

[2018년 여름]

# Tensorflow를 이용한 Deep Learning의 기초

제3강: Tensorflow Low-Level API  
Overview

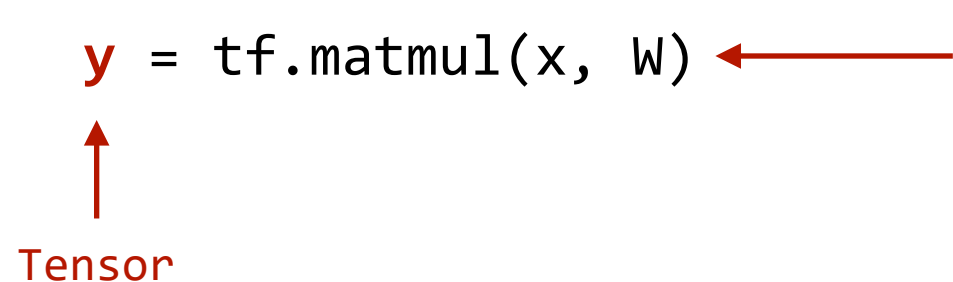


- Tensor는 스칼라, 벡터, 행렬 등을 임의의 차원으로 확장한 개념, 한마디로 다차원 배열
- Tensor의 rank는 차원의 개수, shape은 각 차원 별 길이를 표현하는 리스트

```
3.                                # a rank 0 tensor; a scalar with shape [],  
[1., 2., 3.]                     # a rank 1 tensor; a vector with shape [3]  
[[1., 2., 3.], [4., 5., 6.]]     # a rank 2 tensor; a matrix with shape [2, 3]  
[[[1., 2., 3.]], [[7., 8., 9.]]] # a rank 3 tensor with shape [2, 1, 3]
```

- TensorFlow는 텐서의 값을 표현하기 위해서 numpy array를 사용

- Tensorflow 프로그램에서 Tensor를 주로 어떤 (수학적) 연산의 결과로 생성된다.

$y = \text{tf.matmul}(x, W)$ 


여기서  $x$ 와  $W$ 는 변수, `placeholder`, 혹은 이전 layer에서 생성된 **tensor**일 것이다.

▶ Placeholder나 Variable도 Tensor인가? Yes. 일종의 추가 기능을 가진 **tensor wrapper**라고 볼 수 있다.

- 하지만 다음과 같이 다양한 방법으로 직접 Tensor를 생성할 수도 있다.

```

a = tf.constant(3.0, dtype=tf.float32)
rank_three_tensor = tf.ones([3, 4, 5])
my_image = tf.zeros([10, 299, 299, 3])
r = tf.random_uniform(shape=[10, 5], minval=-0.1, maxval=0.1, dtype=tf.float32)
integer_seq_tsr = tf.range(start=6, limit=15, delta=3)    # [6, 9, 12]
linear_tsr = tf.linspace(start=2.0, stop=3.0, num=5)    # [2., 2.25, 2.5, 2.75, 3.]
filled_tsr = tf.fill([2, 3], 9)

```

```
rank_three_tensor = tf.ones([3, 4, 5])  
print(rank_three_tensor.shape) # will print [3, 4, 5], 모든 텐서는 shape속성을 가짐  
  
matrix = tf.reshape(rank_three_tensor, [6, 10]) # Reshape into 6x10 matrix  
matrixB = tf.reshape(matrix, [3, -1]) # Reshape into a 3x20 matrix  
# -1은 알아서 계산하라는 뜻  
matrixAlt = tf.reshape(matrixB, [4, 3, -1]) # Reshape to a 4x3x5 tensor
```

지난 시간에는 numpy array의 ravel 함수로 3채널 이미지를 하나의 벡터로 flatten했지만  
이렇게 tf.reshape 함수로 flatten할 수도 있다.

