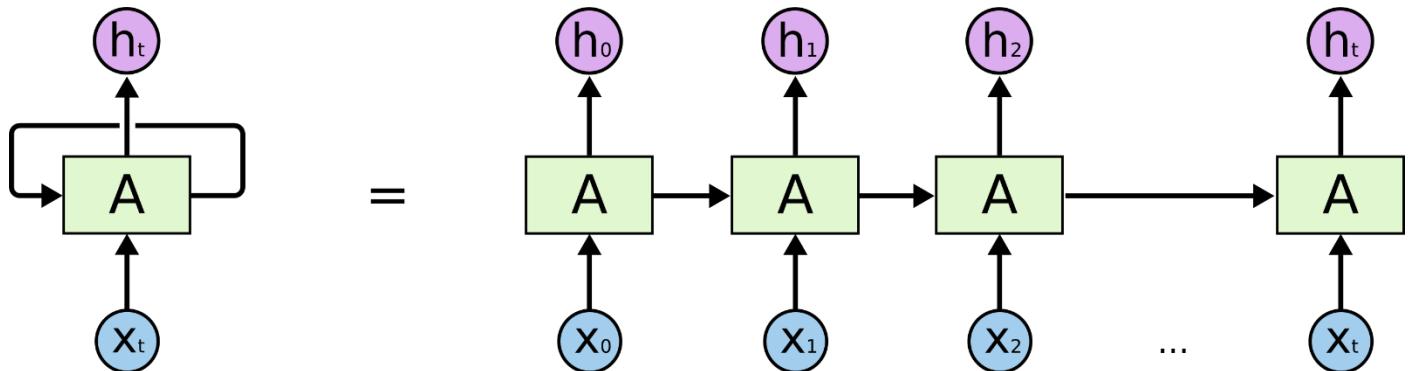
이러한 구조를 기준의 방법으로 구현할 수 있을까요? 먼저 CNN을 떠올려봅시다. CNN은 입력에 넣을 이미지에 대해 대표할 만한 특징을 여러 번 뽑아내고, 이를 압축하여 학습하는 데는 매우 유용했습니다. 그리고 그것에 학습을 시키기 위해 이 과정을 무한히 반복해야 합니다. 그런데 말입니다. 입력 값으로 넣은 이미지를 간의 변화를 예측할 수도 있을까요?

정답을 먼저 드리자면 어렵습니다. 입력 계층부터 출력 계층까지 연결되어 있지만, 이 계층들이 유기적으로 연결되어있지 않기 때문입니다. 그래서 고안된 것이 순환 개념을 적용한 순환 신경망(RNN, Recurrent Neural Network)입니다.

### III. RNN은 어떤 알고리즘이죠?

인공 신경망에 순환이라는 개념을 적용한 알고리즘이 RNN이라는 것까지 소개했습니다. 이제 RNN이 어떤 알고리즘인지에 대해 소개하겠습니다.

순환 개념이라는 것을 적용한 RNN을 구조화 하면 어떤 그림이 나올까요? 아래 사진을 봅시다.

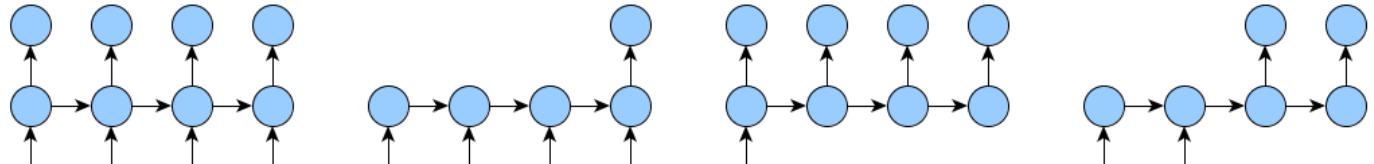


[그림 26] RNN의 구조 예

지금껏 해온 방법과 조금 달라서 낯설게 보이시죠? 그렇지만 조금만 알면 이해하기 쉬울 겁니다. 위 그림의 왼쪽과 오른쪽은 같은 그림이며, 그림의 왼쪽의 순환 구조를 풀어서 표현하면 오른쪽과 같은 것을 보여줍니다. 여기서 RNN의 구조는 [입력 계층]-[은닉 계층]-[출력 계층] 이렇게 나란히 줄지어 표현하지 않고 왼쪽으로 90도를 돌린 형태로 보면 됩니다. 아래의  $x_0-x_t$ 는 입력 부분입니다. 그리고 위쪽의  $h_0-h_t$ 는 입력에 대한 상태(앞으로 은닉 상태라고 부를 것입니다.)를 나타낸 부분인데 여기를 은닉 계층이라고 생각하면 됩니다. 또 가운데  $A$ 는 뉴런(여기 RNN에서는 뉴런이라는 표현 대신 셀(Cell), 메모리(Memory))라는 표현을 주로 사용합니다.)입니다. 그리고 각각의 셀마다 결과값을 출력할 수 있습니다.

다시 설명하자면 시퀀스라고 불리는 연속된 데이터를 모델에 넣어서 학습하기 위해서는 입력 받은 값  $x(t)$ 에 대해서 항상 상태 값  $h(t)$ 이 나오게 됩니다. 그 상태 값  $h(t)$ 은 다음 시퀀스 데이터  $x(t+1)$ 와 결합하여 또 다른 상태 값  $h(t+1)$ 을 나타내게 됩니다. 이 구조는 이전 시점에서 학습된 정보가 다음 시간에 재 사용될 수 있는데 가장 효율적입니다.

Many to Many      Many to One      One to Many      Many to Many



[그림 27] 다양한 RNN 구조의 예

위의 그림에서 볼 수 있듯이 RNN의 구조는 다양하게 표현할 수 있습니다. 여기서 입력 받은 값들을 은닉 계층

으로 넘겨주기 위해선 다른 알고리즘과 같이 선형 모델로 구현하면 됩니다. 또한 마지막으로 출력할 때는 활성화 함수(주로 하이퍼볼릭 탄젠트)를 사용합니다.

$$h(t) = \tanh(W(h(t-1), x(t)) + b)$$

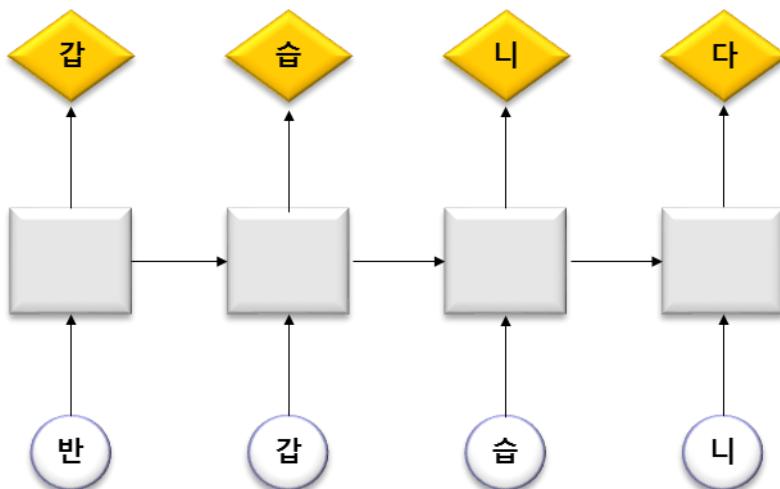
이처럼 은닉 상태가 과거 문맥(t-1)에 대한 정보를 갖고 순환하므로 앞으로의 상태를 연속 데이터로 예측할 때 도움됩니다.

