

수학포기자를 위한 딥러닝과 텐서플로우 이해

조대협 (<http://bcho.tistory.com>)

2017.10.17

이 글은 제가 텐서플로우와 딥러닝을 공부하면서 블로그에 메모해뒀던 내용을 모아놓은 글입니다.

텐서플로우 초기버전부터 작성하였기 때문에, 다소 코드가 안맞는 부분이 있을 수 있으니 이 점 양해 부탁드립니다. 이 글은 개인이 스터디용으로 자유롭게 사용하실 수 있으며, 단체나 기타 상용 목적으로 사용은 금지 됩니다.

되도록이면 수학을 어려워하는 분들을 위해서 수학적인 배경은 자제했으며, 실제로 서비스에 적용하고자 하는 분들을 위주로 텐서플로우 라이브러리를 기반으로 하여 활용 관점에서 접근하였습니다.

목차

목차	0
1. 머신러닝의 개요	8
들어가기에 앞서서	8
머신러닝	9
지도 학습과 비지도 학습	10
지도 학습 (Supervised Learning)	10
비지도 학습 (Unsupervised learning)	10
머신러닝의 대표적인 문제 Regression과 Classification 문제	11
Classification	11
Regression	12
머신 러닝과 딥러닝	12
2. 선형회귀를 통한 머신 러닝의 개념 이해	14
거리에 따른 택시 요금 문제	15

가설 (Hypothesis) 정의	15
코스트(비용) 함수	17
옵티마이저 (Optimizer)	18
경사 하강법	18
학습	19
모델 평가(테스팅)	19
예측	20
머신 러닝의 순서	20
학습 단계	20
예측 단계	21
3. 텐서플로를 이용한 선형 회귀 모델 개발	21
텐서플로 개발 환경 셋업	22
구글 클라우드 계정 및 프로젝트 생성	22
프로젝트 생성	23
도커 설치 (수정 필요)	24
구글 클라우드 데이터 랩 설치	26
학습하기	26
테스트 데이터 만들기	27
학습 로직 구현	27
학습 속도(러닝 레이트 / Learning Rate) 조정하기	31
오퍼 슈팅 (Over shooting)	32
스몰 러닝 레이트(Small Learning Rate)	34
3. 로지스틱 회귀를 이용한 이항 분류 모델	36
분류 문제(Classification)의 정의	36
로지스틱스 회귀 분석 (Logistics Regression)	37
선형 회귀 분석 (Linear regression) 으로 분류 문제 접근하기	37
시그모이드(sigmoid) 함수	40
가설 (Hyphothesis)	41
디시전 바운드리(Decision boundary)	41
코스트 함수 (비용함수/Cost function)	42
옵티마이저 (Optimizer)	47
예측 (Prediction)	47
정리	47
참고	48
4. 텐서플로 기본	48
텐서플로 환경 설정	48
구글 데이터랩	48
아나콘다	49

텐서플로우의 자료형	49
상수형 (Constant)	49
그래프와 세션의 개념	50
플레이스 홀더 (Placeholder)	51
변수형 (Variable)	53
행렬과 텐서플로우	56
행렬의 기본 개념	56
행과 열	56
곱셈	56
행렬의 덧셈과 뺄셈	57
텐서 플로우에서 행렬의 표현	58
브로드 캐스팅	60
텐서플로우 용어 참고	62
5. 소프트맥스를 이용한 다항 분류 모델 구현	62
MNIST	62
데이터셋	63
이미지	64
라벨	64
소프트맥스 회귀(Softmax regression)	65
모델 정의	65
코스트(비용) 함수	66
텐서플로우로 구현	67
데이터 로딩	68
모델 정의	68
코스트함수와 옵티마이저 정의	68
세션 초기화	69
트레이닝 시작	69
결과값 출력	70
모델 검증	70
6. 딥러닝과 컨볼루션 네트워크	72
딥러닝의 개념과 배경	72
인경 신경망 알고리즘의 기본 개념	72
Perceptron	74
Perceptron의 XOR 문제	75
MLP (Multi Layer Perceptron) 다중 계층 퍼셉트론의 등장	76
Back Propagation 을 이용한 MLP 문제 해결	77
Back Propagation 문제와 ReLu를 이용한 해결	77
뉴럴 네트워크의 초기값 문제	78

딥러닝	78
컨볼루셔널 네트워크	78
컨볼루셔널 레이어 (Convolutional Layer)	79
필터 (Filter)	80
필터 개념 이해	80
다중 필터의 적용	81
Stride	82
Padding	83
필터는 어떻게 만드는 것일까?	84
Activation function	84
풀링 (Sub sampling or Pooling)	85
Max pooling (맥스 풀링)	86
컨볼루셔널 레이어	86
Fully connected Layer	87
Softmax 함수	87
Dropout 계층	88
이렇게 복잡한데 어떻게 구현을 하나요?	90
구현은 할 수 있겠는데, 그러면 이 모델은 어떻게 만드나요?	91
결론	92
그림 출처 및 참고 문서	93
7. 컨볼루셔널 네트워크를 이용한 MNIST 의 다항 분류 모델 구현	94
MNIST CNN 모델	94
입력 데이터	94
컨볼루셔널 계층	95
마지막 풀리 커넥티드 계층	95
학습(트레이닝) 코드	95
데이터 로딩 파트	97
첫번째 컨볼루셔널 계층	97
필터의 정의	98
필터 적용	99
스트라이드 (Strides)	99
패딩 (Padding)	100
활성함수 (Activation function)의 적용	101
Max Pooling	101
행렬의 차원 변환	102
두번째 컨볼루셔널 계층	103
풀리 커넥티드 계층	104
비용 함수 정의	105

학습	106
학습 결과 저장	108
숫자 예측하기	108
모델 로딩	108
그래프 구현	110
변수 데이터 로딩	111
HTML을 이용한 숫자 입력	111
입력값 판정	112
예측	113
그래프로 표현	113
8. 텐서플로우 고급 개념	115
CSV 데이터 파일 읽기와 큐의 개념	115
피딩 (Feeding) 개념 복습	115
텐서플로우 큐에 대해서	116
Queue 생성	117
Queue Runner 생성	117
Coordinator 생성	118
Queue Runner용 쓰레드 생성	118
큐 사용	118
쓰레드 정지	118
멀티 쓰레드	118
파일에서 데이터 읽기	120
파일명 목록 읽어오기	120
파일명 Shuffling	122
파일에서 데이터 읽기 (Reader)	123
읽은 데이터를 디코딩 하기 (Decoder)	124
예제	125
텐서플로우에서 배치 데이터 처리	126
데이터 포맷	126
배치 처리 코드	126
메인 코드	127
학습 데이터를 위한 TFRecord 파일 포맷	129
TFRecord 파일 생성	129
TFRecord에서 데이터 읽기	132
텐서보드를 이용한 학습 과정 시각화	134
Histogram	138
9. 컨볼루션 네트워크를 이용하여 얼굴인식 모델을 만들어 보자	140
얼굴 데이터 수집 하기	140

데이타 정제 하기	142
얼굴 추출 프로그램 사용 방법	143
결과 파일 및 디렉토리 구조	144
구글 클라우드 VISION API 사용 준비	144
SERVICE ACCOUNT 키 만들기	145
예제 코드	146
detect_face	150
crop_face	151
개선점	152
얼굴 인식 모델을 만들어보자	152
환경	152
준비된 데이타	153
컨볼루셔널 네트워크 모델	153
코드 설명	154
파일에서 데이타 읽기	154
get_input_queue	154
read_data()	155
read_data_batch()	155
모델 코드	156
Convolutional 계층	156
마지막 계층	157
전체 네트워크 모델 정의	158
모델 학습	159
학습된 모델로 예측하기	163
모델 로딩 하기	163
예측하기	163
얼굴 영역 추출하기	163
얼굴 영역을 크롭하기	164
크롭된 이미지를 텐서플로우에서 읽는다.	165
클라우드를 이용하여, 학습하기	167
구글 클라우드	167
CloudML을 이용하여 학습하기	168
코드 수정	168
디렉토리 구조	170
명령어	170
학습 모니터링	171
클라우드를 이용하여 학습된 모델을 예측하기	173
Export된 모델을 CloudML에 배포하기	173
배포된 모델로 예측 (Prediction)하기	177

10. 텐서플로우 Object Detection API를 이용한 이미지 인식	179
Object Detection API 설치	179
설치 및 테스트	180
Protocol Buffer 설치	180
파이썬 라이브러리 설치	180
Object Detection API 다운로드 및 설치	180
Protocol Buffer 컴파일	180
PATH 조정하기	180
테스팅	181
사용하기	181
Export 된 모델 다운로드	183
기타 파일들	184
다운로드된 모델과 라벨맵 로딩	184
참고 자료	185
Object Detection API에 애완동물 사진을 학습 시켜 보자	185
학습 데이터 다운로드 받기	186
TFRecord 파일 포맷으로 컨버팅 하기	187
학습 환경 준비하기	187
학습 데이터 업로드 하기	187
학습된 모델 다운로드 받아서 업로드 하기	188
설정 파일 변경하기	188
텐서플로우 코드 패키징 및 업로드	188
학습하기	189
학습 진행 상황 확인하기	191
학습된 모델을 Export 하기	192
참고 자료	195
Object Detection API로 연예인 얼굴 학습 하기	195
학습용 데이터 데이터 생성 및 준비	195
체크포인트 업로드	197
클래스의 수	198
학습 데이터 파일 명 및 라벨명	198
테스트 데이터 파일명 및 라벨 파일명	198
코드 패키징	199
모니터링	199
결과	199
11. 텐서 플로우 하이레벨 API	201
하이레벨 API를 쓰면 장점	202
모델 개발이 쉽다	202

스케일링이 용이하다	202
배포가 용이하다	202
텐서플로우 하이레벨 API	203
tf.layers	203
Estimator	203
Estimator 예제	204
학습용 데이터	205
모델 코드	206
데이터 리더	206
모델 정의	207
Custom Estimator	207
입력 인자에 대한 설명	208
Estimator 에서 하는 일	208
데이터 입력 처리	209
네트워크 정의	209
Loss 함수 정의	210
Training Op 정의	210
전체 코드	211
데이터 입력	212
네트워크 정의	212
예측 모드에서는 prediction 값을 리턴해야 하기 때문에, 먼저 예측값을 output_layer에서 나온 값으로, 행렬 차원을 변경하여 저장하고, 만약에 예측 모드 tf.estimator.ModeKeys.PREDICT일 경우 EstimatorSpec에 prediction 값을 넣어서 리턴한다. 이때 dict 형태로 prediction 결과 이름을 age로 값을 predictions 값으로 채워서 리턴한다.	213
Loss 함수 정의	213
Training OP 정의	213
실행	213
12. Custom Estimator 를 이용한 오토인코더구현	215
데이터 전처리	217
학습 코드 구현	217
데이터 입력부	217

```
def input_fn(filename,batch_size=100):
    filename_queue = tf.train.string_input_producer([filename])

    image,label = read_and_decode(filename_queue)
    images,labels = tf.train.batch(
        [image,label],batch_size=batch_size,
        capacity=1000+3*batch_size)
    #images : (100,784), labels : (100,1)
```

return {'inputs':images},labels	218
모델 구현부	218
Estimator 생성	220
실험 (Experiment) 구현	221
검증 코드 구현	221
Export 된 모듈 로딩	222
테스트 코드 구현	222
테스트 코드를 웹으로 구현	223
13. 오토인코더를 이용한 신용카드 비정상 거래 탐지	227
데이터 분석	227
데이터 전처리	230
데이터 정규화	230
데이터 분할	232
데이터 합치기	232
셔플링	233
학습 및 결과	233
결론	234
14. 머신 러닝 파이프 라인 아키텍처	235
미니 프로젝트를 시작하다	235
자동화와 스케일링의 필요성	235
아이 체중 예측 모델을 통한 파이프라인에 대한 이해	236
데이터 전처리를 스케일링하다.	237
머신러닝 파이프라인 아키텍처와 프로세스	238
파이프라인 개발 프로세스	238
아키텍처	239
Inputs	239
Pre processing & Asset creation	241
Train	241
Predict	242
Dataflow Orchestration	242
아직도 갈길은 멀다.	242

1. 머신러닝의 개요

들어가기에 앞서서

몇년전부터 빅데이터와 머신러닝이 유행하면서 이분야를 공부해야겠다고 생각을 하고 코세라의 Andrew.NG 교수님의 강의도 듣고, 통계학 책도 보고, 수학적인 지식이 부족해서 고등학교 수학 참고서도 봤지만, 도저히 답이 나오지 않는다. 머신 러닝에 사용되는 알고리즘은 복잡도가 높고 일반적인 수학 지식으로 이해조차 어려운데, 실제 운영 시스템에 적용할 수 있는 수준의 알고리즘은 석박사급의 전문가적인 지식이 아니면 쉽게 만들 수 없는 것으로 보였다. 예를 들어 인공지능망(뉴럴네트워크:Neural Network) 알고리즘에 대한 원리는 이해할 수 있지만, 실제로 서비스에 사용되는 알고리즘을 보니 보통 60~90개의 계층으로 이루어져 있는데, (그냥 복잡하다는 이야기로 이해하면 됨) 이런 복잡한 알고리즘을 수학 초보자인 내가 만든다는 것은 거의 불가능에 가까워 보였고, 이런것을 만들기 위해서 몇년의 시간을 투자해서 머신러닝 전문가로 커리어패스를 전환할 수 는 있겠지만 많은 시간과 노력이 드는데 반해서, 이미 나에게는 소프트웨어 개발과 백엔드 시스템이라는 전문분야가 있어쑈.

그래도 조금씩 보다보니, 머신 러닝에서 소개되는 알고리즘은 주로 사용되는 것은 약 20개 내외였고, 이미 다 정형화 되어 있어서 그 알고리즘을 만들어내기보다는, 가져다 쓰기만 하면 될 것 같다는 느낌이 들었다. 아직 많이 보지는 못했지만, 실제로 머신 러닝 기반의 시스템들은 나와 있는 알고리즘을 코드로 옮겨서 운영 환경에 올리는 경우가 대부분이었다.

비유를 하자면 우리가 복잡한 해쉬 리스트나, 소팅 알고리즘을 모르고도 간단하게 프로그래밍 언어에 있는 라이브러리를 가져다 쓰는 것과 같은 원리라고나 할까? 그래서, 완벽하게 이해하고 만들기 보다는 기본적인 원리를 파악하고 이미 공개된 알고리즘과 특히 레퍼런스 코드를 가져다가 운영환경에다 쓸 수 있는 정도의 수준을 목표로 하기로 했다.

이제 아주 아주 초보적인 수준의 이해를 가지고, 구글의 텐서플로우 기반으로 머신러닝과 딥러닝을 공부하면서 내용을 공유하고자 한다. 글을 쓰는 나역시도 수포자이며 머신러닝에 대한 초보자이기 때문에, 설명이 부족할 수 도 있고, 틀린 내용이 있을 수 있음을 미리 알리고 시작한다. (틀린 내용은 알려주세요)

머신러닝

머신 러닝은 데이터를 기반으로 학습을 시켜서 몬가를 예측하게 만드는 기법이다. 통계학적으로는 추측 통계학 (Inferential statistics)에 해당하는 영역인데, 근래에 들어서 알파고와 같은 인공지능이나 자동 주행 자동차, 로봇 기술등을 기반으로 주목을 받고 있다.



<그림. 구글의 자동 주행 자동차>

간단한 활용 사례를 보면

- 학습된 컴퓨터에 의한 이메일 스팸 필터링
- 편지지의 우편번호 글자 인식
- 쇼핑몰이나 케이블 TV의 추천 시스템
- 자연어 인식
- 자동차 자율 주행

등을 볼 수 있다.

이러한 시나리오는 지속적인 샘플 데이터를 수집 및 정제하고 지속적으로 알고리즘을 학습해나감에 따라서 최적의 알고리즘을 찾아나가도록 한다.

쇼핑몰의 추천 시스템의 경우 사용자의 구매 패턴을 군집화하여 유사한 패턴을 찾아냄으로써 적절한 상품을 추천하는데,

예를 들어 30대 남성/미혼/연수입 5000만원/차량 보유한 사용자가 카메라,배낭등을 구매했을 경우 여행 상품을 구매할 확률이 높다는 것을 학습하였을때,

이러한 패턴의 사용자에게 여행 상품을 추천해주는 것과 같은 답을 제공할 수 있다.

지도 학습과 비지도 학습

머신러닝은 학습 방법에 따라서 지도 학습 (Supervised Learning)과 비지도 학습 (Unsupervised Learning)으로 분류될 수 있다.

지도 학습 (Supervised Learning)



예를 들어 학생에게 곱셈을 학습 시킬때,
“ $2*3=6$ 이고, $2*4=8$ 이야, 그러면 $2*5=$ 얼마일까? “
처럼 문제에 대한 정답을 주고 학습을 한 후,
나중에 문제를 봤을때 정답을 구하도록 하는 것이 지도 학습 (Supervised Learning)이다.

비지도 학습 (Unsupervised learning)

반대로 비지도 학습은 정답을 주지않고 문제로만 학습을 시키는 방식을 비지도 학습이라고 한다.
예를 들어 영화에 대한 종류를 학습 시키기 위해서, 연령,성별과 영화의 종류 (액션, 드라마, SF)를 학습 시켰을때,
이를 군집화 해보면 20대 남성은 액션 영화를 좋아하고 20대 여성은 드라마 영화를 좋아 하는 것과 같은 군집된 결과를 얻을 수 있고,
이를 기반으로 20대 남성이 좋아하는 영화의 종류는 유사한 군집의 결과인 "액션 영화"라는 답을 내게 되날.

여기서 문제에 대한 답을 전문적인 용어로 이야기 하면 라벨된 데이터 (Labeled data)라고 한다.

머신러닝의 대표적인 문제 Regression과 Classification 문제

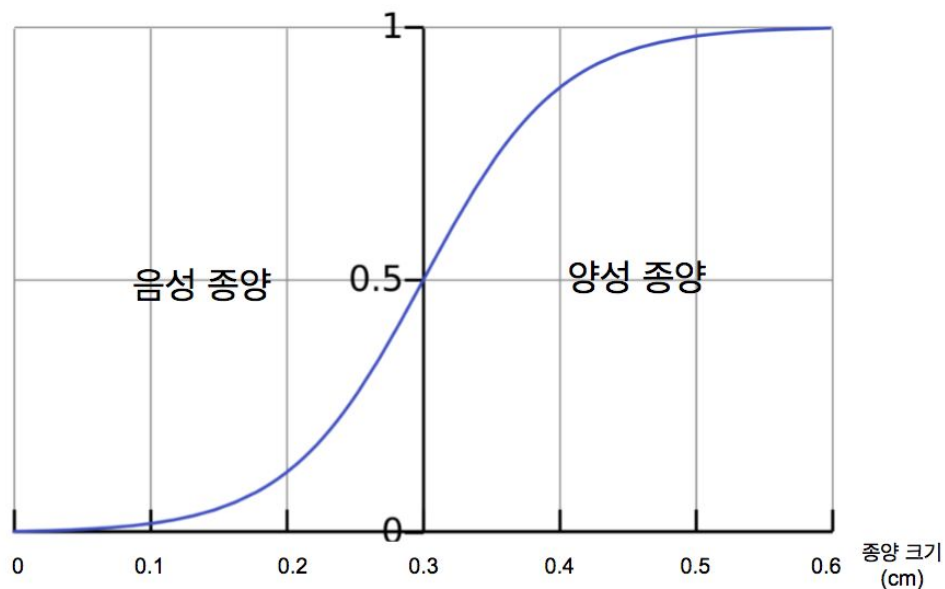
머신러닝을 이용해서 해결하는 문제의 타입은 크게 regression과 classification 문제 두가지로 대표가 된다.

Classification

Classification은

입력값에 대한 결과값이 연속적이지 않고 몇개의 종류로 딱딱 나뉘서 끊어지는 결과가 나오는 것을 이야기 한다. 예를 들어 종양의 크기가 0.3cm 이상이고 20대이면, 암이 양성,

또는 종양의 크기가 0.2cm 이하이고 30대이면,
암이 음성과 같이 결과 값이 "양성암/음성암"과 같이 두개의 결과를 갖는 것이 예가 된다.



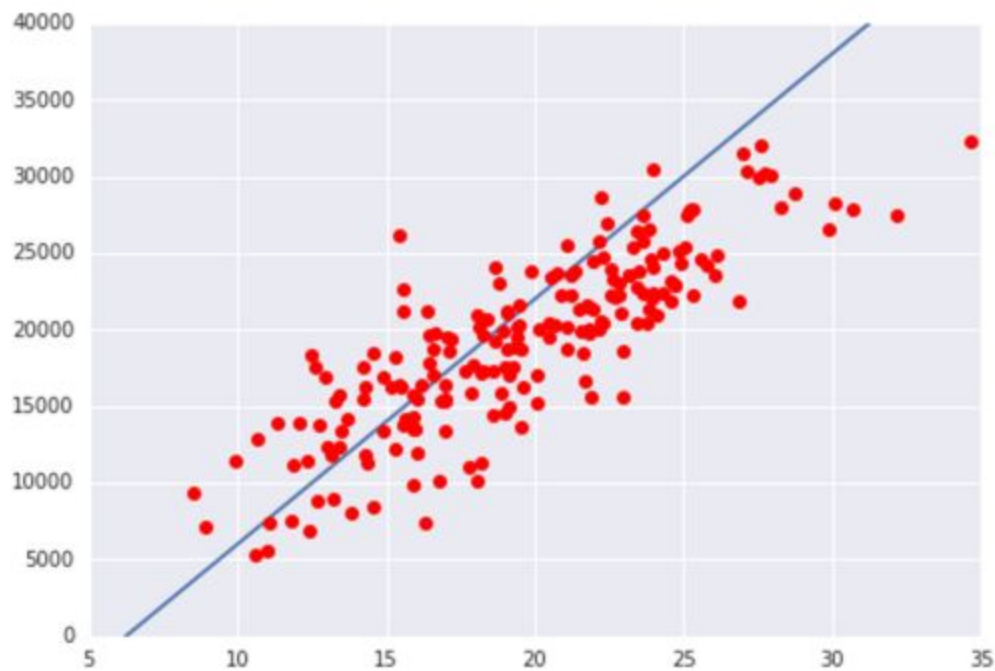
<종양 크기에 따른, 암의 양성/음성 여부에 대한 그래프>

또 다른 예로는 사진을 업로드 했을때, 사진의 물체를 인식할때 "이사진은 개이다."
"이사진은 고양이이다." 처럼 특정 종류에 대한 결과값이 나오는 것 역시 Classification
문제로 볼 수 있다.



Regression

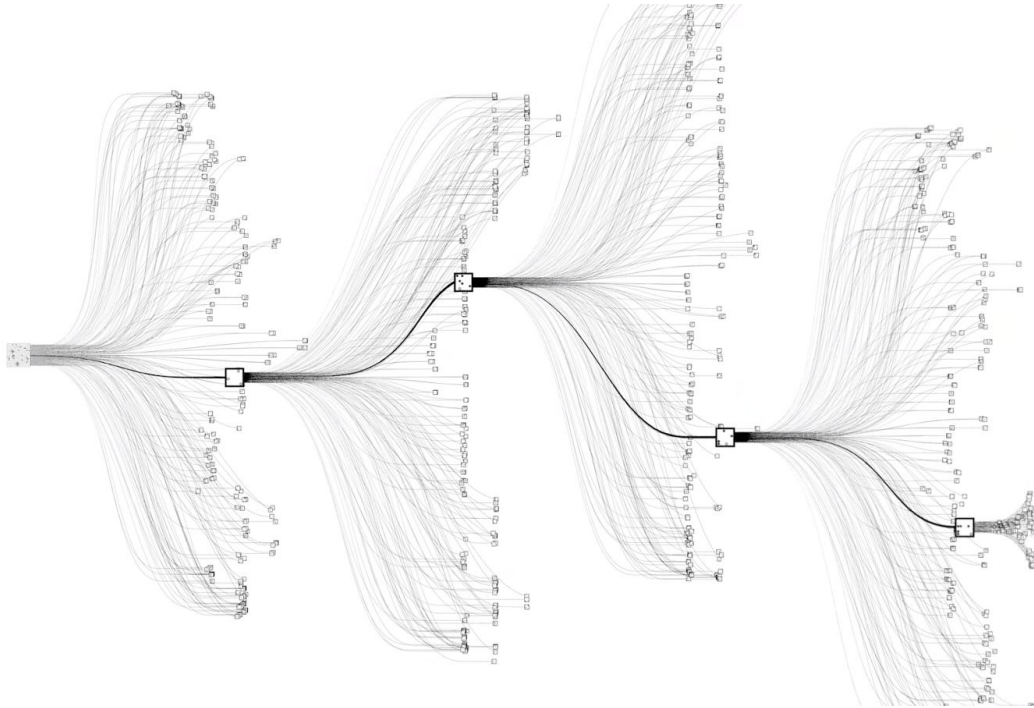
Regression 문제는 결과값이 연속성을 가지고 있을때 Regression 문제라고 한다.
즉 택시의 주행거리에 따른 요금과 같은 문제인데, 변수 택시 주행 거리에 대해서,
결과 택시 값이 기대 되는 경우로 변수와 결과값이 연속적으로 이루어 지는 경우를 말한다.



<그림. 주행 거리에 따른 택시비 >

머신 러닝과 딥러닝

이러한 머신 러닝의 분야중, 인공 지능망 (뉴럴 네트워크 / Artificial neural network)라는 기법이 있는데, 사람의 뇌의 구조를 분석하여, 사람 뇌의 모양이 여러개의 뉴런이 모여서 이루어진것 처럼, 머신 러닝의 학습 모델을 두뇌의 모양과 같이 여러개의 계산 노드를 여러 층으로 연결해서 만들어낸 모델이다.



<알파고에 사용된 뉴럴네트워크 구조>

이 모델은 기존에 다른 기법으로 풀지 못하였던 복잡한 문제를 풀어낼 수 있었지만, 계층을 깊게 하면 계산이 복잡하여 연산이 불가능하다는 이유로 그간 관심을 가지고 있지 못했다

캐나다의 CIFAR (Canadian Institute for Advanced Research) 연구소에서 2006년에 Hinton 교수가 "A fast learning algorithm for deep belief nets" 논문을 발표하게 되는데, 이 논문을 통해서 뉴럴네트워크에 입력하는 초기값을 제대로 입력하면 여러 계층의 레이어에서도 연산이 가능하다는 것을 증명하였고, 2007년 Yosua Bengio 라는 분이 "Greedy Layer-Wise training of deep network"

라는 논문에서 깊게 신경망을 구축하면 굉장히 복잡한 문제를 풀 수 있다는 것을 증명해냈다.

이때 부터 뉴럴네트워크가 다시 주목을 받기 시작했는데,

이때 뉴럴 네트워크라는 모델을 사람들에게 부정적인 인식이 있었기 때문에, 다시 이 뉴럴 네트워크를 딥러닝 (Deep learning)이라는 이름으로 다시 브랜딩을 하였다.

그 이후에 IMAGENET 챌린지라는 머신러닝에 대한 일종의 컨테스트가 있는데,

이 대회는 이미지를 입력하고 머신 러닝을 통해서 컴퓨터가 이미지의 물체등을 인식할 수 있게 하는 대회로, 머신 러닝 알고리즘의 정확도를 측정하는 대회이다. 이 대회에서 2012년 Hinton 교수님 랩에 있던 Alex

라는 박사 과정의 학생이 딥러닝 기반의 머신 러닝 알고리즘으로 혁신적인 결과를 내었고 지금은 이 딥러닝이 머신 러닝의 큰 주류중의 하나로 자리잡게 되었다.



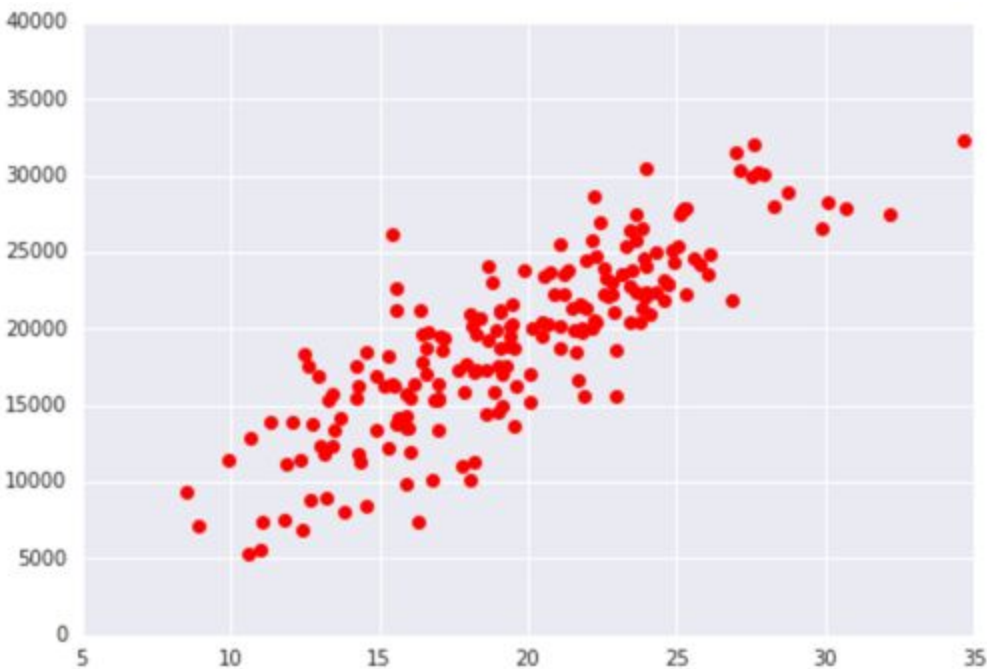
<이미지넷에서 사용되는 이미지>

2. 선형회귀를 통한 머신 러닝의 개념 이해

거리에 따른 택시 요금 문제

머신러닝이란 무엇일까? 개념 이해를 돕기 위해서 선형 회귀 (Linear Regression)이라는 머신러닝 모델을 보자 먼저 선형 회귀 (Linear regression)이 무엇인지 부터 이해를 해야 하는데, 예를 들어서 설명해보자, 택시 요금을 예로 들어보자, 택시 요금은 물론 막히냐 마냐에 따라 편차가 있지만, 대부분 거리에 비례해서 요금이 부과된다. 거리별 요금을 그래프로 나타내보면 대략 다음과 같은 분포를 띄게 된다

원본 데이터의 거리를 x_data 그리고, 그 거리에서 측정되니 택시 요금을 y_origin 이라고 하자.

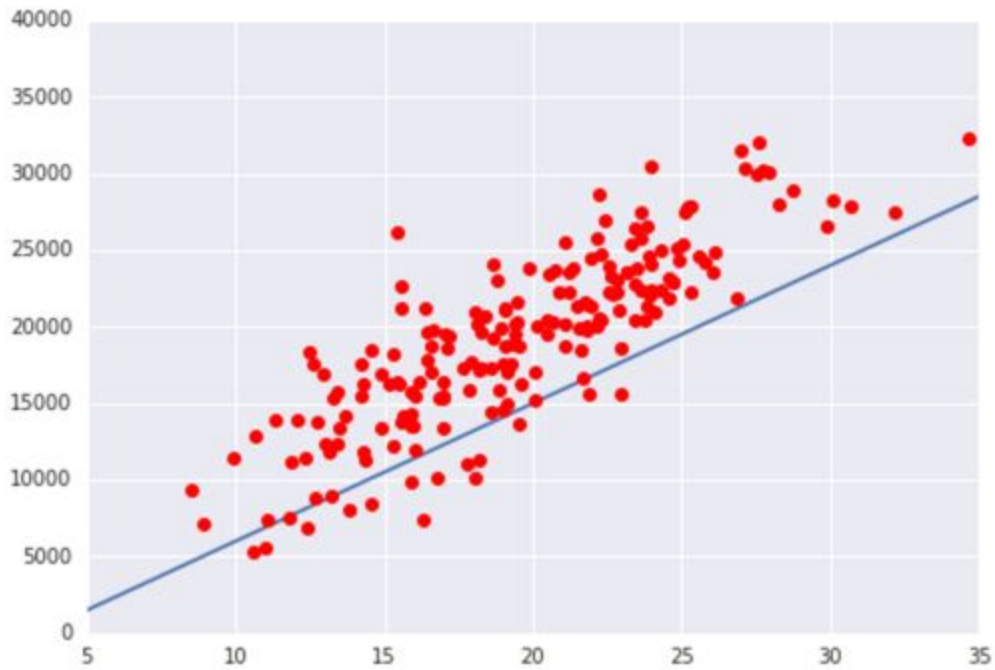


가설 (Hypothesis) 정의

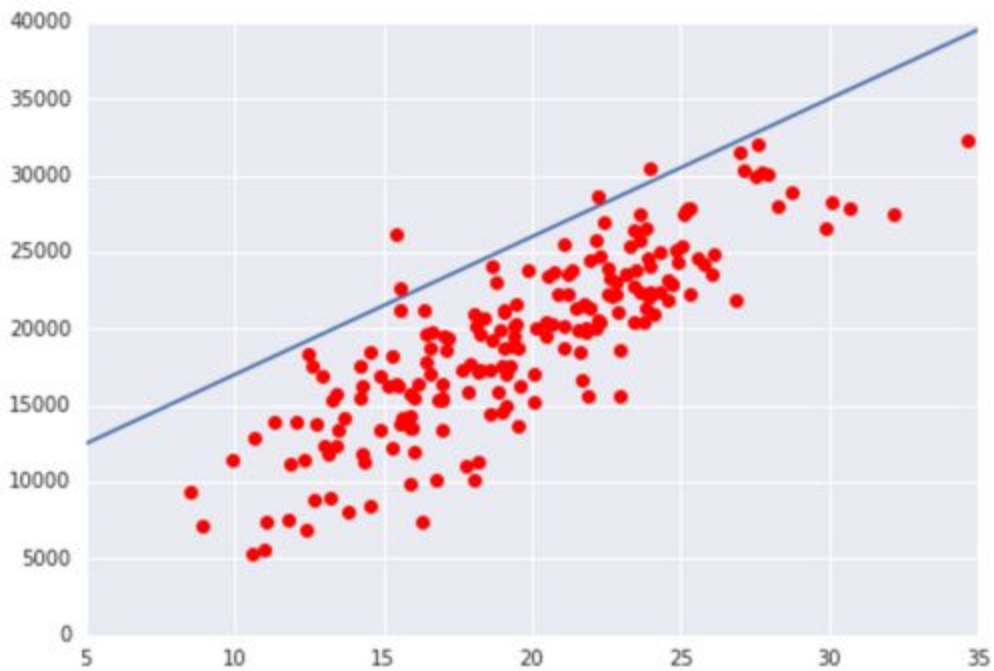
거리와 요금이 서로 비례하기 때문에, 거리(x_data)와 요금(y_data)간의 상관 관계는 다음과 같이 일차 방정식과 형태의 그래프를 그리게 된다고 가정하자.

$$y_data = Wx_data + b$$

이 일차 방정식 형태로 대충 1차원 그래프를 그려보자 같은 형태로 아래와 같이 그래프를 그려봤다.



그래프를 그려보니 그래프의 각이 안맞는것 같다. 그래프의 각도와 높이를 보정해보자

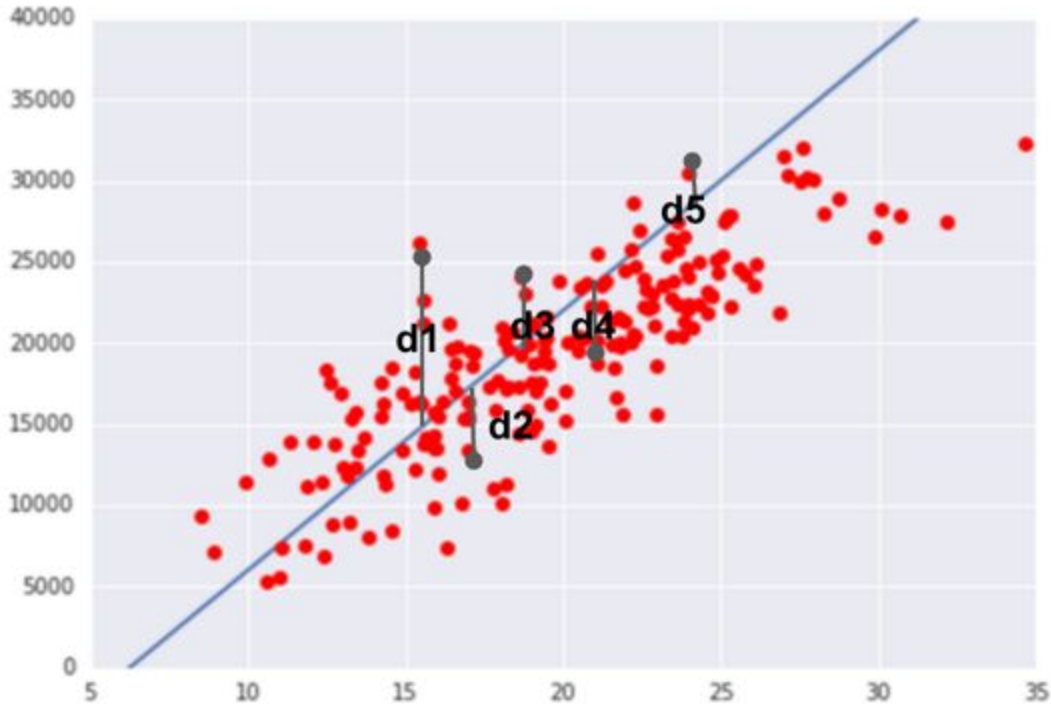


그래프를 보정했지만 또 안 맞는 것 같다. 그렇다면 최적의 그래프의 각도 W 와, 높이 B 는 어떻게 찾아야 하는 것일까?

코스트(비용) 함수

우리가 구하고자 하는 그래프는

실제 값에서 그래프의 값까지 차이가 가장 작은 값을 구하고자 하는 것이다. 아래 그림을 보자, 아래와 같이 $y_data = Wx_data + b$ 와 같은 그래프를 그렸다고 하자.



원래 값에서 우리가 예측한 값의 차이는

$$(\text{원래값과 계산된 값의 차이}) = \text{측정값} - \text{그래프의 값}$$

인데, 차이를 d 라고 하자. 그리고 그래프에 의해서 계산된 값은 y_data 라고 하면 택시 거리 x_data 에서 원래 측정된 값을 y_origin 라고 해서 수식으로 나타내면,

$$d = y_data - y_origin$$

이 된다. 이때 측정값은 여러개가 있기 때문에 n 이라고 하면

n 번째 측정된 택시비와 산식에 의해서 예측된 값의 차이는 d_n 이 된다.

$$d_n = y_data_n - y_origin_n$$

즉 우리가 구하고자 하는 값은 d_n 의 합이 최소가 되는 W 와 b 의 값을 구하고자 하는 것이다.

다르게 설명하면 실제 측정값과,

예측한 값의 차이가 최소가 되는 W 와 b 를 구하고자 하는 것이다.

d_n 은 위의 그래프에서 처럼 그래프 위에도 있을 수 있지만 (이경우 d_n 은 양수), 그래프 아래에도 있을 수 있기 때문에, (이경우 d_n 은 음수). 합을 구하면,

예측 선에서의 실측값 까지의 거리의 합이 되지 않기 때문에,
 d_n 에 대한 절대값을 사용한다고 하자.
 그리고 n 이 측정에 따라 여러개가 될 수 있기 때문에, 평균을 사용하자.

$$(ABS(d_1)+ABS(d_2)+ABS(d_3)+.....+ABS(d_n)) / n$$

- ABS는 절대값

즉 우리가 구하고자 하는 W 와 b 는 위의 함수의 값이 최소가 되는 값을 구하면 된다.
 이렇게 측정된 값에서 연산된 값간의 차이를 연산하는 함수를 비용
 함수 또는 영어로 코스트 함수 (Cost function이라고 한다.

사람이 일일이 계산할 수 없이니 컴퓨터를 이용해서 $W=0.1, 0.2, 0.3,$ $b=0.1, 0.2, 0.3,$
 식으로 넣어보고 이 코스트 함수가 가장 최소화되는 W 와 b 의 값을 찾을 수 있다.

옵티마이저 (Optimizer)

코스트 함수의 최소값을 찾는 알고리즘을 옵티마이저(Optimizer)라고 하는데,
 상황에 따라 여러 종류의 옵티마이저를 사용할 수 있다. 여기서는 경사 하강법 (Gradient
 Descent) 라는 옵티마이저에 대해서 소개하도록 하겠다.

경사 하강법

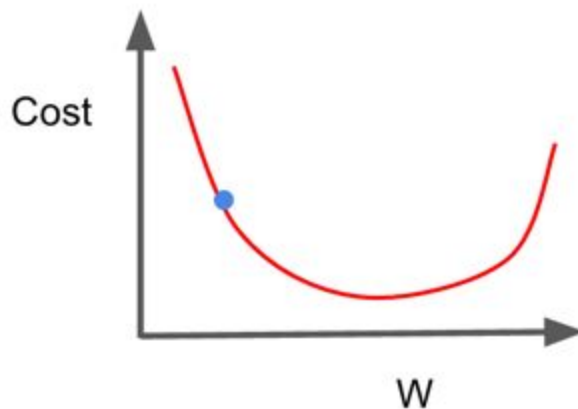
그러면 W 와 b 를 구할때 W 와 b 를 어떤식으로 증가 또는 감소 시켜서 코스트
 함수의 최소값을 가장 효율적으로 찾아낼 수 있을까? 위에서 언급한것 처럼 W 를 0.0에서 부터).
 0.1씩 증가시켜나가고 b 도 같이 0.0에서 부터 1씩 증가 시켜 나갈까? 무한한
 컴퓨팅 자원을 이용하면 되기는 하겠지만, 이렇게 무식하게 계산하지는 않는다.
 코스트 함수를 최적화 시킬 수 있는 여러가지 방법이 있지만, Linear
 regression의 경우에는 경사 하강법 (그레이언트 디센트 : Gradient
 descent)라는 방식을 사용한다.
 경사하강법에 대해서는 자세하게 알필요는 없고 "대략 이런 개념을 사용하는구나" 하는 정도만
 알면 된다.

경사 하강법을 사용하기 위해서는 위의 코스트 함수를, 측정값과 예측값의
 절대값의 평균이 아니라 평균 제곱 오차라는 함수를 사용한다.
 이 함수는 형식으로 정의되는데, 평균 제곱 오차 함수 (Mean square error function)이라고 한다.

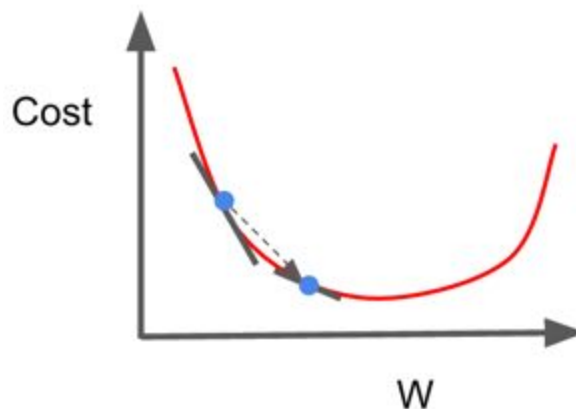
$$Cost = \text{Sum}(y_data_n - y_origin_n)^2 / n$$

풀어서 설명하면, n
 번째의 원래데이터(y_origin_n)와 예측 데이터(y_data_n)의 차이를 제곱(^2)해서, 이 값을 n 으로
 나눈 평균 값이다.
 즉 이 Cost가 최소가 되는 W 와 b 값을 구하면 된다.

편의상 W 하나만을 가지고 설명해보자. 위의 그래프를 W 와 b 에 대한 상관 그래프로 그려보면 다음과 같은 함수 형태가 된다.



이 그래프에서 W 에 대한 적정값에 대한 예측을 시작하는 점을 위의 그림에서 파란 점이라고 하면, 경사 하강법은 현재 W 의 위치에 대해서, 경사가 아래로 되어 있는 부분으로 점을 움직이는 방법이다. 어느 방향으로 W 를 움직이면 $Cost$ 값이 작아지는지는 현재 W 위치에서 비용 함수를 미분하면 된다. (고등학교 수학이 기억이 나지 않을 수 있겠지만 미분의 개념은 그래프에서 그 점에 대한 기울기를 구하는 것이다. 미분의 개념은 <https://www.youtube.com/watch?v=oZyvmtqLmLo&feature=youtu.be> 10분내에 이해하는 미분 이라는 비디오를 보면 쉽게 이해할 수 있다.)



이렇게, 경사를 따라서 아래로 내려가다 보면 $Cost$ 함수가 최소화 되는 W 값을 찾을 수 있다. 이렇게 경사를 따라서 하강 (내려가면서) 최소값을 찾는다고 하여 경사 하강법이라고 한다.

학습

코스트 함수가 정의 되었으면 모델을 이 코스트 함수를 가지고 최적화 한다.

실제 데이터 x_data_n 과 y_data_n 을 넣어서 경사하강법에 의해서 코스트 함수가 최소가 되는 W 와 b 를 구한다. 이 작업은 W 값을 변화시키면서 반복적으로 x_data_n 로 계산을 하여,

실제 측정 데이터와 가설에
의해서 예측된 결과값에 대한 차이를 찾아내고 최적의 W와 b값을 찾아낸다.
이 값을 찾아내서 적용한 가설이 바로 모델이다.

모델 평가(테스팅)

테스트 데이터를 7:3으로 나눠서 7로 학습. 학습후에 3으로 테스트.
테스트를 한후 모델의 정확도를 측정하다.

예측

학습 과정에 의해서 최적의 W와 b를 찾았으면 이제, 이 값들을 이용해서 예측 해보자
학습에 의해서 찾아낸 W가 1600, b가 2000이라고 하면,
앞의 가설에서 정의한 함수는 $Wx + b$ 였기 때문에, 예측 함수는

$$\begin{aligned} y &= Wx + b \\ \text{거리에 따른 택시비} &= W * (\text{거리}) + b \\ \text{거리에 따른 택시비} &= 1600 * (\text{거리}) + 2000 \end{aligned}$$

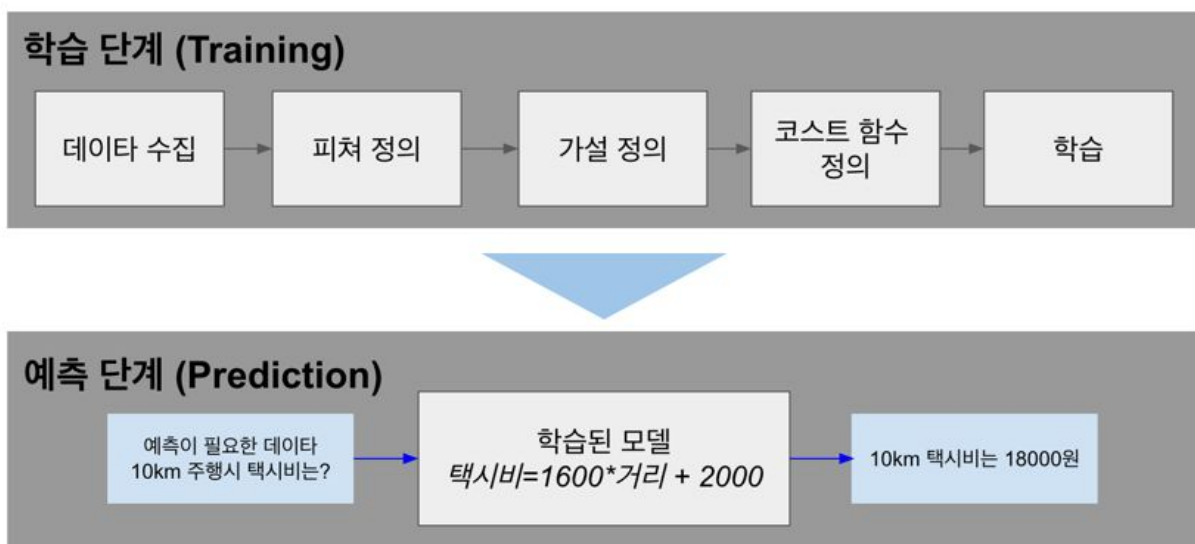
이 되고, 이를 **학습된 모델** 이라고 한다.

이제 예측을 수행해보자, 거리가 10km일 때 택시비는 얼마일까? 공식에 따라
 $\text{택시비} = 1600 * 10\text{km} + 2000$
으로, 18000원이 된다.

머신 러닝의 순서

지금까지 택시 거리와 택시비에 대한 문제를 가지고 머신 러닝에 대한 기본 원리를 살펴보았다.
이를 요약해서 머신 러닝이란 것이 어떤 개념을 가지고 있는지 다시 정리해보자.

기본 개념은 데이터를 기반으로해서 어떤 가설 (공식)을 만들어 낸 다음,
그 가설에서 나온 값이 실제 측정값과의 차이(코스트 함수)가 최소한의 값을 가지도록 변수에 대
한 값을 컴퓨터를 이용해서 찾은 후,
이 찾아진 값을 가지고 학습된 모델을 정의해서 예측을 수행 하는 것이다.



학습 단계

즉 모델을 만들기 위해서, 실제 데이터를 수집하고, 이 수집된 데이터에서 어떤 특징(피쳐)를 가지고 예측을 할것인지 피쳐들을 정의한 다음에, 이 피쳐를 기반으로 예측을 한 가설을 정의하고, 이 가설을 기반으로 학습을 시킨다.

예측 단계

학습이 끝나면 모델 (함수)가 주어지고, 예측은 단순하게, 모델에 값을 넣으면, 학습된 모델에 의해서 결과값을 리턴해준다.

지금까지 Linear regression 분석을 통한 머신러닝의 원리에 대해서 간략하게 알아보았다. 다음 다음장에서는 이 모델을 어떻게 프로그래밍 언어를 이용하여 학습을 시키고 운영을 하는지에 대해서 알아보도록 하겠다.

3. 텐서 플로우를 이용한 선형 회귀 모델 개발

앞에서 살펴본 선형 회귀(Linear regression)

머신 러닝 모델을 실제 프로그래밍 코드를 만들어서 학습을 시켜보자.

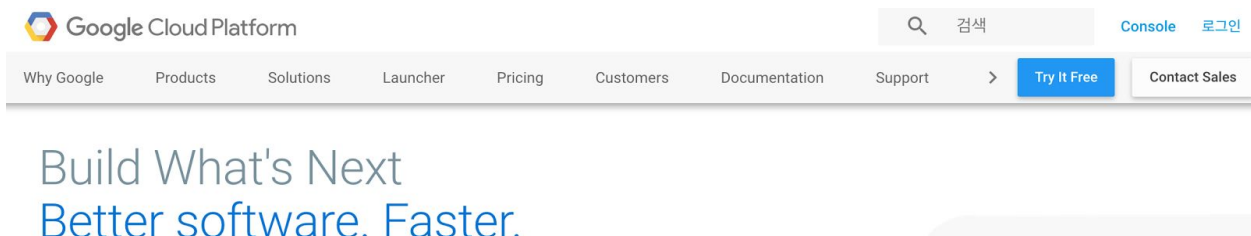
여러가지 언어를 사용할 수 있지만, 이 글에서는 텐서플로우를 기반으로 설명한다.

텐서플로우 개발 환경 셋업

텐서 플로우 개발 환경을 설정하는 방법은 여러가지가 있지만, 구글 클라우드의 데이타랩 (datalab) 환경을 사용하기로 한다. 텐서플로우 환경을 설정하려면 파이썬 설치 및 연관된 수학 라이브러리를 설치해야 하는 등 설치가 까다롭기 때문에, 구글 클라우드에서 제공하는 파이썬 노트북 (Jupyter 노트북 : <http://jupyter.org/>) 이 패키징 된 도커 이미지를 사용하기로 한다. 파이썬 노트북은 일종의 위키나 연습장 같은 개념으로 연산등에 필요한 메모를 해가면서 텐서 플로우나 파이썬 코드도 적어넣고 실행도 할 수 있기 때문에 데이터 관련 작업을 하기 매우 편리하다. 또한 도커로 패키징된 데이타랩 환경은 로컬에서나 클라우드 등 아무곳에서나 실행할 수 있기 때문에 편리하고 별도의 과금이 되지 않기 때문에 편리하게 사용할 수 있다.

구글 클라우드 계정 및 프로젝트 생성

GCP 클라우드를 사용하기 위해서는 구글 계정에 가입한다. 기존에 gmail 계정이 있으면 gmail 계정을 사용하면 된다. <http://www.google.com/cloud> 로 가서, 좌측 상단에 Try it Free 버튼을 눌러서 구글 클라우드에 가입한다.



다음 콘솔에서 상단의 Google Cloud Platform 을 누르면 좌측에 메뉴가 나타나는데, 메뉴 중에서 "결제" 메뉴를 선택한후 결제 계정 추가를 통해서 개인 신용카드 정보를 등록한다.



Google Cloud Platform



제품 및 서비스 필터링



홈

API

API 관리자



결제



Cloud Launcher



IAM 및 관리자

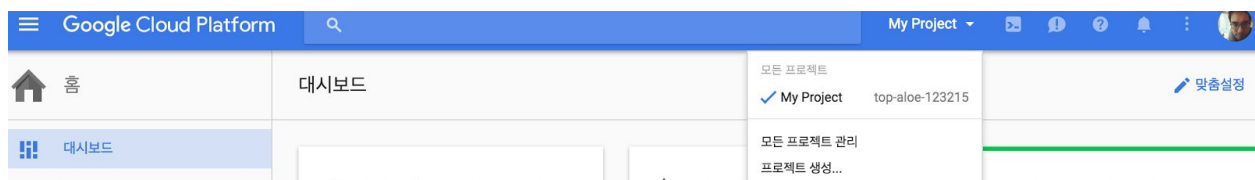
개인 신용 카드 정보를 등록해야 모든 서비스를 제한 없이 사용할 수 있다. 단 Trial의 경우 자동으로 한달간 300\$의 비용을 사용할 수 있는 크레딧이 자동으로 등록되니, 이 범위를 넘지 않으면 자동으로 결제가 되는 일이 없으니 크게 걱정할 필요는 없다.

프로젝트 생성

계정 생성 및 결제 계정 세팅이 끝났으면 프로젝트를 생성한다.

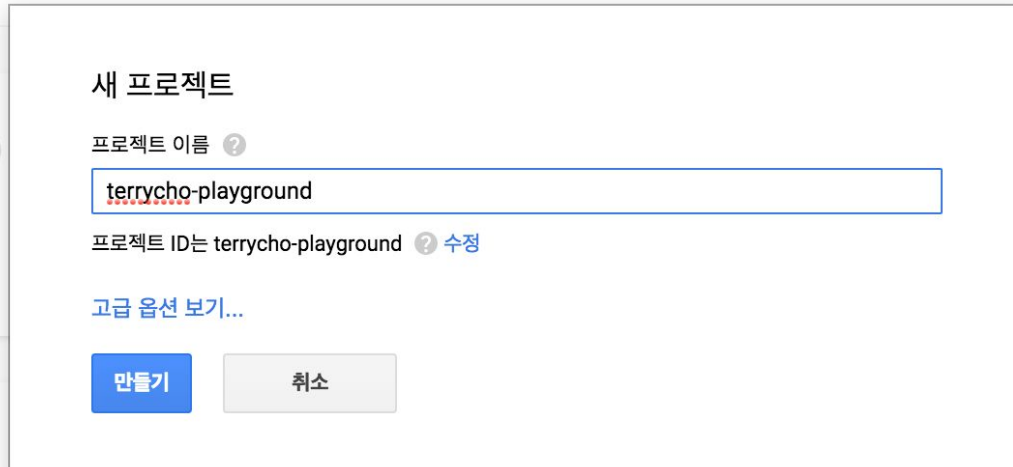
프로젝트는 VM이나 네트워크 자원, SQL등 클라우드 내의 자원을 묶어서 관리하는 하나의 집합이다. 여러 사람이 하나의 클라우드를 사용할때 이렇게 프로젝트를 별도로 만들어서 별도로 과금을 하거나 각 시스템이나 팀별로 프로젝트를 나눠서 정의하면 관리하기가 용이하다.

화면 우측 상단에서 프로젝트 생성 메뉴를 선택하여 프로젝트를 생성한다.



프로젝트 생성 버튼을 누르면 아래와 같이 프로젝트 이름을 입력 받는 창이 나온다. 여기에 프로젝트명을 넣으면 된다.

보드



도커 설치 (수정 필요)

이 글에서는 로컬 맥북 환경에 데이터랩을 설치하는 방법을 설명한다.

데이터 랩은 앞에서 언급한것과 같이 구글 클라우드 플랫폼 상의 VM에 설치할 수 도 있고, 맥, 윈도우 기반의 로컬 데스크탑에도 설치할 수 있다. 각 플랫폼별 설치 가이드는

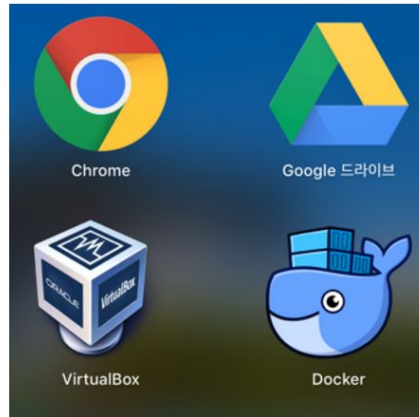
<https://cloud.google.com/datalab/docs/quickstarts/quickstart-local> 를 참고하기 바란다. 이

문서에서는 맥 OS를 기반으로 설치하는 방법을 설명한다.

데이터 랩은 컨테이너 솔루션인 도커로 패키징이 되어 있다. 그래서 도커 런타임을 설치해야 한다.

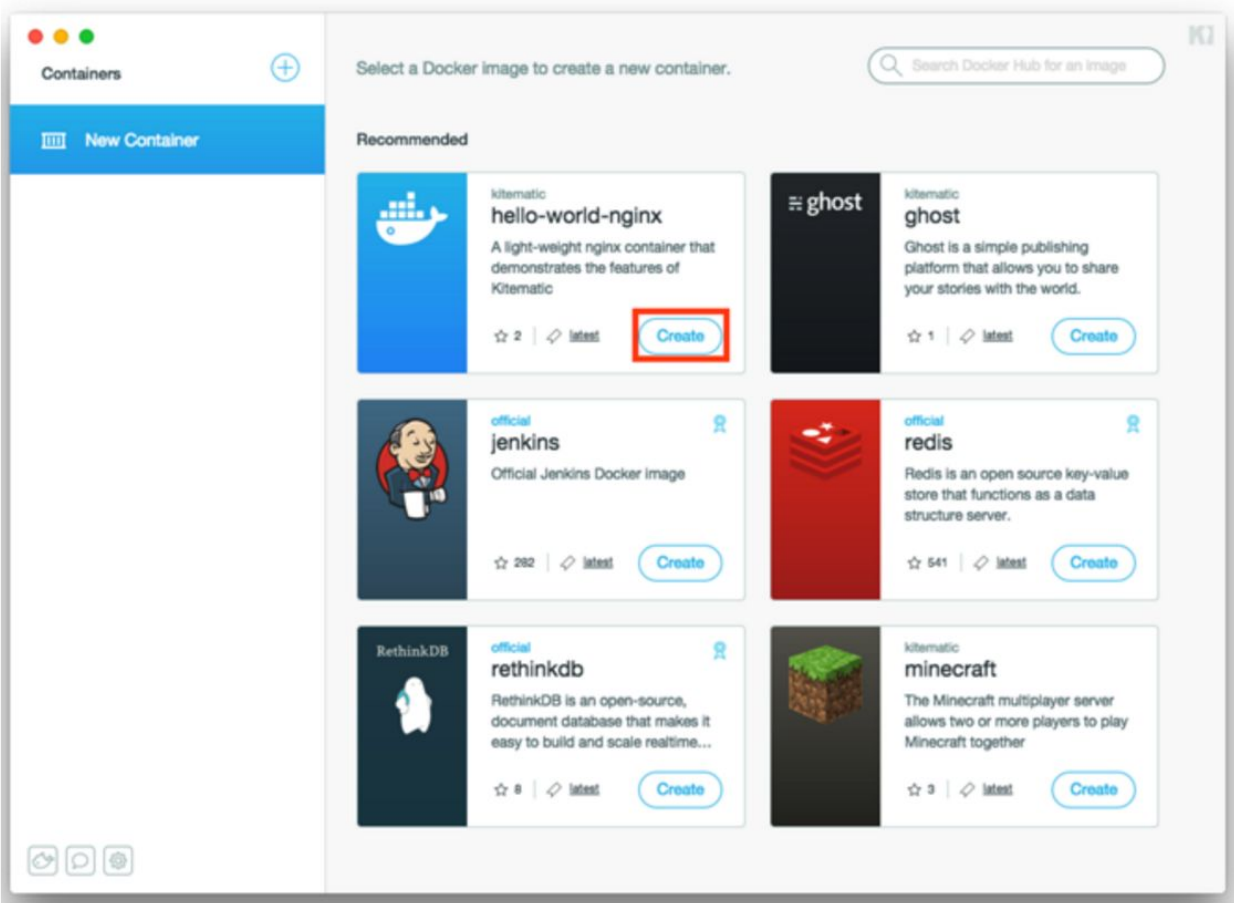
<https://www.docker.com/products/docker> 에서 도커 런타임을 다운 받아서 설치한다.

도커 런타임을 설치하면 애플리케이션 목록에 다음과 같이 고래 모양의 도커 런타임 아이콘이 나오는 것을 확인할 수 있다.



하나 주의할점이라면 맥에서 예전의 도커 런타임은 오라클의 버추얼 박스를 이용했었으나, 제반 설정등이 복잡하기 때문에, 이미 오라클 버추얼 박스 기반의 도커 런타임을 설치했다면 이 기회에, 도커 런타임을 새로 설치하기를 권장한다.

다음으로 도커 사용을 도와주는 툴로 Kitematic 이라는 툴을 설치한다. (<https://kitematic.com/>) 이 툴은 도커 컨테이너에 관련한 명령을 내리거나 이미지를 손쉽게 관리할 수 있는 GUI 환경을 제공한다.



구글 클라우드 데이터 랩 설치

Kitematic의 설치가 끝났으면 데이터랩 컨테이너 이미지를 받아서 실행해보자, Kitematic 좌측 하단의 “Docker CLI” 버튼을 누르면, 도커 호스트 VM의 셸 스크립트를 수행할 수 있는 터미널이 구동된다.



터미널에서 다음 명령어를 실행하자

```
docker run -it -p 8081:8080 -v "${HOME}:/content" \
```



```
-e "PROJECT_ID=terrycho-firebase" \  
gcr.io/cloud-datalab/datalab:local
```

데이타랩은 8080 포트로 실행이 되고 있는데, 위에서 8081:8080은 도커 컨테이너안에서 8080으로 실행되고 있는 데이타 랩을 외부에서 8081로 접속을 하겠다고 정의하였고, PROJECT_ID는 데이타랩이 접속할 구글 클라우드 프로젝트의 ID를 적어주면 된다.

여기서는 terrycho-firebase를 사용하였다.

명령을 실행하면, 데이타랩 이미지가 다운로드 되고 실행이 될것이다.

실행이 된 다음에는 브라우저에서 <http://localhost:8081>로 접속하면 다음과 같이 데이타랩이 수행된 것을 볼 수 있다.



학습하기

이제 텐서 플로우 기반의 머신러닝을 위한 개발 환경 설정이 끝났다.

이제 선형 회귀 모델을 학습 시켜보자

테스트 데이타 만들기

학습을 하려면 데이타가 있어야 하는데, 여기서는 랜덤으로 데이타를 생성해내도록 하겠다.

다음은 데이타를 생성하는 텐서 플로우코드이다.

텐서 플로우 자체에 대한 설명과 문법은 나중에 기회가 되면 별도로 설명하도록 하겠다.

```
import numpy as np  
num_points = 200  
vectors_set = []
```

```

for i in xrange(num_points):
    x = np.random.normal(5,5)+15
    y = x*1000+ (np.random.normal(0,3))*1000
    vectors_set.append([x,y])

x_data = [v[0] for v in vectors_set ]
y_data = [v[1] for v in vectors_set ]

```

for 루프에서 xrange로 200개의 샘플 데이터를 생성하도록 하였다.

x는 택시 주행거리로,

정규 분포를 따르는 난수를 생성하되 5를 중심으로 표준편차가 5인 데이터를 생성하도록 하였다. 그래프를 양수로 만들기 위해서 +15를 해주었다.

다음으로 y값은 택시비인데, 주행거리(x) * 1000 + 정규 분포를 따르는 난수로 중심값은 0, 그리고 표준편차를 3으로 따르는 난수를 생성한후, 이 값에 1000을 곱하였다.

x_data에는 x 값들을, 그리고 y_data에는 y값들을 배열형태로 저장하였다.

값들이 제대로 나왔는지 그래프를 그려서 확인해보자. 아래는 그래프를 그리는 코드이다.

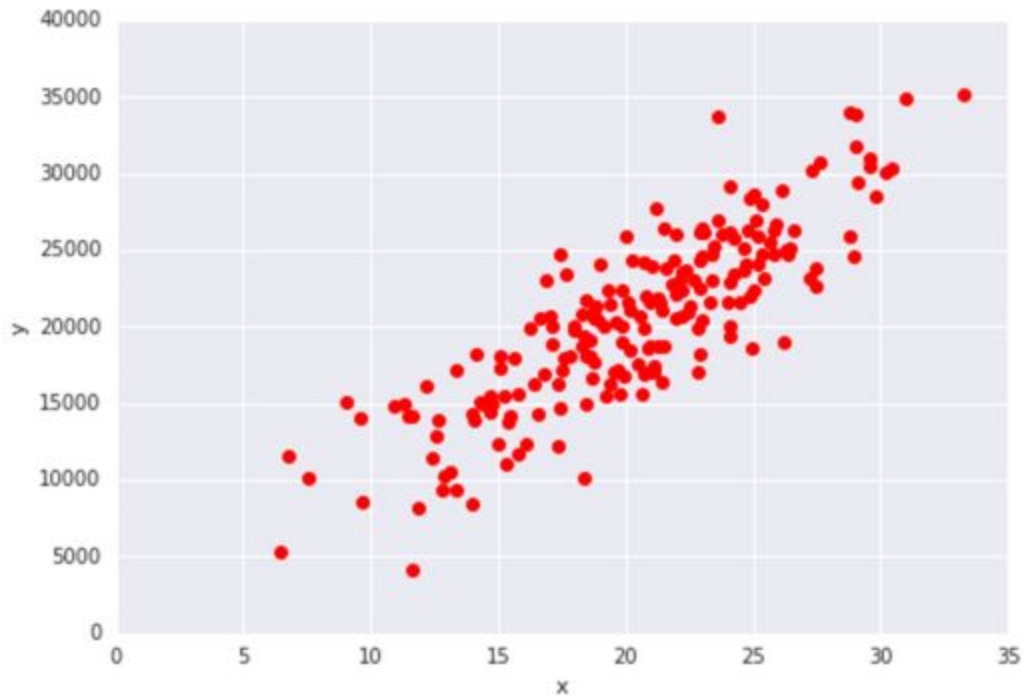
Pyplot이라는 모듈을 이용하여 plot이라는 함수를 이용하여 그래프를 그렸다. Y축은 0~40000, X축은 0~35까지의 범위를 갖도록 하였다.

```

import matplotlib.pyplot as plt
plt.plot(x_data,y_data,'ro')
plt.ylim([0,40000])
plt.xlim([0,35])
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()

```

그려진 그래프의 모양은 다음과 같다.



학습 로직 구현

이제 앞에서 생성한 데이터를 기반으로해서 선형 회귀 학습을 시작해보자. 코드는 다음과 같다.

```
import tensorflow as tf

W = tf.Variable(tf.random_uniform([1],-1.0,1.0))
b = tf.Variable(tf.zeros([1]))
y = W * x_data + b

loss = tf.reduce_mean(tf.square(y-y_data))
optimizer = tf.train.GradientDescentOptimizer(0.0015)
train = optimizer.minimize(loss)

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

for step in xrange(10):
    sess.run(train)
    print(step,sess.run(W),sess.run(b))
    print(step,sess.run(loss))

plt.plot(x_data,y_data,'ro')
plt.plot(x_data,sess.run(W)*x_data + sess.run(b))
```

```
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```

W의 초기값은 random_uniform으로 생성을 한다. 초기값은 -1.0~1.0 사이의 값으로 생성하도록 하였다.

(random_uniform 에서 첫번째 인자 [1]은 텐서의 차원을 설명하는데, 1은 1차원으로 배열과 같은 형태가 2는 2차원으로 행렬과 같은 형태, 3은 3차원 행렬 형태가 된다.)

다음 b는 tf.zeros([1])으로 정의했는데, 1차원 텐서로 값이 0이 된다. (zeros) 학습을 하고자 하는 공식 (가설은) $y = W * x_data + b$ 이 된다.

다음으로 코스트 함수와 옵티마이저를 지정하는데, 코스트 함수는 앞 글에서 설명한것과 같이 가설에 의해 계산된 값 y에서 측정값 y_data를 뺀후에, 이를 제곱하여 평균한 값이다. 코드로 옮기면 다음과 같다.

```
loss = tf.reduce_mean(tf.square(y-y_data))
```

코스트 함수에서 최소 값을 구하기 위해서 옵티마이저로 경사하강법 (Gradient descent) 알고리즘을 사용하기 때문에, 옵티마이저로 tf.train.GradientDescentOptimizer(0.0015) 과 같이 지정하였다.

인자로 들어가는 0.0015는 경사 하강법에서 학습 단계별로 움직이는 학습 속도를 정의하는 것으로 러닝 레이트 (Learning rate라고 한다)) 이 내용은 뒤에서 다시 자세하게 설명하겠다. 코스트 함수와 옵티마이저(Gradient descent)가 정의되었으면 트레이닝 모델에 적용한다.

```
train = optimizer.minimize(loss)
```

는 경사 하강법(Gradient descent) 을 이용하여 코스트 함수 (loss)가 최소가 되는 값을 찾으라는 이야기이다.

다음 코드에서는 for loop로 학습을 10번을 반복해가면서 학습을 하라는 이야기로, for step in xrange(10):

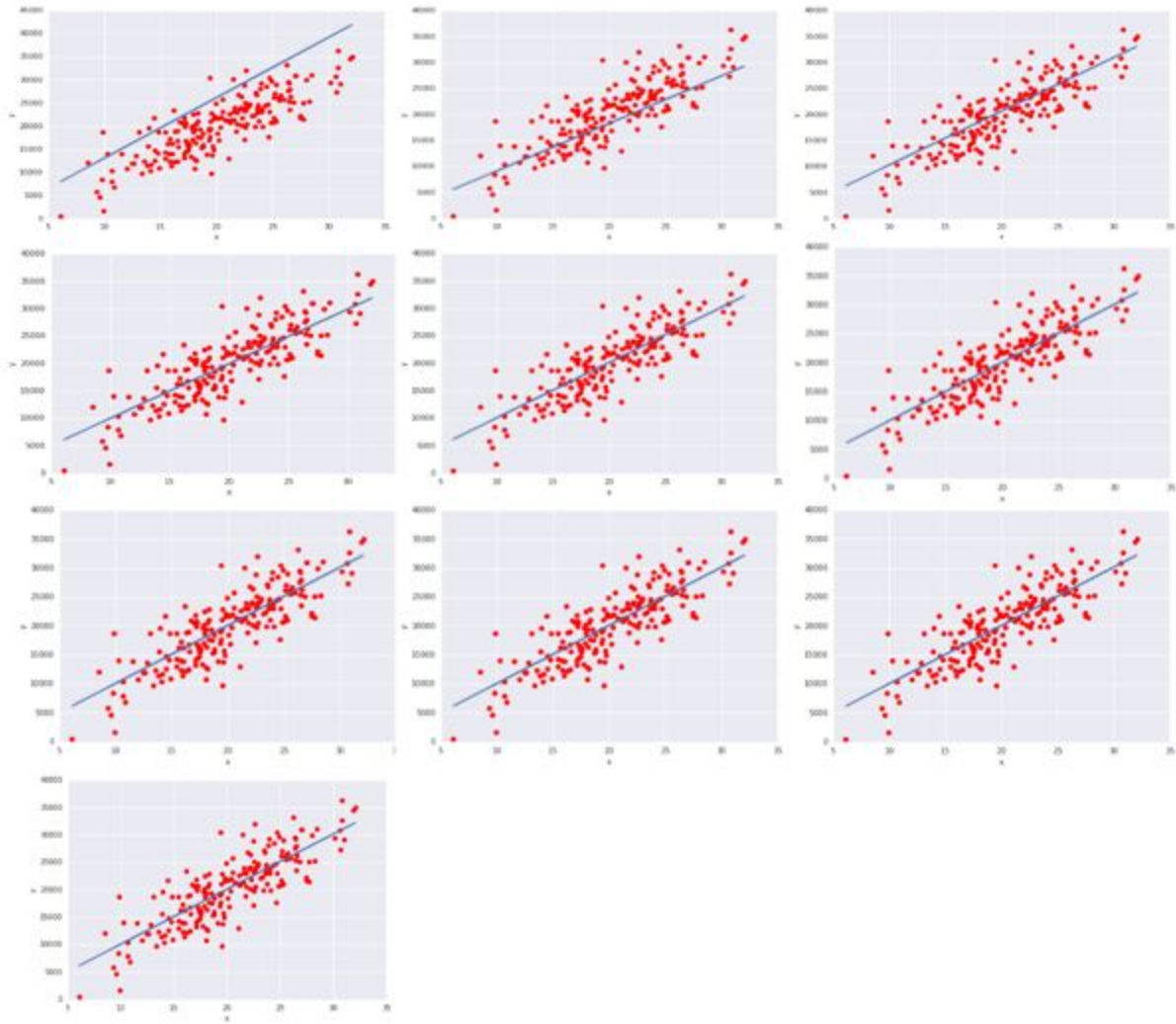
```
    sess.run(train)
    print(step,sess.run(W),sess.run(b))
    print(step,sess.run(loss))
```

학습 단계별로, W,b값 그리고 loss의 값을 화면으로 출력하도록 하였다. 그리고 학습이 어떻게 되는지 그래프로 표현하기 위해서

```
plt.plot(x_data,sess.run(W)*x_data + sess.run(b))
```

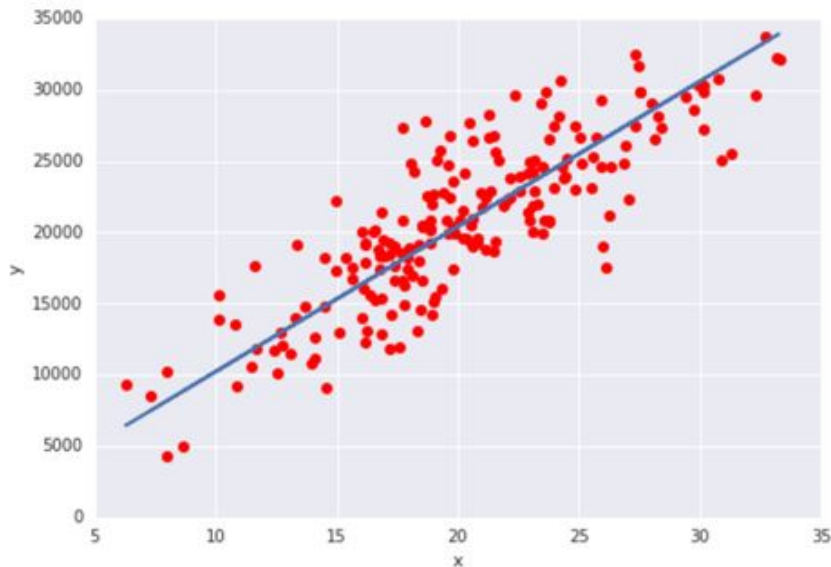
X_data를 가로축으로 하고, $W*x_data + b$ 의 값을 그래프로 출력하도록 하였다.

이렇게 해서 학습을 진행하면 다음과 같은 그래프가 순차적으로 출력되는 것을 확인할 수 있다.



그래프가 점점 데이터의 중앙에 수렴하면서 조정되는 것을 확인할 수 있다.
이렇게 해서 맨 마지막에 다음과 같은 결과가 출력된다.

```
(9, array([ 1018.00439453], dtype=float32), array([ 51.36595535], dtype=float32))  
(9, 10272684.0)
```



W는 1018, b는 51 그리고 코스트의 값은 10272684.0이 됨을 확인할 수 있다.
이렇게 학습이 끝났고, 이제 거리에 따른 택시비는
(택시비) = 1018 * (거리) + 51로
이 공식을 가지고 거리에 따른 택시비를 예측할 수 있다.

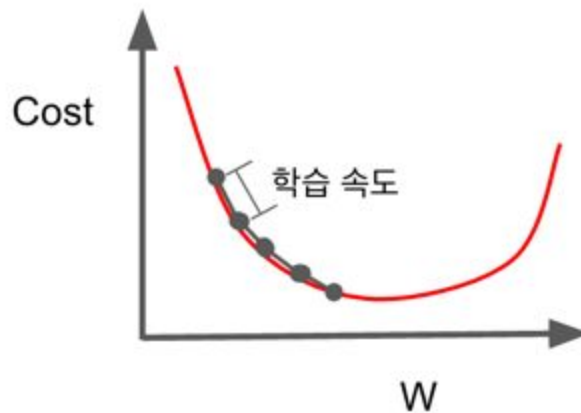
테스트에 사용한 모든 데이터는 링크를 참고하면 얻을 수 있다.

<https://github.com/bwcho75/tensorflowML/blob/master/1.%20Linear%20Regression.ipynb>

학습 속도(러닝 레이트 / Learning Rate) 조정하기

앞의 예제에서 optimizer를 `tf.train.GradientDescentOptimizer(0.0015)`
에서 0.0015로 학습 속도를 지정하였다. 그렇다면 학습 속도란 무엇인가?

선형 회귀 분석의 알고리즘을 되 짚어보면,
가설에 의한 값과 원래값의 차이를 최소화 하는 값을 구하는 것이 이 알고리즘의 내용이고,
이를 코스트 함수들의 최소값을 구하는 것을 통해서 해결한다.
W의 값을 조정해 가면서 코스트의 값이 최소가 되는 값을 찾는데, 이때 경사 하강법 (Gradient descent) 방법을 사용하고 경사의 방향에 따라서 W의 값을 조정하는데, 다음 W의 값이 되는
부분으로 이동하는 폭이 학습 속도 즉 러닝 레이트이다. (아래 그림)



이 예제에서는 학습 속도를 0.0015로 설정하고,
 매번 학습 마다 W 를 경사 방향으로 0.0015씩 움직이도록 하였다.
 그러면 적정 학습 속도를 어떻게 구할까?

오퍼 슈팅 (Over shooting)

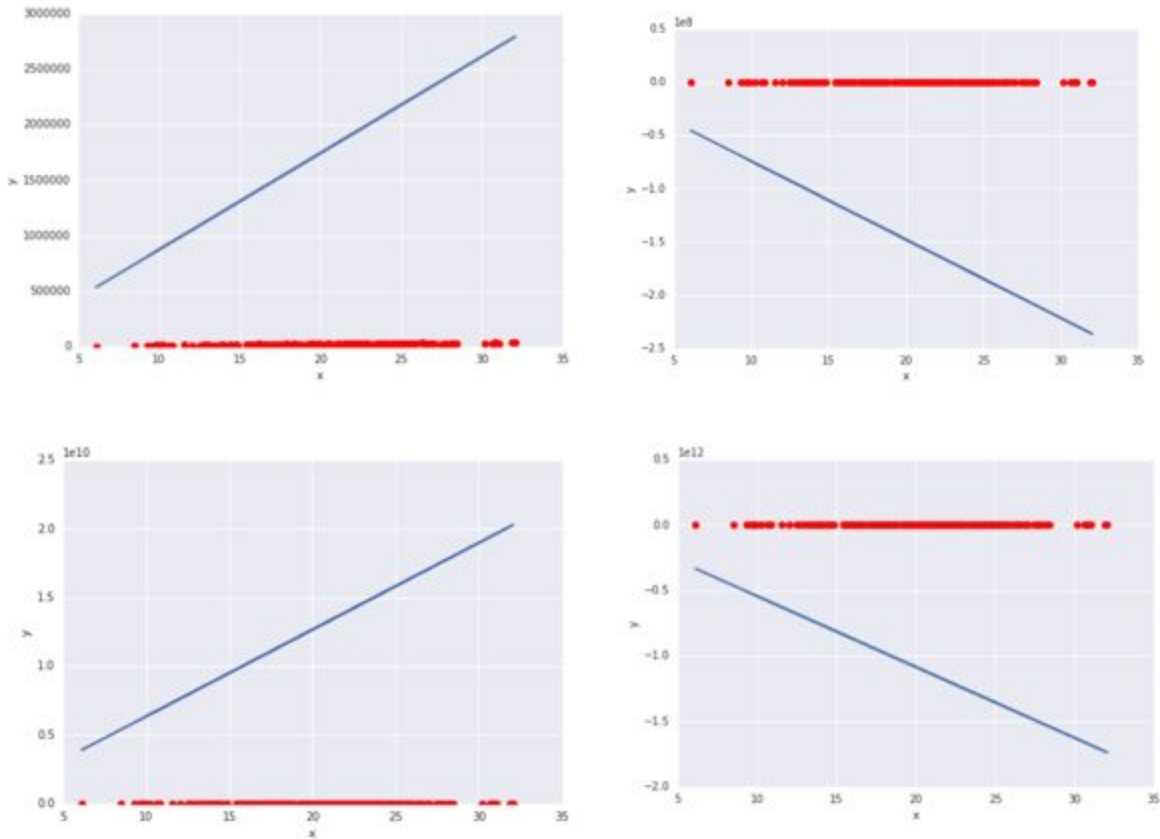
먼저 학습 속도가 크면 어떤일이 벌어지는지를 보자
 학습 속도를 0.1로 주고 학습을 시키면 어떤 결과가 생길까?
 W, b 그리고 cost 함수를 찍어보면 다음과 같은 결과가 나온다.

