Fundamentals of Deep Learning

Chapter 3. 텐서플로로 신경망 구현하기

3.1. 텐서플로란?

이 책 전체를 추상적인 딥러닝 모델을 설명하는 데 할애할 수도 있겠지만, 필자는 독자가 이 책을 다 읽을 무렵 딥 모델이 어떻게 동작하는지 이해하고 자신들이 직면한 문제에 대해 처음부터 모델을 만들 수 있는 능력을 갖추기를 바람. 이제 이론적으로 더 나은 딥러닝 모델에 대해 이해했으므로 이 알고리즘들의 일부를 소프트웨어로 구현하는 데 이번 장을 할애하겠음.

이 책에서 사용할 기본 도구는 텐서플로(TensorFlow)임. 텐서플로는 개발자가 딥러닝 모델을 설계하고 만들고 학습시키는 것을 더 쉽게 하도록 2015년 구글이 공개한 오픈소스 소프트웨어 라이브러리임. 텐서플로는 구글 개발자들이 사내 모델을 만드는 데 사용한 내부 라이브러리에서 비롯되었고, 내부 방식에 따라 테스트하고 검증했기 때문에 오픈소스 버전에 부가 기능이 추가될 것으로 기대되고 있음. 텐서플로는 개발자들에게 주어진 몇 가지 선택지의 하나에 불과하지만, 이것이 가진 세심한 디자인과 사용의 용이성 때문에 이 책에서는 텐서플로를 사용하기로 함. 다음 절에서는 텐서플로와 다른 대안들을 간단히 비교해보겠음.

크게 보면, 텐서플로는 데이터 흐름에 대한 그래프로써 사용자들이 마음대로 계산을 표현할 수 있게 하는 파이썬 라이브러리임. 이 그래프의 꼭짓점(node)은 수학적 연산들을 나타내며, 변(edge)들은 한 꼭짓점에서 다른 꼭짓점까지 전달되는 데이터들을 표현함. 텐서플로에서 데이터는 텐서(tensor)로 표현되며, 이 텐서는 다차원 배열임. 계산을 고려하는 이런 체계는 많은 다른 분야에서도 유용하지만, 텐서플로는 딥러닝 분야의 실무와 연구에서 주로 사용함.

신경망을 텐서로 생각하거나 그 반대로 생각하는 것은 무의미한 일이 아니며 오히려 이 책을 통해 발전시켜야 할 하나의 능력임. 이 방법으로 심층 신경망을 표현하는 것은 최신 하드웨어가 제공하는 속도의 이점을 취할 수 있고, 이는 모델 구현을 위한 깔끔하고 표현력 있는 방법을 제공함. 이번 장에서는 텐서플로의 기본 개념을 설명하고 두 개의 간단한 예제인 로지스틱 회귀와 다층 전방향 신경망을 살펴보겠음. 그러나 그 전에 텐서플로가 어떤 점에서 딥러닝을 표현하는 다른 체계들과 견줄 만한지에 대해 좀 더 살펴보자.

3.2. 텐서플로와 대안들을 어떻게 비교할까?

텐서플로 이외에도 심층 신경망을 만드는 체계로써 최근 몇 년간 급부상한 티아노와 토치, 카페, 니온, 케라스 등 많은 라이브러리가 있음. 두 가지 간단한 기준(표현력과 활발한 개발자 커뮤니티의 존재)을 바탕으로 텐서플로와 티아노, 토치로 범위를 좁혀 비교해보자.

세 가지 모두 두꺼운 개발차층을 가진 커뮤니티를 자랑함. 사용자들이 거의 제한 없이 텐서를 조작할 수 있고, 자동 미분이라는 특징이 있음. 그러나 이 중에서 토치는 루아로 쓰인 프레임워크라는 단점이 있음. 루아는 파이썬과 매우 유사한 스크립트 언어지만, 딥러닝 커뮤니티 밖에서는 일반적으로 덜 사용됨. 초보자들이 딥러닝 모델을 만들기 위해 완전히 새로운 언어를 배우는 부담을 주고 싶지 않아서 선택지를 텐서플로와 티아노로 더욱 한정했음.

이 두 선택지 사이에서 결정을 내리기가 어려웠음. 하지만 몇 가지 미묘한 이유로 결국 텐서플로를 선택했음. 첫째, 티아노는 딥러닝 구조의 특징 종류를 설정하는 데 상당한 시간이 걸리는 추가적인 그래프 편집 단계가 있음. 학습 시간에 비교하면 짧지만, 이 편집 단계는 새로운 코드를 작성하고 디버깅하는 동안 매우 실망스러웠음. 둘째, 텐서플로는 티아노보다 훨씬 더 명확한 인터페이스를 가지고 있음. 프레임워크의 표현력을 손상하지 않으면서 많은 종류의 모델을 훨씬 적은 수의 줄로 표현할 수 있음. 마지막으로, 텐서플로는 제품 제작을 염두에 두고 만들어진 반면에, 티아노는 연구자들이 거의 순수한 연구 목적으로 설계했음. 결과적으로 텐서플로는 바로 꺼내 쓸 수 있으며 실제 시스템에서 더 나은 선택을 할 수 있는 많은 기능을 가지고 있음. 티아노와 토치에 대한 친숙함이 오픈소스 예제들을 탐색할 때 굉장히 도움이 될 수 있지만, 이 프레임워크들의 개요는 이 책에서는 다루지 않겠음.

3.3. 텐서플로 설치하기

- 스킵

3.4. 텐서플로 변수 만들기와 조작하기

텐서플로에서 딥러닝 모델을 만들 때 모델의 파라미터를 나타내기 위해 변수를 사용함. 텐서플로 변수는 텐서가 포함된 메모리에 존재하는 버퍼임. 그러나 한 그래프가 실행될 때 인스턴스화되고 그 후에 즉시 깨끗하게 지워지는 일반적인 텐서와 달리 이 변수들은 다수의 그래프 실행 후에도 살아남음. 결과적으로 텐서의 변수는 다음 세 가지 속성을 가짐.