## I. 이제 구현할 모델링 흐름을 설명해주세요.

CNN에서는 설명이 길어졌지만 RNN에서는 곧 바로 실습을 하여 RNN에 대한 이해를 돋도록 하겠습니다. 여기 '반갑습니다'라는 단어를 학습하여 각 글자 다음을 예측할 수 있는 자동 완성 기능을 만들어보겠습니다.

우리는 RNN으로 만들어진 모델에 연속 데이터를 학습하면 그 다음 데이터를 추측하는 데 매우 효과적임을 이미 알고 있습니다. 여기 '반갑습니다'라는 단어가 있습니다. 우리는 가장 기본적인 RNN 모델을 통해서 이를 학습시킬 것입니다.

우선 결과부터 생각해 봅시다. 자동 완성을 해주는 것이 목적이므로 '반'을 입력했을 때는 그 다음에 '갑'을 써야 한다는 것을 알려줘야 할 것이고, 또 '반갑습니'까지 썼을 때는 '다'를 입력하여 '반갑습니다'로 입력되도록 나타내주는 모델을 상상해볼 수 있겠습니다. 그러면 이를 그림으로 나타내 볼까요?



[그림 28] '반갑습니다'에 대한 자동 완성 모델

위 그림처럼 실제 코딩에서는 이런 식으로 예측할 것입니다. 입력 데이터로 ['반', '갑', '습', '니']를 입력하게 되면, ['갑', '습', '니', '다']가 출력되도록 구현할 것입니다.

## II. 직접 RNN으로 작성해서 보여주세요.

- 운영체제: Ubuntu 14.04 LTS 이상, CentOS 7 이상
- 텐서플로우 버전: r0.11 이상 (CPU & GPU 버전)
- 파이썬 버전: 3.5.x

이제 간단한 RNN 모델을 만들기 위해 필요한 라이브러리를 불러옵니다. 우리는 "반갑습니다"라는 단어의 한글수

는 총 5개 이므로 이를 One-hot Vector로 나타낼 것입니다. 그러면 알파벳 개수는 총 5개이므로 반=[1,0,0,0,0], 갑=[0,1,0,0,0], 습=[0,0,1,0,0], 니=[0,0,0,1,0], 다=[0,0,0,0,1] 이런 식으로 나타낼 수 있습니다.

## 실행 코드

```
# RNN for '반갑습니다'
import tensorflow as tf
import numpy as np

# '반갑습니다'를 벡터형태로 변환합니다.
char_rdic = ['반', '갑', '습', '니', '다']
char_dic = {w : i for i, w in enumerate(char_rdic)}

print (char_dic)

ground_truth = [char_dic[c] for c in '반갑습니다']
print (ground_truth)

x_data = tf.one_hot(ground_truth[:-1], len(char_dic), 1.0, 0.0, -1)
print(x_data)
```

## 출력 결과

```
{'다': 4, '니': 3, '갑': 1, '반': 0, '습': 2}
[0, 1, 2, 3, 4]
Tensor("one_hot_1:0", shape=(4, 5), dtype=float32)
```

이제 RNN 모델을 만들 것입니다. 은식 상태의 크기(셀의 크기)와 배치 사이즈, 출력 값의 크기를 지정해야 합니다. 앞에서 모델링 구현에서 보여줬듯이 은식 상태의 크기는 4로 정하고, 배치 사이즈는 단어 하나만 입력 값으로 넣을 수 있기에 1로 정합니다. 마지막으로 출력 값은 총 5개의 한글 중 하나로 나타내야 하므로 그 크기는 5가 됩니다.

이를 텐서플로우의 함수인 BasicRNNCell로 모델 구현을 위한 주요 파라미터 값을 지정해줍니다. 여기서 값을 지정하는 것은 은닉 상태(h), 입력 값(x), 출력 값(y)을 지정해주는 것입니다.

여기서 rnn\_cell.zero\_state 함수는 은닉 상태의 셀을 0으로 초기화 시키는 것을 의미합니다. 그리고 tf.split을 통해 입력 데이터가 순환되는 횟수(메모리 셀의 개수)를 지정합니다. 여기서는 한글이 4개만 사용되므로 4로 지정했습니다.

마지막으로 tf.nn.rnn 함수로 출력 값과 은닉 상태가 출력될 수 있도록 합니다.

## 실행 코드

```
# RNN 파라미터를 정의합니다.
rnn_size = len(char_dic)
batch_size = 1
output_size = len(char_dic)

# RNN Model을 정의합니다.
rnn_cell = tf.nn.rnn_cell.BasicRNNCell(num_units = rnn_size, input_size = None)
print(rnn_cell)

initial_state = rnn_cell.zero_state(batch_size, tf.float32)
print(initial_state)
```

```

x_split = tf.split(0, len(char_dic) - 1, x_data)
print(x_split)

outputs, state = tf.nn.rnn(cell = rnn_cell, inputs = x_split, initial_state = initial_state)

```

다음으로는 위에서 출력 값에 소프트맥스 활성화 함수를 적용하기 위해 필요한 가중치 W와 편향 b를 지정하고 손실 함수를 정의합니다. 특히 손실 함수를 정의할 때 지금까지 써온 크로스 엔트로피(Cross Entropy Function)으로 구할 수 있으나, 여기서는 순환 구조로 된 RNN에서 사용하기엔 계산이 복잡해지므로 sequence\_loss\_by\_example라는 함수를 사용하여 구합니다.

이 함수를 사용하기에 필요한 파라미터들은 ①logits: 출력 값, ②targets: 실제 값, ③weights: 가중치 총 3개입니다. 먼저 logits의 경우 2D이며, Targets의 경우는 1D, weights는 1로 됩니다. 보통 weights는 1로 두는데 그 크기는 메모리 셀의 개수(time step의 수) X 배치 사이즈가 됩니다. 이제 손실을 지정하는 것으로써 모델 구현이 완료 되었습니다.

## 실행 코드

```

# 가중치 및 편향 그리고 손실함수를 정의합니다.
W = tf.Variable(tf.zeros([rnn_size, output_size]))
b = tf.Variable(tf.zeros([output_size]))

output = tf.reshape(tf.concat(1, outputs), [-1, rnn_size])
logits = tf.nn.xw_plus_b(output, W, b)
probs = tf.nn.softmax(logits)

targets = tf.reshape(ground_truth[1:], [-1]) # a shape of [-1] flattens into 1-D
targets.get_shape()

weights = tf.ones([(len(char_dic)-1) * batch_size])
weights.get_shape()

loss = tf.nn.seq2seq.sequence_loss_by_example([logits], [targets], [weights])
cost = tf.reduce_sum(loss) / batch_size
train_op = tf.train.RMSPropOptimizer(0.01, 0.9).minimize(cost)

```

이제 session을 정의하고 실행합시다.