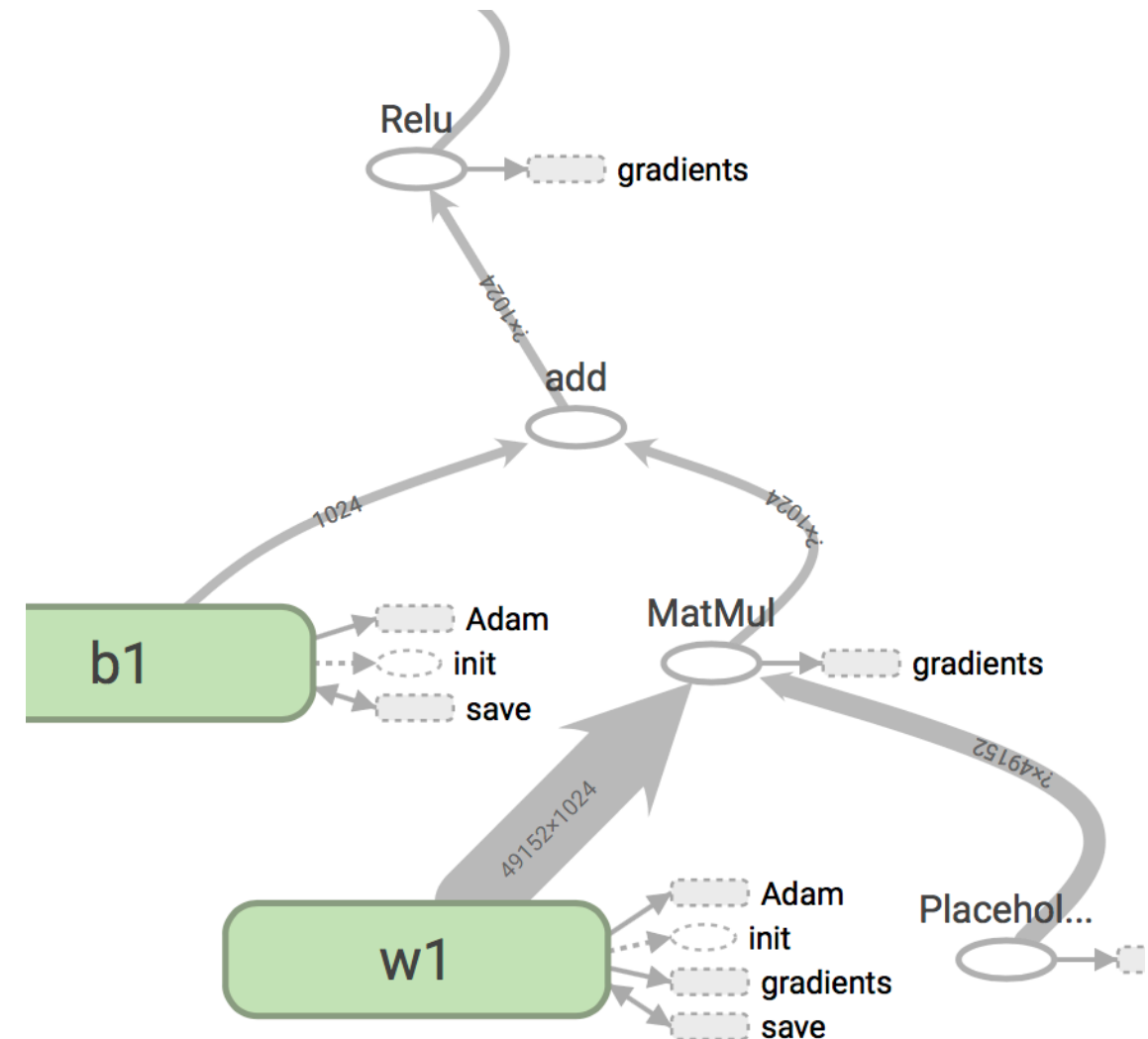
```
# Cast a constant integer tensor into floating point.  
float_tensor = tf.cast(tf.constant([1, 2, 3]), dtype=tf.float32)  
print(float_tensor.dtype)
```

지난 시간에는 numpy array의 astype 함수로 unit8 타입을 float32타입으로 변환했지만  
이렇게 tf.cast 함수로 변환할 수도 있다.