* 변수는 그래프가 처음 사용되기 전에 명시적으로 초기화되어야 함.
* 모델의 최적 파라미터 설정값을 탐색하므로 반복할 때마다 변수를 수정하기 위한 경사법을 사용할 수 있음.
* 추후 사용을 위해 변수에 저장된 값을 디스크에 저장하고 다시 불러올 수 있음.

이 세 가지 속성 덕분에 머신러닝 모델을 구축하는 데 텐서플로가 특별한 유용성을 지니게 됨.

변수를 만드는 것은 간단함. 그리고 텐서플로는 변수를 초기화하는 여러 방법을 제공함. 전방향 신경망에서 두 층 사이의 뉴런을 연결하는 가중치를 나타내는 변수의 초기화를 다음과 같이 시작해보자.

weights = tf.Variable(tf.random\_normal([300, 200], stddev=0.5), name=’weights’)

여기서 두 개의 인자를 tf.Variable로 넘김. 첫 번째 인자인 tf.random\_normal은 표준 편자를 0.5로 하는 정규분포로 초기화된 텐서를 생성하는 연산자임. 300개의 뉴런이 있는 층과 200개의 뉴런이 있는 층을 연결한 가중치들을 의미하는 300\*200 크기의 텐서를 지정하고, tf.Variable 호출로 이름을 전달함. 이 이름은 계산 그래프에서 적절한 노드를 참조할 수 있게 하는 고유한 식별자임. 이 예에서는 가중치가 학습될 수 있음을 의미함. 다시 말해, 자동으로 weights의 경사를 계산하고 적용함. weights를 학습할 수 없다면 tf.Variable을 호출할 때 다음과 같이 옵션 플래그를 전달함.

weights = tf.Variable(tf.random\_normal([300, 200], stddev=0.5), name=’weights’, trainable=False)

tf.random\_normal 외에도 다음과 같이 텐서플로 변수를 초기화하는 몇 가지 방법이 있음.

shape = [2, 3]

tf.zeros(shape, dtype=tf.float32, name=None)

tf.ones(shape, dtype=tf.float32, name=None)

tf.random\_normal(shape, mean=0.0, stddev=1.0, dtype=tf.float32, seed=None, name=None)

tf.truncated\_normal(shape, mean=0.0, stddev=1.0, dtype=tf.float32, seed=None, name=None)

tf.random\_uniform(shape, minval=0, maxval=None, dtype=tf.float32, seed=None, name=None)

tf.Variable을 호출할 때 계산 그래프에 다음과 같이 세 가지 연산이 추가됨.

* 변수를 초기화하는 데 사용하는 텐서를 생성하는 연산
* 변수 사용 전 초기화한 텐서로 변수를 채우는 책임이 있는 tf.assign 연산
* 변수의 현재 값을 유지하는 변수 연산

이것은 밑의 그림과 같이 나타낼 수 있음.



앞에서 언급한 세 연산처럼 어떤 텐서플로 변수를 사용하기 전에 변수가 요구한 값으로 적절하게 초기화되도록 tf.assign 연산이 실행되어야 함. tf.initialize\_all\_variables( )를 실행함으로써 변수를 초기화할 수 있는데, 이것은 그래프 내 모든 tf.assign 연산을 작동시킴. 또한, tf.initalize\_variables(vat1, var2, ...)를 사용하면 계산 그래프에서 선택적으로 특정 변수들만 초기화할 수 있음. 이것은 텐서플로에서 세션들을 논의할 때 좀 더 자세하게 설명하겠음.

3.5. 텐서플로 연산

변수 초기화를 설명할 때 연산에 대해 간단히 언급했는데, 이것은 텐서플로에서 가능한 연산의 극히 일부일 뿐임. 큰 그림에서 보면, 텐서플로 연산은 계산 그래프에서 텐서에 적용하는 추상적 변환을 나타냄. 연산은 사전에 제공되거나 런타임에 유추할 수 있는 속성을 가질 수 있음. 예를 들어, 한 속성은 예쌍되는 입력의 형태를 설명하는 역할을 함. 변수의 이름을 지정하는 것처럼 연산도 계산 그래프 안에서 쉬운 참조를 위해 선택적으로 이름 속성을 제공할 수 있음.

연산은 하나 또는 그 이상의 커널로 이루어지는데, 이 커널은 장치 종속적 구현을 나타냄. 예를 들어, 연산은 GPU에서 좀 더 효과적으로 표현되므로 CPU와 GPU 커널을 구별할 수 있음. 이렇게 구분되는 것은 행렬에 대한 텐서플로 연산이 많은 경우임.

사용 가능한 연산 종류를 간단히 소개하기 위해 텐서플로에서 연산의 다양한 범주를 상세히 기술한 텐서플로 백서 원본에서 밑의 표를 발췌함.

|  |  |
| --- | --- |
| 범주 | 예 |
| 요소별 수학적 연산 | Add, Sub, Mul, Div, Exp, Log, Greater, Less, Equal, ... |
| 배열 연산 | Concat, Slice, Split, Constant, Rank, Shape, Shuffle, ... |
| 행렬 연산 | MatMul, MatrixInverse, MatrixDeterminant, ... |
| 상태 저장 연산 | Variable, Assign, AssignAdd, ... |
| 신경망 구성 요소 | SoftMax, Sigmoid, ReLU, Convolution2D, MaxPool, ... |
| 체크포인트 관련 연산 | Save, Restore, ... |
| 큐와 동기화 연산 | Enqueue, Dequeue, MutexAcquire, MutexRelease, ... |
| 흐름 제어 연산 | Merge, Switch, Enter, Leave, NextIteration, ... |

3.6. placeholder 텐서

이제 텐서플로 변수와 연산에 대해 확실히 이해하고 있고 텐서플로 계산 그래프의 구성 요소에 대한 설명도 다함. 단 하나 빠진 것은 어떻게 입력을 딥 모델로 전달하는가임. 한 개의 변수는 한 번만 초기화되는 것을 의미하기 때문에 충분하지 않음. 그 대신에 계산 그래프가 한 번씩 실행될 때마다 값을 채울 수 있는 구성 요소가 필요함.

