Computation Graphs

: 서로 상호작용하는 연산을 만들고 실행하면서 머신러닝 알고리즘을 구현. -> 복잡한 기능 구조를 직관적으로 표현

Enity (Node, Vertex)

그래프: enity의 집합

node들은 변을 통해 서로 연결되어 있음

데이터 흐름 그래프에서의 변은 어떤 노드에서 다른 노드로 흘러가는 데이터의 방향을 지정

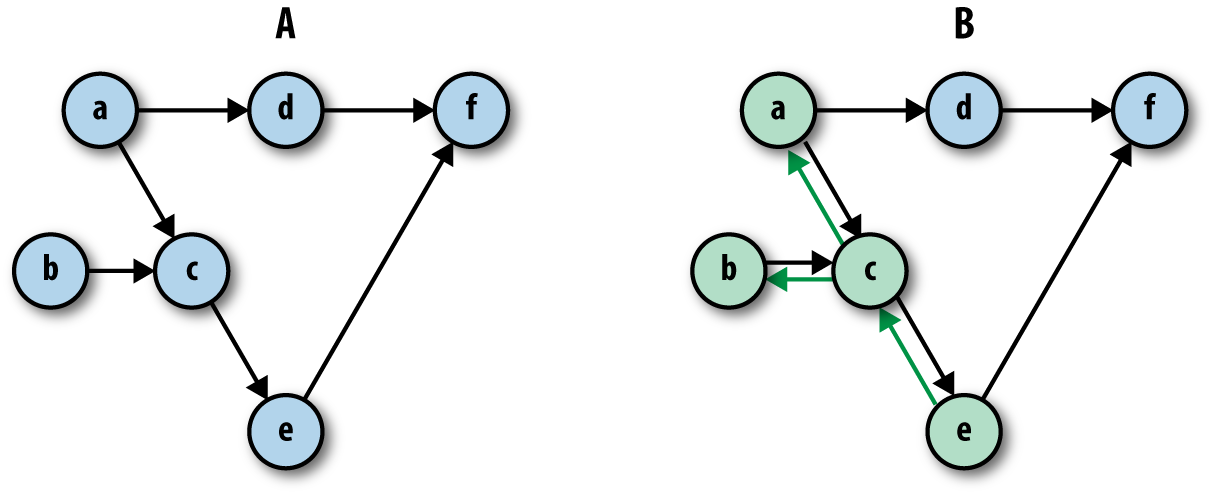
각 node는 하나의 연산을 나타내는데, 입력값을 받을 수 있으며 다른 노드로 전달할 결괏값을 출력할 수 있음

The Benefits of Graph Computation

node y의 입력이 node x의 결괏값에 영향을 받을 때 node y는 node x에 의존한다고 말함

두 node가 하나의 변으로 직접 연결: 직접적으로 의존

아닌 경우: 간접적으로 의존



Graphs, Session, Fetches

텐서플로의 동작은 그래프를 (1)만들고 (2)실행하는 두 단계

tf.<operator>

(Computation\_Graphs.py)

|  |  |  |
| --- | --- | --- |
| 텐서플로 연산 | 축약형 연산자 | 설명 |
| tf.add() | a + b | 대응되는 원소끼리 a와 b를 더한다 |
| tf.multiply() | a \* b | 대응되는 원소끼리 a와 b를 곱한다 |
| tf.subtract() | a – b | 대응되는 원소끼리 a에서 b를 뺀다 |
| tf.divide() | a / b | 파이썬 3.x의 방식으로 a를 b로 나눈다 |
| tf.pow() | a \*\* b | 대응되는 원소끼리 a의 b의 제곱을 계산한다 |
| tf.mod() | a % b | 대응되는 원소끼리 a를 b로 나눈 나머지를 구한다 |
| tf.logical\_and() | a & b | 대응되는 원소끼리 a와 b의 논리곱을 구한다 |
| tf.greater() | a > b | 대응되는 원소끼리 a > b의 참거짓값을 반환한다 |
| tf.greater\_equal() | a >= b | 대응되는 원소끼리 a >= b의 참거짓값을 반환한다 |
| tf.less\_equal() | a <= b | 대응되는 원소끼리 a <= b의 참거짓값을 반환한다 |
| tf.less() | a < b | 대응되는 원소끼리 a < b의 참거짓값을 반환한다 |
| tf.negative() | -a | a의 각 원소의 반대 부호의 값을 반환한다 |
| tf.logical\_not() | ~a | a의 각 원소의 반대의 참거짓값을 반환한다 |
| tf.abs() | abs(a) | a의 각 원소의 절댓값을 반환한다 |
| tf.logical\_or() | a | b | 대응되는 원소끼리 a와 b의 논리합을 구한다 |