## 실행 코드

```

# 모델 학습 후 결과값을 출력합니다.
with tf.Session() as sess:
    tf.initialize_all_variables().run()
    for i in range(100):
        sess.run(train_op)
        result = sess.run(tf.argmax(probs, 1))
        print(result, [char_rdic[t] for t in result])

```

위 코드에서 훈련 위해 100번 반복을 해볼 것입니다. 따라서 출력되는 부분도 100줄이 나올 것입니다. 뒤에 몇 줄을 제외하면 나머지는 자동완성 기능을 구현하지 못하다는 것을 알 수 있습니다. 왜냐하면 훈련이 덜 됐기 때문이고, 마지막에는 훈련이 충분히 완료 되었으므로 '반' 입력했을 때 다음에 올 알파벳이 ['갑', '습', '니', '다']'으로 자동 완성이 됐음을 알 수 있습니다.

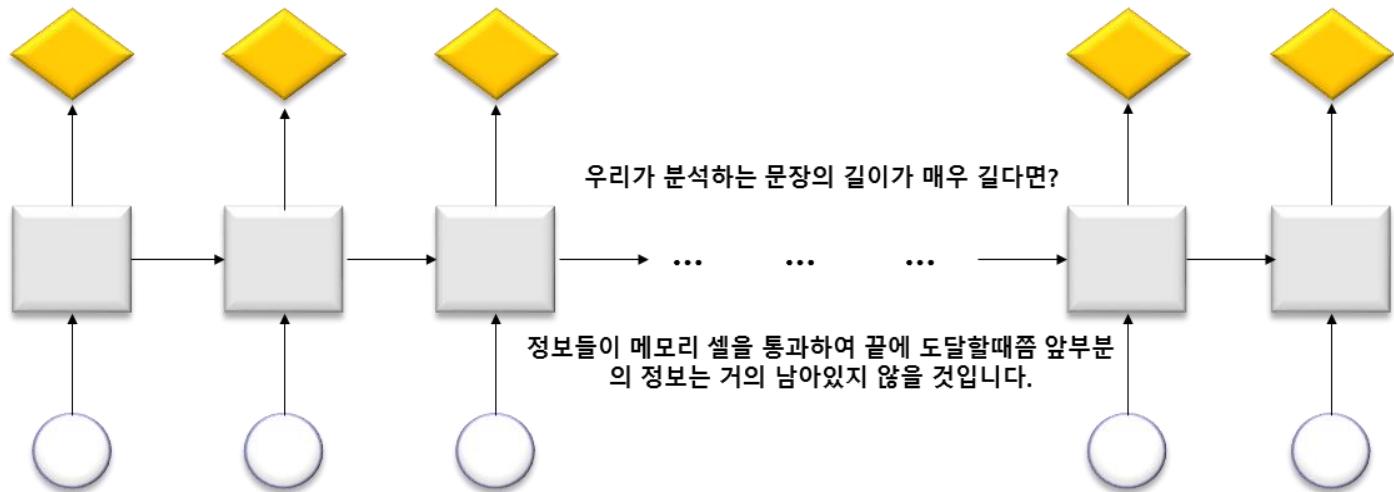
## 출력 결과

```
[1 2 4 4] ['갑', '습', '다', '다']
..... (생략) .....
[1 2 3 4] ['갑', '습', '니', '다']
```

## I. LSTM은 RNN의 어떤 점 때문에 등장했나요?

사실 RNN에 대해서 배웠으나 이것 자체를 사용하지는 않습니다. 왜냐하면 RNN에 치명적인 단점이 있기 때문입니다. 그 단점을 극복하기 위해 개선된 LSTM의 탄생 배경에 대해서 알아봅시다.

바로 위에서 적용한 '반갑습니다' 자동 완성 모델을 다시 떠올려봅시다. 그리고 그 모델을 어떻게 구현했는지 그 그림을 생각해봅시다. 그것은 단어 하나에 대해서 만들어진 4개의 메모리 셀이었습니다. 그리고 앞으로 이런식으로 문장을 학습한다고 생각해봅시다. 어떠할 것 같나요? 무척 길어지고 복잡해질 것입니다. 길어지고 복잡해지면 어떤 현상이 발생할까요? 메모리는 맨 초기에 입력 받은 값을 기억해낼 수 있을까요?



그럼 다른 예를 들어봅시다. 예를 들어 "나는 미국에서 자랐다. 나의 취미는 컴퓨터 게임이다. 내가 좋아하는 음식은 피자이다. ..... 많은 문장들 ..... 생략 .....). 나는 □□로 대화하는 것에 능숙하다." 라는 문장을 RNN으로 학습했습니다. 이때 □□을 예측한다면 정답은 '영어'가 될 것입니다. 그러나 이와 관련된 힌트를 주는 '미국에서 자랐다'라는 정보는 거리상 멀리 떨어져 있기 때문에 기본적인 순환 신경망으로는 팔호를 예측하기 힘듭니다.

여기서 거리상 멀리 떨어져 있다는 말은 RNN으로 모델을 구현했을 때 만들어진 메모리 셀이 너무나 깊어져서 이를 상태를 지속할 수 없음을 의미합니다. (구체적으로는 Vanishing Gradient Problem이 발생합니다.) 여하튼 바로 이 단점을 극복하기 위해 고안된 방법이 바로 LSTM(Long Short-Term Memory)입니다. 이것은 1997년 Hochreiter와 Schmidhuber에 의해 소개되었으며 RNN의 한 종류입니다.

### # Vanishing Gradient

#### Problem 이란?

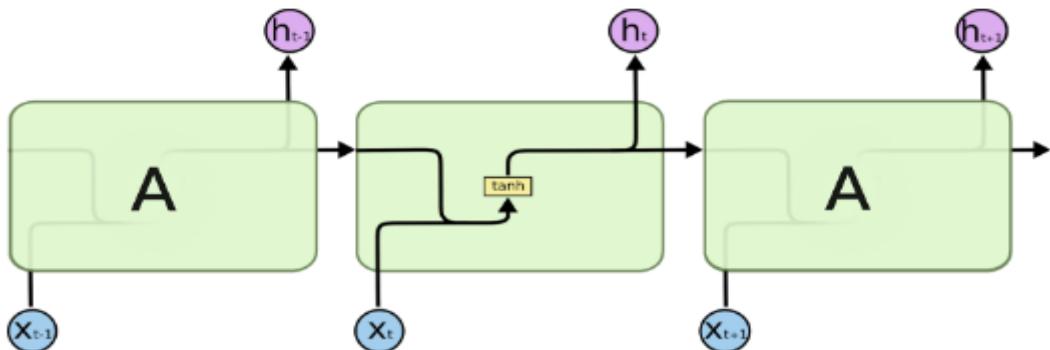
메모리가 깊어지면 가중치도 그만큼 곱해지게 됩니다. 그때 곱해진 가중치 때문에 결국 기울기가 기하급수적으로 사라지게 되어 문제가 발생하게 됩니다.