텐서플로는 placeholder라는 구조체를 사용해 이 문제를 해결했음. placeholder는 다음과 같이 인스턴스화되며, 마치 일반적인 텐서플로 변수와 텐서처럼 연산에 사용할 수 있음.

x = tf.placeholder(tf.float32, name=’x’, shape=[None, 784])

W = tf.Variable(tf.random\_uniform([784, 10], -1, 1), name=’W’)

multiply = tf.matmul(x, W)

placeholder를 정의하는데, 여기서 x는 float32에 저장된 데이터의 미니배치를 나타냄. x가 784개 열을 가진다는 것을 알 수 있으며, 이것은 각 데이터 표본이 784차원이라는 것을 의미함. 또한, x는 정의되지 않은 행의 수도 갖는다는 사실에 주목해야 함. 이것은 x가 무작위 수의 데이터 표본들로 초기화될 수 있음을 의미함. 데이터 표본마다 따로 W를 곱할 수도 있지만, 텐서로 전체 미니배치를 표현해 모든 데이터 표본에 대한 결과를 병렬로 계산할 수 있게 함. 그 결과 W에 상응하는 곱셈 텐서의 i번째 행이 i번째 데이터 표본에 곱해짐.

계산 그래프가 처음 만들어질 때 변수 초기화가 필요한 것처럼 placeholder도 계산 그래프가 실행될 때마다 초기값이 채워져야 함. 이 일이 어떻게 이루어지는지는 다음 절에서 더 자세히 살펴보자.

3.7. 텐서플로의 세션

텐서플로 프로그램은 세션을 사용해 계산 그래프와 상호작용함. 텐서플로 세션은 초반 그래프 생성을 담당하며, 모든 변수를 적절하게 초기화하고 계산 그래프를 실행하는 데 사용할 수 있음. 각 부분을 탐색해보기 위해 다음과 같은 간단한 파이썬 스크립트를 살펴보자.

import tensorflow as tf

from read\_data import get\_minibatch()

x = tf.placeholder(tf.float32, name=’x’, shape=[None, 784])

W = tf.Variable(tf.random\_uniform([784, 10], -1, 1), name=’W’)

b = tf.Variable(tf.zeros([10]), name=’biases’)

output = tf.matmul(x, W) + b

init\_op = tf.global\_variables\_initialize()

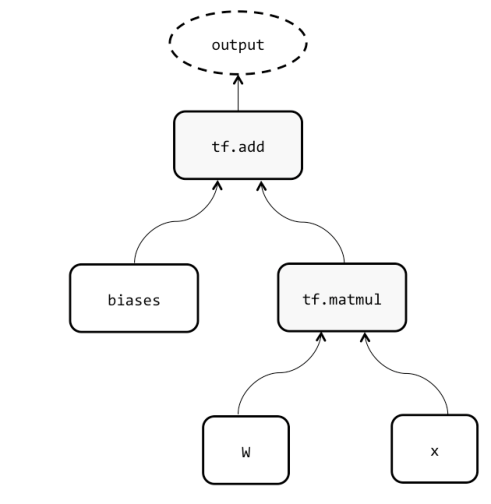
sess = tf.Session()

sess.run(init\_op)

feed\_dict = {‘x’ : get\_minibatch()}

sess.run(output, feed\_dict=feed\_dict)

import 문 다음 네 줄은 세션이 최종으로 인스턴스화될 때 세션에 의해 만들어진 계산 그래프를 설명함. 그래프는 밑의 그림과 같음. 그 다음 sess.run(init\_op)에서 초기화 연산을 실행하고자 세션 변수를 사용할 때 요구되는 변수들을 초기화함. 마지막으로 sess.run 함수를 다시 호출해 부분 그래프를 실행하는데, 이번에는 필요한 입력 데이터로 placeholder를 채운 feed\_dict와 함께 계산하기를 원하는 텐서를 전달함.



sess.run 인터페이스는 신경망 학습에도 사용할 수 있음. 이것은 MNIST에 첫 머신러닝 모델을 학습시키려 텐서플로를 이용할 때 더 자세히 살펴봄. 하지만 어떻게 코드의 한 줄이 이런 넓은 범위의 다양한 기능은 정확히 수행할까? 그 대답은 계산 그래프가 지닌 강력한 표현력에 있음. 이 모든 기능은 sess.run의 인수로써 전달되는 텐서플로 연산으로 표현됨. sess.run은 관련 하위 그래프를 구성하는 모든 종속성을 식별하기 위해 계산 그래프를 따라 내려가고, 모든 placeholder 변수가 feed\_dict로 채워진 식별된 하위 그래프에 속해 있는지 확인한 후, 원래 인수들을 평가하기 위해 하위 그래프를 역으로 따라 올라가게 하는 일을 함.