Creating a Session and Running it

sess = tf.Session() # 그래프 시작

outs = sess.run(f) # 연산 그래프 실행

sess.close(f) # 세션을 닫아서 자원 해제

Constructing and Managing Graphs

새로운 그래프를 만들어 g에 할당

import tensorflow as tf

print(tf.get\_default\_graph())

g = tf.Graph() # g에 새로운 그래프 할당

print(g)

with 구문

콘텍스트 관리자에서 정의한 메서드로 블록의 실행을 감싸는 데 사용됨. 콘텍스트 관리자는 코드 블록을 서정하는 .\_\_enter\_\_()와 블록을 종료하는 .\_\_exit\_\_()라는 특별한 메서드 함수가 있음. 시작할 때 ‘설정’이 필요하고 코드가 잘 수행되었건 아니면 예외가 발생했던 간에 종료 시점에 ‘해제’가 필요한 코드를 수행하는 경우에 매우 편리한 방법임.

fetches

sess.run() 메서드에 변수를 인수로 지정하여 하나의 특정 노드의 실행을 요청

이 인수를 fetches라고 하며, 연산하고자 하는 그래프의 요소에 해당됨.

Flowing Tensor

tensorflow에서 Node와 edge가 표현되는 방법과 이들의 특성을 제어하는 방법을 설명

tensor의 작동 원리를 설명하기 위해서 텐서의 값을 초기화하는 데 사용되는 소스 연산에 초점을 맞춤.

Nodes are operations, Edges are tensor objection

tf.add() 등으로 그래프에서 노드를 만들 때, 실제로는 연산 인스턴스가 생성됨. 생성된 연산들은 그래프가 실행되기 전에는 실제 값을 내놓지 않고, 계산된 결과를 다른 노드로 전달할 수 있는 핸들, 즉 흐름으로서 참조됨.

소스 연산을 통한 속성 설정

tensorflow의 각 tensor 객체는 저마다의 name, shape, dtype 같은 속성이 있어 해당 객체의 특징을 식별하고 설정할 수 있음. 이 속성은 노드를 만들 때 설정할 수 있고, 지정하지 않으면 tensorflow가 자동으로 설정함. 소스 연산(source operation)이라는 이름의 연산으로 tensor 객체를 생성함으로써 확인할 수 있음. 소스 연산은 이전에 처리된 입력을 사용하지 않고 데이터를 생성하는 연산을 말함.

Data type

tensor 객체를 생성할 때 데이터 타입을 지정하지 않았기 때문에 tensorflow가 자동으로 데이터 타입을 유추함. ex) 5는 정수로 간주, 5.1과 같이 소수점이 있는 수는 부동소수점 숫자로 간주. tensor 객체를 만들 때 데이터 타입을 지정하여 명시적으로 선택할 수도 있음. tensor 객체의 데이터 타입은 dtype속성을 통해 확인할 수 있음.

형 변환

일치하지 않는 두 데이터 타입을 가지고 연산을 실행하면 예외가 발생하므로 그래프에서 데이터 타입이 일치하는지 확인하는 것은 매우 중요. tensor 객체의 데이터 타입을 변경하려면 tf.cast()에 해당 텐서와 새로이 적용할 데이터 타입을 각각 첫 번째와 두 번째 인수로 넘긴다.