```
(0, array([ 86515.3671875], dtype=float32), array([ 4038.51806641], dtype=float32))
(0, 3.1747764e+12) ← cost
(1, array([-7322238.], dtype=float32), array([-341854.6875], dtype=float32))
(1, 2.3281766e+16)
(2, array([ 6.27127488e+08], dtype=float32), array([ 29278710.], dtype=float32))
(2, 1.7073398e+20)
(3, array([-5.37040691e+10], dtype=float32), array([-2.50728218e+09], dtype=float32))
(3, 1.252057e+24)
(4, array([ 4.59895629e+12], dtype=float32), array([ 2.14711517e+11], dtype=float32))
(4, 9.1818105e+27)
(5, array([-3.93832261e+14], dtype=float32), array([-1.83868557e+13], dtype=float32))
(5, 6.7333667e+31)
(6, array([ 3.37258807e+16], dtype=float32), array([ 1.57456078e+15], dtype=float32))
(6, 4.9378326e+35)
(7, array([-2.88812128e+18], dtype=float32), array([-1.34837741e+17], dtype=float32))
(7, inf)
(8, array([ 2.47324691e+20], dtype=float32), array([ 1.15468523e+19], dtype=float32))
(8, inf)
(9, array([-2.11796860e+22], dtype=float32), array([-9.88816316e+20], dtype=float32))
(9, inf)
```

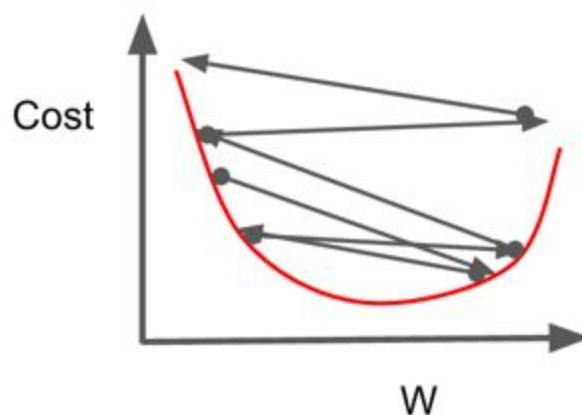
Cost 값이 3.1e+12, 2.3e+16, 1.7e+20

... 오히려 커지다가 7, 8, 9에서는 inf(무한대)로 가버리는 것을 볼 수 있다.

그래프를 보면 다음과 같은 형태의 그래프가 나온다.



학습이 진행될 수 록, 코스트 함수의 결과 값이 작아지면서 수렴이 되어야 하는데, 그래프의 각이 서로 반대로 왔다갔다 하면서 발산을 하는 모습을 볼 수 있다. 코스트 함수의 그래프를 보고 생각해 보면 그 원인을 알 수 있다.



학습 속도의 값이 크다 보니, 값이 아래 골짜기로 수렴하지 않고 오히려 반대편으로 넘어가면서 점점 오히려 그래프 바깥 방향으로 발산하면서, W값이 발산을 해서 결국은 무한대로 간다. 이를 오버 슈팅 문제라고 한다.

그래서, 학습 과정에서 코스트 값이 수렴하지 않고 점점 커지면서 inf(무한대)로 발산하게 되면, 학습 속도가 지나치게 큰것으로 판단할 수 있다.

스몰 러닝 레이트(Small Learning Rate)

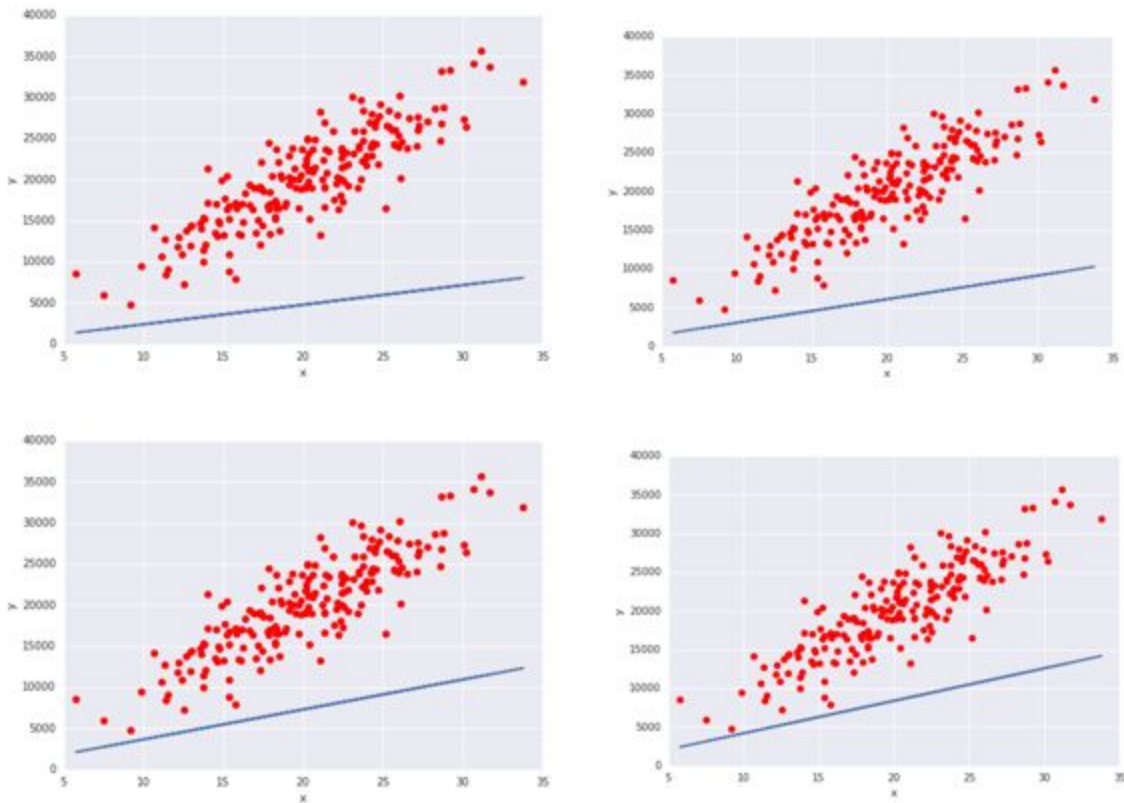
반대로 학습 속도가 매우 작을때는 어떤일이 발생할까?

학습속도를 0.0001로 작게 설정을 해보자.

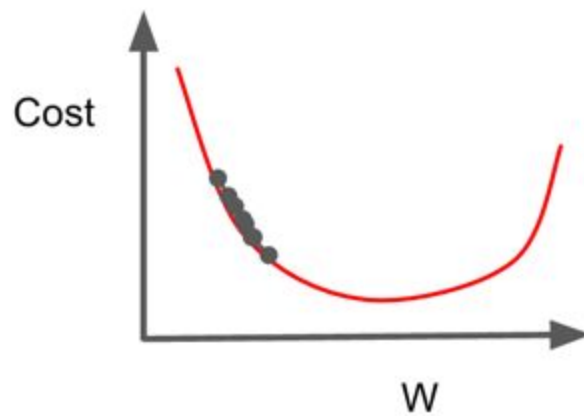
```
(0, array([ 86.40672302], dtype=float32), array([ 4.03895712], dtype=float32))
(0, 3.6995174e+08)
(1, array([ 165.43540955], dtype=float32), array([ 7.72794485], dtype=float32))
(1, 3.1007162e+08)
(2, array([ 237.61743164], dtype=float32), array([ 11.09728241], dtype=float32))
(2, 2.6011749e+08)
(3, array([ 303.54595947], dtype=float32), array([ 14.17466259], dtype=float32))
(3, 2.18444e+08)
(4, array([ 363.76275635], dtype=float32), array([ 16.98538017], dtype=float32))
(4, 1.8367851e+08)
(5, array([ 418.76269531], dtype=float32), array([ 19.55253601], dtype=float32))
(5, 1.5467589e+08)
(6, array([ 468.99768066], dtype=float32), array([ 21.89723206], dtype=float32))
(6, 1.304809e+08)
(7, array([ 514.8805542], dtype=float32), array([ 24.03874016], dtype=float32))
(7, 1.1029658e+08)
(8, array([ 556.78839111], dtype=float32), array([ 25.99466515], dtype=float32))
(8, 93458072.0)
(9, array([ 595.06555176], dtype=float32), array([ 27.78108406], dtype=float32))
(9, 79410816.0)
```

코스트값이 점점 작은 값으로 작아지는 것을 볼 수 있지만 계속 감소할 뿐 어떤 값에서 정체 되거나 수렴이 되는 형태가 아니다.

그래프로 표현해보면 아래 그래프와 같이 점점 입력 데이터에 그래프가 가까워 지는 것을 볼 수 있지만, 입력 데이터에 그래프가 겹쳐지기 전에 학습이 중지 됨을 알 수 있다.



이런 문제는 학습속도가 너무 작을 경우 아래 그림 처럼, 코스트 값의 최소 값에 도달하기전에, 학습이 끝나버리는 문제로 Small learning rate 라고 한다.



이 경우에는 학습 횟수를 늘리거나 또는 학습 속도를 조절함으로써 해결이 가능하다.

3. 로지스틱 회귀를 이용한 이항 분류 모델

1장에서 머신러닝의 종류는 결과값의 타입이 연속형인 Regression (회귀) 문제와, 몇가지 정해진 분류로 결과(이산형)가 나오는 Classification(분류) 문제가 있다고 하였다. 2,3장에 걸쳐서 회귀 문제에 대해서 알아보았고, 이번장에서는 로지스틱 회귀를 이용한 분류 문제에 대해서 알아보자. 이 글의 내용은 Sung.Kim 교수님의 “모두를 위한 딥러닝”(<http://hunkim.github.io/ml/>) 을 참고하였다. 여러 자료들을 찾아봤는데, 이 강의 처럼 쉽게 설명해놓은 강의는 없는것 같다.

분류 문제(Classification)의 정의

분류 문제란 학습된 모델을 가지고, 입력된 값을 미리 정해진 결과로 분류해주는 모델을 이야기 한다.

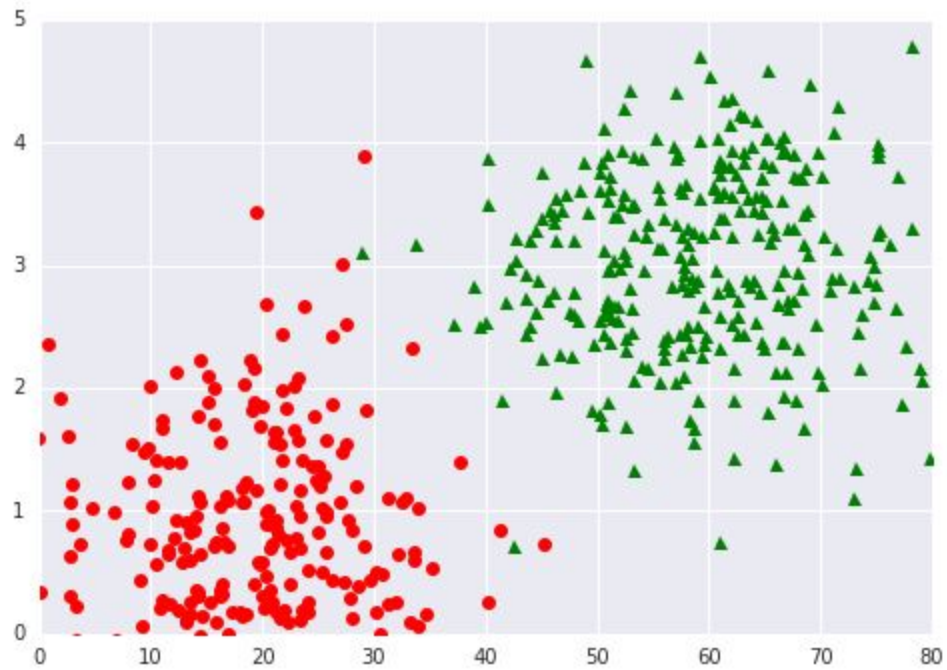
분류 결과가 참/거짓과 같이 두개만 있을때 이항 분류 분석, 두개 이상일때는 다항 분류 분석이라고 하는데, 이번장에서 살펴볼 로지스틱 회귀 분석은 분류된 결과가 두 가지만 있는 이항 분류 모델이다. (다항 분류 모델은 로지스틱 회귀에 이어서 소프트맥스 회귀 분석에서 설명하도록 하겠다.)

이항 분류의 대표적인 예는 다음과 같다.

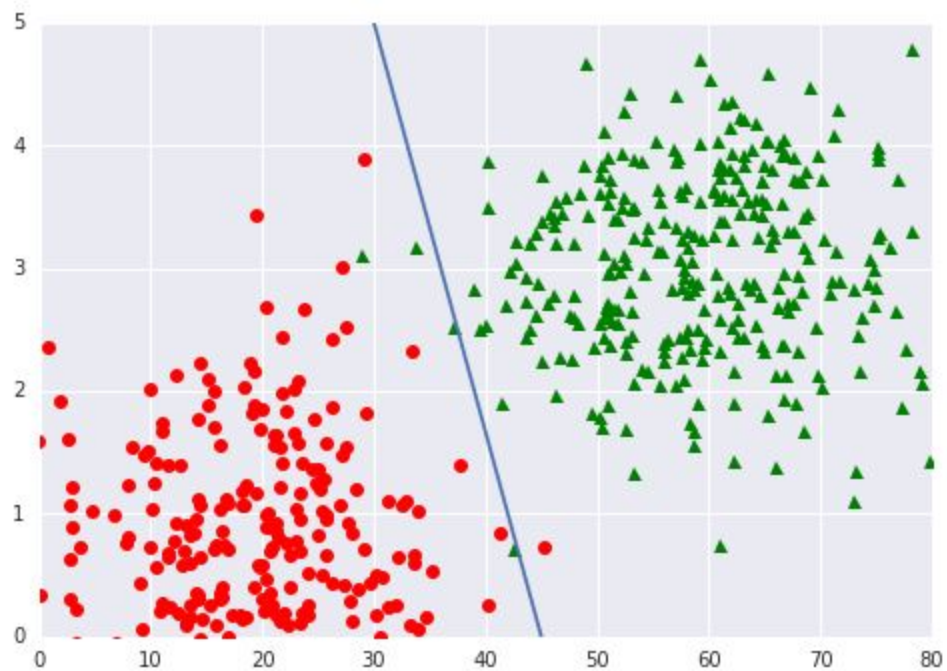
- 이메일 스팸과 정상 이메일 검출
- 신용카드 거래에서 정상 거래와 이상 거래 검출
- 게임에서 어뷰징 사용자와 정상 사용자 검출

등이 이항 분류의 예가 될 수 있다.

예를 들어 아래와 같은 데이터가 있다고 가정하자



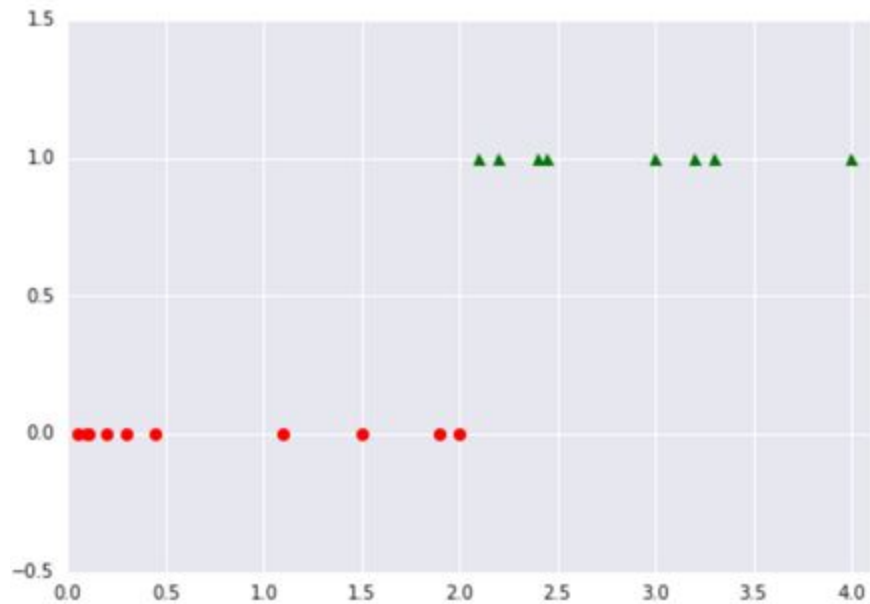
붉은 동그라미로 표시된 데이터와, 녹색 세모로 표시된 데이터를 분류하고 싶을때, 아래와 같이 이항 분류 문제는 이를 분류할 수 있는 이상적인 직선 그래프를 찾는 것이다.



로지스틱스 회귀 분석 (Logistics Regression)

선형 회귀 분석 (Linear regression) 으로 분류 문제 접근하기

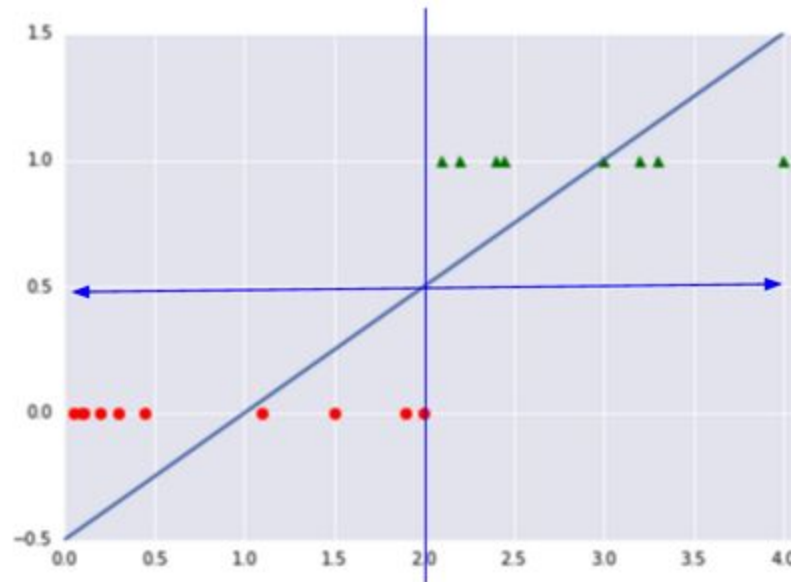
이항 분류 모델에 대한 예를 들어보자. 종양의 크기에 따라서 양성 종양(암)인지 음성 종양인지를 판별하는 문제가 있다고 하자. 아래 그림은 종양의 크기에 대한 양성 and 음성 여부를 그래프로 나타낸 것인데, X축은 종양의 크기, Y축은 종양의 양성 and 음성 여부를 나타낸다. 1이면 양성 0이면 음성이다.



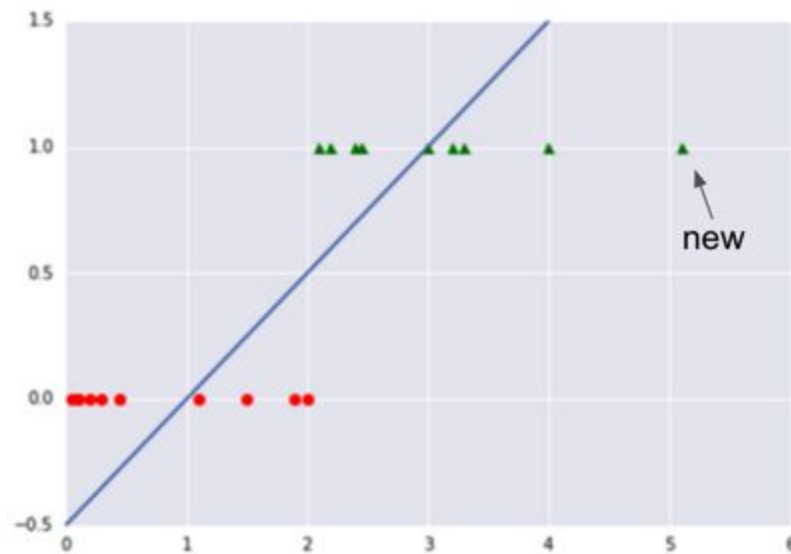
이 문제를 선형 회귀 모델로 정의해서 그래프를 그려보면 다음과 같다.

$$y = W * x$$

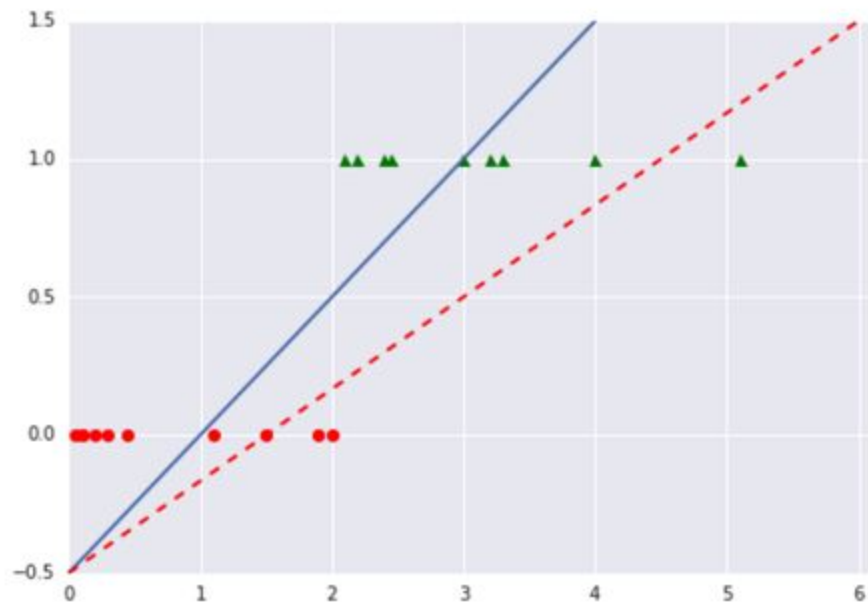
와 같은 그래프가 그려지고 대략 아래 그림에서 보는 것과 같이 $y > 0.5$ 보다 크면 양성 암, $y < 0.5$ 보다 작으면 음성암으로 판단할 수 있다.



그런데, 만약에 새로운 트레이닝 데이터에서, 종양의 크기가 큰 데이터가 들어오면 어떻게 될까?
아래 그림을 보자, 예를 들어 새로운 트레이닝 데이터에 종양의 크기가 5인 경우 양성 암이라는
데이터가 새로 들어왔다고 하자



이 경우 앞에서 선형 회귀로 만든 그래프의 기울기가 새로 들어온 데이터를 포함하면 맞지 않기
때문에 선형 회귀로 재학습을 시키게 되면 다음과 같은 기울기(점선 그래프)로 변하게 된다.



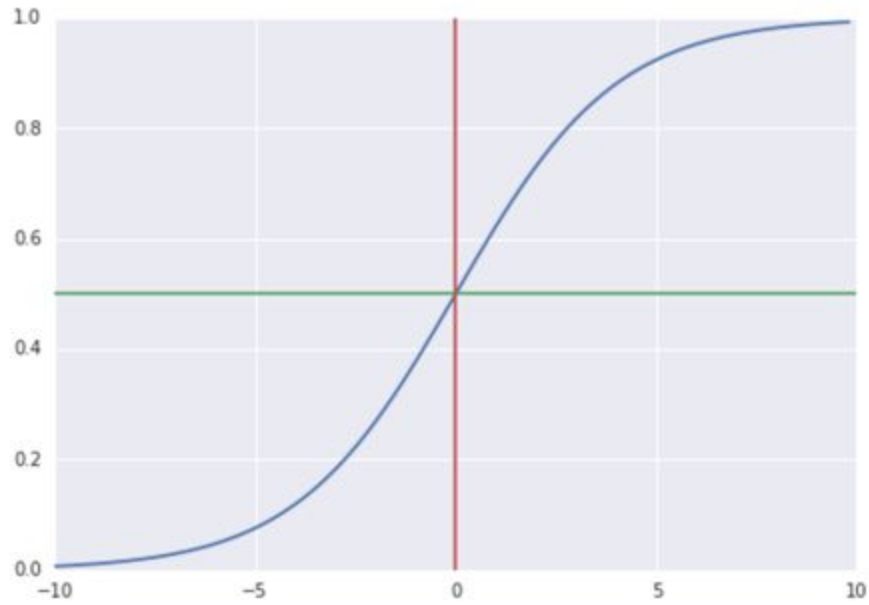
이 경우에는 앞에서 암이 양성인 여부를 판단할때 사용한 y 가 0.5라는 기준은 더이상 사용할 수 없게 되고, y 가 0.2 일때를, 새 기준으로 잡아서, 암의 양성/음성 여부를 판단해야 한다. 그러면 새로운 데이터가 들어올때 마다 기준점을 다시 잡아야 하는것인가? 또한 그렇게 만든 모델로 예측을 한다면, 학습에 사용되지 않은 큰 데이터가 들어온다면 오류가 발생할 수 도 있다. 그래서, 선형 회귀 분석 모델(Linear regression) 은 이항 분류에 적절하지 않다. 그렇다면 어떤 모델이 적절할까 ?

참고

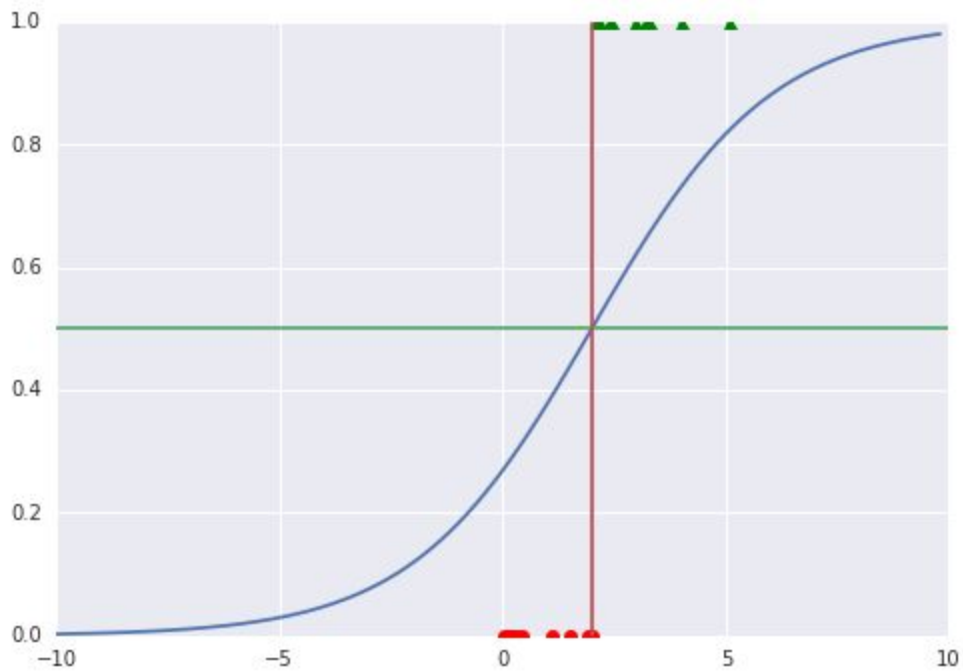
- <https://www.coursera.org/learn/machine-learning/lecture/wlPeP/classification>
- <http://stats.stackexchange.com/questions/22381/why-not-approach-classification-through-regression>

시그모이드(sigmoid) 함수

이런 형태의 이항 분류 분석에 적절한 함수로 시그모이드(sigmoid)함수라는 것이 있다. 그래프의 모양은 다음과 같다. S 자 형태의 그래프 모양으로 중심축 ($x=0$)을 중심으로 좌측은 0으로 수렴하고 우측은 1로 수렴한다.



이 시그모이드 함수에 앞의 데이터를 적용해보면 다음과 같은 형태가 된다.



그림과 같이 y축을 0.5를 기준으로 판단할때 y가 0.5 일때 x가 2 인데, $x < 2$ 인 부분은 $y=0$ 으로 음성, $x > 2$ 인 부분은 $y=1$ 로 양성이 된다.

큰 데이터 ($x=100$)가 추가된다하더라도 시그모이드 함수는 그 값이 1로 수렴되기 때문에, 앞의 선형 회귀 분석의 경우 처럼 암의 양성/음성인 경우를 결정하는 y와 x값이 변화하지 않는다.

가설 (Hypothesis)

그래서, 이 시그모이드 함수를 사용하여 가설을 정의할 수 있다.

가설은 아래와 같다

$$y = \text{sigmoid}(Wx + b)$$

결과 값 y 는 0 또는 1의 값을 갖는다. 시그모이드 함수를 수학 공식으로 표현하면, 아래와 같다.
(그렇다는 것만 알아두고 외워서 쓰자)

$$1 / (1 + \text{math.exp}(-(W*x+b))$$

디시전 바운드리(Decision boundary)

$y = \text{sigmoid}(W*x + b)$ 을 가설 함수로 그려진 위의 그래프에서 W 는 1, b 는 -2로 그래프로 우측으로 두칸을 이동하였다.

W 에 1, b 에 -2 를 대입해보면 $y = \text{sigmoid}(1*x-2)$ 의 형태의 그래프인데, “ x 가 2를 기준으로 좌,우측이 양성암이냐 아니냐” 를 결정하기 때문에 때문에,

- $1*x-2 < 0$ 이면, $y=0$ 이 되고
- $1*x-2 > 0$ 이면, $y=1$ 이 된다.

이를 일반화 해보면, 시그모이드(sigmoid) 함수 내에 들어가는

- $W*x+b < 0$ 이면, $y=0$ 이 되고,
- $W*x+b > 0$ 이면, $y=1$ 이 된다.

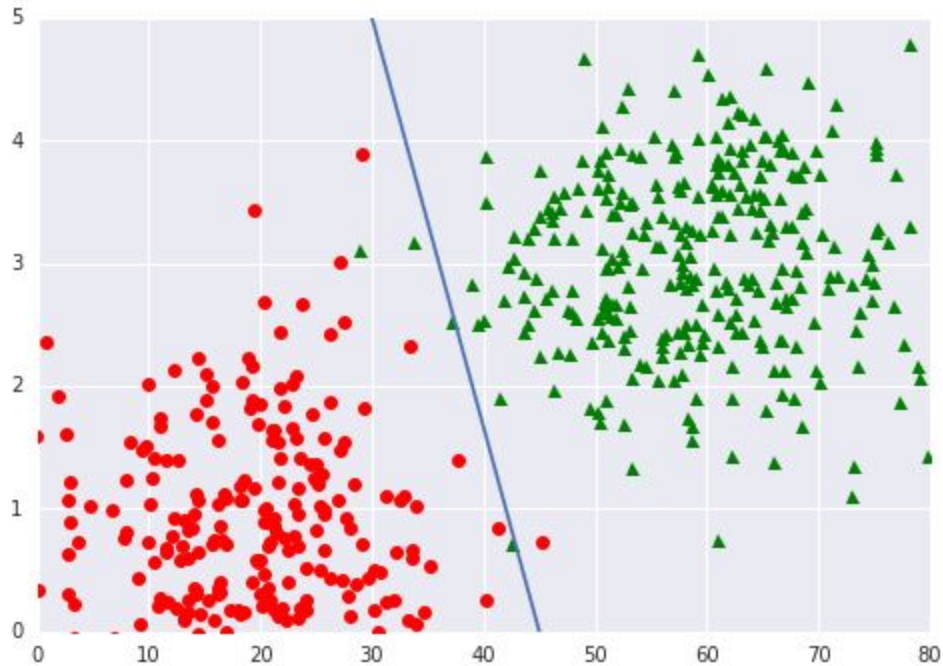
즉 로지스틱 회귀 분석은 위의 조건을 만족하는 W 와 b 의 값을 찾는 문제이다.

그리고 시그모이드 함수내의

$$z=W*x+b$$

그래프를 기준으로 나눠서 y 가 0또는 1이 되는 기준을 삼는데, 이 그래프를 기준으로 결정을 하기 때문에, 이를 디시전 바운드리 (Decision boundary) 라고 한다.

변수가 x 하나가 아니라, x_1, x_2 가 있는 문제를 살펴보자



이 문제에서 가설 함수 $y = \text{sigmoid}(W_1 \cdot x_1 + W_2 \cdot x_2 + b)$ 가 될것이고,
 $z = W_1 \cdot x_1 + W_2 \cdot x_2 + b$ 가 디시전 바운드리 함수가 되며, 위의 그래프상에서는 붉은선과 초록선을 나누는 직선이 되고 이것이 바로 디시전 바운드리가 된다.

코스트 함수 (비용함수/Cost function)

자 그러면 가설 함수를 정의 했으니 걱정 W 와 b 값을 찾기 위해서 코스트 함수를 정의해보자.
 다시 한번 앞에서 코스트 함수의 개념을 되짚어 보면, 코스트 함수의 개념은 가설 함수에 의해서 예측된 값과 트레이닝을 위해서 입력된 값(실제값) 사이의 차이를 계산해주는 함수로, 예측된 값과 입력된 값 들의 차이에 대한 평균 값을 구한다.
 로지스틱 회귀에서 사용되는 코스트 함수는 다음과 같다.

$$\text{cost_function} = (1/n) * \text{Sum}(-y_{\text{origin}} * \log(\text{sigmoid}(Wx+b)) - (1-y_{\text{origin}}) * \log(1-(\text{sigmoid}(Wx+b))))$$

- n 의 트레이닝 데이터의 수
- Y_{origin} 는 트레이닝에 사용된 x 에 대한 입력값

의미를 설명하겠지만, 머리가 아프면 넘어가도 좋다. 그냥 가져다 쓰면 된다.

그러면 어떻게 저런 코스트 함수가 사용되었는지를 알아보자.

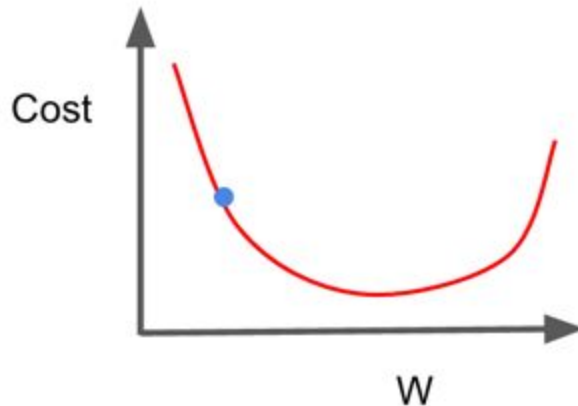
선형회귀분석(Linear regression)의 코스트 함수를 다시 한번 살펴보자

코스트 함수는 측정값과 가설에 의해서 예측된 값의 차이의 제곱 평균을 나타내는 함수였다.

선형 회귀 분석에서의 코스트 함수는 다음과 같았다.

$$\text{Cost} = \text{Sum}((y_data_n - y_origin_n)^2) / n$$

그리고 이 함수를 그래프로 그려보면 다음과 같이 매끈한 그래프가 나왔다.



그래프의 모양이 매끈한 골짜기 모양이었기 때문에 경사 하강법(Gradient descent)을 사용할 수 있었다.

그러면 로지스틱 회귀 분석에도 기존의 코스트 함수를 이용하여 경사하강법을 적용할 수 있는지 보자

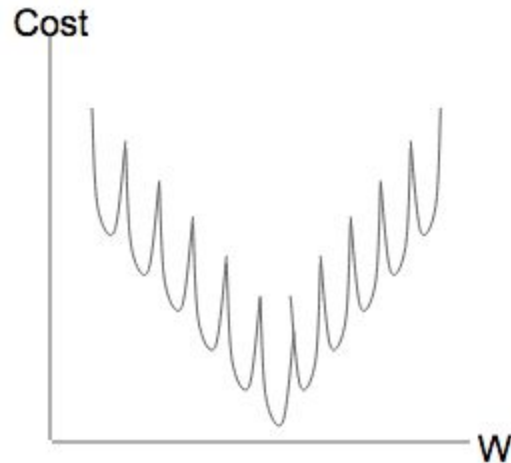
코스트 함수에서 y_data_n 에 가설 함수를 대입 시켜 보면

$$\text{Cost} = \text{Sum}((\text{sigmoid}(Wx + b) - y_origin)^2) / n$$

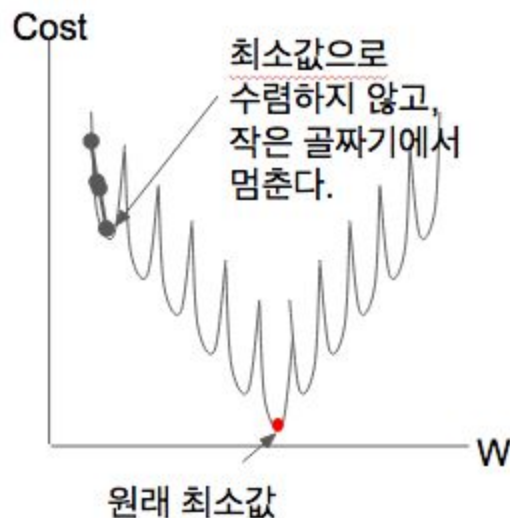
Sigmoid 함수를 풀어서표현하면

$$\text{Cost} = \text{Sum}(1 / (1 + \text{math.exp}(-(W*x+b))) - y_origin)^2) / n$$

가 되는데, $\text{exp}(-(W*x+b))$ (즉 $e^{-(W*x+b)}$) 형태로 표현되기 때문에, e 가 들어간 코스트 함수의 그래프는 다음과 같은 형태를 띄게 된다.



경사 하강법은 그래프를 타고 내려가면서 가장 작은 값을 찾는 알고리즘인데, (물이 골짜기를 따라서 내려가듯이), 이 코스트 함수의 그래프는 작은 골짜기들이 모여서 큰 골짜기 형태를 만든 모양이 된다. 그래서 경사 하강법을 적용할 경우 아래 그림과 같이 코스트 함수의 최소값으로 수렴하지 않고, 중간에 작은 골짜기에서 수렴해 버린다.



그래서 로지스틱 회귀 분석에서는 경사 하강법을 사용하기 위해서 이 코스트 함수를 메끈한 형태로 만들 필요가 있고, 새로운 코스트 함수를 사용한다. “e” 때문에 이런형태의 그래프가 그려지는 건데, e를 상쇄할 수 있는 역치함수는 log 함수가 있다. 그래서 log 함수를 적용하여 메끈한 형태의 코스트 함수를 정의해보자.

코스트 함수는 $y=1$ 일때와 $y=0$ 일때 나눠서 계산해야 한다. 각각의 함수를 보면 다음과 같다.

- $y=1$
 $\text{cost} = (1/n) * \text{sum}(-\log(\text{가설}))$
 $\text{cost} = (1/n) * \text{sum}(-\log(\text{sigmoid}(Wx+b)))$
- $y=0$
 $\text{cost} = (1/n) * \text{sum}(-\log(1-\text{가설}))$

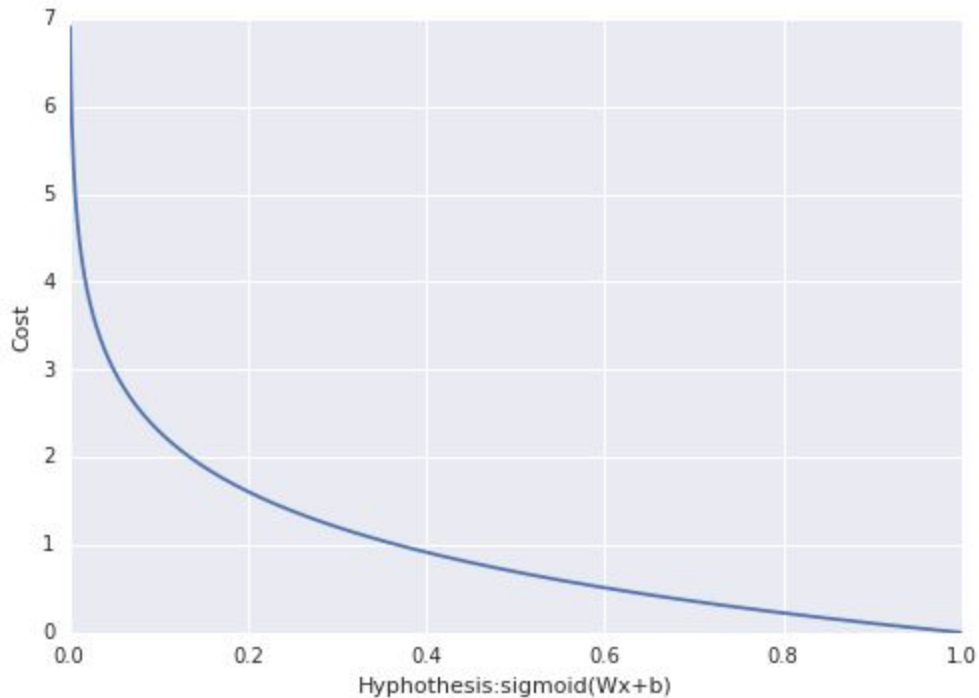
$$\text{cost} = (1/n) * \text{sum}(-\log(1-\text{sigmoid}(Wx+b)))$$

코스트 함수는 측정한 값과, 가설에 의해 예측된 값의 차이를 나타내는 함수로, 개별값이 작을수록 적절한 모델이 된다. 그래서 측정값과 가설값이 같거나 유사할수록 코스트 함수의 결과값이 작게 나와야 한다.

먼저 $y=1$ (측정한 값)일때를 보자,

코스트 함수는 $\text{cost} = (1/n) * \text{sum}(-\log(\text{sigmoid}(Wx+b)))$ 이다.

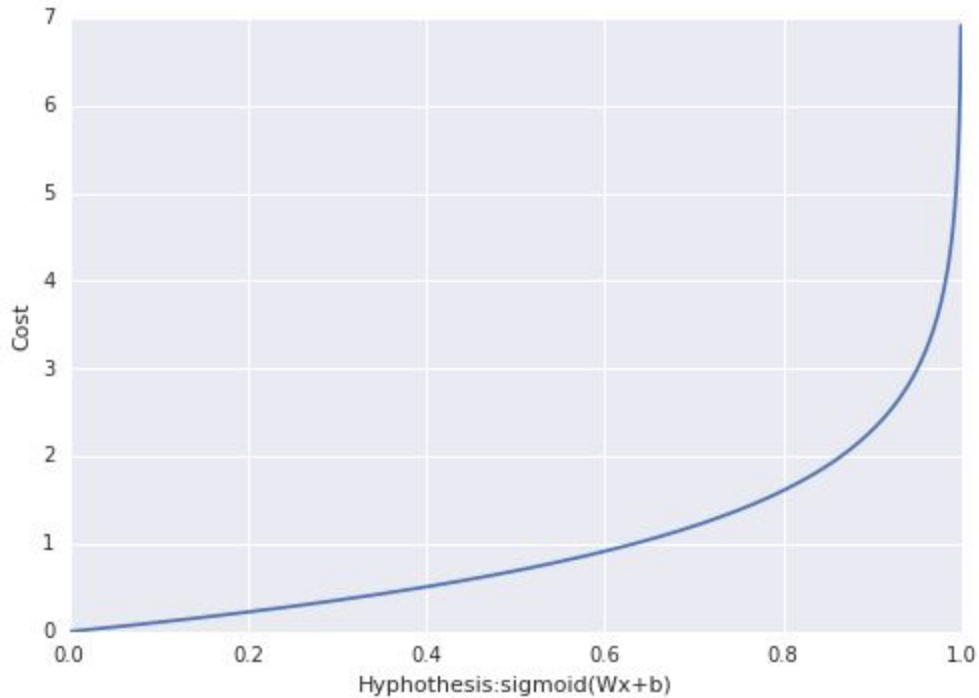
전체 코스트 평균 말고, 개별 값에 대한 측정값과 예측값에 대한 차이는 이 함수에서 $-\log(\text{sigmoid}(Wx+b))$ 이다. 시그모이드 함수 특성상 $\text{sigmoid}(Wx+b)$ 는 0~1까지의 범위를 가지기 때문에 $-\log(0\sim 1)$ 을 그래프로 그려보면 다음과 같다.



측정값이 1이기 때문에, 가설함수 (시그모이드 함수 $\text{sigmoid}(Wx+b)$)에 의한 결과가 1이면 예측이 잘된 것이고, 1에서 멀어져서 0으로 갈수록 예측된 값과 측정된 값의 차이가 크다고 할 수 있는데, 위의 그래프에서 보면, 가설에 의해 계산한 결과(x축)가 1에 가까울수록 코스트(y축)은 0으로 수렴하고, 가설에 의해 계산한 결과가 0에 수렴할수록, 코스트는 높아지는 것을 볼 수 있다. 즉 $y=1$ 에서는 가설이 1에 수렴해야 하기 때문에, 1에 가까워질수록 코스트가 낮아지는 그래프를 띄게 된다.

y 가 0일때도 마찬가지로 원리인데 측정값이 0이기 때문에, 가설에 의한 결과값이 0이 되어야 한다. 0에서 멀어질 경우 코스트가 늘어나고 0에서 가까워질 경우 코스트가 줄어드는 형태의 비용 함수를 정의해야 한다.

마찬가지로 코스트 함수의 평균이 아닌 개별값을 보면, $-\log(1-\text{sigmoid}(Wx+b))$ 이 되고 $\text{sigmoid}(Wx+b)$ 는 0..1의 범위이기 때문에, $-\log(1-(0..1))$ 이 된다. 단 1에서 0..1을 뺄셈을 했기 때문에, 그래프는 $-\log(1.0, 0.99, 0.98, \dots)$ 형태로 $y=1$ 인 경우와 반대 모양의 그래프가 된다.



자 이제, $y=1$ 인 경우와 $y=0$ 인 경우에 대한 각각의 코스트 함수를 정의하였다. 이를 코딩으로 옮기려면 $y=1$ 인 경우와 $y=0$ 인 경우에 대해서 각각 다른 코스트 함수로 처리하기 위해서 if 문을 사용해야 하지만, 그러면 코딩이 복잡해지기 때문에 이를 하나의 식으로 간단히 할 수 있는 방법을 찾아보자

- $y=1$
 $\text{cost_y1} = (1/n) * \sum (-\log(\text{sigmoid}(Wx+b)))$
- $y=0$
 $\text{cost_y0} = (1/n) * \sum (-\log(1-\text{sigmoid}(Wx+b)))$

$y=1$ 일때는 $y=0$ 인 코스트 함수 cost_y0 가 0이 되고, $y=0$ 일때는 $y=1$ 의 코스트 함수 cost_y1 가 0이 되면 된다.

즉 $\text{cost} = y * \text{cost_y1} + (1-y) * \text{cost_y0}$ 형태가 되면,

- $y=1$ 이면, $\text{cost} = 1 * \text{cost_y1} + (1-1) * \text{cost_y0} = \text{cost_y1}$ 이 되고
- $y=0$ 이면, $\text{cost} = 0 * \text{cost_y1} + (1-0) * \text{cost_y0} = \text{cost_y0}$ 이 된다.

그래서 $y=1$ 인 코스트 함수와 $y=0$ 인 코스트 함수를 위의 식 cost_y1 과 cost_y0 에 각각 대입해보면

$$\text{cost} = y * [(1/n) * \sum (-\log(\text{sigmoid}(Wx+b)))] + (1-y) * [(1/n) * \sum (-\log(1-\text{sigmoid}(Wx+b)))]$$

으로 평균 함수인 $(1/n) * \sum$ 을 앞으로 빼면

$$\text{cost} = (1/n) * \sum ($$

```

y*(-log(sigmoid(Wx+b)))
+ (1-y)(-log(1-sigmoid(Wx+b) ))
)

```

가 된다.

수식을 따라가면서 이해하면 제일 좋겠지만, 코스트 함수가 저런 원리로 생성이 되는구나 정도 이해하고, 그냥 가져도 쓰자. (왠지 주입식 교육 같은 느낌이 들기는 하지만)

옵티마이저 (Optimizer)

코스트 함수가 정해졌으면, 코스트를 최소화할 옵티마이저로 어떤 옵티마이저를 사용해야 할지 결정해야 한다.

앞에서 경사 하강법에 적절하도록 코스트 함수를 수정했기 때문에, 경사 하강법(Gradient descent) 알고리즘을 사용한다. 이 경사하강법은 텐서플로우 코딩에서는 간단하게

```

optimizer = tf.train.GradientDescentOptimizer(learningRate)
train = optimizer.minimize(cost_function)

```

정의하여 옵티마이저로 사용할 수 있다.

예측 (Prediction)

자 이렇게 해서, 로지스틱 회귀에 대한 학습을 끝내고, W와 b 값을 구했다고 하자. 그렇다면 이 학습된 모델을 가지고, 들어오는 데이터에 대한 분류는 어떻게 할것인가? (예측은 어떻게 할것인가)

가설 함수를 다시 생각해보면 가설함수는 $\text{sigmoid}(Wx + b)$ 이다.

그래서 예측이 필요한 값 x가 들어왔을때, 가설함수에 의한 결과값이 y' 이라고 하면, y'은 0..1 까지의 실수가 되고, y'은 0.75일 경우, x에 대한 결과가 1일 확률은 75%, 0일 확률은 (100-75%인) 25%가 된다.

앞에서 종양의 크기에 대한 양성암 문제를 다시 예를 들어보면, 종양의 크기 $x=5$ 이면, $\text{sigmoid}(Wx+b)$ 에 의해서 계산된 결과가 0.95라고 하면, 이 경우 양성 종양인 확률은 95%, 음성인 확률은 5% 이다.

정리

분류 문제는 회귀 분석과 함께 머신러닝에서 가장 대표적인 문제이고, 로지스틱 회귀 분석은 분류문제에 있어서 기본적으로 대표가 되는 모델이다. 수식이 많이 나와서 다소 복잡할 수 있겠지만 정리를 해보면 다음과 같다.

먼저 가설 함수 ($Wx+b$) 를 정의하고, 이 가설 함수에 사용된 변수 W 와 b 에 대한 적정값을 계산하기 위해서 코스트 함수를 정의한후, 이 코스트 함수를 이용한 W 와 b 를 구하기 위해서 옵티마이저를 정의해서 학습을 통해서 W 와 b 값을 구한다.

이 과정에서 복잡한 수학적인 설명이 있었지만, 다음과 같은 접근 방법이 좋지 않을까 한다.

1. 로지스틱 회귀 분석은 결과가 참/거짓인 이항 분석 문제에 사용된다.
2. 비용 함수가 있다는 것을 알고 로지스틱 회귀 분석용 비용 함수를 가져다 쓴다. 단 비용 함수가 낮을 수록 학습이 정확하다는 의미 정도는 알아야 학습 도중에 비용 함수의 결과를 보고 학습의 정확도를 파악할 수 있다.
3. 옵티마이저는 그라디언트 디센트 알고리즘을 사용한다. 그냥 가져다 쓴다. 단 여기서 학습 속도(Learning Rate, 2장 선형 회귀 분석을 구현하는 문서에서도 설명하였음)의 의미를 파악하고, 학습 프로그램을 돌릴때 이 패러미터를 조정하면서 사용한다.
4. 나온 결과값 (W 와 b) 값을 이용하여 예측을 수행한다.

참고

- Sung Kim 교수님의 딥러닝 강좌 <https://hunkim.github.io/ml/>
- Andrew ng 교수님 강의 <https://www.youtube.com/watch?v=LLx4dilP83I>
- Andrew ng 교수님의 Decision boundary에 대한 설명
- Andrew ng 교수님 강의 노트
http://www.holehouse.org/mlclass/06_Logistic_Regression.html
- https://www.youtube.com/channel/UCRyIQSBvSybbaNY_JCyg_vA

4. 텐서플로우 기본

딥러닝에 대한 대략적인 개념을 익히고 실제로 코딩을 해보려고 하니, 모 하나를 할때 마다 탁탁 막힌다. 파이썬이니 괜찮겠지 했는데, (사실 파이썬도 다 까먹어서 헛갈린다.) 이걸 라이브러리로 도배가 되어 있다.

당연히 텐서플로우 프레임웍은 이해를 해야 하고, 데이터를 정제하고 시각화 하는데, numpy, pandas와 같은 추가적인 프레임웍에 대한 이해가 필요하다.

node.js 시작했을때도 자바스크립트 때문에 많이 헤매고 몇달이 지난후에야 어느정도 이해하게 되었는데, 역시나 차근차근 기초 부터 살펴봐야 하지 않나 싶다.

텐서 플로우에 대해 공부한 내용들을 하나씩 정리할 예정인데, 이 컨텐츠들은 유튜브의 이찬우님의 강의를 기반으로 정리하였다. 무엇보다 한글이고 개념을 쉽게 풀어서 정리해주시기 때문에, 웬만한 교재 보다 낫다.

<https://www.youtube.com/watch?v=a74pFg8paVc>

텐서플로우 환경 설정

텐서 플로우 환경을 설정 하는 방법은 쉽지 않다. 텐서플로우 뿐 아니라, 여러 파이썬 버전과 그에 맞는 라이브러리도 함께 설정해야 하기 때문에 여간 까다로운게 아닌데, 텐서플로우 환경은 크게 대략 두 가지 환경으로 쉽게 설정이 가능하다.

구글 데이타랩

첫번째 방법은 구글에서 주피터 노트북을 도커로 패키징해놓은 패키지를 이용하는 방법이다. 도커 패키지안에, numpy,pandas,matplotlib,tensorflow,python 등 텐서플로우 개발에 필요한 모든 환경이 패키징 되어 있다.

아나콘다

다음 방법은 일반적으로 가장 많이 사용하는 방법인데, 파이썬 수학과 관련 라이브러리를 패키징해놓은 아나콘다를 이용하는 방법이 있다. 자세한 환경 설정 방법은 https://www.tensorflow.org/versions/r0.12/get_started/os_setup.html#anaconda-installation 를 참고하기 바란다. 아나콘다를 설치해놓고, tensorflow 환경(environment)를 정의한 후에, 주피터 노트북을 설치하면 된다. <http://stackoverflow.com/questions/37061089/trouble-with-tensorflow-in-jupyter-notebook> 참고

Tensorflow 환경을 만든 후에,
\$ source activate tensorflow
를 실행해서 텐서 플로우 환경으로 전환한후, 아래와 같이 ipython 을 설치한후에, 주피터(jupyter) 노트북을 설치하면 된다.

```
(tensorflow) username$ conda install ipython  
(tensorflow) username$ pip install jupyter #(use pip3 for python3)
```

아나콘다 기반의 텐서플로우 환경 설정은 나중에 시간이 될때 다른 글을 통해서 다시 설명하도록 하겠다.

텐서플로우의 자료형

텐서플로우는 뉴럴네트워크에 최적화되어 있는 개발 프레임워크이기 때문에, 그 자료형과, 실행 방식이 약간 일반적인 프로그래밍 방식과 상의하다. 그래서 삽질을 많이 했다.

상수형 (Constant)

상수형은 말 그대로 상수를 저장하는 데이타 형이다.

- `tf.constant(value, dtype=None, shape=None, name='Const', verify_shape=False)`와 같은 형태로 정의 된다. 각 정의되는 내용을 보면
 - `value` : 상수의 값이다.
 - `dtype` : 상수의 데이터형이다. `tf.float32`와 같이 실수,정수등의 데이터 타입을 정의한다.
 - `shape` : 행렬의 차원을 정의한다. `shape=[3,3]`으로 정의해주면, 이 상수는 3x3 행렬을 저장하게 된다.
 - `name` : `name`은 이 상수의 이름을 정의한다. `name`에 대해서는 나중에 좀 더 자세하게 설명하도록 하겠다.

간단한 예제를 하나 보자.

a,b,c 상수에, 각각 5,10,2 의 값을 넣은 후에, `d=a*b+c` 를 계산해서 계산 결과 d를 출력하려고 한다.

```
import tensorflow as tf

a = tf.constant([5],dtype=tf.float32)
b = tf.constant([10],dtype=tf.float32)
c = tf.constant([2],dtype=tf.float32)

d = a*b+c

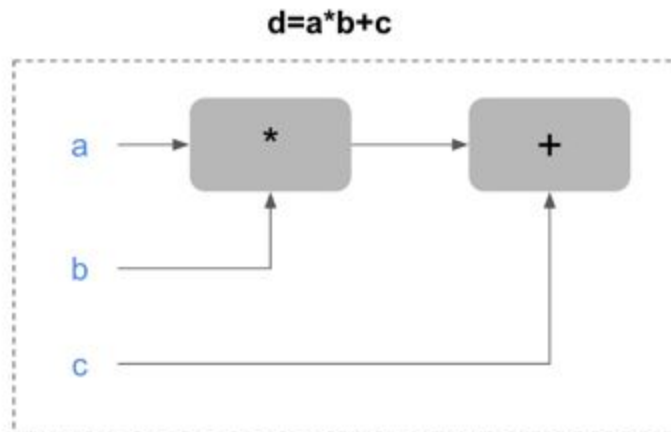
print d
```

그런데, 막상 실행해보면, `a*b+c`의 값이 아니라 다음과 같이 `Tensor...` 라는 문자열이 출력된다.

```
Tensor("add_8:0", shape=(1,), dtype=float32)
```

그래프와 세션의 개념

먼저 그래프와 세션이라는 개념을 이해해야 텐서플로우의 프로그래밍 모델을 이해할 수 있다. 위의 `d=a*b+c` 에서 d 역시 계산을 수행하는 것이 아니라 다음과 같이 `a*b+c` 그래프를 정의하는 것이다.



실제로 값을 뽑아내려면, 이 정의된 그래프에 a,b,c 값을 넣어서 실행해야 하는데, 세션(Session)을 생성하여, 그래프를 실행해야 한다. 세션은 그래프를 인자로 받아서 실행을 해주는 일종의 러너(Runner)라고 생각하면 된다.

자 그러면 위의 코드를 수정해보자

```
import tensorflow as tf

a = tf.constant([5],dtype=tf.float32)
b = tf.constant([10],dtype=tf.float32)
c = tf.constant([2],dtype=tf.float32)

d = a*b+c

sess = tf.Session()
result = sess.run(d)
print result
```

tf.Session()을 통하여 세션을 생성하고, 이 세션에 그래프 d를 실행하도록 sess.run(d)를 실행한다

이 그래프의 실행결과는 리턴값으로 result에 저장이 되고, 출력을 해보면 다음과 같이 정상적으로 52라는 값이 나오는 것을 볼 수 있다.

```
In [40]: import tensorflow as tf

a = tf.constant([5],dtype=tf.float32)
b = tf.constant([10],dtype=tf.float32)
c = tf.constant([2],dtype=tf.float32)

d = a*b+c

sess = tf.Session()
result = sess.run(d)
print result

[ 52.]
```

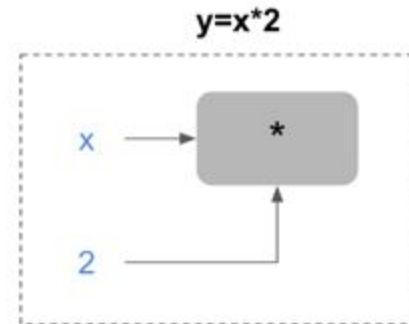
플레이스 홀더 (Placeholder)

자아 이제 상수의 개념을 알았으면, 이제는 플레이스 홀더에 대해서 알아보자.

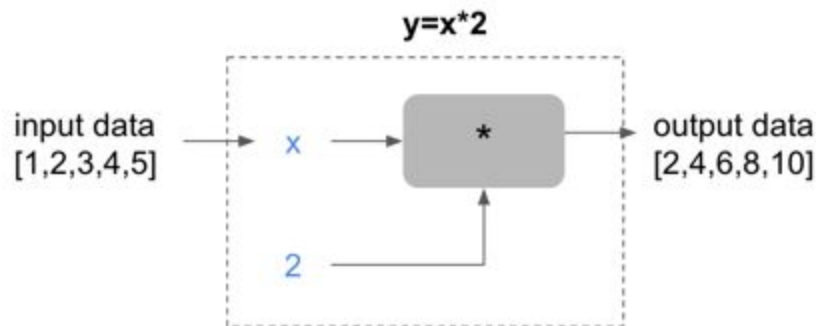
$y = x * 2$ 를 그래프를 통해서 실행한다고 하자. 입력값으로는 1,2,3,4,5를 넣고, 출력은 2,4,6,8,10을 기대한다고 하자. 이렇게 여러 입력값을 그래프에서 넣는 경우는 머신러닝에서 $y=W*x + b$ 와 같은 그래프가 있다고 할 때, x는 학습을 위한 데이터가 된다.

즉 지금 살펴보고자 하는 데이터 타입은 학습을 위한 학습용 데이터를 위한 데이터 타입이다.

$y=x*2$ 를 정의하면 내부적으로 다음과 같은 그래프가 된다.



그러면, x에는 값을 1,2,3,4,5를 넣어서 결과값을 그래프를 통해서 계산해 내야한다. 개념적으로 보면 다음과 같다.



이렇게 학습용 데이터를 담는 그릇을 플레이스홀더(placeholder)라고 한다.

플레이스홀더에 대해서 알아보면, 플레이스 홀더의 위의 그래프에서 x 즉 입력값을 저장하는 일종의 통(버킷)이다.

`tf.placeholder(dtype,shape,name)`

으로 정의된다.

플레이스 홀더 정의에 사용되는 변수들을 보면

- `dtype` : 플레이스홀더에 저장되는 데이터형이다. `tf.float32`와 같이 실수,정수등의 데이터 타입을 정의한다.
- `shape` : 행렬의 차원을 정의한다. `shape=[3,3]`으로 정의해주면, 이 플레이스홀더는 3x3 행렬을 저장하게 된다.
- `name` : name은 이 플레이스 홀더의 이름을 정의한다. name에 대해서는 나중에 좀 더 자세하게 설명하도록 하겠다.

그러면 이 x에 학습용 데이터를 어떻게 넣을 것인가? 이를 피딩(feeding)이라고 한다.

다음 예제를 보자

```
import tensorflow as tf

input_data = [1,2,3,4,5]
x = tf.placeholder(dtype=tf.float32)
y = x * 2

sess = tf.Session()
result = sess.run(y,feed_dict={x:input_data})

print result
```

처음 input_data=[1,2,3,4,5]으로 정의하고
다음으로 x=tf.placeholder(dtype=tf.float32) 를 이용하여, x를 float32 데이터형을 가지는
플레이스 홀더로 정의하다. shape은 편의상 생략하였다.
그리고 y=x * 2 로 그래프를 정의하였다.

세션이 실행될때, x라는 통에 값을 하나씩 집어 넣는데, (앞에서도 말했듯이 이를 피딩이라고
한다.)
sess.run(y,feed_dict={x:input_data}) 와 같이 세션을 통해서 그래프를 실행할 때, feed_dict
변수를 이용해서 플레이스홀더 x에, input_data를 피드하면, 세션에 의해서 그래프가 실행되면서
x는 feed_dict에 의해서 정해진 피드 데이터 [1,2,3,4,5]를 하나씩 읽어서 실행한다.

변수형 (Variable)

마지막 데이터형은 변수형으로,
y=W*x+b 라는 학습용 가설이 있을때, x가 입력데이터 였다면, W와 b는 학습을 통해서 구해야
하는 값이 된다. 이를 변수(Variable)이라고 하는데, 변수형은 Variable 형의 객체로 생성이 된다.

- tf.Variable.__init__(initial_value=None, trainable=True, collections=None, validate_shape=True, caching_device=None, name=None, variable_def=None, dtype=None, expected_shape=None, import_scope=None)

변수형에 값을 넣는 것은 다음과 같이 한다.

```
var = tf.Variable([1,2,3,4,5], dtype=tf.float32)
```

자 그러면 값을 넣어보고 코드를 실행해보자

```
import tensorflow as tf

input_data = [1,2,3,4,5]
```

```

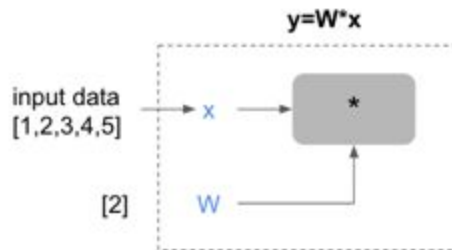
x = tf.placeholder(dtype=tf.float32)
W = tf.Variable([2],dtype=tf.float32)
y = W*x

sess = tf.Session()
result = sess.run(y,feed_dict={x:input_data})

print result

```

우리가 기대하는 결과는 다음과 같다. $y=W*x$ 와 같은 그래프를 가지고,



x는 [1,2,3,4,5] 값을 피딩하면서, 변수 W에 지정된 2를 곱해서 결과를 내기를 바란다. 그렇지만 코드를 실행해보면 다음과 같이 에러가 출력되는 것을 확인할 수 있다.

```

In [47]: import tensorflow as tf

input_data = [1,2,3,4,5]
x = tf.placeholder(dtype=tf.float32)
W = tf.Variable([2],dtype=tf.float32)
y = W*x

sess = tf.Session()
result = sess.run(y,feed_dict={x:input_data})

print result

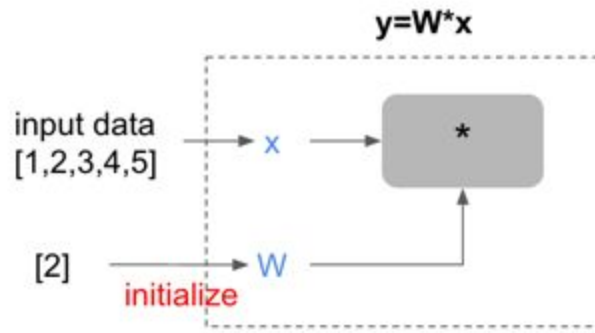
```

```

-----
FailedPreconditionError                                Traceback (most recent call last)
<ipython-input-47-964facc7a53c> in <module>()
      7
      8 sess = tf.Session()
----> 9 result = sess.run(y,feed_dict={x:input_data})
     10
     11 print result

```

이유는 텐서플로우에서 변수형은 그래프를 실행하기 전에 초기화를 해줘야 그 값이 변수에 지정이 된다.



세션을 초기화 하는 순간 변수 W에 그 값이 지정되는데, 초기화를 하는 방법은 다음과 같이 변수들을 `global_variables_initializer()` 를 이용해서 초기화 한후, 초기화된 결과를 세션에 전달해 줘야 한다.

```
init = tf.global_variables_initializer()
sess.run(init)
```

그러면 초기화를 추가한 코드를 보자

```
import tensorflow as tf

input_data = [1,2,3,4,5]
x = tf.placeholder(dtype=tf.float32)
W = tf.Variable([2],dtype=tf.float32)
y = W*x

sess = tf.Session()
init = tf.global_variables_initializer()
sess.run(init)
result = sess.run(y,feed_dict={x:input_data})

print result
```

초기화를 수행한 후, 코드를 수행해보면 다음과 같이 우리가 기대했던 결과가 출력됨을 확인할 수 있다.

```
import tensorflow as tf

input_data = [1,2,3,4,5]
x = tf.placeholder(dtype=tf.float32)
W = tf.Variable([2],dtype=tf.float32)
y = W*x

sess = tf.Session()
init = tf.global_variables_initializer()
sess.run(init)
result = sess.run(y,feed_dict={x:input_data})

print result
```

[2. 4. 6. 8. 10.]

텐서플로우를 처음 시작할때, Optimizer나 모델등에 대해 이해하는 것도 중요하지만, “데이터를 가지고 학습을 시켜서 적절한 값을 찾는다” 라는 머신러닝 학습 모델의 특성상, 모델을 그래프로 정의하고, 세션을 만들어서 그래프를 실행하고, 세션이 실행될때 그래프에 동적으로 값을 넣어가면서 (피딩) 실행한 다는 기본 개념을 잘 이해해야, 텐서플로우 프로그래밍을 제대로 시작할 수 있다.

머신러닝은 거의 모든 연산을 행렬을 활용한다. 텐서플로우도 이 행렬을 기반으로 하고, 이 행렬의 차원을 shape 라는 개념으로 표현하는데, 행렬에 대한 기본적인 개념이 없으면 헷갈리기 좋다. 그래서 이 글에서는 간략하게 행렬의 기본 개념과 텐서플로우내에서 표현 방법에 대해서 알아보도록 한다.

행렬과 텐서플로우

행렬의 기본 개념

행과 열

행렬의 가장 기본 개념은 행렬이다. $m \times n$ 행렬이 있을때, m 은 행, n 은 열을 나타내며, 행은 세로의 줄수, 열은 가로줄 수 를 나타낸다. 아래는 3×4 (3행4열) 행렬이다.

$$\begin{pmatrix} A_{11}, A_{12}, A_{13}, A_{14} \\ A_{21}, A_{22}, A_{23}, A_{24} \\ A_{31}, A_{32}, A_{33}, A_{34} \end{pmatrix} \begin{matrix} \downarrow 3\text{행} \\ \rightarrow 4\text{열} \end{matrix}$$

\$

곱셈

곱셈은 앞의 행렬에서 행과, 뒤의 행렬의 열을 순차적으로 곱해준다.
아래 그림을 보면 쉽게 이해가 될것이다.

$$\begin{pmatrix} A_{11}, A_{12}, A_{13} \\ A_{21}, A_{22}, A_{23} \\ A_{31}, A_{32}, A_{33} \end{pmatrix} \times \begin{pmatrix} B_{11}, B_{12} \\ B_{21}, B_{22} \\ B_{31}, B_{32} \end{pmatrix} = \begin{pmatrix} A_{11} * B_{11} + A_{12} * B_{21} + A_{13} * B_{31} \\ A_{11} * B_{12} + A_{12} * B_{22} + A_{13} * B_{32} \\ A_{21} * B_{11} + A_{22} * B_{21} + A_{23} * B_{31} \\ A_{21} * B_{12} + A_{22} * B_{22} + A_{23} * B_{32} \\ A_{31} * B_{11} + A_{32} * B_{21} + A_{33} * B_{31} \\ A_{31} * B_{12} + A_{32} * B_{22} + A_{33} * B_{32} \end{pmatrix}$$

이렇게 앞 행렬의 행과 열을 곱해나가면 결과적으로 아래와 같은 결과가 나온다.

$$\begin{pmatrix} A_{11}, A_{12}, A_{13} \\ A_{21}, A_{22}, A_{23} \\ A_{31}, A_{32}, A_{33} \end{pmatrix} \times \begin{pmatrix} B_{11}, B_{12} \\ B_{21}, B_{22} \\ B_{31}, B_{32} \end{pmatrix} = \begin{pmatrix} A_{11} * B_{11} + A_{12} * B_{21} + A_{13} * B_{31}, A_{11} * B_{12} + A_{12} * B_{22} + A_{13} * B_{32} \\ A_{21} * B_{11} + A_{22} * B_{21} + A_{23} * B_{31}, A_{21} * B_{12} + A_{22} * B_{22} + A_{23} * B_{32} \\ A_{31} * B_{11} + A_{32} * B_{21} + A_{33} * B_{31}, A_{31} * B_{12} + A_{32} * B_{22} + A_{33} * B_{32} \end{pmatrix}$$

$\begin{matrix} 3 \times 3 \\ (m \times k) \end{matrix} \quad \begin{matrix} 3 \times 2 \\ (k \times n) \end{matrix} \quad \begin{matrix} 3 \times 2 \\ (m \times n) \end{matrix}$

이때 앞의 행렬의 열과, 뒤의 행렬의 행이 같아야 곱할 수 있다.

즉 axb 행렬과 mxn 행렬이 있을때, 이 두 행렬을 곱하려면 b 와 m 이 같아야 한다.
그리고 이 두 행렬을 곱하면 axn 사이즈의 행렬이 나온다.

행렬의 덧셈과 뺄셈

행렬의 덧셈과 뺄셈은 단순하다. 같은 행과 열에 있는 값을 더하거나 빼주면 되는데, 단지 주의할점은 덧셈과 뺄셈을 하는 두개의 행렬의 차원이 동일해야 한다.

$$\begin{pmatrix} A_{11}, A_{12} \\ A_{21}, A_{22} \\ A_{31}, A_{32} \end{pmatrix} + \begin{pmatrix} B_{11}, B_{12} \\ B_{21}, B_{22} \\ B_{31}, B_{32} \end{pmatrix} = \begin{pmatrix} A_{11}+B_{11}, A_{12}+B_{12} \\ A_{21}+B_{21}, A_{22}+B_{22} \\ A_{31}+B_{31}, A_{32}+B_{32} \end{pmatrix}$$

텐서 플로우에서 행렬의 표현

행렬에 대해서 간단하게 되짚어 봤으면, 그러면 텐서 플로우에서는 어떻게 행렬을 표현하는지 알아보자

$$(1.0 \ 2.0 \ 3.0) \times \begin{pmatrix} 2.0 \\ 2.0 \\ 2.0 \end{pmatrix}$$

을 하는 코드를 살펴보자

예제코드

```
import tensorflow as tf

x = tf.constant([ [1.0,2.0,3.0] ])
w = tf.constant([ [2.0],[2.0],[2.0] ])
y = tf.matmul(x,w)
print x.get_shape()

sess = tf.Session()
init = tf.global_variables_initializer()
sess.run(init)
result = sess.run(y)

print result
```

실행 결과

```
(1, 3)
[[ 12.]]
```

텐서플로우에서 행렬의 곱셈은 일반 * 를 사용하지 않고, 텐서플로우 함수 “tf.matmul” 을 사용한다.

중간에, x.get_shape()를 통해서, 행렬 x의 shape를 출력했는데, shape는 행렬의 차원이라고 생각하면 된다. x는 1행3열인 1x3 행렬이기 때문에, 위의 결과와 같이 (1,3)이 출력된다.

앞의 예제에서는 constant 에 저장된 행렬에 대한 곱셈을 했는데, 당연히 Variable 형에서도 가능하다.

예제 코드

```
import tensorflow as tf
```

```
x = tf.Variable([ [1.,2.,3.] ], dtype=tf.float32)
w = tf.constant([ [2.],[2.],[2.]], dtype=tf.float32)
y = tf.matmul(x,w)
```

```
sess = tf.Session()
init = tf.global_variables_initializer()
sess.run(init)
result = sess.run(y)
```

```
print result
```

Constant 및 Variable 뿐 아니라, Placeholder에도 행렬로 저장이 가능하다 다음은 Placeholder에 행렬 데이터를 feeding 해주는 예제이다.

입력 데이터 행렬 x는 Placeholder 타입으로 3x3 행렬이고, 여기에 곱하는 값 w는 1x3 행렬이다.

$$\begin{pmatrix} 1.0 & 2.0 & 3.0 \\ 1.0 & 2.0 & 3.0 \\ 2.0 & 3.0 & 4.0 \end{pmatrix} \times \begin{pmatrix} 2.0 \\ 2.0 \\ 2.0 \end{pmatrix}$$

예제 코드는 다음과 같다.

예제코드

```
import tensorflow as tf
```

```
input_data = [ [1.,2.,3.],[1.,2.,3.],[2.,3.,4.] ] #3x3 matrix
x = tf.placeholder(dtype=tf.float32,shape=[None,3])
w = tf.Variable([ [2.],[2.],[2.] ], dtype = tf.float32) #3x1 matrix
y = tf.matmul(x,w)
```

```
sess = tf.Session()
init = tf.global_variables_initializer()
sess.run(init)
```

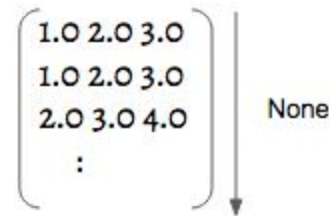
```
result = sess.run(y,feed_dict={x:input_data})
```

```
print result
```

실행결과

```
[[ 12.]  
 [ 12.]  
 [ 18.]]
```

이 예제에서 주의 깊게 봐야 할부분은 placeholder x 를 정의하는 부분인데, shape=[None,3] 으로 정의했다 3x3 행렬이기 때문에, shape=[3,3]으로 지정해도 되지만 None 이란, 갯수를 알수 없음을 의미하는 것으로, 텐서플로우 머신러닝 학습에서 학습 데이터가 계속해서 들어오고 학습 때마다 데이터의 양이 다를 수 있기 때문에, 이를 지정하지 않고 None으로 해놓으면 들어오는 숫자 만큼에 맞춰서 저장을 한다.



브로드 캐스팅

텐서플로우 그리고 파이썬으로 행렬 프로그래밍을 하다보면 헛갈리는 개념이 브로드 캐스팅이라는 개념이 있다. 먼저 다음 코드를 보자

예제코드

```
import tensorflow as tf

input_data = [
    [1,1,1],[2,2,2]
]
x = tf.placeholder(dtype=tf.float32,shape=[2,3])
w =tf.Variable([[2],[2],[2]],dtype=tf.float32)
b =tf.Variable([4],dtype=tf.float32)
y = tf.matmul(x,w)+b

print x.get_shape()
sess = tf.Session()
init = tf.global_variables_initializer()
sess.run(init)
result = sess.run(y,feed_dict={x:input_data})
```

print result

실행결과

(2, 3)

[[24.]

[48.]]

행렬 x는 2x3 행렬이고 w는 3x1 행렬이다. x*w를 하면 2*1 행렬이 나온다.

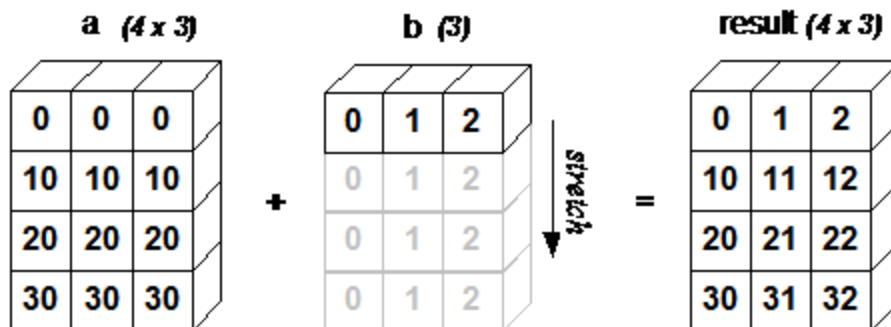
문제는 +b 인데, b는 1*1 행렬이다. 행렬의 덧셈과 뺄셈은 차원이 맞아야 하는데, 이 경우 더하고자 하는 대상은 2*1, 더하려는 b는 1*1로 행렬의 차원이 다르다. 그런데 어떻게 덧셈이 될까?

이 개념이 브로드 캐스팅이라는 개념인데, 위에서는 1*1인 b행렬을 더하는 대상에 맞게 2*1 행렬로 자동으로 늘려서 (stretch) 계산한다.

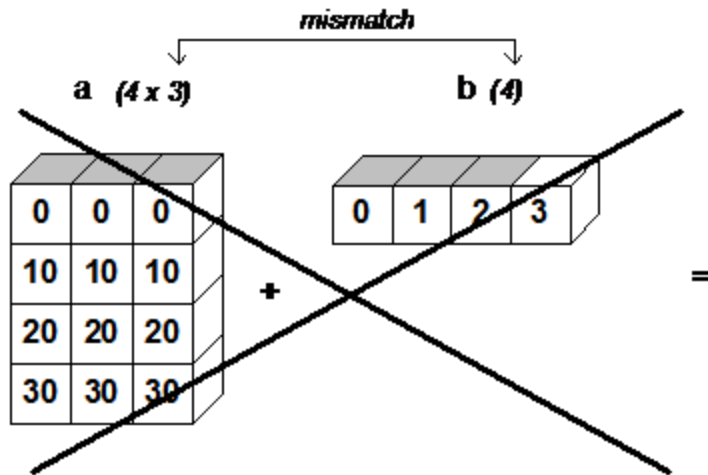
브로드 캐스팅은 행렬 연산 (덧셈, 뺄셈, 곱셈)에서 차원이 맞지 않을때, 행렬을 자동으로 늘려줘서(Stretch) 차원을 맞춰주는 개념으로 늘리는 것은 가능하지만 줄이는 것은 불가능하다.

브로드 캐스팅 개념은 <http://scipy.github.io/old-wiki/pages/ErrataBroadcastingDoc> 에 잘 설명되어 있으니 참고하기 바란다.

아래는 4x3 행렬 a와 1x3 행렬 b를 더하는 연산인데, 차원이 맞지 않기 때문에, 행렬 b의 열을 늘려서 1x3 → 4x3 으로 맞춰서 연산한 예이다.

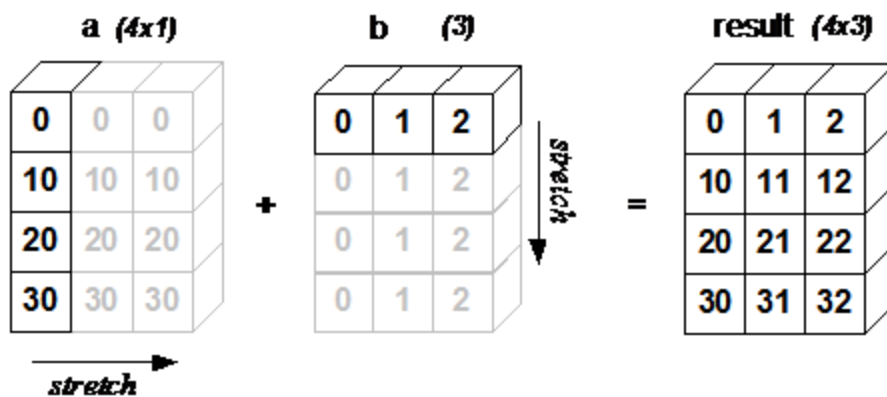


만약에 행렬 b가 아래 그림과 같이 1x4 일 경우에는 열을 4 → 3으로 줄이고, 세로 행을 1 → 4로 늘려야 하는데, 앞에서 언급한바와 같이, 브로드 캐스팅은 행이나 열을 줄이는 것은 불가능하다.



다음은 양쪽 행렬을 둘다 늘린 케이스 이다.

4x1 행렬 a와 1x3 행렬 b를 더하면 양쪽을 다 수용할 수 있는 큰 차원인 4x3 행렬로 변환하여 덧셈을 수행한다.



텐서플로우 용어 참고

텐서플로우에서는 행렬을 차원에 따라서 다음과 같이 호칭한다.

행렬이 아닌 숫자나 상수는 Scalar, 1차원 행렬을 Vector, 2차원 행렬을 Matrix, 3차원 행렬을 3-Tensor 또는 cube, 그리고 이 이상의 다차원 행렬을 N-Tensor라고 한다.

Rank	Math entity	Python example
0	Scalar (magnitude only)	<code>s = 483</code>
1	Vector (magnitude and direction)	<code>v = [1.1, 2.2, 3.3]</code>
2	Matrix (table of numbers)	<code>m = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]</code>
3	3-Tensor (cube of numbers)	<code>t = [[[2], [4], [6]], [[8], [10], [12]], [[14], [16], [18]]]</code>
n	n-Tensor (you get the idea)	<code>....</code>

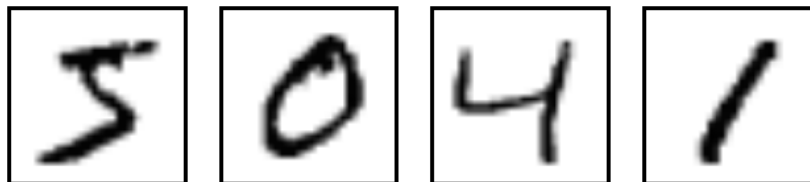
그리고 행렬의 차원을 Rank라고 부른다. scalar는 Rank가 0, Vector는 Rank가 1, Matrix는 Rank가 2가 된다.

5. 소프트맥스를 이용한 다항 분류 모델 구현

텐서플로우와 머신러닝에 대한 개념에 대해서 대략적으로 이해 했으면 간단한 코드를 한번 짜보자.

MNIST

그러면 이제 실제로 텐서플로우로 모델을 만들어서 학습을 시켜보자. 예제에 사용할 시나리오는 MNIST (Mixed National Institute of Standards and Technology database) 라는 데이터로, 손으로 쓴 숫자이다. 이 손으로 쓴 숫자 이미지를 0~9 사이의 숫자로 인식하는 예제이다.



이 예제는 텐서플로우 MNIST 튜토리얼 (<https://www.tensorflow.org/tutorials/mnist/beginners/>) 을 기반으로 작성하였는데, 설명이 빠진 부분과 소스코드 일부분이 수정되었으니 내용이 약간 다르다는 것을 인지해주기를 바란다.

MNIST 숫자 이미지를 인식하는 모델을 softmax 알고리즘을 이용하여 만든 후에, 트레이닝을 시키고, 정확도를 체크해보도록 하겠다.

데이터셋

MNIST 데이터는 텐서플로우 내에 라이브러리 형태로 내장이 되어 있어서 쉽게 사용이 가능하다. tensorflow.examples.tutorials.mnist 패키지에 데이터가 들어 있는데, read_data_sets 명령어를 이용하면 쉽게 데이터를 로딩할 수 있다.

데이터 로딩 코드

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data
```

```
mnist = input_data.read_data_sets('/tmp/tensorflow/mnist/input_data', one_hot=True)
```

Mnist 데이터셋에는 총 60,000개의 데이터가 있는데, 이 데이터는 크게 아래와 같이 세 종류의 데이터 셋으로 나뉜다. 모델 학습을 위한 학습용 데이터인 `mnist.train` 그리고, 학습된 모델을 테스트하기 위한 테스트 데이터 셋은 `mnist.test`, 그리고 모델을 확인하기 위한 `mnist.validation` 데이터셋으로 구별된다.