3.8. 변수 범위 탐색과 변수 공유

아직 이 문제에 부딪히지는 않았지만, 복잡한 모델을 만들 때는 인스턴스화하고 싶은 변수들의 큰 집합을 한 곳에서 함께 재사용하고 공유해야 할 때가 많음. 불행히도, 모듈화와 가독성을 강화하기 위해 노력하는 것이 자칫 의도치 않은 결과를 낳을 수 있음. 다음 예제를 살펴보자.

def my\_network(input):  
 W\_1 = tf.Variable(tf.random\_uniform([784, 100], -1, 1), name='W\_1')  
 b\_1 = tf.Variable(tf.zeros([100]), name='biases\_1')  
 output\_1 = tf.matmul(input, W\_1) + b\_1  
  
 W\_2 = tf.Variable(tf.random\_uniform([100, 50], -1, 1), name='W\_2')  
 b\_2 = tf.Variable(tf.zeros([50]), name='biases\_2')  
 output\_2 = tf.matmul(output\_1, W\_2) + b\_2  
  
 W\_3 = tf.Variable(tf.random\_uniform([50, 10], -1, 1), name='W\_3')  
 b\_3 = tf.Variable(tf.zeros([10]), name='biases\_3')  
 output\_3 = tf.matmul(output\_2, W\_3) + b\_3  
  
 print("Printing names of weight parameters")  
 print(W\_1.name, W\_2.name, W\_3.name)  
 print("Printing names of bias parameters")  
 print(b\_1.name, b\_2.name, b\_3.name)  
  
 return output\_3

이 신경망 설정은 세 개 층을 설명하는 여섯 개의 변수로 구성됨. 결과적으로 이 신경망을 여러 번 사용하려고 하면, my\_network처럼 여러 번 호출할 수 있는 작은 함수로 캡슐화하는 것이 좋음. 그러나 두 개의 다른 입력으로 이 신경망을 사용하려고 시도하면 다음과 같이 예상하지 못한 결과를 얻게 됨.

In [1]: i\_1 = tf.placeholder(tf.float32, [1000, 784], name=’i\_1’)

In [2]: my\_network(i\_1)

Printing names of weight parameters

W\_1:0 W\_2:0 W\_3:0

Printing names of bias parameters

biases\_1:0 biases\_2:0 biases\_3:0

In [1]: i\_2 = tf.placeholder(tf.float32, [1000, 784], name=’i\_2’)

In [2]: my\_network(i\_2)

Printing names of weight parameters

W\_1\_1:0 W\_2\_1:0 W\_3\_1:0

Printing names of bias parameters

biases\_1\_1:0 biases\_2\_1:0 biases\_3\_1:0

자세히 살펴보면, my\_network의 두 번째 호출은 첫 번째 호출과 동일한 변수를 사용하지 않음. 대신 변수들의 두 번째 집합을 만들었음. 대부분 경우에 사본 만들기를 원하지 않고 모델과 이 모델의 변수들을 재사용하길 원하는데, 이 경우 tf.Variable을 사용해서는 안 됨. 그 대신 텐서플로의 변수 범위 지정을 이용한 더 진보한 명명 체계를 사용해야 함.

텐서플로의 변수 범위 지정은 다음에 나오는 두 함수에 의해 광범위하게 제어됨.

* tf.get\_variable(<name>, <shape>, <initializer>)
* tf.variable\_scope(<scope\_name>)

텐서플로의 변수 범위 지정을 사용해 더 깔끔한 방식으로 my\_network를 다시 작성해보자. 변수들의 새 이름은 ‘layer1/W, layer2/b, layer2/W’ 등으로 다음과 같이 네임스페이스가 붙여짐.

def layer(input, weight\_shape, bias\_shape):  
 weight\_init = tf.random\_uniform\_initializer(minval=-1, maxval=1)  
 bias\_init = tf.constant\_initializer(value=0)  
 W = tf.get\_variable('W', weight\_shape, initializer=weight\_init)  
 b = tf.get\_variable('b', bias\_shape, initializer=bias\_init)  
  
 return tf.matmul(input, W) + b  
  
  
def refined\_my\_network(input):  
 with tf.variable\_scope('layer\_1'):  
 output\_1 = layer(input, [784, 100], [100])  
 with tf.variable\_scope('layer\_2'):  
 output\_2 = layer(output\_1, [100, 50], [50])  
 with tf.variable\_scope('layer\_3'):  
 output\_3 = layer(output\_2, [50, 10], [10])  
  
 return output\_3

앞의 코드에서 했던 것처럼 두 차례에 걸친 refined\_my\_network 호출을 다음과 같이 시도해보자.

In [1]: i\_1 = tf.placeholder(tf.float32, [1000, 784], name=’i\_1’)

In [2]: refined\_my\_network(i\_1)

In [1]: i\_2 = tf.placeholder(tf.float32, [1000, 784], name=’i\_2’)

In [2]: refined\_my\_network(i\_2)

ValueError: Over-sharing: Variable layer\_1/W already exists...

tf.Variable과는 달리 tf.get\_variable 명령어는 주어진 이름의 변수가 인스턴스화되지 않았는지를 확인함. 기본으로 공유는 허용되지 않지만, 변수 범위 내에서 공유될 수 있게 하려면 다음과 같이 명시적으로 지정할 수 있음.

with tf.variable\_scope('shared\_variables') as scope:  
 i\_1 = tf.placeholder(tf.float32, [1000, 784], name='i\_1')  
 refined\_my\_network(i\_1)  
 scope.reuse\_variables()  
 i\_2 = tf.placeholder(tf.float32, [1000, 784], name='i\_2')  
 refined\_my\_network(i\_2)

이렇게 하면 모듈성을 유지하면서 변수 공유도 허용할 수 있음. 그리고 부차적으로 명명 체계도 더 깔끔해짐.

3.9. CPU와 GPU로 모델 관리하기

텐서플로는 우리가 원한다면 모델을 만들고 학습시키기 위한 다양한 컴퓨팅 기기들을 사용할 수 있게 해줌. 지원하는 기기들은 문자열 ID로 표시되며 일반적으로 다음과 같이 구성됨.