|  |  |  |
| --- | --- | --- |
| 데이터 타입 이름 | 파이썬 데이터 타입 | 설명 |
| DT\_FLOAT | tf.float32 | 32비트 부동소수점 숫자 |
| DT\_DOUBLE | tf.float64 | 64비트 부동소수점 숫자 |
| DT\_INT8 | tf.int8 | 8비트 정수 |
| DT\_INT16 | tf.int16 | 16비트 정수 |
| DT\_INT32 | tf.int32 | 32비트 정수 |
| DT\_INT64 | tf.int64 | 64비트 정수 |
| DT\_UINT8 | tf.uint16 | 8비트 부호 없는 정수 |
| DT\_UINT16 | tf.uint32 | 16비트 부호 없는 정수 |
| DT\_STRING | tf.string | 가변 길이 바이트 배열 |
| DT\_BOOL | tf.bool | 참거짓값 |
| DT\_COMPLEX64 | tf.complex64 | 2개의 32비트 부동소숫점 숫자로 구성된 복소수 |
| DT\_COMPLEX128 | tf.complex128 | 2개의 64비트 부동소숫점 숫자로 수성된 복소수 |
| DT\_QINT8 | tf.qint8 | 양자화 연산에 사용되는 8비트 정수 |
| DT\_QINT32 | tf.qint32 | 양자화 연산에 사용되는 32비트 정수 |
| DT\_QUINT8 | tf.quint8 | 양자화 연산에 사용되는 8비트 부호 없는 정수 |

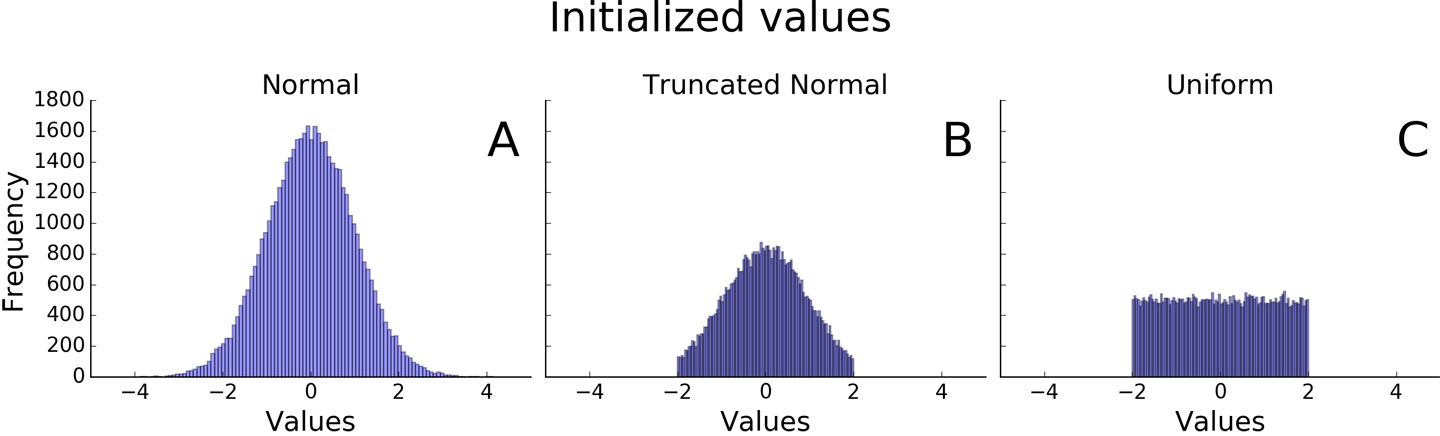
Tensor Arrays and Shapes

tensor라는 명칭이 서로 다른 두 개의 용도로 사용되기 때문에 혼란의 여지가 있음. 앞의 절에서 사용된 tensor는 그래프에서 연산의 결과에 대한 핸들로 파이썬 API에서 사용하는 객체의 이름임. 하지만 tensor는 n차원 배열을 가리키는 수학 용어이기도 함.

파이썬 리스트를 사용한 2\*3 행렬의 입력과 3차원 넘파이 배열을 사용한 2\*2\*3, 즉 2\*3 행렬 2개를 입력으로 사용한 예제

(ex\_Tensor\_Array.py)

난수 생성기는 tensorflow 변수의 초깃값을 지정하는 데 널리 사용되므로 매우 중요함. 유용한 초기화 방법의 예로, 먼저 형태, 평균, 표준편차를 tf.random.normal()에 순서대로 인수를 넣으면 **정규분포**(Normal)를 따르는 난수들을 생성할 수 있음. **절단정규분포**(Truncated Normal) 초기화는 이름에서 알 수 있듯 평균으로부터 표준편차를 기준으로 크거나 작은 값들을 제거한 것이고 **균등분포**(Uniform) 초기화는 정해진 구간 [a, b) 사이에서 균등하게 값을 추출함.