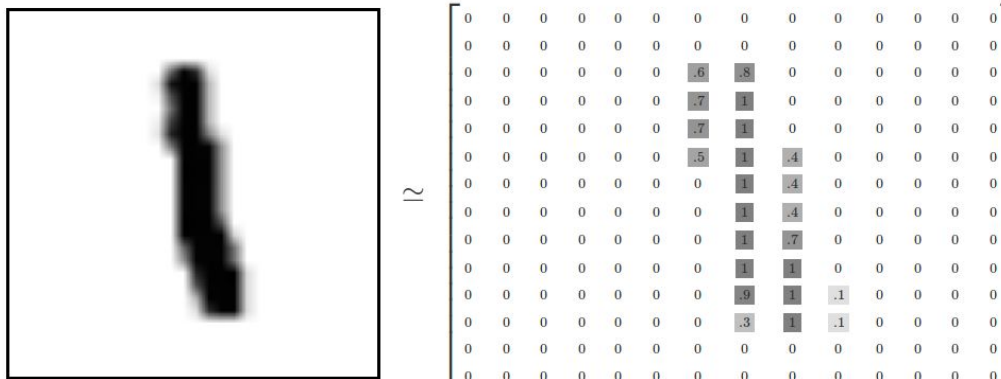
각 데이터는 아래와 같이 학습용 데이터 55000개, 테스트용 10,000개, 그리고 확인용 데이터 5000개로 구성되어 있다.

데이터셋 명	행렬 차원	데이터 종류	노트
<code>mnist.train.images</code>	55000 x 784	학습 이미지 데이터	
<code>mnist.train.labels</code>	55000 x 10	학습 라벨 데이터	
<code>mnist.test.images</code>	10000 x 784	테스트용 이미지 데이터	
<code>mnist.test.labels</code>	10000 x 10	테스트용 라벨 데이터	
<code>mnist.validation.images</code>	5000 x 784	확인용 이미지 데이터	
<code>mnist.validation.labels</code>	5000 x 10	확인용 라벨 데이터	

각 데이터셋은 학습을 위한 글자 이미지를 저장한 데이터 `image` 와, 그 이미지가 어떤 숫자인지를 나타낸 라벨 데이터인 `label`로 두개의 데이터 셋으로 구성되어 있다.

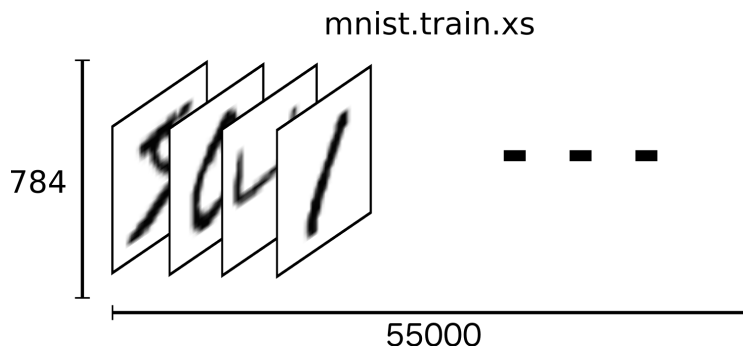
이미지

먼저 이미지 데이터를 보면 아래 그림과 같이 28x28 로 구성되어 있는데,



이를 2차원 행렬에서 1차원으로 짤막한 형태로 784개의 열을 가진 1차원 행렬로 변환되어 저장되어 있다.

`mnist.train.image`는 이러한 784개의 열로 구성된 이미지가 55000개가 저장되어 있다.



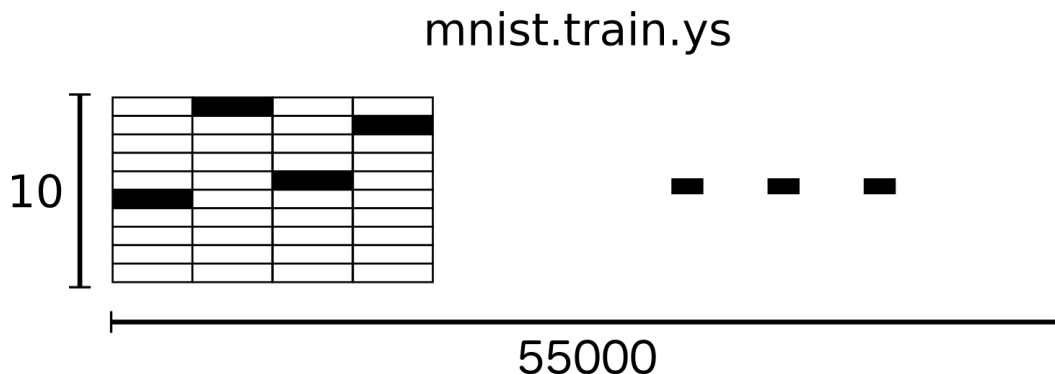
텐서플로우의 행렬을 나타내는 `shape`의 형태로는 `shape=[55000,784]` 이 된다.

마찬가지로, `mnist.train.image` 도 784개의 열로 구성된 숫자 이미지 데이터를 10000개를 가지고 있고 텐서플로우의 `shape`으로는 `shape=[10000,784]` 로 표현될 수 있다.

라벨

Label 은 이미지가 나타내는 숫자가 어떤 숫자인지를 나타내는 라벨 데이터로 10개의 숫자로 이루어진 1행 행렬이다. 0~9 순서로, 그 숫자이면 1 아니면 0으로 표현된다. 예를 들어 1인 경우는 `[0,1,0,0,0,0,0,0,0,0]` 9인 경우는 `[0,0,0,0,0,0,0,0,0,1]` 로 표현된다.

이미지 데이터에 대한 라벨이기 때문에, 당연히 이미지 데이터 수만큼의 라벨을 가지게 된다.



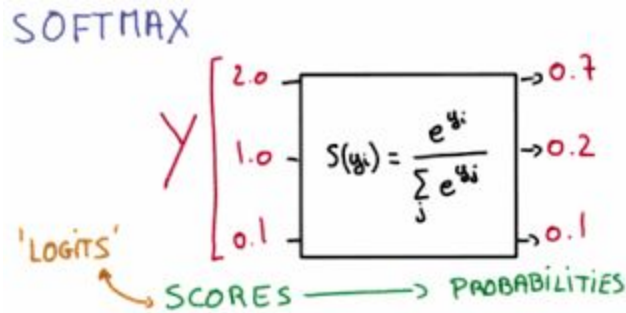
Train 데이터 셋은 이미지가 55000개 있기 때문에, Train의 label의 수 역시도 55000개가 된다.

소프트맥스 회귀(Softmax regression)

숫자 이미지를 인식하는 모델은 많지만, 여기서는 간단한 알고리즘 중 하나인 소프트 맥스 회귀 모델을 사용하겠다.

소프트맥스 회귀에 대한 알고리즘 자체는 자세히 설명하지 않는다. 소프트맥스 회귀는 classification 알고리즘중의 하나로, 들어온 값이 어떤 분류인지 구분해주는 알고리즘이다.

예를 들어 A,B,C 3개의 결과로 분류해주는 소프트맥스의 경우 결과값은 [0.7,0.2,0.1] 와 같이 각각 A,B,C일 확률을 리턴해준다. (결과값의 합은 1.0이 된다.)



(cf. 로지스틱 회귀는 두 가지로만 분류가 가능하지만, 소프트맥스 회귀는 n 개의 분류로 구분이 가능하다.)

모델 정의

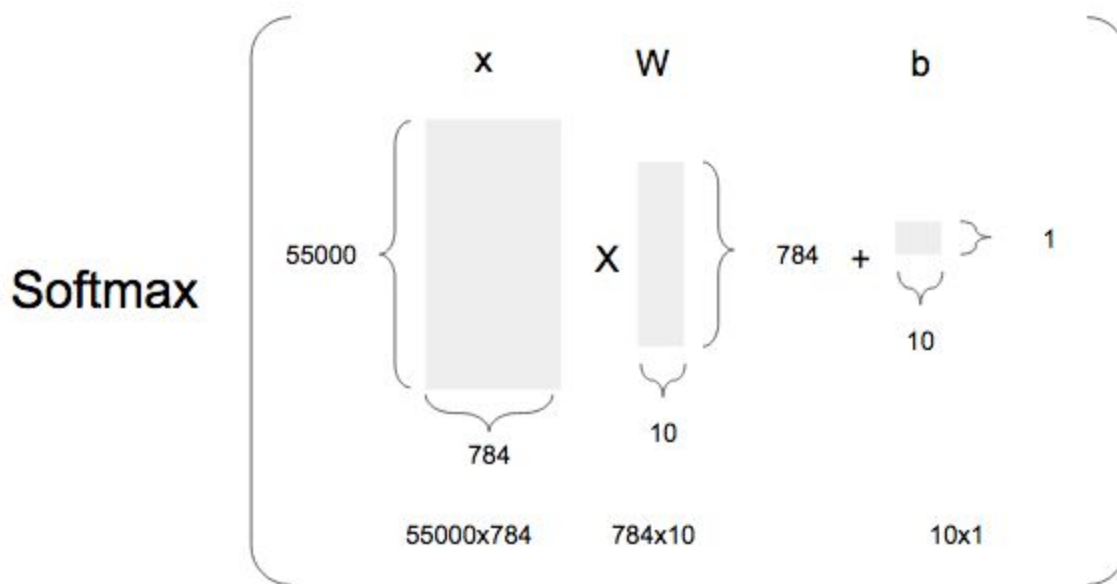
소프트맥스로 분류를 할때, x라는 값이 들어 왔을때, 분류를 한다고 가정했을때, 모델에서 사용하는 가설은 다음과 같다.

$$y = \text{softmax}(W \cdot x + b)$$

W는 weight, 그리고 b는 bias 값이다.

y는 최종적으로 10개의 숫자를 감별하는 결과가 나와야 하기 때문에, 크기가 10인 행렬이 되고, 10개의 결과를 만들기 위해서 W역시 10개가 되어야 하며, 이미지 하나는 784개의 숫자로 되어 있기 때문에, 10개의 값을 각각 784개의 숫자에 적용해야 하기 때문에, W는 784x10 행렬이 된다. 그리고, b는 10개의 값에 각각 더하는 값이기 때문에, 크기가 10인 행렬이 된다.

이를 표현해보면 다음과 같은 그림이 된다.



이를 텐서플로우 코드로 표현하면 다음과 같다.

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

```
W = tf.Variable(tf.zeros([784, 10]))
```

```
b = tf.Variable(tf.zeros([10]))
```

```
k = tf.matmul(x, W) + b
```

```
y = tf.nn.softmax(k)
```

우리가 구하고자 하는 값은 x 값으로 학습을 시켜서 0~9를 가장 잘 구별해내는 W 와 b 의 값을 찾는 일이다.

여기서 코드를 주의깊게 봤다면 하나의 의문이 생길것이다.

x 의 데이터는 총 55000개로, 55000x784 행렬이 되고, W 는 784x10 행렬이다. 이 둘을 곱하면, 55000x10 행렬이 되는데, b 는 1x10 행렬로 차원이 달라서 합이 되지 않는다.

텐서플로우와 파이썬에서는 이렇게 차원이 다른 행렬을 큰 행렬의 크기로 늘려주는 기능이 있는데, 이를 브로드 캐스팅이라고 한다. (브로드 캐스팅 개념 참고 -

<http://bcho.tistory.com/1153>)

브로드 캐스팅에 의해서 b 는 55000x10 사이즈로 자동으로 늘어나고 각 행에는 첫행과 같은 데이터들로 채워지게 된다.

소프트맥스 알고리즘을 이해하고 사용해도 좋지만, 텐서플로우에는 이미 `tf.nn.softmax` 라는 함수로 만들어져 있고, 대부분 많이 알려진 머신러닝 모델들은 샘플들이 많이 있기 때문에, 대략적인 원리만 이해하고 가져다 쓰는 것을 권장한다. 보통 모델을 다 이해하려고 하다가 수학에서 부딪혀서 포기하는 경우가 많은데, 디테일한 모델을 이해하기 힘들면, 그냥 함수나 예제코드를 가져다 쓰는 방법으로 접근하자. 우리가 일반적인 프로그래밍에서도 해쉬테이블이나 트리와 같은 자료구조에 대해서 대략적인 개념만 이해하고 미리 정의된 라이브러리를 사용하지 직접 해쉬 테이블등을 구현하는 경우는 드물다.

코스트(비용) 함수

이 소프트맥스 함수에 대한 코스트 함수는 크로스엔트로피 (Cross entropy) 함수의 평균을 이용하는데, 복잡한 산식 없이 그냥 외워서 쓰자. 다행히도 크로스엔트로피 함수역시 함수로 구현이 되어있다.

```
Cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(tf.matmul(x, W) + b, y_))
```

가설에 의해 계산된 값 y 를 넣지 않고 `tf.matmul(x, W) + b` 를 넣은 이유는

`tf.nn.softmax_cross_entropy_with_logits` 함수 자체가 softmax를 포함하기 때문이다.

$y_$ 은 학습을 위해서 입력된 값이다.

텐서플로우로 구현

자 그럼 학습을 위한 전체 코드를 보자

샘플코드

```
# Import data
from tensorflow.examples.tutorials.mnist import input_data
import tensorflow as tf

mnist = input_data.read_data_sets('/tmp/tensorflow/mnist/input_data', one_hot=True)

# Create the model
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))
k = tf.matmul(x, W) + b
y = tf.nn.softmax(k)

# Define loss and optimizer
y_ = tf.placeholder(tf.float32, [None, 10])
learning_rate = 0.5
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(k, y_))
train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)

print ("Training")
sess = tf.Session()
init = tf.global_variables_initializer() #.run()
sess.run(init)
for _ in range(1000):
    # 1000번씩, 전체 데이터에서 100개씩 뽑아서 트레이닝을 함.
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})

print ('b is ',sess.run(b))
print('W is',sess.run(W))
```

데이터 로딩

```
# Import data
from tensorflow.examples.tutorials.mnist import input_data
import tensorflow as tf

mnist = input_data.read_data_sets('/tmp/tensorflow/mnist/input_data', one_hot=True)
```

앞에서 데이터에 대해서 설명한 것과 같이 데이터를 로딩하는 부분이다. `read_data_sets`에 들어가 있는 디렉토리는 샘플데이터를 온라인에서 다운 받는데, 그 데이터를 임시로 저장해놓을 위치이다.

모델 정의

다음은 소프트맥스를 이용하여 모델을 정의한다.

```
# Create the model
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))
k = tf.matmul(x, W) + b
y = tf.nn.softmax(k)
```

x는 트레이닝 데이터를 저장하는 스테이크홀더, W는 Weight, b는 bias 값이고, 모델은 $y = \text{tf.nn.softmax}(\text{tf.matmul}(x, W) + b)$ 이 된다.

코스트함수와 옵티마이저 정의

모델을 정의했으면 학습을 위해서, 코스트 함수를 정의한다.

```
# Define loss and optimizer
y_ = tf.placeholder(tf.float32, [None, 10])
learning_rate = 0.5
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(y, y_))
train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
```

코스트 함수는 크로스 엔트로피 함수의 평균값을 사용한다. 크로스엔트로피 함수는 아래와 같은 모양인데, 이 값을 전체 트레이닝 데이터셋의 수로 나눠 준다.

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

그래서 최종적으로 cost 함수는 $\text{cost} =$

$\text{tf.reduce_mean}(\text{tf.nn.softmax_cross_entropy_with_logits}(k, y_))$ 이 된다.

이 때 주의할점은 y가 아니라 k를 넣어야 한다. softmax_cross_entropy_with_logits 함수는 softmax를 같이 하기 때문에, 위의 y값은 이미 softmax를 해버린 함수이기 때문에 softmax가 중복될 수 있다.

이 코스트 함수를 가지고 코스트가 최소화가 되는 W와 b를 구해야 하는데, 옵티마이저를 사용한다. 여기서는 경사 하강법(Gradient Descent Optimizer)를 사용하였고 경사하강법에 대한 개념은 <http://bcho.tistory.com/1141> 를 참고하기 바란다.

GradientDescent에서 learning rate는 학습속도 인데, 학습 속도에 대한 개념은 <http://bcho.tistory.com/1141> 글을 참고하기 바란다.

세션 초기화

```
print ("Training")
sess = tf.Session()
init = tf.global_variables_initializer() #.run()
sess.run(init)
```

tf.Session() 을 이용해서 세션을 만들고, global_variable_initializer()를 이용하여, 변수들을 모두 초기화한후, 초기화 값을 sess.run에 넘겨서 세션을 초기화 한다.

트레이닝 시작

세션이 생성되었으면 이제 트레이닝을 시작한다.

```
for _ in range(1000):
    # 1000번씩, 전체 데이터에서 100개씩 뽑아서 트레이닝을 함.
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
```

여기서 주목할점은 Batch training 과 Stochastic training 인데, Batch training이란, 학습을 할때 전체 데이터를 가지고 한번에 학습을 하는게 아니라 전체 데이터셋을 몇 개로 쪼갬후 나눠서 트레이닝을 하는 방법을 배치 트레이닝이라고 한다. 그중에서 여기에 사용된 배치 방법은 Stochastic training 이라는 방법인데, 원칙대로라면 전체 55000개 의 학습데이터가 있기 때문에 배치 사이즈를 100으로 했다면, 100개씩 550번 순차적으로 데이터를 읽어서 학습을 해야겠지만, Stochastic training은 전체 데이터중 일부를 샘플링해서 학습하는 방법으로, 여기서는 배치 한번에 100개씩의 데이터를 뽑아서 1000번 배치로 학습을 하였다.

(텐서플로우 문서에 따르면, 전체 데이터를 순차적으로 학습 시키기에는 연산 비용이 비싸기 때문에, 샘플링을 해도 비슷한 정확도를 낼 수 있기 때문에, 예제 차원에서 간단하게, Stochastic training을 사용한것으로 보인다.)

결과값 출력

```
print ('b is ',sess.run(b))
```

```
print('W is',sess.run(W))
```

마지막으로 학습에서 구해진 W와 b를 출력해보자

다음은 실행 결과 스크린 샷이다.

```
Extracting /tmp/tensorflow/mnist/input_data/train-images-idx3-ubyte.gz
Extracting /tmp/tensorflow/mnist/input_data/train-labels-idx1-ubyte.gz
Extracting /tmp/tensorflow/mnist/input_data/t10k-images-idx3-ubyte.gz
Extracting /tmp/tensorflow/mnist/input_data/t10k-labels-idx1-ubyte.gz
Training
('b is ', array([-0.11368412,  0.34618276, -0.06817129, -0.12768586,  0.14948681,
                  0.36769056,  0.03084462,  0.29830146, -0.73656094, -0.1464057 ], dtype=float32))
('W is', array([[ 0.,  0.,  0., ...,  0.,  0.,  0.],
                 [ 0.,  0.,  0., ...,  0.,  0.,  0.],
                 [ 0.,  0.,  0., ...,  0.,  0.,  0.],
                 ...,
                 [ 0.,  0.,  0., ...,  0.,  0.,  0.],
                 [ 0.,  0.,  0., ...,  0.,  0.,  0.],
                 [ 0.,  0.,  0., ...,  0.,  0.,  0.]], dtype=float32))
```

먼저 앞에서 데이터를 로딩하도록 지정한 디렉토리에, 학습용 데이터를 다운 받아서 압축 받는 것을 확인할 수 있다. (Extracting.. 부분)

그 다음 학습이 끝난후에, b와 W 값이 출력되었다. W는 784 라인이기 때문에, 중간을 생략하고 출력되었으나, 각 행을 모두 찍어보면 아래와 같이 W 값이 들어가 있는 것을 볼 수 있다.

```
(9, array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.], dtype=float32))
(10, array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.], dtype=float32))
(11, array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.], dtype=float32))
(12, array([-5.53695190e-06, -7.94481912e-06, -1.17996518e-04,
            -1.55862726e-05, -6.33553544e-04, -7.89074238e-06,
             9.26191802e-04, -6.59273519e-06, -3.50811206e-05,
            -9.60088291e-05], dtype=float32))
(13, array([-2.87244147e-05, -2.74839131e-05,  3.27478920e-04,
            -3.99608616e-05, -1.66243373e-03, -2.43990653e-05,
             1.85890321e-03, -2.96408980e-05, -1.02268248e-04,
            -2.71470431e-04], dtype=float32))
(14, array([-1.84730688e-05, -1.12254556e-05,  6.51939656e-04,
            -6.49023241e-06, -3.06211441e-04, -7.92438004e-06,
            -1.88217658e-04, -1.69203449e-05, -2.83239824e-05,
            -6.81530873e-05], dtype=float32))
```

모델 검증

이제 모델을 만들고 학습을 시켰으니, 이 모델이 얼마나 정확하게 작동하는지를 테스트 해보자. mnist.test.image 와 mnist.test.labels 데이터셋을 이용하여 테스트를 진행하는데, 앞에서 나온 모델에 mnist.test.image 데이터를 넣어서 예측을 한 후에, 그 결과를 mnist.test.labels (정답)과 비교해서 정답률이 얼마나 되는지를 비교한다.

다음은 모델 테스트 코드이다. 이 코드를 위의 코드 뒤에 붙여서 실행하면 된다.

모델 검증 코드

```
print ("Testing model")
```

```
# Test trained model
correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
print('accuracy ', sess.run(accuracy, feed_dict={x: mnist.test.images,
                                                  y_: mnist.test.labels}))

print ("done")
```

correct_prediction = tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))
 코드를 보자, tf.argmax 함수를 이해해야 하는데, argmax(y,1)은 행렬 y에서 몇번째에 가장 큰 값이 들어가 있는지를 리턴해주는 함수이다. 아래 예제 코드를 보면

```
session = tf.InteractiveSession()
```

```
data = tf.constant([9,2,11,4])
idx = tf.argmax(data,0)
print idx.eval()
```

```
session.close()
```

[9,2,11,4] 에서 최대수는 11이고, 이 위치는 두번째 (0 부터 시작한다)이기 때문에 0을 리턴한다. 두번째 변수는 어느축으로 카운트를 할것인지를 선택한다. , 1차원 배열의 경우에는 0을 사용한다.

여기서 y는 2차원 행렬인데, 0이면 같은 열에서 최대값인 순서, 1이면 같은 행에서 최대값인 순서를 리턴한다.

그럼 원래 코드로 돌아오면 tf.argmax(y,1)은 y의 각행에서 가장 큰 값의 순서를 찾는다. y의 각행을 0~9으로 인식한 이미지의 확률을 가지고 있다.

아래는 4를 인식한 y 값인데, 4의 값이 0.7로 가장높기 (4일 확률이 70%, 3일 확률이 10%, 1일 확률이 20%로 이해하면 된다.) 때문에, 4로 인식된다.

0	1	2	3	4	5	6	7	8	9
0.0	0.2	0.0	0.1	0.7	0.0	0.0	0.0	0.0	0.0

여기서 tf.argmax(y,1)을 사용하면, 행별로 가장 큰 값을 리턴하기 때문에, 위의 값에서는 4가 리턴이된다.

테스트용 데이터에서 원래 정답이 4로 되어 있다면, argmax(y_,1)도 4를 리턴하기 때문에, tf.equal(tf.argmax(y, 1), tf.argmax(y_, 1))는 tf.equals(4,4)로 True를 리턴하게 된다.

모든 테스트 셋에 대해서 검증을 하고 나서 그 결과에서 True만 더해서, 전체 트레이닝 데이터의 수로 나눠 주면 결국 정확도가 나오는데, tf.cast(boolean, tf.float32)를 하면 텐서플로우의 bool

값을 float32 (실수)로 변환해준다. True는 1.0으로 False는 0.0으로 변환해준다. 이렇게 변환된 값들의 전체 평균을 구하면 되기 때문에, tf.reduce_mean을 사용한다.

이렇게 정확도를 구하는 함수가 정의되었으면 이제 정확도를 구하기 위해 데이터를 넣어보자
sess.run(accuracy, feed_dict={x: mnist.test.images, y_: mnist.test.labels})
x에 mnist.test.images 데이터셋으로 이미지 데이터를 입력받아서 y (예측 결과)를 계산하고,
y_에는 mnist.test.labels 정답을 입력 받아서, y와 y_로 정확도 accuracy를 구해서 출력한다.

최종 출력된 accuracy 정확도는 0.9 로 대략 90% 정도가 나온다.

```
Testing model  
( 'accuracy ', 0.90719998)  
done
```

다른 알고리즘의 정확도는

http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html 를 참고하면 된다.

다음글에서는 소프트맥스 모델 대신 CNN (Convolutional Neural Network)를 이용하여, 조금 더 정확도가 높은 MNIST를 구현하고 테스트해보도록 하겠다.

참고 자료

- 텐서플로우 MNIST <https://www.tensorflow.org/tutorials/mnist/beginners/>

6. 딥러닝과 컨볼루셔널 네트워크

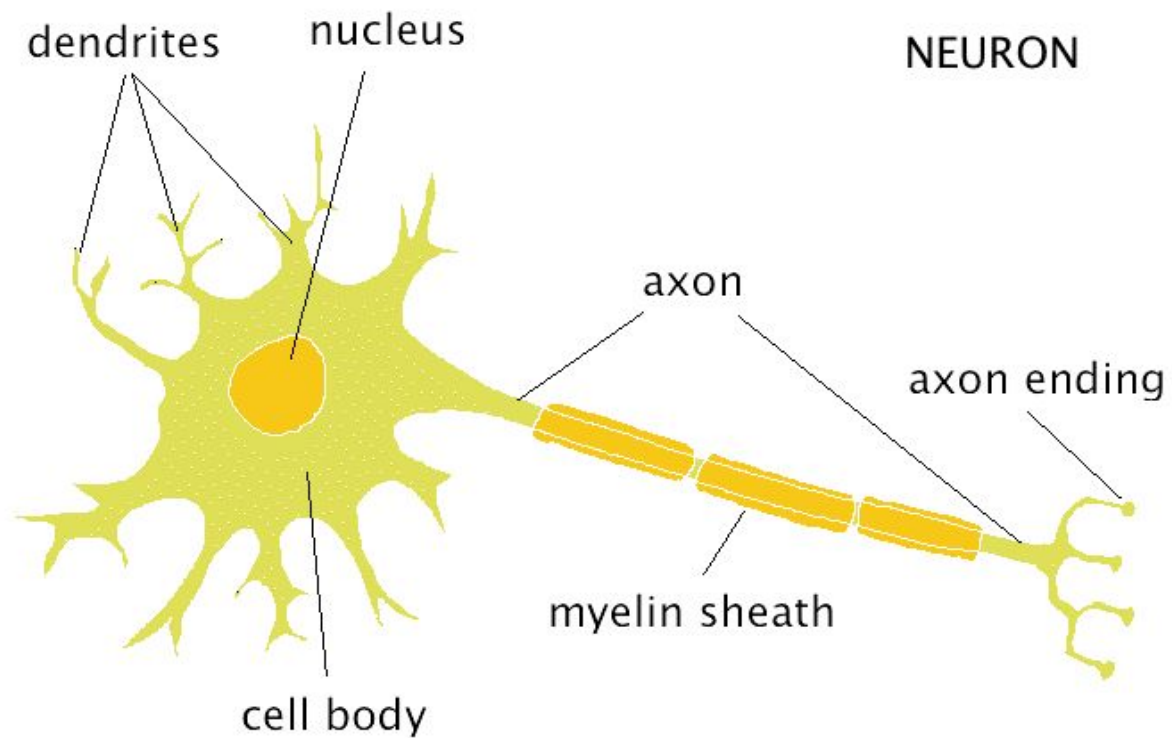
딥러닝의 개념과 배경

인경 신경망 알고리즘의 기본 개념

알파고나 머신러닝에서 많이 언급되는 알고리즘은 단연 딥러닝이다.

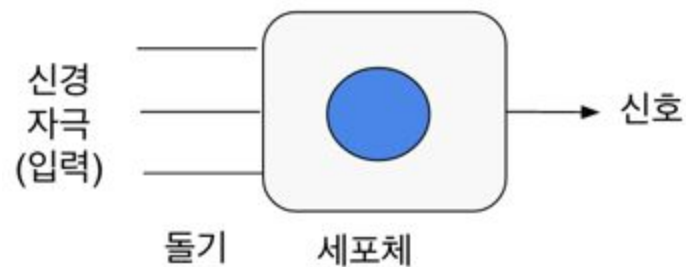
이 딥러닝은 머신러닝의 하나의 종류로 인공 신경망 알고리즘의 새로운 이름이다.

인공 신경망은 사람의 두뇌가 여러개의 뉴론으로 연결되서 복잡한 연산을 수행한다는데서 영감을 받아서, 머신러닝의 연산을 여러개의 간단한 노드를 뉴론 처럼 상호 연결해서 복잡한 연산을 하겠다는 아이디어이다.



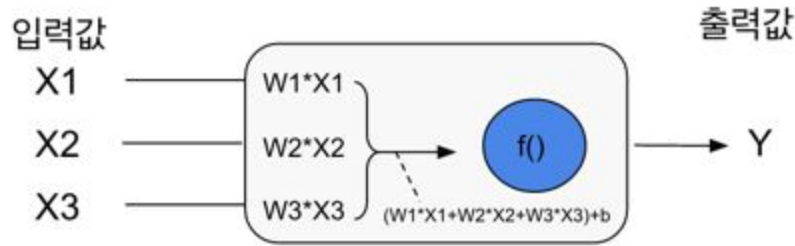
<출처 : <http://webspace.ship.edu/cgboer/theneuron.html> >

이 뉴런의 구조를 조금 더 단순하게 표현해보면 다음과 같은 모양이 된다.



뉴런은 돌기를 통해서 여러 신경 자극 (예를 들어 피부에서 촉각)을 입력 받고, 이를 세포체가 인지하여 신호로 변환해준다. 즉 신경 자극을 입력 받아서 신호라는 결과로 변환해주는 과정을 거치는데,

이를 컴퓨터로 형상화 해보면 다음과 같은 형태가 된다.



뉴런의 돌기처럼 외부에서 입력값 X_1, X_2, X_3 를 읽어드리고, 이 입력값들은 돌기를 거치면서 인식되어 각각 $W_1 \cdot X_1, W_2 \cdot X_2, W_3 \cdot X_3$ 로 변환이 되어 세포체에 도착하여 여러 돌기에서 들어온 값은 $(W_1 \cdot X_1 + W_2 \cdot X_2 + W_3 \cdot X_3) + b$ 값으로 취합된다.

이렇게 취합된 값은 세포체내에서 인지를 위해서 어떤 함수 $f(x)$ 를 거치게 되고, 이 값이 일정 값을 넘게 되면, Y 에 1이라는 신호를 주고, 일정값을 넘지 않으면 0이라는 값을 준다.

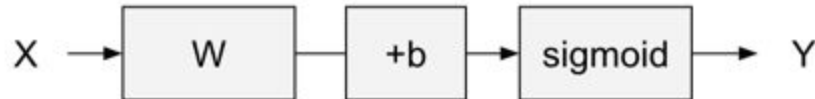
즉 뉴런을 본떠서 입력값 $X_1 \dots n$ 에 대해서, 출력값 Y 가 0 또는 1이 되는 알고리즘을 만든것이다.

Perceptron

이를 수식을 사용하여 한번 더 단순화를 시켜보면

X 를 행렬이라고 하고, $X = [X_1, X_2, X_3]$ 라고 하자.

그리고 역시 이에 대응되는 행렬 W 를 정의하고 $W = [W_1, W_2, W_3]$ 라고 하면



<뉴런을 본떠서 만든 Perceptron>

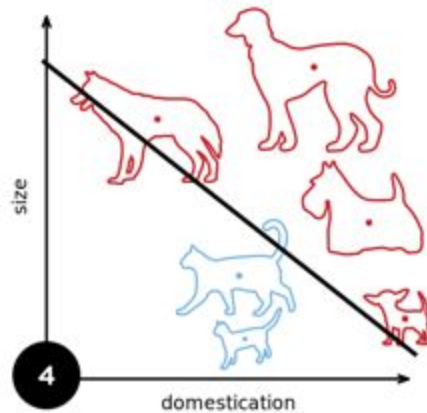
입력 X 를 받아서 W 를 곱한 후에, 함수 $f(x)$ 를 거쳐서 0 또는 1의 결과를 내는 Y 를 낸다.

즉 입력 X 를 받아서 참(1)인지 거짓(0) 인지를 판별해주는 계산 유닛을 Perceptron이라고 한다.

이 Perceptron은 결국 $W \cdot X + b$ 인 선을 그려서 이 선을 기준으로 1 또는 0을 판단하는 알고리즘이다.

예를 들어서 동물의 크기 (X_1)와 동물의 복종도 (X_2)라는 값을 가지고, 개인지 고양이인지를 구별하는 Perceptron이 있을때,

$W \cdot X + b$ 로 그래프를 그려보면 ($X = [X_1, X_2]$, $W = [W_1, W_2]$) 다음과 같은 직선이 되고, 이 직선 위부분이면 개, 아랫 부분이면 고양이 식으로 분류가 가능하다.



이 Perceptron은 입력에 따라서 Y를 1,0으로 분류해주는 알고리즘으로 앞에서 설명한 로지스틱 회귀 알고리즘을 사용할 수 있는데, 이때 로지스틱 회귀에서 사용한 함수 $f(x)$ 는 sigmoid 함수를 사용하였기 때문에, 여기서는 $f(x)$ 를 이 sigmoid 함수를 사용했다. 이 함수 $f(x)$ 를 Activation function이라고 한다. 이 Activation function은 중요하니 반드시 기억해놓기 바란다.

(참고. 손쉬운 이해를 위해서 로지스틱 회귀와 유사하게 sigmoid 함수를 사용했지만, sigmoid 함수이외에 다양한 함수를 Activation 함수로 사용할 수 있으며, 요즘은 sigmoid 함수의 정확도가 다른 Activation function에 비해 떨어지기 때문에, ReLu와 같은 다른 Activation function을 사용한다. 이 Activation function)에 대해서는 나중에 설명하겠다.)

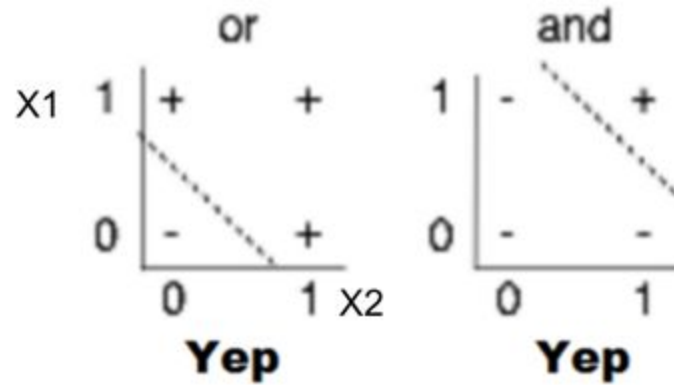
Perceptron의 XOR 문제

그런데 이 Perceptron는 결정적인 문제를 가지고 있는데, 직선을 그려서 AND, OR 문제를 해결할 수는 있지만, XOR 문제를 풀어낼 수 가 없다는 것이다.

다음과 같은 Perceptron이 있을때



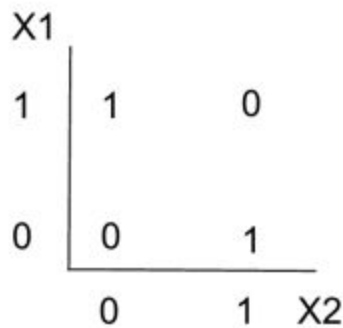
다음 그림 처럼 AND나 OR 문제는 직선을 그려서 해결이 가능하다.



그러나 다음과 같은 XOR 문제는 $WX+b$ 의 그래프로 해결이 가능할까?

		x_2	
		1	0
x_1	1	0	1
	0	1	0

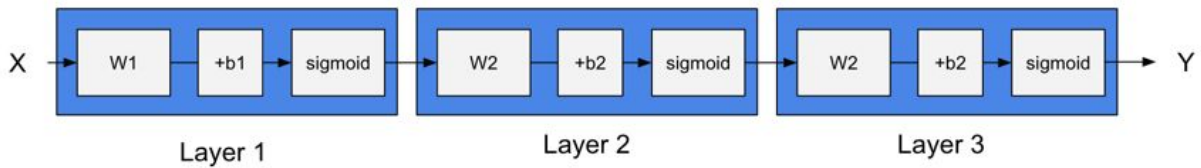
<그림 XOR 문제>



하나의 선을 긋는 Perceptron으로는 이 문제의 해결이 불가능하다.

MLP (Multi Layer Perceptron) 다중 계층 퍼셉트론의 등장

이렇게 단일 Perceptron으로 XOR 문제를 풀 수 없음을 증명되었는데, 1969년에 Marvin Minsky 교수가, 이 문제를 해결 하는 방법으로 Perceptron을 다중으로 겹치면 이 문제를 해결할 수 있음을 증명하였다.

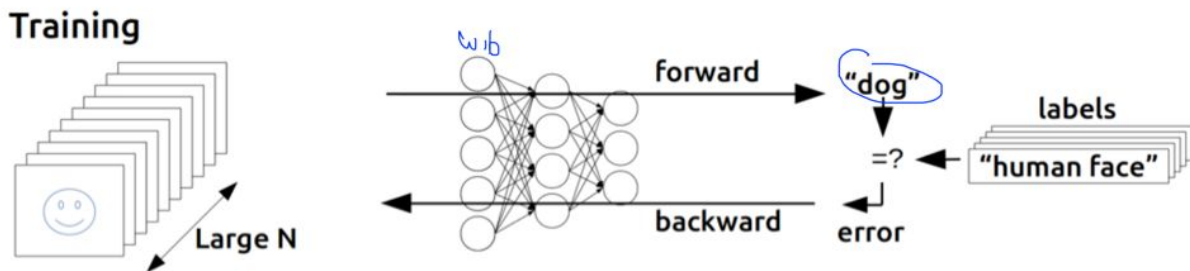


<그림 Multi Layer Perceptron의 개념도>

그런데, 이 MLP 역시 다른 문제를 가지고 있는데, MLP에서 학습을 통해서 구하고자 하는 것은 최적의 W 와 b 의 값을 찾아내는 것인데, 레이어가 복잡해질수록, 연산이 복잡해져서 현실적으로 이 W 와 b 의 값을 구하는 것이 불가능하다는 것을 Marvin Minsky 교수가 증명하였다.

Back Propagation 을 이용한 MLP 문제 해결

이런 문제를 해결하기 위해서 Back propagation이라는 알고리즘이 도입되었는데, 기본 개념은 뉴럴 네트워크를 순방향으로 한번 연산을 한 다음에, 그 결과 값을 가지고, 뉴럴 네트워크를 역방향 (backward)로 계산하면서 값을 구한다는 개념이다.



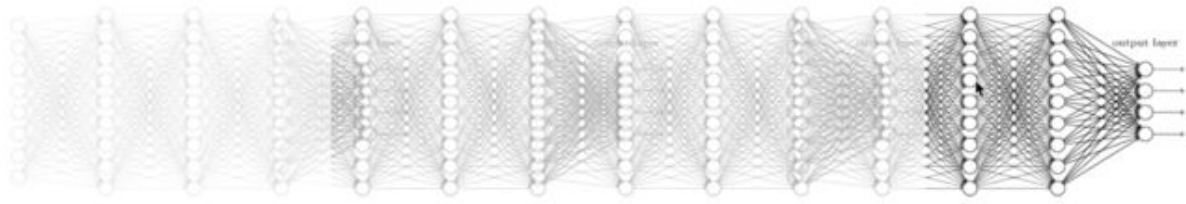
Backpropagation의 개념에 대해서는 다음글에서 자세하게 설명하도록 한다.

Back Propagation 문제와 ReLu를 이용한 해결

그러나 이 Back Propagation 역시 문제를 가지고 있었는데, 뉴럴 네트워크가 깊어질수록 Backpropagation이 제대로 안된다.

즉 순방향(foward)한 결과를 역방향(backward)로 반영하면서 계산을 해야 하는데, 레이어가 깊을수록 뒤에 있는 값이 앞으로 전달이 되지 않는 문제이다. 이를 Vanishing Gradient 문제라고 하는데, 그림으로 개념을 표현해보면 다음과 같다.

뒤에서 계산한 값이 앞의 레이어로 전달이 잘 되지 않는 것을 표현하기 위해서 흐리게 네트워크를 표현하였다.



이는 ReLu라는 activation function (앞에서는 sigmoid 함수를 사용했다.)으로 해결이 되었다.

뉴럴 네트워크의 초기값 문제

이 문제를 캐나다 CIFAR 연구소의 Hinton 교수님이 “뉴럴네트워크는 학습을 할때 초기값을 잘 주면 학습이 가능하다” 라는 것을 증명하면서 깊은 레이어를 가진 뉴럴 네트워크의 사용이 가능하게 된다.

이때 소개된 알고리즘이 초기값을 계산할 수 있는 RBM (Restricted Boltzmann Machine)이라는 알고리즘으로 이 알고리즘을 적용한 뉴럴 네트워크는 특히 머신러닝 알고리즘을 테스트 하는 ImageNet에서 CNN (Convolutional Neural Network)가 독보적인 성능을 내면서 뉴럴 네트워크가 주목 받기 시작하였다.

딥러닝

딥러닝이라는 어원은 새로운 알고리즘이나 개념을 이야기 하는 것이 아니고, 뉴럴 네트워크가 새롭게 주목을 받기 시작하면서 Hinton 교수님 등이 뉴럴네트워크에 대한 리브랜딩의 의미로 뉴럴 네트워크를 새로운 이름 “딥러닝”으로 부르기 시작하면서 시작 되었다.

추가

뉴럴네트워크와 딥러닝의 대략적인 개념과 역사에 대해서 알아보았다.

이 글에서는 뉴럴 네트워크에 대한 대략적인 개념만을 설명하고 있는데, 주로 언급되는 단어를 중심으로 기억하기를 바란다.

- Perceptron
- MLP (Multi Layer Perceptron)
- Back propagation
- ReLu
- RBM

이외에도, Drop Out, Mini Batch, Ensemble 과 같은 개념이 있는데, 이 개념은 추후에 다시 설명하고, 딥러닝에서 이미지 인식에 많이 사용되는 CNN (Convolutional Neural Network)을 나중에 소개하도록 하겠다.

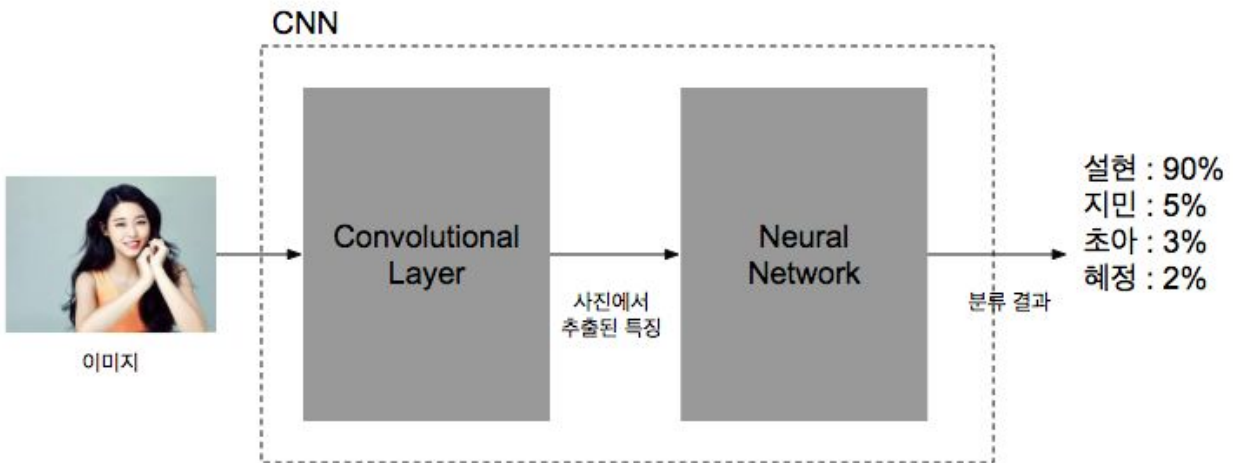
컨볼루셔널 네트워크

이번 글에서는 딥러닝 중에서 이미지 인식에 많이 사용되는 컨볼루셔널 뉴럴 네트워크 (Convolutional neural network) 이하 CNN에 대해서 알아보도록 하자.

이 글을 읽기에 앞서서 머신러닝에 대한 기본 개념이 없는 경우는 다음 글들을 참고하기 바란다.

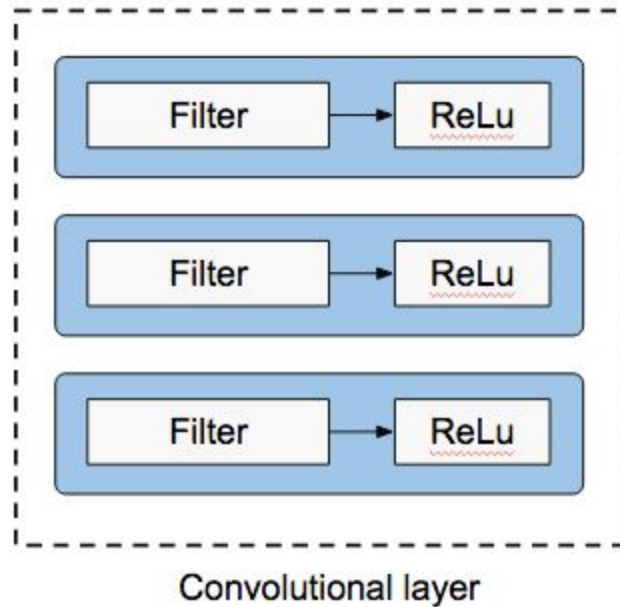
- 머신러닝의 개요 <http://bcho.tistory.com/1140>
- 머신러닝의 기본 원리는 <http://bcho.tistory.com/1139>
- 이산 분류의 원리에 대해서는 <http://bcho.tistory.com/1142>
- 인공 신경망에 대한 개념은 <http://bcho.tistory.com/1147>

CNN은 전통적인 뉴럴 네트워크 앞에 여러 계층의 컨볼루셔널 계층을 붙인 모양이 되는데, 그 이유는 다음과 같다. CNN은 앞의 컨볼루셔널 계층을 통해서 입력 받은 이미지에 대한 특징(Feature)를 추출하게 되고, 이렇게 추출된 특징을 기반으로 기존의 뉴럴 네트워크를 이용하여 분류를 해내게 된다.



컨볼루셔널 레이어 (Convolutional Layer)

컨볼루셔널 레이어는 앞에서 설명 했듯이 입력 데이터로 부터 특징을 추출하는 역할을 한다. 컨볼루셔널 레이어는 특징을 추출하는 기능을 하는 필터(Filter)와, 이 필터의 값을 비선형 값으로 바꾸어 주는 액티베이션 함수(Activation 함수)로 이루어진다. 그럼 각 부분의 개념과 원리에 대해서 살펴보도록 하자.



<그림 Filter와 Activation 함수로 이루어진 Convolutional 계층>

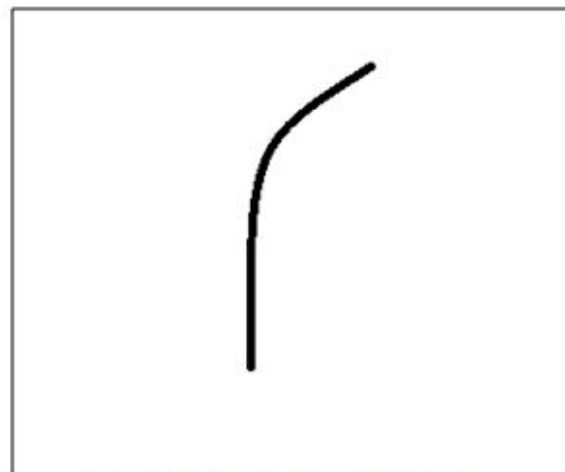
필터 (Filter)

필터 개념 이해

필터는 그 특징이 데이터에 있는지 없는지를 검출해주는 함수이다. 예를 들어 아래와 같이 곡선을 검출해주는 필터가 있다고 하자.

0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

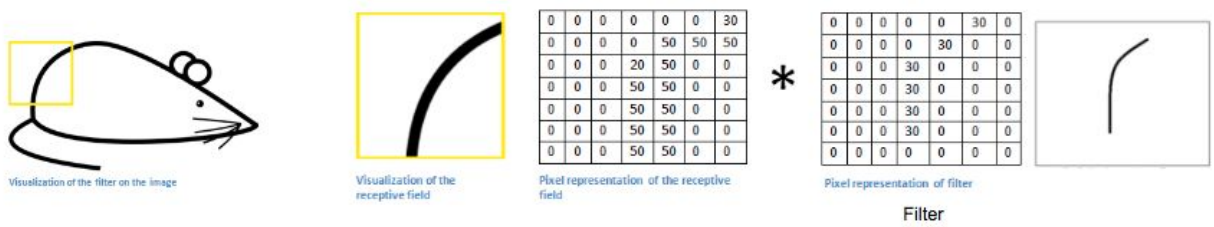
Pixel representation of filter



Visualization of a curve detector filter

필터는 구현에서는 위의 그림 좌측 처럼 행렬로 정의가 된다.
입력 받은 이미지 역시 행렬로 변환이 되는데, 아래 그림을 보자.

위 그림에서 좌측 상단의 이미지 부분을 잘라내서 필터를 적용하는 결과이다.

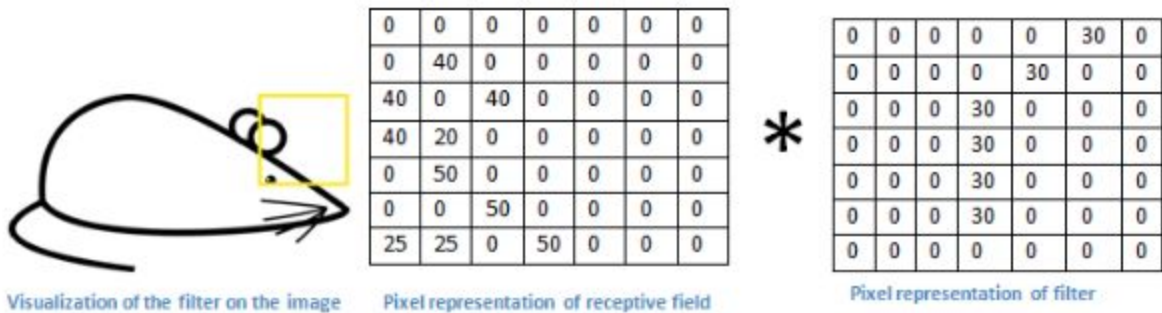


잘라낸 이미지와, 필터를 곱하면

Multiplication and Summation = $(50 \times 30) + (50 \times 30) + (50 \times 30) + (20 \times 30) + (50 \times 30) = 6600$ (A large number!)

과 같이 결과 값이 매우 큰 값이 나온다.

만약에 아래 그림처럼 쥐 그림에서 곡선이 없는 부분에 같은 필터를 적용해보면



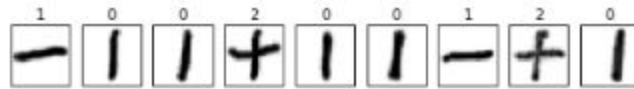
결과 값이 0에 수렴하게 나온다.

즉 필터는 입력받은 데이터에서 그 특성을 가지고 있으면 결과 값이 큰 값이 나오고, 특성을 가지고 있지 않으면 결과 값이 0에 가까운 값이 나오게 되어서 데이터가 그 특성을 가지고 있는지 여부를 알 수 있게 해준다.

다중 필터의 적용

입력값에는 여러가지 특징이 있기 때문에 하나의 필터가 아닌 여러개의 다중 필터를 같이 적용하게 된다.

다음과 같이 |, +, - 모양을 가지고 있는 데이터가 있다고 하자



각 데이터가 |와 -의 패턴(특징을) 가지고 있는지를 파악하기 위해서 먼저 | (세로) 필터를 적용해보면 다음과 같은 결과가 나온다.

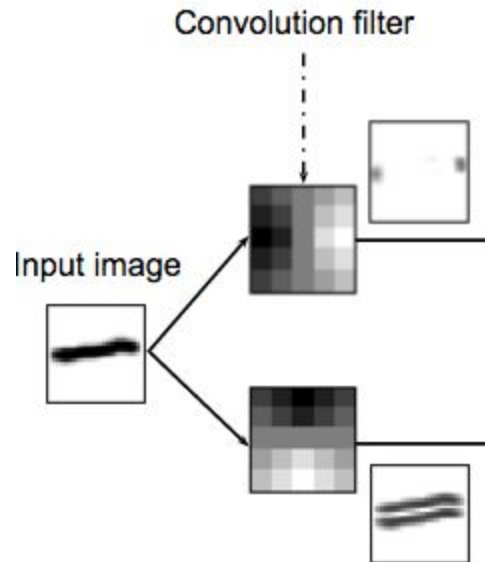


(맨앞의 상자는 필터이다.) 두번째 상자부터 원본 이미지에 세로선(|)이 없는 경우 결과 이미지에 출력이 없고, 세로선이 있는 경우에는 결과 이미지에 세로 선이 있는 것을 확인할 수 있다.

마찬가지로 가로선(-) 특징이 있는지 가로 선을 추출하는 필터를 적용해보면 다음과 같은 결과를 얻을 수 있다.



이렇게 각기 다른 특징을 추출하는 필터를 조합하여 네트워크에 적용하면, 원본 데이터가 어떤 형태의 특징을 가지고 있는지 없는지를 판단해 낼 수 있다. 다음은 하나의 입력 데이터에 앞서 적용한 세로와 가로선에 대한 필터를 동시에 적용한 네트워크의 모양이다.



Stride

그러면 이 필터를 어떻게 원본 이미지에 적용할까? 큰 사진 전체에 하나의 큰 필터 하나만을 적용할까?

아래 그림을 보자, 5x5 원본 이미지가 있을때, 3x3인 필터를 좌측 상단에서 부터 왼쪽으로 한칸씩 그 다음 한줄을 내려서 또 왼쪽으로 한칸씩 적용해서 특징을 추출해낸다.

오른쪽 Convolved Feature 행렬이 바로 원본 이미지에 3x3 필터를 적용하여 얻어낸 결과 이다.

1 _{x1}	1 _{x0}	1 _{x1}	0	0
0 _{x0}	1 _{x1}	1 _{x0}	1	0
0 _{x1}	0 _{x0}	1 _{x1}	1	1
0	0	1	1	0
0	1	1	0	0

Image

4		

Convolved
Feature

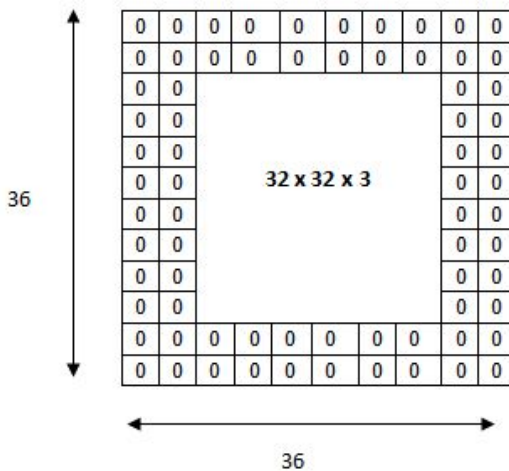
이렇게 필터를 적용 하는 간격 (여기서는 우측으로 한칸씩 그리고 아래로 한칸씩 적용하였다.) 값을 Stride라고 하고, 필터를 적용해서 얻어낸 결과를 Feature map 또는 activation map 이라고 한다.

Padding

앞에서 원본 데이터에 필터를 적용한 내용을 보면 필터를 적용한 후의 결과값은 필터 적용전 보다 작아졌다. 5x5 원본 이미지가 3x3의 1 stride 값을 가지고 적용되었을때, 결과 값은 3x3으로 크기가 작아졌다.

그런데, CNN 네트워크는 하나의 필터 레이어가 아니라 여러 단계에 걸쳐서 계속 필터를 연속적으로 적용하여 특징을 추출하는 것을 최적화 해나가는데, 필터 적용 후 결과 값이 작아지게 되면 처음에 비해서 특징이 많이 유실 될 수 가 있다. 필터를 거쳐감에 따라서 특징이 유실되는 것을 기대했다면 문제가 없겠지만, 아직까지 충분히 특징이 추출되기 전에, 결과 값이 작아지면 특징이 유실된다. 이를 방지 하기 위한 방법으로 padding 이라는 기법이 있는데, padding은 결과 값이 작아지는 것을 방지하기 위해서 입력값 주위로 0 값을 넣어서 입력 값의 크기를 인위적으로 키워서, 결과값이 작아지는 것을 방지 하는 기법이다.

다음 그림을 보자, 32x32x3 입력값이 있을때, 5x5x3 필터를 적용 시키면 결과값 (feature map)의 크기는 28x28x3 이 된다. 이렇게 사이즈가 작아지는 것을 원하지 않았다면 padding을 적용하는데, input 계층 주위로 0을 둘러 싸서, 결과 값이 작아지고 (피쳐가 소실 되는것)을 막는다 32x32x3 입력값 주위로 2 두께로 0을 둘러싸주면 36x36x3 이 되고 5x5x3 필터 적용하더라도, 결과값 은 32x32x3으로 유지된다.



The input volume is $32 \times 32 \times 3$. If we imagine two borders of zeros around the volume, this gives us a $36 \times 36 \times 3$ volume. Then, when we apply our conv layer with our three $5 \times 5 \times 3$ filters and a stride of 1, then we will also get a $32 \times 32 \times 3$ output volume.

< 그림, 32x32x3 데이터에 폭이 2인 padding을 적용한 예 >

패딩은 결과 값을 작아지는 것을 막아서 특징이 유실되는 것을 막는 것 뿐 아니라, 오버피팅도 방지하게 되는데, 원본 데이터에 0 값을 넣어서 원래의 특징을 희석 시켜 버리고, 이것을 기반으로 머신러닝 모델이 트레이닝 값에만 정확하게 맞아 들어가는 오버피팅 현상을 방지한다.

오버 피팅에 대해서는 별도의 다른 글을 통해서 설명한다.

필터는 어떻게 만드는 것일까?

그렇다면 CNN에서 사용되는 이런 필터는 어떻게 만드는 것일까? CNN의 신박한 기능이 바로 여기에 있는데, 이 필터는 데이터를 넣고 학습을 시키면, 자동으로 학습 데이터에서 학습을 통해서 특징을 인식하고 필터를 만들어 낸다.

Activation function

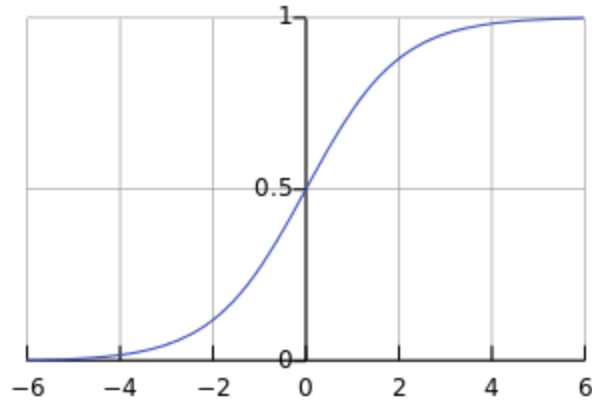
필터들을 통해서 Feature map이 추출되었으면, 이 Feature map에 Activation function을 적용하게 된다.

Activation function의 개념을 설명하면, 위의 주 그림에서 곡선값의 특징이 들어가 있는지 안들어가는지의 필터를 통해서 추출한 값이 들어가 있는 예에서는 6000, 안 들어가는 예에서는 0으로 나왔다.

이 값이 정량적인 값으로 나오기 때문에, 그 특징이 “있다 없다”의 비선형 값으로 바꿔 주는 과정이 필요한데, 이 것이 바로 Activation 함수이다.

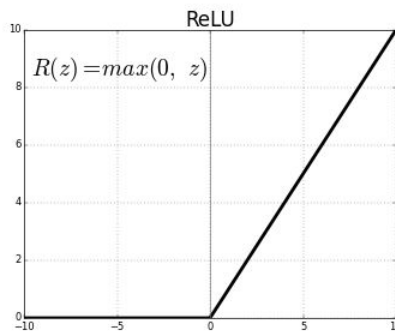
예전에 로지스틱 회귀 (<http://bcho.tistory.com/1142>)에서 설명하였던 시그모이드(sigmoid) 함수가 이 Activation 함수에 해당한다.

간단하게 짚고 넘어가면, 결과 값을 참/거짓으로 나타내는 것이 아니라, 참에 가까워면 0.5~1사이에서 1에 가까운 값을 거짓에 가까우면 0~0.5 사이의 값으로 리턴하는 것이다.



<그림. Sigmoid 함수>

뉴럴 네트워크나 CNN (CNN도 뉴럴 네트워크이다.) 이 Activation 함수로 이 sigmoid 함수는 잘 사용하지 않고, 아래 그림과 같은 ReLu 함수를 주요 사용한다.



<그림. ReLu 함수>

이 함수를 이용하는 이유는 뉴럴 네트워크에서 신경망이 깊어질 수록 학습이 어렵기 때문에, 전체 레이어를 한번 계산한 후, 그 계산 값을 재 활용하여 다시 계산하는 Back propagation이라는 방법을 사용하는데, sigmoid 함수를 activation 함수로 사용할 경우, 레이어가 깊어지면 이 Back propagation이 제대로 작동을 하지 않기 때문에, (값을 뒤에서 앞으로 전달할때 희석이 되는 현상. 이를 Gradient Vanishing 이라고 한다.) ReLu라는 함수를 사용한다.

풀링 (Sub sampling or Pooling)

이렇게 컨볼루셔널 레이어를 거쳐서 추출된 특징들은 필요에 따라서 서브 샘플링 (sub sampling)이라는 과정을 거친다.

컨볼루셔널 계층을 통해서 어느정도 특징이 추출 되었으면, 이 모든 특징을 가지고 판단을 할 필요가 없다.

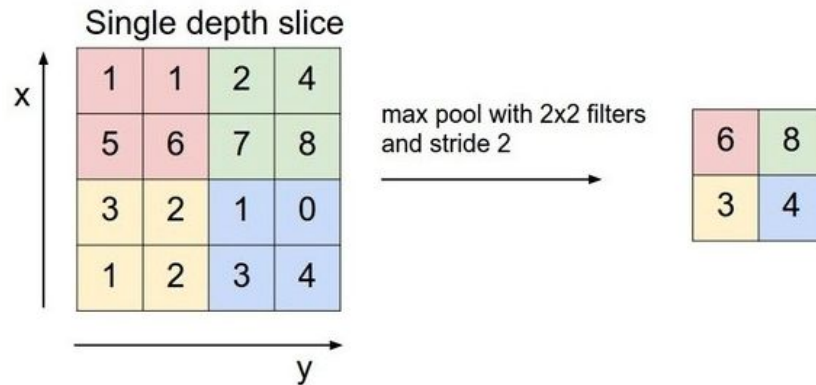
쉽게 예를 들면, 우리가 고해상도 사진을 보고 물체를 판별할 수 있지만, 작은 사진을 가지고도 그 사진의 내용이 어떤 사진인지 판단할 수 있는 원리이다.

그래서, 추출된 Activation map을 인위로 줄이는 작업을 하는데, 이 작업을 sub sampling 또는 pooling 이라고 한다. Sub sampling은 여러가지 방법이 있는데, max pooling, average pooling, L2-norm pooling 등이 있고, 그중에서 max pooling 이라는 기법이 많이 사용된다.

Max pooling (맥스 풀링)

맥스 풀링은 Activation map을 $M \times N$ 의 크기로 잘라낸 후, 그 안에서 가장 큰 값을 뽑아내는 방법이다.

아래 그림을 보면 4×4 Activation map에서 2×2 맥스 풀링 필터를 stride를 2로 하여 2칸씩 이동하면서 맥스 풀링을 한 예인데, 좌측 상단에서는 6이 가장 큰 값이기 때문에 6을 뽑아내고, 우측 상단에는 2,4,7,8 중 8이 가장 크기 때문에 8을 뽑아 내었다.



맥스 풀링은 특징의 값이 큰 값이 다른 특징들을 대표한다는 개념을 기반으로 하고 있다. (주의 풀링은 액티베이션 함수마다 매번 적용하는 것이 아니라, 데이터의 크기를 줄이고 싶을 때 선택적으로 사용하는 것이다.)

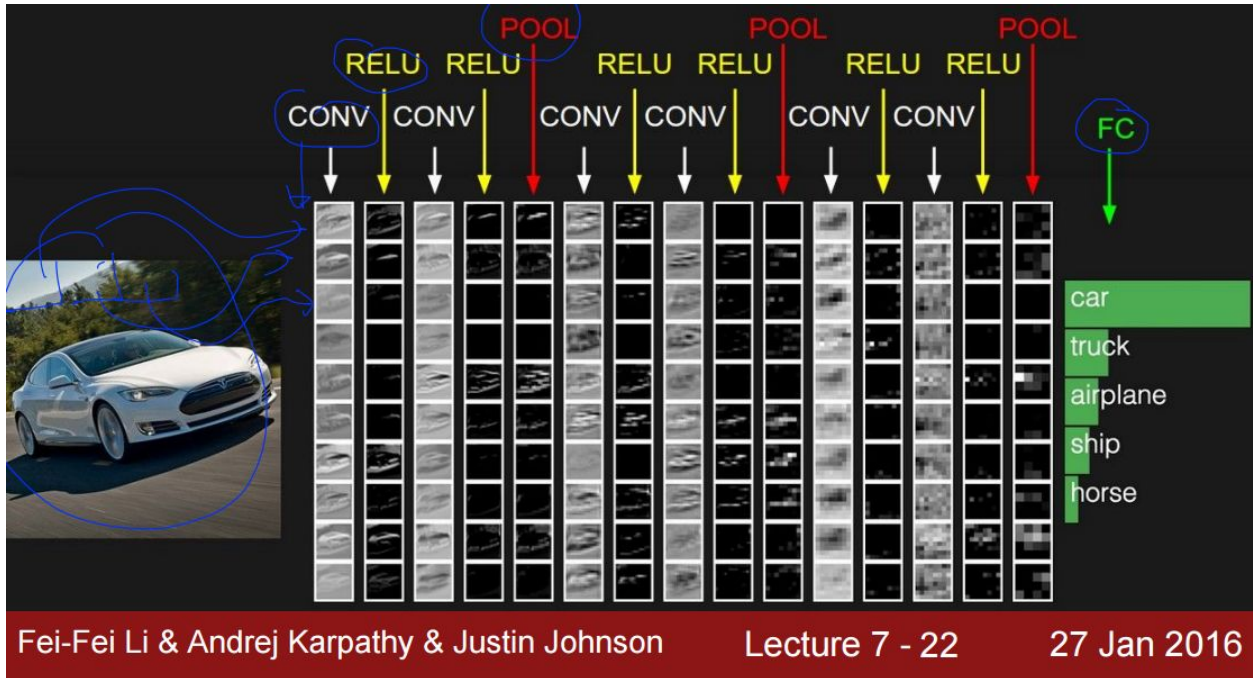
이런 sampling 을 통해서 얻을 수 있는 장점은 다음과 같다.

- 전체 데이터의 사이즈가 줄어들기 때문에 연산에 들어가는 컴퓨팅 리소스가 적어지고
- 데이터의 크기를 줄이면서 소실이 발생하기 때문에, 오버피팅을 방지할 수 있다.

컨볼루셔널 레이어

이렇게 컨볼루셔널 필터와 액티베이션 함수 (ReLU) 그리고 풀링 레이어를 반복적으로 조합하여 특징을 추출한다.

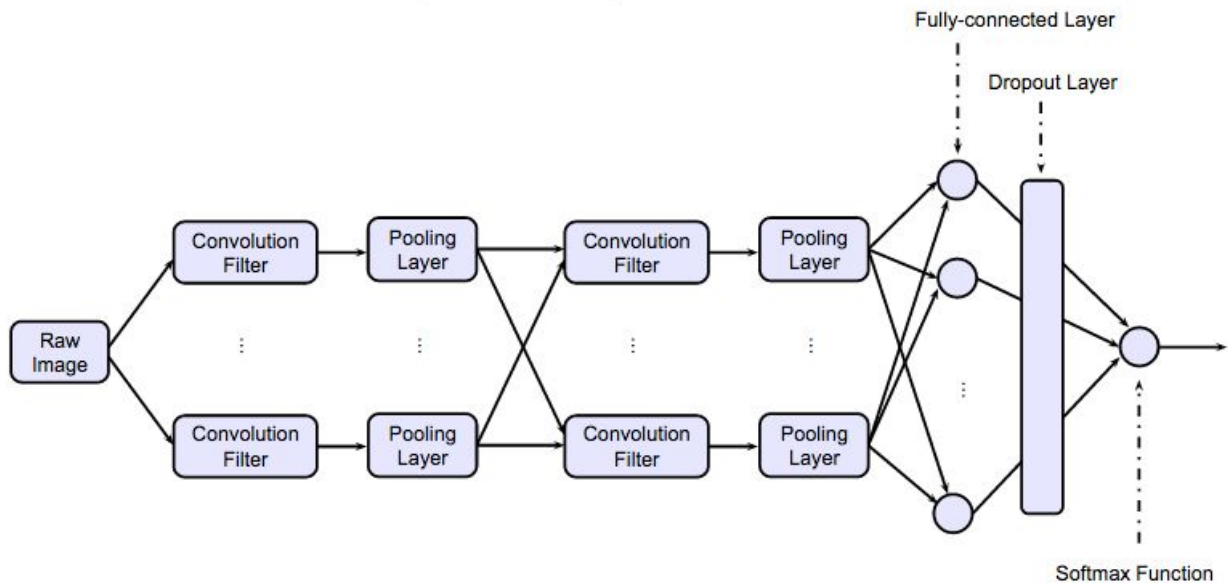
아래 그림을 보면 여러개의 컨볼루셔널 필터(CONV)와 액티베이션 함수 (ReLU)와 풀링 (POOL) 사용된 것을 볼 수 있다.



Fully connected Layer

컨볼루션 계층에서 특징이 추출이 되었으면 이 추출된 특징 값을 기존의 뉴럴 네트워크 (인공 신경 망)에 넣어서 분류를 한다.

그래서 CNN의 최종 네트워크 모양은 다음과 같이 된다.

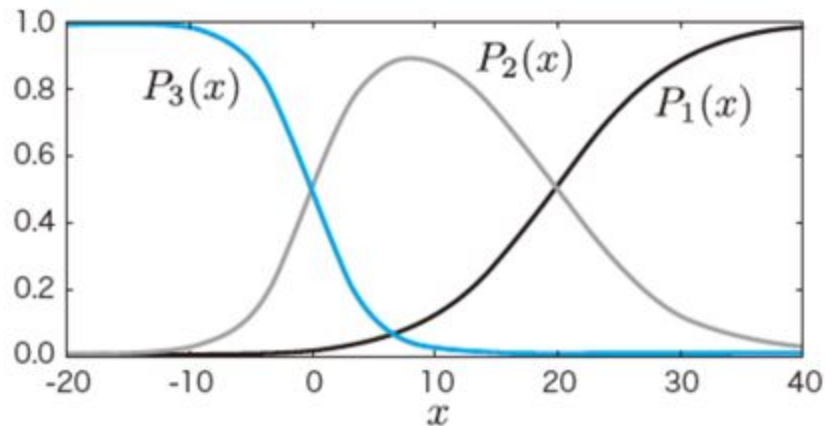


<그림. CNN 네트워크의 모양>

Softmax 함수

Fully connected network (일반적인 뉴럴 네트워크)에 대해서는 이미 알고 있겠지만, 위의 그림에서 Softmax 함수가 가장 마지막에 표현되었기 때문에, 다시 한번 짚고 넘어가자. Softmax도 앞에서 언급한 sigmoid나 ReLu와 같은 액티베이션 함수의 일종이다.