## II. LSTM은 어떤 알고리즘이죠?

이제 여러분은 LSTM이 왜 등장하게 됐는지에 대해 알게 됐습니다. 단점을 극복하여 개선된 알고리즘인 LSTM에 대해 본격적으로 알아봅시다. 이 알고리즘은 장기기억과 단기기억 모두 유지하는 것이 가능합니다.

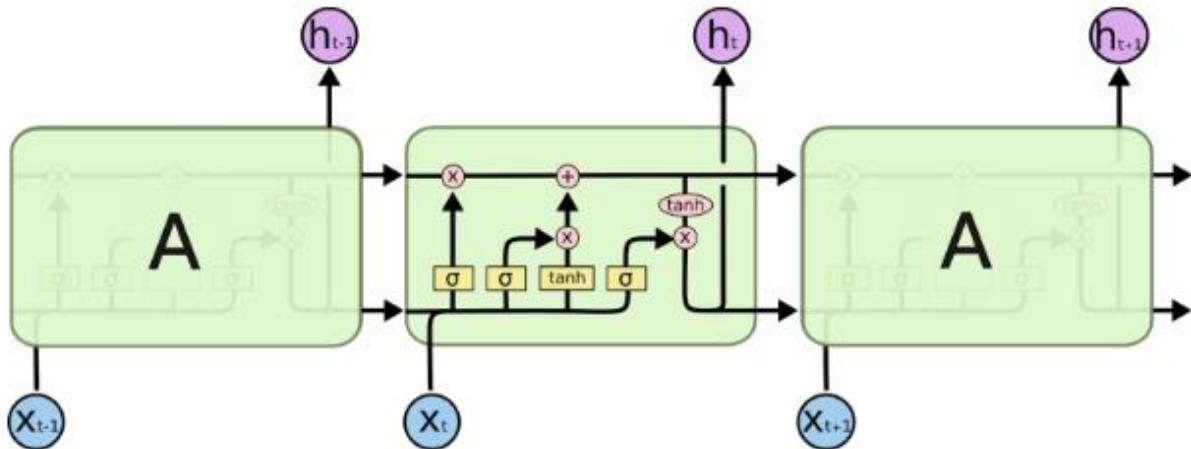
LSTM은 1997년 Hochreiter와 Schmidhuber에 의해 소개된 개념이며 이후 널리 퍼졌습니다. 기존의 순환 신경망인 RNN과 LSTM을 그림으로 비교해보겠습니다.

바로 아래의 RNN 구조는 앞서 소개한 RNN의 내부 구조를 뜯어본 것이라고 생각해봅시다. 내부적으로는 탄젠트 같은 간단한 구조만 포함하고 있습니다.



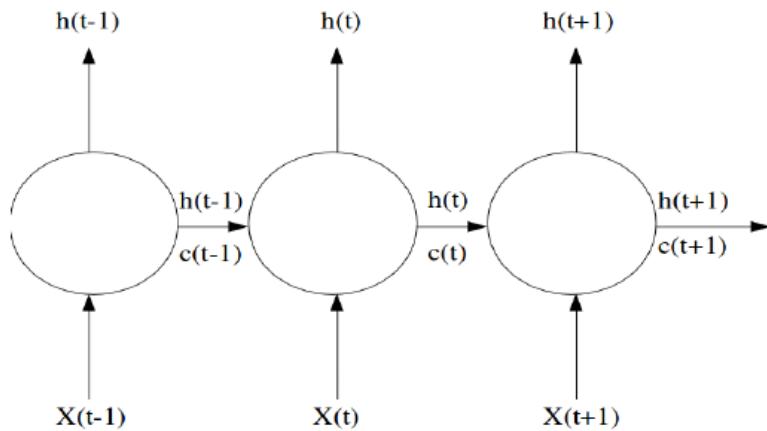
[그림 29] RNN의 전체적인 구조

반면에 LSTM의 구조를 봅시다. 비슷한 구조로 되어있지만 더 복잡하게 돼있습니다. 구체적으로 보면 LSTM은 네 개의 층으로 구성되며 이들은 모두 상호작용을 합니다. 어떤 점이 장기 기억을 못하는 RNN의 단점을 극복하게 만들었을까요?



[그림 30] LSTM의 전체적인 구조

바로 위의 그림은 복잡하지요? 좀더 단순한 그림으로 설명하도록 하겠습니다.



[그림 31] 단순화된 LSTM의 구조

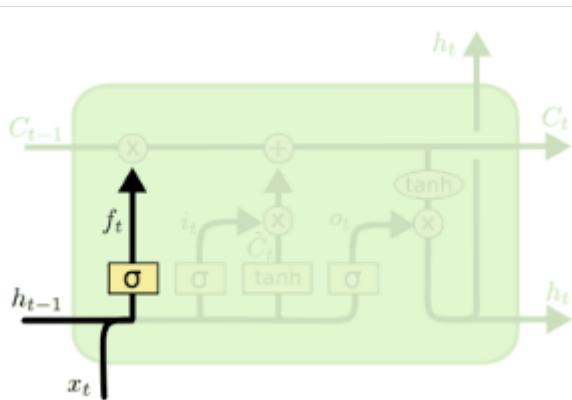
우선 RNN과 공통인 부분은 메모리 셀의 은닉상태와 현재 입력 값, 가중치  $w$ 에 대해서 선형 결합을 하는 것입니다. 이에 대한 식은 아래와 같습니다.

$$p = W(h(t-1), x(t)) + b$$

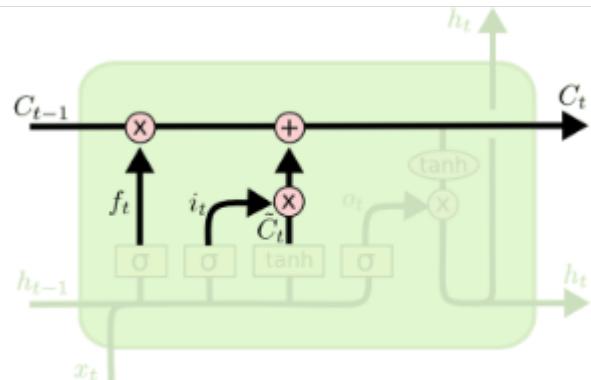
그럼 여기 새로 추가된 값인  $C(t)$ 는 무엇일까요?  $C$ 는 셀 상태(Cell State)라고 불리는 새로 추가된 개념인데 이것이 바로 RNN의 단점을 극복하게 만들었습니다. 따라서 계산이 좀 더 복잡해졌습니다.