- **Tensorflow 프로그램은 보통 크게 두 부분으로 나뉘어진다.**
  - 그래프를 **구성**한다. (`tf.Graph`)
  - 그래프를 **실행**한다. (`tf.Session`)

## 그래프는 두 유형의 객체로 구성된다.

- **Operations (or 'ops')**: 그래프의 **노드**. 각각의 노드는 하나 혹은 그 이상의 텐서를 **소비**하고 하나 혹은 그 이상의 텐서를 **생성**하는 어떤 **연산**을 표현
- **Tensors**: 그래프의 **에지**. 그래프를 통해 흘러갈 **값**들을 표현



```
fc1 = tf.nn.relu(tf.matmul(images_batch, w1)+b1)
```



- 그래프는 “단지 프로그램의 구조를 사람에게 보여주기 위한 개념도” 같은 것이 아니다.
- Tensorflow 프로그램은 실제로 그래프를 구성한다. 즉, 그래프는 프로그램이 눈에 보이지 않게 뒤에서 실제로 구성하는 자료구조이다.
- 그렇다면 그래프는 어디에 있나? 그래프 자료구조에 접근할 수 있는 핸들은 어디에?
- 눈에 보이지 않는 이 그래프는 default graph라고 부르며, `tf.get_default_graph()` 함수로 액세스 할 수 있다.

```
c = tf.constant(4.0) ← 모든 텐서와 연산은 생성되면서 자동으로 default graph에 추가 되고  
assert c.graph is tf.get_default_graph()
```

모든 텐서는 자신이 소속된 그래프를 참조하는 멤버(.graph)를 가진다.

- ▶ default graph 대신 명시적으로 그래프를 생성하여 사용할 수도 있다.
- ▶ 2개 이상의 그래프가 필요할 경우 그렇게 한다.



```
a = tf.constant(3.0, dtype=tf.float32)
b = tf.constant(4.0) # also tf.float32 implicitly
total = a + b
print(a)
print(b)
print(total)
```

- ▶ 다음과 같이 출력된다.  
Tensor("Const:0", shape=(), dtype=float32)  
Tensor("Const\_1:0", shape=(), dtype=float32)  
Tensor("add:0", shape=(), dtype=float32)
- ▶ 기대했던 3.0, 4.0, 그리고 7.0과 같은 값은 출력되지 않는다. 왜냐하면 단지 그래프를 구성했을 뿐이기 때문이다.
- ▶ 출력된 tf.Tensor 객체들은 나중에 실행될 연산들의 결과를 표현할 뿐이다.

- TensorFlow는 TensorBoard라는 이름의 유틸리티를 제공한다. Tensorboard의 기능 중 하나는 **그래프를 시각화**하는 것이다.
- 먼저 다음과 같이 그래프를 **summary file**로 저장한다:

```
writer = tf.summary.FileWriter('tmp/logs')  
writer.add_graph(tf.get_default_graph())  
writer.flush()
```

← directory 이름은 바뀌도 상관없다.
- tmp/logs** 디렉토리에 다음과 같은 이름의 이벤트 파일이 생성된다:

```
events.out.tfevents.{timestamp}.{hostname}
```
- 터미널에서 다음의 명령으로 Tensorboard를 실행한다.(먼저 tfenv3.5를 activate한다.)

```
tensorboard --logdir=tmp/logs
```
- 브라우저에서 localhost:6006으로 접속하면 다음과 같은 그래프를 볼 수 있다.



- 그래프의 각 연산들은 유일한 이름을 가진다. 이 연산의 이름은 Python 변수의 이름인 `a`, `b`와는 무관하다.

- 코드를 다음과 같이 수정하고 Tensorboard로 그래프를 확인해보자.

```
a = tf.constant(3.0, dtype=tf.float32, name='MyConst1')
b = tf.constant(4.0, name='MyConst2')
total = tf.add(a, b, name='MyAdd')
print(a)
print(b)
print(total)
```

- 연산이 생성하는 텐서들 역시 이름을 부여받는데 “`add:0`”과 같이 연산의 이름 뒤에 `index`가 붙는 형태를 가진다. `print`문의 결과에서 텐서의 이름을 확인할 수 있다.

```
Tensor("MyConst1:0", shape=(), dtype=float32)
Tensor("MyConst2:0", shape=(), dtype=float32)
Tensor("MyAdd:0", shape=(), dtype=float32)
```

▶ 연산이나 텐서에 이름을 부여하는 것은 매우 큰 네트워크를 시각화하거나, 네트워크 실행 도중의 중간 값들에 관한 통계 분석 등에 유용하다.

- 그래프를 실행하기 위해서는 다음과 같이 `tf.Session` 객체를 생성하고 그것의 `run` 메서드를 실행한다.

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

```
print(sess.run(total)) ← “Tensor total을 fetch한다”라고 말한다.
```

- `Session.run` 함수로 어떤 **노드의 출력을 요청**하면 TensorFlow는 그래프를 **탐색하면서(backtrack)** 요청된 노드의 출력을 위해서 필요한 모든 다른 노드들을 `run`한다.