Sigmoid 함수가 이산 분류 (결과값에 따라 참 또는 거짓을 나타내는) 함수라면, Softmax 는 여러개의 분류를 가질 수 있는 함수이다. 아래 그림이 Softmax 함수의 그림이다.



이것이 의미하는 바는 다음과 같다. $P_3(x)$ 는 특징(feature) x 에 대해서 P_3 일 확률, $P_1(x)$ 는 특징 x 에 대해서 P_1 인 확률이다.

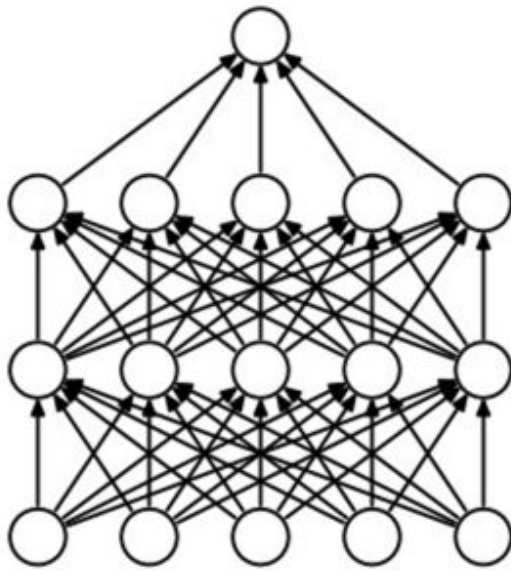
P_n 값은 항상 0~1.0의 범위를 가지며, $P_1+P_2+...+P_n = 1$ 이 된다.

예를 들어서 사람을 넣었을때, 설현일 확률 0.9, 지현인 확률 0.1 식으로 표시가 되는 것이다.

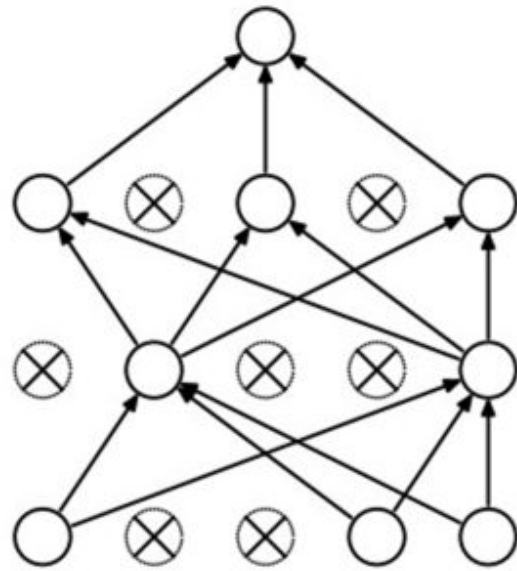
Dropout 계층

위 CNN 그래프에서 특이한 점중 하나는 Fully connected 네트워크와 Softmax 함수 중간에 Dropout layer (드롭아웃) 라는 계층이 있는 것을 볼 수 있다.

드롭 아웃은 오버피팅(over-fit)을 막기 위한 방법으로 뉴럴 네트워크가 학습중일때, 랜덤하게 뉴런을 꺼서 학습을 방해함으로써, 학습이 학습용 데이터에 치우치는 현상을 막아준다.



(a) Standard Neural Net



(b) After applying dropout.

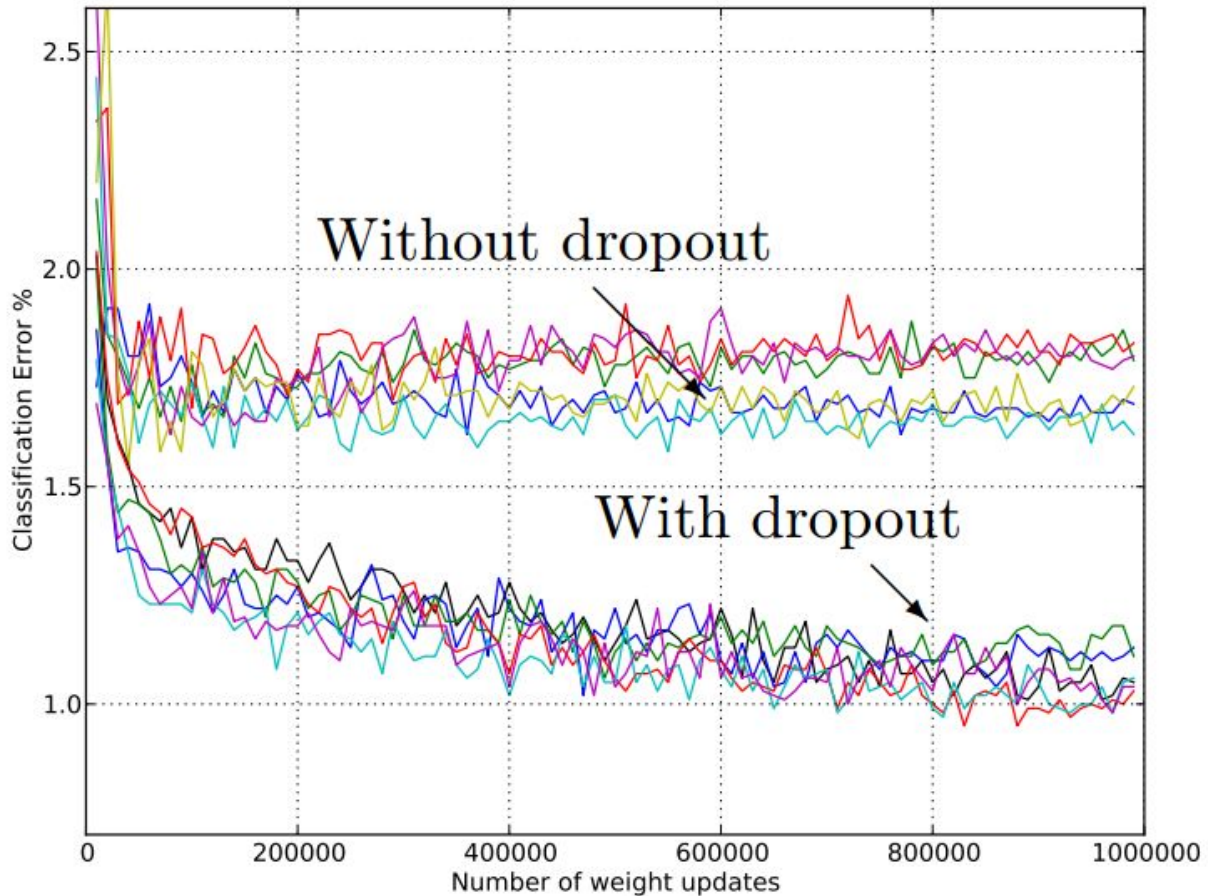
<그림. 드롭 아웃을 적용한 네트워크 >

그림 출처 :

https://leonardoaraujosantos.gitbooks.io/artificial-intelligence/content/dropout_layer.html

일반적으로 CNN에서는 이 드롭아웃 레이어를 Fully connected network 뒤에 놓지만, 상황에 따라서는 max pooling 계층 뒤에 놓기도 한다.

다음은 드롭아웃을 적용하고 학습시킨 모델과 드롭 아웃을 적용하지 않은 모델 사이의 예측 정확도를 비교한 결과 이다.



<그림. 드롭아웃을 적용한 경우와 적용하지 않고 학습한 경우, 에러율의 차이 >

이렇게 복잡한데 어떻게 구현을 하나요?

대략적인 개념은 이해를 했다. 그렇다면 구현을 어떻게 해야 할까? 앞에서 설명을 할때, softmax 나 뉴런에 대한 세부 알고리즘 ReLu 등과 같은 알고리즘에 대한 수학적 공식을 설명하지 않았다. 그렇다면 이것 하나하나 공부해야 할까?

아니다. 작년에 구글에서 머신러닝용 프로그래밍 프레임워크로 텐서 플로우라는 것을 발표했다. 이 텐서 플로는 (<http://www.tensorflow.org>)는 이런 머신 러닝에 특화된 프레임워크로, 머신러닝에 필요한 대부분의 함수들을 이미 구현하여 제공한다.

실제로 CNN을 구현한 코드를 보자. 이 코드는 홍콩 과학기술 대학교의 김성훈 교수님의 강의를 김성훈님이란 분이 텐서 플로우 코드로 구현하여 공유해놓은 코드중 CNN 구현 예제이다.

https://github.com/FuZer/Study_TensorFlow/blob/master/08%20-%20CNN/CNN.py

```

l1a = tf.nn.relu(tf.nn.conv2d(X, w,                                # l1a shape=(?, 28, 28, 32)
                        strides=[1, 1, 1, 1], padding='SAME'))
l1 = tf.nn.max_pool(l1a, ksize=[1, 2, 2, 1],                    # l1 shape=(?, 14, 14, 32)
                    strides=[1, 2, 2, 1], padding='SAME')
l1 = tf.nn.dropout(l1, p_keep_conv)

```

첫번째 줄을 보면, `tf.nn.conv2d` 라는 함수를 사용하였는데, 이 함수는 컨볼루션 필터를 적용한 함수이다. 처음 `X`는 입력값이며, 두번째 `w` 값은 필터 값을 각각 행렬로 정의한다. 그 다음 `strides` 값을 정의해주고, 마지막으로 `padding` 인자를 통해서 `padding` 사이즈를 정한다. 컨볼루션 필터를 적용한 후 액티베이션 함수로 `tf.nn.relu`를 이용하여 ReLu 함수를 적용한 것을 볼 수 있다.

다음으로는 `tf.nn.max_pool` 함수를 이용하여, max pooling을 적용하고 마지막으로 `tf.nn.dropout` 함수를 이용하여 dropout을 적용하였다.

전문적인 수학 지식이 없이도, 이미 잘 추상화된 텐서플로우 함수를 이용하면, 기본적인 개념만 가지고도 머신러닝 알고리즘 구현이 가능하다.

텐서 플로우를 공부하는 방법은 여러가지가 있겠지만, 유튜브에서 이찬우님이 강의 하고 계신 텐서 플로우 강의를 듣는 것을 추천한다. 한글이고 설명이 매우 쉽다. 그리고 매주 일요일에 생방송을 하는데, 궁금한것도 물어볼 수 있다.

https://www.youtube.com/channel/UCRyIQSBvSybbaNY_JCyg_vA

그리고 텐서플로우 사이트의 튜토리얼도 상당히 잘되어 있는데,

<https://www.tensorflow.org/versions/r0.12/tutorials/index.html> 를 보면 되고 한글화도 잘 진행되고 있다. 한글화된 문서는 <https://tensorflowkorea.gitbooks.io/tensorflow-kr/content/> 에서 찾을 수 있다.

구현은 할 수 있겠는데, 그러면 이 모델은 어떻게 만드나요?

그럼 텐서플로우를 이용하여 모델을 구현할 수 있다는 것은 알았는데, 그렇다면 모델은 어떻게 만들까? 정확도를 높이려면 수십 계층의 뉴럴 네트워크를 설계해야 하고, max pooling 함수의 위치와 padding 등 여러가지를 고려해야 하는데, 과연 이게 가능할까?

물론 전문적인 지식을 가진 데이터 과학자라면 이런 모델을 직접 설계하고 구현하고 테스트 하는게 맞겠지만, 이런 모델들은 이미 다양한 모델이 만들어져서 공개 되어 있다.

그중에서 CNN 모델은 매년 이미지넷 (<http://www.image-net.org/>) 이라는데서 주최하는 ILSVRC (Large Scale Visual Recognition Competition) 이라는 대회에서, 주최측이 제시하는 그림을 누가 잘 인식하는지를 겨루는 대회이다.



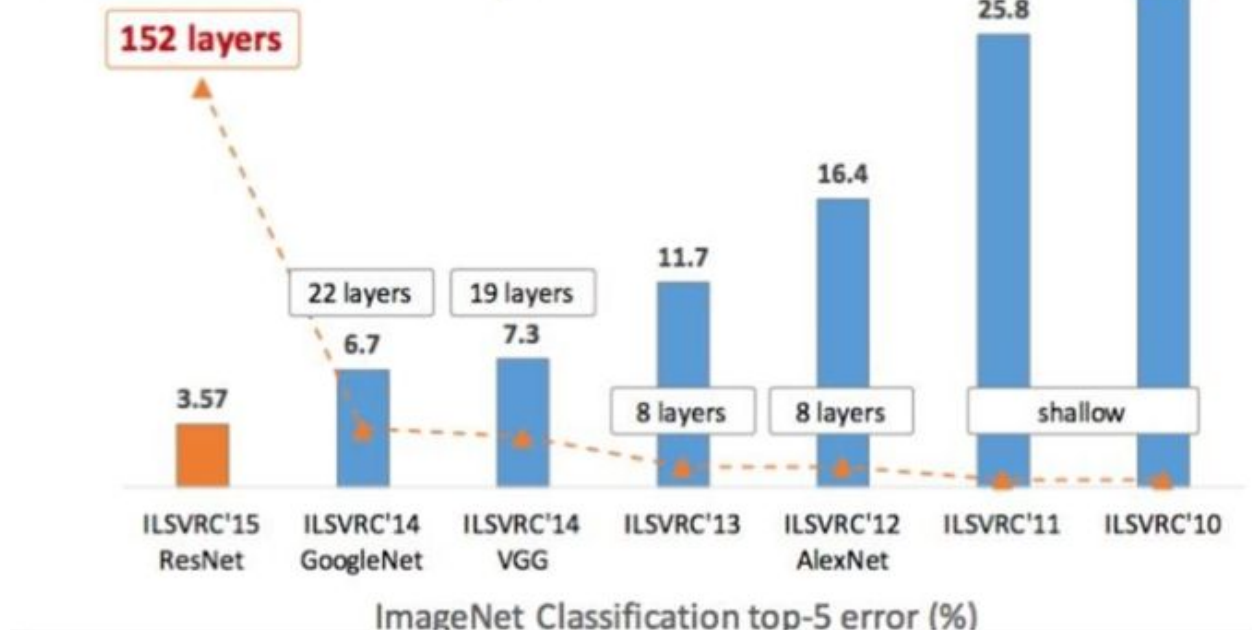
<그림. 이미지넷 대회에 사용되는 이미지들 일부>

이 대회에서는 천만장의 이미지를 학습하여, 15만장의 이미지를 인식하는 정답률을 겨루게 된다. 매년 알고리즘이 향상되는데, 딥러닝이 주목 받은 계기가 된 AlexNet은 12년도 우승으로, 8개의 계층으로 16.4%의 에러율을 내었고, 14년에는 19개 계층을 가진 VGG 알고리즘이 7.3%의 오차율을 기록하였고, 14년에는 구글넷이 22개의 레이어로 6.7%의 오차율을 기록하였다. 그리고 최근에는 마이크로소프트의 152개의 레이어로 ResNet이 3.57%의 오차율을 기록하였다.

(참고로 인간의 평균 오류율은 5% 내외이다.)

현재는 ResNet을 가장 많이 참고해서 사용하고 있고, 쉽게 사용하려면 VGG 모델을 사용하고 있다.

Revolution of Depth



결론

머신러닝과 딥러닝에 대해서 공부를 하면서 이게 더이상 수학자나 과학자만의 영역이 아니라 개발자도 들어갈 수 있는 영역이라는 것을 알 수 있었고, 많은 딥러닝과 머신러닝 강의가 복잡한 수학 공식으로 설명이 되지만, 이건 아무래도 설명하는 사람이 수학쪽에 배경을 두고 있기 때문 일것이고, 요즘은 텐서플로우 프레임워크를 사용하면 복잡한 수학적 지식이 없이 기본적인 머신러닝에 대한 이해만을 가지고도 머신러닝 알고리즘을 개발 및 서비스에 적용이 가능한 시대가 되었다고 본다.

그림 출처 및 참고 문서

- 김성훈 교수님의 모두를 위한 딥러닝 <https://hunkim.github.io/ml/>
- A Beginner's guide to understanding convolutional network <https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks/>

7. 컨볼루셔널 네트워크를 이용한 MNIST 의 다항 분류 모델 구현

딥러닝을 이용한 숫자 이미지 인식 #1/2

지난 글(<http://bcho.tistory.com/1154>) 을 통해서 소프트맥스 회귀를 통해서, 숫자를 인식하는 모델을 만들어서 학습 시켜 봤다.

이번글에서는 소프트맥스보다 정확성이 높은 컨볼루셔널 네트워크를 이용해서 숫자 이미지를 인식하는 모델을 만들어 보겠다.

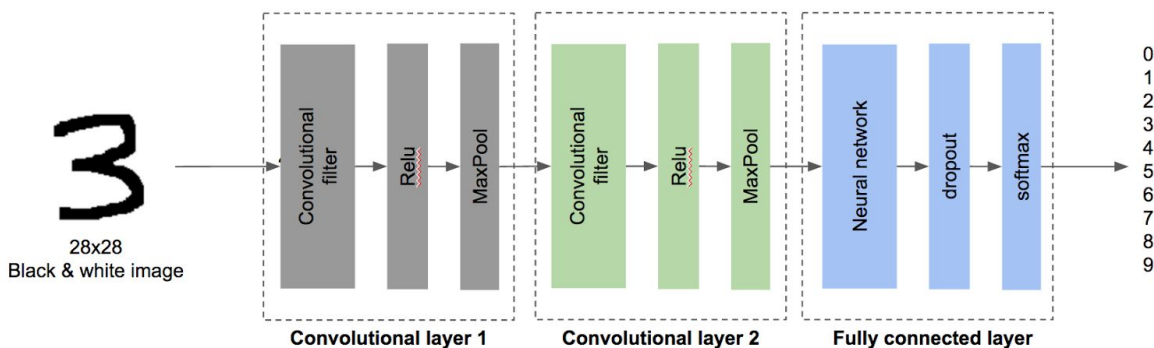
이 글의 목적은 CNN 자체의 설명이나, 수학적 이론에 대한 이해가 목적이 아니다. 최소한의 수학적 지식만 가지고, CNN 네트워크 모델을 텐서플로우로 구현하는데에 그 목적을 둔다. CNN을 이해하기 위해서는 Softmax 등의 함수를 이해하는게 좋기 때문에 가급적이면 <http://bcho.tistory.com/1154> 예제를 먼저 보고 이 문서를 보는게 좋다. 그 다음에 CNN 모델에 대한 개념적인 이해를 위해서 <http://bcho.tistory.com/1149> 문서를 참고하고 이 문서를 보는 것이 좋다.

이번 글은 CNN을 적용하는 것 이외에, 다음과 같은 몇가지 팁을 추가로 소개한다.

- 학습이 된 모델을 저장하고 다시 로딩 하는 방법
- 학습된 모델을 이용하여 실제로 주피터 노트북에서 글씨를 써보고 인식하는 방법

MNIST CNN 모델

우리가 만들고자 하는 모델은 두개의 컨볼루셔널 레이어(Convolutional layer)과, 마지막에 풀리 커넥티드 레이어 (fully connected layer)을 가지고 있는 컨볼루셔널 네트워크 모델(CNN) 이다. 모델의 모양을 그려보면 다음과 같다.



입력 데이터

입력으로 사용되는 데이터는 앞의 소프트맥스 예제에서 사용한 데이터와 동일한 손으로 쓴 숫자들이다. 각 숫자 이미지는 28x28 픽셀로 되어 있고, 흑백이미지이기 때문에 데이터는 28x28x1 행렬이 된다. (만약에 칼라 RGB라면 28x28x3이 된다.)

컨볼루션 계층

총 두 개의 컨볼루션 계층을 사용했으며, 각 계층에서 컨볼루션 필터를 사용해서, 특징을 추출한다음에, 액티베이션 함수 (Activation function)으로, ReLu를 적용한 후, 맥스풀링 (Max Pooling)을 이용하여, 주요 특징을 정리해낸다.

이와 같은 컨볼루션 필터를 두개를 중첩하여 적용하였다.

마지막 풀리 커넥티드 계층

컨볼루션 필터를 통해서 추출된 특징은 풀리 커넥티드 레이어(Fully connected layer)에 의해서 분류 되는데, 풀리 커넥티드 레이어는 하나의 뉴럴 네트워크를 사용하고, 그 뒤에 드롭아웃 (Dropout) 계층을 넣어서, 오버피팅(Overfitting)이 발생하는 것을 방지한다. 마지막으로 소프트맥스 (Softmax) 함수를 이용하여 0~9 열개의 숫자로 분류를 한다.

학습(트레이닝) 코드

이를 구현하기 위한 코드는 다음과 같다.

코드

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.examples.tutorials.mnist import input_data

tf.reset_default_graph()

np.random.seed(20160704)
tf.set_random_seed(20160704)

# load data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)

# define first layer
num_filters1 = 32

x = tf.placeholder(tf.float32, [None, 784])
x_image = tf.reshape(x, [-1,28,28,1])

W_conv1 = tf.Variable(tf.truncated_normal([5,5,1,num_filters1],
                                          stddev=0.1))
h_conv1 = tf.nn.conv2d(x_image, W_conv1,
                      strides=[1,1,1,1], padding='SAME')
```



```

b_conv1 = tf.Variable(tf.constant(0.1, shape=[num_filters1]))
h_conv1_cutoff = tf.nn.relu(h_conv1 + b_conv1)

h_pool1 = tf.nn.max_pool(h_conv1_cutoff, ksize=[1,2,2,1],
                        strides=[1,2,2,1], padding='SAME')

# define second layer
num_filters2 = 64

W_conv2 = tf.Variable(
    tf.truncated_normal([5,5,num_filters1,num_filters2],
                        stddev=0.1))
h_conv2 = tf.nn.conv2d(h_pool1, W_conv2,
                        strides=[1,1,1,1], padding='SAME')

b_conv2 = tf.Variable(tf.constant(0.1, shape=[num_filters2]))
h_conv2_cutoff = tf.nn.relu(h_conv2 + b_conv2)

h_pool2 = tf.nn.max_pool(h_conv2_cutoff, ksize=[1,2,2,1],
                        strides=[1,2,2,1], padding='SAME')

# define fully connected layer
h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*num_filters2])

num_units1 = 7*7*num_filters2
num_units2 = 1024

w2 = tf.Variable(tf.truncated_normal([num_units1, num_units2]))
b2 = tf.Variable(tf.constant(0.1, shape=[num_units2]))
hidden2 = tf.nn.relu(tf.matmul(h_pool2_flat, w2) + b2)

keep_prob = tf.placeholder(tf.float32)
hidden2_drop = tf.nn.dropout(hidden2, keep_prob)

w0 = tf.Variable(tf.zeros([num_units2, 10]))
b0 = tf.Variable(tf.zeros([10]))
k = tf.matmul(hidden2_drop, w0) + b0
p = tf.nn.softmax(k)

#define loss (cost) function
t = tf.placeholder(tf.float32, [None, 10])
loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(k,t))
train_step = tf.train.AdamOptimizer(0.0001).minimize(loss)
correct_prediction = tf.equal(tf.argmax(p, 1), tf.argmax(t, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

# prepare session
sess = tf.InteractiveSession()
sess.run(tf.global_variables_initializer())
saver = tf.train.Saver()

# start training
i = 0
for _ in range(1000):
    i += 1
    batch_xs, batch_ts = mnist.train.next_batch(50)
    sess.run(train_step,
              feed_dict={x:batch_xs, t:batch_ts, keep_prob:0.5})
    if i % 500 == 0:
        loss_vals, acc_vals = [], []

```

```

for c in range(4):
    start = len(mnist.test.labels) / 4 * c
    end = len(mnist.test.labels) / 4 * (c+1)
    loss_val, acc_val = sess.run([loss, accuracy],
        feed_dict={x:mnist.test.images[start:end],
                    t:mnist.test.labels[start:end],
                    keep_prob:1.0})
    loss_vals.append(loss_val)
    acc_vals.append(acc_val)
loss_val = np.sum(loss_vals)
acc_val = np.mean(acc_vals)
print ('Step: %d, Loss: %f, Accuracy: %f'
        % (i, loss_val, acc_val))

saver.save(sess, 'cnn_session')
sess.close()

```

데이터 로딩 파트

그러면 코드를 하나씩 살펴보도록 하자.

맨 처음 블록은 데이터를 로딩하고 각종 변수를 초기화 하는 부분이다.

```

import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.examples.tutorials.mnist import input_data

```

#Call `tf.reset_default_graph()` before you build your model (and the Saver). This will ensure that the variables get the names you intended, but it will invalidate previously-created graphs.

```
tf.reset_default_graph()
```

```

np.random.seed(20160704)
tf.set_random_seed(20160704)

```

```

# load data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)

```

`input_data` 는 텐서플로에 내장되어 있는 MNIST (손으로 쓴 숫자 데이터)셋으로, `read_data_sets` 메서드를 이용하여 데이터를 읽었다. 데이터 로딩 부분은 앞의 소프트맥스 MNIST와 같으니 참고하기 바란다.

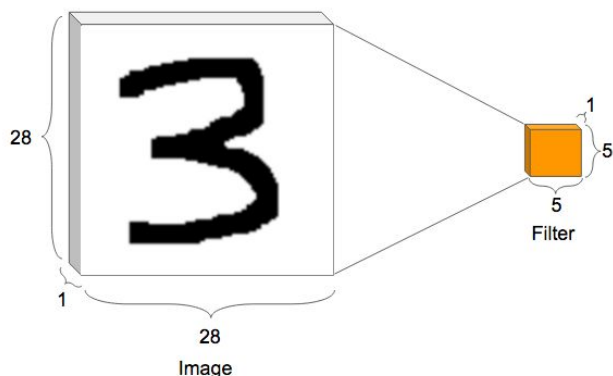
여기서 특히 주목해야 할 부분은 `tf.reset_default_graph()` 인데, 주피터 노트북과 같은 환경에서 실행을 하게 되면, 주피터 커널을 리스타트하지 않는 이상 변수들의 컨텍스트가 그대로 유지 되기 때문에, 위의 코드를 같은 커널에서 `tf.reset_default_graph()` 없이, 두 번 이상 실행하게 되면 에러가 난다. 그 이유는 텐서플로우 그래프를 만들어놓고, 그 그래프가 지워지지 않은 상태에서 다시 같은 그래프를 생성하면서 나오는 에러인데, `tf.reset_default_graph()` 메서드는 기존에 생성된 디폴트 그래프를 모두 삭제해서 그래프가 중복되는 것을 막아준다. 일반적인 파이썬 코드에서는 크게 문제가 없지만, 컨텍스트가 계속 유지되는 주피터 노트북 같은 경우에는 발생할 수 있는 문제이니, 반드시 디폴트 그래프를 리셋해주도록 하자

첫번째 컨볼루셔널 계층

필터의 정의

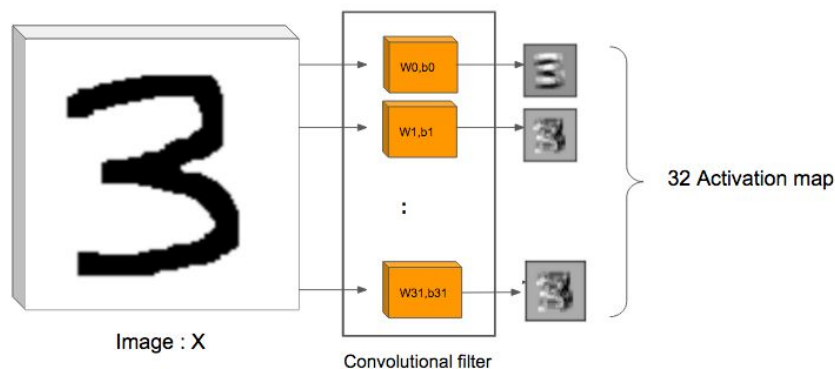
다음은 첫번째 컨볼루셔널 계층을 정의 한다. 컨볼루셔널 계층을 이해하려면 컨볼루셔널 필터에 대한 개념을 이해해야 하는데, 다시 한번 되짚어 보자.

컨볼루셔널 계층에서 하는 일은 입력 데이터에 필터를 적용하여, 특징을 추출해 낸다.



이 예제에서 입력 받는 이미지 데이터는 28x28x1 행렬로 표현된 흑백 숫자 이미지이고, 예제 코드에서는 5x5x1 사이즈의 필터를 적용한다.

5x5x1 사이즈의 필터 32개를 적용하여, 총 32개의 특징을 추출할것이다.



코드

필터 정의 부분까지 코드로 살펴보면 다음과 같다.

```
# define first layer  
num_filters1 = 32
```

```
x = tf.placeholder(tf.float32, [None, 784])  
x_image = tf.reshape(x, [-1,28,28,1])
```

```
W_conv1 = tf.Variable(tf.truncated_normal([5,5,1,num_filters1],
```

x는 입력되는 이미지 데이터로, 2차원 행렬(28x28)이 아니라, 1차원 벡터(784)로 되어 있고, 데이터의 수는 무제한으로 정의하지 않았다. 그래서 placeholder정의에서 shape이 [None,784] 로 정의 되어 있다.

예제에서는 연산을 편하게 하기 위해서 2차원 행렬을 사용할것이기 때문에, 784 1차원 벡터를 28x28x1 행렬로 변환을 해준다.

x_image는 784x무한개인 이미지 데이터 x를, (28x28x1)이미지의 무한개 행렬로 reshape를 이용하여 변경하였다. [-1,28,28,1]은 28x28x1 행렬을 무한개(-1)로 정의하였다.

필터를 정의하는데, 필터는 앞서 설명한것과 같이 5x5x1 필터를 사용할것이고, 필터의 수는 32개이기 때문에, 필터 W_conv1의 차원(shape)은 [5,5,1,32] 가된다. (코드에서 32는 num_filters1 이라는 변수에 저장하여 사용하였다.) 그리고 W_conv1의 초기값은 [5,5,1,32] 차원을 가지는 난수를 생성하도록 tf.truncated_normal을 사용해서 임의의 수가 지정되도록 하였다.

필터 적용

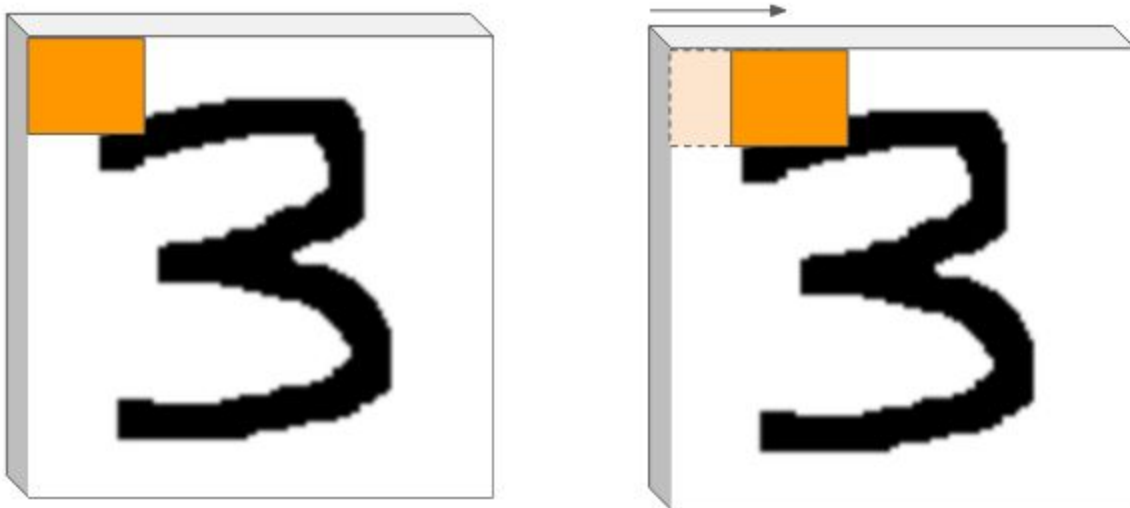
필터를 정의했으면 필터를 입력 데이터(이미지)에 적용한다.

```
h_conv1 = tf.nn.conv2d(x_image, W_conv1,
                      strides=[1,1,1,1], padding='SAME')
```

필터를 적용하는 방법은 tf.nn.conv2d를 이용하면 되는데, 28x28x1 사이즈의 입력 데이터인 x_image에 앞에서 정의한 필터 W_conv1을 적용하였다.

스트라이드 (Strides)

필터는 이미지의 좌측 상단 부터 아래 그림과 같이 일정한 간격으로 이동하면서 적용된다.



이를 개념적으로 표현하면 다음과 같은 모양이 된다.

1	1	1	0	0
0	1	1	1	0
0	0	1 _{x1}	1 _{x0}	1 _{x1}
0	0	1 _{x0}	1 _{x1}	0 _{x0}
0	1	1 _{x1}	0 _{x0}	0 _{x1}

Image

4	3	4
2	4	3
2	3	4

Convolved
Feature

이렇게 필터를 움직이는 간격을 스트라이드 (Stride)라고 한다.

예제에서는 우측으로 한칸 그리고 끝까지 이동하면 아래로 한칸을 이동하도록 각각 가로와 세로의 스트라이드 값을 1로 세팅하였다.

코드에서 보면

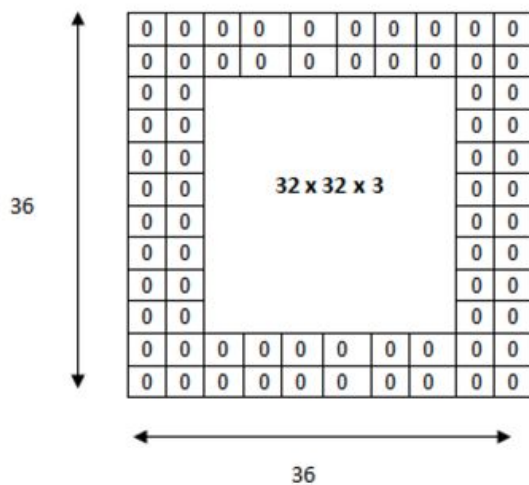
```
h_conv1 = tf.nn.conv2d(x_image, W_conv1,
                        strides=[1,1,1,1], padding='SAME')
```

에서 `strides=[1,1,1,1]` 로 정의한것을 볼 수 있다. 맨앞과 맨뒤는 통상적으로 1을 쓰고, 두번째 1은 가로 스트라이드 값, 그리고 세번째 1은 세로 스트라이드 값이 된다.

패딩 (Padding)

위의 그림과 같이 필터를 적용하여 추출된 특징 행렬은 원래 입력된 이미지 보다 작게 된다.

연속해서 필터를 이런 방식으로 적용하다 보면 필터링 된 특징들이 작아지게되는데, 만약에 특징을 다 추출하기 전에 특징들이 의도하지 않게 유실되는 것을 막기 위해서 패딩이라는 것을 사용한다.



The input volume is 32 x 32 x 3. If we imagine two borders of zeros around the volume, this gives us a 36 x 36 x 3 volume. Then, when we apply our conv layer with our three 5 x 5 x 3 filters and a stride of 1, then we will also get a 32 x 32 x 3 output volume.

패딩이란, 입력된 데이터 행렬 주위로, 무의미한 값을 감싸서 원본 데이터의 크기를 크게 해서, 필터를 거치고 나온 특징 행렬의 크기가 작아지는 것을 방지한다. 또한 무의미한 값을 넣음으로써, 오버피팅이 발생하는 것을 방지할 수 있다. 코드상에서 padding 변수를 이용하여 패딩 방법을 정의하였다.

```
h_conv1 = tf.nn.conv2d(x_image, W_conv1,
                      strides=[1,1,1,1], padding='SAME')
```

padding='SAME'을 주게 되면, 텐서플로우가 자동으로 패딩을 삽입하여 입력값과 출력값 (특징 행렬)의 크기가 같도록 한다. padding='VALID'를 주게 되면, 패딩을 적용하지 않고 필터를 적용하여 출력값 (특징 행렬)의 크기가 작아진다.

활성함수 (Activation function)의 적용

필터 적용이 끝났으면, 이 필터링된 값에 활성함수를 적용한다. 컨볼루션 네트워크에서 일반적으로 사용하는 활성함수는 ReLU 함수이다.

코드

```
b_conv1 = tf.Variable(tf.constant(0.1, shape=[num_filters1]))
h_conv1_cutoff = tf.nn.relu(h_conv1 + b_conv1)
```

먼저 bias 값 ($y=WX+b$ 에서 b)인 b_conv1 을 정의하고, $tf.nn.relu$ 를 이용하여, 필터된 결과(h_conv1)에 bias 값을 더한 값을 ReLU 함수로 적용하였다.

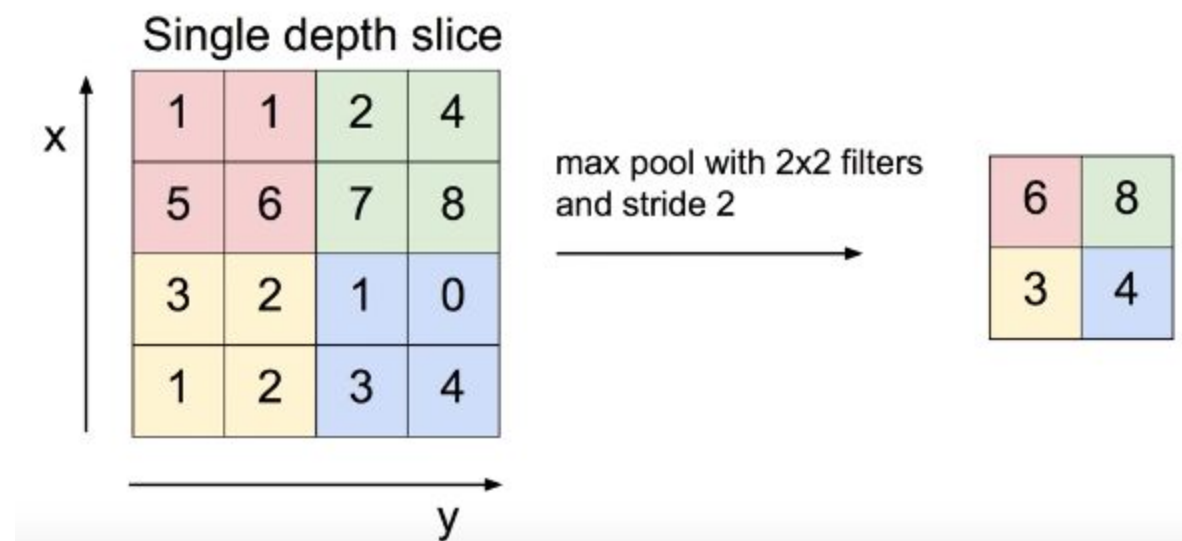
Max Pooling

추출된 특징 모두를 가지고 특징을 판단할 필요가 없이, 일부 특징만을 가지고도 특징을 판단할 수 있다. 즉 예를 들어서 고해상도의 큰 사진을 가지고도 어떤 물체를 식별할 수 있지만, 작은 사진을 가지고도 물체를 식별할 수 있다. 이렇게 특징의 수를 줄이는 방법을 서브샘플링 (sub sampling)이라고 하는데, 서브샘플링을 해서 전체 특징의 수를 의도적으로 줄이는 이유는

데이터의 크기를 줄이기 때문에, 컴퓨팅 파워를 절약할 수 있고, 데이터가 줄어드는 과정에서 데이터가 유실이 되기 때문에, 오버 피팅을 방지할 수 있다.

이러한 서브 샘플링에는 여러가지 방법이 있지만 예제에서는 맥스 풀링 (max pooling)이라는 방법을 사용했는데, 맥스 풀링은 풀링 사이즈 (mxn)로 입력데이터를 나눈후 그 중에서 가장 큰 값만을 대표값으로 추출하는 것이다.

아래 그림을 보면 원본 데이터에서 2x2 사이즈로 맥스 풀링을 해서 결과를 각 셀별로 최대값을 뽑아내었고, 이 셀을 가로 2칸씩 그리고 그다음에는 세로로 2칸씩 이동하는 stride 값을 적용하였다.



코드

```
h_pool1 = tf.nn.max_pool(h_conv1_cutoff, ksize=[1,2,2,1],
                        strides=[1,2,2,1], padding='SAME')
```

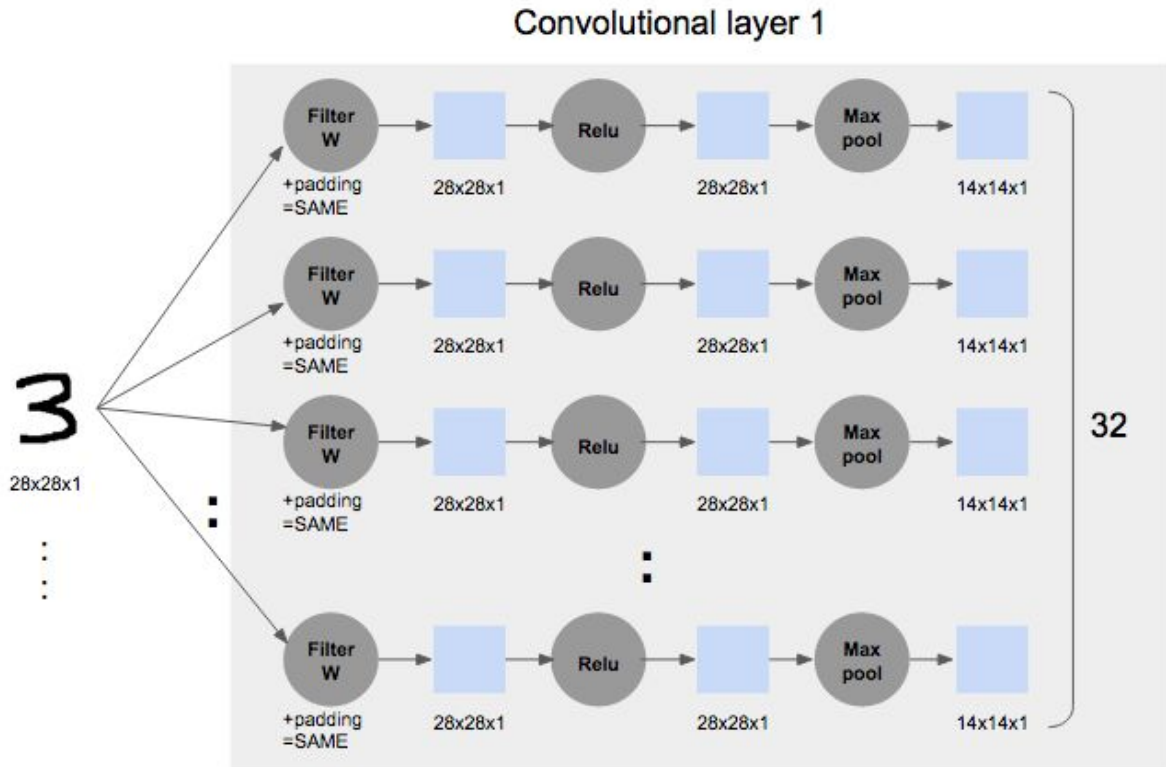
Max pooling은 tf.nn.max_pool이라는 함수를 이용해서 적용할 수 있는데, 첫번째 인자는 활성화 함수 ReLu를 적용하고 나온 결과 값인 h_conv1_cutoff 이고, 두 번째 인자인 ksize는 풀링 필터의 사이즈로 [1,2,2,1]은 2x2 크기로 묶어서 풀링을 한다는 의미이다.

다음 stride는 컨볼루션 필터 적용과 마찬가지로 풀링 필터를 가로와 세로로 얼마만큼씩 움직일 것인데, strides=[1,2,2,1]로, 가로로 2칸, 세로로 2칸씩 움직이도록 정의하였다.

행렬의 차원 변환

텐서플로우를 이용해서 CNN을 만들때 각각 개별의 알고리즘을 이해할 필요는 없지만 각 계층을 추가하거나 연결하기 위해서는 행렬의 차원이 어떻게 바뀌는지는 이해해야 한다.

다음 그림을 보자



첫번째 컨볼루셔널 계층은 위의 그림과 같이, 처음에 28x28x1의 이미지가 들어가면 32개의 컨볼루셔널 필터 W를 적용하게 되고, 각각은 28x28x1의 결과 행렬을 만들어낸다. 컨볼루셔널 필터를 거치게 되면 결과 행렬의 크기는 작아져야 정상이지만, 결과 행렬의 크기를 입력 행렬의 크기와 동일하게 유지하도록 padding='SAME'으로 설정하였다.

다음으로 bias 값 b를 더한 후 (위의 그림에는 생략하였다) 이 값에 액티베이션 함수 ReLu를 적용하고 나면 행렬 크기에 변화 없이 28x28x1 행렬 32개가 나온다. 이 각각의 행렬에 size가 2x2이고, stride가 2인 맥스풀링 필터를 적용하게 되면 각각의 행렬의 크기가 반으로 줄어들어 14x14x1 행렬 32개가 리턴된다.

두번째 컨볼루셔널 계층

이제 두번째 컨볼루셔널 계층을 살펴보자. 첫번째 컨볼루셔널 계층과 다를 것이 없다.

코드

```
# define second layer
num_filters2 = 64

W_conv2 = tf.Variable(
    tf.truncated_normal([5,5,num_filters1,num_filters2],
        stddev=0.1))
h_conv2 = tf.nn.conv2d(h_pool1, W_conv2,
    strides=[1,1,1,1], padding='SAME')
```



```

b_conv2 = tf.Variable(tf.constant(0.1, shape=[num_filters2]))
h_conv2_cutoff = tf.nn.relu(h_conv2 + b_conv2)

h_pool2 = tf.nn.max_pool(h_conv2_cutoff, ksize=[1,2,2,1],
                        strides=[1,2,2,1], padding='SAME')

```

단 필터값인 W_conv2의 차원이 [5,5,32,64] ([5,5,num_filters1,num_filters2] 부분)로 변경되었다.

```

W_conv2 = tf.Variable(
    tf.truncated_normal([5,5,num_filters1,num_filters2],
                        stddev=0.1))

```

필터의 사이즈가 5x5이고, 입력되는 값이 32개이기 때문에, 32가 들어가고, 총 64개의 필터를 적용하기 때문에 마지막 부분이 64가 된다.

첫번째 필터와 똑같이 stride를 1,1을 줘서 가로,세로로 각각 1씩 움직이고, padding='SAME'으로 입력과 출력 사이즈를 같게 하였다.

```

h_pool2 = tf.nn.max_pool(h_conv2_cutoff, ksize=[1,2,2,1],
                        strides=[1,2,2,1], padding='SAME')

```

맥스풀링 역시 첫번째 필터와 마찬가지로 2,2 사이즈의 필터(ksize=[1,2,2,1])를 적용하고 stride값을 2,2로 줘서 (strides=[1,2,2,1]) 가로 세로로 두칸씩 움직이게 하여 결과의 크기가 반으로 줄어들게 하였다.

14x14 크기의 입력값 32개가 들어가서, 7x7 크기의 행렬 64개가 리턴된다.

풀리 커넥티드 계층

두개의 컨볼루셔널 계층을 통해서 특징을 뽑아냈으면, 이 특징을 가지고 입력된 이미지가 0~9 중 어느 숫자인지를 풀리 커넥티드 계층 (Fully connected layer)를 통해서 판단한다.

코드

```

# define fully connected layer
h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*num_filters2])

num_units1 = 7*7*num_filters2
num_units2 = 1024

w2 = tf.Variable(tf.truncated_normal([num_units1, num_units2]))
b2 = tf.Variable(tf.constant(0.1, shape=[num_units2]))
hidden2 = tf.nn.relu(tf.matmul(h_pool2_flat, w2) + b2)

keep_prob = tf.placeholder(tf.float32)
hidden2_drop = tf.nn.dropout(hidden2, keep_prob)

w0 = tf.Variable(tf.zeros([num_units2, 10]))
b0 = tf.Variable(tf.zeros([10]))
k = tf.matmul(hidden2_drop, w0) + b0
p = tf.nn.softmax(k)

```

입력된 64개의 7x7 행렬을 1차원 행렬로 변환한다.

```
h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*num_filters2])
```

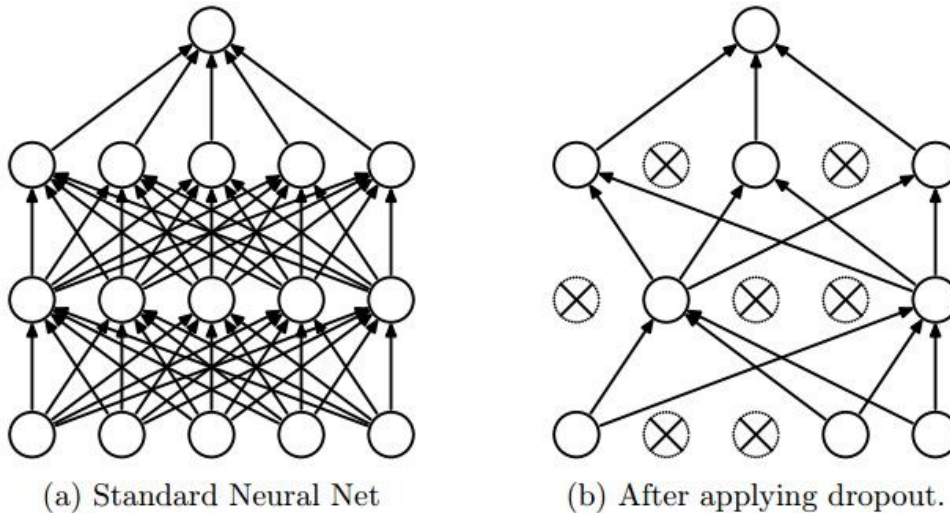
다음으로 풀리 커넥티드 레이어에 넣는데, 이때 입력값은 64x7x7 개의 벡터 값을 1024개의 뉴런을 이용하여 학습한다.

```
w2 = tf.Variable(tf.truncated_normal([num_units1, num_units2]))
b2 = tf.Variable(tf.constant(0.1, shape=[num_units2]))
```

그래서 w2의 값은 [num_units1,num_units2]로 num_units1은 64x7x7 로 입력값의 수를, num_unit2는 뉴런의 수를 나타낸다. 다음 아래와 같이 이 뉴런으로 계산을 한 후 액티베이션 함수 ReLu를 적용한다.

```
hidden2 = tf.nn.relu(tf.matmul(h_pool2_flat, w2) + b2)
```

다음 레이어에서는 드롭 아웃을 정의하는데, 드롭 아웃은 오버피팅(과적합)을 막기 위한 계층으로, 원리는 다음 그림과 같이 몇몇 노드간의 연결을 끊어서 학습된 데이터가 도달하지 않도록 하여서 오버피팅이 발생하는 것을 방지하는 기법이다.



출처 : <http://cs231n.github.io/neural-networks-2/>

텐서 플로우에서 드롭 아웃을 적용하는 것은 매우 간단하다. 아래 코드와 같이 tf.nn.dropout 이라는 함수를 이용하여, 앞의 네트워크에서 전달된 값 (hidden2)를 넣고 keep_prob에, 연결 비율을 넣으면 된다.

```
keep_prob = tf.placeholder(tf.float32)
hidden2_drop = tf.nn.dropout(hidden2, keep_prob)
```

연결 비율이란 네트워크가 전체가 다 연결되어 있으면 1.0, 만약에 50%를 드롭아웃 시키면 0.5 식으로 입력한다.

드롭 아웃이 끝난후에는 결과를 가지고 소프트맥스 함수를 이용하여 10개의 카테고리 분류한다.

```
w0 = tf.Variable(tf.zeros([num_units2, 10]))
b0 = tf.Variable(tf.zeros([10]))
k = tf.matmul(hidden2_drop, w0) + b0
p = tf.nn.softmax(k)
```

비용 함수 정의

여기까지 모델 정의가 끝났다. 이제 이 모델을 학습 시키기 위해서 비용함수(코스트 함수)를 정의해보자.

코스트 함수는 크로스엔트로피 함수를 이용한다.

```
#define loss (cost) function
t = tf.placeholder(tf.float32, [None, 10])
loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(k,t))
train_step = tf.train.AdamOptimizer(0.0001).minimize(loss)
```

k는 앞의 모델에 의해서 앞의 모델에서

```
k = tf.matmul(hidden2_drop, w0) + b0
p = tf.nn.softmax(k)
```

으로 softmax를 적용하기 전의 값이다. Tf.nn.softmax_cross_entropy_with_logits 는 softmax가 포함되어 있는 함수이기 때문에, p를 적용하게 되면 softmax 함수가 중첩 적용되기 때문에, softmax 적용전의 값인 k 를 넣었다.

WARNING: This op expects unscaled logits, since it performs a softmax on logits internally for efficiency. Do not call this op with the output of softmax, as it will produce incorrect results

https://github.com/tensorflow/tensorflow/blob/master/tensorflow/g3doc/api_docs/python/functions_and_classes/shard7/tf.nn.softmax_cross_entropy_with_logits.md

t는 플레이스 홀더로 정의하였는데, 나중에 학습 데이터 셋에서 읽을 라벨 (그 그림이 0..9 중 어느 숫자인지)이다.

그리고 이 비용 함수를 최적화 하기 위해서 최적화 함수 AdamOptimizer를 사용하였다. (앞의 소프트맥스 예제에서는 GradientOptimizer를 사용하였는데, 일반적으로 AdamOptimizer가 좀 더 무난하다.)

학습

이제 모델 정의와, 모델의 비용함수와 최적화 함수까지 다 정의하였다. 그러면 이 그래프들을 데이터를 넣어서 학습 시켜보자. 학습은 배치 트레이닝을 이용할것이다.

학습 도중 학습의 진행상황을 보기 위해서 학습된 모델을 중간중간 테스트할것이다. 테스트할때마다 학습의 정확도를 측정하여 출력하는데, 이를 위해서 정확도를 계산하는 함수를 아래와 같이 정의한다.

```
#define validation function
correct_prediction = tf.equal(tf.argmax(p, 1), tf.argmax(t, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

correct_prediction은 학습 결과와 입력된 라벨(정답)을 비교하여 맞았는지 틀렸는지를 리턴한다.

argmax는 인자에서 가장 큰 값의 인덱스를 리턴하는데, 0~9 배열이 들어가 있기 때문에 가장 큰 값이 학습에 의해 예측된 숫자이다. p는 예측에 의한 결과 값이고, t는 라벨 값이다 이 두 값을 비교하여 가장 큰 값이 있는 인덱스가 일치하면 예측이 성공한것이다.

correct_prediction은 bool 값이기 때문에, 이 값을 숫자로 바꾸기 위해서 tf.reduce_mean을 사용하여, accuracy에 저장하였다.

이제 학습을 세션을 시작하고, 변수들을 초기화 한다.

```
# prepare session
sess = tf.InteractiveSession()
sess.run(tf.global_variables_initializer())
saver = tf.train.Saver()
```

다음 배치 학습을 시작한다.

```
# start training
i = 0
for _ in range(10000):
    i += 1
    batch_xs, batch_ts = mnist.train.next_batch(50)
    sess.run(train_step,
              feed_dict={x:batch_xs, t:batch_ts, keep_prob:0.5})
    if i % 500 == 0:
        loss_vals, acc_vals = [], []
        for c in range(4):
            start = len(mnist.test.labels) / 4 * c
            end = len(mnist.test.labels) / 4 * (c+1)
            loss_val, acc_val = sess.run([loss, accuracy],
                                         feed_dict={x:mnist.test.images[start:end],
                                                     t:mnist.test.labels[start:end],
                                                     keep_prob:1.0})
            loss_vals.append(loss_val)
            acc_vals.append(acc_val)
        loss_val = np.sum(loss_vals)
        acc_val = np.mean(acc_vals)
        print ('Step: %d, Loss: %f, Accuracy: %f'
              % (i, loss_val, acc_val))
```

학습은 10,000번 루프를 돌면서 한번에 50개씩 배치로 데이터를 읽어서 학습을 진행하고, 500 번째 마다 중간 학습 결과를 출력한다. 중간 학습 결과에서는 10,000 중 몇번째 학습인지와, 비용값 그리고 정확도를 출력해준다.

코드를 보자

```
batch_xs, batch_ts = mnist.train.next_batch(50)
```

MNIST 학습용 데이터 셋에서 50개 단위로 데이터를 읽는다. batch_xs에는 학습에 사용할 28x28x1 사이즈의 이미지와, batch_ts에는 그 이미지에 대한 라벨 (0..9중 어떤 수인지) 가 들어 있다.

읽은 데이터를 feed_dict를 통해서 피딩(입력)하고 트레이닝 세션을 시작한다.

```
sess.run(train_step,
          feed_dict={x:batch_xs, t:batch_ts, keep_prob:0.5})
```

이때 마지막 인자에 keep_prob를 0.5로 피딩하는 것을 볼 수 있는데, keep_prob는 앞의 드롭아웃 계층에서 정의한 변수로 드롭아웃을 거치지 않을 비율을 정의한다. 여기서는 0.5 즉 50%의 네트워크를 인위적으로 끊도록 하였다.

배치로 학습을 진행하다가 500번 마다 중간중간 정확도와 학습 비용을 계산하여 출력한다.

```
if i % 500 == 0:
    loss_vals, acc_vals = [], []
```

여기서 주목할 점은 아래 코드 처럼 한번에 검증을 하지 않고 테스트 데이터를 4등분 한후, 1/4씩 테스트 데이터를 로딩해서 학습비용(loss)와 학습 정확도(accuracy)를 계산하는 것을 볼 수 있다.

```
for c in range(4):
    start = len(mnist.test.labels) / 4 * c
    end = len(mnist.test.labels) / 4 * (c+1)
    loss_val, acc_val = sess.run([loss, accuracy],
        feed_dict={x:mnist.test.images[start:end],
                    t:mnist.test.labels[start:end],
                    keep_prob:1.0})
    loss_vals.append(loss_val)
    acc_vals.append(acc_val)
```

이유는 한꺼번에 많은 데이터를 로딩해서 검증을 할 경우 메모리 문제가 생길 수 있기 때문에, 4번에 나눠 걸쳐서 읽고 검증한 다음에 아래와 같이 학습 비용은 4번의 학습 비용을 합하고, 정확도는 4번의 학습 정확도를 평균으로 내어 출력하였다.

```
loss_val = np.sum(loss_vals)
acc_val = np.mean(acc_vals)
print ('Step: %d, Loss: %f, Accuracy: %f'
        % (i, loss_val, acc_val))
```

학습 결과 저장

학습을 통해서 최적의 W와 b값을 구했으면 이 값을 예측에 이용해야 하는데, W 값들이 많고, 이를 일일이 출력해서 파일로 저장하는 것도 번거롭고 해서, 텐서플로우에서는 학습된 모델을 저장할 수 있는 기능을 제공한다. 학습을 통해서 계산된 모든 변수 값을 저장할 수 있는데, 앞에서 세션을 생성할때 생성한 Saver (saver = tf.train.Saver())를 이용하면 현재 학습 세션을 저장할 수 있다.

코드

```
saver.save(sess, 'cnn_session')
sess.close()
```

이렇게 하면 현재 디렉토리에 cnn_session* 형태의 파일로 학습된 세션 값들이 저장된다. 그래서 추후 예측을 할때 다시 학습할 필요 없이 이 파일을 로딩해서, 모델의 값들을 복귀한 후에, 예측을 할 수 있다. 이 파일을 읽어서 예측을 하는 것은 다음글에서 다루기로 한다.

예제 코드 : https://github.com/bwcho75/tensorflowML/blob/master/MNIST_CNN_Training.ipynb

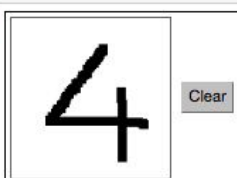
숫자 예측하기

앞서 MNIST 데이터를 이용한 필기체 숫자를 인식하는 모델을 컨볼루션 신경망 (CNN)을 이용하여 만들었다. 이번에는 이 모델을 이용해서 필기체 숫자 이미지를 인식하는 코드를 만들어 보자

조금 더 테스트를 쉽게 하기 위해서, 파이썬 주피터 노트북내에서 HTML 을 이용하여 마우스로 숫자를 그릴 수 있도록 하고, 그려진 이미지를 어떤 숫자인지 인식하도록 만들어 보겠다.

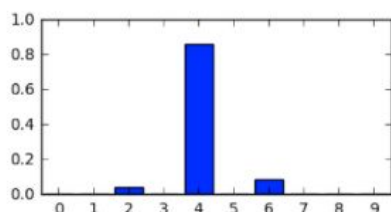
```
In [8]: from IPython.display import HTML
HTML(input_form + javascript|
```

Out[8]:



```
In [5]: p_val = sess.run(p, feed_dict={x:[image], keep_prob:1.0})
```

```
fig = plt.figure(figsize=(4,2))
pred = p_val[0]
subplot = fig.add_subplot(1,1,1)
subplot.set_xticks(range(10))
subplot.set_xlim(-0.5,9.5)
subplot.set_ylim(0,1)
subplot.bar(range(10), pred, align='center')
plt.show()
```



```
In [4]: convl_vals, cutoffl_vals = sess.run(
[h_conv1, h_conv1_cutoff], feed_dict={x:[image], keep_prob:1.0})
```

```
fig = plt.figure(figsize=(16,4))

for f in range(num_filters1):
    subplot = fig.add_subplot(4, 16, f+1)
    subplot.set_xticks([])
    subplot.set_yticks([])
    subplot.imshow(convl_vals[0,:, :, f],
                    cmap=plt.cm.gray_r, interpolation='nearest')

plt.show()
```



모델 로딩

먼저 앞의 예제에서 학습을한 모델을 로딩해보도록 하자.

이 코드는 주피터 노트북에서 작성할때, 모델을 학습 시키는 코드 (<http://bcho.tistory.com/1156>)와 별도의 새노트북에서 구현을 하도록 한다.

코드

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.examples.tutorials.mnist import input_data

#이미 그래프가 있을 경우 중복이 될 수 있기 때문에, 기존 그래프를 모두 리셋한다.
tf.reset_default_graph()

num_filters1 = 32

x = tf.placeholder(tf.float32, [None, 784])
x_image = tf.reshape(x, [-1,28,28,1])

# layer 1
W_conv1 = tf.Variable(tf.truncated_normal([5,5,1,num_filters1],
                                          stddev=0.1))
h_conv1 = tf.nn.conv2d(x_image, W_conv1,
                      strides=[1,1,1,1], padding='SAME')

b_conv1 = tf.Variable(tf.constant(0.1, shape=[num_filters1]))
h_conv1_cutoff = tf.nn.relu(h_conv1 + b_conv1)

h_pool1 = tf.nn.max_pool(h_conv1_cutoff, ksize=[1,2,2,1],
                        strides=[1,2,2,1], padding='SAME')

num_filters2 = 64

# layer 2
W_conv2 = tf.Variable(
    tf.truncated_normal([5,5,num_filters1,num_filters2],
                        stddev=0.1))
h_conv2 = tf.nn.conv2d(h_pool1, W_conv2,
                      strides=[1,1,1,1], padding='SAME')

b_conv2 = tf.Variable(tf.constant(0.1, shape=[num_filters2]))
h_conv2_cutoff = tf.nn.relu(h_conv2 + b_conv2)

h_pool2 = tf.nn.max_pool(h_conv2_cutoff, ksize=[1,2,2,1],
                        strides=[1,2,2,1], padding='SAME')

# fully connected layer
h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*num_filters2])

num_units1 = 7*7*num_filters2
num_units2 = 1024

w2 = tf.Variable(tf.truncated_normal([num_units1, num_units2]))
b2 = tf.Variable(tf.constant(0.1, shape=[num_units2]))
hidden2 = tf.nn.relu(tf.matmul(h_pool2_flat, w2) + b2)
```

```

keep_prob = tf.placeholder(tf.float32)
hidden2_drop = tf.nn.dropout(hidden2, keep_prob)

w0 = tf.Variable(tf.zeros([num_units2, 10]))
b0 = tf.Variable(tf.zeros([10]))
k = tf.matmul(hidden2_drop, w0) + b0
p = tf.nn.softmax(k)

# prepare session
sess = tf.InteractiveSession()
sess.run(tf.global_variables_initializer())
saver = tf.train.Saver()
saver.restore(sess, '/Users/terrycho/anaconda/work/cnn_session')

print 'reload has been done'

```

그래프 구현

코드를 살펴보면, #prepare session 부분 전까지는 이전 코드에서의 그래프를 정의하는 부분과 동일하다. 이 코드는 우리가 만든 컨볼루션 네트워크를 복원하는 부분이다.

변수 데이터 로딩

그래프의 복원이 끝나면, 저장한 세션의 값을 다시 로딩해서 학습된 W와 b값들을 다시 로딩한다.

```

# prepare session
sess = tf.InteractiveSession()
sess.run(tf.global_variables_initializer())
saver = tf.train.Saver()
saver.restore(sess, '/Users/terrycho/anaconda/work/cnn_session')

```

이때 saver.restore 부분에서 앞의 예제에서 저장한 세션의 이름을 지정해준다.

HTML을 이용한 숫자 입력

그래프와 모델 복원이 끝났으면 이 모델을 이용하여, 숫자를 인식해본다.
테스트하기 편리하게 HTML로 마우스로 숫자를 그릴 수 있는 화면을 만들어보겠다.
주피터 노트북에서 새로운 Cell에 아래와 같은 내용을 입력한다.

코드

```

input_form = """
<table>
<td style="border-style: none;">
<div style="border: solid 2px #666; width: 143px; height: 144px;">
<canvas width="140" height="140"></canvas>
</div></td>
<td style="border-style: none;">
<button onclick="clear_value()">Clear</button>
</td>
</table>
"""

```



```

javascript = """
<script type="text/Javascript">
  var pixels = [];
  for (var i = 0; i < 28*28; i++) pixels[i] = 0
  var click = 0;

  var canvas = document.querySelector("canvas");
  canvas.addEventListener("mousemove", function(e){
    if (e.buttons == 1) {
      click = 1;
      canvas.getContext("2d").fillStyle = "rgb(0,0,0)";
      canvas.getContext("2d").fillRect(e.offsetX, e.offsetY, 8, 8);
      x = Math.floor(e.offsetY * 0.2)
      y = Math.floor(e.offsetX * 0.2) + 1
      for (var dy = 0; dy < 2; dy++){
        for (var dx = 0; dx < 2; dx++){
          if ((x + dx < 28) && (y + dy < 28)){
            pixels[(y+dy)+(x+dx)*28] = 1
          }
        }
      }
    } else {
      if (click == 1) set_value()
      click = 0;
    }
  });

  function set_value(){
    var result = ""
    for (var i = 0; i < 28*28; i++) result += pixels[i] + ","
    var kernel = IPython.notebook.kernel;
    kernel.execute("image = [" + result + "]");
  }

  function clear_value(){
    canvas.getContext("2d").fillStyle = "rgb(255,255,255)";
    canvas.getContext("2d").fillRect(0, 0, 140, 140);
    for (var i = 0; i < 28*28; i++) pixels[i] = 0
  }
</script>
"""

```

다음 새로운 셀에서, 다음 코드를 입력하여, 앞서 코딩한 HTML 파일을 실행할 수 있도록 한다.

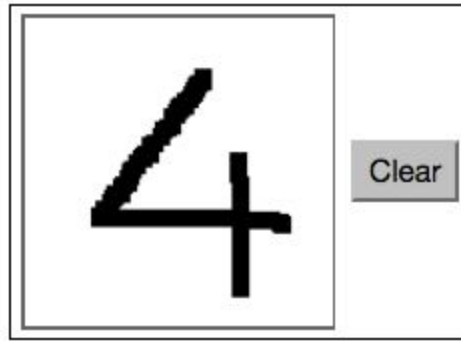
```

from IPython.display import HTML
HTML(input_form + javascript)

```

이제 앞에서 만든 두 셀을 실행시켜 보면 다음과 같이 HTML 기반으로 마우스를 이용하여 숫자를 입력할 수 있는 박스가 나오는것을 확인할 수 있다.

Out[8]:



입력값 판정

앞의 HTML에서 그린 이미지는 앞의 코드의 `set_value`라는 함수에 의해서, `image` 라는 변수로 784 크기의 벡터에 저장된다. 이 값을 이용하여, 이 그림이 어떤 숫자인지를 앞서 만든 모델을 이용해서 예측을 해본다.

코드

```
p_val = sess.run(p, feed_dict={x:[image], keep_prob:1.0})
```

```
fig = plt.figure(figsize=(4,2))
pred = p_val[0]
subplot = fig.add_subplot(1,1,1)
subplot.set_xticks(range(10))
subplot.set_xlim(-0.5,9.5)
subplot.set_ylim(0,1)
subplot.bar(range(10), pred, align='center')
plt.show()
```

예측

예측을 하는 방법은 쉽다. 이미지 데이터가 `image` 라는 변수에 들어가 있기 때문에, 어떤 숫자인지에 대한 확률을 나타내는 `p` 의 값을 구하면 된다.

```
p_val = sess.run(p, feed_dict={x:[image], keep_prob:1.0})
```

를 이용하여 `x`에 `image`를 넣고, 그리고 `dropout` 비율을 0%로 하기 위해서 `keep_prob`를 1.0 (100%)로 한다. (예측이기 때문에 당연히 `dropout`은 필요하지 않다.)
이렇게 하면 이 이미지가 어떤 숫자인지에 대한 확률이 `p`에 저장된다.

그래프로 표현

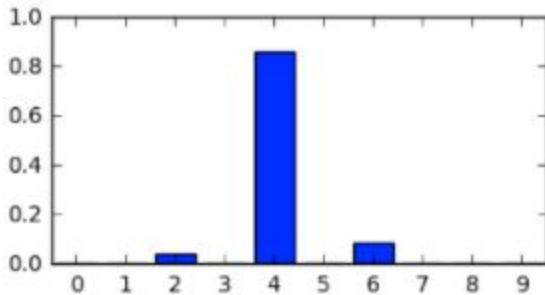
그러면 이 `p`의 값을 찍어 보자

```
fig = plt.figure(figsize=(4,2))
pred = p_val[0]
subplot = fig.add_subplot(1,1,1)
subplot.set_xticks(range(10))
```

```
subplot.set_xlim(-0.5,9.5)
subplot.set_ylim(0,1)
subplot.bar(range(10), pred, align='center')
plt.show()
```

그래프를 이용하여 0~9 까지의 숫자 (가로축)일 확률을 0.0~1.0 까지 (세로축)으로 출력하게 된다.

다음은 위에서 입력한 숫자 “4”를 인식한 결과이다.



(보너스) 첫번째 컨볼루셔널 계층 결과 출력

컨볼루셔널 네트워크를 학습시키다 보면 종종 컨볼루셔널 계층을 통과하여 추출된 특징 이미지들이 어떤 모양을 가지고 있는지를 확인하고 싶을때가 있다. 그래서 각 필터를 통과한 값을 이미지로 출력하여 확인하고는 하는데, 여기서는 이렇게 각 필터를 통과하여 인식된 특징이 어떤 모양인지를 출력하는 방법을 소개한다.

아래는 우리가 만든 네트워크 중에서 첫번째 컨볼루셔널 필터를 통과한 결과 h_conv1과, 그리고 이 결과에 bias 값을 더하고 활성화 함수인 Relu를 적용한 결과를 출력하는 예제이다.

코드

```
conv1_vals, cutoff1_vals = sess.run(
    [h_conv1, h_conv1_cutoff], feed_dict={x:[image], keep_prob:1.0})
```

```
fig = plt.figure(figsize=(16,4))
```

```
for f in range(num_filters1):
    subplot = fig.add_subplot(4, 16, f+1)
    subplot.set_xticks([])
    subplot.set_yticks([])
    subplot.imshow(conv1_vals[0,:,:f],
                    cmap=plt.cm.gray_r, interpolation='nearest')
plt.show()
```

x에 image를 입력하고, dropout을 없이 모든 네트워크를 통과하도록 keep_prob:1.0으로 주고, 첫번째 컨볼루셔널 필터를 통과한 값 h_conv1 과, 이 값에 bias와 Relu를 적용한 값 h_conv1_cutoff를 계산하였다.

```
conv1_vals, cutoff1_vals = sess.run(
    [h_conv1, h_conv1_cutoff], feed_dict={x:[image], keep_prob:1.0})
```

첫번째 필터는 총 32개로 구성되어 있기 때문에, 32개의 결과값을 imshow 함수를 이용하여 흑백으로 출력하였다.



다음은 bias와 Relu를 통과한 값인 h_conv_cutoff를 출력하는 예제이다. 위의 코드와 동일하며 subplot.imshow에서 전달해주는 인자만 conv1_vals → cutoff1_vals로 변경되었다.

코드

```
fig = plt.figure(figsize=(16,4))

for f in range(num_filters1):
    subplot = fig.add_subplot(4, 16, f+1)
    subplot.set_xticks([])
    subplot.set_yticks([])
    subplot.imshow(cutoff1_vals[0,:,:f],
                   cmap=plt.cm.gray_r, interpolation='nearest')

plt.show()
```

출력 결과는 다음과 같다



이제까지 컨볼루션 네트워크를 이용한 이미지 인식을 텐서플로우로 구현하는 방법을 MNIST(필기체 숫자 데이터)를 이용하여 구현하였다.

실제로 이미지를 인식하려면 전체적인 흐름은 같지만, 이미지를 전/후처리 해내야 하고 또한 한대의 머신이 아닌 여러대의 머신과 GPU와 같은 하드웨어 장비를 사용한다. 다음 글에서는 MNIST가 아니라 실제 칼라 이미지를 인식하는 방법에 대해서 데이터 전처리에서 부터 서비스까지 전체 과정에 대해서 설명하도록 하겠다.

예제 코드 :

https://github.com/bwcho75/tensorflowML/blob/master/MNIST_CNN_Prediction.ipynb

8. 텐서플로우 고급 개념

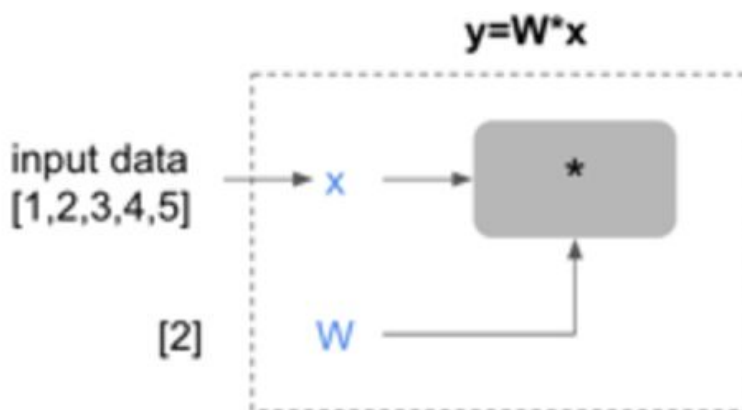
CSV 데이터 파일 읽기와 큐의 개념

텐서플로우를 학습하면서 실제 모델을 만들어보려고 하니 생각보다 데이터 처리에 대한 부분에서 많은 노하우가 필요하다는 것을 알게되었다. MNIST와 같은 예제는 데이터가 다 이쁘게 정리되어서 학습 하기 좋은 형태로 되어 있지만, 실제로 내 모델을 만들고 학습을 하기 위해서는 데이터에 대한 정제와 분류 작업등이 많이 필요하다.

이번글에서는 학습에 필요한 데이터를 파일에서 읽을때 필요한 큐에 대한 개념에 대해서 알아보도록 한다.

피딩 (Feeding) 개념 복습

텐서플로우에서 모델을 학습 시킬때, 학습 데이터를 모델에 적용하는 방법은 일반적으로 피딩 (feeding)이라는 방법을 사용한다. 메모리상의 어떤 변수 리스트 형태로 값을 저장한 후에, 모델을 세션에서 실행할 때, 리스트에서 값을 하나씩 읽어서 모델에 집어 넣는 방식이다.



위의 그림을 보면, $y=W*x$ 라는 모델에서 학습 데이터 x 는 $[1,2,3,4,5]$ 로, 첫번째 학습에는 1, 두번째 학습에는 2를 적용하는 식으로 피딩이 된다.

그런데, 이렇게 피딩을 하려면, 학습 데이터 $[1,2,3,4,5]$ 가 메모리에 모두 적재되어야 하는데, 실제로 모델을 만들어서 학습을 할때는 데이터의 양이 많기 때문에 메모리에 모두 적재하고 학습을 할 수 가 없고, 파일에서 읽어드리면서 학습을 해야 한다.

텐서플로우 큐에 대해서

이러한 문제를 해결하기 위해서는 파일에서 데이터를 읽어가면서, 읽은 데이터를 순차적으로 모델에 피딩하면 되는데, 이때 큐를 사용한다.

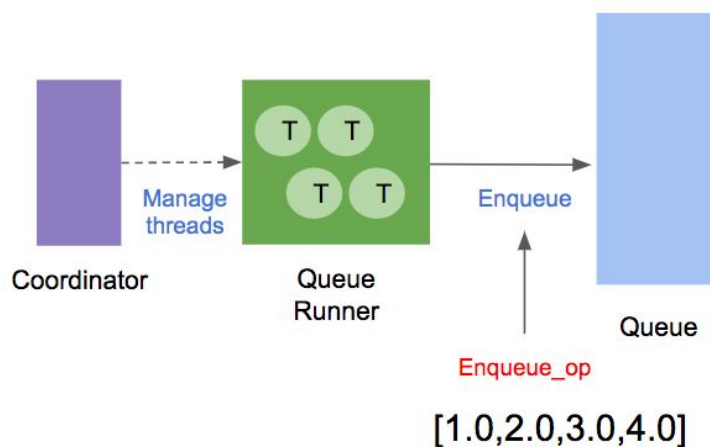
파일에서 데이터를 읽는 방법에 앞서서 큐를 설명하면, 큐에 데이터를 넣는 것(Enqueue)은 Queue Runner 라는 것이 한다.

이 Queue Runner가 큐에 어떤 데이터를 어떻게 넣을지를 정의 하는 것이

Enqueue_operation인데, 데이터를 읽어서 실제로 어떻게 Queue에 Enqueue 하는지를 정의한다.

이 Queue Runner는 멀티 쓰레드로 작동하는데, Queue Runner 안의 쓰레드들을 관리해 주기 위해서 별도로 Coordinator라는 것을 사용한다.

이 개념을 정리해서 도식화 해주면 다음과 같다.



Queue Runner 는 여러개의 쓰레드 (T)를 가지고 있고, 이 쓰레드들은 Coordinator들에 의해서 관리된다. Queue Runner 가 Queue에 데이터를 넣을때는 Enqueue_op이라는 operation에 의해 정의된 데로 데이터를 Queue에 집어 넣는다.

위의 개념을 코드로 구현해보자

```
import tensorflow as tf

QUEUE_LENGTH = 20
q = tf.FIFOQueue(QUEUE_LENGTH, "float")
enq_ops = q.enqueue_many([1.0, 2.0, 3.0, 4.0],) )
qr = tf.train.QueueRunner(q, [enq_ops, enq_ops, enq_ops])

sess = tf.Session()
# Create a coordinator, launch the queue runner threads.
```

```

coord = tf.train.Coordinator()
threads = qr.create_threads(sess, coord=coord, start=True)

for step in xrange(20):
    print(sess.run(q.dequeue()))

coord.request_stop()
coord.join(threads)

sess.close()

```

Queue 생성

tf.FIFOQUEUE를 이용해서 큐를 생성한다.

```
q = tf.FIFOQueue(QUEUE_LENGTH, "float")
```

첫번째 인자는 큐의 길이를 정하고, 두번째는 dtype으로 큐에 들어갈 데이터형을 지정한다.

Queue Runner 생성

다음은 Queue Runner를 만들기 위해서 enqueue_operation 과, QueueRunner를 생성한다.