기존의 RNN의 계산과 마찬가지로 은닉 상태( $h$ )의 값은 상위 계층의 입력 값으로 전달됩니다. 그러나 새로 생긴 메모리 셀( $c$ ) 값은 은닉 상태( $h$ )와 다르게 상위 계층으로 전달되지 않습니다. 그럼 이 셀 상태 값은 어떻게 계산이 될까요?

새로운 셀 상태( $C_{t+1}$ ) 값을 구하기 위해 새로운 개념이 추가됩니다. 바로 삭제 게이트(Forget Gate)와 입력 게이트(Input Gate)라고 불리는 것들입니다. 삭제 게이트는 말 뜻에서 알 수 있듯이 이전 셀의 상태( $C_{t-1}$ )의 값을 중에서 삭제할 것을 결정하는 게이트입니다. 이때 결정하는 방법은 활성화 함수인 시그모이드를 통해서 결정됩니다.



[그림 32] 삭제게이트 과정



[그림 33] 입력게이트 과정

위 그림을 예로 들어볼까요? 삭제 게이트의 층은  $h_{t-1}$ 과  $x_t$ 만을 보고 이전 셀 상태( $C_{t-1}$ )의 값을 결정합니다. 여기서 그 값이 0이 나오면 이전 값을 지우는 것이고 1이 나오게 되면 완전히 유지하는 프로세스를 가집니다. 이에 대한 식은 이렇습니다.

$$f(t) = c(t-1) * \text{sigmoid}(p(f))$$

그러면 입력 게이트는 어떤 것일까요? 이것은 입력될 새로운 정보를 셀 상태에 저장할지 말지에 대해서 결정하는 게이트입니다. 이 게이트에 대한 계산을 하기 위해선 초기  $p$  값에서 시그모이드를 적용한 값과 탄젠트 값을 적용한 것을 곱하여 구합니다.

$$i(t) = \text{sigmoid}(p(i)) * \text{tanh}(p(j))$$

이제 삭제 게이트와 입력 게이트에 대한 연산이 끝났습니다. 앞서 이것들은 새로운 셀의 상태를 계산하기 위해 필요한 것이라고 말했습니다. 새로운 셀의 상태는 바로 이 둘의 값을 더하면 얻어집니다.

$$c(t) = f(t) + i(t)$$

자 여기까지 잘 따라오셨나요? 마지막으로 셀의 상태를 알게 됐으니 은닉 계층에 대한 계산을 알아야 할 차례입니다. 이것도 RNN 의 은닉상태 값을 구하는 것과 유사하게 셀 상태에 탄젠트 활성화 함수를 적용한 값에 시그모이드 활성화 함수를 구한 값이 됩니다.

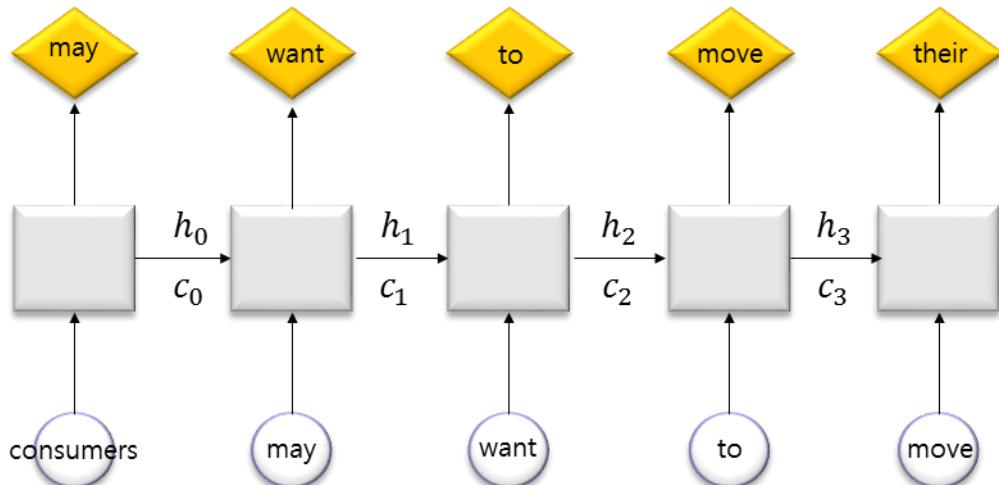
$$h(t) = \text{tanh}(c(t)) * \text{sigmoid}(p(o))$$

이렇게 계산된 셀의 상태와 은닉 상태 값들은 추후 계산을 위해서 저장된 후 전달됩니다. 이 과정은 신경망 구조를 거치면서 계속 반복 됩니다.

## I. 이제 구현할 모델링 흐름을 설명해주세요.

LSTM에 대해 이해했다면 이제 언어 학습 모델을 구현하도록 하겠습니다.

### [LSTM 구성도]



### [분석 과정]



### [분석 환경]

- 운영체제: Ubuntu 14.04 LTS 이상, CentOS 7 이상
- 텐서플로우 버전: r0.11 이상 (CPU & GPU 버전)
- 파이썬 버전: 3.5.x

위 환경을 바탕으로 모델을 구축하겠습니다.

## II. Penn TreeBank 데이터는 어떤 것인가요?

Penn TreeBank는 언어 모델 학습을 위해 많이 사용되는 데이터셋입니다. 이를 활용하여 언어 모델을 학습시켜 봅시다.

Penn TreeBank 데이터 중에서 하위 Data 디렉토리에 있는 데이터만 쓸 것입니다. 사용할 데이터셋에 대해서 설명하자면 ptb.train.txt, ptb.valid.txt, ptb.test.txt 총 3가지의 파일을 사용할 것입니다.

Ptb.train.txt는 모델을 학습 시킬 때 사용하며 ptb.valid.txt 파일을 사용하여 모델을 검증하고 마지막으로 ptb.test.txt 파일을 사용해서 모델의 성능을 평가합니다. 학습 데이터인 ptb.train.txt를 간략히 살펴보면 총 42,068개의 라인으로 되어있고 전체 단어 수는 887,521개입니다. 이것들은 모두 소문자로 되어 있으며 희귀한 단어는 <unk>로 표시되어 있고 숫자는 N으로 바뀌어 있습니다.

### # 데이터 샘플(ptb.train.txt)

```
aer banknote berlitz calloway centrust cluett fromstein gitano guterman hydro-quebec ipo kia memotec mlx  
nahb punts rake regatta rubens sim snack-food ssangyong swapo wachter  
pierre <unk> N years old will join the board as a nonexecutive director nov.
```

### # 데이터 샘플(ptb.valid.txt)

```
consumers may want to move their telephones a little closer to the tv set  
<unk> <unk> watching abc 's monday night football can now vote during <unk> for the greatest play in N  
years from among four or five <unk> <unk>  
two weeks ago viewers of several nbc <unk> consumer segments started calling a N number for advice on  
various <unk> issues
```

### # 데이터 샘플(ptb.test.txt)

```
no it was n't black monday  
but while the new york stock exchange did n't fall apart friday as the dow jones industrial average  
plunged N points most of it in the final hour it barely managed to stay this side of chaos
```

그러면 이것을 어떻게 LSTM으로 학습시킬 수 있을까요?