* “/cpu:0” 컴퓨터의 CPU
* “/gpu:0” GPU가 하나라면, 컴퓨터의 첫 번째 GPU
* “/gpu:1” GPU가 또 하나 있다면, 컴퓨터의 두 번째 GPU

텐서플로 연산이 CPU와 GPU 커널이 모두 있고 GPU사용이 가능할 때, 텐서플로는 GPU용 구현 코드 사용을 자동으로 선택함. 계산 그래프가 어떤 장치들을 사용하는지 검사하기 위해 log\_device\_placement를 True로 설정해 텐서플로 세션을 다음과 같이 초기화할 수 있음.

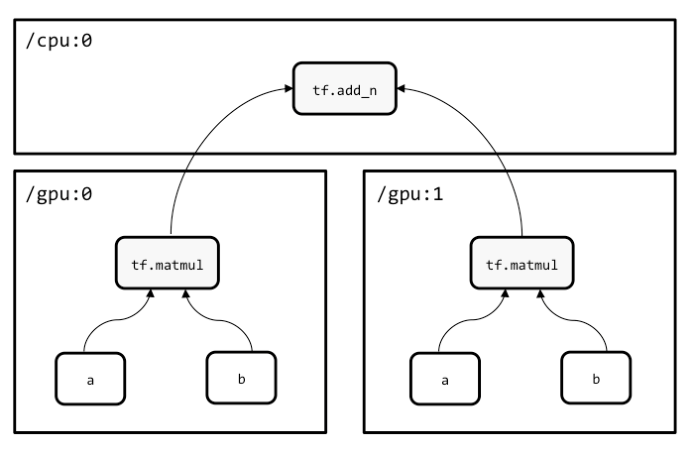
sess = tf.Session(config=tf.ConfigProto(log\_device\_placement=True))

특정 장치를 사용하고 싶다면 tf.device로 적절한 장치를 선택할 수 있음. 하지만 선택된 장치를 사용할 수 없다면 오류가 발생함. 선택된 장치가 존재하지 않을 때 텐서플로에서 가용 가능한 다른 장치를 찾기를 원한다면 다음과 같이 세션 allow\_soft\_placement 플래그를 세션 변수로 전달할 수 있음.

with tf.device('/gpu:2'):  
 a = tf.constant([1.0, 2.0, 3.0, 4.0], shape=[2, 2], name='a')  
 b = tf.constant([1.0, 2.0], shape=[2, 1], name='b')  
 c = tf.matmul(a, b)  
  
sess = tf.Session(config=tf.ConfigProto(allow\_soft\_placement=True, log\_device\_placement=True))  
  
sess.run(c)

또한, 텐서플로는 밑의 그림과 같이 타워 방식(tower-like fashion)의 모델을 작성해 여러 GPU를 사용하는 모델을 만들 수 있음. 다음은 다중 GPU 코드의 예제임.

c = []  
  
for d in ['/gpu:0', '/gpu:1']:  
 with tf.device(d):  
 a = tf.constant([1.0, 2.0, 3.0, 4.0], shape=[2, 2], name='a')  
 b = tf.constant([1.0, 2.0], shape=[2, 1], name='b')  
 c.append(tf.matmul(a, b))  
  
with tf.device('/cpu:0'):  
 sum = tf.add\_n(c)  
  
sess = tf.Session(config=tf.ConfigProto(log\_device\_placement=True))  
  
sess.run(sum)



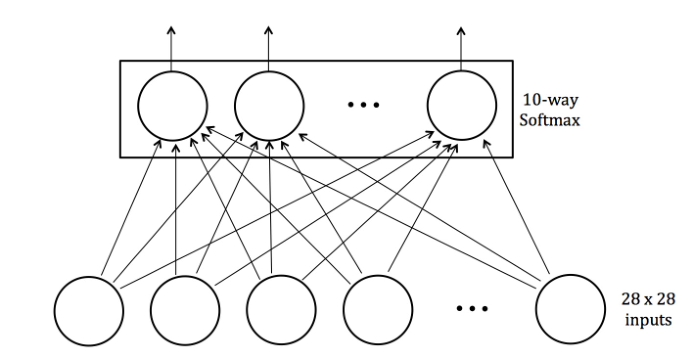
타워 방식으로 다중 GPU 모델 만들기

3.10. 텐서플로에서 로지스틱 회귀 모델 지정하기

지금까지 텐서플로의 모든 기본 개념을 전개해 왔음. 이제 MNIST 데이터셋을 다루기 위한 간단한 모델을 만들어보자. 기억을 되살려, 우리의 목표는 28\*28 흑백 이미지에서 손으로 쓴 숫자들을 알아보는 것임. 첫 번째 신경망은 로지스틱 회귀라는 단순한 머신러닝 알고리즘으로 구현함.

큰 그림에서 로지스틱 회귀는 입력이 대상 분류의 하나에 속할 확률을 계산하는 방법임. 우리 예제의 경우 주어진 입력 이미지가 0, 1, ..., 또는 9일 확률을 계산하는 것임. 모델은 망 연결의 가중치를 나타내는 행렬 W를 사용함. 이뿐 아니라 이전에 논의한 소프트맥스 표현을 사용해 입력 x가 i번째 분류에 속하는지를 추정하기 위한 바이어스에 해당하는 벡터 b도 사용함.

우리의 목표는 가능한 정확하게, 입력을 가장 효과적으로 분류하는 W와 b의 값을 학습시키는 것임. 밑의 그림에서 보듯이 로지스틱 회귀 신경망을 그림으로 표현할 수 있음(바이어스 연결은 혼란을 줄이기 위해 표시하지 않음).