```
enq_ops = q.enqueue_many([1.0,2.0,3.0,4.0],)
```

```
qr = tf.train.QueueRunner(q,[enq_ops,enq_ops,enq_ops])
```

enqueue operation인 enq_ops는 위와 같이 한번에 [1.0,2.0,3.0,4.0] 을 큐에 넣는 operation으로 지정한다.

그리고 Queue Runner를 정의하는데, 앞에 만든 큐에 데이터를 넣을것이기 때문에 인자로 큐 'q'를 넘기고 list 형태로 enq_ops를 3개를 넘긴다. 3개를 넘기는 이유는 Queue Runner가 멀티쓰레드 기반이기 때문에 각 쓰레드에서 Enqueue시 사용할 Operation을 넘기는 것으로, 3개를 넘긴것은 3개의 쓰레드에 Enqueue 함수를 각각 지정한 것이다.

만약 동일한 enqueue operation을 여러개의 쓰레드로 넘길 경우 위 코드처럼 일일이 enqueue operation을 쓸 필요 없이

```
qr = tf.train.QueueRunner(q,[enq_ops]*NUM_OF_THREAD)
```

[enq_ops] 에 쓰레드 수 (NUM_OF_THREAD)를 곱해주면 된다.

Coordinator 생성

이제 Queue Runner에서 사용할 쓰레드들을 관리할 Coordinator를 생성하자

```
coord = tf.train.Coordinator()
```

Queue Runner용 쓰레드 생성

Queue Runner와 쓰레드를 관리할 Coordinator 가 생성되었으면, Queue Runner에서 사용할 쓰레드들을 생성하자

```
threads = qr.create_threads(sess, coord=coord, start=True)
```

생성시에는 세션과, Coordinator를 지정하고, start=True로 해준다.

start=True로 설정하지 않으면, 쓰레드가 생성은 되었지만, 동작을 하지 않기 때문에, 큐에 메시지를 넣지 않는다.

큐 사용

이제 큐에서 데이터를 꺼내와 보자. 아래코드는 큐에서 20번 데이터를 꺼내와서 출력하는 코드이다.

```
for step in xrange(20):
    print(sess.run(q.dequeue()))
```

큐가 비워지면, QueueRunner를 이용하여 계속해서 데이터를 채워 넣는다. 즉 큐가 비기전에 계속해서 [1.0,2.0,3.0,4.0] 데이터가 큐에 계속 쌓인다.

쓰레드 정지

큐 사용이 끝났으면 Queue Runner의 쓰레드들을 모두 정지 시켜야 한다.

```
coord.request_stop()
```

을 이용하면 모든 쓰레드들을 정지 시킨다.

```
coord.join(threads)
```

는 다음 코드를 진행하기전에, Queue Runner의 모든 쓰레드들이 정지될때 까지 기다리는 코드이다.

멀티 쓰레드

Queue Runner가 멀티 쓰레드라고 하는데, 그렇다면 쓰레드들이 어떻게 데이터를 큐에 넣고 enqueue 연산은 어떻게 동작할까?

그래서, 간단한 테스트를 해봤다. 3개의 쓰레드를 만든 후에, 각 쓰레드에 따른 enqueue operation을 다르게 지정해봤다.

```
import tensorflow as tf
```

```
QUEUE_LENGTH = 20
```

```
q = tf.FIFOQueue(QUEUE_LENGTH,"float")
```

```
enq_ops1 = q.enqueue_many([1.0,2.0,3.0],) )
```

```
enq_ops2 = q.enqueue_many([4.0,5.0,6.0],) )
```

```
enq_ops3 = q.enqueue_many([6.0,7.0,8.0],) )
```

```
qr = tf.train.QueueRunner(q,[enq_ops1,enq_ops2,enq_ops3])
```

```
sess = tf.Session()
```

```
# Create a coordinator, launch the queue runner threads.
```

```
coord = tf.train.Coordinator()
```

```
threads = qr.create_threads(sess, coord=coord, start=True)
```

```
for step in xrange(20):
```

```
    print(sess.run(q.dequeue()))
```

```
coord.request_stop()
```

```
coord.join(threads)
```

```
sess.close()
```


실행을 했더니, 다음과 같은 결과를 얻었다.

첫번째 실행 결과

1.0
2.0
3.0
4.0
5.0
6.0
6.0
7.0
8.0

두번째 실행결과

1.0
2.0
3.0
1.0
2.0
3.0
4.0
5.0
6.0

결과에서 보는것과 같이 Queue Runner의 3개의 쓰레드중 하나가 무작위로 (순서에 상관없이) 실행되서 데이터가 들어가는 것을 볼 수 있었다.

파일에서 데이터 읽기

자 그러면 이 큐를 이용해서, 파일 목록을 읽고, 파일을 열어서 학습 데이터를 추출해서 학습 파이프라인에 데이터를 넣어주면 된다.

텐서 플로우에서는 파일에서 데이터를 읽는 처리를 위해서 앞에서 설명한 큐 뿐만 아니라 Reader와 Decoder와 같은 부가적인 기능을 제공한다.

1. 파일 목록을 읽는다.
2. 읽은 파일목록을 filename queue에 저장한다.
3. Reader 가 filename queue 에서 파일명을 하나씩 읽어온다.
4. Decoder에서 해당 파일을 열어서 데이터를 읽어들인다.
5. 필요하면 읽어드린 데이터를 텐서플로우 모델에 맞게 정제한다. (이미지를 리사이즈 하거나, 칼라 사진을 흑백으로 바꾸거나 하는 등의 작업)
6. 텐서 플로우에 맞게 정제된 학습 데이터를 학습 데이터 큐인 Example Queue에 저장한다.
7. 모델에서 Example Queue로 부터 학습 데이터를 읽어서 학습을 한다.

파일명 목록 읽어오기

먼저 파일 목록을 읽는 부분은 파일 목록을 읽어서 각 파일명을 큐에 넣은 부분을 살펴보자. 다음 예제코드는 파일명 목록을 받은 후에, filename queue에 파일명을 넣은 후에, 파일명을 하나씩 꺼내는 예제이다.

```
import tensorflow as tf
```

```
filename_queue = tf.train.string_input_producer(["1","2","3"],shuffle=False)
```

```
with tf.Session() as sess:
```

```
    coord = tf.train.Coordinator()
```

```
    threads = tf.train.start_queue_runners(coord=coord,sess=sess)
```

```
    for step in xrange(10):
```

```
        print(sess.run(filename_queue.dequeue())) )
```

```
    coord.request_stop()
```

```
    coord.join(threads)
```

코드를 보면 큐 생성이나, enqueue operation 처리들이 다소 다른것을 볼 수 있는데, 이는 텐서플로우에서는 학습용 파일 목록을 편리하게 처리 하기 위해서 조금 더 추상화된 함수들을 제공하기 때문이다.

```
filename_queue = tf.train.string_input_producer(["1","2","3"],shuffle=False)
```

train.xx_input_producer() 함수는 입력 받은 큐를 만드는 역할을 한다.

위의 명령을 수행하면, filename queue 가 FIFO (First In First Out)형태로 생긴다.

File names
["1","2","3"]



Filename
queue

큐가 생기기는 하지만, 실제로 큐에 파일명이 들어가지는 않는다. (아직 Queue Runner와 쓰레드들을 생성하지 않았기 때문에)

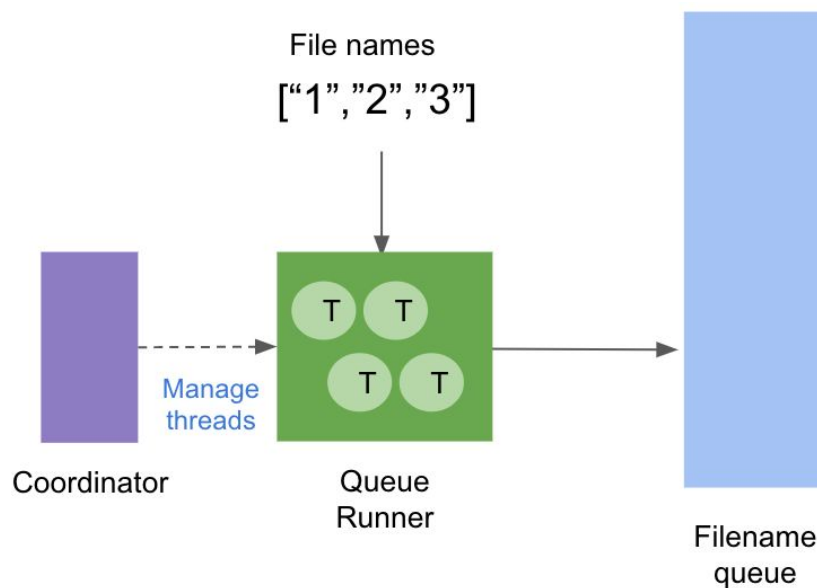
다음으로 쓰레드를 관리하기 위한 Coordinator 를 생성한다.

```
coord = tf.train.Coordinator()
```

Coordinator 가 생성이 되었으면 Queue Runner와 Queue Runner에서 사용할 Thread들을 생성해주는데, `start_queue_runner` 라는 함수로, 이 기능들을 모두 구현해왔다.

```
threads = tf.train.start_queue_runners(coord=coord, sess=sess)
```

이 함수는 Queue Runner와, 쓰레드 생성 및 시작 뿐 만 아니라 Queue Runner 쓰레드가 사용하는 enqueue operation 까지 파일형태에 맞춰서 자동으로 생성 및 지정해준다.



Queue, Queue Runner, Coordinator와 Queue Runner가 사용할 쓰레드들이 생성되고 시작되었기 때문에, Queue Runner는 filename queue에 파일명을 enqueue 하기 시작한다.

파일명 Shuffling

위의 예제를 실행하면 파일명이 다음과 같이 1,2,3 이 순차적으로 반복되서 나오는 것을 볼 수 있다.

실행 결과

```
1
2
3
1
2
3
1
2
3
1
```

만약에 파일명을 랜덤하게 섞어서 나오게 하려면 어떻게해야 할까? (매번 학습시 학습데이터가 일정 패턴으로 몰려서 편향되지 않고, 랜덤하게 나와서 학습 효과를 높이하고자 할때)

```
filename_queue = tf.train.string_input_producer(["1","2","3"],shuffle=False)
```

큐를 만들때, 다음과 같이 셔플 옵션을 True로 주면 된다.

```
filename_queue = tf.train.string_input_producer(["1","2","3"],shuffle=True)
```

실행 결과

```
2
1
3
2
3
1
2
3
1
1
```

지금까지 파일명을 지정해서 이 파일명들을 filename queue에 넣는 방법에 대해서 알아보았다.

다음은 이 file name queue에서 파일을 순차적으로 꺼내서

- 파일을 읽어드리고
- 각 파일을 파싱해서 학습 데이터를 만들고
- 학습 데이터용 큐 (example queue)에 넣는 방법

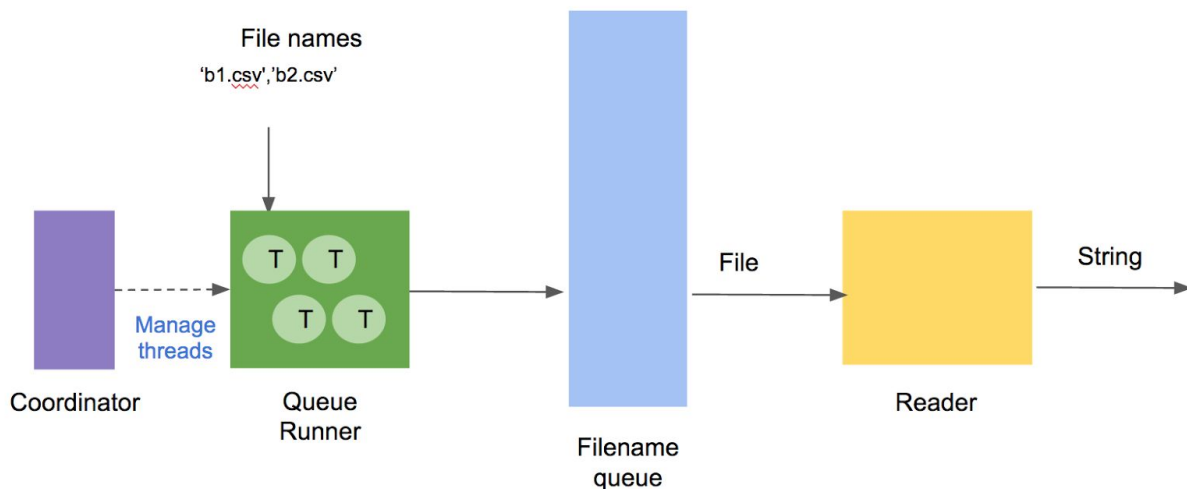
에 대해서 설명하도록 한다.

파일에서 데이터 읽기 (Reader)

filename_queue에 파일명이 저장되었으면, 이 파일들을 하나씩 읽어서 처리하는 방법을 알아본다.

파일에서 데이터를 읽어오는 컴포넌트를 Reader라고 한다. 이 Reader들은 filename_queue에 저장된 파일들을 하나씩 읽어서, 그 안에 있는 데이터를 읽어서 리턴한다.

예를 들어 TextLineReader의 경우에는 , 텍스트 파일에서, 한줄씩 읽어서 문자열을 리턴한다.



꼭 텐서플로우에서 미리 정해져있는 Reader 들을 사용할 필요는 없지만, 미리 정의된 Reader를 쓰면 조금 더 편리하다.

미리 정의된 Reader로는 Text File에서, 각 필드가 일정한 길이를 가지고 있을때 사용할 수 있는, FixedLengthRecordReader 그리고, 텐서플로우 데이터를 바이너리 포맷으로 저장하는 TFRecord 포맷에 대한 리더인 TFRecordReader 등이 있다.

Reader를 사용하는 방법은 다음과 같다.

```

reader = tf.TextLineReader()
key,value = reader.read(filename_queue)
  
```

먼저 Reader 변수를 지정한 다음, reader.read를 이용하여 filename_queue 로 부터 파일을 읽게 하면 value에 파일에서 읽은 값이 리턴이 된다

예를 들어 csv 파일에 아래와 같은 문자열이 들어가 있다고 할때

```

167c9599-c97d-4d42-bdb1-027ddaed07c0,1,2016,REG,3:54
67ea7e52-333e-43f3-a668-6d7893baa8fb,1,2016,REG,2:11
9e44593b-a870-446e-aed5-90a22ab0c952,1,2016,REG,2:32
48832a52-e56c-467f-a1ef-c6f8c6e908ea,1,2016,REG,2:17
  
```

위의 코드 처럼, TextLineReader를 이용하여 파일을 읽게 되면 value에는 처음에는 “167c9599-c97d-4d42-bdb1-027ddaed07c0,1,2016,REG,3:54”이, 다음에는 “67ea7e52-333e-43f3-a668-6d7893baa8fb,1,2016,REG,2:11” 문자열이 순차적으로 리턴된다.

읽은 데이터를 디코딩 하기 (Decoder)

Reader에서 읽은 값은 파일의 원시 데이터 (raw) 데이터이다. 아직 파싱(해석)이 된 데이터가 아닌데,

예를 들어 Reader를 이용해서 csv 파일을 읽었을 때, Reader에서 리턴되는 값은 csv 파일의 각 줄인 문자열이지, csv 파일의 각 필드 데이터가 아니다.

즉 우리가 학습에서 사용할 데이터는

167c9599-c97d-4d42-bdb1-027ddaed07c0,1,2016,REG,3:54

하나의 문자열이 아니라

Id = "167c9599-c97d-4d42-bdb1-027ddaed07c0",

Num = 1

Year = 2016

rType = "REG"

rTime = "3:54"

과 같이 문자열이 파싱된 각 필드의 값이 필요하다.

이렇게 읽어드린 데이터를 파싱 (해석) 하는 컴포넌트를 Decoder라고 한다.

Reader와 마찬가지로, Decoder 역시 미리 정해진 Decoder 타입이 있는데, JSON, CSV 등 여러가지 데이터 포맷에 대한 Decoder를 지원한다.

위의 CSV 문자열을 csv 디코더를 이용하여 파싱해보자

```
record_defaults = [ ["null"],[1],[1900],["null"],["null"]]
id, num, year, rtype, rtime = tf.decode_csv(
    value, record_defaults=record_defaults,field_delim=',')
```

csv decoder를 사용하기 위해서는 각 필드의 디폴트 값을 지정해줘야 한다. record_default는 각 필드의 디폴트 값을 지정해 주는 것은 물론이고, 각 필드의 데이터 타입을 (string,int,float etc)를 정의 하는 역할을 한다.

디폴트 값은 csv 데이터에서 해당 필드가 비워져 있을때 채워 진다.

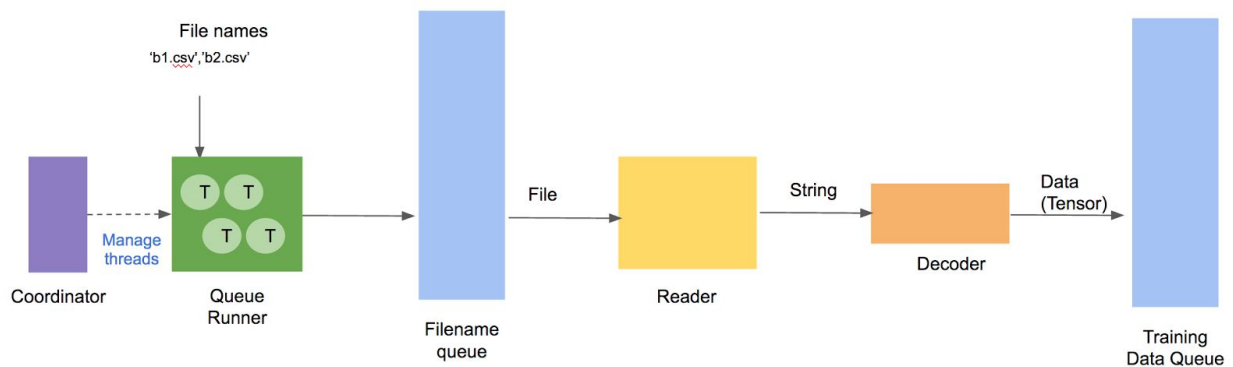
위에서는 record_default에서 첫번째 필드는 string 형이고 디폴트는 "null"로, 두번째 필드는 integer 형이고, 디폴트 값은 1로, 세번째 필드는 integer 형이고 디폴트는 1900 으로, 네번째와 다섯번째 필드는 모두 string형이고, 디폴트 값을 "null" 로 지정하였다.

이 디폴트 값 세팅을 가지고 tf.decode_csv를 이용하여 파싱 한다.

value는 앞에서 읽어 드린 CSV 문자열이다. record_defaults= 를 이용하여 레코드의 형과 디폴트 값을 record_defaults에 정해진 값으로 지정하였고, CSV 파일에서 각 필드를 구분하기 위한 구분자를 ','를 사용한다는 것을 명시 하였다.

다음 Session을 실행하여, 이 Decoder를 실행하면 csv의 각 행을 파싱하여, 각 필드를 id,num,year,rtype,rtime이라는 필드에 리턴하게 된다.

이를 정리해보면 다음과 같은 구조를 가지게 된다.



예제

위에서 설명한 CSV 파일명을 받아서 TextLineReader를 이용하여 각 파일을 읽고, 각 파일에서 CSV 포맷의 데이터를 읽어서 출력하는 예제의 전체 코드를 보면 다음과 같다.

```
import tensorflow as tf
from numpy.random.mtrand import shuffle

#define filename queue
filename_queue =
tf.train.string_input_producer(['/Users/terrycho/training_datav2/queue_test_data/b1.csv'
                               ,'/Users/terrycho/training_datav2/queue_test_data/c2.csv']
                               ,shuffle=False,name='filename_queue')

# define reader
reader = tf.TextLineReader()
key,value = reader.read(filename_queue)

#define decoder
record_defaults = [ ["null"],[1],[1900],["null"],["null"]]
id, num, year, rtype , rtime = tf.decode_csv(
    value, record_defaults=record_defaults,field_delim=',')

with tf.Session() as sess:

    coord = tf.train.Coordinator()
    threads = tf.train.start_queue_runners(sess=sess, coord=coord)

    for i in range(100):
        print(sess.run([id, num, year, rtype , rtime]))

    coord.request_stop()
    coord.join(threads)
```

지금까지 파일에서 데이터를 읽어와서 학습 데이터로 사용하는 방법에 대해서 알아보았다. 다음에는 이미지 기반의 CNN 모델을 학습 시키기 위해서 이미지 데이터를 전처리 하고 읽는 방법에 대해서 설명하도록 하겠다.

텐서플로우에서 배치 데이터 처리

텐서플로우에서 파일에서 데이터를 읽은 후에, 배치처리로 placeholder에서 읽는 예제를 설명한다.

텐서의 shape 의 차원과 세션의 실행 시점등이 헛갈려서 시행착오가 많았기 때문에 글로 정리해놓는다.

데이터 포맷

읽어 드릴 데이터 포맷은 다음과 같다. 비행기 노선 정보에 대한 데이터로 “년도,항공사 코드,편명”을 기록한 CSV 파일이다.

2014,VX,121

2014,WN,1873

2014,WN,2787

배치 처리 코드

이 데이터를 텐서 플로우에서 읽어와서 배치로 place holder에 feeding 하는 코드 이다
먼저 read_data는 csv 파일에서 데이터를 읽어와서 파싱을 한 후 각 컬럼을 year,flight,time 으로 리턴하는 함수이다.

```
def read_data(file_name):
    try:
        csv_file = tf.train.string_input_producer([file_name],name='filename_queue')
        textReader = tf.TextLineReader()
        _,line = textReader.read(csv_file)
        year,flight,time = tf.decode_csv(line,record_defaults=[ [1900],[0],[0] ],field_delim=',')
    except:
        print "Unexpected error:",sys.exc_info()[0]
        exit()
    return year,flight,time
```

string_input_producer를 통해서 파일명들을 큐잉해서 하나씩 읽는데,여기서는 편의상 하나의 파일만 읽도록 하였는데, 여러개의 파일을 병렬로 처리하고자 한다면, [file_name] 부분에 리스트 형으로 여러개의 파일 목록을 지정해주면 된다.

다음 각 파일을 TextReader를 이용하여 라인 단위로 읽은 후 decode_csv를 이용하여, “,”로 분리된 컬럼을 각각 읽어와서 year,flight,time 에 저장하여 리턴하였다.

다음 함수는 read_data_batch 라는 함수인데, 앞에서 정의한 read_data 함수를 호출하여, 읽어드린 year,flight,time 을 배치로 묶어서 리턴하는 함수 이다.

```
def read_data_batch(file_name,batch_size=10):
```



```

year,flight,time = read_data(file_name)
batch_year,batch_flight,batch_time = tf.train.batch([year,flight,time],batch_size=batch_size)

return batch_year,batch_flight,batch_time

```

tf.train.batch 함수가 배치로 묶어서 리턴을 하는 함수인데, batch로 묶고자 하는 tensor 들을 인자로 준 다음에, batch_size (한번에 묶어서 리턴하고자 하는 텐서들의 개수)를 정해주면 된다.

위의 예제에서는 batch_size를 10으로 해줬기 때문에, batch_year = [1900,1901.....,1909] 와 같은 형태로 10개의 년도를 하나의 텐서에 묶어서 리턴해준다.

즉 입력 텐서의 shape이 [x,y,z] 일 경우 tf.train.batch를 통한 출력은 [batch_size,x,y,z] 가 된다.(이 부분이 핵심)

메인 코드

자 이제 메인 코드를 보자

def main():

```

    print 'start session'
    #coornator 위에 코드가 있어야 한다
    #데이터를 읽어 넣기 전에 미리 그래프가 만들어져 있어야 함.
    batch_year,batch_flight,batch_time = read_data_batch(TRAINING_FILE)
    year = tf.placeholder(tf.int32,[None,])
    flight = tf.placeholder(tf.string,[None,])
    time = tf.placeholder(tf.int32,[None,])

    tt = time * 10

```

tt = time * 10 이라는 공식을 실행하기 위해서 time 이라는 값을 읽어서 피딩하는 예제인데 먼저 read_data_batch를 이용하여 데이터를 읽는 그래프를 생성한다. 이때 주의해야할점은 이 함수를 수행한다고 해서, 바로 데이터를 읽기 시작하는 것이 아니라, 데이터의 흐름을 정의하는 그래프만 생성된다는 것을 주의하자

다음으로는 year,flight,time placeholder를 정의한다.

year,flight,time 은 0 차원의 scalar 텐서이지만, 값이 연속적으로 들어오기 때문에, [None,] 로 정의한다.

즉 year = [1900,1901,1902,1903,.....] 형태이기 때문에 1차원 Vector 형태의 shape으로 [None,] 로 정의한다.

Placeholder 들에 대한 정의가 끝났으면, 세션을 정의하고 데이터를 읽어드리기 위한 Queue runner를 수행한다. 앞의 과정까지 텐서 그래프를 다 그렸고, 이 그래프 값을 부어넣기 위해서, Queue runner 를 수행한 것이다.

```

with tf.Session() as sess:
    try:

        coord = tf.train.Coordinator()
        threads = tf.train.start_queue_runners(sess=sess, coord=coord)

```

Queue runner를 실행하였기 때문에 데이터가 데이터 큐로 들어가기 시작하고, 이 큐에 들어간 데이터를 읽어드리기 위해서, 세션을 실행한다.

```
y_,f_,t_ = sess.run([batch_year, batch_flight, batch_time])
print sess.run(tt, feed_dict={time:t_})
```

세션을 실행하면, batch_year, batch_flight, batch_time 값을 읽어서 y_,f_,t_ 변수에 각각 집어 넣은 다음에, t_ 값을 tt 공식의 time 변수에 feeding 하여, 값을 계산한다.

모든 작업이 끝났으면 아래와 같이 Queue runner를 정지 시킨다.

```
coord.request_stop()
coord.join(threads)
```

다음은 앞에서 설명한 전체 코드이다.

```
import tensorflow as tf
import numpy as np
import sys
```

```
TRAINING_FILE = '/Users/terrycho/dev/data/flight.csv'
```

```
## read training data and label
```

```
def read_data(file_name):
```

```
    try:
```

```
        csv_file = tf.train.string_input_producer([file_name], name='filename_queue')
```

```
        textReader = tf.TextLineReader()
```

```
        _, line = textReader.read(csv_file)
```

```
        year, flight, time = tf.decode_csv(line, record_defaults=[1900, "", 0], field_delim=',')
```

```
    except:
```

```
        print "Unexpected error:", sys.exc_info()[0]
```

```
        exit()
```

```
    return year, flight, time
```

```
def read_data_batch(file_name, batch_size=10):
```

```
    year, flight, time = read_data(file_name)
```

```
    batch_year, batch_flight, batch_time = tf.train.batch([year, flight, time], batch_size=batch_size)
```

```
    return batch_year, batch_flight, batch_time
```

```
def main():
```

```
    print 'start session'
```

```
    #coornator 위에 코드가 있어야 한다
```

```
    #데이터를 집어 넣기 전에 미리 그래프가 만들어져 있어야 함.
```

```
    batch_year, batch_flight, batch_time = read_data_batch(TRAINING_FILE)
```

```
    year = tf.placeholder(tf.int32, [None,])
```

```
    flight = tf.placeholder(tf.string, [None,])
```

```
    time = tf.placeholder(tf.int32, [None,])
```

```
    tt = time * 10
```

```
    with tf.Session() as sess:
```

```
        try:
```

```
            coord = tf.train.Coordinator()
```

```
            threads = tf.train.start_queue_runners(sess=sess, coord=coord)
```

```
            for i in range(10):
```

```
                y_, f_, t_ = sess.run([batch_year, batch_flight, batch_time])
```

```
                print sess.run(tt, feed_dict={time:t_})
```

```

    print 'stop batch'
    coord.request_stop()
    coord.join(threads)
except:
    print "Unexpected error:", sys.exc_info()[0]

main()

```

학습 데이터를 위한 TFRecord 파일 포맷

TFRecord 파일은 텐서플로우의 학습 데이터 등을 저장하기 위한 바이너리 데이터 포맷으로, 구글의 Protocol Buffer 포맷으로 데이터를 파일에 Serialize 하여 저장한다.

CSV 파일에서와 같이 숫자나 텍스트 데이터를 읽을 때는 크게 지장이 없지만, 이미지를 데이터를 읽을 경우 이미지는 JPEG나 PNG 형태의 파일로 저장되어 있고 이에 대한 메타 데이터와 라벨은 별도의 파일에 저장되어 있기 때문에, 학습 데이터를 읽을 때 메타데이터나 라벨용 파일 하나만 읽는 것이 아니라 이미지 파일도 별도로 읽어야 하기 때문에, 코드가 복잡해진다.

또한 이미지를 JPG나 PNG 포맷으로 읽어서 매번 디코딩을 하게 되면, 그 성능이 저하되서 학습단계에서 데이터를 읽는 부분에서 많은 성능 저하가 발생한다.

이와 같이 성능과 개발의 편의성을 이유로 TFRecord 파일 포맷을 이용하는 것이 좋다.

그러면 간단한 예제를 통해서 TFRecord 파일을 쓰고 읽는 방법에 대해서 알아보도록 하자
본 예제는 <http://warmspringwinds.github.io/tensorflow/tf-slim/2016/12/21/tfrecords-guide/> 글과 https://github.com/tensorflow/models/blob/master/object_detection/g3doc/using_your_own_data_set.md 글을 참고하였다.

TFRecord 파일 생성

TFRecord 파일 생성은 tf.train.Example에 Feature를 딕셔너리 형태로 정의한 후에, tf.train.Example 객체를 TFRecord 파일 포맷 Writer인 tf.python_io.TFRecordWriter를 통해서 파일로 저장하면 된다.

다음 코드를 보자, 이 코드는 Tensorflow Object Detection API를 자신의 데이터로 학습시키기 위해서 데이터를 TFRecord 형태로 변환하여 저장하는 코드의 내용이다.
이미지를 저장할 때 사물의 위치를 사각형 좌표로 지정하고 저장한다.

```

def create_cat_tf_example(encoded_image_data):

    height = 1032

```

```

width = 1200
filename = 'example_cat.jpg'
image_format = 'jpg'

xmins = [322.0 / 1200.0]
xmaxs = [1062.0 / 1200.0]
ymins = [174.0 / 1032.0]
ymaxs = [761.0 / 1032.0]
classes_text = ['Cat']
classes = [1]

tf_example = tf.train.Example(features=tf.train.Features(feature={
    'image/height': dataset_util.int64_feature(height),
    'image/width': dataset_util.int64_feature(width),
    'image/filename': dataset_util.bytes_feature(filename),
    'image/source_id': dataset_util.bytes_feature(filename),
    'image/encoded': dataset_util.bytes_feature(encoded_image_data),
    'image/format': dataset_util.bytes_feature(image_format),
    'image/object/bbox/xmin': dataset_util.float_list_feature(xmins),
    'image/object/bbox/xmax': dataset_util.float_list_feature(xmaxs),
    'image/object/bbox/ymin': dataset_util.float_list_feature(ymins),
    'image/object/bbox/ymax': dataset_util.float_list_feature(ymaxs),
    'image/object/class/text': dataset_util.bytes_list_feature(classes_text),
    'image/object/class/label': dataset_util.int64_list_feature(classes),
}))
return tf_example

```

저장되는 내용은 이미지의 높이와 너비 (height,width), 파일명 (filename), 인코딩 포맷 (format), 이미지 바이너리 (encoded), 이미지내에서 물체의 위치를 가르키는 사각형 위치 (xmin,ymin,xmax,ymax) 와 라벨 값등이 저장된다.

코드를 유심히 살펴보면 생각보다 이해하기어렵지 않다.

tf.train.Example 객체를 만들고 이때 인자로 features에 TFRecord에 저장될 값들의 목록을 딕셔너리 형태로 저장한다.

이때 저장되는 데이터의 이름과 값을 지정해야 하는데

```
'image/height': dataset_util.int64_feature(height),
```

를 보면 'image/height' 이 데이터의 이름이 되고, dataset_util.int64_feature(height), 가 height 값을 텐서플로우용 리스트형으로 변형하여 이를 학습용 피쳐 타입으로 변환하여 저장한다. 이 예제는 Object Detection API의 일부이기 때문에, dataset_util이라는 모듈을 사용했지만, 실제로 이 함수의 내부를 보면 tf.train.Feature(int64_list=tf.train.Int64List(value=values)) 로 구현이 되어 있다.

다음 이렇게 생성된 tf.train.Example 객체를 tf.python_io.TFRecordWriter 를 이용해서 다음과 같이 파일에 써주면 된다.

```
writer = tf.python_io.TFRecordWriter(tfrecored_filename)
```

```
writer.write(tf_example.SerializeToString())
```

다음은 코드 전체이다.

```
import tensorflow as tf
from PIL import Image
from object_detection.utils import dataset_util

def create_cat_tf_example(encoded_image_data):

    height = 1032
    width = 1200
    filename = 'example_cat.jpg'
    image_format = 'jpg'

    xmins = [322.0 / 1200.0]
    xmaxs = [1062.0 / 1200.0]
    ymins = [174.0 / 1032.0]
    ymaxs = [761.0 / 1032.0]
    classes_text = ['Cat']
    classes = [1]

    tf_example = tf.train.Example(features=tf.train.Features(feature={
        'image/height': dataset_util.int64_feature(height),
        'image/width': dataset_util.int64_feature(width),
        'image/filename': dataset_util.bytes_feature(filename),
        'image/source_id': dataset_util.bytes_feature(filename),
        'image/encoded': dataset_util.bytes_feature(encoded_image_data),
        'image/format': dataset_util.bytes_feature(image_format),
        'image/object/bbox/xmin': dataset_util.float_list_feature(xmins),
        'image/object/bbox/xmax': dataset_util.float_list_feature(xmaxs),
        'image/object/bbox/ymin': dataset_util.float_list_feature(ymins),
        'image/object/bbox/ymax': dataset_util.float_list_feature(ymaxs),
        'image/object/class/text': dataset_util.bytes_list_feature(classes_text),
        'image/object/class/label': dataset_util.int64_list_feature(classes),
    }))
    return tf_example

def read_imagebytes(imagefile):
    file = open(imagefile, 'rb')
    bytes = file.read()

    return bytes

def main():
    print('Converting example_cat.jpg to example_cat.tfrecord')
    tfrecord_filename = 'example_cat.tfrecord'
    bytes = read_imagebytes('example_cat.jpg')
    tf_example = create_cat_tf_example(bytes)
```

```
writer = tf.python_io.TFRecordWriter(tfrecored_filename)
writer.write(tf_example.SerializeToString())

main()
```

참고로 이 예제는 앞서서도 언급하였듯이 Object Detection API에 대한 의존성을 가지고 있기 때문에 일반적인 텐서플로우 개발환경에서는 실행이 되지 않는다. [Tensorflow Object Detection API](#) 를 인스톨해야 dataset_util 를 사용할 수 있기 때문에 Object Detection API 설치가 필요하다. 만약에 Object Detection API 설치 없이 TFRecord Writer를 짜보고 싶은 경우에는 <http://warmspringwinds.github.io/tensorflow/tf-slim/2016/12/21/tfrecoreds-guide/> 문서에 예제가 간단하게 잘 정리되어 있으니 참고하기 바란다.

TFRecord에서 데이터 읽기

데이터를 읽는 방법도 크게 다르지 않다. 쓰는 순서의 반대라고 보면 되는데, TFReader를 통해서 Serialized 된 데이터를 읽고, 이를 Feature 목록을 넣어서 파싱한 후에, 파싱된 데이터셋에서 각 피처를 하나하나 읽으면 된다.

코드를 보자

```
def readRecord(filename_queue):
    reader = tf.TFRecordReader()
    _,serialized_example = reader.read(filename_queue)

    """
    keys_to_features = {
        'image/height': tf.FixedLenFeature([], tf.int64, 1),
        'image/width': tf.FixedLenFeature([], tf.int64, 1),
        'image/filename': tf.FixedLenFeature([], tf.string, default_value=""),
        #'image/key/sha256': tf.FixedLenFeature([], tf.string, default_value=""),
        'image/source_id': tf.FixedLenFeature([], tf.string, default_value=""),
        'image/encoded': tf.FixedLenFeature([], tf.string, default_value=""),
        : (중략)
    }

    features = tf.parse_single_example(serialized_example,features= keys_to_features)

    height = tf.cast(features['image/height'],tf.int64)
    width = tf.cast(features['image/width'],tf.int64)
    filename = tf.cast(features['image/filename'],tf.string)
    source_id = tf.cast(features['image/source_id'],tf.string)
    encoded = tf.cast(features['image/encoded'],tf.string)
    : (중략)

    return height,width,filename,source_id,encoded,image_format
```

TFRecoderReader를 이용하여 파일을 읽는데, 파일을 직접읽지 않고 filename_queue를 이용해서 읽는다. 코드 전체를 보면, 이 큐는

filename_queue = tf.train.string_input_producer([tfrecord_filename])
 로, 파일 이름 목록을 가지고 리턴하는 string_input_producer를 사용하였다.
 파일을 읽으면 데이터는 직렬화 된 상태로 리턴이 되어 serialized_example 에 저장된다.
 이를 각 개별 피쳐로 디코딩 하기 위해서 피쳐 목록을 keys_to_features 딕셔너리에 저장한다.
 이때, 각 피쳐의 이름과, 타입을 정의한다.
 다음은 'image/height' 피쳐를 정의하여 int 64 타입으로 읽어드리는 부분이다.

```
'image/height': tf.FixedLenFeature([], tf.int64, 1),
```

피쳐 목록과, 데이터 타입은 TFRecord 파일을 쓸때 사용한 이름과 데이터 타입을 그대로 사용하면 된다.
 피쳐 목록이 정의되었으면

```
features = tf.parse_single_example(serialized_example, features= keys_to_features)
```

를 통해서 피쳐를 파싱해낸다. 파싱된 피쳐는 features에 딕셔너리 형태로 저장된다.
 다음 리턴 받은 텐서를 각 변수에 저장한다. 이때 타입 캐스팅을 해서 저장한다. (이미 타입을 맞춰서 데이터를 꺼냈기 때문에, 별도의 캐스팅은 필요없지만 확실히 하기 위해서 캐스팅을 한다.)
 다음은 'image/height' 를 피쳐를 배열에서 뽑아서, int64 타입으로 변환하여 height 에 저장하는 부분이다.

```
height = tf.cast(features['image/height'], tf.int64)
```

리턴값은 텐서가 되는데, 이 값을 출력하는 코드를 보자
 당연히 텐서이기 때문에 Session 을 시작해야 하는데, 먼저 파일에서 데이터를 읽기 위해서 filename_queue를 정의한다.

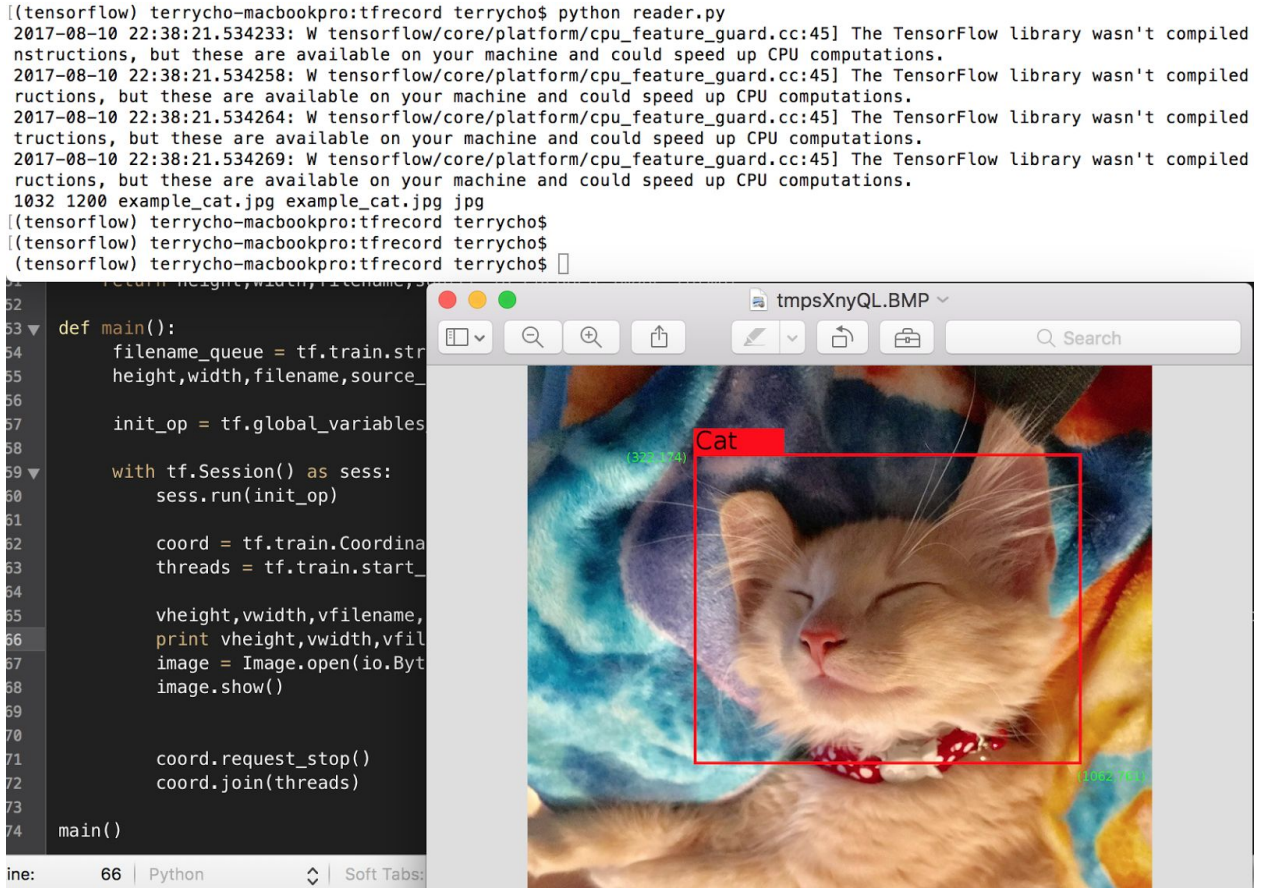
```
filename_queue = tf.train.string_input_producer([tfrecord_filename])
```

다음 filename_queue를 인자로 넘겨서 height,source_id 등의 값을 *.tfrecord 파일에서 읽어서 텐서로 리턴 받는다.

```
height,width,filename,source_id,encoded,image_format = readRecord(filename_queue)
```

다음 세션을 시작하고, readRecord 에서 리턴된 값을 받아온다.
 vheight,vwidth,vfilename,vsource_id,vencoded,vimage_format =
 sess.run([height,width,filename,source_id,encoded,image_format])

마지막으로 받은 값을 화면에 출력한다.



간단하게 tfRecord 파일 포맷으로 학습용 데이터를 쓰는 방법을 알아보았다. 텐서플로우 코드가 간단해 지고 성능에 도움이 되는 만큼 데이터 전처리 단계에서 가급적이면 학습 데이터를 tfrecord 타입으로 바꿔서 학습하는 것을 권장한다. (특히 이미지 데이터!!)

참고 자료

- https://www.tensorflow.org/programmers_guide/reading_data

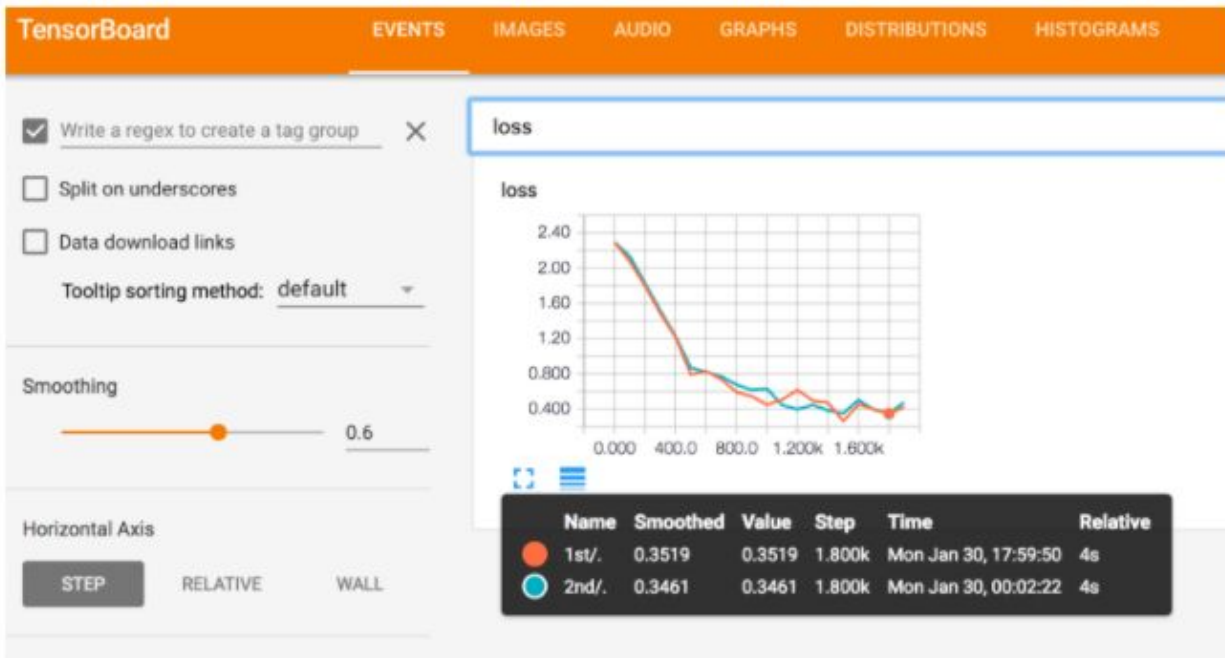
텐서보드를 이용한 학습 과정 시각화

텐서보드를 이용하여 학습 과정을 시각화 해보자

텐서플로우로 머신러닝 모델을 만들어서 학습해보면, 각 인자에 어떤 값들이 학습이 진행되면서 어떻게 변화하는지 모니터링 하기가 어렵다. 앞의 예제들에서는 보통 콘솔에 텍스트로 loss 값이나, accuracy 값을 찍어서, 학습 상황을 봤는데, 텐서보드는 학습에 사용되는 각종 지표들이 어떻게 변화하는지 손쉽게 시각화를 해준다.

예를 들어 보면 다음 그림은 학습을 할때 마다 loss 값이 어떻게 변화하는지를 보여주는 그래프이다.

가로축은 학습 횟수를 세로축은 모델의 loss 값을 나타낸다.



잘 보면 두개의 그래프가 그려져 있는 것을 볼 수 있는데, 1st 그래프는 첫번째 학습, 2nd 는 두번째 학습에서 추출한 loss 값이다.

그러면 어떻게 학습 과정을 시각화할 수 있는지를 알아보자

학습 과정을 시각화 하려면 학습중에 시각화 하려는 데이터를 tf.summary 모듈을 이용해서 중간중간에 파일로 기록해놨다가, 학습이 끝난 후에 이 파일을 텐서 보드를 통해서 읽어서 시각화 한다. 이를 위해서 다음과 같이 크게 4가지 메서드가 주로 사용이 된다.

- `tf.summary.merge_all`
Summary를 사용하기 위해서 초기화 한다.
- `tf.summary.scalar(name,value)`
Summary에 추가할 텐서를 정의 한다. name에는 이름, value에는 텐서를 정의한다.
Scalar 형 텐서로 (즉 다차원 행렬이 아닌, 단일 값을 가지는 텐서형만 사용이 가능하다.)
주로 accuracy나 loss와 같은 스칼라형 텐서에 사용한다.
- `tf.summary.histogram(name,value)`
값(value) 에 대한 분포도를 보고자 할때 사용한다. .scalar와는 다르게 다차원 텐서를 사용할 수 있다. 입력 데이터에 대한 분포도나, Weight, Bias값의 변화를 모니터링할 수 있다.
- `tf.train.SummaryWriter`
파일에 summary 데이터를 쓸때 사용한다.

예제는 <https://www.tensorflow.org/tutorials/mnist/tf/> 를 참고하면 된다.

mnist.py에서 아래와 같이 loss 값을 모니터링 하기 위해서 tf.summary.scalar를 이용하여 'loss'라는 이름으로 loss 텐서를 모니터링하기 위해서 추가하였다.

```
def training(loss, learning_rate):
    """Sets up the training Ops.

    Creates a summarizer to track the loss over time in TensorBoard.

    Creates an optimizer and applies the gradients to all trainable variables.

    The Op returned by this function is what must be passed to the
    `sess.run()` call to cause the model to train.

    Args:
        loss: Loss tensor, from loss().
        learning_rate: The learning rate to use for gradient descent.

    Returns:
        train_op: The Op for training.
    """
    # Add a scalar summary for the snapshot loss.
    tf.summary.scalar('loss', loss)
```

다음 fully_connected_feed.py에서

```
# Build the summary Tensor based on the TF collection of Summaries.
summary = tf.summary.merge_all()
```

Summary를 초기화 하고, 세션이 시작된 후에, summary_writer를 아래와 같이 초기화 하였다.

```
# Create a session for running Ops on the Graph.
sess = tf.Session()

# Instantiate a SummaryWriter to output summaries and the Graph.
summary_writer = tf.summary.FileWriter(FLAGS.log_dir, sess.graph)
```

이때, 파일 경로 (FLAGS.log_dir)을 설정하고, 텐서 플로우의 세션 그래프(sess.graph)를 인자로 넘긴다.

```
...# Write the summaries and print an overview fairly often.
...if step % 100 == 0:
    # Print status to stdout.
    print('Step %d: loss = %.2f (%.3f sec)' % (step, loss_value, duration))
    # Update the events file.
    summary_str = sess.run(summary, feed_dict=feed_dict)

    #print('Summary str: %s' % summary_str)
    summary_writer.add_summary(summary_str, step)
    summary_writer.flush()
```

다음 트레이닝 과정에서, 100번마다, summary 값을 문자열로 변환하여, summary_writer를 이용하여 파일에 저장하였다.

트레이닝이 끝나면 위에서 지정된 디렉토리에 아래와 같이 summary 데이터 파일이 생성 된다.

```
[terrycho-macbookpro:~ terrycho$ ls /tmp/tensorflow/mnist/logs/fully_connected_feed
checkpoint
events.out.tfevents.1485769342.terrycho-macbookpro.roam.corp.google.com
model.ckpt-1999.data-00000-of-00001
model.ckpt-1999.index
model.ckpt-1999.meta
model.ckpt-999.data-00000-of-00001
model.ckpt-999.index
model.ckpt-999.meta
```

이를 시각화 하려면 콘솔에서 tensorboard --logdir="Summary 파일 디렉토리 경로" 를 지정해주면 6060 포트로 텐서보드 웹 사이트가 준비된다.

```
[terrycho-macbookpro:~ terrycho$ tensorboard --logdir=/tmp/tensorflow/mnist/logs/fully_connected_feed
Starting TensorBoard 2.9 on port 6006
```

웹 브라우저를 열어서 localhost:6060에 접속해보면 다음과 같은 그림이 나온다.



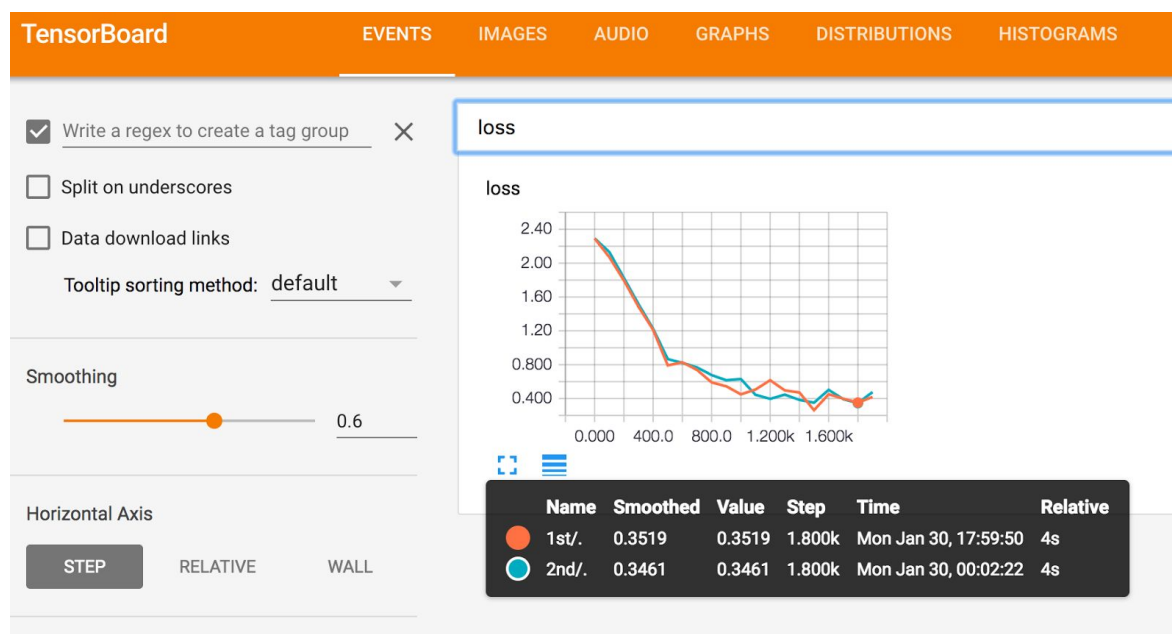
Loss 값이 트레이닝이 수행됨에 따라 작아 지는 것을 볼 수 있다. (총 2000번 트레이닝을 하였다.)

세로축은 loss 값, 가로축은 학습 스텝이 된다.

만약에 여러번 학습을 하면서 모델을 튜닝했다면, 각 학습 별로 loss 값이나 accuracy 값이 어떻게 변하는지 그래프를 중첩하여 비교하고 싶을 수 있는데, 이 경우에는

% tensorboard --logdir=이름1:로그경로2,이름2:로그경로2,....

이런식으로 “이름:로그경로”를 ,로 구분하여 여러개를 써주면 그래프를 중첩하여 볼 수 있다. 아래는 1st, 2nd 두개의 이름으로 두개의 summary 로그를 중첩하여 시각화하여 각 학습 별로 loss 값이 어떻게 변화 하는지를 보여주는 그래프 이다.



Histogram

히스토그램은 다차원 텐서에 대한 분포를 볼 수 있는 방법인데, https://github.com/lisourcell/tensorboard_demo 에 히스토그램을 텐서보드로 모니터링할 수 있는 좋은 샘플이 있다. 이 코드는 세개의 히든레이어를 갖는 뉴럴네트워크인데, (사실 좀 코드는 이상하다. Bias 값도 더하지 않았고, 일반 레이어 없이 dropout 레이어만 엮었다. 모델 자체가 맞는지 틀리는지는 따지지 말고 어떻게 Histogram을 모니터링 하는지를 살펴보자)

모델 그래프는 다음과 같다.

```

7 # This network is the same as the previous one except with an extra hidden layer + dropout
8 def model(X, w_h, w_h2, w_o, p_keep_input, p_keep_hidden):
9     # Add layer name scopes for better graph visualization
10    with tf.name_scope("layer1"):
11        X = tf.nn.dropout(X, p_keep_input)
12        h = tf.nn.relu(tf.matmul(X, w_h))
13    with tf.name_scope("layer2"):
14        h = tf.nn.dropout(h, p_keep_hidden)
15        h2 = tf.nn.relu(tf.matmul(h, w_h2))
16    with tf.name_scope("layer3"):
17        h2 = tf.nn.dropout(h2, p_keep_hidden)
18    return tf.matmul(h2, w_o)

```

다음, 각 레이어에서 사용된 weight 값인 w_h, w_{h2}, w_o 를 모니터링 하기 위해서 이 텐서들을 `tf.histogram_summary`를 이용하여 summary에 저장 한다.

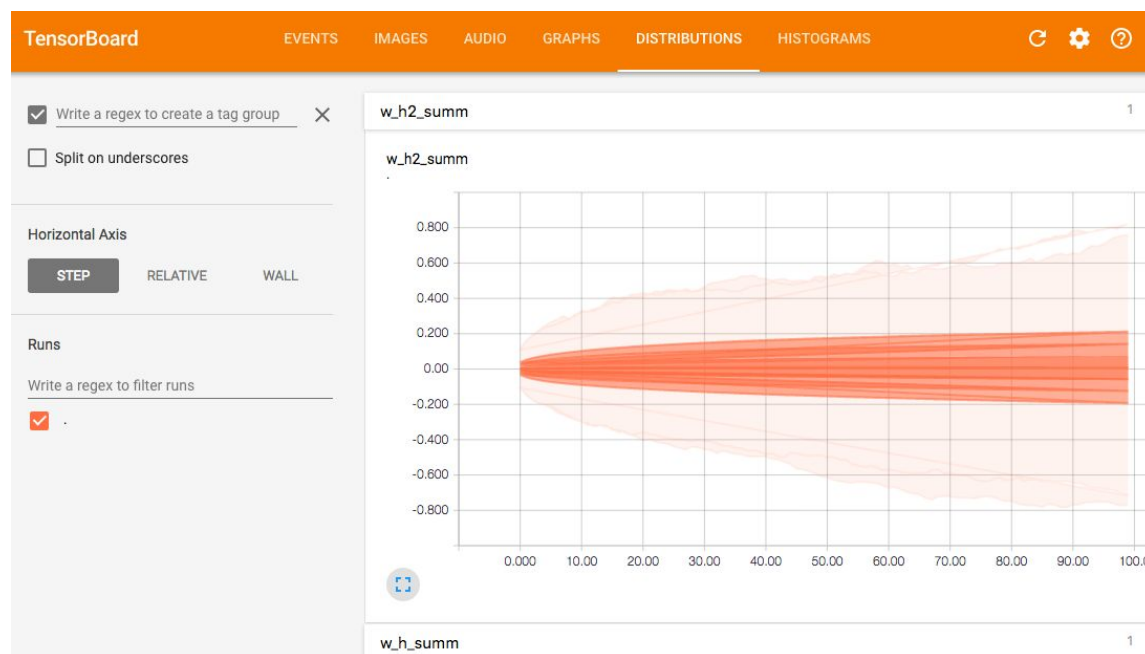
```

33 #Step 4 - Add histogram summaries for weights
34 tf.histogram_summary("w_h_summ", w_h)
35 tf.histogram_summary("w_h2_summ", w_h2)
36 tf.histogram_summary("w_o_summ", w_o)

```

이렇게 저장된 데이터를 텐서 보드로 시각화 해보면

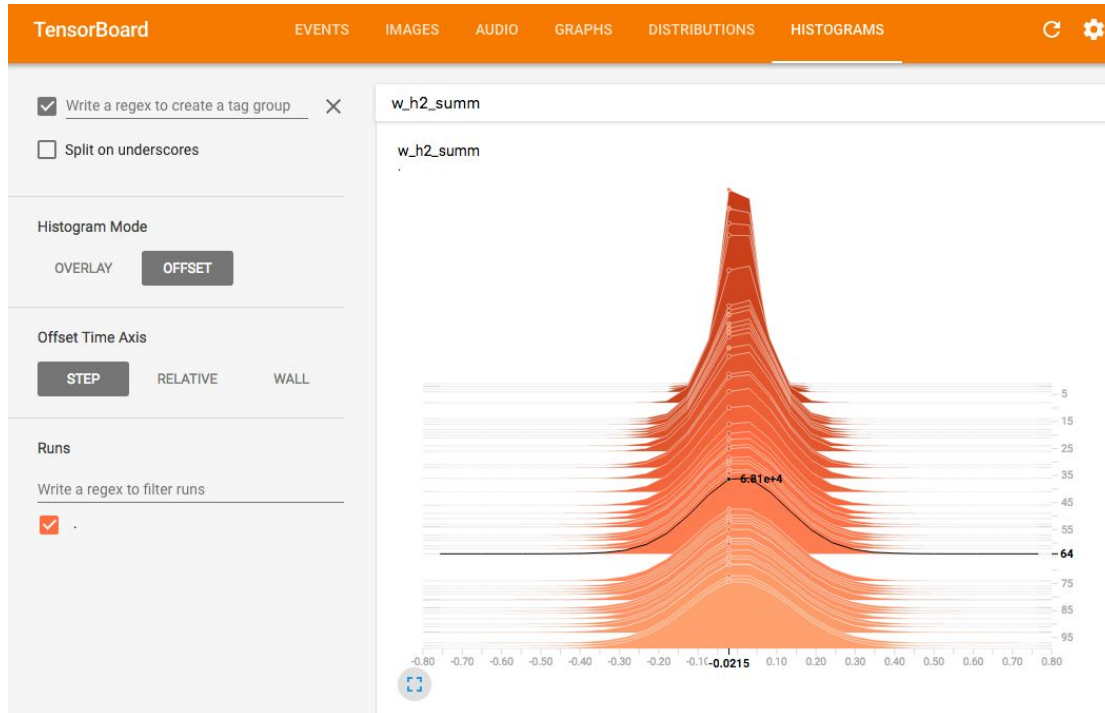
Distribution 탭에서는 다음과 같은 값을 볼 수 있다.



w_h_summ 값의 분포인데, 세로 축은 w 의 값, 가로축은 학습 횟수 이다.

학습이 시작되는 초기에는 w값이 0을 중심으로 좌우 대칭으로 모여 있는 것을 볼 수 있다. 잘 보면, 선이 있는 것을 볼 수 있는데, 색이 진할 수 록, 값이 많이 모여 있는 것이고 흐릴 수 록 값이 적게 있는 것이다.

다른 뷰로는 Histogram View를 보면, 다음과 같은 그래프를 볼 수 있는데,



세로축이 학습 횟수, 가로축이 Weight의 값이다. 그래프가 여러개가 중첩 되어 있는 것을 볼 수 있는데, 각각의 그래프는 각 학습시에 나온 Weight의 값으로, 위의 그래프에서 보면 중앙에 값이 집중되어 있다가, 아래 그래프를 보면 값이 점차적으로 옆으로 퍼지는 것을 볼 수 있다.

9. 컨볼루셔널 네트워크를 이용하여 얼굴인식 모델을 만들어 보자

CNN 모델을 이용해서 얼굴을 인식하는 모델을 개발하는 강좌를 시작하고자 한다. 첫번째는 얼굴 인식 모델을 학습 시키기 위한 학습 데이터 수집 및 정제 방법이다. 이전글 <http://bcho.tistory.com/1166> 을 개선한 내용이다.

얼굴 데이터 수집 하기

먼저 얼굴 데이터를 수집해야 하는데, 얼굴 데이터의 경우에는 개인 초상권과 사진에 대한 저작권이 걸려있기 때문에 수집하기가 쉽지 않다. 그래서 얼굴 사진 공유 사이트를 사용하였다. PubFig (Public Figures Face Database - <http://www.cs.columbia.edu/CAVE/databases/pubfig/>) 를 사용하였다.

PubFig: Public Figures Face Database

[Explore](#) [Download](#) [Results](#)



Introduction

The PubFig database is a large, real-world face dataset consisting of **58,797** images of **200** people collected from the internet. Unlike most other existing face datasets, these images are taken in completely uncontrolled situations with non-cooperative subjects. Thus, there is large variation in pose, lighting, expression, scene, camera, imaging conditions and parameters, etc. The PubFig dataset is similar in spirit to the [Labeled Faces in the Wild \(LFW\) dataset](#) created at UMass-Amherst, although there are some significant differences in the two:

이 데이터셋에는 약 200명에 대한 58,000여장의 이미지를 저장하고 있는데, 이 중의 일부만을 사용하였다.

Download 페이지로 가면, txt 파일 형태

(http://www.cs.columbia.edu/CAVE/databases/pubfig/download/dev_urls.txt) 로 아래와 같이

```
Abhishek Bachan      1
http://1.bp.blogspot.com/_Y7rzCyUABel/SNIltEyEnjI/AAAAAAAAABOg/E1keU_52aFc/s400/ash_
abhishek_365x470.jpg      183,60,297,174      f533da9fbd1c770428c8961f3fa48950
Abhishek Bachan      2
http://1.bp.blogspot.com/_v9nTKD7D57Q/SQ3HUQHsp_I/AAAAAAAAQuo/DfPcHPX2t_o/s400/
normal_14thbombaytimes013.jpg      49,71,143,165 e36a8b24f0761ec75bdc0489d8fd570b
Abhishek Bachan      3
http://2.bp.blogspot.com/_v9nTKD7D57Q/SL5KwcwQIRI/AAAAAAAAANxM/mJPzEHP11rU/s400/
ERTYH.jpg      32,68,142,178 583608783525c2ac419b41e538a6925d
```

사람이름, 이미지 번호, 다운로드 URL, 사진 크기, MD5 체크섬을 이 필드로 저장되어 있다. 이 파일을 이용하여 다운로드 URL에서 사진을 다운받아서, 사람이름으로된 폴더에 저장한다. 물론 수동으로 할 수 없으니 HTTP Client를 이용하여, URL에서 사진을 다운로드 하게 하고, 이를 사람이름 폴더 별로 저장하도록 해야 한다.

HTTP Client를 이용하여 파일을 다운로드 받는 코드는 일반적인 코드이기 때문에 별도로 설명하지 않는다.

본인의 경우에는 Win이 만든

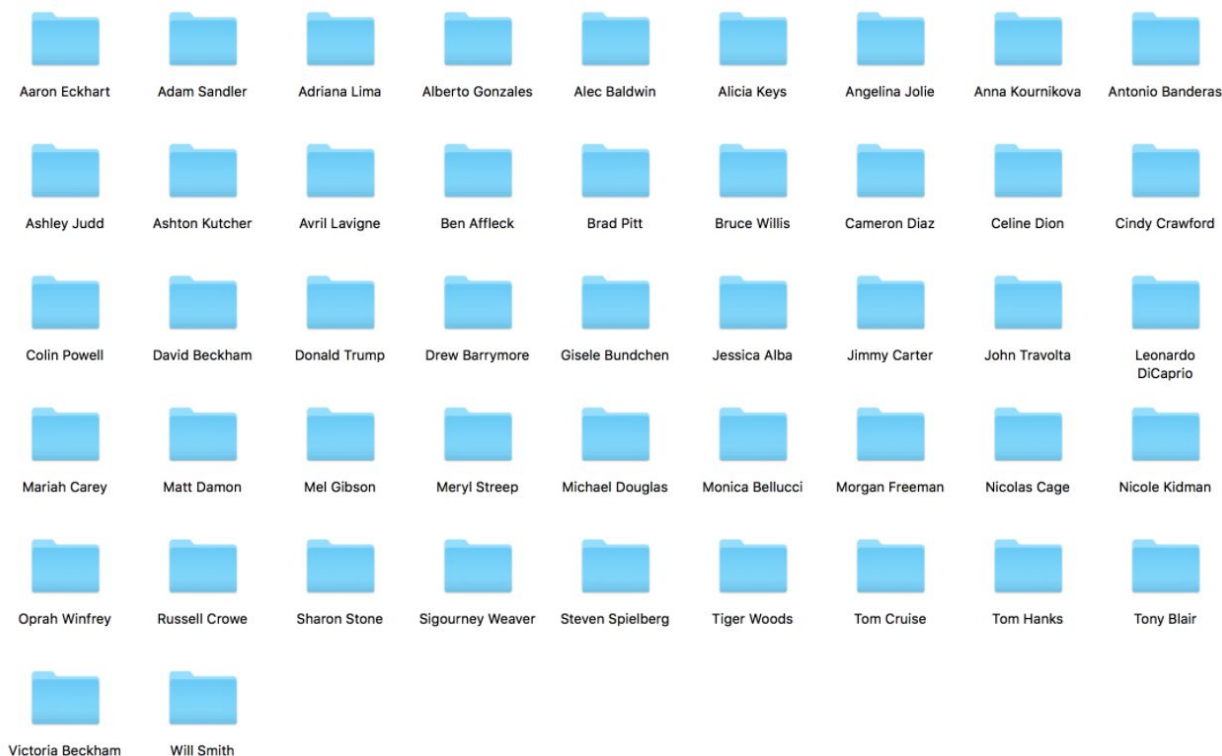
https://github.com/wwoo/tf_face/blob/master/tf/face_extract/pubfig_get.py 코드를 이용하여 데이터를 다운로드 받았다.

사용법은 https://github.com/wwoo/tf_face 에 나와 있는데,

```
$> python tf/face_extract/pubfig_get.py tf/face_extract/eval_urls.txt ./data
```

를 실행하면 ./data 디렉토리에 이미지를 다운로드 받아서 사람 이름별 폴더에 저장해준다.

evals_urls.txt에는 위에서 언급한 dev_urls.txt 형태의 데이터가 들어간다.



사람 종류가 너무 많으면 데이터를 정제하는 작업이 어렵고, (왜 어려운지는 뒤에 나옴) 학습 시간이 많이 걸리기 때문에, 최종적으로 4 명의 데이터를 다운로드 받아서 작업하였다.

데이터 정제 하기

학습을 제대로 시키기 위해서는 앞에서 다운 받은 데이터를 다음 규칙에 따라서 정제한다.

- 얼굴이 제대로 나타난 사진만 골라낸다.
두명 이상의 얼굴이 있는 경우 다른 사람의 얼굴이 추출 될 수 있기 때문에 제외 하고



또는 얼굴이 없는 사진은 학습에 사용할 수 없기 때문에 (404 Not found 이미지와 같은) 이를 제외 한다.



THIS IMAGE OR VIDEO
HAS BEEN
MOVED OR DELETED

- 7:3의 비율로 학습용 데이터와 검증용 데이터를 분류한다

얼굴 추출 프로그램 사용 방법

그러면 학습용으로 수집된 데이터에서 얼굴만 추출해서 정제하는 프로그램을 작성해보도록 하자
전체 코드는

https://github.com/bwcho75/facerecognition/blob/master/com/terry/face/extract/crop_face.py 에
있으며

사용 방법은 crop_face.py 를 "{적절한 디렉토리}/com/terry/face/extract/" 에 복사해놓고
"{적절한 디렉토리}" 에서

```
%python com/terry/face/extract/crop_face.py "정제할 데이터가 있는 디렉토리" "결과 디렉토리"  
"학습 데이터의 개수"
```

식으로 사용한다.

"정제할 데이터가 있는 디렉토리"의 구조는 디렉토리 안에, 학습 데이터가 각각 사람 이름 폴더
안에 들어가 있으면 된다.

예를 들어 브래드피트 사진은 "브래드피트"라는 폴더 안에 들어가 있으면 된다.

마지막으로 "학습데이터의 개수"는, 추출할 학습 데이터의 수이다.

예를 들어 브래드피트 사진이 120장 있고, 안젤리나 졸리 사진이 100장 있을때, 추출할 데이터
수를 70 장으로 주면, 학습 데이터를 70장만 추출하고 종료한다.

이 기능을 추가한 이유는 학습에서 특정 라벨 (브래드피트, 안젤리나 졸리)에 데이터가 많은 경우 학습이 치우쳐져서 학습데이터가 많이 들어간 쪽으로 결과를 판명하는 경우가 많다. 그래서 학습시에는 학습 데이터를 라벨(사람)마다 동등하게 해야 하는데, 수집된 데이터가 라벨마다 그 수가 일정하지 않기 때문에 정제 단계에서 수동으로 데이터 수를 맞추기 위해서 (가장 적은 데이터 수에 맞춘다) 이 기능이 추가 되었다.

사용 예는 다음과 같다.

```
% python com/terrycho/face/extract/crop_face.py /Users/terrycho/Desktop/TrainingData10class  
~/training_data_class5 120
```

/Users/terrycho/Desktop/TrainingData10class 에 있는 데이터를 정제해서
~/training_data_class5 에 저장하고, 이때 학습용 데이터를 라벨(사람)당 120개씩만 추출해서
저장하도록 한다.

결과 파일 및 디렉토리 구조

얼굴 추출 프로그램은 지정된 정제할 데이터가 있는 디렉토리에 있는 이미지들을 읽어서, 구글 클라우드 vision API를 이용하여 얼굴이 하나만 있는지를 확인하고 얼굴이 하나만 있으면, 그 얼굴을 잘라낸 후 96x96 사이즈로 리사이즈 한후에 결과 디렉토리 아래 /training 폴더에 학습용 이미지 파일들을 저장하고 결과 디렉토리 폴더 아래 training.txt로 학습용 이미지 파일 목록과 라벨을 저장한다.

training.txt 파일의 포맷은 다음과 같다.

“,”로 분리되는 CSV 파일 포맷으로 {파일명},{사람 이름},{라벨}
형태이며 다음과 같이 저장된다.

```
/Users/terrycho/training_data_class5/training/007df_Angelina_Jolie__Esquir5.jpg,Angelina Jolie,0  
/Users/terrycho/training_data_class5/training/00998_00999_Angelina_Jolie_036.jpg,Angelina Jolie,0  
/Users/terrycho/training_data_class5/training/00999_Angelina_Jolie_0091.jpg,Angelina Jolie,0  
/Users/terrycho/training_data_class5/training/0114.jpg,Angelina Jolie,0
```

라벨은 사람마다 순서대로 0,1,2,3,...,N 식의 정수를 부여한다.

마찬가지로 검증용 데이터도 결과 디렉토리 아래 /validate 라는 폴더에 검증용 이미지 파일을
저장하고 검증용 이미지 목록을 validate.txt 파일로 저장한다.

구글 클라우드 VISION API 사용 준비

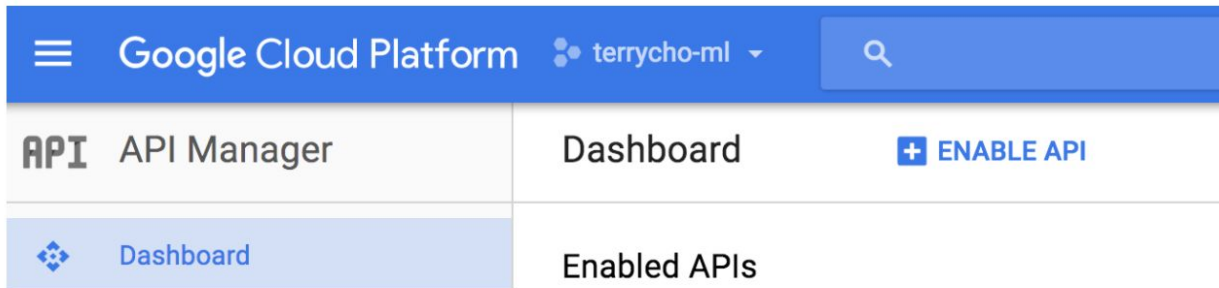
앞에서 설명한바와 같이 데이터를 정제하는 프로그램에서는

- 사진상의 얼굴의 개수
- 얼굴의 위치

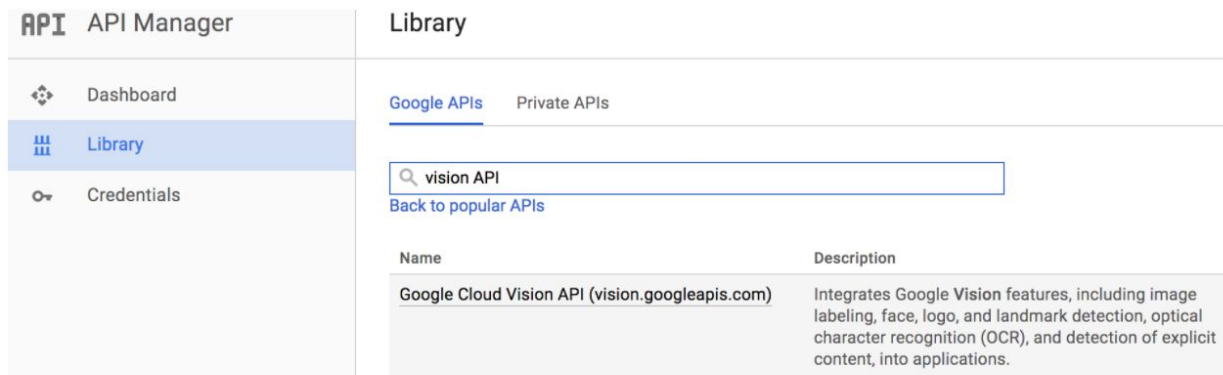
를 추출하기 위해서 구글 클라우드 VISION API를 사용한다.

이 VISION API를 사용하기 위해서는 해당 구글 클라우드 프로젝트에서 VISION API를
사용하도록 ENABLE 해줘야 한다.

VISION API를 ENABLE하기 위해서는 아래 화면과 같이 구글 클라우드 콘솔 > API Manager
들어간후



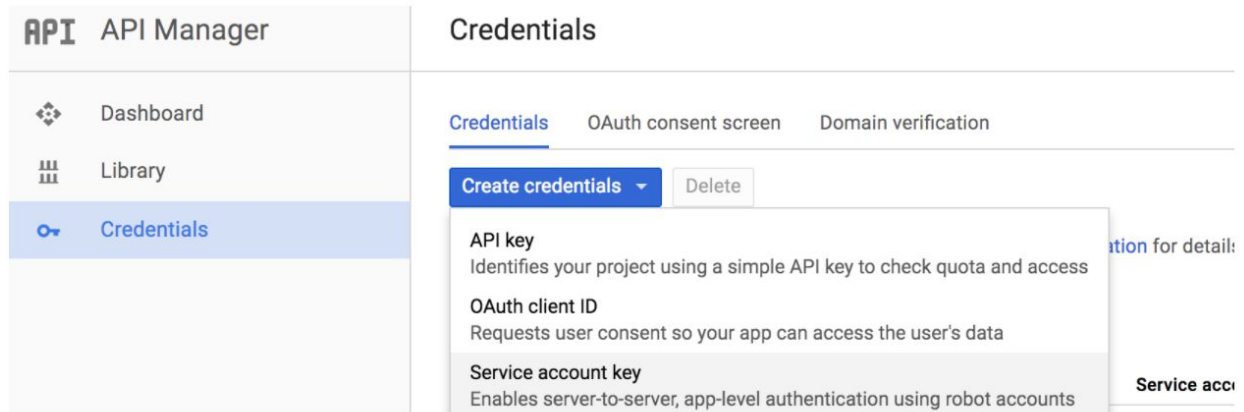
+ENABLE API를 클릭하여 아래 그림과 같이 Vision API를 클릭하여 ENABLE 시켜준다.



SERVICE ACCOUNT 키 만들기

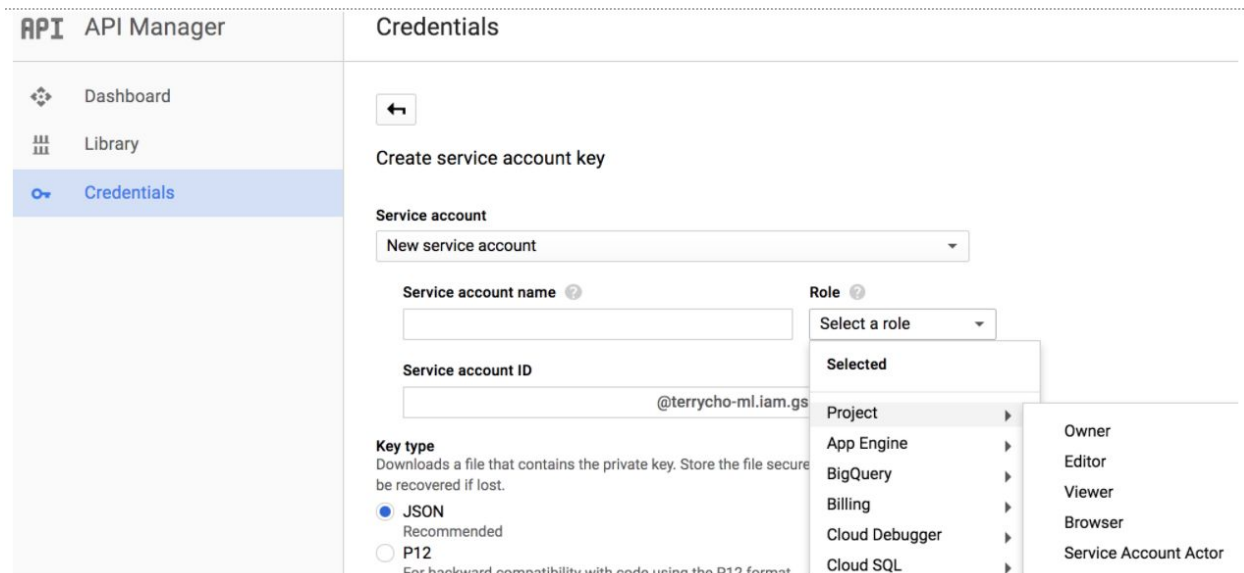
다음으로 이 VISION API를 호출하기 위해서는 API 토큰이 필요한데, SERVICE ACCOUNT 라는 JSON 파일을 다운 받아서 사용한다.

구글 클라우드 콘솔에서 API Manager로 들어간후 Credentials 메뉴에서 Create credential 메뉴를 선택한후, Service account key 메뉴를 선택한다



다음 Create Service Account key를 만들도록 하고, accountname과 id와 같은 정보를 넣는다. 이때 중요한것이 이 키가 가지고 있는 사용자 권한을 설정해야 하는데, 편의상 모든 권한을 가지고 있는 Project Owner 권한으로 키를 생성한다.

(주의. 실제 운영환경에서 전체 권한을 가지는 키는 보안상의 위험하기 때문에 특정 서비스에 대한 접근 권한만을 가지도록 지정하여 Service account를 생성하기를 권장한다.)



Service account key가 생성이 되면, json 파일 형태로 다운로드가 된다. 여기서는 terrycho-ml-80abc460730c.json 이름으로 저장하였다. 이 파일은 나중에 데이터 정제 코드에서 사용하기 때문에 잘 저장해놓도록 하자.

예제 코드

코드 흐름은 다음과 같다.