## III. 직접 LSTM으로 작성해주세요.



데이터는 터미널을 열어서 wget 명령어로 다운로드 하겠습니다. 받은 압축파일을 tar 명령어로 해제하겠습니다.

### 실행 코드

```
# 터미널에서 실행  
# $ wget http://www.fit.vutbr.cz/~imikolov/rnnlm/simple-examples.tgz  
# $ tar xvf simple-examples.tgz
```



## 실행 코드

```
# 라이브러리 및 데이터를 불러옵니다.
import numpy as np
import tensorflow as tf
import collections
import argparse
import time
import os
from six.moves import cPickle

data_dir      = "/home/eduuser/NN/Data/Penn_TreeBank"

input_file   = os.path.join(data_dir, "ptb.train.txt")
with open(input_file, "r") as f:
    data_train = f.read().split()
```

먼저 필요 패키지를 import하고 다운로드한 데이터를 모델에 활용하기 위해 사전 작업을 합니다. 데이터 셋이 저장되어 있는 디렉토리를 설정 후 Read().split() 메서드를 활용하여 해당 데이터를 단어 단위로 불러옵니다.

그 후 아래와 같이 데이터 셋의 단어를 카운팅하고 이를 많이 나온 단어 순으로 정렬합니다.

## 실행 코드

```
# 언어 데이터 전처리 과정
# 각 단어의 등장 횟수를 셹니다.
counter = collections.Counter(data)
count_pairs = sorted(counter.items(), key=lambda x: (-x[1], x[0]))
print ("Type of 'count_pairs' is %s and length is %d"
      % (type(count_pairs), len(count_pairs)))
for i in range(5):
    print (count_pairs[i])
```

## 출력 결과

```
Type of 'count_pairs' is <class 'list'> and length is 9999
('the', 50770)
('<unk>', 45020)
('N', 32481)
('of', 24400)
('to', 23638)
```

결과를 확인해 보면 총 9999 개의 단어가 있음을 알 수 있습니다. 그 중 많이 나온 상위 5 개 단어는 'the', '<unk>', 'N', 'of' 그리고 'to'임을 알게 됐습니다. 그리고 나온 횟수대로 정렬된 단어에 대해 0부터 9999 까지 고유 번호를 지정하는 사전을 만듭니다.

## 실행 코드

```
# 각 단어에 고유 번호를 부여하고 해당 단어와 번호로 이루어진 사전을 만듭니다.
chars, counts = zip(*count_pairs)
```

```
vocab = dict(zip(chars, range(len(chars))))
```

## 출력 결과

```
vocab['the'] is 0
vocab['<unk>'] is 1
vocab['N'] is 2
vocab['of'] is 3
vocab['to'] is 4
```

만들어진 사전을 이용하여 학습용 데이터인 ptb.train.txt에 위에서 지정한 고유 번호를 부여합니다. 마찬가지로 모델 평가를 위해 test 테스트 셋을 불러오고 사전을 활용하여 고유 번호를 부여합니다.

## 실행 코드

```
# 테스트 데이터 셋을 불러옵니다.
input_file = os.path.join(data_dir, "ptb.test.txt")
with open(input_file, "r") as f:
    data_test = f.read().split()

# 학습 및 테스트 셋에 앞에서 만든 사전을 적용하여 고유번호를 부여합니다.
corpus_train = np.array(list(map(vocab.get, data_train)))
corpus_test = np.array(list(map(vocab.get, data_test)))
check_len = 30
print ("\n'corpus_train' looks like %s" % (corpus_train[20:check_len]))
```

## 출력 결과

```
'corpus_train' looks like [9995 9996 9997 9998 9255 1 2 71 392 32]
```

## [배치 데이터 생성]

이제 학습 및 평가를 위한 데이터 전 처리 과정이 끝났습니다. 이러한 데이터를 활용하여 배치를 생성합시다.

## 실행 코드

```
# 배치데이터를 생성합니다.
batch_size = 20
seq_length = 20
num_batches_train = int(corpus_train.size / (batch_size * seq_length))
num_batches_test = int(corpus_test.size / (batch_size * seq_length))

corpus_train_reduced = corpus_train[::(num_batches_train*batch_size*seq_length)]
corpus_test_reduced = corpus_test[::(num_batches_test*batch_size*seq_length)]

xdata_train = corpus_train_reduced
ydata_train = np.copy(xdata_train)
ydata_train[:-1] = xdata_train[1:]
ydata_train[-1] = xdata_train[0]

xdata_test = corpus_test_reduced
ydata_test = np.copy(xdata_test)
ydata_test[:-1] = xdata_test[1:]
ydata_test[-1] = xdata_test[0]
```

## 출력 결과

```
print ('xdata_train is ... %s and length is %d' % (xdata_train, xdata_train.size))
```

```
xdata is ... [9969 9970 9971 ..., 368 13 29] and length is 887200
print ('ydata_train is ... %s and length is %d' % (ydata_train, ydata_train.size))
ydata is ... [9970 9971 9973 ..., 13 29 9969] and length is 887200
```

먼저 배치 크기와 seq\_length를 지정합니다. seq\_length는 LSTM에서 메모리 셀의 개수로 순환 신경망을 사용하여 연속적으로 처리할 데이터의 양을 뜻합니다. 전체 데이터의 단어 수를 배치 크기와 seq\_length의 곱으로 나누어 총 배치의 개수를 구하고 이를 num\_batches에 저장합니다. 이때 나누어 지는 값이 정수가 되도록 소수점 이하는 버립니다.

이제 배치 크기와 seq\_length의 곱으로 정확히 나누어 떨어지도록 데이터 크기를 다시 조정하고 입력 데이터와 타겟 데이터를 정의합니다. 여기서 입력 데이터와 타겟 데이터가 서로 매치가 되도록 타겟 데이터는 입력 데이터 기준 2번째 원소부터 시작하도록 지정합니다.

테스트 셋에도 마찬가지로 위의 작업을 적용하며, 이제 배치 데이터의 기본 구성이 끝났으므로 실제 배치 실행 시 사용될 데이터를 정의합시다.

## 실행 코드

```
xbatches_train = np.split(xdata_train.reshape(batch_size, -1), num_batches_train, 1)
ybatches_train = np.split(ydata_train.reshape(batch_size, -1), num_batches_train, 1)
xbatches_test = np.split(xdata_test.reshape(batch_size, -1), num_batches_test, 1)
ybatches_test = np.split(ydata_test.reshape(batch_size, -1), num_batches_test, 1)
```

## 출력 결과

```
print ("Type of 'xbatches_train' is %s and length is %d"
      % (type(xbatches_train), len(xbatches_train)))
Type of 'xbatches_train' is <class 'list'> and length is 2218

print ("Type of 'ybatches_train' is %s and length is %d"
      % (type(ybatches_train), len(ybatches_train)))
Type of 'ybatches_train' is <class 'list'> and length is 2218
```

배치 데이터 xbatches\_train과 ybatches\_train은 numpy의 split 매서드를 사용하여 앞에서 정의한 배치 크기와 seq\_length에 따라 2218개의 20X20 array를 원소로 가지는 List를 생성합니다. 테스트 셋에도 마찬가지로 적용합니다.