로지스틱 회귀에 대한 신경망 해석은 오히려 단순하는 점에 주목해야 함. 이것은 어떠한 은닉층도 없는데, 이는 복잡한 관계를 학습할 능력이 제한적이라는 것을 의미함. 각 입력에 대해 10가지 가능한 결과들이 있기 때문에 크기가 10인 소프트맥스 출력이 있음. 또한, 이미지의 모든 픽셀에 한 개의 입력 뉴런이 있어야 해서 크기가 784인 입력층은 가짐. 앞으로 보게 되겠지만, 모델은 데이터셋을 정확하게 분류하는 방향으로 적절히 진행됨. 그러나 개선의 여지는 많이 남아 있음. 이 장의 뒷부분과 5장에서 정확도를 크게 개선하기 위해 노력할 것임. 하지만 그 전에 컴퓨터에 학습시킬 수 있는 로지스틱 회귀 신경망을 텐서플로로 어떻게 구현할 지 살펴보겠음.

다음 네 단계로 로지스틱 회귀 모델을 만듬.

1. **추론(inference)** 주어진 미니배치로 출력 분류에 대한 하나의 확률 분포를 만듬.
2. **손실(loss)** 오차 함수의 값을 계산함. 이 경우 교차 엔트로피 손실(cross-entropy loss)임.
3. **학습(training)** 모델 파라미터들의 경사 계산과 모델의 갱신을 담당함.
4. **평가(evaluate)** 모델의 효용성을 결정함.

MNIST 이미지들을 표현하는 784차원 벡터로 구성된 미니배치가 주어질 때 입력층과 출력층을 연결하는 가중치로 표현된 행렬과 입력의 곱에 소프트맥스를 적용해 로지스틱 회귀를 표현할 수 있음. 출력 텐서의 각 행은 미니배치에서 각 해당 데이터 표본의 출력 분류에 대한 확률 분포로 표현함.

def inference(x):  
 init = tf.constant\_initializer(value=0)  
 W = tf.get\_variable('W', [784, 10], initializer=init)  
 b = tf.get\_variable('b', [10], initializer=init)  
 output = tf.nn.softmax(tf.matmul(x, W), b)  
 return output

미니배치에 대한 정확한 레이블이 주어지면 데이터 표본마다 평균오차를 계산할 수 있는데, 다음과 같이 미니배치에 대한 교차 엔트로피 손실을 계산하는 코드를 사용해 이것을 구현할 수 있음.

def loss(output, y):  
 dot\_product = y \* tf.log(output)  
 # 축 0을 따라 축소하면 각 열이 단일 값으로 축소되지만  
 # 축 1을 따라 축소하면 각 행이 단일 값으로 축소됨.  
 # 일반적으로 축 i를 따라 축소하면 텐서의 i번째 차원이 크기 1로 축소됨.  
 xentropy = -tf.reduce\_sum(dot\_product, reduction\_indices=1)  
 loss = tf.reduce\_mean(xentropy)  
 return loss

그 다음 발생한 현재 비용이 주어지면 경사를 계산하고, 모델의 파라미터를 적절히 수정해야 함. 텐서플로는 특별한 학습 동작을 만들어주는 내장된 최적화 도구를 제공해 쉽게 이것을 해결함. 이 특별한 학습 동작은 텐서플로 세션을 통해 실행할 수 있음. 또한, 학습 동작을 만들 때 처리가 끝난 미니배치 수를 나타내는 변수를 전달한다는 점에 주의하자. 학습 동작이 실행될 때마다 진행 상황을 지속해서 추적하기 위해 이 단계 변수가 증가됨.

def training(cost, global\_step):  
 optimizer = tf.train.GradientDescentOptimizer(learning\_rate=0.1)  
 train\_op = optimizer.minimize(cost, global\_step=global\_step)  
 return train\_op

마지막으로, 검증 또는 테스트 데이터셋에 대한 모델 평가를 위해 간단한 계산용 하의 그래프를 함께 넣음.

def evaluate(output, y):  
 correct\_prediction = tf.equal(tf.argmax(output, 1), tf.argmax(y, 1))  
 accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32))  
 return accuracy

이것으로 로지스틱 회귀 모델에 대한 텐서플로 그래프 설정은 끝임.

3.11. 로지스틱 회귀 모델 기록하기와 학습시키기

모든 주요 부분을 한데 맞출 차례임. 모델을 학습시킬 때 중요한 정보를 남기기 위해 몇 가지 요약된 통계를 기록함. 예를 들어, 각 미니배치에 대한 비용과 검증 오차, 파라미터들의 분포를 기록하는 명령인 tf.summary.scalar19와 tf.summary.histogram을 사용함. 참고로, 비용 함수에 대한 스칼라 요약 통계는 다음과 같음.

def training(cost, global\_step):  
 tf.summary.scalar('cost', cost)  
 optimizer = tf.train.GradientDescentOptimizer(learning\_rate)  
 train\_op = optimizer.minimize(cost, global\_step=global\_step)  
 return train\_op

에포크마다 남겨진 모든 요약 통계를 수집하기 위해 tf.summary.merge\_all을 실행하고 디스크에 기록하기 위해 tf.summary.FileWriter를 사용함. 다음 절에서는 내장된 텐서보드(TensorBoard)라는 도구로 이 기록들을 어떻게 시각화하는지 설명함.

요약 통계뿐 아니라 모델 파라미터도 tf.train.Saver 모델 저장기를 사용해 저장함. 저장기는 기본으로 5개의 최신 체크포인트를 유지하며 향후 복원할 수 있음.

현재까지 설명한 코드를 모두 함께 넣으면, 다음과 같은 파이썬 스크립트를 얻게 됨.