하나의 `tf.Tensor` 객체는 나중에 하나의 값을 생성해낼 부분적으로 정의된 계산을 표현한다. (A `tf.Tensor` object represents a partially defined computation that will eventually produce a value.)

`tf.Tensor`들은 값을 갖지 않는다. 다만 계산 그래프의 구성요소들에 대한 핸들 (**handle**)일 뿐이다. (`tf.Tensors` do not have values, they are just handles to elements in the computation graph.)

Tensorflow에서 많은 오해의 근원은 여기서 `tf.constant` 혹은 `tf.Variable()`, `tf.placeholder()` 등을 “생성자” 혹은 그에 준하는 “어떤 creator 메서드”일 거라고 생각하는데서 시작된다.

이게 생성자라면 어떤 객체를 생성한 후 그 객체에 대한 참조(reference) 혹은 어떤 핸들을 반환할 것이고, 따라서 Python 변수 `a`는 그 객체를 참조할 것이라고 생각하게 된다.



```
a = tf.constant(3.0)
```

그런데 `print(a)`를 해보면

```
Tensor("Const1:0", shape=(), dtype=float32)
```

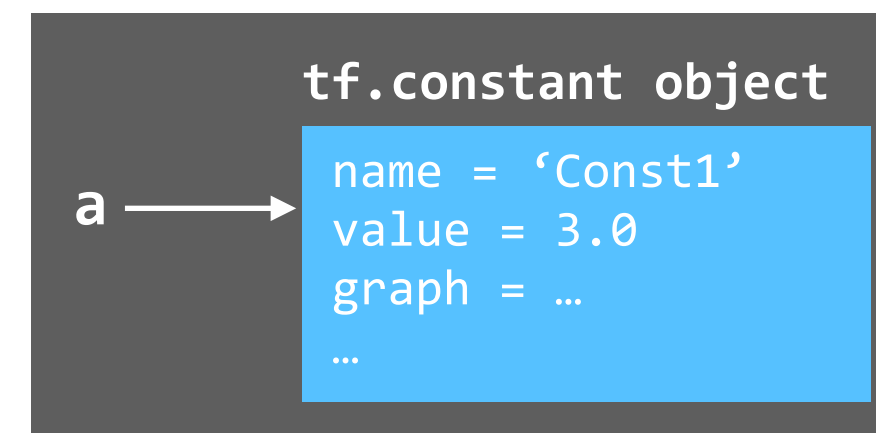
이런 식으로 출력되니까 `tf.constant`에 의해서 생성된 객체가 `tf.Tensor` 혹은 그것의 서브 클래스 객체일 거라고 생각하게 되는데,

default graph에는 `const1`이라는 이름의 노드가 존재한다.