이제 모델을 구축하기 위한 적절한 파라미터를 정의합니다.

## 실행 코드

```
# LSTM 파라미터를 지정합니다.
vocab_size = len(vocab)
rnn_size   = 200
num_layers = 2
```

```

grad_clip = 5.
# LSTM 모델을 정의합니다.
unitcell = tf.nn.rnn_cell.BasicLSTMCell(rnn_size, state_is_tuple=True)
cell = tf.nn.rnn_cell.MultiRNNCell([unitcell] * num_layers, state_is_tuple=True)
input_data = tf.placeholder(tf.int32, [batch_size, seq_length])
targets = tf.placeholder(tf.int32, [batch_size, seq_length])
istate = cell.zero_state(batch_size, tf.float32)

```

본 예제에서 사용되는 LSTM은 셀 상태와 은닉 상태의 크기가 `rnn_size = 200`이고 총 2개의 layers로 구성되어 있습니다. 그리고 경사 하강법을 통해 구해지는 기울기(gradient)의 값이 과다하게 커지는 것을 막아주기 위해 클리핑(clipping)기법을 활용할 것이다.

이때 `grad_clip`의 값은 기울기(gradient)의 상한선 역할을 합니다. 이렇게 지정된 파라미터 값들을 기준으로 위와 같이 모델에 사용될 메모리 셀과 input 데이터, targets 그리고 셀/은닉 상태의 초기 값이 정의됩니다. 여기서 주목할 함수는 `MultiRNNCell`로 2개 이상의 계층을 가진 RNN을 구성할 때 사용되며, 본 예제에서는 `num_layers`를 2로 설정했습니다.

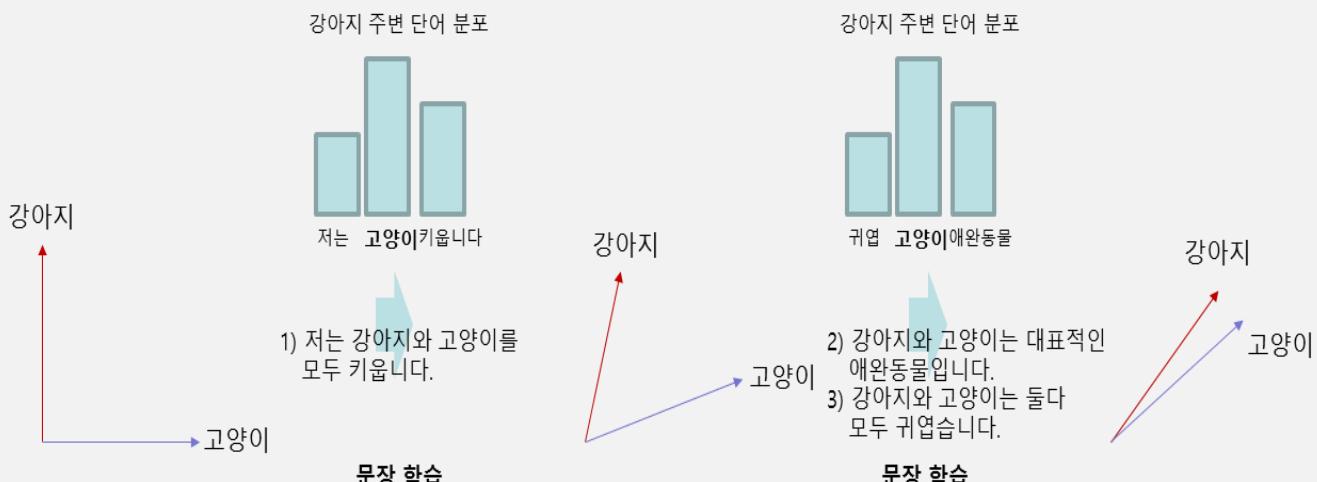
이제 최종 출력 값에 소프트맥스를 적용하기 위한 가중치와 편향을 설정하고 Word-Embedding 작업을 수행합시다.

#### # Word Embedding 이란?

앞에서 데이터를 전 처리 한 것 처럼 단순히 단어에 고유번호를 지정해 주는 것은 각 단어들 사이의 연관 관계를 전혀 고려하지 못합니다. 예를 들어 '강아지'를 '143' 그리고 '고양이'에게 '234' 라는 고유번호를 지정해주었다고 가정해봅시다.

이 두 가지 고유번호들은 '강아지'와 '고양이'사이의 연관관계를 전혀 표현하지 못하는 의미 없는 번호에 불과합니다. 이러한 문제를 해결하기 위해 'Vector Representations of Words' 방식이 생겨나게 되었습니다.

이는 벡터 공간 속에 단어들을 표현하는 것으로 서로 연관 관계가 있는 단어들은 벡터 공간 속에서 '서로 가까이 박혀있다'(embedded nearby each other) 고 표현한다. 즉 word-embedding은 단어를 벡터로 표현하여 단어들 간의 연관성까지 표현하는 기법입니다.



조금 더 구체적으로 살펴보면 위와 같이 '강아지'와 '고양이' 두 단어는 처음에는 그 연관성을 모르기 때문에 서로 직각을 이루는 벡터로 표현될 것입니다. 하지만 문장을 학습함으로써 특히 '강아지'와 '고양이'가 동시에 들어있는 문장을 학습함으로써 컴퓨터는 '강아지'와 '고양이'가 서로 연관성이 있다는 것을 인지하게 되고 그 결과로 두 번째 그리고 세 번째 벡터처럼 두 벡터가 이루는 각도가 점점 줄어 들 것입니다. 다시 말해 두 단어의 거리가 가까워진다는 뜻입니다. 만약 '강아지'라는 단어에 항상 '고양이'라는 단어가 등장한다면 두 단어를 표현하는 벡터는 일직선 상에 놓여지게 될 것입니다.

## 실행 코드

```
# 가중치 및 편향을 정의하고 Word-Embedding 작업을 수행합니다.  
with tf.variable_scope('RnnLm'):  
    softmax_w = tf.get_variable("softmax_w", [rnn_size, vocab_size])  
    softmax_b = tf.get_variable("softmax_b", [vocab_size])  
    with tf.device("/cpu:0"):  
        embedding = tf.get_variable("embedding", [vocab_size, rnn_size])  
        inputs = tf.split(1, seq_length, tf.nn.embedding_lookup(embedding, input_data))  
        inputs = [tf.squeeze(_input, [1]) for _input in inputs]
```

각 단어를 rnn\_size와 같은 200개의 원소를 가지는 벡터가 되도록 embedding작업을 수행하고 새로운 입력 데이터인 Inputs을 생성합니다. 따라서 Inputs은 총 20개의 20X200의 형태의 텐서로 구성됩니다.