# 파라미터  
learning\_rate = 0.01  
training\_epochs = 1000  
batch\_size = 100  
display\_step = 1  
  
with tf.Graph().as\_default():  
 # 28\*28=784 형태의 MNIST 데이터 이미지  
 x = tf.placeholder('float', [None, 784])  
 # 0~9 숫자 인식 => 10 클래스  
 y = tf.placeholder('float', [None, 10])  
  
 output = inference(x)  
  
 cost = loss(output, y)  
  
 global\_step = tf.Variable(0, name='global\_step', trainable=False)  
  
 train\_op = training(cost, global\_step)  
  
 eval\_op = evaluate(output, y)  
  
 summary\_op = tf.summary.merge\_all()  
  
 saver = tf.train.Saver()  
  
 sess = tf.Session()  
  
 summary\_writer = tf.summary.FileWriter('logistic\_logs/', graph\_def=sess.graph\_def)  
  
 init\_op = tf.initialize\_all\_variables()  
  
 sess.run(init\_op)  
  
 # 학습 주기  
 for epoch in range(training\_epochs):  
 avg\_cost = 0.  
 total\_batch = int(mnist.train.num\_examples/batch\_size)  
 # 모든 배치에 걸친 루프  
 for i in range(total\_batch):  
 mbatch\_x, mbatch\_y = mnist.train.next\_batch(batch\_size)  
 # 배치 데이터를 사용한 학습  
 feed\_dict = {x : mbatch\_x, y : mbatch\_y}  
 sess.run(train\_op, feed\_dict=feed\_dict)  
 # 평균 손실 계산  
 minibatch\_cost = sess.run(cost, feed\_dict=feed\_dict)  
 avg\_cost += minibatch\_cost/total\_batch  
 # epoch마다 기록 표시  
 if epoch & display\_step == 0:  
 val\_feed\_dict = {  
 x : mnist.validation.images,  
 y : mnist.validation.labels  
 }  
 accuracy = sess.run(eval\_op, feed\_dict=val\_feed\_dict)  
 print("Validation Error:", (1 - accuracy))  
 summary\_str = sess.run(summary\_op, feed\_dict=feed\_dict)  
 summary\_writer.add\_summary(summary\_str, sess.run(global\_step))  
 saver.save(sess, 'logistic\_logs/model-checkpoint', global\_step=global\_step)  
  
 print("Optimization Finished!")  
  
 test\_feed\_dict = {  
 x : mnist.test.images,  
 y : mnist.test.labels  
 }  
  
 accuracy = sess.run(eval\_op, feed\_dict=test\_feed\_dict)  
  
 print("Test Accuracy:", accuracy)

이 스크립트를 실행한 결과, 학습한 100 에포크 내의 테스트 데이터에 대해 91.9%의 최종 정확도를 얻었음. 썩 나쁘지는 않지만, 이 장의 마지막 부분에서 전방향 신경망으로 이 문제에 접근할 때 더 좋은 결과를 내도록 노력해보겠음.

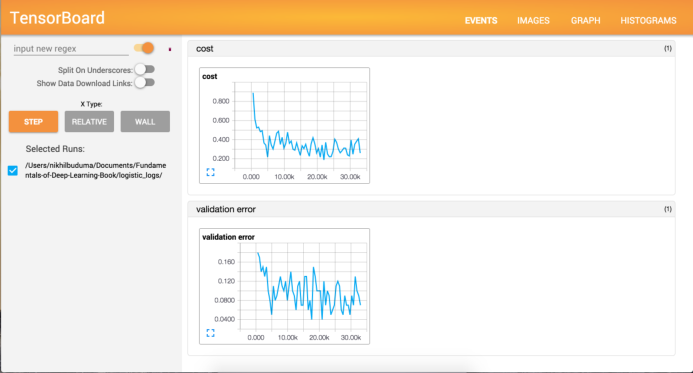
3.12. 텐서보드로 계산 그래프와 학습 시각화하기

앞 절에서 설명한 요약 통계 기록을 설정하면 수집된 데이터를 시각화하기 위한 준비는 끝남. 텐서플로는 텐서보드라는 시각화 도구를 제공하는데, 이 도구는 요약 통계를 통한 탐색에 사용하기 편한 인터페이스를 제공함. 텐서보드는 다음과 같이 실행하기 쉬움.

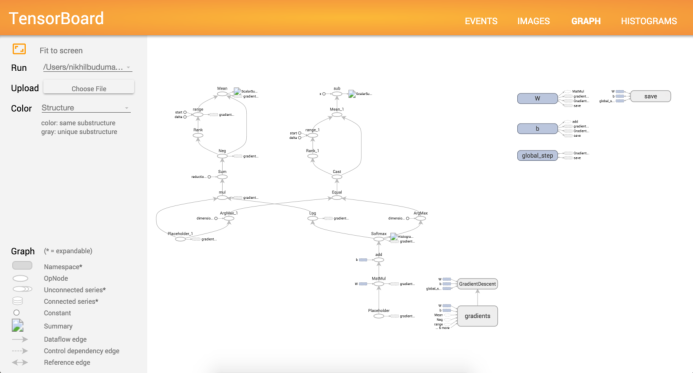
tensorboard –logdir=<absolute\_path\_to\_log\_dir>

여기서 logdir 플래그는 요약 통계들을 정렬하기 위한 디렉터리로 tf.summary.FileWriter를 설정해야 함. 상대 경로가 아닌 절대 경로로 전달해야 하는데, 그렇지 않으면 텐서보드가 기록을 찾을 수 없음. 텐서보드에서 성공적으로 작동됐다면 <http://localhost:6006/>에 데이터를 제공하며, 브라우저에서 이를 확인할 수 있음.