```
crop_face_rootdir(src_dir,des_dir,maxnum)
```

#src_dir에서 디렉토리를 하나씩 읽어서 crop_face_dir을 실행한다.

```
crop_face_dir
```

#각 디렉토리에서 이미지 파일을 하나씩 읽는다.

```
detect_face
```

#읽어드린 이미지 파일에서, Cloud Vision API를 호출하여 얼굴 영역의 좌표를 읽어온다.

#만약에 얼굴이 없거나 또는 얼굴이 2개 이상인 경우에는 None을 리턴한다.

```
crop_face
```

#앞에서 읽어온 얼굴 영역의 좌표로 이미지에서 얼굴 영역만 잘라낸다.

#잘라낸 얼굴을 96x96 사이즈의 크기로 리사이징한다.

#리사이징된 파일을 des_dir에 저장한다.

전체 코드는 다음과 같다.

```
from googleapiclient import discovery
from oauth2client.client import GoogleCredentials
import sys
import io
import base64
from PIL import Image
from PIL import ImageDraw
from genericpath import isfile
import os
from oauth2client.service_account import ServiceAccountCredentials
```

```
NUM_THREADS = 10
MAX_RESULTS = 2
IMAGE_SIZE = 96,96
```

```
# index to transfrom image string label to number
global_label_index = 0
```

```
class FaceDetector():
    def __init__(self):
        # initialize library
        #credentials = GoogleCredentials.get_application_default()
        scopes = ['https://www.googleapis.com/auth/cloud-platform']
        credentials = ServiceAccountCredentials.from_json_keyfile_name(
            './terrycho-ml-80abc460730c.json', scopes=scopes)
        self.service = discovery.build('vision', 'v1', credentials=credentials)
        #print ("Getting vision API client : %s" ,self.service)
```

```
#def extract_face(self,image_file,output_file):
```

```
def detect_face(self,image_file):
    try:
        with io.open(image_file,'rb') as fd:
            image = fd.read()
            batch_request = [{
```

```

        'image':{
            'content':base64.b64encode(image).decode('utf-8')
        },
        'features':[{
            'type':FACE_DETECTION,
            'maxResults':MAX_RESULTS,
        }]
    }]
    fd.close()

request = self.service.images().annotate(body={
    'requests':batch_request, })
response = request.execute()
if 'faceAnnotations' not in response['responses'][0]:
    print('[Error] %s: Cannot find face ' % image_file)
    return None

face = response['responses'][0]['faceAnnotations']

if len(face) > 1 :
    print('[Error] %s: It has more than 2 faces in a file' % image_file)
    return None

box = face[0]['fdBoundingPoly']['vertices']
left = box[0]['x']
top = box[1]['y']

right = box[2]['x']
bottom = box[2]['y']

rect = [left,top,right,bottom]

print("[Info] %s: Find face from in position %s" % (image_file,rect))
return rect
except Exception as e:
    print('[Error] %s: cannot process file : %s' %(image_file,str(e)) )

def rect_face(self,image_file,rect,outputfile):
    try:
        fd = io.open(image_file,'rb')
        image = Image.open(fd)
        draw = ImageDraw.Draw(image)
        draw.rectangle(rect,fill=None,outline="green")
        image.save(outputfile)
        fd.close()
        print("[Info] %s: Mark face with Rect %s and write it to file : %s' %(image_file,rect,outputfile) )
    except Exception as e:
        print('[Error] %s: Rect image writing error : %s' %(image_file,str(e)) )

def crop_face(self,image_file,rect,outputfile):
    try:
        fd = io.open(image_file,'rb')
        image = Image.open(fd)
        crop = image.crop(rect)
        im = crop.resize(IMAGE_SIZE,Image.ANTIALIAS)
        im.save(outputfile,"JPEG")
        fd.close()
        print("[Info] %s: Crop face %s and write it to file : %s' %(image_file,rect,outputfile) )
    except Exception as e:
        print('[Error] %s: Crop image writing error : %s' %(image_file,str(e)) )

def getfiles(self,src_dir):
    files = []
    for f in os.listdir(src_dir):
        if isfile(os.path.join(src_dir,f)):
            if not f.startswith('.'):

```

```

        files.append(os.path.join(src_dir,f))

    return files

# read files in src_dir and generate image that rectangle in face and write into files in des_dir
def rect_faces_dir(self,src_dir,des_dir):
    if not os.path.exists(des_dir):
        os.makedirs(des_dir)

    files = self.getfiles(src_dir)
    for f in files:
        des_file = os.path.join(des_dir,os.path.basename(f))
        rect = self.detect_face(f)
        if rect != None:
            self.rect_face(f, rect, des_file)

# read files in src_dir and crop face only and write it into des_dir
def crop_faces_dir(self,src_dir,des_dir,maxnum):

    # training data will be written in $des_dir/training
    # validation data will be written in $des_dir/validate

    des_dir_training = os.path.join(des_dir,'training')
    des_dir_validate = os.path.join(des_dir,'validate')

    if not os.path.exists(des_dir):
        os.makedirs(des_dir)
    if not os.path.exists(des_dir_training):
        os.makedirs(des_dir_training)
    if not os.path.exists(des_dir_validate):
        os.makedirs(des_dir_validate)

    path,folder_name = os.path.split(src_dir)
    label = folder_name

    # create label file. it will contains file location
    # and label for each file
    training_file = open(des_dir+'/training_file.txt','a')
    validate_file = open(des_dir+'/validate_file.txt','a')

    files = self.getfiles(src_dir)
    cnt = 0
    num = 0 # number of training data
    for f in files:
        rect = self.detect_face(f)

        # replace ',' in file name to '.'
        # because ',' is used for deliminators of image file name and its label
        des_file_name = os.path.basename(f)
        des_file_name = des_file_name.replace(',','_')

        if rect != None:
            # 70% of file will be stored in training data directory
            if(cnt < 8):
                des_file = os.path.join(des_dir_training,des_file_name)
                self.crop_face(f, rect, des_file )
                training_file.write("%s,%s,%d\n"%(des_file,label,global_label_index) )
                num = num + 1
                if (num>maxnum):
                    break
            # 30% of files will be stored in validation data directory
            else: # for validation data
                des_file = os.path.join(des_dir_validate,des_file_name)
                self.crop_face(f, rect, des_file)
                validate_file.write("%s,%s,%d\n"%(des_file,label,global_label_index) )

```

```

        if(cnt>9):
            cnt = 0
            cnt = cnt + 1
        #increase index for image label
        global global_label_index
        global_label_index = global_label_index + 1

    training_file.close()
    validate_file.close()

def getdirs(self,dir):
    dirs = []
    for f in os.listdir(dir):
        f=os.path.join(dir,f)
        if os.path.isdir(f):
            if not f.startswith('.'):
                dirs.append(f)

    return dirs

def crop_faces_rootdir(self,src_dir,des_dir,maxnum):
    # crop file from sub-directoris in src_dir
    dirs = self.getdirs(src_dir)

    #list sub directory
    for d in dirs:
        print('[INFO] : ### Starting cropping in directory %s ###'%d)
        self.crop_faces_dir(d, des_dir,maxnum)
    #loop and run face crop

#usage
# arg[1] : src directory
# arg[2] : destination diectory
# arg[3] : max number of samples per class
def main(argv):
    srcdir= argv[1]
    desdir = argv[2]
    maxnum = int(argv[3])

    detector = FaceDetector()

    detector.crop_faces_rootdir(srcdir, desdir,maxnum)
    #detector.crop_faces_dir(inputfile,outputfile)
    #rect = detector.detect_face(inputfile)
    #detector.rect_image(inputfile, rect, outputfile)
    #detector.crop_face(inputfile, rect, outputfile)

if __name__ == "__main__":
    main(sys.argv)

```

그러면 코드를 자세하게 살펴보자

```

credentials = ServiceAccountCredentials.from_json_keyfile_name(
    './terrycho-ml-80abc460730c.json', scopes=scopes)

```

부분은 Google Cloud Vision API를 호출하기 위한 API 토큰 파일을 지정하는 부분으로 여기서는 “terrycho-ml-80abc460730c.json” 파일로 지정하였다. 앞의 “구글 클라우드 VISION API 사용준비”에서 생성한 API 키 파일명을 여기에 기술하면 된다.

detect_face

이미지에서 얼굴을 찾아서 영역을 리턴해주는 함수이다.

```
with io.open(image_file, 'rb') as fd:
    image = fd.read()
    batch_request = [{
        'image': {
            'content': base64.b64encode(image).decode('utf-8')
        },
        'features': [{
            'type': 'FACE_DETECTION',
            'maxResults': MAX_RESULTS,
        }]
    }]
    fd.close()

request = self.service.images().annotate(body={
    'requests': batch_request, })
response = request.execute()
```

먼저 image_file을 읽은 후에, Vision API를 호출하기 위해서 request를 만든다. request 포맷은 JSON 형태로 image 엘리먼트에, 이미지 데이터를 base64 인코딩 형태로 인코딩을 해서 전송한다. features 엘리먼트에는 Google Cloud Vision API에서 추출할 특징에 대해서 정의하는데, type을 “FACE_DETECTION”으로 하고 한번에 추출되는 최대 얼굴 수를 지정하였다. (여기서는 2개로 지정하였다.)

```
if 'faceAnnotations' not in response['responses'][0]:
    print('[Error] %s: Cannot find face ' % image_file)
    return None

face = response['responses'][0]['faceAnnotations']

if len(face) > 1 :
    print('[Error] %s: It has more than 2 faces in a file' % image_file)
    return None
```

리턴 값은 JSON으로 오게되는데, “faceAnnotations”에 얼굴 데이터가 오게 된다. 만약에 “faceAnnotation”이 없으면 얼굴이 없는 것으로 해서 None을 리턴하고, “faceAnnotation”이 있더라도, faceAnnotation이 2개 이상이면 에러 처리를 한다.

```
if len(face) > 1 :
    print('[Error] %s: It has more than 2 faces in a file' % image_file)
```

```

        return None

    box = face[0]['fdBoundingPoly']['vertices']
    left = box[0]['x']
    top = box[1]['y']

    right = box[2]['x']
    bottom = box[2]['y']

    rect = [left,top,right,bottom]
    print("[Info] %s: Find face from in position %s" % (image_file,rect))
    return rect

```

다음으로, face 엘리먼트에서 ['fdBoundingPoly']['vertices'] 값을 가지고 오는데, 이 값은 얼굴 위치에 대한 사각형 좌표이다. 이 좌표를 읽어서 rect 리스트에 넣어서 리턴한다.

crop_face

crop_face는 앞의 detect_face에서 받은 얼굴 좌표를 이용하여 얼굴을 잘라내고, 96x96 사이즈로 리사이즈 한후에 파일로 저장하는 함수 이다.

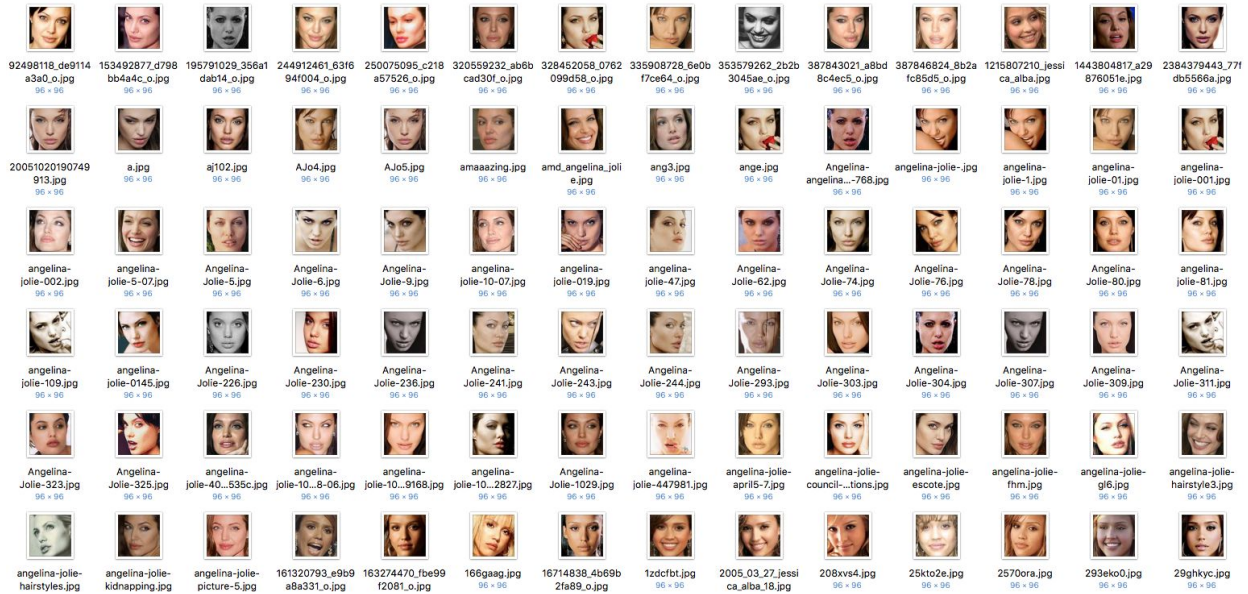
```

def crop_face(self,image_file,rect,outputfile):
    try:
        fd = io.open(image_file,'rb')
        image = Image.open(fd)
        crop = image.crop(rect)
        im = crop.resize(IMAGE_SIZE,Image.ANTIALIAS)
        im.save(outputfile,"JPEG")
        fd.close()
        print("[Info] %s: Crop face %s and write it to file : %s" %(image_file,rect,outputfile) )
    except Exception as e:
        print("[Error] %s: Crop image writing error : %s" %(image_file,str(e)) )

```

애매재 리사이즈와 크롭을 위해서 파이썬 PIL 라이브러리 (파이썬 이미지 라이브러리 - <http://www.pythonware.com/products/pil/>)를 사용하였다.

이렇게 해서 정해진 데이터 셋의 결과는 다음과 같다.



개선점

실습용으로 만든 코드이기 때문에 최적화가 많이되어 있지 않다. 특히나, 이 코드는 싱글 쓰레드로 돌기 때문에 만약에 실제 학습을 한다면, 데이터 정제 시간이 매우 느리기 때문에, 멀티 쓰레드 모델로 개선을 하기를 권장하며, 학습 데이터의 범위를 나눠서 (A~C,D~F,...) 여러대의 서버에 분산해서 처리하는 방안등으로 개선하기를 권장한다.

얼굴 인식 모델을 만들어보자

환경

본 예제는 텐서플로우 1.1과 파이썬 2.7 그리고 Jupyter 노트북 환경 및 구글 클라우드를 사용하여 개발되었다.

준비된 데이터

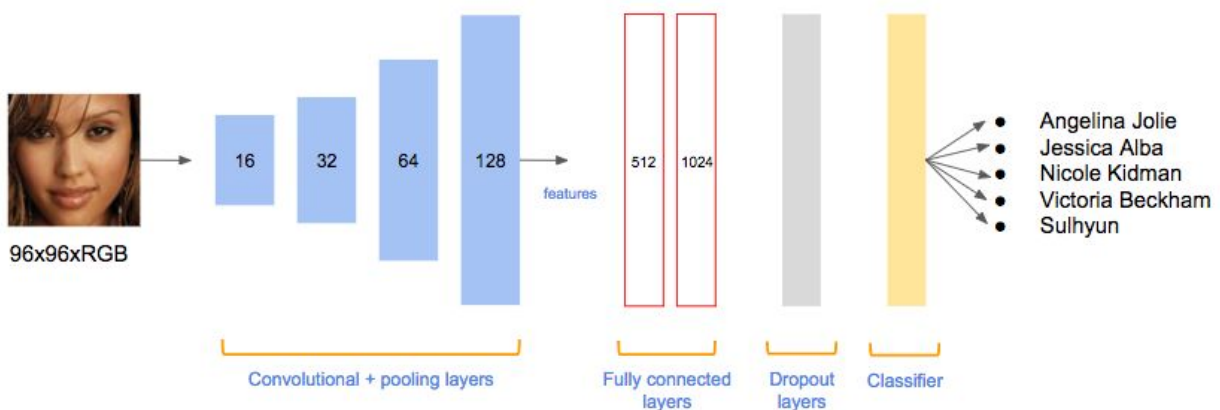
학습에 사용한 데이터는 96x96 사이즈의 얼굴 이미지로, 총 5명의 사진(안젤리나 졸리, 니콜키드만, 제시카 알바, 빅토리아 베컴, 설현)을 이용하였으며, 인당 학습 데이터 40장 테스트 데이터 10장으로 총 250장의 얼굴 이미지를 사용하였다.

사전 데이터를 준비할때, 정면 얼굴을 사용하였으며, 얼굴 각도 변화 폭이 최대한 적은 이미지를 사용하였다. (참고 : <https://www.slideshare.net/Byungwook/ss-76098082>) 만약에 이 모델로 학습이 제대로 되지 않는다면 학습에 사용된 데이터가 적절하지 않은것이기 때문에 데이터를 정제해서 학습하기를 권장한다.

컨볼루셔널 네트워크 모델

얼굴 인식을 위해서, 머신러닝 모델 중 이미지 인식에 탁월한 성능을 보이는 CNN 모델을 사용하였다. 테스트용 모델이기 때문에 모델은 복잡하지 않게 설계하였다.

학습과 예측에 사용되는 이미지는 96x96픽셀의 RGB 컬러 이미지를 사용하였다. 아래 그림과 같은 모델을 사용했는데, 총 4개의 Convolutional 계층과, 2개의 Fully connected 계층, 하나의 Dropout 계층을 사용하였다.



Convolutional 계층의 크기는 각각 16,32,64,128개를 사용하였고, 사용된 Convolutional 필터의 사이즈는 3x3 이다.

Fully connected 계층은 각각 512, 1024를 사용하였고 Dropout 계층에서는 Keep_prob값을 0.7로 뒤서 30%의 뉴론이 drop out 되도록 하여 학습을 진행하였다.

학습 결과 5개의 카테고리에 대해서 총 200장의 이미지로 맥북 프로 i7 CPU 기준 7000 스텝정도의 학습을 진행한 결과 테스트 정확도 기준 90% 정도의 정확도를 얻을 수 있었다.

코드 설명

텐서플로우로 구현된 코드를 살펴보자

파일에서 데이터 읽기

먼저 학습 데이터를 읽어오는 부분이다.

학습과 테스트에서 읽어드리는 데이터의 포맷은 다음과 같다

```
/Users/terrycho/training_data_class5_40/validate/s1.jpg,Sulhyun,3  
이미지 파일 경로, 사람 이름, 숫자 라벨
```

파일에서 데이터를 읽어서 처리 하는 함수는 read_data_batch(), read_data(), get_input_queue() 세가지 함수가 사용된다.

- `get_input_queue()` 함수는 CSV 파일을 한줄씩 읽어서, 파일 경로 및 숫자 라벨 두가지를 리턴할 수 있는 큐를 만들어서 리턴한다.
- `read_data()` 함수는 `get_input_queue()`에서 리턴한 큐로 부터 데이터를 하나씩 읽어서 리턴한다.
- `read_batch_data()`함수는 `read_data()` 함수를 이용하여, 데이터를 읽어서 일정 단위(배치)로 묶어서 리턴을 하고, 그 과정에서 이미지 데이터를 뺄뺄하기 하는 작업을 한다. 즉 호출 구조는 다음과 같다.

```
read_batch_data():
→ Queue = get_input_queue()
→ image,label = read_data(Queue)
→ image_data = 이미지 데이터 뺄뺄하기
Return image_data,label
```

실제 코드를 보자

get_input_queue

`get_input_queue()` 함수는 CSV 파일을 읽어서 image와 labels을 리턴하는 input queue를 만들어서 리턴하는 함수이다.

```
def get_input_queue(csv_file_name,num_epochs = None):
    train_images = []
    train_labels = []
    for line in open(csv_file_name,'r'):
        cols = re.split(',|\n',line)
        train_images.append(cols[0])
        # 3rd column is label and needs to be converted to int type
        train_labels.append(int(cols[2]) )

    input_queue = tf.train.slice_input_producer([train_images,train_labels],
                                                num_epochs = num_epochs,shuffle = True)

    return input_queue
```

CSV 파일을 순차적으로 읽은 후에, `train_images`와 `train_labels`라는 배열에 넣은 다음 `tf.train.slice_input_producer`를 이용하여 큐를 만들어냈다. 이때 중요한 점은 `shuffle=True`라는 옵션을 준것인데, 만약에 이 옵션을 주지 않으면, 학습 데이터를 큐에서 읽을때 CSV에서 읽은 순차적으로 데이터를 리턴한다. 즉 현재 데이터 포맷은 Jessica Alba가 40개, Jolie 가 40개, Nicole Kidman이 40개 .. 식으로 순서대로 들어가 있기 때문에, Jessica Alba를 40개 리턴한 후 Jolie를 40개 리턴하는 식이 된다. 이럴 경우 Convolutional 네트워크가 Jessica Alba에 치우쳐지기 때문에 제대로 학습이 되지 않는다. Shuffle은 필수이다.

read_data()

input_queue에서 데이터를 읽는 부분인데 특이한 점은 input_queue에서 읽어드린 이미지 파일명의 파일을 읽어서 데이터 객체로 저장해야 한다. 텐서플로우에서는 tf.image.decode_jpeg, tf.image.decode_png 등을 이용하여 이러한 기능을 제공한다.

```
def read_data(input_queue):
    image_file = input_queue[0]
    label = input_queue[1]

    image = tf.image.decode_jpeg(tf.read_file(image_file), channels=FLAGS.image_color)

    return image, label, image_file
```

read_data_batch()

마지막으로 read_data_batch() 함수 부분이다. get_input_queue에서 읽은 큐를 가지고 read_data 함수에 넣어서 이미지 데이터와 라벨을 읽어서 리턴하는 값을 받아서 일정 단위로 (배치) 묶어서 리턴하는 함수이다. 중요한 부분이 데이터를 뺏기기 하는 부분이 있다. 이 모델에서 학습 데이터가 클래스당 40개 밖에 되지 않기 때문에 학습데이터가 부족하다. 그래서 여기서 사용한 방법은 read_data에서 리턴된 이미지 데이터에 대해서 tf.image.random_xx 함수를 이용하여 좌우를 바꾸거나, brightness, contrast, hue, saturation 함수를 이용하여 매번 색을 바꿔서 리턴하도록 하였다.

```
def read_data_batch(csv_file_name, batch_size=FLAGS.batch_size):
    input_queue = get_input_queue(csv_file_name)
    image, label, file_name = read_data(input_queue)
    image = tf.reshape(image, [FLAGS.image_size, FLAGS.image_size, FLAGS.image_color])

    # random image
    image = tf.image.random_flip_left_right(image)
    image = tf.image.random_brightness(image, max_delta=0.5)
    image = tf.image.random_contrast(image, lower=0.2, upper=2.0)
    image = tf.image.random_hue(image, max_delta=0.08)
    image = tf.image.random_saturation(image, lower=0.2, upper=2.0)

    batch_image, batch_label, batch_file = tf.train.batch([image, label, file_name], batch_size=batch_size)
    #, enqueue_many=True)
    batch_file = tf.reshape(batch_file, [batch_size, 1])

    batch_label_on_hot = tf.one_hot(tf.to_int64(batch_label),
                                    FLAGS.num_classes, on_value=1.0, off_value=0.0)
    return batch_image, batch_label_on_hot, batch_file
```

그리고 마지막 부분에 label을 tf.one_hot을 이용해서 변환한 것을 볼 수 있는데, 입력된 label은 0, 1, 2, 3, 4 과 같은 단일 정수이다. 그런데, CNN에서 나오는 결과는 정수가 아니라 클래스가 5개인 (분류하는 사람이 5명이기 때문에) 행렬이다. 즉 Jessica Alba 일 가능성이 90%이고, Jolie 일

가능성이 10%이면 결과는 [0.9,0.1,0,0,0] 식으로 리턴이 되기 때문에, 입력된 라벨 0은 [1,0,0,0,0], 라벨 1은 [0,1,0,0,0] 라벨 2는 [0,0,1,0,0] 식으로 변환되어야 한다. tf.one_hot 이라는 함수가 이 기능을 수행해준다.

모델 코드

모델은 앞서 설명했듯이 4개의 Convolutional 계층과, 2개의 Fully connected 계층 그리고 Dropout 계층을 사용한다. 각각의 계층별로는 코드가 다르지 않고 인지만 다르니 하나씩만 설명하도록 한다.

Convolutional 계층

아래 코드는 두번째 Convolutional 계층의 코드이다.

- `FLAGS.conv2_layer_size` 는 이 Convolutional 계층의 뉴런의 수로 32개를 사용한다.
- `FLAGS.conv2_filter_size` 는 필터 사이즈를 지정하는데, 3x3 을 사용한다.
- `FLAGS.stride2 = 1` 는 필터의 이동 속도로 한칸씩 이동하도록 정의했다.

```
# convolutional network layer 2
def conv2(input_data):
    FLAGS.conv2_filter_size = 3
    FLAGS.conv2_layer_size = 32
    FLAGS.stride2 = 1

    with tf.name_scope('conv_2'):
        W_conv2 = tf.Variable(tf.truncated_normal(
            [FLAGS.conv2_filter_size, FLAGS.conv2_filter_size, FLAGS.conv1_layer_size, FLAGS.conv2_layer_size],
            stddev=0.1))
        b2 = tf.Variable(tf.truncated_normal(
            [FLAGS.conv2_layer_size], stddev=0.1))
        h_conv2 = tf.nn.conv2d(input_data, W_conv2, strides=[1, 1, 1, 1], padding='SAME')
        h_conv2_relu = tf.nn.relu(tf.add(h_conv2, b2))
        h_conv2_maxpool = tf.nn.max_pool(h_conv2_relu,
            [1, 2, 2, 1],
            [1, 2, 2, 1], padding='SAME')

    return h_conv2_maxpool
```

다음 Weight 값 `W_conv2` 와 Bias 값 `b2`를 지정한후에, 간단하게 `tf.nn.conv2d` 함수를 이용하면 2차원의 Convolutional 네트워크를 정의해준다. 다음 결과가 나오면 이 결과를 액티베이션 함수인 `relu` 함수에 넣은 후에, 마지막으로 max pooling 을 이용하여 결과를 뽑아낸다.

각 값의 의미에 대해서는 <http://bcho.tistory.com/1149> 의 컨볼루션 네트워크 개념 글을 참고하기 바란다.

같은 방법으로 총 4개의 Convolutional 계층을 중첩한다.

Fully Connected 계층

앞서 정의한 4개의 Convolutional 계층을 통과하면 다음 두개의 Fully Connected 계층을 통과하게 되는데 모양은 다음과 같다.

- `FLAGS.fc1_layer_size = 512` 를 통하여 Fully connected 계층의 뉴런 수를 512개로 지정하였다.

fully connected layer 1

def fc1(input_data):

input_layer_size = 6*6*FLAGS.conv4_layer_size

FLAGS.fc1_layer_size = 512

with tf.name_scope('fc_1'):

앞에서 입력받은 다차원 텐서를 fcc에 넣기 위해서 1차원으로 피는 작업

input_data_reshape = tf.reshape(input_data, [-1, input_layer_size])

W_fc1 = tf.Variable(tf.truncated_normal([input_layer_size, FLAGS.fc1_layer_size], stddev=0.1))

b_fc1 = tf.Variable(tf.truncated_normal([FLAGS.fc1_layer_size], stddev=0.1))

h_fc1 = tf.add(tf.matmul(input_data_reshape, W_fc1), b_fc1) # $h_{fc1} = input_data * W_{fc1} + b_{fc1}$

h_fc1_relu = tf.nn.relu(h_fc1)

return h_fc1_relu

Fully connected 계층은 단순히 $relu(W*x + b)$ 함수이기 때문에 이 함수를 위와 같이 그대로 적용하였다.

마지막 계층

Fully connected 계층을 거쳐 나온 데이터는 Dropout 계층을 거친후에, 5개의 카테고리에 대한 확률로 결과를 내기 위해서 final_out 계층을 거치게 되는데, 이 과정에서 softmax 함수를 사용해야 하나, 학습 과정에서는 별도로 softmax 함수를 사용하지 않는다. softmax는 나온 결과의 합이 1.0이 되도록 값을 변환해주는 것인데, 학습 과정에서는 5개의 결과 값이 어떤 값이 나오던 가장 큰 값에 해당하는 것이 예측된 값이기 때문에, 그 값과 입력된 라벨을 비교하면 되기 때문이다.

즉 예를 들어 Jessica Alba일 확률이 100%면 실제 예측에서는 [1,0,0,0,0] 식으로 결과가 나와야 되지만, 학습 중은 Jessica Alaba 로 예측이 되었다고만 알면 되기 때문에 결과가 [1292,-0.221,-0.221,-0.221] 식으로 나오더라도 최대값만 찾으면 되기 때문에 별도로 softmax 함수를 적용할 필요가 없다. Softmax 함수는 연산 비용이 큰 함수이기 때문에 일반적으로 학습 단계에서는 적용하지 않는다.

마지막 계층의 코드는 다음과 같다.

final layer

def final_out(input_data):

with tf.name_scope('final_out'):

W_fo = tf.Variable(tf.truncated_normal([FLAGS.fc2_layer_size, FLAGS.num_classes], stddev=0.1))

b_fo = tf.Variable(tf.truncated_normal([FLAGS.num_classes], stddev=0.1))


```
h_fo = tf.add(tf.matmul(input_data,W_fo) , b_fo) # h_fc1 = input_data*W_fc1 + b_fc1
```

```
# 최종 레이어에 softmax 함수는 적용하지 않았다.
```

```
return h_fo
```

전체 네트워크 모델 정의

이제 각 CNN의 각 계층을 함수로 정의 하였으면 각 계층을 묶어 보도록 하자. 묶는 법은 간단하다 앞 계층에서 나온 계층을 순서대로 배열하고 앞에서 나온 결과를 뒤의 계층에 넣는 식으로 묶으면 된다.

```
# build cnn_graph
```

```
def build_model(images,keep_prob):
```

```
    # define CNN network graph
```

```
    # output shape will be (*,48,48,16)
```

```
    r_cnn1 = conv1(images) # convolutional layer 1
```

```
    print ("shape after cnn1 ",r_cnn1.get_shape())
```

```
    # output shape will be (*,24,24,32)
```

```
    r_cnn2 = conv2(r_cnn1) # convolutional layer 2
```

```
    print ("shape after cnn2 :",r_cnn2.get_shape() )
```

```
    # output shape will be (*,12,12,64)
```

```
    r_cnn3 = conv3(r_cnn2) # convolutional layer 3
```

```
    print ("shape after cnn3 :",r_cnn3.get_shape() )
```

```
    # output shape will be (*,6,6,128)
```

```
    r_cnn4 = conv4(r_cnn3) # convolutional layer 4
```

```
    print ("shape after cnn4 :",r_cnn4.get_shape() )
```

```
    # fully connected layer 1
```

```
    r_fc1 = fc1(r_cnn4)
```

```
    print ("shape after fc1 :",r_fc1.get_shape() )
```

```
    # fully connected layer2
```

```
    r_fc2 = fc2(r_fc1)
```

```
    print ("shape after fc2 :",r_fc2.get_shape() )
```

```
    ## drop out
```

```
    # 참고 http://stackoverflow.com/questions/34597316/why-input-is-scaled-in-tf-nn-dropout-in-tensorflow
```

```
    # 트레이닝시에는 keep_prob < 1.0 , Test 시에는 1.0으로 한다.
```

```
    r_dropout = tf.nn.dropout(r_fc2,keep_prob)
```

```
    print ("shape after dropout :",r_dropout.get_shape() )
```

```
    # final layer
```

```
    r_out = final_out(r_dropout)
```

```
    print ("shape after final layer :",r_out.get_shape() )
```

```
    return r_out
```

이 build_model 함수는 image 를 입력 값으로 받아서 어떤 카테고리에 속할지를 리턴하는 컨볼루션 네트워크이다. 중간에 Dropout 계층이 추가되어 있는데, tf.nn.dropout 함수를 이용하면 간단하게 dropout 계층을 구현할 수 있다. r_fc2는 Dropout 계층 앞의 Fully Connected 계층에서 나온 값이고, 두번째 인자로 남긴 keep_prob는 Dropout 비율이다.

```
r_dropout = tf.nn.dropout(r_fc2,keep_prob)
print ("shape after dropout :",r_dropout.get_shape() )
```

모델 학습

데이터를 읽는 부분과 학습용 모델 정의가 끝났으면 실제로 학습을 시켜보자

```
def main(argv=None):
```

```
    # define placeholders for image data & label for training dataset
```

```
    images = tf.placeholder(tf.float32,[None,FLAGS.image_size,FLAGS.image_size,FLAGS.image_color])
```

```
    labels = tf.placeholder(tf.int32,[None,FLAGS.num_classes])
```

```
    image_batch,label_batch,file_batch = read_data_batch(TRAINING_FILE)
```

먼저 학습용 모델에 넣기 위한 image 데이터를 읽어드릴 placeholder를 images로 정의하고, 다음으로 모델에 의해 계산된 결과와 비교하기 위해서 학습데이터에서 읽어드린 label 데이터를 저장하기 위한 placeholder를 labels로 정의한다. 다음 image_batch,label_batch,file_batch 변수에 배치로 학습용 데이터를 읽어드린다. 그리고 dropout 계층에서 dropout 비율을 지정할 keep_prob를 placeholder로 정의한다. 각 변수가 지정되었으면, build_model 함수를 호출하여, images 값과 keep_prob 값을 넘겨서 Convolutional 네트워크에 값을 넣도록 그래프를 정의하고 그 결과 값을 prediction으로 정의한다.

```
    keep_prob = tf.placeholder(tf.float32) # dropout ratio
```

```
    prediction = build_model(images,keep_prob)
```

```
    # define loss function
```

```
    loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=prediction,labels=labels))
```

```
    tf.summary.scalar('loss',loss)
```

```
    #define optimizer
```

```
    optimizer = tf.train.AdamOptimizer(FLAGS.learning_rate)
```

```
    train = optimizer.minimize(loss)
```

중간 중간에 학습 과정을 시각화 하기 위해서 tf.summary.scalar 함수를 이용하여 loss 값을 저장하였다.

그래프 생성이 완료 되었으면, 학습에서 계산할 비용 함수를 정의한다. 비용함수는 softmax cross entropy 함수를 이용하여, 모델에 의해서 예측된 값 prediction 과, 학습 파일에서 읽어드린 label 값을 비교하여 loss 값에 저장한다.

그리고 이 비용 최적화 함수를 위해서 옵티마이저를 AdamOptimizer를 정의하여, loss 값을 최적화 하도록 하였다.

학습용 모델 정의와, 비용 함수, 옵티마이저 정의가 끝났으면 학습 중간 중간 학습된 모델을 테스트하기 위한 Validation 관련 항목등을 정의한다.

```
# for validation
#with tf.name_scope("prediction"):

validate_image_batch,validate_label_batch,validate_file_batch = read_data_batch(VALIDATION_FILE)
label_max = tf.argmax(labels,1)
pre_max = tf.argmax(prediction,1)
correct_pred = tf.equal(tf.argmax(prediction,1),tf.argmax(labels,1))
accuracy = tf.reduce_mean(tf.cast(correct_pred,tfloat32))

tf.summary.scalar('accuracy',accuracy)

startTime = datetime.now()
```

학습용 데이터가 아니라 검증용 데이터를 VALIDATION_FILE에서 읽어서 데이터를 validate_image_batch,validate_label_batch,validate_file_batch에 저장한다. 다음, 정확도 체크를 위해서 학습에서 예측된 라벨값과, 학습 데이터용 라벨값을 비교하여 같은지 틀린지를 비교하고, 이를 가지고 평균을 내서 정확도 (accuracy)로 사용한다.

학습용 모델과, 테스트용 데이터 등이 준비되었으면 이제 학습을 시작한다. 학습을 시작하기 전에, 학습된 모델을 저장하기 위해서 tf.train.Saver()를 지정한다. 그리고, 그래프로 loss와 accuracy등을 저장하기 위해서 Summary write를 저장한다. 다음 tf.global_variable_initializer()를 수행하여 변수를 초기화 하고, queue에서 데이터를 읽기 위해서 tf.train.Coordinator를 선언하고 tf.start_queue_runners를 지정하여, queue 러너를 실행한다.

```
#build the summary tensor based on the tF collection of Summaries
summary = tf.summary.merge_all()

with tf.Session(config=tf.ConfigProto(allow_soft_placement=True, log_device_placement=True)) as sess:
    saver = tf.train.Saver() # create saver to store training model into file
    summary_writer = tf.summary.FileWriter(FLAGS.log_dir,sess.graph)

    init_op = tf.global_variables_initializer() # use this for tensorflow 0.12rc0
    coord = tf.train.Coordinator()
    threads = tf.train.start_queue_runners(sess=sess, coord=coord)
    sess.run(init_op)
```

변수 초기화와 세션이 준비되었기 때문에 이제 학습을 시작해보자. for 루프를 이용하여 총 10,000 스텝의 학습을 하도록 하였다.

```
for i in range(10000):
    images_,labels_ = sess.run([image_batch,label_batch])
```

다음 image_batch와 label_batch에서 값을 읽어서 앞에서 정의한 모델에 넣고 train 그래프 (AdamOptimizer를 정의한)를 실행한다.

```
sess.run(train,feed_dict={images:images_,labels:labels_,keep_prob:0.7})
```

이때 앞에서 읽은 images_와, labels_ 데이터를 피딩하고 keep_prob 값을 0.7로 하여 30% 정도의 값을 Dropout 시킨다.

다음 10 스텝 마다 학습 상태를 체크하도록 하였다.

```
if i % 10 == 0:
    now = datetime.now()-startTime
    print('## time:',now,' steps:',i)

    # print out training status
    rt = sess.run([label_max,pre_max,loss,accuracy],feed_dict={images:images_
                                                                , labels:labels_
                                                                , keep_prob:1.0})
    print ('Prediction loss:',rt[2],' accuracy:',rt[3])
    위와 같이 loss 값과 accuracy 값을 받아서 출력하여 현재 모델의 비용 함수 값과 정확도를 측정하고

    # validation steps
    validate_images_,validate_labels_ = sess.run([validate_image_batch,validate_label_batch])
    rv = sess.run([label_max,pre_max,loss,accuracy],feed_dict={images:validate_images_
                                                                , labels:validate_labels_
                                                                , keep_prob:1.0})
    print ("Validation loss:",rv[2],' accuracy:',rv[3])
    학습용 데이터가 아니라 위와 같이 테스트용 데이터를 피딩하여, 테스트용 데이터로 정확도를 검증한다.
    이때 keep_prob를 1.0으로 해서 Dropout 없이 100% 네트워크를 활용한다.
```

```
if(rv[3] > 0.9):
    Break
```

만약에 테스트 정확도가 90% 이상이면 학습을 멈춘다. 그리고 아래와 같이 Summary

```
# validation accuracy
summary_str = sess.run(summary,feed_dict={images:validate_images_
                                          , labels:validate_labels_
                                          , keep_prob:1.0})

summary_writer.add_summary(summary_str,i)
summary_writer.flush()

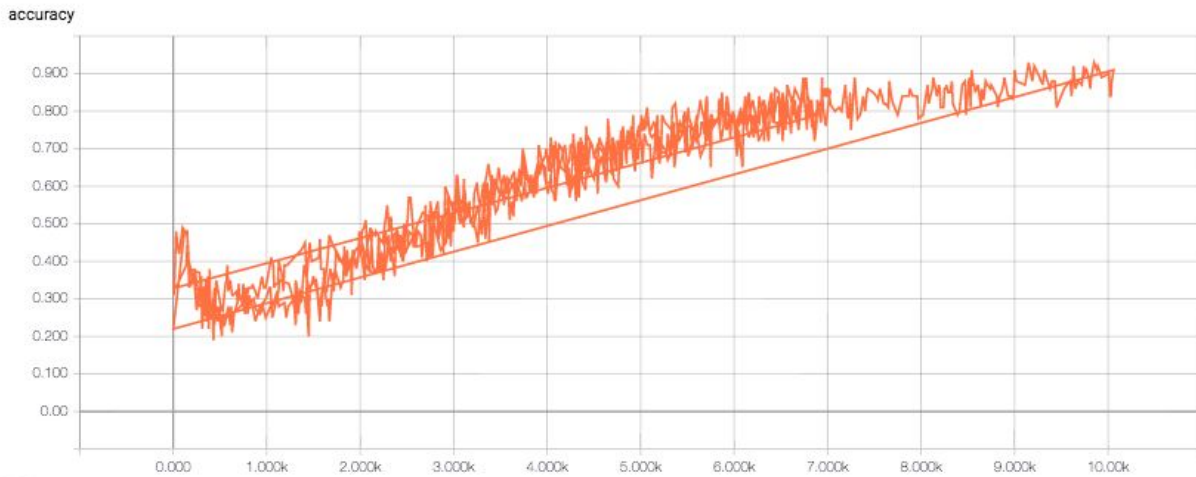
마지막으로 다음과 같이 학습이 다된 모델을 saver.save를 이용하여 저장하고, 사용된 리소스들을 정리한다.
saver.save(sess, 'face_recog') # save session
coord.request_stop()
coord.join(threads)
print('finish')

main()
```

이렇게 학습을 끝내면 본인의 경우 약 7000 스텝에서 테스트 정확도 91%로 끝난것을 확인할 수 있다.

```
( '## time:', datetime.timedelta(0, 9722, 726640), ' steps:', 7000)
('Prediction loss:', 0.11166426, ' accuracy:', 0.94)
('Validation loss:', 0.32143793, ' accuracy:', 0.91000003)
finish
```

아래는 텐서보드를 이용하여 학습 과정을 시각화한 내용이다.



코드는 공개가 가능하지만 학습에 사용한 데이터는 저작권 문제로 공유가 불가능하다. 약 200장의 사진만 제대로 수집을 하면 되기 때문에 각자 수집을 해서 학습을 도전해보는 것을 권장한다. (더 많은 인물에 대한 시도를 해보는것도 좋겠다.)

혹시나 이 튜토리얼을 따라하면서 학습 데이터를 공개할 수 있는 분들이 있다면 다른 분들에게도 많은 도움이 될것이라고 생각한다. 가능하면 데이터가 공개되었으면 좋겠다.

전체 코드는

<https://github.com/bwcho75/facerecognition/blob/master/1.%2BFace%2BRecognition%2BTraining.ipynb> 에 있다.

그리고 직접 사진을 수집해보면, 데이터 수집 및 가공이 얼마나 어려운지 알 수 있기 때문에 직접 한번 시도해보는 것도 권장한다. 아래는 크롬브라우저 플러그인으로 구글 검색에서 나온 이미지를 싹 긁을 수 있는 플러그인이다. Bulk Download Images (ZIG)

<https://www.youtube.com/watch?v=k5ioaelzEBM>

이 플러그인을 이용하면 손쉽게 특정 인물의 데이터를 수집할 수 있다.

다음 글에서는 학습이 끝난 데이터를 이용해서 실제로 예측을 해보는 부분에 대해서 소개하도록 하겠다.

앞글에 걸쳐서 얼굴 인식을 위한 데이터를 수집 및 정제하고, 이를 기반으로 얼굴 인식 모델을 학습 시켰다.

- <http://bcho.tistory.com/1178> 얼굴인식 모델 개발 및 학습 시키기

- <http://bcho.tistory.com/1176> 학습 데이터 준비하고

이번글에서는 학습이 된 데이터를 가지고, 사진을 넣어서 실제로 인식하는 코드를 만들어보자

학습된 모델로 예측하기

모델 로딩 하기

모델 학습에 사용한 CNN 모델을 똑같이 정의한다. conv1(),conv2(),conv3(),conv4(),fc1(),fc2(), build_model() 등 학습에 사용된 CNN 네트워크를 똑같이 정의하면 된다.

다음으로 이 모델에 학습된 값들을 채워 넣어야 한다.

```
# build graph
images = tf.placeholder(tf.float32,[None,FLAGS.image_size,FLAGS.image_size,FLAGS.image_color])
keep_prob = tf.placeholder(tf.float32) # dropout ratio
```

예측에 사용할 image 를 넘길 인자를 images라는 플레이스홀더로 정의하고, dropout 비율을 정하는 keep_prob도 플레이스 홀더로 정의한다.

```
prediction = tf.nn.softmax(build_model(images,keep_prob))
```

그래프를 만드는데, build_model에 의해서 나온 예측 결과에 softmax 함수를 적용한다. 학습시에는 softmax 함수의 비용이 크기 때문에 적용하지 않았지만, 예측에서는 결과를 쉽게 알아보기 위해서 softmax 함수를 적용한다. Softmax 함수는 카테고리 별로 확률을 보여줄때 전체 값을 1.0으로 해서 보여주는것인데, 만약에 Jolie,Sulyun,Victora 3개의 카테고리가 있을때 각각의 확률이 70%,20%,10%이면 Softmax를 적용한 결과는 [0.7,0.2,0.1] 식으로 출력해준다.

```
sess = tf.InteractiveSession()
sess.run(tf.global_variables_initializer())
```

다음 텐서플로우 세션을 초기화 하고,

```
saver = tf.train.Saver()
saver.restore(sess, 'face_recog')
```

마지막으로 Saver의 restore 함수를 이용하여 'face_recog'라는 이름으로 저장된 학습 결과를 리스토어 한다. (앞의 예제에서, 학습이 완료된 모델을 'face_recog'라는 이름으로 저장하였다.)

예측하기

로딩 된 모델을 가지고 예측을 하는 방법은 다음과 같다. 이미지 파일을 읽은 후에, 구글 클라우드 VISION API를 이용하여, 얼굴의 위치를 추출한후, 얼굴 이미지만 크롭핑을 한후에, 크롭된

이미지를 텐서플로우 데이터형으로 바꾼 후에, 앞서 로딩한 모델에 입력하여 예측된 결과를 받게 된다.

얼굴 영역 추출하기

먼저 vision API로 얼굴 영역을 추출하는 부분이다. 앞의 이미지 전처리에 사용된 부분과 다르지 않다.

```
import google.auth
import io
import os
from oauth2client.client import GoogleCredentials
from google.cloud import vision
from PIL import Image
from PIL import ImageDraw

FLAGS.image_size = 96

# set service account file into OS environment value
os.environ["GOOGLE_APPLICATION_CREDENTIALS"] = "/Users/terrycho/keys/terrycho-ml.json"
```

위와 같이 구글 클라우드 Vision API를 사용하기 위해서 액세스 토큰을 Service Account 파일로 다운 받아서 위와 같이 GOOGLE_APPLICATION_CREDENTIALS 환경 변수에 세팅 하였다.

```
visionClient = vision.Client()
print ('[INFO] processing %s'%(imagefile))

#detect face
image = visionClient.image(filename=imagefile)
faces = image.detect_faces()
face = faces[0]
```

다음 vision API 클라이언트를 생성한 후에, detect_faces() 를 이용하여 얼굴 정보를 추출해낸다.

```
print 'number of faces ',len(faces)

#get face location in the photo
left = face.fd_bounds.vertices[0].x_coordinate
top = face.fd_bounds.vertices[0].y_coordinate
right = face.fd_bounds.vertices[2].x_coordinate
bottom = face.fd_bounds.vertices[2].y_coordinate
rect = [left,top,right,bottom]
```

추출된 얼굴 정보에서 첫번째 얼굴의 위치 (상하좌우) 좌표를 리턴 받는다.

얼굴 영역을 크롭하기

앞에서 입력 받은 상하좌우 좌표를 이용하여, 이미지 파일을 열고, 크롭한다.

```
fd = io.open(imagefile, 'rb')
image = Image.open(fd)

import matplotlib.pyplot as plt
# display original image
print "Original image"
plt.imshow(image)
plt.show()

# draw green box for face in the original image
print "Detect face boundary box "
draw = ImageDraw.Draw(image)
draw.rectangle(rect, fill=None, outline="green")

plt.imshow(image)
plt.show()

crop = image.crop(rect)
im = crop.resize((FLAGS.image_size, FLAGS.image_size), Image.ANTIALIAS)
plt.show()
im.save('cropped'+imagefile)
```

크롭된 이미지를 텐서플로우에서 읽는다.

```
print "Cropped image"
tfimage = tf.image.decode_jpeg(tf.read_file('cropped'+imagefile), channels=3)
tfimage_value = tfimage.eval()
```

크롭된 파일을 decode_jpeg() 메서드로 읽은 후에, 값을 tfimage.eval()로 읽어드린다.

```
tfimages = []
tfimages.append(tfimage_value)
```

앞에서 정의된 모델이 한개의 이미지를 인식하는게 아니라 여러개의 이미지 파일을 동시에 읽도록 되어 있기 때문에, tfimages라는 리스트를 만든 후, 인식할 이미지를 붙여서 전달한다.

```
plt.imshow(tfimage_value)
plt.show()
fd.close()

p_val = sess.run(prediction, feed_dict={images:tfimages, keep_prob:1.0})
```



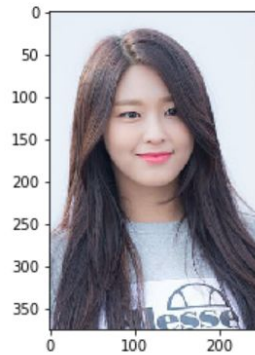
```
name_labels = ['Jessica Alba', 'Angelina Jolie', 'Nicole Kidman', 'Sulhyun', 'Victoria Beckham']
i = 0
for p in p_val[0]:
    print('%s %f' % (name_labels[i], float(p)))
    i = i + 1
```

tfimages 에 이미지를 넣어서 모델에 넣고 prediction 값을 리턴 받는다. dropout은 사용하지 않기 때문에, keep_prob을 1.0으로 한다.

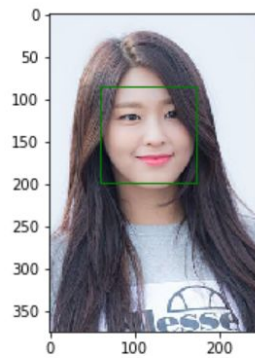
나온 결과를 가지고 Jessica, Jolie, Nicole Kidman, Sulhyun, Victoria Beckham 일 확률을 각각 출력한다.

다음은 설현 사진을 가지고 예측을 한 결과 이다.

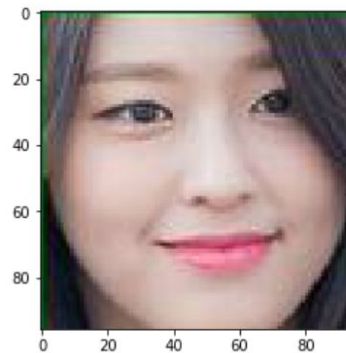
```
[INFO] processing sulhyun2.jpg
number of faces 1
Original image
```



Detect face boundary box



Cropped image



```
Jessica Alba 0.000000
Angelina Jolie 0.000000
Nicole Kidman 0.000000
Sulhyun 1.000000
Victoria Beckham 0.000000
```

이 코드는 학습된 모델을 기반으로 얼굴을 인식이 가능하기는 하지만 실제 운영 환경에 적용하기에는 부족하다. 파이썬 모델 코드를 그대로 옮겼기 때문에, 성능도 상대적으로 떨어지고, 실제 운영에서는 모델을 업그레이드 배포 할 수 있고, 여러 서버를 이용하여 스케일링도 지원해야 한다.

그래서 텐서플로우에서는 Tensorflow Serving 이라는 예측 서비스 엔진을 제공하고 구글 클라우드에서는 Tensorflow Serving의 매니지드 서비스인, CloudML 서비스를 제공한다.

앞의 두 글이 로컬 환경에서 학습과 예측을 진행했는데, 다음 글에서는 상용 서비스에 올릴 수 있는 수준으로 학습과 예측을 할 수 있는 방법에 대해서 알아보도록 하겠다.

클라우드를 이용하여, 학습하기

앞에서 모델을 만들고 학습도 다했다. 이제, 이 모델을 실제 운영 환경에서 운영할 수 있는 스케일로 포팅을 하고자 한다.

로컬 환경 대비 실제 운영 환경으로 확장할때 고려해야 하는 사항은

- 대규모 학습 데이터를 저장할 수 있는 공간
- 대규모 학습 데이터를 전처리하기 위한 병렬 처리 환경
이 내용은 이미 <http://bcho.tistory.com/1177>에서 다루었다.
- 대규모 학습 데이터를 빠르게 학습 시킬 수 있는 컴퓨팅 파워
- 학습된 데이터를 이용한 대규모 예측 서비스를 할 수 있는 기능

위의 요건을 만족하면서 텐서플로우로 환경을 올리는 방법은 여러가지가 있지만, 클라우드를 선택하기로 한다.

이유는

- 첫번째 모델 개발에 집중하고, 텐서플로우의 설치 및 운영 등에 신경쓰지 않도록 한다.
단순한 텐서플로우 설치뿐만 아니라 여러 장비를 동시에 이용하여 분산 학습을 하려면, 클러스터 구성 및 유지가 부담이 된다.
- 클라우드 컴퓨팅 파워를 이용하여, 대규모 데이터에 대한 전처리를 수행하고 개개별 학습 속도를 높이는 것은 물론이고, 모델을 튜닝하여 동시에 여러 모델을 학습 시킬 수 있다.
- 대용량 학습 데이터를 저장하기 위한 스토리지 인프라에 대한 구성 및 운영 비용을 절감한다.

즉 설정이나 운영은 클라우드에 맡겨 놓고, 클라우드의 무한한 자원과 컴퓨팅 파워를 이용하여 빠르게 모델을 학습하기 위함이다.

구글 클라우드

아무래도 일하는 성격상 구글 클라우드를 먼저 볼 수 밖에 없는데, 구글 클라우드에서는 텐서플로우의 매니지드 서비스인 CloudML을 제공한다.

CloudML은 별도의 설치나 환경 설정 없이 텐서플로우로 만든 모델을 학습 시키거나 학습된 결과로 예측을 하는 것이 가능하다. 주요 특징을 보면 다음과 같다.

- 학습시에, 별도의 설정 없이 텐서플로우 클러스터 크기 조절이 가능하다. 싱글 머신에서부터 GPU 머신 그리고 여러대의 클러스터 머신 사용이 가능하다
- 하이퍼 패러미터 튜닝이 가능하다. DNN의 네트워크의 폭과 깊이도 하이퍼 패러미터로 지정할 수 있으며, CloudML은 이런 하이퍼패러미터의 최적값을 자동으로 찾아준다.
- 예측 서비스에서는 Tensorflow Serv를 별도의 빌드할 필요 없이 미리 환경 설정이 다되어 있으며 (bazel 빌드의 끔직함을 겪어보신 분들은 이해하실듯) gRPC가 아닌 간단한 JSON 호출로 예측 (PREDICTION) 요청을 할 수 있다
- 분당 과금이다. 이게 강력한 기능인데, 구글 클라우드는 기본적으로 분당 과금으로 CPU를 사용하던, GPU를 사용하던 정확히 사용한 만큼만 과금하기 때문에, 필요할때 필요한 만큼만 사용하면 된다. 일부 클라우드의 경우에는 시간당 과금을 사용하기 때문에, 8대의 GPU머신에서 1시간 5분을 학습하더라도 8대에 대해서 2시간 요금을 내야하기 때문에 상대적으로 비용 부담이 높다.
- 가장 큰 메리트는 TPU (Tensorflow Processing Unit)을 지원한다는 것인데, 딥러닝 전용 GPU라고 생각하면 된다. 일반적인 CPU또는 GPU대비 15~30배 정도 빠른 성능을 제공한다.



현재는 Close Alpha로 특정 사용자에게만 시범 서비스를 제공하고 있지만 곧 CloudML을 통해서 일반 사용자에게도 서비스가 제공될 예정이다.

CloudML을 이용하여 학습하기

코드 수정

CloudML에서 학습을 시키려면 약간의 코드를 수정해야 한다. 수정해야 하는 이유는 학습 데이터를 같이 올릴 수 없기 때문인데, 여기에는 두 가지 방법이 있다.

- 학습 데이터를 GCS (Google Cloud Storage)에 올려놓은 후, 학습이 시작되기 전에 로컬 디렉토리로 복사해 오거나

- 또는 학습 데이터를 바로 GCS로 부터 읽어오도록 할 수 있다.

첫번째 방법은 gsutil 이라는 GCS 명령어를 이용하여 학습 시작전에 GCS에서 학습 데이터를 카피해오면 되고,

두번째 방법은 학습 데이터의 파일명을 GCS 로 지정하면 된다.

예를 들어 텐서 플로우 코드에서 이미지 파일을 아래와 같이 로컬 경로에서 읽어왔다면

```
image =
tf.image.decode_jpeg(tf.read_file("/local/trainingdata/"+image_file),channels=FLAGS.image_color)
```

GCS에서 읽어오려면 GCS 경로로 바꿔 주면 된다. GCS 버킷명이 terrycho-training-data라고 하면

```
image =
tf.image.decode_jpeg(tf.read_file("gs://terrycho-training-data/trainingdata/"+image_file),channels=FLAGS.image_color)
```

첫번째 방법의 경우에는 데이터가 아주 많지 않고, 분산 학습이 아닌경우 매우 속도가 빠르다.

두번째 방법의 경우에는 데이터가 아주아주 많아서 분산 학습이 필요할때 사용한다. 아무래도 로컬 파일 액세스가 GCS 액세스 보다 빠르기 때문이다.

다음은 첫번째 방식으로 학습 데이터를 로컬에 복사해서 학습하는 방식의 코드이다.

https://github.com/bwcho75/facerecognition/blob/master/CloudML%20Version/face_recog_model/model_localfile.py

코드 내용은 앞서 만들 모델 코드와 다를것이 없고 단지 아래 부분과, 파일 경로 부분만 다르다

```
def gcs_copy(source, dest):
    print('Recursively copying from %s to %s' %
          (source, dest))
    subprocess.check_call(['gsutil', '-q', '-m', 'cp', '-R']
                          + [source] + [dest])
```

gcs_copy 함수는 GCS의 source 경로에서 파일을 dest 경로로 복사해주는 명령이다.

```
def prepare_data():
    # load training and testing data index file into local
    gcs_copy('gs://' + DESTINATION_BUCKET + '/' + TRAINING_FILE, '.')
    gcs_copy('gs://' + DESTINATION_BUCKET + '/' + VALIDATION_FILE, '.')

    # loading training and testing images to local
    image_url = 'gs://' + DESTINATION_BUCKET + '/images/'

    if not os.path.exists(FLAGS.local_image_dir):
        os.makedirs(FLAGS.local_image_dir)
```

```
gcs_copy( image_url,FLAGS.local_image_dir)
```

```
prepare_data()
```

```
main()
```

그리고 prepare_data를 이용해서, 학습과 테스트용 이미지 목록 파일을 복사하고, 이미지들도 로컬에 복사한다.

로컬에 데이터 복사가 끝나면 main()함수를 호출하여 모델을 정의하고 학습을 시작한다.

디렉토리 구조

코드를 수정하였으면, CloudML을 이용하여 학습을 하려면, 파일들을 패키징 해야 한다. 별 다를것은 없고

[작업 디렉토리]

- + __init__.py
- + {모델 파일명}.py

식으로 디렉토리를 구성하면 된다.

얼굴 학습 모델을 model_localfile.py라는 이름으로 저장하였다

명령어

이제 학습용 모델이 준비되었으면, 이 모델을 CloudML에 집어 넣으면 된다.

명령어가 다소 길기 때문에, 셸 스크립트로 만들어놓거나 또는 파이썬 노트북에 노트 형식으로 만들어 놓으면 사용이 간편하다. 다음은 파이썬 노트북으로 만들어놓은 내용이다.

```
import google.auth
import os
import datetime
```

```
os.environ["GOOGLE_APPLICATION_CREDENTIALS"] = "/Users/terrycho/keys/terrycho-ml.json"
job_name = 'preparefacedata'+ datetime.datetime.now().strftime("%y%m%d%H%M%S")
```

리모트로 구글 클라우드의 CloudML을 호출하기 때문에,
GOOGLE_APPLICATION_CREDENTIALS에 서비스 어카운트 파일을 지정한다.
그리고 CloudML에 학습을 실행하면, 각 학습은 JOB으로 등록되는데, 손쉽게 JOB을 찾아서 모니터링 하거나 중지할 수 있도록, JOB ID를 현재 시간으로 생성한다.

```
print job_name
# Job name whatever you want
```

```

JOB_NAME=job_name
# the directory of folder that include your source and init file
PACKAGE_PATH='/Users/terrycho/anaconda/work/face_recog/face_recog_model'
# format: folder_name.source_file_name
MODULE_NAME='face_recog_model.model_localfile'
# bucket you created
STAGING_BUCKET='gs://terrycho-face-recog-stage'
# I recommend "europe-west1" region because there are not enough GPUs in US region for
you.....
REGION='us-east1'
# Default is CPU computation. set BASIC_GPU to use Tesla K80 !
SCALE_TIER='BASIC_GPU'

# Submit job with these settings
!gcloud ml-engine jobs submit training $JOB_NAME \
--package-path=$PACKAGE_PATH \
--module-name=$MODULE_NAME \
--staging-bucket=$STAGING_BUCKET \
--region=$REGION \
--scale-tier=$SCALE_TIER \

```

다음은 cloudml 명령어를 실행하면 된다. 각 인자를 보면

- JOB_NAME은 학습 JOB의 이름이다.
- package-path는 __init__.py와 학습 모델 및 관련 파일들이 있는 디렉토리가 된다.
- module-name은 package-path안에 있는 학습 실행 파일이다.
- staging-bucket은 CloudML에서 학습 코드를 올리는 임시 Google Cloud Storage로, Google Cloud Storage 만든 후에, 그 버킷 경로를 지정하면 된다.
- region은 CloudML을 사용한 리전을 선택한다.
- 마지막으로 scale-tier는 학습 머신의 사이즈를 지정한다.

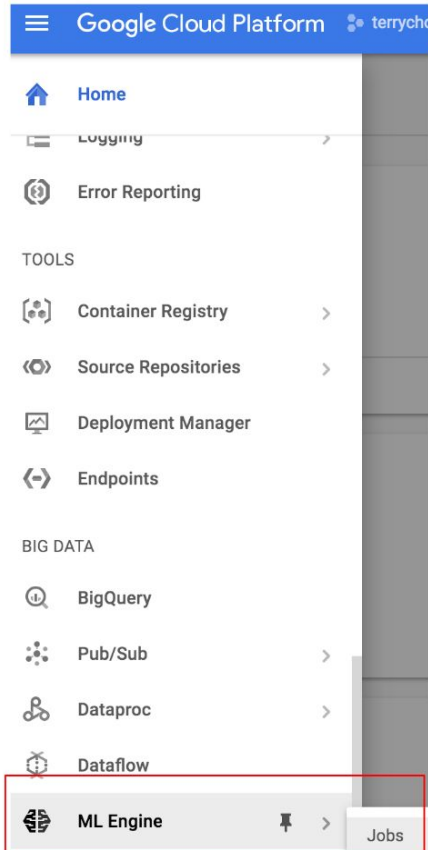
스케일 티어	설명
BASIC	싱글 머신. CPU
BASIC_GPU	싱글 머신 + K80 GPU
STANDARD_1	분산 머신
PREMIUM_1	대규모 분산 머신
CUSTOM	사용자가 클러스터 크기를 마음대로 설정

일반적인 모델은 BASIC_GPU를 사용하면 되고, 모델이 분산 학습이 가능하도록 개발되었으면 STANDARD_1 이나 PREMIUM_1을 사용하면 된다.

이렇게 명령을 수행하면 모델코드가 CloudML로 전송되고, 전송된 코드는 CloudML에서 실행된다.

학습 모니터링

학습이 시작되면 JOB을 구글 클라우드 콘솔의 CloudML 메뉴에서 모니터링을 할 수 있다.



다음은 CloudML에서의 JOB 목록이다. (진짜 없어 보인다...)

 ML Engine

 Jobs

 Models

Jobs

REFRESH

Filter jobs

Job ID	Type	Creation time	Elapsed time	Logs	
 preparefacedata170621221932	Training	Jun 21, 2017, 10:19:36 PM	1 hr 51 min	View logs	Stop
 preparefacedata170621200849	Training	Jun 21, 2017, 8:08:53 PM	1 hr 56 min	View logs	

실행중인 JOB에서 STOP 버튼을 누르면 실행중인 JOB을 정지시킬 수도 있고, View Logs 버튼을 누르면, 학습 JOB에서 나오는 로그를 볼 수 있다. (텐서플로우 코드내에서 print로 찍은 내용들도 모두 여기 나온다.)

Stackdriver Logging

CREATE METRIC CREATE EXPORT

Filter by label or text search

Cloud ML Job, preparefacedata17062122193... All logs Any log level Jump to date

2017-06-21 JST

↓

- 23:35:19.320 ('## time:', datetime.timedelta(0, 4208, 343029), ' steps:', 130)
- 23:35:19.320 ('Validation loss:', 19.279289, ' accuracy:', 0.51999998)
- 23:35:19.320 ('Prediction loss:', 11.954926, ' accuracy:', 0.57999998)
- 23:35:19.319 ('## time:', datetime.timedelta(0, 3961, 451005), ' steps:', 120)
- 23:35:19.319 ('Validation loss:', 33.642128, ' accuracy:', 0.38)
- 23:35:19.319 ('Prediction loss:', 12.621526, ' accuracy:', 0.56)
- 23:35:19.319 ('## time:', datetime.timedelta(0, 3716, 231488), ' steps:', 110)

여기까지 간단하게나마 CloudML을 이용하여 모델을 학습하는 방법을 알아보았다. 본인의 경우 연예인 인식 모델을 MAC PRO 15" i7 (NO GPU)에서 학습한 경우 7000 스텝까지 약 8시간이 소요되었는데, CloudML의 BASIC_GPU를 사용하였을때는 10,000 스텝에 약 1시간 15분 정도 (GCS를 사용하지 않고 직접 파일을 로컬에 복사해놓고 돌린 경우) 가 소요되었다. (빠르다)

여기서 사용된 전체 코드는

<https://github.com/bwcho75/facerecognition/tree/master/CloudML%20Version> 에 있다.

- model_gcs.py 는 학습데이터를 GCS에서 부터 읽으면서 학습하는 버전이고
- model_localfile.py는 학습데이터를 로컬 디스크에 복사해놓고 학습하는 버전이다.

다음 글에서는 학습된 모델을 배포하여 실제로 예측을 실행할 수 있는 API를 개발해보도록 하겠다.

클라우드를 이용하여 학습된 모델을 예측하기

지난글 (<http://bcho.tistory.com/1189>) 에서 학습된 모델을 *.pb 파일 포맷으로 Export 하였다. 그러면 이 Export 된 모델을 이용하여 예측 (prediction)을 하는 방법에 대해서 알아보겠다. 앞글에서도 언급했듯이, 예측은 Google CloudML을 이용한다.

전체 코드를

https://github.com/bwcho75/facerecognition/blob/master/CloudML%20Version/face_recog_model/%2528wwoo%2529%2BML%2BEngine%2Bprediction.ipynb 를 참고하기 바란다.

Export된 모델을 CloudML에 배포하기

학습된 모델을 CloudML에 배포하기 위해서는 export된 *.pb 파일과 variables 폴더를 구글 클라우드 스토리지 (GCS / Google Cloud Storage) 에 업로드해야 한다.
아니면 학습때 모델 Export를 GCS로 시킬 수 도 있다.

아래는 terrycho-face-recog-export 라는 GCS 버킷아래 /export 디렉토리에, export 된 *.pb 파일과 variables 폴더가 저장된 모습이다.

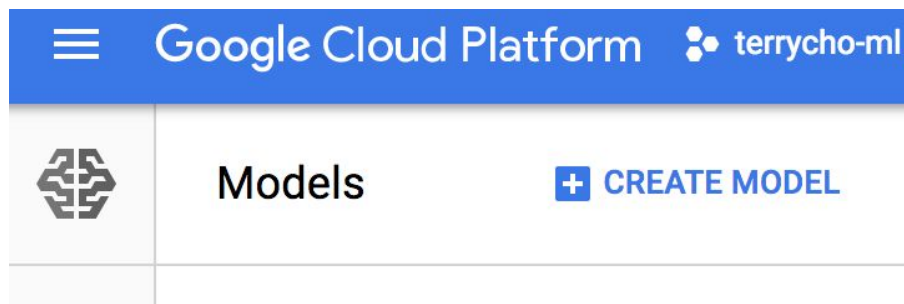
[Buckets](#) / [terrycho-face-recog-export](#) / export

<input type="checkbox"/>	Name	Size	Type
<input type="checkbox"/>	 saved_model.pb	46.66 KB	application/octet-stream
<input type="checkbox"/>	 variables/	—	Folder

다음 구글 클라우드 콘솔에서 ML Engine을 선택하여, Models 메뉴를 고른다. 이 메뉴는 모델을 배포하고 Prediction을 해주는 기능이다.



Models 화면으로 들어오면 Create Model 버튼이 나온다. 이 버튼을 이용해서 모델을 생성한다.



모델 생성시에 아래와 같이 단순히 모델이름을 넣어주면된다.

[←](#)

Create model

A model is a container for your model version from the command line and add it

Model name
Model names must be unique within each

Description (Optional)

[Create](#) [Cancel](#)

모델 이름을 넣어준 후에, 해당 모델에 실제 Export된 모델 파일을 배포해줘야 하는데, CloudML은 버전 기능을 제공한다. 그래서 아래 그림과 같이 Create Version 버튼을 눌러서 새로운 버전을 생성한다.

[←](#)

Model details

[+ CREATE VERSION](#)

face_recognition_blog

Versions

This model has no versions yet. Create at least one version to start model. [Create a version](#)

Create Version 메뉴에서는 Name에 버전명을 쓰고, Source에는 Export된 *.pb 파일과 variables 폴더가 저장된 GCS 경로를 선택한다.

Create version

To create a new version of your model, submit a train and specify the output below. [Learn more](#)

Name
Name is permanent.

Description (Optional)

Source
Enter the Google Cloud Storage output path you specified i

☒
terrycho-face-recog-export/export/

Create

Cancel

아래는 terrycho-face-recog-export 버킷을 선택한 후, 그 버킷안에 export 폴더를 선택하는 화면이다.

Select folder

<


terrycho-face-recog-exp... ▼

+

🔍

export/ >

선택을 해서 배포를 하면 아래와 같이 v7 버전이름을 모델이 배포가 된다.



[←](#)
Model details

+

CREATE VERSION

🗑️

DELETE

☰

💡

face_recognition_blog

Versions

Name	Creation time
✓ v7	Aug 15, 2017, 10:32:31 PM

배포된 모델로 예측 (Prediction)하기

그러면 배포된 모델을 사용해서 예측을 해보자. 아래가 전체코드이다.

```

from googleapiclient import discovery
from oauth2client.client import GoogleCredentials
import numpy
import base64
import logging

from IPython.display import display, Image

cropped_image = "croppedjolie.jpg"
display(Image(cropped_image))

PROJECT = 'terrycho-ml'
MODEL_NAME = 'face_recog'
MODEL_VERSION = 'v7'

def call_ml_service(img_str):
    parent = 'projects/{}/models/{}/versions/{}'.format(PROJECT, MODEL_NAME, MODEL_VERSION)
    pred = None

    request_dict = {
        "instances": [
            {
                "image": {
                    "b64": img_str
                }
            }
        ]
    }

    try:
        credentials = GoogleCredentials.get_application_default()
        cloudml_svc = discovery.build('ml', 'v1', credentials=credentials)
        request = cloudml_svc.projects().predict(name=parent, body=request_dict)
        response = request.execute()
        print(response)
        #pred = response['predictions'][0]['scores']
        #pred = numpy.asarray(pred)

    except Exception, e:
        logging.exception("Something went wrong!")

```

```

return pred

# base64 encode the same image
with open(cropped_image, 'rb') as image_file:
    encoded_string = base64.b64encode(image_file.read())

# See what ML Engine thinks
online_prediction = call_ml_service(encoded_string)

print online_prediction

```

코드를 살펴보면

```

credentials = GoogleCredentials.get_application_default()
cloudml_svc = discovery.build('ml', 'v1', credentials=credentials)

```

에서 discovery.build를 이용해서 구글 클라우드 API 중, 'ML' 이라는 API의 버전 'v1'을 불러왔다. CloudML 1.0 이다. 다음 credentials는 get_application_default()로 디폴트 credential을 사용하였다.

다음으로, CloudML에 request 를 보내야 하는데, 코드 윗쪽으로 이동해서 보면

```

def call_ml_service(img_str):
    parent = 'projects/{}/models/{}/versions/{}'.format(PROJECT, MODEL_NAME, MODEL_VERSION)
    pred = None

    request_dict = {
        "instances": [
            {
                "image": {
                    "b64": img_str
                }
            }
        ]
    }

```

를 보면 request body에 보낼 JSON을 request_dict로 정의하였다. 이때, 이미지를 "b64"라는 키로 img_str을 넘겼는데, 이 부분은 이미지 파일을 읽어서 base64 스트링으로 인코딩 한 값이다.

```
request = cloudml_svc.projects().predict(name=parent, body=request_dict)
```

다음 request 를 만드는데, 앞에서 선언한 cloudml_svc객체를 이용하여 prediction request 객체를 생성한다. 이때 parent 에는 모델의 경로가 들어가고 body에는 앞서 정의한 이미지가 들어있는 JSON 문자열이 된다.

```
parent = 'projects/{}/models/{}/versions/{}'.format(PROJECT, MODEL_NAME, MODEL_VERSION)
```

Parent에는 모델의 경로를 나타내는데, projects/{프로젝트명}/models/{모델명}/versions/{버전명} 형태로 표현되며, 여기서는 projects/terrycho-ml/models/face_recog/versions/v7 의 경로를 사용하였다.

이렇게 request 객체가 만들어지면 이를 request.execute()로 이를 호출하고 결과를 받는다.

```
response = request.execute()
```

```
print(response)
```

결과를 받아서 출력해보면 다음과 같은 결과가 나온다.



```
{u'predictions': [{u'prediction': [2.1277719497447833e-06, 0.9999629259109497, 2.6442356102052145e-05, 1.942615455163832e-07, 8.264843927463517e-06]}]}
```

2번째 라벨이 0.99% 확률로 유사한 결과가 나온것을 볼 수 있다. 라벨 순서 대로 첫번째가 제시카 알바, 두번째가 안젤리나 졸리, 세번째가 니콜 키드만, 네번째가 설현, 다섯번째가 빅토리아 베컴이다.

이제 까지 여러회에 걸쳐서 텐서플로를 이용하여 CNN 모델을 구현하고, 이 모델을 기반으로 얼굴 인식을 학습 시키고 예측 시키는 모델 개발까지 모두 끝 맞췄다.

실제 운영 환경에서 사용하기에는 모델이 단순하지만, 여기에 CNN 네트워크만 고도화하면 충분히 사용할만한 모델을 개발할 수 있을 것이라고 본다. CNN 네트워크에 대한 이론 보다는 실제 구현하면서 데이터 전처리 및 학습과, 학습된 모델의 저장 및 이를 이용한 예측 까지 전체 흐름을 설명하기 위해서 노력하였다.

다음은 이 얼굴 인식 모델을 실제 운영환경에서 사용할만한 수준의 품질이 되는 모델을 사용하는 방법을 설명하고자 한다.

직접 CNN 모델을 만들어도 되지만, 얼마전에, 발표된 Tensorflow Object Detection API (https://github.com/tensorflow/models/tree/master/object_detection)는 높은 정확도를 제공하는 이미지 인식 모델을 라이브러리 형태로 제공하고 있다. 다음 글에서는 이 Object Detection API를 이용하여 연예인 얼굴을 학습 시키고 인식하는 모델을 개발하고 학습 및 예측 하는 방법에 대해서 알아보도록 하겠다.

10. 텐서플로우 Object Detection API를 이용한 이미지 인식

Object Detection API 설치

Tensorflow Object Detection API는, Tensorflow 를 이용하여 이미지를 인식할 수 있도록 개발된 모델로, 라이브러리 형태로 제공되며, 각기 다른 정확도와 속도를 가지고 있는 5개의 모델을

제공한다. 머신러닝이나 텐서플로우에 대한 개념이 거의 없더라도 라이브러리 형태로 손쉽게 사용할 수 있으며, 직접 사용자 데이터를 업로드해서 학습을 하여, 내 시나리오에 맞는 Object Detection System을 손쉽게 만들 수 있다.

Object Detection API를 설치하기 위해서는 텐서플로우 1.x 와 파이썬 2.7x 버전이 사전 설치되어 있어야 한다. 이 글에서는 파이썬 2.7.13과 텐서플로우 2.7.13 버전을 기준으로 하고, 맥에 설치하는 것을 기준으로 한다. 리눅스나 다른 플랫폼 설치의 원본 설치 문서 https://github.com/tensorflow/models/blob/master/object_detection/g3doc/installation.md 를 참고하기 바란다.

설치 및 테스트

Protocol Buffer 설치

Object Detection API는 내부적으로 Protocol Buffer를 사용한다. MAC에서 Protocol Buffer를 설치 하는 방법은 <https://github.com/google/protobuf/tree/master/python> 와 <http://bcho.tistory.com/1182> 를 참고하기 바란다.

설치가 되었는지를 확인하려면, 프롬프트 상에서 protoc 명령을 실행해보면 된다.

```
[(objectdetection) terrycho-macbookpro:trainingdata_out terrycho$ protoc  
Missing input file.
```

파이썬 라이브러리 설치

프로토콜 버퍼 설치가 끝났으면, 필요한 파이썬 라이브러리를 설치한다.