그래프의 노드는 operation이라고 했는데,

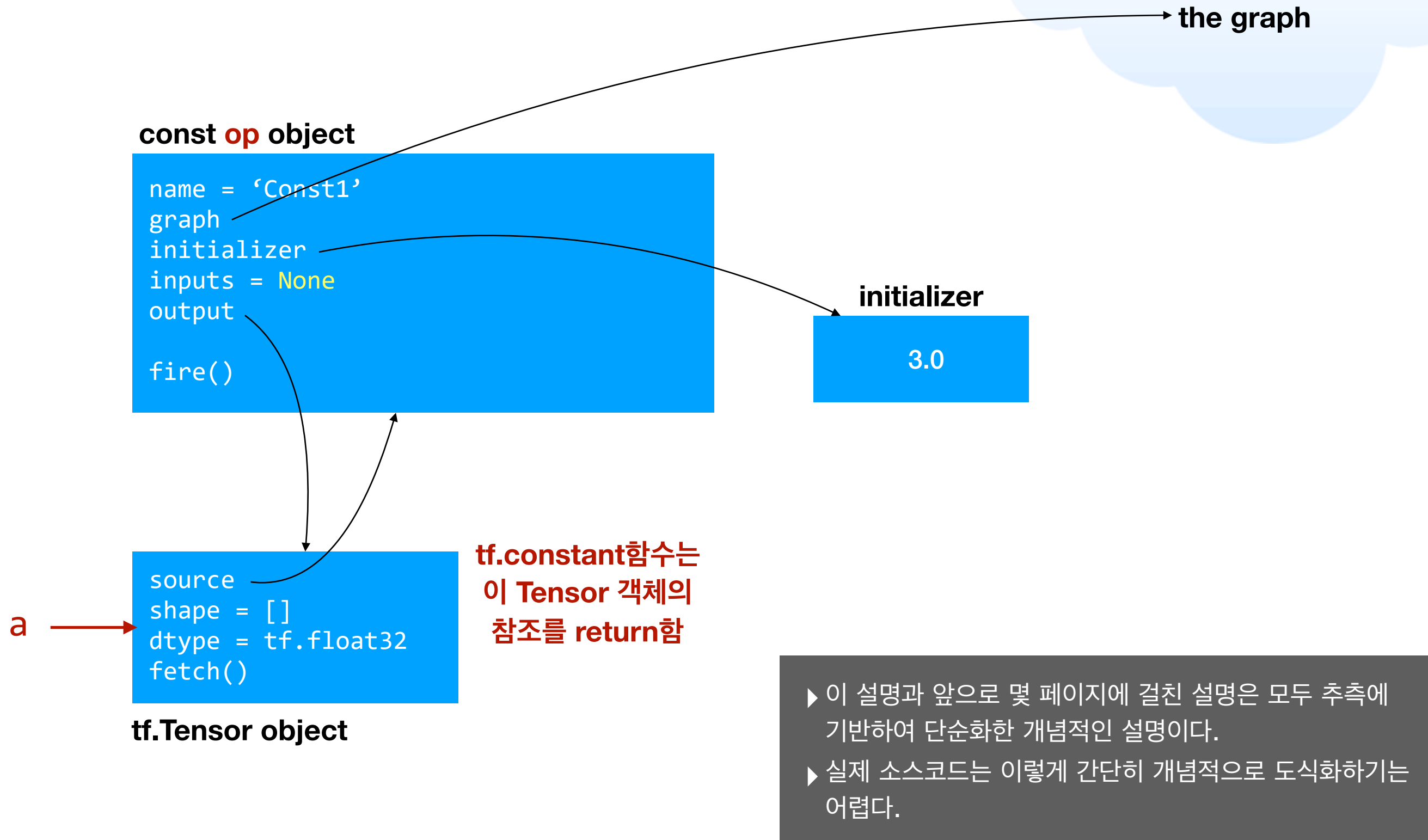
그럼 뭐가 어떻게 돌아가는거야....

what we wrongly expect...