밑의 그림에서 보듯이 첫 번째 탭은 수집된 단일 수치에 대한 요약 정보를 포함함. 시간이 갈수록 미니배치 비용과 검증 오차 두 가지가 줄어드는 것을 볼 수 있음.



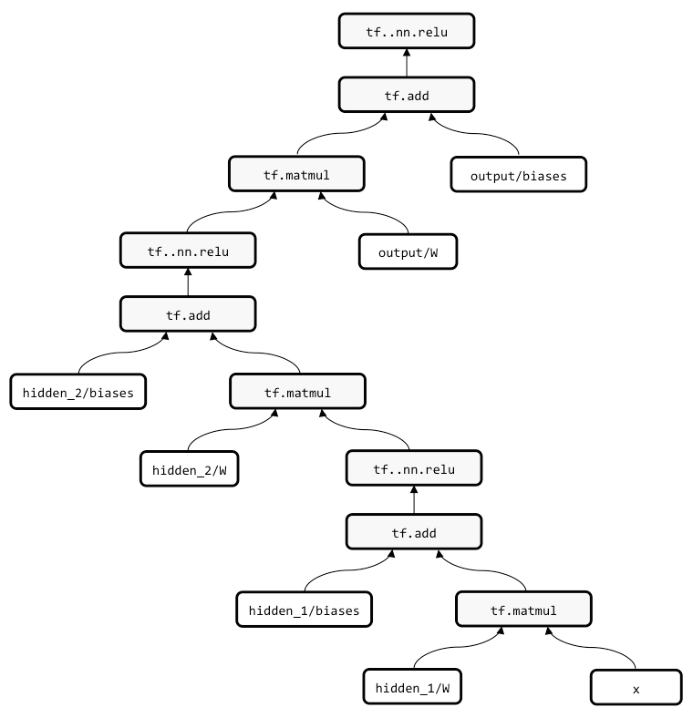
그리고 밑의 그림을 보면 지금까지 만들어 온 전체 계산 그래프를 시각화할 수 있는 탭도 있음. 이 탭은 해석하기 쉽지 않지만, 예기치 않은 작동에 직면했을 때 유용한 디버깅 도구가 될 수 있음.



3.13. 텐서플로에서 MNIST를 위한 다층 모델 만들기

로지스틱 회귀 모델을 사용할 때 MNIS 데이터셋에서 8.1%의 오차율을 얻었음. 이것이 괜찮아 보일지도 모르나 중요한 실제 응용 프로그램에서는 특별히 유용하지 않음. 예를 들어, 4자리 숫자를 쓴 개인 수표를 읽는 시스템을 사용한다면 거의 30%에 달하는 수표에서 오차가 발생하게 됨. 좀 더 실용적인 MNIST 숫자 판독기를 만들기 위해 도전 과제를 해결할 수 있는 전방향 신경망을 만들어보자.

밑의 그림처럼 각각 256개의 ReLU 뉴런을 갖는 2개의 은닉층으로 전방향 모델을 구성함.



로지스틱 회귀 예제에서 몇 가지 수정만으로 코드 대부분을 다음과 같이 재사용할 수 있음.

def layer(input, weight\_shape, bias\_shape):  
 weight\_stddev = (2.0/weight\_shape[0])\*\*0.5  
 w\_init = tf.random\_normal\_initializer(stddev=weight\_stddev)  
 bias\_init = tf.constant\_initializer(value=0)  
 W = tf.get\_variable("W", weight\_shape, initializer=w\_init)  
 b = tf.get\_variable("b", bias\_shape, initializer=bias\_init)  
 return tf.nn.relu(tf.matmul(input, W) + b)  
  
  
def inference(x):  
 with tf.variable\_scope("hidden\_1"):  
 hidden\_1 = layer(x, [784, 256], [256])  
 with tf.variable\_scope("hidden\_2"):  
 hidden\_2 = layer(hidden\_1, [256, 256], [256])  
 with tf.variable\_scope("output"):  
 output = layer(hidden\_2, [256, 10], [10])  
  
 return output

새로운 코드 대부분은 자명하지만, 초기화 전략은 몇 가지 추가 설명이 필요함. 심층 신경망의 성능은 해당 파라미터들의 효과적인 초기화에 크게 좌우됨. 다음 장에서 설명하겠지만, 매우 어려운 기본 확률적 경사 하강법(Vanilla Stochastic Gradient Descent)으로 최적화하는 심층 신경망의 오차 곡면에는 많은 특징이 있음. 이 문제는 모델에서 층의 수가 증가함에 따라 악화됨. 똑똑하게 초기화하는 것이 이 문제를 완화하는 한 가지 방법임.

ReLU 단위에 대한 연구에서는 망에서 가중치 분산은 이 되어야 한다고 설명하고 있음. 여기서 은 뉴런으로 들어가는 입력의 수임. 호기심 있는 독자라면 초기화 전략이 바뀔 때 어떤 일이 일어나는지를 조사해보기를 바람. 예를 들어, tf.random\_normal\_initializer를 tf.random\_uniform\_initializer로 다시 바꿀 때 로지스틱 회귀 예제에서는 성능이 심각하게 나빠짐.

마지막으로 조금 더 나은 성능을 위해 신경망의 추론 단계 대신 손실을 계산하면서 소프트맥스를 수행함. 결과는 다음과 같이 바뀜.

def loss(output, y):  
 xentropy = tf.nn.softmax\_cross\_entropy\_with\_logits(output, y)  
 loss = tf.reduce\_mean(xentropy)  
 return loss

300 번의 에포크 동안 이 프로그램을 실행하면 로지스틱 회귀 모델에 대한 훨씬 개선된 결과를 얻을 수 있음. 이 모델은 98.2%의 정확도로 동작하는데, 이것은 첫 번째 시도와 비교해서 숫자 당 오차율을 거의 78% 줄임.