sess = tf.InteractiveSession() # 객체의 데이터 내용 쉽게 탐색

c = tf.linspace(0.0, 4.0, 5) # tf.linspace(a, b, n): a와 b 사이에서 같은 간격으로 n개의 값 생성

# .eval(): 세션 객체를 계속 참조하지 않고도 값을 자세히 들여다볼 수 있음.

print(“The content of ‘c’:\n {}\n”.format(c.eval())

sess.close()

**Outs:**

The content of ‘c’:

[ 0. 1. 2. 3. 4.]

|  |  |
| --- | --- |
| 텐서플로 연산 | 설명 |
| tf.constant(value) | 인수 value에 지정한 값 또는 값들로 채워진 tensor를 생성 |
| tf.fill(shape, value) | shape에 지정한 형태의 tensor를 만들고, value에 지정한 값으로 초기화 |
| tf.zeros(shape) | shape에 지정한 형태의 tensor를 만들고, 모든 원소의 값을 0으로 초기화 |
| tf.zeros\_like(tensor) | tensor와 동일한 타입과 형태의 tensor를 만들고, 모든 원소의 값을 0으로 초기화 |
| tf.ones(shape) | shape에 지정한 형태의 텐서를 만들고, 모든 원소의 값을 1로 초기화 |
| tf.ones\_like(tensor) | tensor와 동일한 타입과 형태의 tensor를 만들고, 모든 원소의 값을 1로 초기화 |
| tf.random\_normal(shape, mean, stddev) | 정규분포를 따르는 난수를 생성 |
| tf.truncated\_normal(shape, mean, stddev) | 절단정규분포를 따르는 난수를 생성 |
| tf.random\_uniform(shape, minval, maxval) | [minval, maxval) 구간의 균등분포의 값을 생성 |
| tf.random\_shuffle(tensor) | 첫 번째 차원에 따라 tensor를 무작위로 뒤섞음 |

행렬곱

tensorflow는 두 tensor 객체 A와 B에 대해 tf.matmul(A, B) 함수로 수행

행렬 A를 저장하는 tensor와 벡터 x를 저장하는 tensor가 있다고 가정하고, 다음과 같은 행렬곱 계산

Ax = b

(ex\_Matmul.py)

Name

각 tensor 객체마다 고유의 이름을 가짐. 이 이름은 TensorFlow 내부에서 사용하는 스트링 형태의 이름임. dtype과 마찬가지로 .name 속성을 사용해 객체의 이름을 확인할 수 있음.

with tf.Graph().as\_default():

c1 = tf.constant(4, dtype = tf.float64, name = ‘c’)

c2 = tf.constant(4, dtype = tf.int32, name = ‘c’)

Name scopes

크고 복잡한 그래프를 처리해야 하는 경우 이를 쉽게 추적하고 관리하기 위해 node를 묶는게 편리할 때가 있음. 이를 위해 node를 이름별로 계층적으로 그룹화할 수 있다. with 구문을 tf.nama\_scope(“접두사”)와 함께 이용.

with tf.Graph().as\_default():

c1 = tf.constant(4, dtype = tf.float64, name = ‘c’)

with tf.name\_scope(“prefix\_name”):

c2 = tf.constant(4, dtype = tf.int32, name = ‘c’)

c3 = tf.constant(4, dtype = tf.float64, name = ‘c’)

Variables, Placeholders, and Simple Optimization

Variables

주어진 모델의 매개변수를 조정하는 최적화 과정을 위해 tensorflow는 Variables라고 불리는 특별한 객체를 사용함. Session이 실행될 때마다 ‘리필’되는 다른 tensor 객체와 달리, variables는 그래프에서 고정된 상태를 유지할 수 있음. 이 특징이 중요한 까닭은 variables의 현재 상태가, 이후 반복 과정 속에서의 상태에 영향을 줄 수 있기 때문임.

variables의 사용은 두 단계로 나뉨. 먼저 tf.Variables() 함수를 사용해 variables를 만들고 어떤 값으로 초기화할지를 정의함. 이후 tf.global\_variables\_initializer() 메서드를 사용하여 session에서 초기화 연산을 명시적으로 수행해야 함. 이 메서드는 변수에 메모리를 할당하고 초깃값을 설정함.