# tf.constant는 생성자가 아닌 훨씬 복잡한 일을 하는 Python 함수이다.

`a = tf.constant(3.0)`





# tf.constant는 생성자가 아닌 훨씬 복잡한 일을 하는 Python 함수이다.

`a = tf.constant(3.0)`

→ the graph

## const op object

```
name = 'Const1'  
graph  
initializer  
inputs = None  
output  
fire()
```

initializer

3.0

**a** →

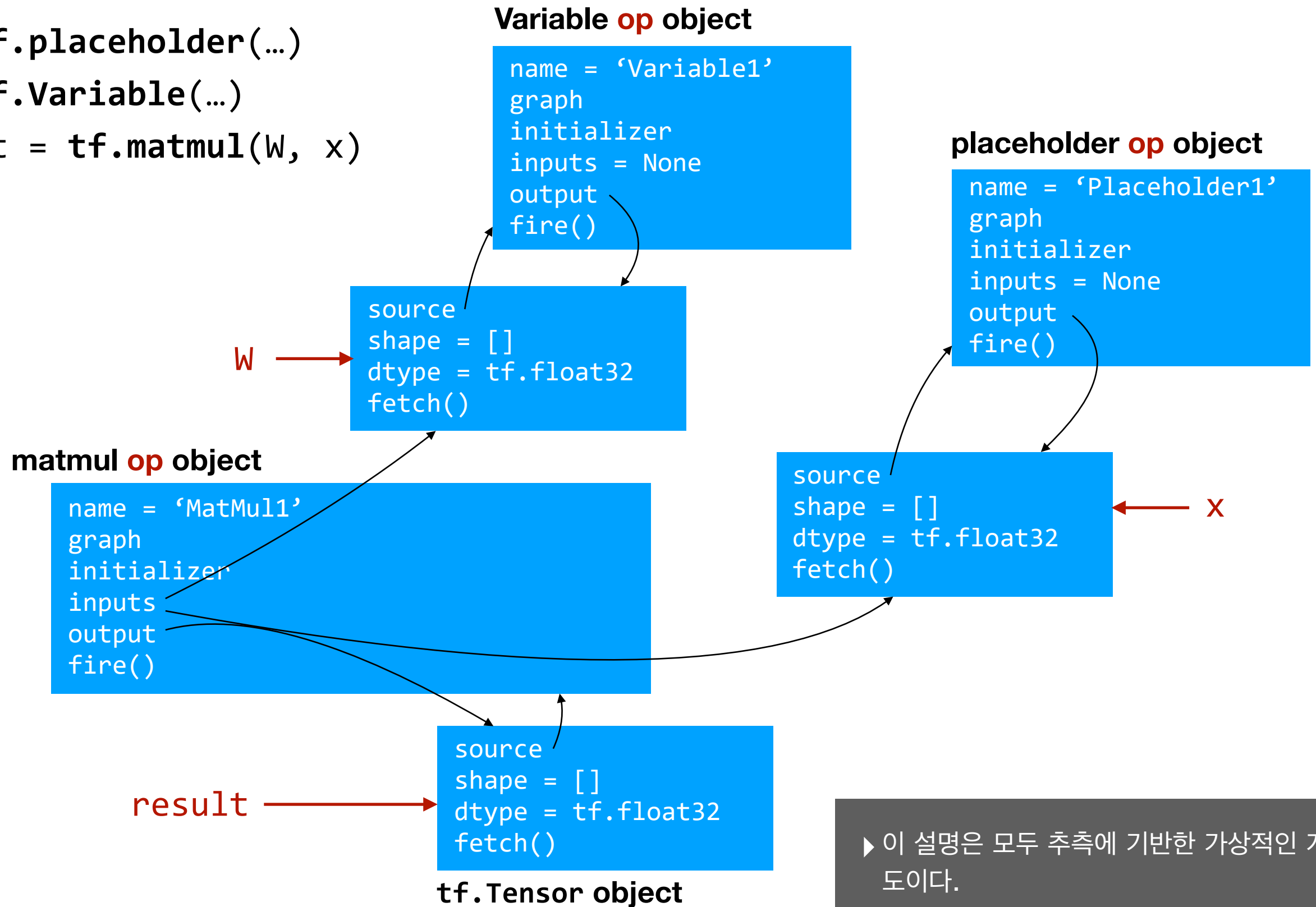
```
source  
shape = []  
dtype = tf.float32  
fetch()
```

**tf.Tensor object**

- ▶ `value = sess.run(a)`를 실행하면
- ▶ `run` 함수는 매개변수 `a`가 참조하는 객체의 `fetch()` 메서드를 실행한다.
- ▶ `fetch()` 메서드는 자신의 `source` 객체를 찾아가서 `fire()` 메서드를 실행한다.
- ▶ `fire()` 메서드는 `const operation`을 수행한다. `const operation`은 `initializer`가 제공하는 값 3.0을 `return`한다.
- ▶ `fetch` 함수는 `return`된 값 3.0을 `sess.run` 함수에게 `return`하고,
- ▶ `less.run` 함수는 3.0을 `return`한다.

# tf.constant는 생성자가 아닌 훨씬 복잡한 일을 하는 Python 함수이다.

```
x = tf.placeholder(...)  
W = tf.Variable(...)  
result = tf.matmul(W, x)
```



▶ 이 설명은 모두 추측에 기반한 가상의 개념도이다.

- 텐서의 값은 run과 run 사이에 보존되지 않는다. 하지만 한 번의 run 내에서는 오직 하나의 값만을 가진다.
- 예를 들어 텐서 `vec`은 각 run 호출마다 서로 다른 값을 가지지만, `out1`과 `out2`를 동시에 fetch하는 마지막 run에서는 일관된 하나의 값을 가진다.

```
vec = tf.random_uniform(shape=(3,))
out1 = vec + 1
out2 = vec + 2
print(sess.run(vec))
print(sess.run(vec))
print(sess.run((out1, out2)))
```

[출력]

```
[ 0.52917576  0.64076328  0.68353939]
[ 0.66192627  0.89126778  0.06254101]
```

```
(
  array([ 1.88408756,  1.87149239,  1.84057522], dtype=float32),
  array([ 2.88408756,  2.87149239,  2.84057522], dtype=float32)
)
```

텐서 `vec`이 일관된 값을 가짐을 확인할 수 있다.