## 실행 코드

```
# 예측값과 손실 함수를 정의합니다.  
outputs, last_state = tf.nn.seq2seq.rnn_decoder(inputs, istate, cell  
                                                , loop_function=None, scope = 'RnnLm')  
output = tf.reshape(tf.concat(1, outputs), [-1, rnn_size])  
logits = tf.nn.xw_plus_b(output, softmax_w, softmax_b)  
probs = tf.nn.softmax(logits)  
  
loss = tf.nn.seq2seq.sequence_loss_by_example([logits], # Input  
                                              [tf.reshape(targets, [-1])], # Target  
                                              [tf.ones([batch_size * seq_length])]) # Weight  
)
```

이제 seq2seq.rnn\_decoder함수를 사용하여 Outputs과 은닉 상태 그리고 셀 상태를 추출한 후 앞에서 학습한 '반갑습니다' 예제와 마찬가지로 Loss Function인 sequence\_loss\_by\_example에 들어갈 인자들을 정의합시다.

## 실행 코드

```
# 학습을 위한 매개 변수를 정의합니다.  
cost      = tf.reduce_sum(loss) / batch_size / seq_length  
final_state = last_state  
lr        = tf.Variable(0.0, trainable=False)  
tvars     = tf.trainable_variables()  
grads, _ = tf.clip_by_global_norm(tf.gradients(cost, tvars), grad_clip)  
_optm    = tf.train.AdamOptimizer(lr)  
optm     = _optm.apply_gradients(zip(grads, tvars))
```

모델을 학습하기 앞서 loss 및 은닉, 셀 상태 그리고 최적화 방법과 효율적인 최적화를 위한 인자들을 정의합니다. 위 코드에서 grads는 앞에서 설명한 클리핑(clipping)작업을 수행한 결과값이고 lr은 learning ratio의 약자로 학습속도를 조절해주는 변수입니다. 최적화 알고리즘은 ADAM 최적화 알고리즘을 사용하였으며 최종적으로 클리핑 작업을 수행한 gradients를 적용하여 최적화 알고리즘을 실행할 것입니다.



먼저 학습에 필요한 인자들을 정의해봅시다.

## 실행 코드

```
# 모델을 학습합니다.  
num_epochs      = 5  
learning_rate   = 0.002  
decay_rate      = 0.97
```

num\_epochs는 loss를 최소화하기 위해 학습을 반복하는 횟수입니다. 그리고 나머지 두 개의 인자는 앞에서 설명하였듯이 학습 속도를 효율적으로 바꿔주기 위한 값입니다.

이제 모델을 학습시켜보자.

## 실행 코드

```
sess = tf.Session()  
sess.run(tf.initialize_all_variables())  
init_time = time.time()  
train_cost_list = []  
for epoch in range(num_epochs):  
    sess.run(tf.assign(lr, learning_rate * (decay_rate ** epoch)))  
    state = sess.run(istate)  
    batchidx = 0  
    for iteration in range(num_batches_train):  
        start_time = time.time()  
        xbatch = xbatches_train[batchidx]  
        ybatch = ybatches_train[batchidx]  
        batchidx = batchidx + 1  
        train_cost, state, _ = sess.run([cost, final_state, optm]  
            , feed_dict={input_data: xbatch, targets: ybatch, istate: state})  
        train_cost_list.append(train_cost)  
    train_cost_avg = np.mean(train_cost_list)  
    train_perplexity = np.exp(train_cost_avg)  
    total_iter = epoch*num_batches_train + iteration  
    end_time = time.time();  
    duration = end_time - start_time  
    train_cost_list  
    if total_iter % 100 == 0:  
        print ("[%d/%d] train perplexity: %.4f / Each batch learning took %.4f sec"  
            % (total_iter, num_epochs*num_batches_train, train_perplexity, duration))
```

## 출력 결과

```
[33000/33270] train perplexity: 81.6634 / Each batch learning took 0.0478 sec  
[33100/33270] train perplexity: 80.5063 / Each batch learning took 0.0482 sec  
[33200/33270] train perplexity: 79.4723 / Each batch learning took 0.0480 sec
```

먼저 lr변수에 값을 할당하는 tf.assign구문을 살펴봅시다. 학습을 효율적으로 하기 위해 처음에는 학습 속도를 크게 잡아 오차를 빠르게 줄여나갑니다. 그리고 학습이 반복될 때마다 학습 속도를 작아지도록 하여 최적 값에 가까워 질수록 점차 조심스럽게 수렴이 되도록 합니다.

이렇게 학습 속도를 결정한 후 앞에서 생성한 배치 데이터를 차례대로 feed\_dict 인수를 통해 전달 후 최적화 알고리즘을 수행하는 session을 실행시키면 학습이 완료됩니다.



지금까지 훈련용 데이터 셋을 사용하여 모델을 학습하였습니다. 이제는 해당 모델의 성능을 평가할 차례이다.

언어 모델의 경우 모델 성능을 평가 할 때 자주 사용되는 지표는 '혼잡도(perplexity)' 입니다. '혼잡도'는 각 배치마다 발생하는 cost의 누적 값을 진행된 seq\_length의 합으로 나눈 값인데 해당 모델의 경우 '혼잡도'가 120이하가 되어야 합니다.

## 실행 코드

```
# 모델을 테스트 후 최종 혼잡도를 출력합니다.
test_cost_list = []
state_test      = sess.run(istate)
batchidx_test  = 0
for test in range(num_batches_test):
    xbatch_test      = xbatches_test[batchidx_test]
    ybatch_test      = ybatches_test[batchidx_test]
    batchidx_test   = batchidx_test + 1
    test_cost, state_test = sess.run([cost, final_state],
        , feed_dict={input_data: xbatch_test, targets: ybatch_test, istate: state_test})
    test_cost_list.append(test_cost)
test_cost_avg = np.mean(test_cost_list)
test_perplexity = np.exp(test_cost_avg)
print ("Test perplexity: %.3f" % test_perplexity)
```

## 출력 결과

Test perplexity: 119.635

## 레퍼런스

- [1] [https://dnddnjs.gitbooks.io/rl/content/neural\\_network.html](https://dnddnjs.gitbooks.io/rl/content/neural_network.html)
- [2] [http://www.nfx.co.kr/techpaper/keyword\\_view.asp?idx=235](http://www.nfx.co.kr/techpaper/keyword_view.asp?idx=235) 푸리에변환
- [3] <http://www.slideshare.net/ssuser06e0c5/i-64267027> 자연어 처리 기술
- [4] <https://shuuki4.wordpress.com/2016/01/27/word2vec-%EA%B4%80%EB%A0%A8-%EC%9D%B4%EB%A1%A0-%EC%A0%95%EB%A6%AC/> 자연어 처리 기술 및 단어 벡터화 관련
- [5] <https://deeplearning4j.org/usingrnnns> 다양한 RNN 구조의 예