```
% pip install pillow  
% pip install lxml  
% pip install jupyter  
% pip install matplotlib
```

Object Detection API 다운로드 및 설치

Object Detection API 설치의 간단하게, 라이브러리를 다운 받으면 된다. 설치할 디렉토리로 들어가서 git clone 명령어를 통해서, 라이브러리를 다운로드 받자

```
% git clone https://github.com/tensorflow/models
```

Protocol Buffer 컴파일

다음 프로토콜 버퍼를 사용하기 위해서 protoc로 proto 파일을 컴파일 한데, Object Detection API를 설치한 디렉토리에서 models 디렉토리로 들어간 후에, 다음 명령어를 수행한다.

```
protoc object_detection/protos/*.proto --python_out=.
```


PATH 조정하기

설치가 끝났으면 Object Detection API를 PATH와 파이썬 라이브러리 경로인 PYTHONPATH에 추가한다. 맥에서는 사용자 홈디렉토리의 .bash_profile 에 추가 하면되낀.

PYTHONPATH 환경 변수에 {Object Detection API 설치 디렉토리}/models/slim 디렉토리와 Object Detection API 설치 디렉토리}/models/models 디렉토리를 추가한다.
같은 디렉토리를 PATH에도 추가해준다.

```
export
PYTHONPATH=$PYTHONPATH:/Users/terrycho/dev/workspace/objectdetection/models:/Users/terrycho/dev/worksp
ace/objectdetection/models/slim
export
PATH=$PATH:/Users/terrycho/dev/workspace/objectdetection/models:/Users/terrycho/dev/workspace/objectdetection
/models/slim
```

테스팅

설치가 제대로 되었는지를 확인하기 위해서 {Object Detection API 설치 디렉토리}/models/
디렉토리에서 다음 명령을 실행해보자

```
% python object_detection/builders/model_builder_test.py
```

문제 없이 실행이 되었으면 제대로 설치가 된것이다.

```
[(objectdetection) terrycho-macbookpro:models terrycho$ python object_detection/builders/model_builder_test.py
.....
-----
Ran 7 tests in 0.038s
OK
```

사용하기

설치가 끝났으면 실제로 사용해 보자, Object Detection API를 인스톨한 디렉토리 아래
models/object_detection/object_detection_tutorial.ipynb 에 테스트용 노트북 파일이 있다. 이
파일을 주피터 노트북 (<http://jupyter.org/>)을 이용하여 실행해보자.

(원본 코드

https://github.com/tensorflow/models/blob/master/object_detection/object_detection_tutorial.ipynb)

localhost:8889/notebooks/dev... object_detection_tutorial (autosaved) Logout

File Edit View Insert Cell Kernel Help Not Trusted Python 2

Object Detection Demo

Welcome to the object detection inference walkthrough! This notebook will walk you step by step through the process of using a pre-trained model to detect objects in an image. Make sure to follow the [installation instructions](#) before you start.

Imports

```
In [1]: import numpy as np
import os
import six.moves.urllib as urllib
import sys
import tarfile
import tensorflow as tf
import zipfile

from collections import defaultdict
from io import StringIO
from matplotlib import pyplot as plt
from PIL import Image
```

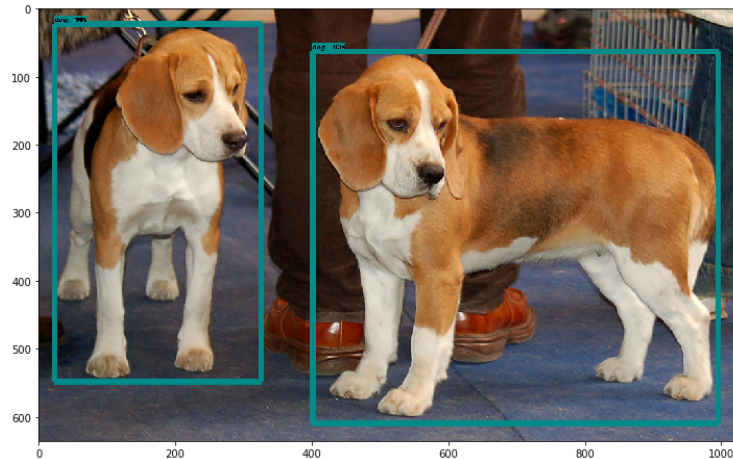
Env setup

```
In [2]: # This is needed to display the images.
%matplotlib inline

# This is needed since the notebook is stored in the object_detection folder.
sys.path.append("..")
```

실행을 하면 결과로 아래와 같이 물체를 인식한 결과를 보여준다.





이 중에서 중요한 부분은 Model Preparation이라는 부분으로, 여기서 하는 일은 크게 아래 3가지와 같다.

- Export 된 모델 다운로드
- 다운로드된 모델 로딩
- 라벨맵 로딩

Export 된 모델 다운로드

Object Detection API는 여러가지 종류의 미리 훈련된 모델을 가지고 있다.

모델 종류는

https://github.com/tensorflow/models/blob/master/object_detection/g3doc/detection_model_zoo.md 를 보면 되는데, 다음과 같은 모델들을 지원하고 있다. COCO mAP가 높을 수록 정확도가 높은 모델인데, 대신 예측에 걸리는 속도가 더 느리다.

Model name	Speed	COCO mAP	Outputs
ssd_mobilenet_v1_coco	fast	21	Boxes
ssd_inception_v2_coco	fast	24	Boxes
rfcn_resnet101_coco	medium	30	Boxes
faster_rcnn_resnet101_coco	medium	32	Boxes
faster_rcnn_inception_resnet_v2_atrous_coco	slow	37	Boxes

모델은 *.gz 형태로 다운로드가 되는데, 이 파일안에는 다음과 같은 내용들이 들어있다.

- Check point (model.ckpt.data-00000-of-00001, model.ckpt.index, model.ckpt.meta)
텐서플로우 학습 체크 포인트로, 나중에, 다른 데이터를 학습 시킬때 Transfer

Learning을 이용할때, 텐서플로우 그래프에 이 체크포인트를 로딩하여, 그 체크포인트 당시의 상태로 학습 시켜놓을 수 있다. 이 예제에서는 사용하지 않지만, 다른 데이터를 이용하여 학습할때 사용한다.

- 학습된 모델 그래프 (frozen_inference_graph.pb)
학습이 완료된 그래프에 대한 내용을 Export 해놓은 파일이다. 이 예제에서는 이 모델 파일을 다시 로딩하여 Prediction을 수행한다.
- Graph proto (pgrah.pbtxt)

기타 파일들

이외에도 기타 다른 파일들이 있는데, 다른 파일들은 이미 Object Detection API 안에 이미 다운로드 되어 있다.

- 라벨맵
라벨맵은 {Object Detection API 설치 디렉토리}/models/object_detection/data 디렉토리 안에 몇몇 샘플 모델에 대한 라벨맵이 저장되어 있다. 라벨맵은 모델에서 사용한 분류 클래스에 대한 정보로 name,id,display_name 식으로 정의되며, name은 텍스트 라벨, id는 라벨을 숫자로 표현한 값 (반드시 1부터 시작해야 한다.), display_name은 Prediction 결과를 원본 이미지에서 인식한 물체들을 박스처리해서 출력하는데 이때 박스에 어떤 물체인지 출력해주는 문자열에 들어가는 텍스트 이다.
여기서 사용한 라벨맵은 mscoco_label_map.pbtxt 파일이 사용되었다.
- 학습 CONFIG 파일
모델 학습과 예측에 사용되는 각종 설정 정보를 저장한 파일로 위에서 미리 정의된 모델별로 각각 다른 설정 파일을 가지고 있으며 설정 파일의 위치는 {Object Detection API 설치 디렉토리}/models/object_detection/samples/configs 에 {모델명}.config 에 저장되어 있다.

다운로드된 모델과 라벨맵 로딩

위에서 많은 파일이 다운로드되고 언급되었지만 예측 (Prediction)에는 학습된 그래프 모델을 저장한 frozen_inference_graph.pb 파일과, 분류 라벨이 저장된 mscoco_label_map.pbtxt 두 개만 사용된다.

다음 코드 부분에서 모델을 다운 로드 받고, 모델 파일과 라벨 파일의 경로를 지정하였다.

```
# What model to download.
MODEL_NAME = 'ssd_mobilenet_v1_coco_11_06_2017'
MODEL_FILE = MODEL_NAME + '.tar.gz'
DOWNLOAD_BASE = 'http://download.tensorflow.org/models/object_detection/'

# Path to frozen detection graph. This is the actual model that is used for the object detection.
PATH_TO_CKPT = MODEL_NAME + '/frozen_inference_graph.pb'

# List of the strings that is used to add correct label for each box.
```

```
PATH_TO_LABELS = os.path.join('data', 'mscoco_label_map.pbtxt')
```

```
NUM_CLASSES = 90
```

그리고 마지막 부분에 분류 클래스의 수를 설정한다. 여기서는 90개의 클래스로 정의하였다. 만약에 모델을 바꾸고자 한다면 PATH_TO_CKPT를 다른 모델 파일로 경로만 변경해주면 된다.

다음으로 frozen_inference_graph.pb 로 부터 모델을 읽어서 그래프를 재생성하였다.

```
detection_graph = tf.Graph()
with detection_graph.as_default():
    od_graph_def = tf.GraphDef()
    with tf.gfile.GFile(PATH_TO_CKPT, 'rb') as fid:
        serialized_graph = fid.read()
        od_graph_def.ParseFromString(serialized_graph)
    tf.import_graph_def(od_graph_def, name="")
```

나머지 부분은 이미지를 읽어서, 로딩된 모델을 이용하여 물체를 Detection 하는 코드이다.

여기까지 간단하게 Tensorflow Object Detection API를 설치 및 사용하는 방법에 대해서 알아보았다.

다음 글에서는 다른 데이터로 모델을 학습해서 예측하는 부분에 대해서 알아보도록 하겠다.

참고 자료

- Tensorflow Object Detection API home :
https://github.com/tensorflow/models/tree/master/object_detection
- Install guide -
https://github.com/tensorflow/models/blob/master/object_detection/g3doc/installation.md
- Quick start -
https://github.com/tensorflow/models/blob/master/object_detection/object_detection_tutorial.ipynb
- Install to google cloud platform -
<https://cloud.google.com/solutions/creating-object-detection-application-tensorflow>

Object Detection API에 애완동물 사진을 학습 시켜 보자

Object Detection API에 이번에는 애완동물 사진 데이터를 학습시켜 보도록 한다.

애완 동물 학습 데이터의 원본은 Oxford-IIIT Pets lives 로

<http://www.robots.ox.ac.uk/~vgg/data/pets/> 에 있다. 약 37개의 클래스에, 클래스당 200개 정도의 이미지를 가지고 있다.



이번 글에서는 이 애완동물 데이터를 다운 받아서, Object Detection API에 학습 시키는 것까지 진행을 한다.

데이터를 다운로드 받은 후, Object Detection API에 학습 시키기 위해서, 데이터 포맷을 TFRecord 형태로 변환한 후, 학습을 하는 과정을 설명한다.

주의할점 : 이 튜토리얼은 총 37개의 클래스 약 7000장의 이미지를 학습시키는데, 17시간 이상이 소요되며, 구글 클라우드 CloudML의 텐서플로우 클러스터에서 분산 러닝을 하도록 설명하고 있는데, 대략 3000\$ 이상의 비용이 든다. 전체 흐름과 과정을 이해하기 위해서는 17시간을 풀 트레이닝 시키지 말고 학습 횟수를 줄이거나 아니면 중간에서 학습을 멈춰서 비용이 많이 나오지 않도록 하는 것을 권장한다. (테스트 결과 1000번 정도 학습을 하면 3만원 정도의 비용에서 학습이 가능하고 어느정도 유의미한 결과를 얻을 수 있다)

학습 데이터 다운로드 받기

```
%curl -O http://www.robots.ox.ac.uk/~vgg/data/pets/data/images.tar.gz
```

```
%curl -O http://www.robots.ox.ac.uk/~vgg/data/pets/data/annotations.tar.gz
```

※ 맥이기 때문에, curl -O 를 사용했는데, Linux의 경우에는 wget을 사용하면 된다.

파일을 다운로드 받았으면 압축을 풀어보자

- images.tar.gz에는 애완동물의 학습용 이미지가 들어가 있다.
- annotations.tar.gz 는 각 이미지에 대한 메타 데이터가 들어있다. 이미지 마다 나타난 동물의 종류, 사진상 동물의 위치 (박스)

TFRecord 파일 포맷으로 컨버팅 하기

압축을 푼 메타데이터와 이미지 파일을 이용해서 tfrecord 파일 형태로 컨버팅을 해야 한다. Tfreord 내에는 이미지 바이너리, 이미지에 대한 정보 (이미지 크기, 인식할 물체의 위치, 라벨)등이 들어간다. 상세 데이터 포맷에 대해서는 다음글에서 설명하도록 한다.

이 데이터를 가지고 tfrecord 타입으로 컨버팅 하는 코드는

```
object_detection/create_pet_tf_record.py
```

에 이미 작성되어 있다. 아래 코드를 이용해서 실행해주면 자동으로 pet_train.record에 학습용 데이터를 pet_val.record에 테스트용 데이터를 생성해준다.

```
python object_detection/create_pet_tf_record.py \
  --label_map_path=object_detection/data/pet_label_map.pbtxt \
  --data_dir=`pwd` \
  --output_dir=`pwd`
```

학습 환경 준비하기

데이터가 준비되었으면 학습을 위한 환경을 준비해야 한다.

학습은 구글 클라우드 플랫폼의 CloudML을 사용한다. CloudML은 구글 클라우드 플랫폼의 Tensorflow managed 서비스로, Tensorflow 클러스터 설치나 운영 필요 없이 간단하게 명령어만으로 여러대의 머신에서 학습을 가능하게 해준다.

CloudML을 사용하기 위해서는 몇가지 환경 설정을 해줘야 한다.

- 먼저 학습용 데이터 (tfrecord)파일을 구글 클라우드 스토리지 (GCS)로 업로드 해야 한다.
- Object Detection API에서 사물 인식에 사용된 모델의 체크 포인트를 업로드 해야 한다.
- 클라우드에서 학습을 하기 때문에, 텐서플로우 코드를 패키징해서 업로드해야 한다.

학습 데이터 업로드 하기

데이터를 업로드하기전에, 구글 클라우드 콘솔에서 구글 클라우드 스토리지 버킷을 생성한다. 생성된 버킷명을 YOUR_GCS_BUCKET 환경 변수에 저장한다.

```
export YOUR_GCS_BUCKET=${YOUR_GCS_BUCKET}
```

다음 gsutil 유틸리티를 이용하여 YOUR_GCS_BUCKET 버킷으로 학습용 데이터와, 라벨맵 데이터를 업로드 한다.


```
gsutil cp pet_train.record gs://${YOUR_GCS_BUCKET}/data/pet_train.record
gsutil cp pet_val.record gs://${YOUR_GCS_BUCKET}/data/pet_val.record
gsutil cp object_detection/data/pet_label_map.pbtxt
gs://${YOUR_GCS_BUCKET}/data/pet_label_map.pbtxt
```

학습된 모델 다운로드 받아서 업로드 하기

다음은 학습된 모델을 받아서, 그중에서 체크포인트를 GCS에 올린다.

```
curl -O
http://storage.googleapis.com/download.tensorflow.org/models/object_detection/faster_rcnn_resnet101_coco_11_06_2017.tar.gz
tar -xvf faster_rcnn_resnet101_coco_11_06_2017.tar.gz
gsutil cp faster_rcnn_resnet101_coco_11_06_2017/model.ckpt.* gs://${YOUR_GCS_BUCKET}/data/
```

체크 포인트를 다운받아서 업로드 하는 이유는, 트랜스퍼 러닝 (Transfer Learning)을 하기 위함인데, 하나도 학습이 되지 않은 모델을 학습을 시키는데는 시간이 많이 들어간다. 트랜스퍼러닝은 이미 학습이 되어 있는 모델로 다른 데이터를 학습 시키는 방법인데, 사물을 인식하는 상태로 학습되어 있는 모델을 다른 물체 (여기서는 애완동물)를 학습하는데 사용하면 학습 시간을 많이 줄 일 수 있다. 이런 이유로, 사물 인식용으로 학습된 체크포인트를 로딩해서 이 체크포인트 부터 학습을 하기 위함이다.

설정 파일 변경하기

Object Detection API를 사용하기 위해서는 학습에 대한 설정 정보를 정의해야 한다.

이 설정 파일안에는 학습 데이터의 위치, 클래스의 수 및 각종 하이퍼 패러미터들이 정의되어 있다. 패러미터에 대한 자세한 설명은

https://github.com/tensorflow/models/blob/master/object_detection/g3doc/configuring_jobs.md를 참고하기 바란다. 이 예제에서는 설정 파일을 따로 만들지 않고 애완동물 사진 학습을 위해서 미리 정의되어 있는 템플릿 설정 파일을 이용하도록 한다. 설정 파일은 미리 정의된 모델에 따라 다른데, 여기서는 faster_rcnn_resnet101_pets 모델을 사용하기 때문에 object_detection/samples/configs/faster_rcnn_resnet101_pets.config 파일을 사용한다.

파일의 위치가 PATH_TO_BE_CONFIGURED 문자열로 정의되어 있는데, 이를 앞에서 만든 GCS 버킷명으로 변경해야 하기 때문에, 아래와 같이 sed 명령을 이용하여 해당 문자열을 변경하자

```
Linux : sed -i "s|PATH_TO_BE_CONFIGURED|gs://${YOUR_GCS_BUCKET}/data|g"
object_detection/samples/configs/faster_rcnn_resnet101_pets.config
```

```
Max : sed -i " -e "s|PATH_TO_BE_CONFIGURED|gs://${YOUR_GCS_BUCKET}/data|g"
object_detection/samples/configs/faster_rcnn_resnet101_pets.config
```


설정 파일 작성이 끝났으면 이를 GCS 버킷에 올린 후에, 학습시에 사용하도록 한다. 다음 명령어는 설정 파일을 GCS 버킷에 올리는 명령이다.

```
gsutil cp object_detection/samples/configs/faster_rcnn_resnet101_pets.config \
gs://${YOUR_GCS_BUCKET}/data/faster_rcnn_resnet101_pets.config
```

텐서플로우 코드 패키징 및 업로드

학습에 사용할 데이터와 체크포인트등을 업로드 했으면, 다음 텐서플로우 코드를 패키징 해야 한다. 이 글에서는 학습을 로컬 머신이 아니라 구글 클라우드의 텐서플로우 메니지드 서비스인 CloudML을 사용하는데, 이를 위해서는 텐서플로우코드와 코드에서 사용하는 파이썬 라이브러리들을 패키징해서 올려야 한다.

Object Detection API 모델 디렉토리에서 다음 명령어를 실행하면, model 디렉토리와 model/slim 디렉토리에 있는 텐서플로우 코드 및 관련 라이브러리를 같이 패키징하게된다.

```
# From tensorflow/models/
python setup.py sdist
(cd slim && python setup.py sdist)
```

명령을 실행하고 나면 패키징된 파일들은 dist/object_detection-0.1.tar.gz 와 slim/dist/slim-0.1.tar.gz 에 저장되게 된다.

학습하기

구글 CloudML을 이용하여 학습하기. 그러면 학습을 시작해보자. 학습은 200,000 스텝에 총 17시간 정도가 소요되며, 비용이 3000\$ 이상이 소요되니, 비용이 넉넉하지 않다면, 학습을 중간에 중단 시키기를 권장한다. 테스트 목적이라면 약 10~20분 정도면 충분하지 않을까 한다. 아니면 앞의 config 파일에서 training step을 작게 낮춰서 실행하기 바란다.

```
# From tensorflow/models/
gcloud ml-engine jobs submit training `whoami`_object_detection_`date +%s` \
--job-dir=gs://${YOUR_GCS_BUCKET}/train \
--packages dist/object_detection-0.1.tar.gz,slim/dist/slim-0.1.tar.gz \
--module-name object_detection.train \
--region asia-east1 \
--config object_detection/samples/cloud/cloud.yml \
-- \
--train_dir=gs://${YOUR_GCS_BUCKET}/train \
```

```
--pipeline_config_path=gs://${YOUR_GCS_BUCKET}/data/faster_rcnn_resnet101_pets.conf  
ig
```

학습을 시킬 텐서플로우 클러스터에 대한 정보는 `object_detection/samples/cloud/cloud.yml`에 들어 있다. 내용을 보면,

trainingInput:

```
runtimeVersion: "1.0"  
scaleTier: CUSTOM  
masterType: standard_gpu  
workerCount: 5  
workerType: standard_gpu  
parameterServerCount: 3  
parameterServerType: standard
```

scaleTier로 클러스터의 종류를 정의할 수 있는데, 서버 1대에서 부터 여러대의 클러스터까지 다양하게 적용이 가능하다. 여기서는 모델이 크기가 다소 크기 때문에, Custom으로 설정하였다.

역할	서버 타입	댓수
Master server	standard_gpu	1
Worker	standard_gpu	5
Parameter Server	standard	5

각 서버의 스펙은 상세 스펙은 나와있지 않고, 상대값으로 정의되어 있는데 대략 내용이 다음과 같다.

Comparing machine types

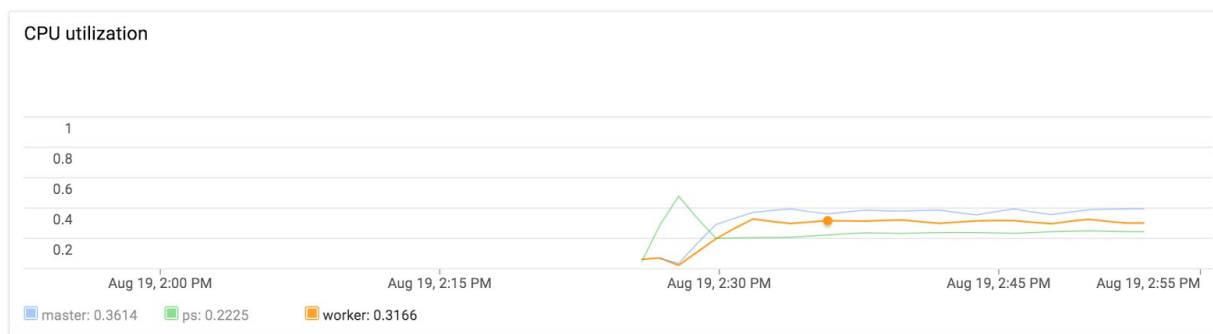
Even though the exact specifications of the machine types are subject to change at any time, you can compare them in terms of relative capability. The following table uses rough "t-shirt" sizing to describe the machine types.

Machine type	CPU	GPUs	Memory	ML units
standard	XS	-	M	1
large_model	S	-	XL	3
complex_model_s	S	-	S	2
complex_model_m	M	-	M	3
complex_model_l	L	-	L	6
standard_gpu	XS	1	M	3
complex_model_m_gpu	M	4	M	12
complex_model_l_gpu	L	8	L	24

Each increase in size constitutes roughly double capacity in the area being measured. Possible sizes are (in increasing order): XS, S, M, L, XL, XXL.

출처 https://cloud.google.com/ml-engine/docs/concepts/training-overview#machine_type_table

학습을 시작하고 나면 CloudML 콘솔에서 실행중인 Job을 볼 수 있고, Job을 클릭하면 자원의 사용 현황을 볼 수 있다. (CPU와 메모리 사용량)



학습을 시작한 후에, 학습된 모델을 Evaluate할 수 있는데, Object Detection API에서는 학습 말고 Evaluation 모델을 별도로 나눠서, 잡을 나눠서 수행하도록 하였다. 학습중에 생성되는 체크포인트 파일을 읽어서 Evaluation을 하는 형태이다. 다음을 Evaluation을 실행하는 명령어인데, 위의 학습 작업이 시작한 후에, 한시간 정도 후부터 실행해도 실행 상태를 볼 수 있다.

```
# From tensorflow/models/
gcloud ml-engine jobs submit training `whoami`_object_detection_eval_`date +%s` \
  --job-dir=gs://${YOUR_GCS_BUCKET}/train \
  --packages dist/object_detection-0.1.tar.gz,slim/dist/slim-0.1.tar.gz \
  --module-name object_detection.eval \
  --region asia-east1 \
  --scale-tier BASIC_GPU \
  -- \
  --checkpoint_dir=gs://${YOUR_GCS_BUCKET}/train \
  --eval_dir=gs://${YOUR_GCS_BUCKET}/eval \

--pipeline_config_path=gs://${YOUR_GCS_BUCKET}/data/faster_rcnn_resnet101_pets.conf
ig
```

Training 17:08 us-central : "consumedMLUnits": 358.03 = 3005\$
 Evaluation 04:41 us-central : "consumedMLUnits": 13.72 = 31.37\$

1000 스텝으로 줄였을 경우

Training 00:09:22 us-central : "consumedMLUnits": 298.83 = 24\$

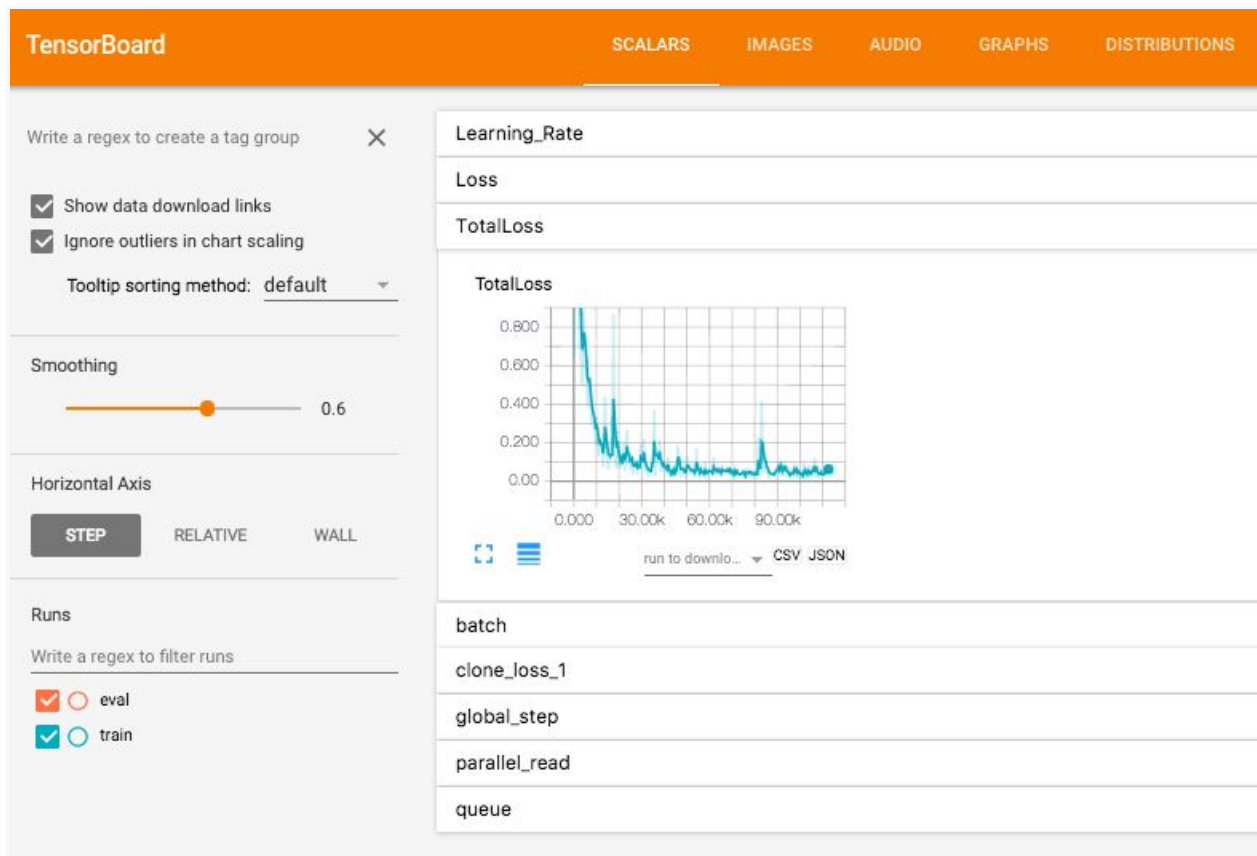
학습 진행 상황 확인하기

학습이 진행중에도, Evaluation을 시작했으면, Tensorboard를 이용하여 학습 진행 상황을 모니터링 할 수 있다. 학습 진행 데이터가 gs://\${YOUR_GCS_BUCKET}에 저장되기 때문에, 이 버킷에 있는 데이터를 Tensorboard로 모니터링 하면 된다.

실행 방법은 먼저 GCS에 접속이 가능하도록 auth 정보를 설정하고, Tensorboard에 로그 파일 경로를 GCS 버킷으로 지정하면 된다.

```
gcloud auth application-default login
tensorboard --logdir=gs://${YOUR_GCS_BUCKET}
```

아래는 실제 실행 결과이다.



Evaluataion이 끝났으면, 테스트된 이미지도 IMAGES 탭에서 확인이 가능하다.

TensorBoard

SCALARSIMAGESAUDIO

Write a regex to create a tag group

X

Runs

Write a regex to filter runs

☒ ☐ eval

☒ ☐ train

image-0

image-1

image-1

eval

step 200006 (Mon Aug 21 2017 13:54:00 GMT+0900 (KST))

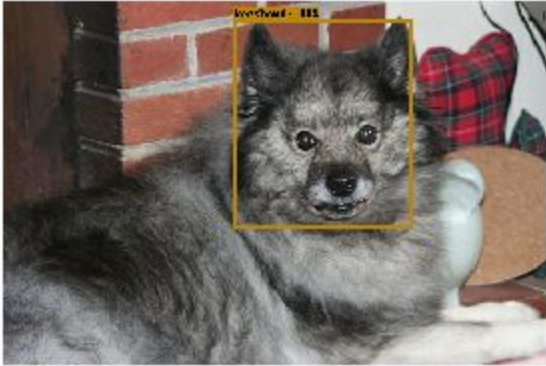



image-2

image-2

eval

step 200006 (Mon Aug 21 2017 13:54:01 GMT+0900 (KST))



















학습된 모델을 Export 하기

학습이 완료되었으면, 이 모델을 예측 (Prediction)에 사용하기 위해서 Export 할 수 있다. 이렇게 Export 된 이미지는 나중에 다시 로딩하여 예측(Prediction)코드에서 로딩을 하여 사용이 가능하다.

`${YOUR_GCS_BUCKET}`에 가면 체크 포인트 파일들이 저장되어 있는데, 이 체크 포인트를 이용하여 모델을 Export 한다.

Buckets / [terrycho-petdetection](#) / train

<input type="checkbox"/>	 graph.pbtxt
<input type="checkbox"/>	 model.ckpt-195370.data-00000-of-00001
<input type="checkbox"/>	 model.ckpt-195370.index
<input type="checkbox"/>	 model.ckpt-195370.meta
<input type="checkbox"/>	 model.ckpt-197676.data-00000-of-00001
<input type="checkbox"/>	 model.ckpt-197676.index
<input type="checkbox"/>	 model.ckpt-197676.meta
<input type="checkbox"/>	 model.ckpt-200003.data-00000-of-00001
<input type="checkbox"/>	 model.ckpt-200003.index
<input type="checkbox"/>	 model.ckpt-200003.meta
<input type="checkbox"/>	 model.ckpt-200005.data-00000-of-00001
<input type="checkbox"/>	 model.ckpt-200005.index
<input type="checkbox"/>	 model.ckpt-200005.meta
<input type="checkbox"/>	 model.ckpt-200006.data-00000-of-00001
<input type="checkbox"/>	 model.ckpt-200006.index
<input type="checkbox"/>	 model.ckpt-200006.meta

GCS 버킷에서 Export 하고자 하는 Check Point 번호를 선택한 후에 Export 하면 된다, 여기서는 200006 Check Point를 Export 해보겠다.

`${CHECKPOINT_NUMBER}` 환경 변수를
`export CHECKPOINT_NUMBER=200006`
으로 설정한 다음에 다음 명령어를 실행한다.

```
# From tensorflow/models
gsutil cp gs://$YOUR_GCS_BUCKET/train/model.ckpt-$(CHECKPOINT_NUMBER).*.
python object_detection/export_inference_graph.py \
  --input_type image_tensor \
  --pipeline_config_path
object_detection/samples/configs/faster_rcnn_resnet101_pets.config \
  --trained_checkpoint_prefix model.ckpt-$(CHECKPOINT_NUMBER) \
  --output_directory output_inference_graph.pb
```

명령을 실행하고 나면 output_inference_graph.pb 디렉토리에 모델이 Export 된것을 확인할 수 있다.

- 모델을 변경하는 자료는 https://github.com/tensorflow/models/blob/master/object_detection/g3doc/configuring_jobs.md 를 참고하기 바라고,
- 학습된 모델을 이용하여 예측 하는 예제는 https://github.com/tensorflow/models/blob/master/object_detection/g3doc/running_notebook.md 를 참고하기 바란다.

참고 자료

- https://github.com/tensorflow/models/blob/master/object_detection/g3doc/running_pets.md

Object Detection API로 연예인 얼굴 학습 하기

이번글에서는 Tensorflow Object Detection API를 이용하여 직접 이미지를 인식할 수 있는 방법에 대해서 알아보자. 이미 가지고 있는 데이터를 가지고 다양한 상품에 대한 인식이나, 사람 얼굴에 대한 인식 모델을 머신러닝에 대한 전문적인 지식 없이도 손쉽게 만들 수 있다. .

학습용 데이터 데이터 생성 및 준비

Object Detection API를 학습 시키기 위해서는 <http://bcho.tistory.com/1193> 예제와 같이 TFRecord 형태로 학습용 파일과 테스트용 파일이 필요하다. TFRecord 파일 포맷에 대한 설명은 <http://bcho.tistory.com/1190> 를 참고하면 된다.

이미지 파일을 TFRecord로 컨버팅하는 전체 소스 코드는

https://github.com/bwcho75/objectdetection/blob/master/custom/create_face_data.py 를 참고하기 바란다.

구글 클라우드 VISION API를 이용하여, 얼굴이 있는지 여부를 파악하고, 얼굴 각도가 너무 많이 틀어진 경우에는 필터링 해낸 후에, 얼굴의 위치 좌표를 추출하여 TFRecord 파일에 쓰는 흐름이다.

VISION API를 사용하기 때문에 반드시 서비스 어카운트 (Service Account/JSON 파일)를 구글 클라우드 콘솔에서 만들어서 설치하고 실행하기 바란다.

사용 방법은

```
python create_face_data.py {이미지 소스 디렉토리} {이미지 아웃풋 디렉토리}
{TFRECORD 파일명}
```

형태로 사용하면 된다.

```
예) python ./custom/create_face_data.py /Users/terrycho/trainingdata_source
/Users/terrycho/trainingdata_out
```

{이미지 소스 디렉토리} 구조는 다음과 같다.

{이미지 소스 디렉토리}/{라벨1}

{이미지 소스 디렉토리}/{라벨2}

{이미지 소스 디렉토리}/{라벨3}

:

예를 들어

/Users/terrycho/trainingdata_source/**Alba**

/Users/terrycho/trainingdata_source/**Jessica**

/Users/terrycho/trainingdata_source/**Victoria**

:

이런식이 된다.

명령을 실행하면, {이미지 아웃풋 디렉토리} 아래

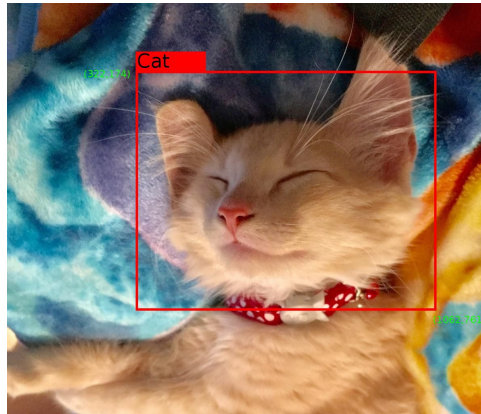
- 학습 파일은 face_training.record
- 테스트 파일은 face_evaluation.record
- 라벨맵은 face_label_map.pbtxt

로 생성된다. 이 세가지 파일이 Object Detection API를 이용한 학습에 필요하고 부가적으로 생성되는 csv 파일이 있는데

- all_files.csv : 소스 디렉토리에 있는 모든 이미지 파일 목록
- filtered_files.csv : 각 이미지명과, 라벨, 얼굴 위치 좌표 (사각형), 이미지 전체 폭과 높이
- converted_result_files.csv : filtered_files에 있는 이미지중, 얼굴의 각도등이 이상한 이미지를 제외하고 학습과 테스트용 데이터 파일에 들어간 이미지 목록으로, 이미지 파일명, 라벨 (텍스트), 라벨 (숫자), 얼굴 좌표 (사각형) 을 저장한다.

여기서 사용한 코드는 간단한 테스트용 코드로, 싱글 쓰레드에 싱글 프로세스 모델로 대규모의 이미지를 처리하기에는 적절하지 않기 때문에, 운영환경으로 올리려면, Apache Beam 등 분산 프레임워크를 이용하여 병렬 처리를 하는 것을 권장한다. <http://bcho.tistory.com/1177> 를 참고하기 바란다.

여기서는 학습하고자 하는 이미지의 바운드리(사각형 경계)를 추출하는 것을 VISION API를 이용해서 자동으로 했지만, 일반적인 경우는 이미지에서 각 경계를 수동으로 추출해서 학습데이터로 생성해야 한다



이런 용도로 사용되는 툴은

<https://medium.com/towards-data-science/how-to-train-your-own-object-detector-with-tensorflows-object-detector-api-bec72ecfe1d9> 문서에 따르면 FastAnnotationTool이나 ImageMagick 과 같은 툴을 추천하고 있다.

- <https://github.com/christopher5106/FastAnnotationTool>
- <http://imagemagick.org/script/index.php#>

이렇게 학습용 파일을 생성하였으면 다음 과정은 앞의 <http://bcho.tistory.com/1193> 에서 언급한 절차와 크게 다르지 않다.

체크포인트 업로드

학습 데이터가 준비 되었으면 학습을 위한 준비를 하는데, 트랜스퍼 러닝 (Transfer learning)을 위해서 기존의 학습된 체크포인트 데이터를 다운 받아서 이를 기반으로 학습을 한다.

Tensorflow Object Detection API는 경량이고 단순한 모델에서 부터 정확도가 비교적 높은 복잡한 모델까지 지원하고 있지만, 복잡도가 높다고 해서 정확도가 꼭 높지는 않을 수 있다. 복잡한 모델일 수록 학습 데이터가 충분해야 하기 때문에, 학습하고자 하는 데이터의 양과 클래스의 종류에 따라서 적절한 모델을 선택하기를 권장한다.

여기서는 faster_rcnn_inception_resnet_v2 모델을 이용했기 때문에 아래와 같이 해당 모델의 체크포인트 데이터를 다운로드 받는다.

```
curl -O
```

```
http://download.tensorflow.org/models/object\_detection/faster\_rcnn\_inception\_resnet\_v2\_atrous\_coco\_11\_06\_2017.tar.gz
```

파일의 압축을 푼 다음 체크 포인트 파일을 학습 데이터용 Google Cloud Storage (GCS) 버킷으로 업로드 한다.

```
gsutil cp faster_rcnn_inception_resnet_v2_atrous_coco_11_06_2017/model.ckpt.*  
gs://${YOUR_GCS_BUCKET}/data/
```

설정 파일 편집 및 업로드

다음 학습에 사용할 모델의 설정을 해야 하는데, object_detection/samples/configs/ 디렉토리에 각 모델별 설정 파일이 들어 있으며, 여기서는 faster_rcnn_inception_resnet_v2_atrous_pets.config 파일을 사용한다.

이 파일에서 수정해야 하는 부분은 다음과 같다.

클래스의 수

클래스 수를 정의한다. 이 예제에서는 총 5개의 클래스로 분류를 하기 때문에 아래와 같이 5로 변경하였다.

```
8 model {  
9   faster_rcnn {  
10     num_classes: 5  
11     image_resizer {
```

학습 데이터 파일 명 및 라벨명

학습에 사용할 학습데이터 파일 (tfrecord)와 라벨 파일명을 지정한다.

```
126 train_input_reader: {  
127   tf_record_input_reader {  
128     input_path: "gs://terrycho-facedetection/data/face_training.record"  
129   }  
130   label_map_path: "gs://terrycho-facedetection/data/face_label_map.pbtxt"  
131 }
```

테스트 데이터 파일명 및 라벨 파일명

학습 후 테스트에 사용할 테스트 파일 (tfrecord)과 라벨 파일명을 지정한다

```
140 eval_input_reader: {
141   tf_record_input_reader {
142     input_path: "gs://terrycho-facedetection/data/face_evaluation.record"
143   }
144   label_map_path: "gs://terrycho-facedetection/data/face_label_map.pbtxt"
145   shuffle: false
146   num_readers: 1
```

만약에 학습 횟수(스텝)을 조정하고 싶으면 num_steps 값을 조정한다. 디폴트 설정은 20만회인데, 여기서는 5만회로 수정하였다.

```
117   # never decay). Remove the below line to train indefinitely.
118   # num_steps: 200000
119   num_steps: 50000
120   data_augmentation_options {
121     random_horizontal_flip {
122     }
```

설정 파일 수정이 끝났으면 gsutil cp 명령을 이용하여 해당 파일을 GCS 버킷에 다음과 같이 업로드 한다.

```
gsutil cp
object_detection/samples/configs/faster_rcnn_inception_resnet_v2_atrous_pets.co
nfig
gs://${YOUR_GCS_BUCKET}/data/faster_rcnn_inception_resnet_v2_atrous_pets.co
nfig
```

코드 패키징

models/ 디렉토리에서 다음 명령을 수행하여, 모델 코드를 패키징한다.

```
python setup.py sdist
(cd slim && python setup.py sdist)
```

학습

```
gcloud ml-engine jobs submit training `whoami`_object_detection_`date +%s` \
  --job-dir=gs://${YOUR_GCS_BUCKET}/train \
  --packages dist/object_detection-0.1.tar.gz,slim/dist/slim-0.1.tar.gz \
  --module-name object_detection.train \
  --region asia-east1 \
  --config object_detection/samples/cloud/cloud.yml \
```

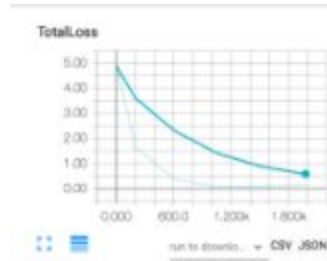
```
-- \
--train_dir=gs://${YOUR_GCS_BUCKET}/train \

--pipeline_config_path=gs://${YOUR_GCS_BUCKET}/data/faster_rcnn_resnet101_
pets.config
```

모니터링

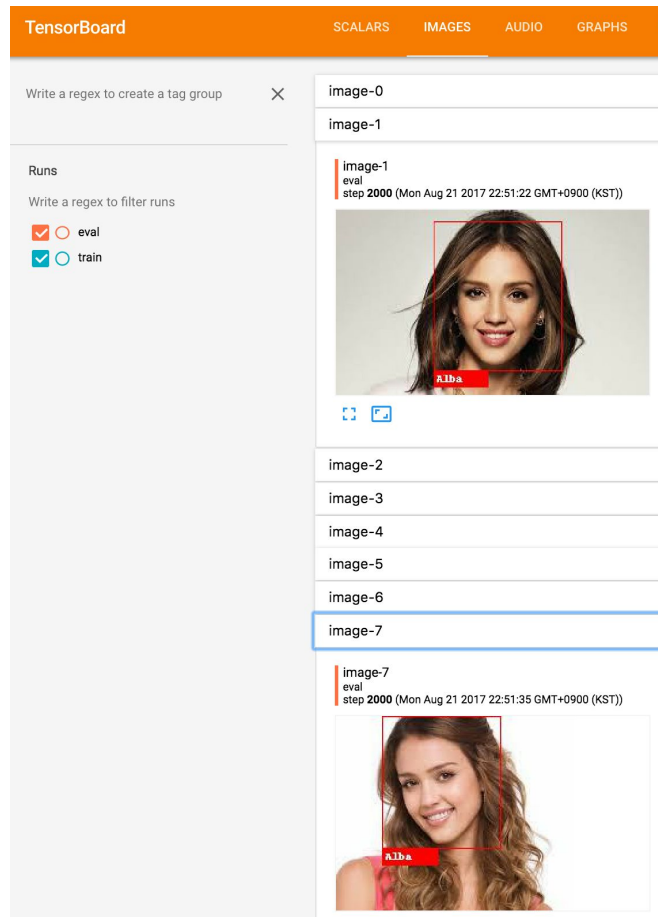
학습이 진행되면 텐서보드를 이용하여 학습 진행 상황을 모니터링할 수 있고, 또한 테스트 트레이닝을 수행하여, 모델에 대한 테스트를 동시 진행할 수 있다. <http://bcho.tistory.com/1193> 와 방법이 동일하니 참고하기 바란다.

학습을 시작하면 텐서보드를 통해서, Loss 값이 수렴하는 것을 확인할 수 있다.



결과

학습이 끝나면 텐서보드에서 테스트된 결과를 볼 수 있다. 이 예제의 경우 모델을 가장 복잡한 모델을 사용했는데 반하여, 총 5개의 클래스에 대해서 클래스당 약 40개 정도의 학습 데이터를 사용했는데, 상대적으로 정확도가 낮았다. 실 서비스에서는 더 많은 데이터를 사용하기를 권장한다.



11. 텐서 플로우 하이레벨 API

머신러닝을 공부하고 구현하다 보니, 모델 개발은 새로운 모델이나 알고리즘을 개발하는 일 보다는, 기존의 알고리즘을 습득해서 내 데이터 모델에 맞도록 포팅하고, 학습 시키는 것이 주된 일이 되고, 오히려, 모델 보다는 데이터에 대한 이해와 전처리에 많은 시간이 소요되었다.

특히 여러번 실험을 하면서 패러미터를 조정하고 피쳐등을 조정하기 위해서는 많은 실험을 할 수 있어야 하는데, 이러기 위해서는 실험(학습)시간이 짧아야 한다. 이를 위해서는 모델 개발 보다 분산 러닝을 하기 위한 코드 변경 작업등이 많이 소요된다.

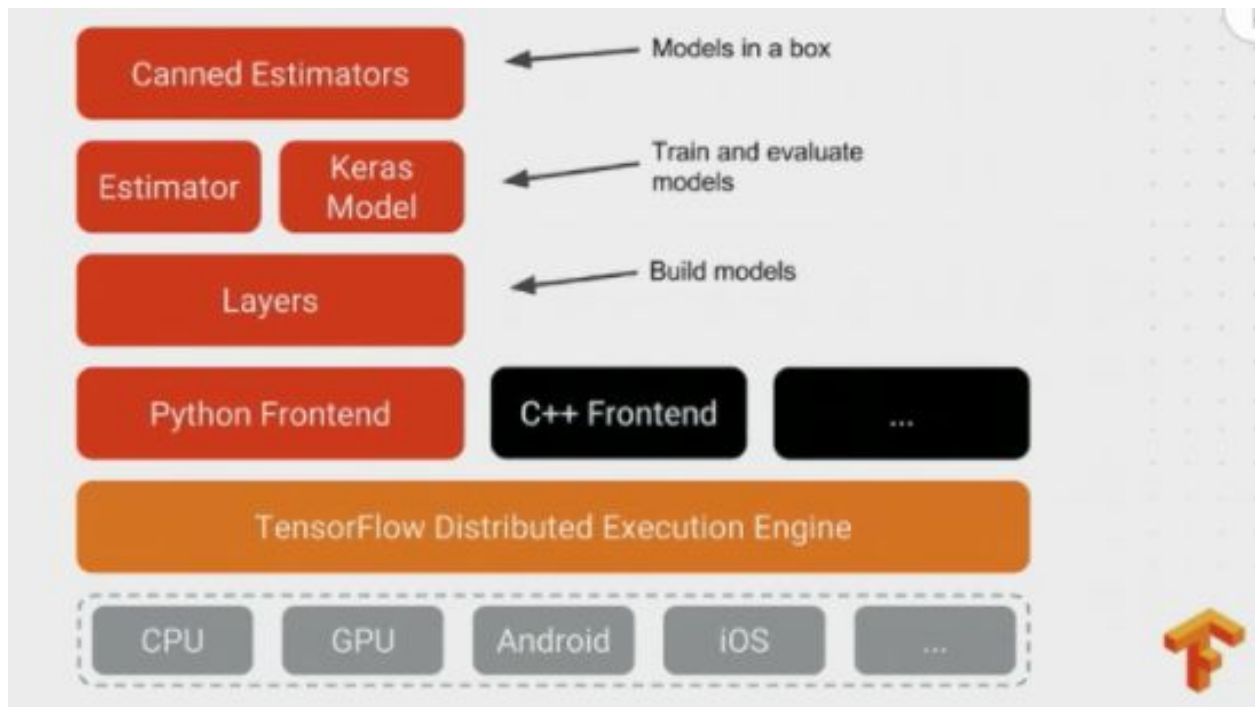
결론을 요약하자면, 실제로 알고리즘을 개발하는 데이터 과학자가 아니라, 머신러닝을 활용만 하는 프랙티셔너 입장이라면, 모델을 개발하는 것 보다는 있는 모델을 선택해서 쉽게 사용할 수 있는 방법을 찾으면 된다.

이런 관점에서 시작한 것이 머신러닝 하이레벨 API 이다. 복잡한 수식이 없이 마치 함수처럼 모델을 선택해서 사용하는 방법인데, 쉽게 이야기 하면, Hash table 알고리즘을 100% 이해하지 않더라도, hashtable 라이브러리를 가져다가 사용하면 되는 것과 같은 원리이다.

머신러닝에서도 이미 이러한 하이레벨 API가 많이 제공되고 있는데, 파이썬 싸이킷런(<http://scikit-learn.org/>) 이나 SparkML 등이 해당한다.

텐서플로워에도 같은 방식으로 하이레벨 API를 제공하는데, 텐서플로워 공식 SDK와 써드파티 오픈소스 라이브러리들이 있다.

그중에서 tf.contrib가 공식 텐서플로워의 하이레벨 API이며, 딥러닝 모델을 간단하게 만들 수 있는 Keras역시 얼마전에 텐서플로워 공식 하이레벨 API로 로 편입되었다.



텐서플로워에서는 Linear regression, SVM등 많이 쓰이는 일반적인 머신러닝 모델에서 부터 Deep Wide Network와 같은 딥 러닝 모델들을 Estimator 라는 형태로 제공하고 있다.

하이레벨 API를 쓰면 장점

그러면 이러한 하이레벨 API를 쓰면 장점이 무엇일까?

모델 개발이 쉽다

모델 개발이 매우 쉽다. 복잡한 모델을 손쉽게 개발할 수 있을뿐더러, 일부 모델들은 Out of box 형태로, 바로 라이브러리 식으로 불러서 사용만 하면 되기 때문에 모델 개발 시간이 줄어들고, 모델에 대한 기본적인 이해만 있더라도 쉽게 개발이 가능하다.

스케일링이 용이하다

큰 모델을 많은 데이터로 학습하기 위해서는 여러 머신에서 학습을 하는 분산 학습이 필요한데, 로우레벨 API를 이용할 경우 분산 학습을 개발하기가 쉽지 않다. 하이레벨 API를 이용할 경우 코드 변경 없이 싱글 머신에서부터 GPU 그리고 분산 학습까지 손쉽게 지원이 되기 때문에, 실험 (학습/테스트) 시간을 많이 절약할 수 있다.

배포가 용이하다

모델을 학습 시킨 후 예측을 위해서 배포를 할 경우, 보통 모델을 *.pb 파일 형태로 Export 해야 하는데, 이 경우 학습에 사용된 그래프 말고 예측을 위한 그래프를 새로 그려야 하는 등 추가적인 작업이 필요하고 쉽지 않은데 반해 하이레벨 API의 경우, 코드 몇줄만으로도 손쉽게 예측 서비스를 위한 그래프를 Export할 수 있다.

텐서플로우 하이레벨 API

tf.layers

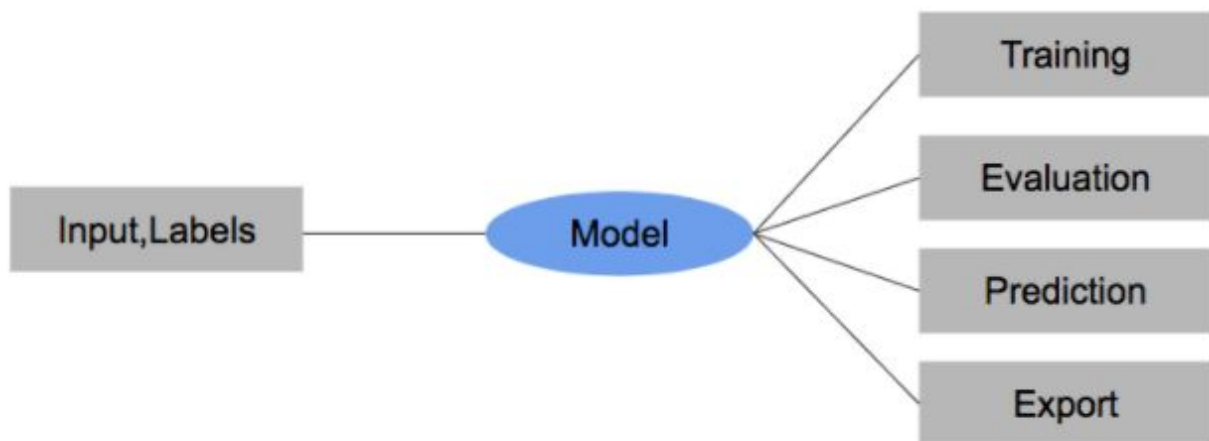
텐서플로우는 특히 딥러닝 (뉴럴네트워크)에 강점을 가지고 있는데, 딥네트워크의 각 계층을 설계 하기 위해서는 컨볼루셔널 필터, 풀링, 스트라이드,드롭 아웃 등 다양한 기법을 사용하게 된다. 이러한 것들을 복잡하게 구현하지 않고, 딥 네트워크를 손쉽게 만들 수 있게 각 레이어에 대한 구현을 함수식으로 제공한다.

다음 그림은 tf.layer로 컨볼루셔널 네트워크 (CNN)을 구현한 예제로 컨볼루셔널 레이어와, 맥스풀링, 드롭아웃, ReLu 액티베이션 함수등을 사용하였다. 각 레이어는 tf.layers 라이브러리 하나씩으로 간단하게 구현되었다.

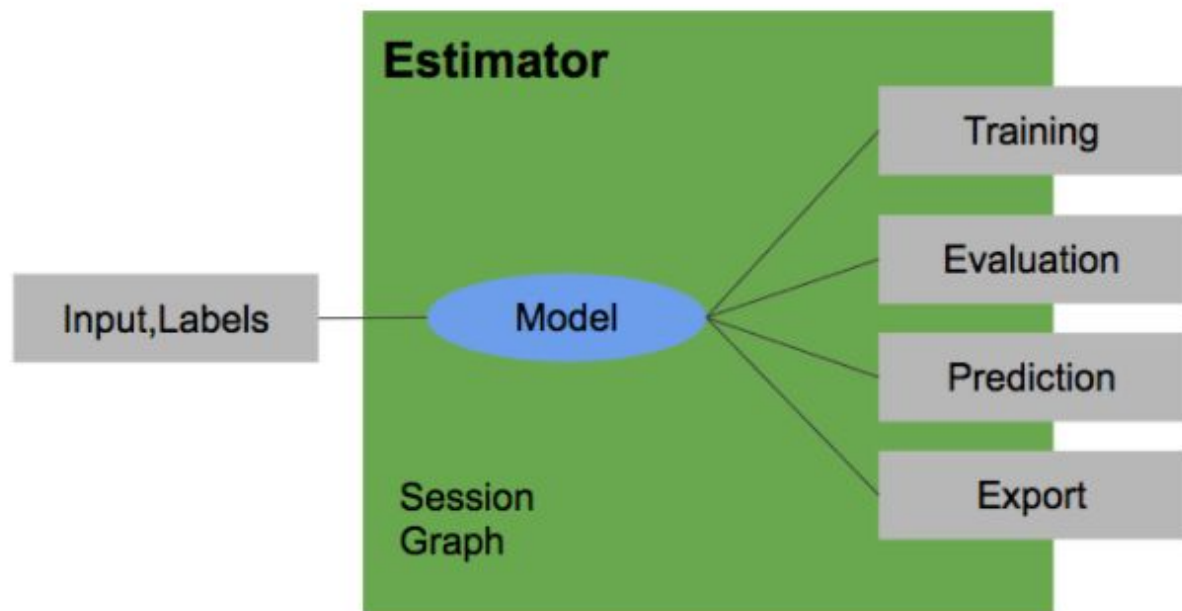


Estimator

일반적으로 머신러닝 개발은 다음과 같은 구조를 갖는다



개발한 모델에 Input, Labels 데이터를 넣은 후, 학습(Training), 테스트(Evaluation), 예측(Prediction)을 한 후, 학습이 완료된 모델을 저장(Export)하여 배포한다. 거의 모든 모델 개발이 위의 구조를 반복하기 때문에, 이러한 구조를 추상화 해놓은 것이 Estimator 이다.



이 추상화를 통해서 Estimator에 데이터를 넣게 되면, Estimator는 Training, Evaluation, Prediction, Export를 위한 인터페이스를 제공한다. 텐서플로우 그래프 구축이나 세션 관리등은 모두 Estimator 안으로 추상화 한다.

Estimator는 직접 개발자가 모델을 직접 구현하여 Estimator를 개발할 수 도 있고 (Custom Estimator) 또는 이미 텐서플로우 `tf.contrib.learn`에 에 미리 모델들이 구현되어 있다. 딥네트워크 기반의 Classifier나 Regressor (DNNClassifier, DNNRegressor), SVM, RNN, KMeans 등이 있기 때문에 간단하게 불러다 사용하기만 하면 된다.

Estimator 예제

Estimator 예제로 간단한 LinearRegression Estimator를 사용하는 예제를 보자

학습용 데이터

먼저 학습용 데이터와 라벨을 생성하였다.

```

import numpy as np
num_points = 300
vectors_set = []
for i in xrange(num_points):
    x = np.random.normal(5,5)+15
    y = x*2+ (np.random.normal(0,3))*2
    vectors_set.append([x,y])

```

```

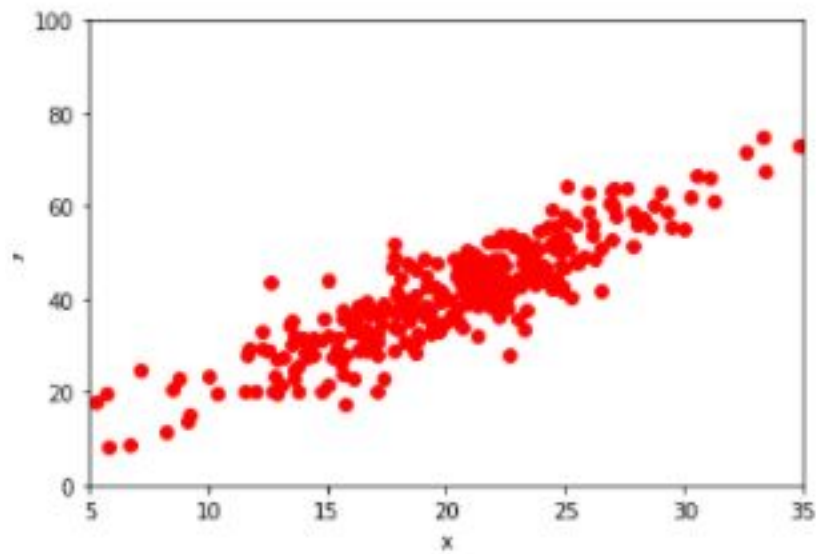
x_data = [v[0] for v in vectors_set ]

```

```
y_data = [v[1] for v in vectors_set ]
```

```
import matplotlib.pyplot as plt
plt.plot(x_data,y_data,'ro')
plt.ylim([0,100])
plt.xlim([5,35])
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```

데이터 분포는 아래와 같다.



모델 코드

데이터 리더

Estimator 를 사용하려면 데이터를 읽어서 Estimator 에 넣어주는 입력 함수를 구현해줘야 한다. 아래는 numpy 배열에서 데이터를 읽어서 리턴해주는 입력 함수이다.

```
input_fn_train = tf.estimator.inputs.numpy_input_fn(
    x = {"x":np.array(x_data[:200],dtype=np.float32)},
    y = np.array(y_data[:200],dtype=np.float32),
    num_epochs=100000,
    batch_size=50,
    shuffle=True
)
```

x_data 배열에서 0~200까지의 데이터를 학습용 데이터로 사용하였고, y_data 0~200을 라벨로 사용하였다. 한번에 50 개씩 리턴하도록 배치를 설정하였고, 100K epoch를 지원하고 데이터를 랜덤하게 리턴하도록 셔플 처리를 하였다.

```
input_fn_eval = tf.estimator.inputs.numpy_input_fn(  
    x = {"x":np.array(x_data[200:300],dtype=np.float32)},  
    y = np.array(y_data[200:300],dtype=np.float32),  
    num_epochs=100000,  
    batch_size=50,  
    shuffle=True  
)
```

```
input_fn_predict = tf.estimator.inputs.numpy_input_fn(  
    x = {"x":np.array([15,20,25,30],dtype=np.float32)},  
    num_epochs=1,  
    shuffle=False  
)
```

같은 방법으로 테스트용 데이터와 예측에 사용할 데이터 입력 함수를 같이 정의하였다.

모델 정의

```
column_x = tf.feature_column.numeric_column("x",dtype=tf.float32)  
columns = [column_x]
```

읽어온 데이터에서, 어떤 컬럼을 학습에 사용할지, 그리고 그 컬럼의 데이터 타입 (연속형인지 분류형인지)를 정한다. `tf.feature_column.numeric_column("x",dtype=tf.float32)` 는 컬럼 명 x를 학습 데이터로 사용하고 x는 연속형 변수로 지정하였다.

다음 columns에 피쳐로 사용할 컬럼 목록을 정한다.

LinearRegression Estimator를 정의하고, 여기에, column을 정해준다. Optimizer나 Learning Rate등은 지정이 가능하다.

```
estimator = tf.contrib.learn.LinearRegressor(feature_columns=columns,optimizer="Adam")  
학습과 예측
```

학습은 .fit 이라는 메서드를 사용하면 되고, 입력 함수와 학습 스텝을 정해주면 된다.

```
estimator.fit(input_fn = input_fn_train,steps=5000)  
estimator.evaluate(input_fn = input_fn_eval,steps=10)  
result = list(estimator.predict(input_fn = input_fn_predict))
```

마지막으로 예측은 predict 를 이용하면 된다.

x=15,20,25,30에 대해서 예측 결과는 다음과 같다.

[31.193062, 41.855644, 52.51823, 63.180817]

텐서플로우의 하이레벨 API를 이용하기 위해서는 Estimator 를 사용하는데, Estimator 는 Predefined model 도 있지만, 직접 모델을 구현할 수 있다.

이 문서는 Custom Estimator를 이용하여 Estimator를 구현하는 방법에 대해서 설명하고 있으며, 대부분 <https://www.tensorflow.org/extend/estimators> 의 내용을 참고하여 작성하였다.

Custom Estimator

Estimator의 스케레톤 코드는 다음과 같다. 모델을 정의하는 함수는 학습을 할 feature와, label을 입력 받고, 모델의 모드 (학습, 테스트, 예측) 모드를 인자로 받아서 모드에 따라서 모델을 다르게 정의할 수 있다. 예를 들어 학습의 경우 드롭 아웃을 사용하지만 테스트 모드에서는 드롭 아웃을 사용하지 않는다.

```
def model_fn(features, labels, mode, params):
    # Logic to do the following:
    # 1. Configure the model via TensorFlow operations
    # 2. Define the loss function for training/evaluation
    # 3. Define the training operation/optimizer
    # 4. Generate predictions
    # 5. Return predictions/loss/train_op/eval_metric_ops in EstimatorSpec
    object
    return EstimatorSpec(mode, predictions, loss, train_op,
eval_metric_ops)
```

입력 인자에 대한 설명

그러면 각 인자를 구체적으로 살펴보자

- features : input_fn을 통해서 입력되는 feature로 dict 형태가 된다.
- labels : input_fn을 통해서 입력되는 label 값으로 텐서 형태이고, predict (예측) 모드 일 경우에는 비어 있게 된다.
- mode : 모드는 모델의 모드로, tf.estimator.ModeKeys 중 하나를 사용하게 된다.
 - tf.estimator.ModeKeys.TRAIN : 학습 모드로 Estimator의 train()을 호출하였을 경우 사용되는 모드이다.
 - tf.estimator.ModeKeys.EVAL : 테스트 모드로, evaluate() 함수를 호출하였을 경우 사용되는 모드이다.
 - tf.estimator.ModeKeys.PREDICT : 예측모드로, predict() 함수를 호출하였을 경우에 사용되는 모드이다.
- param : 추가적으로 입력할 수 있는 패러미터로, dict 포맷을 가지고 있으며, 하이퍼 패러미터등을 이 변수를 통해서 넘겨 받는다.

Estimator 에서 하는 일

Estimator 를 구현할때, Estimator 내의 내용은 모델을 설정하고, 모델의 그래프를 그린 다음에, 모델에 대한 loss 함수를 정의하고, Optimizer를 정의하여 loss 값의 최소값을 찾는다. 그리고 prediction 값을 계산한다.

Estimator의 리턴값

Estimator에서 리턴하는 값은 `tf.estimator.EstimatorSpec` 객체를 리턴하는데, 이 객체는 다음과 같은 값을 갖는다.

- `mode` : Estimator가 수행한 모드. 보통 입력값으로 받은 모드 값이 그대로 리턴된다.
- `prediction` (PREDICT 모드에서만 사용됨) : PREDICT 모드에서 예측을 수행하였을 경우, 예측된 값을 dict 형태로 리턴한다.
- `loss` (EVAL 또는, TRAIN 모드에서 사용됨) : 학습과 테스트중에 loss 값을 리턴한다.
- `train_op` (트레이닝 모드에서만 필요함) : 한 스텝의 학습을 수행하기 위해서 호출하는 함수를 리턴한다. 보통 옵티마이저의 `minimize()`와 같은 함수가 사용된다.

```
optimizer = tf.train.AdamOptimizer(learning_rate=0.001)
train_op = optimizer.minimize(loss, global_step=global_step)
return tf.estimator.EstimatorSpec(mode, loss=loss, train_op=train_op)
```
- `eval_metrics_ops` (optional) : EVAL (테스트) 모드에서 테스트를 위해서 사용된 인자들을 dict 형태로 리턴한다. `tf.metrics`에는 미리 정의된 일반적인 메트릭들이 정의되어 있는데, 예를 들어 `accuracy` 등이 이에 해당한다. 아래는 `tf.metrics.accuracy`를 이용하여 예측값 (predictions)과 라벨(labels)의 값을 계산하여, 메트릭으로 리턴하는 방법이다.

```
eval_metric_ops = {
    "accuracy": tf.metrics.accuracy(labels, predictions) }
```

만약 `rmse`를 evaluation metric으로 사용하고자 하면 다음과 같이 정의한다.

```
eval_metric_ops = {
    "rmse": tf.metrics.root_mean_squared_error(
        tf.cast(labels, tf.float64), predictions)
}
```

만약에 별도의 메트릭을 정의하지 않으면, 디폴트로 loss 값만 EVAL 단계에서 계산되게 된다.

데이터 입력 처리

모델로의 데이터 입력은 Estimator의 모델 함수로 입력되는 features 변수를 통해서 입력 된다. features는 컬럼명으로된 키와, 컬럼 값으로 이루어진 dict 형태의 데이터 형으로, 뉴럴 네트워크 모델에 데이터를 입력하기 위해서는 이중에서 학습에 사용할 컬럼만을 추출하여, 입력 레이어에 넣어 줘야 한다.

이 features 에서 특정 컬럼만을 지정하여 추출한 후에, 그 컬럼의 값을 넣어주는 것은 [tf.feature_column.input_layer](#) 함수를 사용하면 된다.

예제를 보자

```
input_layer = tf.feature_column.input_layer(  
    features=features, feature_columns=[age, height, weight])
```

위의 예제는 features 에서 age,height,weight 컬럼을 추출하여 input layer로 넣는 코드이다.

네트워크 정의

데이터를 읽었으면 이제 뉴럴네트워크를 구성해야 한다. 네트워크의 레이어는 tf.layers 로 간단하게 구현할 수 있다. tf.layer에는 풀링,드롭아웃,일반적인 뉴럴네트워크의 히든 레이어, 컨볼루셔널 네트워크들이 함수로 구현되어 있기 때문에 각 레이어를 하나의 함수로 간단하게 정의가 가능하다.

아래는 히든레이어를 구현하는 tf.layers.dense 함수이다.

```
tf.layers.dense( inputs, units, activation)
```

- inputs는 앞의 레이어를 정의하고
- units는 이 레이어에 크기를 정의하고
- 마지막으로 activation은 sigmoid나, ReLu와 같은 Activation 함수를 정의한다.

다음 예제는 5개의 히든 레이어를 가지는 오토 인코더 네트워크를 정의한 예이다.

```
input_layer = features['inputs'] # 784 pixels  
dense1 = tf.layers.dense(inputs=input_layer, units=256,  
activation=tf.nn.relu)  
dense2 = tf.layers.dense(inputs=dense1, units=128,  
activation=tf.nn.relu)  
dense3 = tf.layers.dense(inputs=dense2, units=16,  
activation=tf.nn.relu)  
dense4 = tf.layers.dense(inputs=dense3, units=128,  
activation=tf.nn.relu)  
dense5 = tf.layers.dense(inputs=dense4, units=256,  
activation=tf.nn.relu)  
output_layer = tf.layers.dense(inputs=dense5, units=784,  
activation=tf.nn.sigmoid)
```

5개의 히든 레이어는 각각 256,128,16,128,256 개의 노드를 가지고 있고, 각각 ReLu를 Activation 함수로 사용하였다.

그리고 마지막 output layer는 784개의 노드를 가지고 sigmoid 함수를 activation 함수로 사용하였다.

Loss 함수 정의

다음 모델에 대한 비용함수(loss/cost function)을 정의한다. 이 글을 읽을 수준이면 비용함수에 대해서 별도로 설명하지 않아도 되리라고 보는데, 비용함수는 예측값과 원래 라벨에 대한 차이의 합을 나타내는 것이 비용함수이다.

```
# Connect the output layer to second hidden layer (no activation fn)

output_layer = tf.layers.dense(second_hidden_layer, 1)
# Reshape output layer to 1-dim Tensor to return predictions
predictions = tf.reshape(output_layer, [-1])
predictions_dict = {"ages": predictions}

# Calculate loss using mean squared error
loss = tf.losses.mean_squared_error(labels, predictions)
```

코드를 보면, 최종 예측된 값은 predictions에 저장되고, 학습 데이터로 부터 받은 라벨 값은 labels에 저장된다. 이 차이를 계산할때, MSE (mean square error)를 사용하였다.

Training Op 정의

비용 함수가 적용되었으면, 이 비용함수의 값을 최적화 하는 것이 학습이기 때문에, 옵티마이저를 정의하고, 옵티마이저를 이용하여 비용함수의 최적화가 되도록 한다.

아래 코드는 Optimizer를 GradientDescentOptimizer로 정의하고, 이 옵티마이저를 이용하여 이용하여 loss 값을 최소화 하도록 하였다.

```
optimizer = tf.train.GradientDescentOptimizer(
    learning_rate=params["learning_rate"])

train_op = optimizer.minimize(
    loss=loss, global_step=tf.train.get_global_step())
```

전체 코드

그러면 위의 내용을 모두 합쳐서 model_fn으로 모아서 해보자.

```
def model_fn(features, labels, mode, params):
    """Model function for Estimator."""
    # Connect the first hidden layer to input layer
    # (features["x"]) with relu activation
    first_hidden_layer = tf.layers.dense(features["x"], 10,
        activation=tf.nn.relu)
```



```

# Connect the second hidden layer to first hidden layer with relu
second_hidden_layer = tf.layers.dense(
    first_hidden_layer, 10, activation=tf.nn.relu)

# Connect the output layer to second hidden layer (no activation fn)
output_layer = tf.layers.dense(second_hidden_layer, 1)

# Reshape output layer to 1-dim Tensor to return predictions
predictions = tf.reshape(output_layer, [-1])

# Provide an estimator spec for `ModeKeys.PREDICT`.
if mode == tf.estimator.ModeKeys.PREDICT:
    return tf.estimator.EstimatorSpec(
        mode=mode,
        predictions={"ages": predictions})

# Calculate loss using mean squared error
loss = tf.losses.mean_squared_error(labels, predictions)

# Calculate root mean squared error as additional eval metric
eval_metric_ops = {
    "rmse": tf.metrics.root_mean_squared_error(
        tf.cast(labels, tf.float64), predictions)
}

optimizer = tf.train.GradientDescentOptimizer(
    learning_rate=params["learning_rate"])

train_op = optimizer.minimize(
    loss=loss, global_step=tf.train.get_global_step())

# Provide an estimator spec for `ModeKeys.EVAL` and `ModeKeys.TRAIN`
modes.

return tf.estimator.EstimatorSpec(
    mode=mode,
    loss=loss,
    train_op=train_op,
    eval_metric_ops=eval_metric_ops)

```

데이터 입력

```

first_hidden_layer = tf.layers.dense(features["x"], 10,
activation=tf.nn.relu)

```

네트워크 정의

```
# Connect the second hidden layer to first hidden layer with relu
second_hidden_layer = tf.layers.dense(
    first_hidden_layer, 10, activation=tf.nn.relu)

# Connect the output layer to second hidden layer (no activation fn)
output_layer = tf.layers.dense(second_hidden_layer, 1)
```

first_hidden_layer의 입력값을 가지고 네트워크를 구성한다. 두번째 레이어는 first_hidden_layer를 입력값으로 하여, 10개의 노드를 가지고, ReLu를 activation 레이어로 가지도록 하였다.

마지막 계층은 두번째 계층에서 나온 결과를 하나의 노드를 이용하여 합쳐서 activation 함수 없이 결과를 냈다.

```
# Reshape output layer to 1-dim Tensor to return predictions
predictions = tf.reshape(output_layer, [-1])

# Provide an estimator spec for `ModeKeys.PREDICT`.
if mode == tf.estimator.ModeKeys.PREDICT:
    return tf.estimator.EstimatorSpec(
        mode=mode,
        predictions={"ages": predictions})
```

예측 모드에서는 prediction 값을 리턴해야 하기 때문에, 먼저 예측값을 output_layer에서 나온 값으로, 행렬 차원을 변경하여 저장하고, 만약에 예측 모드 tf.estimator.ModeKeys.PREDICT일 경우 EstimatorSpec에 predction 값을 넣어서 리턴한다. 이때 dict 형태로 prediction 결과 이름을 age로 값을 predictions 값으로 채워서 리턴한다.

Loss 함수 정의

다음 비용 함수를 정의하고, 테스트 단계(EVAL)에서 사용할 evaluation metrics에 rmse를 테스트 기준으로 메트릭으로 정의한다.

```
# Calculate loss using mean squared error
loss = tf.losses.mean_squared_error(labels, predictions)

# Calculate root mean squared error as additional eval metric
eval_metric_ops = {
    "rmse": tf.metrics.root_mean_squared_error(
        tf.cast(labels, tf.float64), predictions)
}
```

Training OP 정의

비용 함수를 정했으면, 비용 함수를 최적화 하기 위한 옵티마이저를 정의한다. 아래와 같이 GradientDescentOptimizer를 이용하여 loss 함수를 최적화 하도록 하였다.

```
optimizer = tf.train.GradientDescentOptimizer(
    learning_rate=params["learning_rate"])

train_op = optimizer.minimize(
    loss=loss, global_step=tf.train.get_global_step())

# Provide an estimator spec for `ModeKeys.EVAL` and `ModeKeys.TRAIN`
modes.
```

마지막으로, PREDICTION이 아니고, TRAIN,EVAL인 경우에는 EstimatorSpec을 다음과 같이 리턴한다.

Loss 함수와, Training Op를 정의하고 평가용 매트릭스를 정의하여 리턴한다.

```
return tf.estimator.EstimatorSpec(
    mode=mode,
    loss=loss,
    train_op=train_op,
    eval_metric_ops=eval_metric_ops)
```

실행

그러면 완성된 Estimator를 사용해보자

```
train_input_fn = tf.estimator.inputs.numpy_input_fn(
    x={"x": np.array(training_set.data)},
    y=np.array(training_set.target),
    num_epochs=None,
    shuffle=True)
```

```
# Train
```

```
nn.train(input_fn=train_input_fn, steps=5000)
```

```
# Score accuracy
```

```
test_input_fn = tf.estimator.inputs.numpy_input_fn(
    x={"x": np.array(test_set.data)},
    y=np.array(test_set.target),
    num_epochs=1,
    shuffle=False)
```

```
ev = nn.evaluate(input_fn=test_input_fn)
print("Loss: %s" % ev["loss"])
print("Root Mean Squared Error: %s" % ev["rmse"])
```

각 코드를 보면

```
train_input_fn = tf.estimator.inputs.numpy_input_fn(
    x={"x": np.array(training_set.data)},
```

```
y=np.array(training_set.target),  
num_epochs=None,  
shuffle=True)
```

를 이용하여 numpy 의 데이터로 input_fn 함수를 만들었다. training_set.data는 학습 데이터, training_set.target을 학습용 라벨로 설정하고, epoch는 무제한, 그리고 데이터는 셔플 하도록 하였다.

```
nn.train(input_fn=train_input_fn, steps=5000)
```

앞서 정의된 모델에 train_input_fn을 넣어서 총 5000 번 학습을 하도록 하였다.

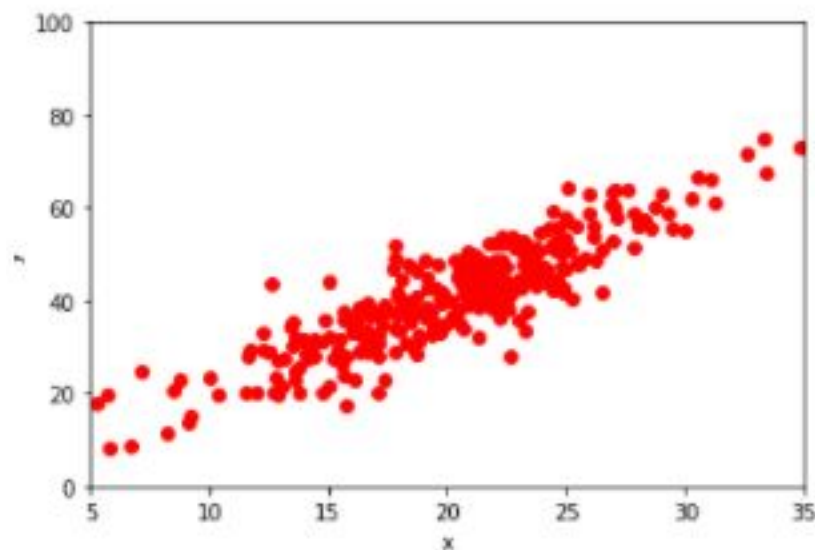
학습이 끝난 모델을 테스트 해야 하는데, 같은 방법으로 test_input_fn을 정의하고

```
ev = nn.evaluate(input_fn=test_input_fn)
```

evaluate를 이용하여, 학습된 모델을 평가한다.