- 어떤 TensorFlow 함수는 `tf.Tensor` 대신 `tf.Operation`을 반환한다. 이 `Operation`에 대해서 `run`을 호출하면 `None`이 반환된다.
- 이 `operation`을 `run`하는 것은 `side-effect`를 위해서이지 값을 반환받기 위해서가 아니다.

예:

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

**`init`이나 `train_op`를 `run`하면  
어떤 값도 반환되지 않는다.**

```
train_op = tf.train.AdamOptimizer().minimize(loss)  
_, _loss = sess.run([train_op, loss], feed_dict={  
    images_batch:images_batch_val,  
    labels_batch:labels_batch_val  
})
```

- 변수는 `tf.get_variable()` 함수로 생성한다. (`tf.Variable` 함수는 권장되지 않음)

```
v = tf.get_variable("my_variable", [1, 2, 3])
```



변수 이름 "my\_variable"은 학습된 변수의 값을 저장하고  
재사용하거나 모델을 export할 때 사용된다.

- default dtype은 `tf.float32`이고, default initializer인 `tf.glorot_uniform_initializer`에 의해서 랜덤하게 초기화된다.  
(`tf.glorot_uniform_initializer`는 Xavier initializer라고도 불리며 대체적으로 최선의 방법으로 간주되고 있다.)
- dtype과 initializer를 명시할 수도 있다.

```
my_int_variable = tf.get_variable("my_int_variable", [1, 2, 3],  
                                dtype=tf.int32, initializer=tf.zeros_initializer)  
other_variable = tf.get_variable("other_variable", dtype=tf.int32,  
                                initializer=tf.constant([23, 42]))
```

- 변수는 사용하기 전에 먼저 초기화해야 한다.
- 학습 가능한 (`tf.GraphKeys.GLOBAL_VARIABLES` collection에 속한) 모든 변수를 한 번에 초기화하기

```
session.run(tf.global_variables_initializer())
```

- 변수들의 그룹(collection)을 만들고, 각각의 변수들을 이 그룹에 소속시킬 수 있다.
- 기본적으로 모든 변수들은 default로 다음의 두 그룹에 소속된다.
  - `tf.GraphKeys.GLOBAL_VARIABLES`: variables that can be shared across multiple devices,
  - `tf.GraphKeys.TRAINABLE_VARIABLES`: variables for which TensorFlow will calculate gradients.
- 가령 어떤 변수를 **non-trainable**하게 하려면 다음과 같이 **LOCAL\_VARIABLES** 그룹에 넣거나

```
my_local = tf.get_variable("my_local", shape=(),  
                           collections=[tf.GraphKeys.LOCAL_VARIABLES])
```

혹은

```
my_non_trainable = tf.get_variable("my_non_trainable",  
                                   shape=(), trainable=False)
```

▶ Pre-trained model을 네트워크의 한 sub-network으로 사용할 경우 그 부분에 속한 변수들을 non-trainable하게 만든다.



- 자신만의 **collection**을 만들 수 있다. 만들어질 **collection**의 이름은 아무 **string**이나 되고, 먼저 변수를 생성한 후 `tf.add_to_collection` 함수로 **collection**에 넣어주면 된다.

```
my_local = tf.get_variable("my_local_var", [2, 3])  
tf.add_to_collection("my_collection_name", my_local)
```

- 특정 **collection**에 속한 모든 변수들의 리스트를 다음과 같이 **retrieve**한다.

```
tf.get_collection("my_collection_name")
```

▶ 어떤 이유로 네트워크를 서로 disconnected된 2개의 서브네트워크로 구성하는 경우가 있다. 이 경우 각 서브 네트워크에 속한 변수들을 별개의 **collection**으로 관리한다.

- 변수는 다른 `tf.Tensor`와 달리 단일 `session.run` 호출 맥락의 외부에 존재한다. 즉, 연속된 `Session.run()` 동안 값이 보존된다.

```
cool_numbers = tf.Variable([3.14159, 2.71828], tf.float32)
my_op = tf.assign(cool_numbers, cool_numbers+1)
sess = tf.Session()
sess.run(tf.global_variables_initializer())
print(sess.run(cool_numbers))
sess.run(my_op)
print(sess.run(cool_numbers))
sess.run(my_op)
print(sess.run(cool_numbers))
```

3번의 session run 동안 값이 유지된다.