(ex\_Variables.py)

Placeholders

입력 데이터를 생성하는 데 소스 연산을 사용하지 않고 입력값을 공급하기 위한 내장 구조를 placeholder라고 부름. 일단 그래프를 구성하고 그래프가 실행되는 시점에 입력 데이터를 밀어 넣는 데 사용함.

placeholder는 shape 인수를 선택적으로 사용할 수 있음. 값이 지정되지 않거나 None으로 지정되면 placeholder는 모든 크기의 데이터를 받을 수 있음. 흔히 샘플 데이터의 개수에 해당되는 행렬의 차원은 None을 사용하는 반면 특징의 길이는 고정된 값을 사용함.

ph = tf.placeholder(tf.float32, shape=(None, 10))

placeholder를 정의하면 반드시 입력값을 밀어 넣어야 하며 그렇게 하지 않으면 예외가 발생함. 입력 데이터는 딕셔너리 형태로 session.run() 메서드에 전달되며, 딕셔너리의 키는 placeholder 변수의 이름에 대응하며 딕셔너리의 값은 리스트나 넘파이 배열로 구성된 데이터의 값임.

sess.run(s, feed\_dict = {x: X\_data, w: w\_data})

(ex\_Placeholders.py)

Optimization

Training for predict

몇 개의 목표변수 y를 가지고 있을 때, 이를 특징 벡터 x를 사용해 설명한다고 가정해보자. 이를 위해 이 둘을 연관 지을 수 있는 모델을 선택해야 함. 학습 데이터의 값들은 원하는 관계를 잘 포착해내도록 모델을 조정하는 데 사용됨.

간단한 회귀문제에서의 회귀 모형

는 가중치 w와 절편 b를 입력 데이터 와 선형 조합한 결과라고 가정.

목표 는 에 가우시안 노이즈(정규분포를 따르는 노이즈) 를 더한 값임.

# placeholders와 variables 정의

x = tf.placeholder(tf.float32, shape=[None, 3])

y\_true = tf.placeholder(tf.float, shape=None)

w = tf.Variable([[0, 0, 0]], dtype=tf.float32, name=’weights’)

b = tf.Variable(0, dtype=tf.float32, name=’bais’)

placeholder 및 variable을 정의하면 이제 모델을 만들 수 있음. 이 예제는 통계학 용어로는 간단한 다변량 선형회귀로, 예측 출력값 y\_pred는 입력값을 담고 있는 x와 가중치 w의 행렬곱에 편향값 b를 더한 것이다.

y\_pred = tf.matmul(w, tf.transpose(x)) + b

Defining a loss function

모델을 생성했으면 모델의 성능을 평가할 수 있는 좋은 척도가 필요함. 모델이 예측한 값과 관측값 사이의 불일치를 포착하려면 ‘거리’를 반영하는 척도가 필요한데, 이 거리를 흔히 목적 또는 손실함수라고 부르며 이 함수의 값을 최소화하는 매개변수의 집합을 찾아네는 것이 바로 모델을 최적화하는 것임.

이상적인 손실 함수는 존재하지 않으며 보통 실제와 이론을 모두 고려해 가장 적절한 것을 선택하게 됨. 손실 함수는 모델에서 취하는 가정, 최소화의 난도, 반드시 피해야 하는 실수 등 여러 요소에 따라 선택한다.

MSE(Mean Squared Error) and cross entropy

보통 가장 흔히 사용하는 손실 함수는 평균제곱오차(MSE)임. 모든 샘플에서 실제 관측값과 모델 예측값 사이의 차를 제곱한 값의 평균이다.

직관적으로 받아들일 수 있는 MSE

이 선형회귀모형에서는 실제 관측값인 벡터 y\_true()와 모델의 예측값인 y\_pred()의 차이를 가지고 이 차이의 제곱을 계산하는 tf.square()를 사용함. 이 연산은 원소끼리 수행되고 tf.reduce\_mean() 함수를 사용해 평균을 구함.

loss = tf.reduce\_mean(tf.square(y\_true – y\_pred)

주로 범주형 데이터에 널리 사용되는 또 다른 손실 함수로는 앞장의 소프트맥스 분류기에서 사용했던 교차 엔트로피가 있다.