평가된 결과를 보기 위해서 loss 값과 rmse 값을 ev['loss'], ev['rmse']로 출력하였다.

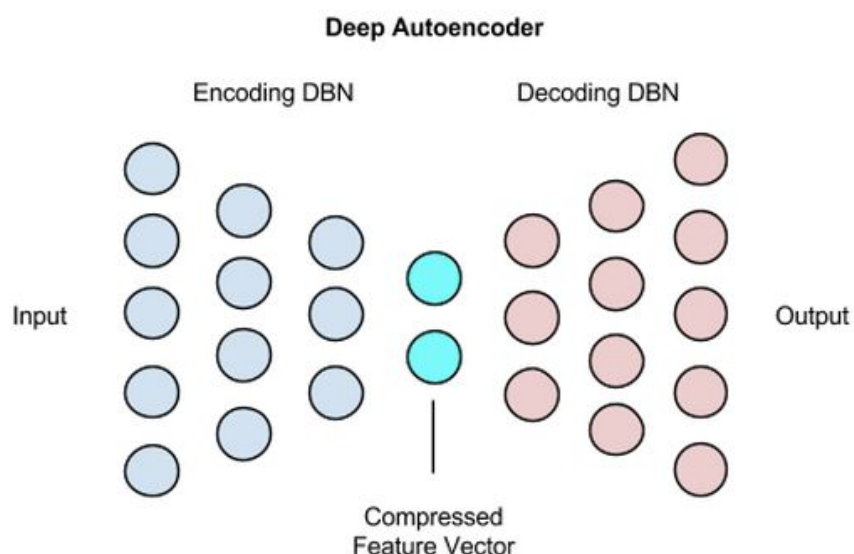
지금까지 Estimator를 만드는 방법에 대해서 알아보았다. 다음 글에서는 Auto Encoder 네트워크를 Estimator로 구현해보도록 하겠다.



그래프와 비교해보면 유사 값이 나오는 것을 확인할 수 있다.

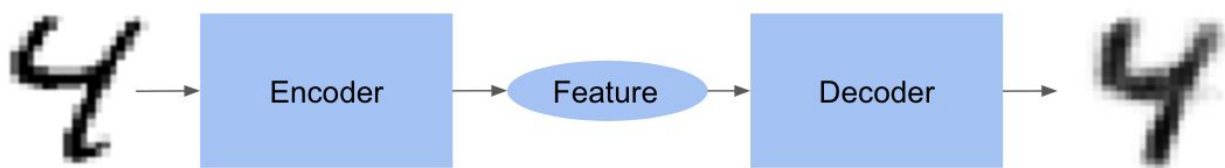
12. Custom Estimator 를 이용한 오토인코더구현

오토 인코더는 딥네트워크 기반의 비지도 학습 모델로, 뉴럴네트워크 두개를 뒤집어서 붙여놓은 형태이다.



<그림 출처 : <https://deeplearning4j.org/deepautoencoder> >

앞에 있는 뉴럴네트워크는 인코더, 뒤에 붙은 네트워크는 디코더가 된다.
인코더를 통해서 입력 데이터에 대한 특징을 추출해내고, 이 결과를 가지고 뉴럴 네트워크를 역으로 붙여서 원본 데이터를 생성해낸다.



이 과정에서 입력과 출력값이 최대한 같아지도록 튜닝함으로써, Feature를 잘 추출할 수 있게 하는것이 오토 인코더의 원리이다.

비정상 거래 검출에 있어서 이를 활용하는 방법은 학습이 되지 않은 데이터의 경우 디코더에 의해 복원이 제대로 되지 않고 원본 데이터와 비교했을때 차이값이 크기 때문에, 정상 거래로 학습된 모델은 비정상 거래가 들어왔을때 결과값이 입력값보다 많이 다를 것이라는 것을 가정한다.

그러면 입력값 대비 출력값이 얼마나 다르면 비정상 거래로 판단할것인가에 대한 임계치 설정이 필요한데, 이는 실제 데이터를 통한 설정이나 또는 통계상의 데이터에 의존할 수 밖에 없다. 예를 들어 전체 신용카드 거래의 0.1%가 비정상 거래라는 것을 가정하면, 입력 값들 중에서 출력값과 차이가 큰 순서대로 데이터를 봤을때 상위 0.1%만을 비정상 거래로 판단한다.

또는 비지도 학습이기 때문에, 나온 데이터로 정상/비정상을 판단하기 보다는 비정상 거래일 가능성을 염두해놓고, 그 거래들을 비정상 거래일 것이라고 예측하고 이 비정상 거래 후보에 대해서 실제 확인이나 다른 지표에 대한 심층 분석을 통해서 비정상 거래를 판별한다.

이러한 과정을 거쳐서 비정상 거래가 판별이 되면, 비정상 거래에 대한 데이터를 라벨링하고 이를 통해서 다음 모델 학습시 임계치 값을 설정하거나 다른 지도 학습 알고리즘으로 변경하는 방법등을 고민해볼 수 있다.

이 코드의 원본은 Etsuji Nakai 님의 https://github.com/enakai00/autoencoder_example 코드를 사용하였다.

데이터 전처리

이 예제에서는 텐서플로우에 포함된 MNIST 데이터 `tensorflow.contrib.learn.python.learn.datasets`를 `tfrecord`로 변경해서 사용한다. TFRecord에 대한 설명은 <http://bcho.tistory.com/1190>를 참고하기 바란다.

MNIST 데이터를 TFRecord로 변경하는 코드는

https://github.com/bwcho75/tensorflowML/blob/master/autoencoder/MNIST/create_record.py에 있다. 이 코드를 실행하면, `/tmp/data/train.tfrecord` `/tmp/data/test.tfrecords`에 학습 및 테스트 데이터 파일이 생성된다. 이 파일들을 아래서 만들 모델이 들어가 있는 디렉토리 아래 `/data` 디렉토리로 옮겨놓자.

학습 코드 구현

학습에 사용되는 모델은 텐서플로우 하이레벨 API인 `tf.layers`와 `Estimator`를 이용해서 구현한다. 하이레벨 API를 사용하는 이유는 <http://bcho.tistory.com/1195> <http://bcho.tistory.com/1196>에서도 설명했듯이 구현이 상대적으로 쉬울뿐더러, 분산 학습이 가능하기 때문이다.

전체 코드는

https://github.com/bwcho75/tensorflowML/blob/master/autoencoder/MNIST/MNIST_AutoEncoder.ipynb에 공유되어 있다.

데이터 입력부

데이터 입력 부분은 tfrecord 파일을 읽어서, 파일 큐를 생성해서 input_fn 을 생성하는 부분이다. 이렇게 생성된 input_fn 함수는 Estimator 를 통해서, 학습과 테스트(검증) 데이터로 피딩되게 된다.

데이터 입력 부분은 read_and_decode 함수와 input_fn 함수로 구현되어 있는데, 각각을 살펴보자

```
def read_and_decode(filename_queue):
    reader = tf.TFRecordReader()
    _, serialized_example = reader.read(filename_queue)

    features = tf.parse_single_example(
        serialized_example,
        features={
            'image_raw': tf.FixedLenFeature([], tf.string),
            'label': tf.FixedLenFeature([], tf.int64),
        })

    image = tf.decode_raw(features['image_raw'], tf.uint8)
    image.set_shape([784]) #image shape is (784,)
    image = tf.cast(image, tf.float32)*(1.0/255)
    label = tf.cast(features['label'], tf.int32)

    return image, label
```

read_and_decode 함수는 filename_queue에서, 파일을 읽어서 순서대로 TFRecordReader를 읽어서 파싱한 후에, image_raw 이름으로 된 피쳐와, label로 된 피쳐를 읽어서 각각 image와 label 이라는 텐서에 저장한다.

image는 차원을 맞추기 위해서 set_shape를 이용하여 1차원으로 784의 길이를 가진 텐서로 변환하고, 학습에 적절하도록 데이터를 regularization 을 하기 위해서, 1.0/255 를 곱해줘서 1~255값의 칼라값을 0~1사이의 값으로 변환한다.

그리고 label값은 0~9를 나타내는 숫자 라벨이기 때문에, tf.int32로 형 변환을 한다. 변환이 끝난 image와 label 텐서를 리턴한다.

```
def input_fn(filename,batch_size=100):
    filename_queue = tf.train.string_input_producer([filename])

    image,label = read_and_decode(filename_queue)
    images,labels = tf.train.batch(
        [image,label],batch_size=batch_size,
        capacity=1000+3*batch_size)
    #images : (100,784), labels : (100,1)
```

```
    return {'inputs':images},labels
```

Input_fn 함수는 실제로 Estimator에 값을 피딩하는 함수로, 입력 받은 filename으로 파일이름 큐를 만들어서 read_and_decode 함수에 전달 한 후, image와 label 값을 리턴받는다. 리턴 받은 값을 바로 리턴하지 않고 배치 학습을 위해서 tf.train.batch를 이용하여 배치 사이즈(batch_size)만큼 묶어서 리턴한다.

모델 구현부

데이터 입력 부분이 완성되었으면, 데이터를 읽어서 학습 하는 부분을 살펴보자.

아래는 모델을 구현한 autoencoder_model_fn 함수이다.
Custom Estimator를 구현하기 위해서 사용한 구조이다.

```
def autoencoder_model_fn(features,labels,mode):
    input_layer = features['inputs']
    dense1 = tf.layers.dense(inputs=input_layer,units=256,activation=tf.nn.relu)
    dense2 = tf.layers.dense(inputs=dense1,units=128,activation=tf.nn.relu)
    dense3 = tf.layers.dense(inputs=dense2,units=16,activation=tf.nn.relu)
    dense4 = tf.layers.dense(inputs=dense3,units=128,activation=tf.nn.relu)
    dense5 = tf.layers.dense(inputs=dense4,units=256,activation=tf.nn.relu)
    output_layer = tf.layers.dense(inputs=dense5,units=784,activation=tf.nn.sigmoid)

    #training and evaluation mode
    if mode in (Modes.TRAIN,Modes.EVAL):
        global_step = tf.contrib.framework.get_or_create_global_step()
        label_indices = tf.cast(labels,tf.int32)
        loss = tf.reduce_sum(tf.square(output_layer - input_layer))
        tf.summary.scalar('OptimizeLoss',loss)

    if mode == Modes.TRAIN:
        optimizer = tf.train.AdamOptimizer(learning_rate=0.001)
        train_op = optimizer.minimize(loss,global_step=global_step)
        return tf.estimator.EstimatorSpec(mode,loss = loss, train_op = train_op)
    if mode == Modes.EVAL:
        eval_metric_ops = None
        return tf.estimator.EstimatorSpec(
```



```

mode,loss=loss,eval_metric_ops = eval_metric_ops)

# prediction mode
if mode == Modes.PREDICT:
    predictions={
        'outputs':output_layer
    }
    export_outputs={
        'outputs':tf.estimator.export.PredictOutput(predictions)
    }
    return tf.estimator.EstimatorSpec(
        mode,predictions=predictions,export_outputs=export_outputs) #이부분 코드 상세 조사할것

```

오토인코더 네트워크를 구현하기 위한 코드는 다음 부분으로 복잡하지 않다

```

input_layer = features['inputs']
dense1 = tf.layers.dense(inputs=input_layer,units=256,activation=tf.nn.relu)
dense2 = tf.layers.dense(inputs=dense1,units=128,activation=tf.nn.relu)
dense3 = tf.layers.dense(inputs=dense2,units=16,activation=tf.nn.relu)
dense4 = tf.layers.dense(inputs=dense3,units=128,activation=tf.nn.relu)
dense5 = tf.layers.dense(inputs=dense4,units=256,activation=tf.nn.relu)
output_layer = tf.layers.dense(inputs=dense5,units=784,activation=tf.nn.sigmoid)

```

input_fn에서 피딩 받은 데이터를 input_layer로 받아서, 각 256,128,16,128,,256의 노드로 되어 있는 5개의 네트워크를 통과한 후에, 최종적으로 784의 아웃풋과 sigmoid 함수를 활성화(activation function)으로 가지는 output layer를 거쳐서 나온다.

다음 모델의 모드 즉 학습, 평가, 그리고 예측 모드에 따라서 loss 함수나 train_op 등이 다르게 정해진다.

```

#training and evaluation mode
if mode in (Modes.TRAIN,Modes.EVAL):
    global_step = tf.contrib.framework.get_or_create_global_step()
    label_indices = tf.cast(labels,tf.int32)
    loss = tf.reduce_sum(tf.square(output_layer - input_layer))
    tf.summary.scalar('OptimizeLoss',loss)

```

학습과 테스트 모드일 경우, global_step을 정하고, loss 함수를 정의한다.

학습 모드일 경우에는 아래와 같이 옵티마이저를 정하고,이 옵티마이저를 이용하여 loss 값을 최적화 하도록 하는 train_op를 정의해서 EstimatorSpec을 만들어서 리턴하였다.

```

if mode == Modes.TRAIN:
    optimizer = tf.train.AdamOptimizer(learning_rate=0.001)
    train_op = optimizer.minimize(loss,global_step=global_step)
    return tf.estimator.EstimatorSpec(mode,loss = loss, train_op = train_op)

```

테스트 모드 일 경우에는 옵티마이저할 필요가 없기 때문에, 옵티마이저를 정의하지 않고 loss 값을 리턴하고, 평가를 위한 Evalutaion metrics를 정해서 리턴한다. 아래 코드는 별도로 evaluation metrics를 정의하지 않고, 디폴트 메트릭스를 사용하였다.

```
if mode == Modes.EVAL:
    eval_metric_ops = None
    return tf.estimator.EstimatorSpec(
        mode, loss=loss, eval_metric_ops = eval_metric_ops)
```

예측 모드일 경우에는 loss 값이나 optimizer 등의 정의가 필요 없고, output값을 어떤 값을 내보낼지만 정의하면 되고, 예측 모델 (prediction model)을 프로토콜 버퍼 포맷으로 export 할때의 구조를 정의하기 위해서 export_outpus 부분만 아래와 같이 정의해주면 된다.

```
# prediction mode
if mode == Modes.PREDICT:
    predictions={
        'outputs':output_layer
    }
    export_outputs={
        'outputs':tf.estimator.export.PredictOutput(predictions)
    }
    return tf.estimator.EstimatorSpec(
        mode, predictions=predictions, export_outputs=export_outputs)
```

Estimator 생성

모델에 대한 정의가 끝났으면, Estimator를 생성하는데, Estimator 정의는 아래와 같이 앞에서 정의한 모델인 autoencoder_model_fn을 정의해주고

```
def build_estimator(model_dir):
    return tf.estimator.Estimator(
        model_fn = autoencoder_model_fn,
        model_dir = model_dir,
        config=tf.contrib.learn.RunConfig(save_checkpoints_secs=180))
```

실험 (Experiment) 구현

앞에서 구현된 Estimator를 이용하여, 학습과 테스트를 진행할 수 있는데, 직접 Estimator를 불러사용하는 방법 이외에 Experiment 라는 클래스를 사용하면, 이 부분을 단순화 할 수 있다. Experiment에는 사용하고자 하는 Estimator와 학습과 테스트용 데이터 셋, 그리고 export 전략 및, 학습,테스트 스텝을 넣어주면 자동으로 Estimator를 이용하여 학습과 테스트를 진행해준다. 아래는 Experiment 를 구현한 예이다.

```
def generate_experiment_fn(data_dir,
    train_batch_size = 100,
```

```

        eval_batch_size = 100,
        train_steps = 1000,
        eval_steps = 1,
        **experiment_args):
def _experiment_fn(output_dir):
    return Experiment(
        build_estimator(output_dir),
        train_input_fn=get_input_fn('./data/train.tfrecords',batch_size=train_batch_size),
        eval_input_fn=get_input_fn('./data/test.tfrecords',batch_size=eval_batch_size),
        export_strategies = [saved_model_export_utils.make_export_strategy(
            serving_input_fn,
            default_output_alternative_key=None,
            exports_to_keep=1)
        ],
        train_steps = train_steps,
        eval_steps = eval_steps,
        **experiment_args
    )
return _experiment_fn

```

```

learn_runner.run(
    generate_experiment_fn(
        data_dir='./data/',
        train_steps=2000),
    OUTDIR)

```

대략 50,000 스텝까지 학습을 진행하면 loss 값 500 정도로 수렴 되는 것을 확인할 수 있다.

검증 코드 구현

검증 코드는 MNIST 데이터에서 테스트용 데이터를 로딩하여 테스트 이미지를 앞에서 학습된 이미지로 인코딩했다가 디코딩 하는 예제이다. 입력 이미지와 출력 이미지가 비슷할 수 록 제대로 학습된것이라고 볼수 있다.

Export 된 모듈 로딩

아래 코드는 앞의 학습과정에서 Export 된 학습된 모델을 로딩하여 새롭게 그래프를 로딩 하는 코드이다.

```

#reset graph
tf.reset_default_graph()

export_dir = OUTDIR+'./export/Servo/'
timestamp = os.listdir(export_dir)[0]

```

```
export_dir = export_dir + timestamp
print(export_dir)
```

```
sess = tf.Session()
meta_graph =
tf.saved_model.loader.load(sess,[tf.saved_model.tag_constants.SERVING],export_dir)
model_signature = meta_graph.signature_def['serving_default']
input_signature = model_signature.inputs
output_signature = model_signature.outputs

print(input_signature.keys())
print(output_signature.keys())
```

tf.reset_default_graph()를 이용하여, 그래프를 리셋 한후, tf.saved_model.loader.load()를 이용하여 export_dir에서 Export 된 파일을 읽어서 로딩한다.
다음 입력값과 출력값의 텐서 이름을 알기 위해서 model_signature.input과 output 시그니처를 읽어낸후 각각 keys()를 이용하여 입력과 출력 텐서 이름을 출력하였다.
이 텐서 이름은 로딩된 그래프에 입력을 넣고 출력 값을 뽑을 때 사용하게 된다.

테스트 코드 구현

학습된 모델이 로딩 되었으면 로딩된 모델을 이용하여 MNIST 테스트 데이터를 오토 인코더에 넣어서 예측을 진행 해본다.

```
from tensorflow.examples.tutorials.mnist import input_data

mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)
images, labels = mnist.test.images, mnist.test.labels

feed_dict = {sess.graph.get_tensor_by_name(input_signature['inputs'].name):
mnist.test.images[:10]}
output = sess.graph.get_tensor_by_name(output_signature['outputs'].name)
results = sess.run(output, feed_dict=feed_dict)

fig = plt.figure(figsize=(4,15))
for i in range(10):
    subplot = fig.add_subplot(10,2,i*2+1)
    subplot.set_xticks([])
    subplot.set_yticks([])
    subplot.imshow(images[i].reshape((28,28)), vmin=0, vmax=1,
                    cmap=plt.cm.gray_r, interpolation="nearest")

    subplot = fig.add_subplot(10,2,i*2+2)
    subplot.set_xticks([])
    subplot.set_yticks([])
```

```
subplot.imshow(results[i].reshape((28,28)), vmin=0, vmax=1,
               cmap=plt.cm.gray_r, interpolation="nearest")
```

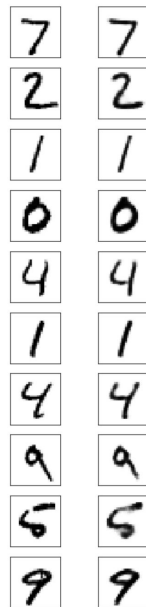
```
plt.show()
```

`feed_dict = {sess.graph.get_tensor_by_name(input_signature['inputs'].name): mnist.test.images[:10]}` 부분은 입력 데이터를 정의하는 부분으로, 앞에 모델 로딩시 사용했던 것과 같이 입력 텐서의 이름을 얻기 위해서 `input_signature`의 이름을 얻은 후, 그래프에서 그 이름으로 텐서를 가지고 온다. 그 이후, 가져온 텐서에 mnist 테스트 데이터셋에서 이미지 부분을 0~9 개를 피딩한다.

출력 값도 마찬가지로 `output_signature`에서 output 텐서 이름을 가지고 온후에, `get_tensor_by_name` 으로 해당 텐서를 가지고 온후에, output 변수에 저장한다.

마지막으로 `sess.run`을 통해서 `feed_dict` 값을 피딩하고, output 텐서를 리턴하여, 결과를 `results`로 리턴한다.

나머지는 리턴된 10개의 prediction result를 matplotlib를 이용하여 시각화 한 결과이다. 아래 결과와 같이 입력값과 출력값이 거의 유사하게 복원되었음을 확인할 수 있다.



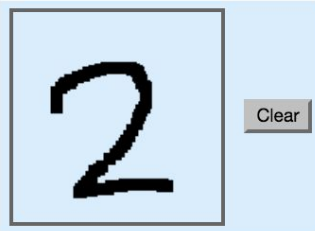
테스트 코드를 웹으로 구현

테스트를 위해서 MNIST 데이터를 입력하는 것 말고, HTML 화면을 이용하여 직접 마우스로 숫자를 그래서 입력할 수 있도록 해보자

코드 구조 자체는 위의 예제와 같기 때문에 별도로 설명하지 않는다.

```
In [93]: from IPython.display import HTML
HTML(input_form + javascript)
```

Out[93]:



```
In [110]: feed_dict = {sess.graph.get_tensor_by_name(input_signature['inputs'].name):[image]}
output = sess.graph.get_tensor_by_name(output_signature['outputs'].name)
results = sess.run(output, feed_dict=feed_dict)

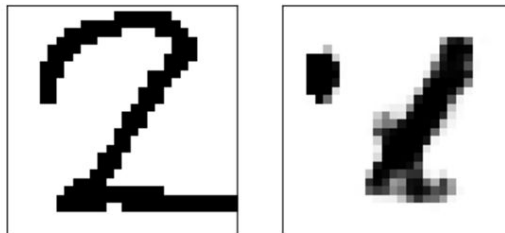
fig = plt.figure(figsize=(6,3))

subplot = fig.add_subplot(1,2,1)
subplot.set_xticks([])
subplot.set_yticks([])
subplot.imshow(np.array(image).reshape((28,28)), vmin=0, vmax=1,
               cmap=plt.cm.gray_r, interpolation="nearest")

subplot = fig.add_subplot(1,2,2)
subplot.set_xticks([])
subplot.set_yticks([])
subplot.imshow(results[0].reshape((28,28)), vmin=0, vmax=1,
               cmap=plt.cm.gray_r, interpolation="nearest")

plt.show()

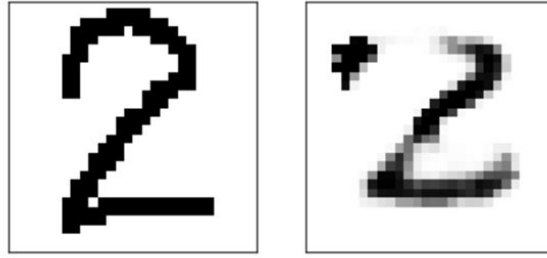
loss = tf.reduce_sum(tf.square(results[0] - image))
loss_val = sess.run(loss)
print("Difference : "+str(loss_val))
```



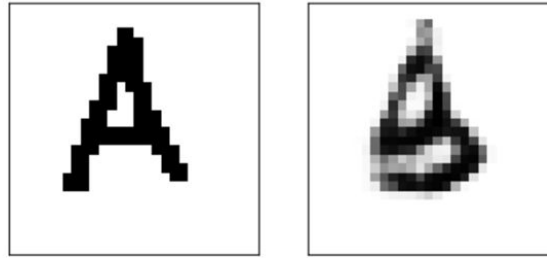
Difference :133.149799809

위의 그림과 같이 HTML 입력 박스에 마우스로 그림을 그리면 아래 그림과 같이 입력값과 함께 복원된 이미지를 보여 준다.

웹을 이용하여 숫자와 알파벳을 입력해서 입력과 결과값을 구분해본 결과, 영문이던 숫자이던 입출력 차이가 영문이나 숫자가 크게 차이가 나지 않아서, 변별력이 크지 않았다.

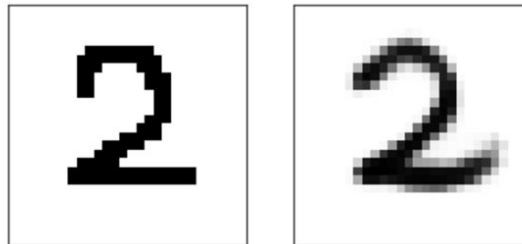


Differnce :118.759889021

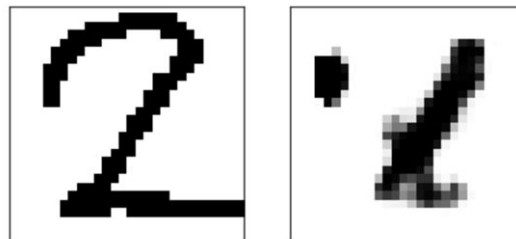


Differnce :44.7308005673

트레이닝 스텝이 이 50,000 스텝 정도면 loss값이 500 근처로 수렴을 하였는데, 1,000,000 스텝을 학습 시켜서 MNIST 데이터에 대한 기억 효과를 극대화 하려고 했지만 큰 효과가 없었다. 여러가지 원인이 있겠지만, HTML에서 손으로 이미지를 인식 받는 만큼, 글자의 위치나 크기에 따라서 loss 값이 크게 차이가 나는 결과를 보였다. 이 부분은 컨볼루션 필터 (Convolution Filter)를 사용하면 해결이 가능할것 같으나 적용은 하지 않았다.



Differnce :20.7133778336



Differnce :133.149799809

또한 학습에 사용된 데이터는 0~255의 흑백 값이지만, 위의 예제에서 웹을 통해 입력받은 값은 흑/백 (0 or 255)인 값이기 때문에 눈으로 보기에는 비슷하지만 실제로는 많이 다른 값이다.

또는 학습 데이터가 모자르거나 또는 네트워크 사이즈가 작았을 것으로 생각하는데, 그 부분은 별도로 테스트 하지 않았다.

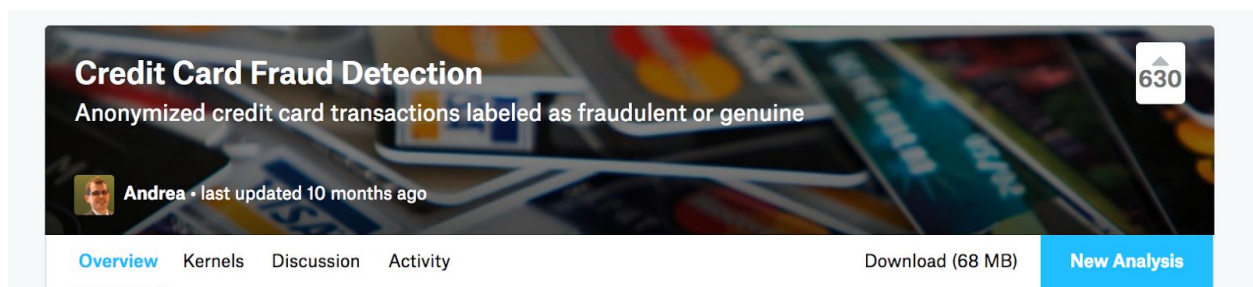
신용 카드 데이터의 경우 손으로 그리는 그림이 아니기 때문에, 이런 문제는 없을 것으로 생각하는데, 만약 문제가 된다면 네트워크 사이즈를 조정해보는 방안으로 진행할 예정이다.

다음 글에서는 신용 카드 데이터를 가지고 오토 인코더를 이용하여 비정상 거래를 검출하기 위해서 학습을 위하여 데이터 전처리를 하는 부분에 대해서 알아보도록 하겠다.

13. 오토인코더를 이용한 신용카드 비정상 거래 탐지

오토 인코더를 사용해서 신용카드 비정상 거래를 찾아내도록 구현하였으나, 예제에 사용된 데이터는 비정상 거래가 라벨링 되어 있는 데이터로, 오토 인코더와 같은 비지도 학습에는 적절하지 않다. 라벨링이 되어 있기 때문에, 지도 학습을 사용하는 것이 좋으나, 비정상 거래 패턴을 찾는 일은 라벨링이 되어 있는 경우가 적기 때문에, 아이디어 차원과 오토 인코더 모델을 소개하기 위한 목적임을 이해하기 바란다.

신용카드 데이터를 가지고, 비정상 거래를 찾는 모델을 오토 인코더를 이용하여 구현해보자 <https://www.kaggle.com/dalpozz/creditcardfraud> 에서 데이터를 다운 받을 수 있다.



CSV 형태로 되어 있으며, 2013년 유럽 카드사의 실 데이터 이다. 2일간의 데이터 이고, 총 284,807건의 트랜잭션 로그중에, 492건이 비정상 데이터이고, 데이터 분포는 비정상 데이터가 0.172%로 심하게 불균형적이다.

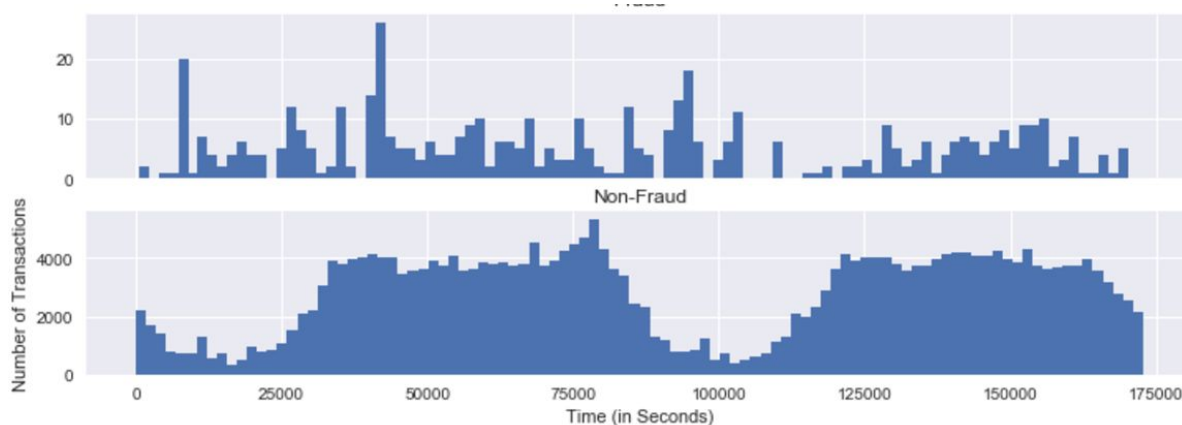
전체 31개의 컬럼중, 첫번째 컬럼은 시간, 30번째 컬럼은 비정상 거래 유무 (1이면 비정상, 0이면 정상) 그리고 마지막 31번째 컬럼은 결재 금액을 나타낸다 2~29번째 컬럼이 특징 데이터 인데, V1~V28로 표현되고 데이터 컬럼명은 보안을 이유로 모두 삭제 되었다.

데이터 분석

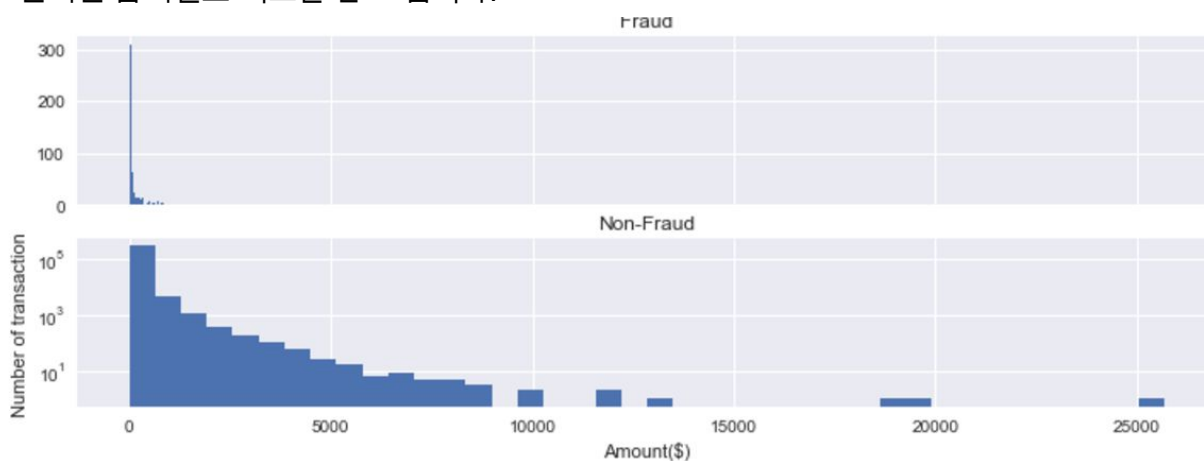
어떤 컬럼들을 피쳐로 정할것인가를 결정하기 위해서 데이터 분석을 시작한다.

데이터 분석 방법은 <https://www.kaggle.com/currie32/predicting-fraud-with-tensorflow> 를 참고하였다.

시간대별 트랜잭션양을 분석해보면 별다른 상관 관계를 찾을 수 없다.

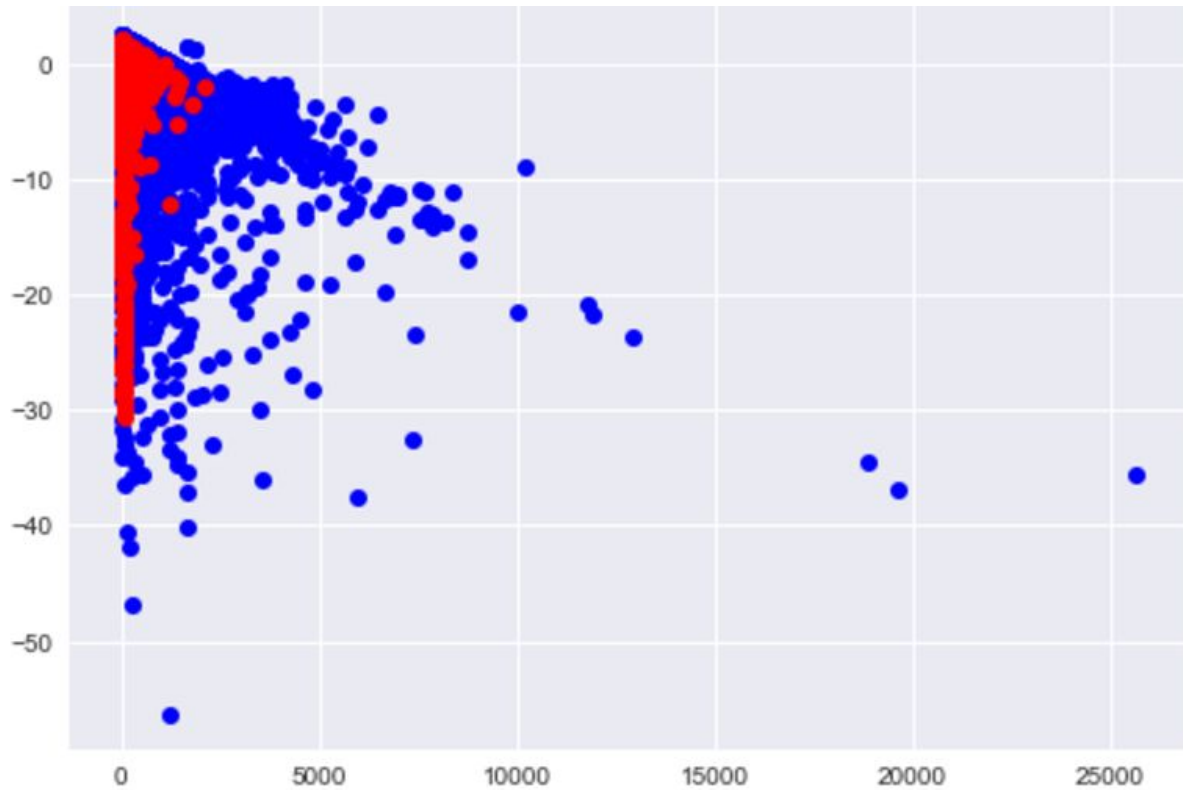


트랜잭션 금액별로 비교를 한 그림이다.



위의 비정상 데이터를 보면, 작은 금액에서 비정상 거래가 많이 일어난것을 볼 수 있지만, 정상 거래군과 비교를 해서 다른 특징을 찾아낼 수 없다.

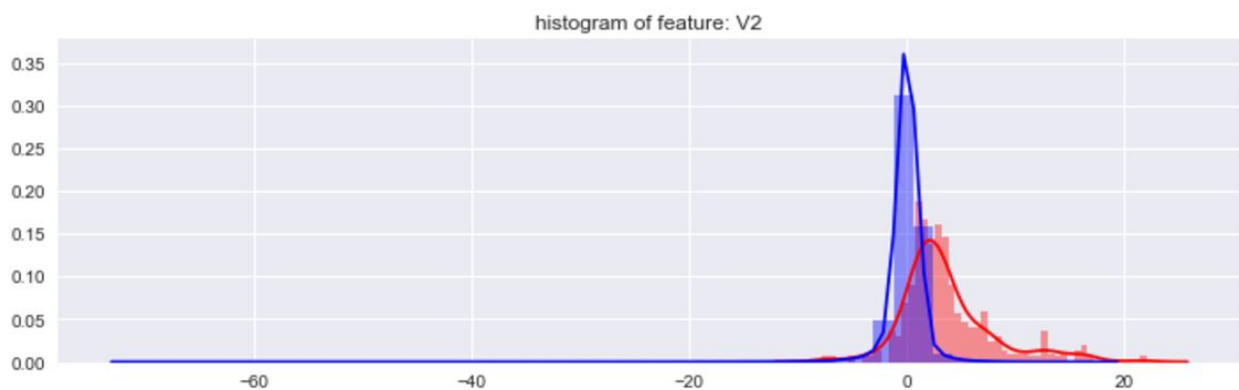
다음은 트랜잭션 금액을 기준으로 V1~V28 피처를 비교 분석해봤다.



붉은 점은 비정상, 파란점이 정상 거래이고, 가로축이 금액, 세로축이 V1 값이다. 이런 방법으로 V1~V8에 대한 그래프를 그려봤으나, 비정상 거래가 항상 정상거래의 부분집합형으로 별다른 특이점을 찾아낼 수 없었다.

다음으로 V1~V28 각 컬럼간의 값 분포를 히스토그램으로 표현한 결과이다.

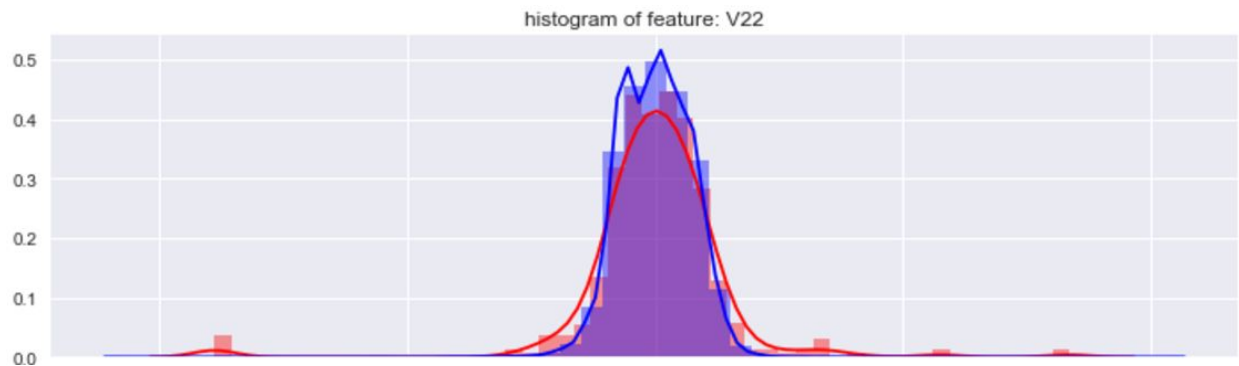
아래는 V2 피쳐의 값을 히스토그램으로 표현한 결과로 파란색이 정상, 붉은 색이 비정상 거래인데, 히스토그램이 차이가 나는 것을 확인할 수 있다.



V4 피쳐 역시 아래 그림과 같이 차이가 있는 것을 볼 수 있다.



V22 피쳐의 경우에는 정상과 비정상 거래의 패턴이 거의 유사하여 변별력이 없는 것을 볼 수 있다.



이런식으로, V1~V28중에 비정상과 정상거래에 차이를 보이는 피쳐들만 선정한다.

위의 그래프들은 생성하는 코드는

[https://github.com/bwcho75/tensorflowML/blob/master/autoencoder/Credit%20card%20fraud%20detection%20\(Data%20Analytics\).ipynb](https://github.com/bwcho75/tensorflowML/blob/master/autoencoder/Credit%20card%20fraud%20detection%20(Data%20Analytics).ipynb) 에 있다.

데이터 전처리

실제 모델을 만들기 전에 앞서 신용카드 거래 데이터를 학습에 적절하도록 전처리를 하도록 한다.

데이터양이 그리 크지 않기 때문에, 데이터 전처리는 파이썬 데이터 라이브러리인 pandas dataframe을 사용하였다. 여기서 사용된 전처리 코드는

https://github.com/bwcho75/tensorflowML/blob/master/autoencoder/creditcard_fraud_detection/2.data_normalization.ipynb 에 공개되어 있다.

신용카드 거래 데이터를 머신러닝 학습의 검증과 테스트에 적절하도록 다음과 같은 절차를 통하여 데이터를 전처리하여 CSV 파일로 저장하였다.

데이터 정규화

학습 데이터에 여러가지 피처를 사용하는데, 예를 들어 피처 V1의 범위가 -10000~10000이고, 피처 V2의 범위가 10~20 이라면, 각 피처의 범위가 차이가 매우 크기 때문에, 경사 하강법등을 이용할때, 학습 시간이 더디거나 또는 제대로 학습이 되지 않을 수 있다. 자세한 내용은 김성훈 교수님의 모두를 위한 딥러닝 강좌중 정규화 부분

https://www.youtube.com/watch?v=1jPjVoDV_uo&feature=youtu.be 을 참고하기 바란다.

그래서 피처의 범위를 보정(정규화)하여 학습을 돕는 과정을 데이터 정규화라고 하는데, 정규화에는 여러가지 방법이 있다. 여기서 사용한 방법은 Feature scaling이라는 방법으로, 모든 피처의 값들을 0~1사이로 변환하는 방법이다. 위에서 언급한 V1은 -10000~10000의 범위가 0~1사이로 사상되는 것이고, V2도 10~20의 범위가 0~1사이로 사상된다.

공식은 아래와 같은데

$$X' = \frac{X - X_{\min}}{X_{\max} - X_{\min}}$$

참고 [https://en.wikipedia.org/wiki/Normalization_\(statistics\)](https://en.wikipedia.org/wiki/Normalization_(statistics))

정규화된 값은 = (원본값 - 피처의 최소값) / (피처의 최대값 - 피처의 최소값)

으로 계산한다.

앞의 V1값에서 0의 경우는 $(0 - (-10000)) / (10000 - (-10000)) = 0.5$ 로 사상이 되는것이다.

그러면 신용카드 데이터에서 V1~V28 컬럼을 Feature scaling을 위해서 정규화를 하려면

```
df_csv = pd.read_csv('./data/creditcard.csv')
```

CSV에서 원본 데이터를 읽는다.

읽어드린 데이터의 일부를 보면 다음과 같다.

	Time	V1	V2	V3
0	0.0	-1.359807	-0.072781	2.536347
1	0.0	1.191857	0.266151	0.166480
2	1.0	-1.358354	-1.340163	1.773209
3	1.0	-0.966272	-0.185226	1.792993
4	2.0	-1.158233	0.877737	1.548718

df_csv 는 데이터의 원본값을 나타내고, df_csv.min() 각 컬럼의 최소값, df_csv.max()는 각 컬럼의 최대값을 나타낸다. 이 값들을 이용하여 위의 Feature Scaling 공식으로 구현하면 아래와 같이 된다

```
df_norm = (df_csv - df_csv.min() ) / (df_csv.max() - df_csv.min() )
```

이렇게 정규화된 값을 출력해보면 다음과 같다.

	Time	V1	V2
0	0.000000	0.935192	0.766490
1	0.000000	0.978542	0.770067
2	0.000006	0.935217	0.753118
3	0.000006	0.941878	0.765304
4	0.000012	0.938617	0.776520

V1 컬럼의 -1.359807이 정규화후에 0.935192 로 변경된것을 확인할 수 있고 다른 필드들도 변경된것을 확인할 수 있다.

데이터 분할

전체 데이터를 정규화 하였으면 데이터를 학습용, 검증용, 테스트용 데이터로 나눠야 하는데, 오토 인코더의 원리는 정상적인 데이터를 학습 시킨후에, 데이터를 넣어서 오토인코더가 학습되어 있는 정상적인 패턴과 얼마나 다른가를 비교하는 것이기 때문에 학습 데이터에는 이상거래를 제외하고 정상적인 거래만으로 학습을 한다.

이를 위해서 먼저 데이터를 정상과 비정상 데이터셋 두가지로 분리한다.

아래 코드는 Class=1이면 비정상, Class=0이면 정상인 데이터로 분리가 되는데, 정상 데이터는 df_norm_nonfraud에 저장하고, 비정상 데이터는 df_norm_fraud에 저장하는 코드이다.

```
# split normalized data by label
df_norm_fraud=df_norm[ df_norm.Class==1.0] #fraud
df_norm_nonfraud=df_norm[ df_norm.Class==0.0] #non_fraud
```

정상 데이터를 60:20:20 비율로 학습용, 테스트용, 검증용으로 나누고, 비정상 데이터는 학습에는 사용되지 않고 테스트용 및 검증용에만 사용되기 때문에, 테스트용 및 검증용으로 50:50 비율로 나눈다.

```
# split non_fraudfor 60%,20%,20% (training,validation,test)
df_norm_nonfraud_train,df_norm_nonfraud_validate,df_norm_nonfraud_test = \
    np.split(df_norm_nonfraud,[int(.6*len(df_norm_nonfraud)),int(.8*len(df_norm_nonfraud))])
```

numpy의 split 함수를 쓰면 쉽게 데이터를 분할 할 수 있다.

[int(.6*len(df_norm_nonfraud)),int(.8*len(df_norm_nonfraud))] 가 데이터를 분할하는 구간을 정의하는데, 데이터 프레임의 60%, 80% 구간을 데이터 분할 구간으로 하면 0~60%, 60~80%,

80~100% 구간 3가지로 나누어서 데이터를 분할하여 리턴한다. 같은 방식으로 아래와 같이 비정상 거래 데이터도 50% 구간을 기준으로 하여 두 덩어리로 데이터를 나눠서 리턴한다.

```
# split fraud data to 50%,50% (validation and test)
df_norm_fraud_validate,df_norm_fraud_test = \
    np.split(df_norm_fraud,[int(0.5*len(df_norm_fraud))])
```

데이터 합치기

다음 이렇게 나뉜 데이터를 테스트용 데이터는 정상과 비정상 거래 데이터를 합치고, 검증용 데이터 역시 정상과 비정상 거래를 합쳐서 각각 테스트용, 검증용 데이터셋을 만들어 낸다. 두개의 데이터 프레임을 합치는 것은 아래와 같이 .append() 메서드를 이용하면 된다.

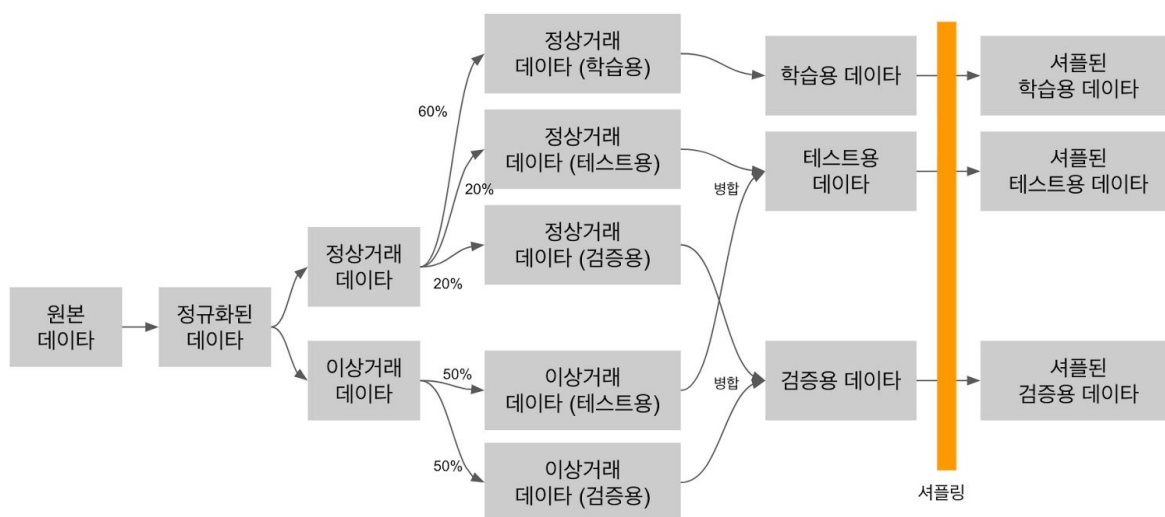
```
df_train = df_norm_nonfraud_train.sample(frac=1)
df_validate = df_norm_nonfraud_validate.append(df_norm_fraud_validate).sample(frac=1)
df_test = df_norm_nonfraud_test.append(df_norm_fraud_test).sample(frac=1)
```

셔플링

데이터를 합치게 되면, 테스트용과 검증용 데이터 파일에서 처음에는 정상데이터가 나오다가 뒷부분에 비정상 데이터가 나오는 형태가 되기 때문에 테스트 결과가 올바르게 나올 수 있는 가능성이 있다. 그래서, 순서를 무작위로 섞는 셔플링(Shuffling) 작업을 수행한다.

셔플링은 위의 코드에서 .sample(frac=1)에 의해서 수행되는데, .sample은 해당 데이터 프레임에서 샘플 데이터를 추출하는 명령으로 frac은 샘플링 비율을 정의한다 1이면 100%로, 전체 데이터를 가져오겠다는 이야기 인데, sample()함수는 데이터를 가지고 오면서 순서를 바꾸기 때문에, 셔플링된 결과를 리턴하게 된다.

전체 파이프라인을 정리해서 도식화 해보면 다음과 같다.



학습 및 결과

전체 모델 코드는

https://github.com/bwcho75/tensorflowML/blob/master/autoencoder/creditcard_fraud_detection/3.model.ipynb에 있다.

코드는 <http://bcho.tistory.com/1198>에 설명한 MNIST 데이터를 이용한 오토인코더 모델과 다르지 않다. 차이는 데이터 피딩을 784개의 피처에서 28개의 피처로만 변환하였고, 데이터를 MNIST 데이터셋에서 CSV에서 읽는 부분만 변경이 되었기 때문에 쉽게 이해할 수 있으리라 본다.

모델을 만들고 학습을 한후에, 이상 거래를 검출해봤다. 학습은 creditcard_validation.csv에 총 57108개의 거래로그가 저장되어 있었고, 그중에, 246개가 비정상 거래였다.

네트워크는 28,20,10,7,10,20,28 형태의 네트워크를 사용하였다.

입출력 값의 차이가 큰것을 기준으로 이 값이 어느 임계치 수준 이상이면 비정상 거래로 검출하도록 하고 실험을 해본 결과 다음과 같은 결과를 얻었다.

임계치	검출된 비정상 거래수	정상거래인데 비정상 거래로 검출된 거래
1.1	112	1
1.0	114	5
0.9	117	7
0.8	124	22

대략 검출 비율은 112~120 개 내외로 / 246개 중에서 50%가 안된다. 검출된 거래가 이상 거래인지 아닌지 여부는 대략 90% 이상이 된다.

결론

네트워크를 튜닝하고나 학습 시키는 피처를 변형 시키면 예상하건데, 50% 보다 높은 70~80%의 이상 거래는 검출할 수 있을 것으로 보인다.

그러나 이번 케이스의 경우는 비정상 거래가 레이블링이 되어 있었기 때문에 이런 실험이 가능했지만, 일반적인 이상 거래 검출의 경우에는 레이블링되어 있는 비정상 거래를 얻기 힘들다. 그래서 오토인코더를 통해서 전체 데이터를 학습 시킨후에, 각 트랜잭션이나 그룹별(사용자나 쇼핑물의 경우 판매자등)로 오토인코더를 통해서 VALIDATION을 한후, 입출력값의 차이가 큰것의 경우에는 비정상 거래일 가능성이 매우 높기 때문에, 입출력값이 차이가 큰것 부터 데이터 탐색을 통하여 이상 거래 패턴을 찾아내고, 이를 통해서 임계치를 조정하여, 이상거래를 지속적으로 검출할 수 있도록 한후에, 이상 거래에 대한 데이터가 어느정도 수집되면 DNN등의 지도 학습 모델을 구축하여 이상 거래를 자동으로 검출할 수 있는 시스템으로 전환하는 단계를 거치는 방법이 더 현실적인 방법이 아닐까 한다.

14. 머신 러닝 파이프 라인 아키텍처

머신러닝을 공부하고 나서는 주로 통계학이나, 모델 자체에 많은 공부를 하는 노력을 드렸었다. 선형대수나 미적분 그리고 방정식에 까지 기본으로 돌아가려고 노력을 했었고, 그 중간에 많은 한계에도 부딪혔지만, 김성훈 교수님의 모두를 위한 딥러닝 강의를 접하고 나서, 수학적인 지식도 중요하지만 수학적인 깊은 지식이 없어도 모델 자체를 이해하고 근래에 발전된 머신러닝 개발 프레임워크를 이용하면 모델 개발이 가능하다는 것을 깨달았다.

계속해서 모델을 공부하고, 머신러닝을 공부하는 분들을 관심있게 지켜보고 실제 머신러닝을 사용하는 업무들을 살펴보니 재미있는 점이 모두 모델 자체 개발에만 집중한다는 것이다. 커뮤니티에 올라오는 글의 대부분은 어떻게 모델을 구현하는지 어떤 알고리즘을 사용하는지에 대한 내용들이 많았고, 실 업무에 적용하는 분들을 보면 많은 곳들이 R을 이용하여 데이터를 분석하고 모델링을 하는데, 데이터를 CSV 파일 형태로 다운 받아서 정제하고 데이터를 분석하고 모델을 개발하는 곳이 많은 것을 보았다. 데이터의 수집 및 전처리 및 개발된 모델에 대한 서비스에 대해서는 상대적으로 많은 정보를 접하지 못했는데, 예상하기로 대부분 모델 개발에 집중하기 때문이 아닌가 싶다.

엔지니어 백그라운드를 가진 나로써는 CSV로 데이터를 끌어다가 정제하고 분석하는 것이 매우 불편해 보이고 이해가 되지 않았다. 빅데이터 분석 시스템에 바로 연결을 하면, CSV로 덤프 받고 업로드 하는 시간등에 대한 고민이 없을텐데.” 왜 그렇게 할까 ?”라는 의문이 계속 생기기 시작하였다.

미니 프로젝트를 시작하다

이런 의문을 가지던중 CNN 네트워크 모델에 대한 대략적인 학습이 끝나고, 실제로 적용하면서 경험을 쌓아보기로 하였다. 그래서 얼굴 인식 모델 개발을 시작하였다. CNN 모델이라는 마법을 사용하면 쉽게 개발이 될줄 알았던 프로젝트가 벌써 몇달이 되어 간다. 학습용 데이터를 구하고, 이를 학습에 적절하도록 전처리 하는 과정에서 많은 실수가 있었고, 그 과정에서 많은 재시도가 있었다.

(자세한 내용은 <http://bcho.tistory.com/1174> ,
<https://www.slideshare.net/Byungwook/ss-76098082> 를 참조)

특히나 데이터 자체를 다시 처리해야 하는 일이 많았기 때문에, 데이터 전처리 코드를 지속적으로 개선하였고 개선된 코드를 이용하여 데이터를 지속적으로 다시 처리해서 데이터의 품질을 높여나갔는데, 처리 시간이 계속해서 많이 걸렸다.

자동화와 스케일링의 필요성

특히 이미지 전처리 부분은 사진에서 얼굴이 하나만 있는 사진을 골라내고 얼굴의 각도와 선글라스 유무등을 확인한후 사용 가능한 사진에서 얼굴을 크롭핑하고 학습용 크기로 리사이즈 하는 코드였는데 (자세한 내용 <http://bcho.tistory.com/1176>) 싱글 쓰레드로 만들다 보니 아무래도 시간이 많이 걸렸다. 실제 운영환경에서는 멀티 쓰레드 또는 멀티 서버를 이용하여 스케일링을 할 필요가 있다고 느꼈다.

또한 이미지 수집에서 부터 필터링, 그리고 학습 및 학습된 모델의 배포와 서비스 까지 이 전 과정을 순차적으로 진행을 하되 반복적인 작업이기 때문에 자동화할 필요성이 있다고 생각했다.

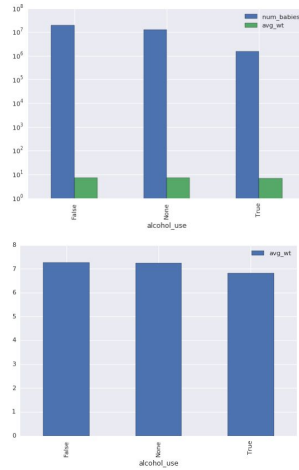
아이 체중 예측 모델을 통한 파이프라인에 대한 이해

그러던 중에 팀 동료로 부터 좋은 예제 하나를 전달 받게 되었다.

미국 아기들의 환경에 따른 출생 체중을 예측하는 간단한 선형 회귀 모델을 구현한 파이썬 노트북인데

(<https://github.com/GoogleCloudPlatform/training-data-analyst/blob/master/blogs/babyweight/babyweight.ipynb>) 하나의 노트북에 전체 단계를 모두 구현해놓았다.

```
In [53]: df = get_distinct_values('alcohol_use')
df.plot(x='alcohol_use', logy='num_babies', kind='bar');
df.plot(x='alcohol_use', y='avg_wt', kind='bar');
```



Other than the ever_born (the total number of babies born to this mother), the factors all seem to play a part in the baby's weight. Male babies are heavier on average than female babies. The mother's age and race play a part (age much more than race – teenage and middle-aged moms tend to have lower-weight babies). Twins, triplets, etc. are lower weight than single births. Premies weigh in lower as do babies born to single moms. Moms who use alcohol or cigarettes have babies that weigh lower on average.

In the rest of this notebook, we will use machine learning to combine all of these factors to come up with a prediction of a baby's weight.

Create ML dataset using Dataflow

Let's use Cloud Dataflow to read in the BigQuery data and write it out as CSV files.

Instead of using Beam/Dataflow, I had two other options:

1. Read from BigQuery directly using TensorFlow. However, using CSV files gives us the advantage of shuffling during read. This is important for distributed training because some workers might be slower than others, and shuffling the data helps prevent the same data from being assigned to the slow workers.
2. Use the BigQuery console (<http://bigquery.cloud.google.com>) to run a Query and save the result as a CSV file. For larger datasets, you may have to select the option to "allow large results" and save the result into a CSV file on Google Cloud Storage. However, in this case, I want to do some preprocessing (on the "race" column). If I didn't need preprocessing, I could have used the web console. Also, I prefer to script it out rather than run queries on the user interface, so I am using Cloud Dataflow for the preprocessing.

Note that after you launch this, the notebook will appear to be hung. Go to the GCP webconsole to the Dataflow section and monitor the running job. It took about 15 minutes for me.

데이터에 대한 분석을 통한 데이터 특성 추출, 추출된 특성을 통한 모델 개발, 모델 학습을 위한 데이터 전처리 그리고 학습 및 학습된 모델을 통한 예측 서비스 까지 모든 과정을 하나의 노트북에 구현해놓았다.

(시간이 있으면 꼭 보기를 강력 추천한다.)

흥미로운 점이 데이터 전처리를 Apache Beam이라는 데이터 처리 플랫폼을 썼고, 그 전처리 코드를 파이썬 노트북에 하나로 다 정리한것이다. (실제로 수행은 로컬에서도 가능하지만, 클라우드에서도 실행이 가능해서 충분한 스케일링을 지원한다.)

Apache Beam의 구글의 빅데이터 분석 프레임워크로 Apache Spark 과 같은 프레임워크라고 보면된다. Google Dataflow라는 이름으로 구글 클라우드에서 서비스가 되는데, Apache Beam이라는 오픈소스로 공개가 되었다. (<http://bcho.tistory.com/1123> <http://bcho.tistory.com/1122> <http://bcho.tistory.com/1124>)

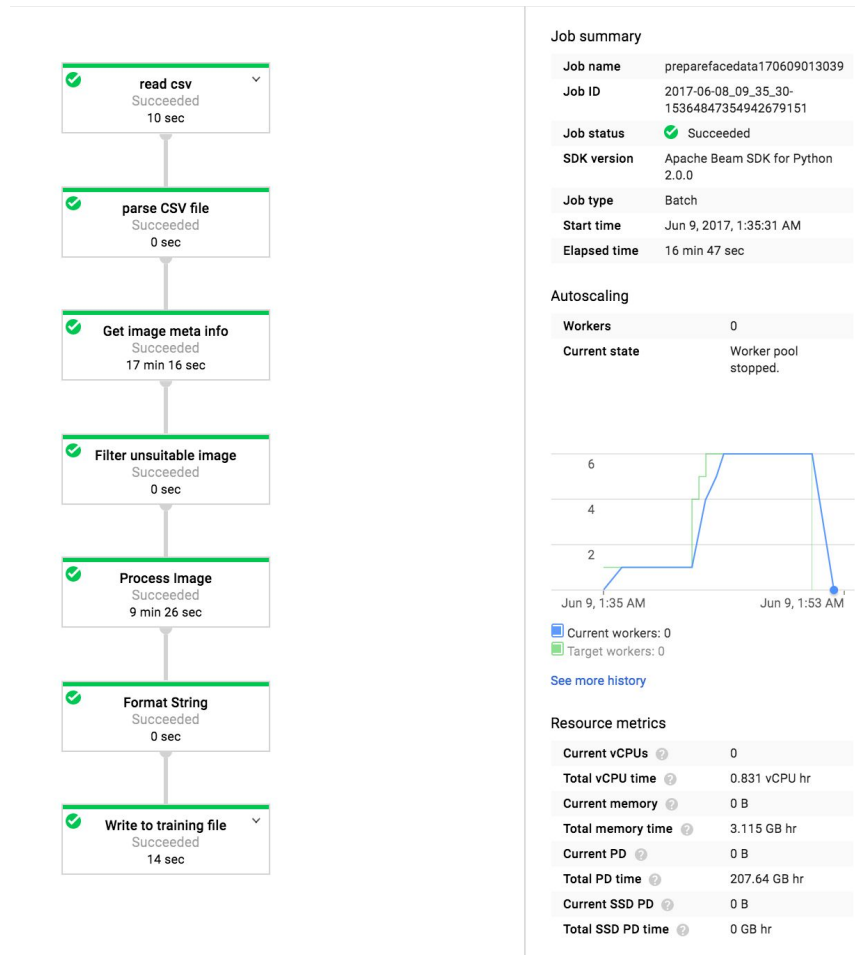
아 이렇게 하는구나 하는 생각이 들었고, 그즈음 실무에서 이와 같은 흐름으로 실제로 머신러닝을 수행하는 것을 볼 기회가 있었다.

데이터 전처리를 스케일링하다.

서비스가 가능한 수준의 전체 머신러닝 서비스 파이프라인을 만들어보고 싶어졌다. 마침 또 Apache Beam의 경우에는 예전에 Java 코드로 실시간 분석을 해본 경험이 있고 이번에 2.0 버전이 릴리즈 되서 이번에는 2.0에서 파이썬을 공부해보기로 하고 개발에 들어갔다.

특히 기존의 데이터 전처리 코드는 싱글 쓰레드로 돌기 때문에 스케일링에 문제가 있었지만, Apache Beam을 사용할 경우 멀티 쓰레드 뿐만 아니라 동시에 여러대의 머신에서 돌릴 수 있고 이러한 병렬성에 대해서는 크게 고민을 하지 않아도 Apache Beam이 이 기능을 다 제공해준다. 또한 이 데이터 전처리 코드를 돌릴 런타임도 별도 설치할 필요가 없이 커멘드 하나로 구글 클라우드에서 돌릴 수 가 있다. (직업 특성상 클라우드 자원을 비교적 자유롭게 사용할 수 있었다.)

Apache Beam으로 전처리 코드를 컨버팅 한결과 기존 싱글 쓰레드 파이썬 코드가 400~500장의 이미지 전처리에 1~2시간이 걸렸던 반면, 전환후에 대략 15~17분이면 끝낼 수 있었다. 전처리 중에는 서버의 대수가 1대에서 시작해서 부하가 많아지자 자동으로 5대까지 늘어났다. 이제는 아무리 많은 데이터가 들어오더라도 서버의 대수만 단순히 늘리면 수분~수십분내에 수십,수만장의 데이터 처리가 가능하게 되었다.



<그림. Apache Beam 기반의 이미지 전처리 시스템 실행 화면 >

Apache Beam 기반의 이미지 전처리 코드는

<https://github.com/bwcho75/facerecognition/blob/master/Preprocess%2Bface%2Brecognition%2Bdata%2Band%2Bgenerate%2Btraining%2Bdata.ipynb> 에 공개해 놨다.

머신러닝 파이프라인 아키텍처와 프로세스

이번 과정을 통해서 머신러닝의 학습 및 예측 시스템 개발이 어느 정도 정형화된 프로세스화가 가능하고 시스템 역시 비슷한 패턴의 아키텍처를 사용할 수 있지 않을까 하는 생각이 들었고, 그 내용을 아래와 같이 정리한다.

파이프라인 개발 프로세스

지금까지 경험한 머신러닝 개발 프로세스는 다음과 같다.

1. 데이터 분석

먼저 머신러닝에 사용할 전체 데이터셋을 분석한다. 그래프도 그려보고 각 변수간의 연관

관계나 분포도를 분석하여, 학습에 사용할 변수를 정의하고 어떤 모델을 사용할지 판단한다.

2. 모델 정의

분석된 데이터를 기반으로 모델을 정의하고, 일부 데이터를 샘플링하여 모델을 돌려보고 유효한 모델인지를 체크한다. 모델이 유효하지 않다면 변수와 모델을 바꿔 가면서 최적의 모델을 찾는다.

3. 데이터 추출 및 전처리

유효한 모델이 개발이 되면, 일부 데이터가 아니라 전체 데이터를 가지고 학습을 한다. 전체 데이터를 추출해서 모델에 넣어서 학습을 하려면 데이터의 크기가 크면 매번 매뉴얼로 하기가 어렵기 때문에 데이터 추출 및 전처리 부분을 자동화 한다.

4. 전체 데이터를 이용한 반복 학습 및 튜닝

모델 자체가 유효하다고 하더라도 전체 데이터를 가지고 학습 및 검증을 한것이 아니기 때문에 의외의 데이터가 나오거나 전처리에 의해서 필터링되지 않은 데이터가 있을 수 있기 때문에 지속적으로 데이터 추출 및 전처리 모듈을 수정해야 하고, 마찬가지로 모델 역시 정확도를 높이기 위해서 지속적으로 튜닝을 한다. 이 과정에서 전체 데이터를 다루기 때문에 모델 역시 성능을 위해서 분산형 구조로 개선되어야 한다.

5. 모델 배포

학습 모델이 완성되었으면 학습된 모델을 가지고 예측을 할 수 있는 시스템을 개발하고 이를 배포한다.

6. 파이프라인 연결 및 자동화

머신러닝의 모델은 위의 과정을 통해서 만들었지만, 데이터가 앞으로도 지속적으로 들어올 것이고 지속적인 개선이 필요하기 때문에 이 전과정을 자동화 한다. 이때 중요한것은 데이터 전처리, 학습, 튜닝, 배포등의 각 과정을 물 흐르듯이 연결하고 자동화를 해야 하는데 이렇게 데이터를 흐르는 길을 데이터 플로우라고 한다. (흔히 Luigi, Rundeck, Airflow와 같은 데이터플로우 오케스트레이션 툴을 이용한다)

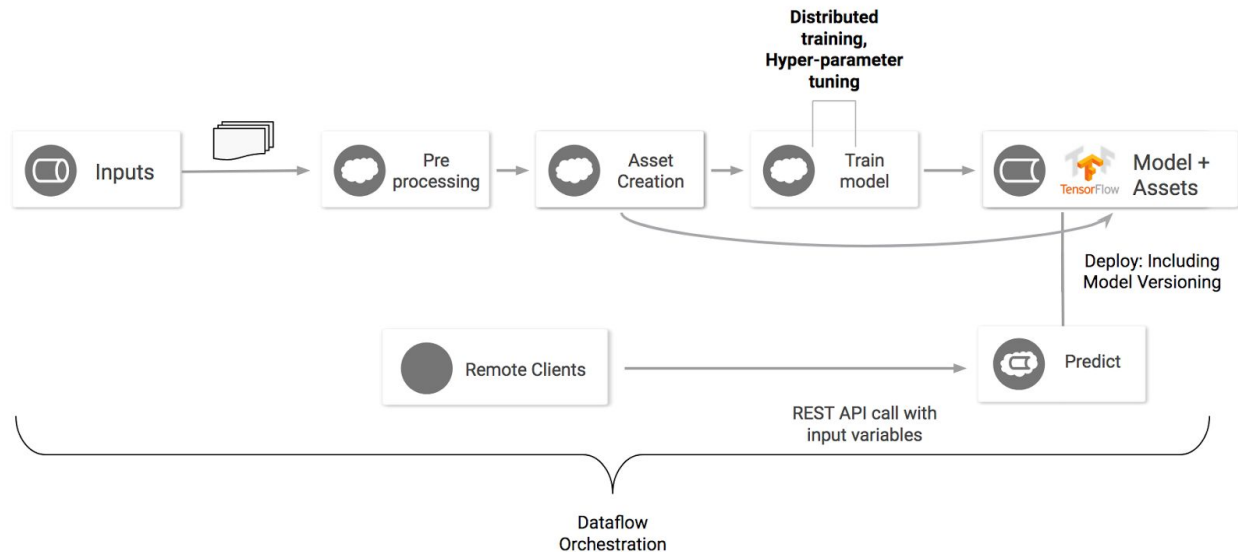
전체적인 프로세스에 대해서 좋은 영상이 있어서 공유한다.

<iframe

src="https://www.facebook.com/plugins/video.php?href=https%3A%2F%2Fwww.facebook.com%2Fgooglecloud%2Fvideos%2F766079863573416%2F&show_text=0&width=560" width="560" height="315" style="border:none;overflow:hidden" scrolling="no" frameborder="0" allowTransparency="true" allowFullScreen="true"></iframe>

아키텍처

위의 프로세스를 기반으로한 머신러닝 파이프라인 아키텍처 는 다음과 같다.



Inputs

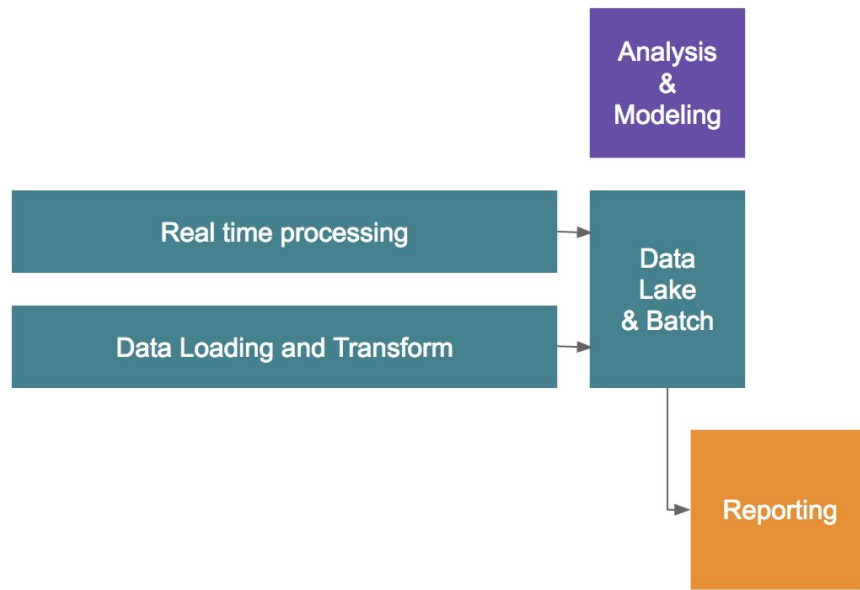
머신 러닝 파이프라인의 가장 처음단은 데이터를 수집하고 이 수집된 데이터를 저장하는 부분이다.

데이터 수집은 시간, 일, 주, 월과 같이 주기적으로 데이터를 수집하는 배치 프로세싱과, 실시간으로 데이터를 수집하는 리얼타임 프로세싱 두가지로 나뉘어 진다. 이 두 파이프라인을 통해서 데이터 소스로 부터 데이터를 수집하고 필터링하고 정제하여, 데이터 레이크에 저장한다. 이 구조는 일반적인 빅데이터 분석 시스템의 구조와 유사하다. (참고 자료 <http://bcho.tistory.com/984> <http://bcho.tistory.com/671>)

개인적으로 머신러닝을 위해서 중요한 부분 중 하나는 데이터 레이크를 얼마나 잘 구축하느냐이다. 데이터 레이크는 모든 데이터가 모여 있는 곳으로 보통 데이터 레이크를 구축할때는 많은 데이터를 모으는 데만 집중하는데, 데이터를 잘 모으는 것은 기본이고 가장 중요한 점은 이 모여 있는 데이터에 대한 접근성을 제공하는 것이다.

무슨 이야기인가 하면, 보통 머신러닝 학습을 위해서 학습 데이터를 받거나 또는 데이터에 대한 연관성 분석등을 하기 위해서는 데이터 레이크에서 데이터를 꺼내오는데, 데이터 레이크를 개발 운영 하는 사람과 데이터를 분석하고 머신러닝 모델을 만드는 사람은 보통 다르기 때문에, 모델을 만드는 사람이 데이터 레이크를 운영하는 사람에게 “무슨 무슨 데이터를 뽑아서 CSV로 전달해 주세요.” 라고 이야기 하는 것이 보통이다. 그런데 이 과정이 번거롭기도 하고 시간이 많이 걸린다.

가장 이상적인 방법은 데이터를 분석하고 모델링 하는 사람이 데이터 레이크 운영팀에 부탁하지 않고서도 손쉽게 빠르게 데이터에 접근해서 데이터를 읽어오고 분석을 할 수 있어야 한다. 직업 특성상 구글의 빅쿼리를 많이 접하게 되는데, 빅쿼리는 대용량 데이터를 저장할 수 있을 뿐만 아니라 파이썬 노트북이나 R 스튜디오 플러그인을 통해서 바로 데이터를 불러와서 분석할 수 있다.



<그림 INPUT 계층의 빅데이터 저장 분석 아키텍처>

Pre processing & Asset creation

Pre processing은 수집한 데이터를 학습 시스템에 넣기 위해서 적절한 데이터만 필터링하고 맞는 포맷으로 바꾸는 작업을 한다. 작은 모델이나 개발등에서는 샘플링된 데이터를 로컬에서 내려 받아서 R이나 numpy/pandas등으로 작업이 가능하지만, 데이터가 수테라에서 수백테라 이상이 되는 빅데이터라면 로컬에서는 작업이 불가능하기 때문에, 데이터 전처리 컴포넌트를 만들어야 한다.

일반적으로 빅데이터 분석에서 사용되는 기술을 사용하면 되는데, 배치성 전처리는 하둡이나 스파크와 같은 기술이 보편적으로 사용되고 실시간 스트리밍 분석은 스파크 스트리밍등이 사용된다.

Train

학습은 전처리된 데이터를 시스템에 넣어서 모델을 학습 시키는 단계이다. 이 부분에서 생각해야 할점은 첫번째는 성능 두번째는 튜닝이다. 성능 부분에서는 GPU등을 이용하여 학습속도를 늘리고 여러대의 머신을 연결하여 학습을 할 수 있는 병렬성이 필요하다. 작은 모델의 경우에는 수시간에서 하루 이틀 정도 소요되겠지만 모델이 크면 한달 이상이 걸리기 때문에 고성능 하드웨어와 병렬 처리를 통해서 학습 시간을 줄이는 접근이 필요하다. 작은 모델의 경우에는 NVIDIA GPU를 데스크탑에 장착해놓고 로컬에서 돌리는 것이 가성비 적으로 유리하고, 큰 모델을 돌리거나 동시에 여러 모델을 학습하고자 할때는 클라우드를 사용하는 것이 절대 적으로 유리하다 특히 구글 클라우드의 경우에는 알파고에서 사용된 GPU의 다음 세대인 TPU (텐서플로우 전용 딥러닝 CPU)를 제공한다. <https://cloud.google.com/tpu/> CPU나 GPU대비 최대 15~30배 정도의 성능 차이가 난다.

학습 단계에서는 세부 변수를 튜닝할 필요가 있는데, 예를 들어 학습 속도나 뉴럴 네트워크의 폭이나 깊이, 드롭 아웃의 수, 컨볼루션 필터의 크기등등이 있다. 이러한 변수들을 하이퍼 패러미터라고 하는데, 학습 과정에서 모델의 정확도를 높이기 위해서 이러한 변수들을 자동으로 튜닝할 수 있는 아키텍처를 가지는 것이 좋다.

텐서플로우등과 같은 머신러닝 전용 프레임워크를 사용하여 직접 모델을 구현하는 방법도 있지만, 모델의 난이도가 그리 높지 않다면 SparkML등과 같이 미리 구현된 모델의 런타임을 사용하는 방법도 있다.

Predict

Predict에서는 학습된 모델을 이용하여 예측 기능을 서비스 하는데, 텐서플로우에서는 Tensorflow Serv를 사용하면 되지만, Tensorflow Serv의 경우에는 bazel 빌드를 이용하여 환경을 구축해야 하고, 대규모 서비스를 이용한 분산 환경 서비스를 따로 개발해야 한다. 거기다가 인터페이스가 gRPC이다. (귀찮다.)

구글 CloudML의 경우에는 별도의 빌드등도 필요 없고 텐서 플로우 모델만 배포하면 대규모 서비스를 할 수 있는 런타임이 바로 제공되고 무엇보다 gRPC 인터페이스뿐만 아니라 HTTP/REST 인터페이스를 제공한다. 만약에 Production에서 머신러닝 모델을 서비스하고자 한다면 구글 CloudML을 고려해보기를 권장한다.

Dataflow Orchestration

이 전과정을 서로 유기적으로 묶어 주는 것을 Dataflow Orchestration이라고 한다.

예를 들어 하루에 한번씩 이 파이프라인을 실행하도록 하고, 파이프라인에서 데이터 전처리 과정을 수행하고, 이 과정이 끝나면 자동으로 학습을 진행하고 학습 정확도가 정해진 수준을 넘으면 자동으로 학습된 모델은 서비스 시스템에 배포하는 이 일련의 과정을 자동으로 순차적으로 수행할 수 있도록 엮어 주는 과정이다.

airbnb에서 개발한 Airflow나 luigi 등의 솔루션을 사용하면 된다.

아직도 갈길은 멀다.

얼굴 인식이라는 간단한 모델을 개발하고 있지만, 전체를 자동화 하고, 클라우드 컴퓨팅을 통해서 학습 시간을 단축 시키고 예측 서비스를 할 수 있는 컴포넌트를 개발해야 하고, 향후에는 하이퍼 패러미터 튜닝을 자동으로 할 수 있는 수준까지 가보려고 한다. 그 후에는 GAN을 통한 얼굴 합성들도 도전하려고 하는데, node.js 공부하는데도 1~2년을 투자한후에나 조금이나마 이해할 수 있게 되었는데, 머신러닝을 시작한지 이제 대략 8개월 정도. 길게 보고 해야 하겠다.