Cross entropy

정답 레이블이 하나인 분류 문제에서는 정답을 선택할 때 분류기가 부과하는 확률에 로그를 취한 값이 됨.

loss = tf.nn.sigmoid\_cross\_entropy\_with\_logits(labels=y\_true, logits=y\_pred)

loss = tf.reduce\_mean(loss)

교차 엔트로피는 두 분포 사이의 유사성을 재는 척도임. 딥러닝에서 사용되는 분류 모델은 보통 각 클래스의 확률값을 계산하므로 실제 클래스(분포 p)와 모델에서 제시한 클래스(분포 q)를 비교할 수 있음. 두 분포가 가까울수록 교차 엔트로피값은 더 작아짐.

The gradient descent optimizer

다음으로 알아야 할 것은 손실 함수를 최소화하는 방법임. 계산이나 분석을 통해 전역 최솟값을 찾아낼 수 있는 경우도 있지만, 대부분의 경우는 최적화 알고리즘을 사용해야 함. 최적화 함수는 점점 손실 함수의 값이 줄어드는 방향으로 반복적으로 가중치 집합을 갱신함.

가장 흔히 사용되는 접근 방법은 가중치의 집합에 대한 손실의 경사를 사용한 경사 하강법(gradient descent)임. 손실을 다변수 함수라 할 때 어떤 점 의 이웃 중 의 가장 가파른 내리막 경사 방향은 의 음의 경사 방향으로 이동할 때 얻어짐.

그러므로 에서 의 경사가 이고 라 할 때 충분히 작은 수 γ에 대해 다음이 성립함.

경사 하강법 알고리즘은 매우 복잡한 네트워크 구조에서도 잘 동작하므로 다양한 문제에 활용이 가능함. 볼록함수에서는 최솟값으로의 수렴이 보장되지만 비 볼록함수에서는 지역 최솟값에 갇히는 문제가 생길 수 있음. 실무에서는 충분히 좋은 결과를 내는 편이며, 딥러닝 분야의 큰 성공 역시 이를 입증한다.

Sampling methods

경삿값은 모델 매개변수에 대해서 계산되고 주어진 입력 샘플 를 사옹해 평가됨. 직관적으로 사용 가능한 정보의 총량으로부터 이익을 보려면 전체 데이터 집합의 경삿값을 계산하는 것이 타당한데, 예를 들어 데이터가 사용 가능한 메모리 이상을 필요로 한다면 매우 느리고 다루기도 어렵다는 단점이 있음.

가장 보편적인 방법은 확률적 경사 하강법(SGD)으로, 각 단계의 계산을 위해 알고리즘에 전체 데이터를 투입하는 대신 데이터의 부분집합을 순차적으로 추출함. 한 번에 적용하는 데이터의 갯수는 한 단계에 하나에서 수백 개까지 다양하지만 가장 일반적인 크기는 50~500개 사이이며, 그래서 이를 흔히 미니배치(mini-batch)라고 부름.

batch 수가 적을수록 빨리 학습할 수 있고, batch 크기가 작을수록 계산은 빨라짐. 그러나 샘플의 수가 적으면 하드웨어 사용률이 낮아지고 분산이 커지기 때문에 비용 함수의 결과가 크게 요동친다는 단점이 있음. 그럼에도 이 요동 덕분에 매개변수의 집합이 더 나은 지역 최솟값을 찾을 수 있는 새로운 곳으로 점프할 수 있으므로 꼭 나쁜 것만은 아니라는 점이 밝혀짐에 따라 상대적으로 작은 batch 사이즈는 이런 점에서 효과적이며 현재 전반적으로 선호되는 방법임.

Gradient descent in TensorFlow

tensorflow의 최적화 함수는 그래프에 새로운 연산을 추가하는 것만으로 경삿값을 계산할 수 있고 또 알아서 미분을 하며 경삿값을 계산함. 즉 연산 그래프의 연산과 구조에서 경사 함수를 자동으로 유도하며, 알아서 경삿값을 계산한다는 의미.

매개변수 중 학습률을 설정하는 것이 중요한데, 이 값은 각 갱신 반복이 얼마나 적극적으로 이루어지는가를 결정함. 손실 함수의 값이 쓸 만할 정도로 빠르게 감소해야 하지만 동시에 목표한 지점을 넘어서서 오히려 손실 함수의 결과가 더 커질 만큼 크게 이동하는 것은 바람직하지 않음.

우선 원하는 학습률을 인수로 정하고 GradientDescentOptimizer() 함수를 사용해 최적화 함수를 생성함. 그 후 optimizer.minimize() 함수에 손실 함수를 인수로 전달하여 변수를 갱신하는 학습 연산을 생성

optimizer = tf.train.GradientDescentOptimizer(learning\_rate)

train = optimizer.minimize(loss)

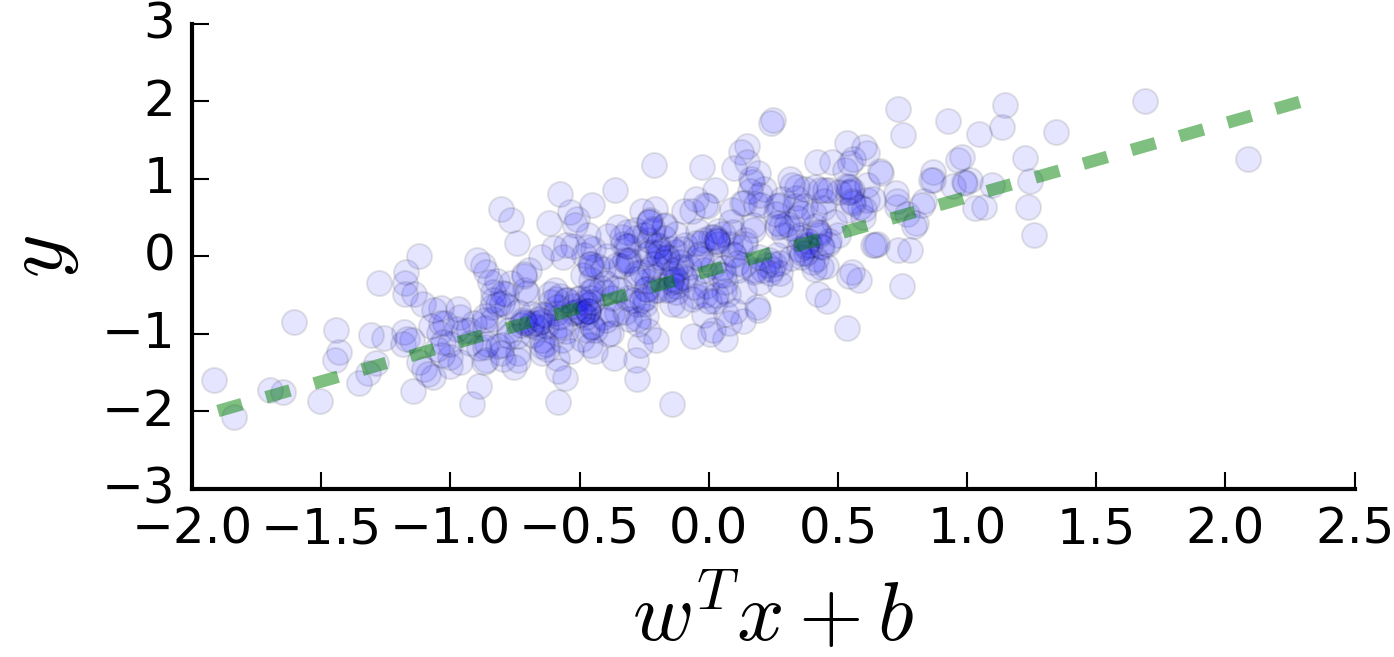
학습 연산은 sess.run() 메서드에 입력값이 주어진 후에 실행됨.

Examples

*Example 1: Linear Regression*

(ex\_Linear\_Regression.py)

목푯값이 각 샘플에 가우시안 노이즈를 추가한 어떤 입력 벡터 x의 선형 조합일 때 가중치 w와 편향값 b 를 찾아내는 문제임. 넘파이를 사용해 합성 데이터를 생성한 후 3개의 특징을 가진 벡터로 x의 샘플 2000개를 만들고 샘플 각각에 가중치의 집합 w([0.3, 0.5, 0.1])을 내적한 후 여기에 편향값 b(-0.2)와 가우시안 노이즈를 더함.



노이즈를 추가한 샘플

그 다음 정확한 예측이 가능하도록 모델을 최적화하여 가중치의 집합 w와 편향 b를 예측함. 각 반복마다 현재 매개변수를 한 번 업데이트하고 sess.run()메서드를 사용해 5번의 반복마다 추정 매개변수의 값을 출력하며, 10회 반복을 실행하고 name\_scope를 사용해 출력의 추론, 손실 함수의 정의, 학습 객체의 설정 및 생성과 관련된 부분을 그룹으로 묶음.

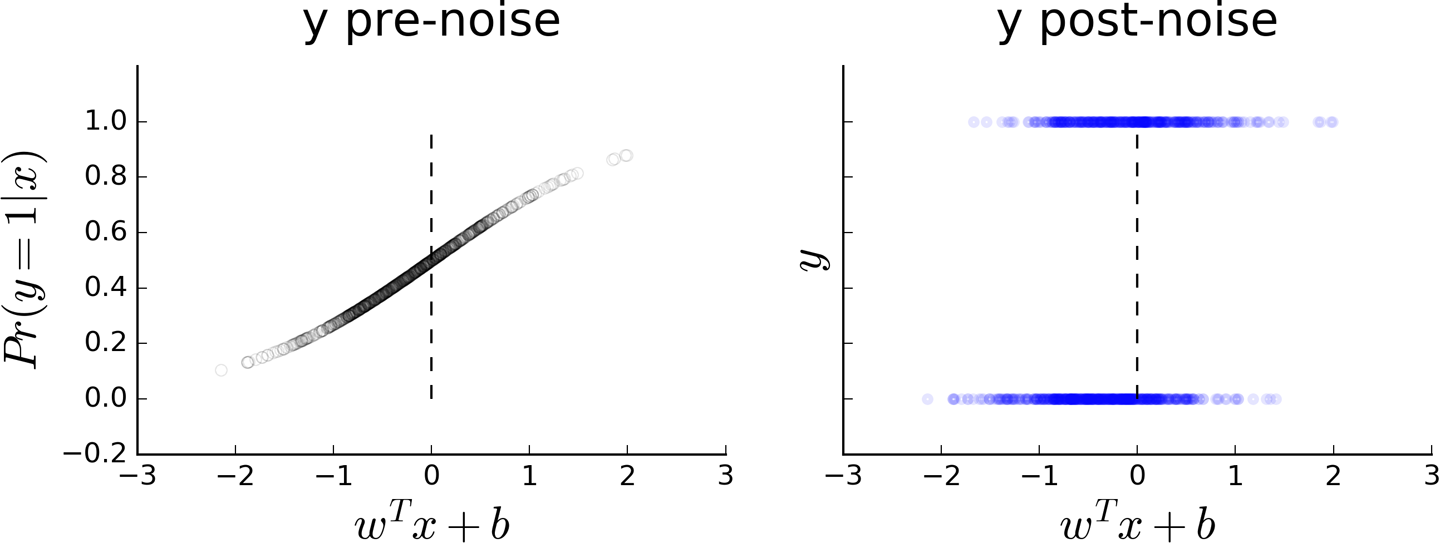
*Example 2: Logistic Regression*

(ex\_Logistic\_Regression.py)

로지스틱 회귀에서 선형 성분 는 로지스틱 함수라고 불리는 비선형 함수의 입력이 되며, 선형 성분의 값을 [0, 1]범위 안으로 눌러 넣는데 효과적임.

이 값은 참(1)또는 거짓(0)의 이진 출력 중 어떤 한 결과가 나올 확률로 간주할 수 있고, 모델의 비결정론적인 부분임.

로지스틱 함수는 일반화할 수 있는 개념이며, 곡선의 경삿값이나 최댓값이 다른 여러 로지스틱 함수를 통칭하고 시그모이드 함수하는 이름으로도 불림.



로지스틱 회귀에 사용하기 위해 생성한 데이터, 각 동그라미는 샘플을 나타냄. 왼쪽 그래프는 입력 데이터의 선형 조합을 로지스틱 함수에 적용하여 생성된 확률값들을 보여주며, 오른쪽 그래프는 왼쪽 그래프의 확률값 중 무작위로 추출하여 이진화한 목표 출력값을 보여줌.

사용할 손실 함수는 교차 엔트로피의 이진 버전으로, 로지스틱 회귀모형에 사용할 수 있는 손실 함수 중 하나임.

y\_pred = tf.sigmoid(y\_pred)

loss = -y\_true\*tf.log(y\_pred) – (1-y\_true)\*tf.log(1-y\_pred)

loss = tf.reduce\_mean(loss)

다행히 tensorflow는 이를 대신해서 사용할 내장 함수가 있음.

tf.nn.sigmoid\_cross\_entropy\_with\_logits(labels=, logits=)