- Tensorflow에서 변수는 프로그램에 의해서 다루어지는 지속적인 상태를 표현한다.

- 다수의 layer를 가진 네트워크를 생성할 때 종종 다음과 같이 하나의 layer를 생성하는 함수를 작성해 사용한다.

```
def dense_relu(input, in_size, out_size):  
    weight = tf.get_variable("weight", [in_size, out_size])  
    bias = tf.get_variable("bias", [out_size])  
    before_relu = tf.matmul(input, weight) + bias  
    return tf.nn.relu(before_relu)
```

- 이 함수는 두 개의 변수를 생성하면서 'weight'와 'bias'라는 이름을 사용한다. 하지만 여러 개의 layer를 구성하기 위해서 이 함수를 반복 호출하면 동일한 이름을 가진 변수가 만들어지므로 오류가 생긴다.

```
out1 = dense_relu(input, 10, 128)  
out2 = dense_relu(out1, 128, 258)    # Error !!!
```

- **dense\_relu** 함수를 서로 다른 **variable scope**로 호출하면 각각의 변수는 유일한 이름을 부여받는다.

```
with tf.variable_scope("layer1"):
    # Variables created here will be named "layer1/weight", "layer1/bias".
    relu1 = dense_relu(input, 10, 128)
with tf.variable_scope("layer2"):
    # Variables created here will be named "layer2/weight", "layer2/bias".
    relu2 = dense_relu(relu1, 128, 256)
```

- 만약 이미 정의된 변수를 재사용(**reuse**)하고 싶으면 동일한 **scope**를 사용하고 **reuse=True**로 설정하면 된다.

```
with tf.variable_scope("model") as scope:
    relu1 = dense_relu(input, 10, 128)
with tf.variable_scope(scope, reuse=True):
    relu2 = dense_relu(relu1, 128, 256)
```

▶ 변수를 재사용하여 공유하는 것은 신경망에서 드문 일은 아니다. 대표적인 예는 CNN이다.

- 그래프는 **placeholder**를 통해서 외부로부터 입력을 받을 수 있다.

```
x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)
z = x + y
print(sess.run(z, feed_dict={x: 3, y: 4.5}))
print(sess.run(z, feed_dict={x: [1, 3], y: [2, 4]}))
```

← shape이 지정되지 않아도 된다.

- **결과**

```
7.5
[ 3.  7.]
```

- Tensorflow는 다양한 loss 함수를 제공한다.

```
loss = tf.losses.mean_squared_error(labels=y_true, predictions=y_pred)
```

```
l2_loss = tf.nn.l2_loss(x-x_hat) # half of L2 norm of t given by sum(t**2)/2
```

```
ce_loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(  
    logits=y_hat, labels=y))
```

- TensorFlow는 다양한 표준적인 최적화 알고리즘들을 제공한다.
- 가장 기본적인 것은 `tf.train.GradientDescentOptimizer`이다.  

```
optimizer = tf.train.GradientDescentOptimizer(0.01)  
train = optimizer.minimize(loss)
```
- Adam optimizer 역시 일반적으로 선호되는 알고리즘이다.  

```
optimizer = tf.train.AdamOptimizer()  
train = optimizer.minimize(loss)
```
- 참고: <http://runder.io/optimizing-gradient-descent/>



## tf.train.Saver: 학습된 변수값 저장하기

신경망의 구조

weights + biases



- 트레이닝된 신경망은 “**그래프**” + “**변수의 값**”이라고 볼 수 있다.
- `tf.train.Saver` 클래스는 학습된 **변수들의 값**을 **저장**하고 **다시 불러오는** 기능을 제공
- 변수들의 값은 **binary checkpoint** 파일들로 저장됨. 이 파일에는 변수의 “**이름**”과 “**값**”이 **dictionary**의 형태로 저장됨
- 이미 **training**된 신경망을 다시 불러와서 추가 학습을 지속하거나, **prediction**용으로 사용할 수 있다.
- 변수의 값과 함께 “**그래프**”까지 저장하려면 **SavedModel**을 사용 (이건 나중에...)

v1과 v2는 변수의 이름이 아니고  
변수를 액세스하는 handle 정도로 생각하면 된다.

v1과 v2가 변수 이름이다.

0으로 초기화하는 initializer이다.

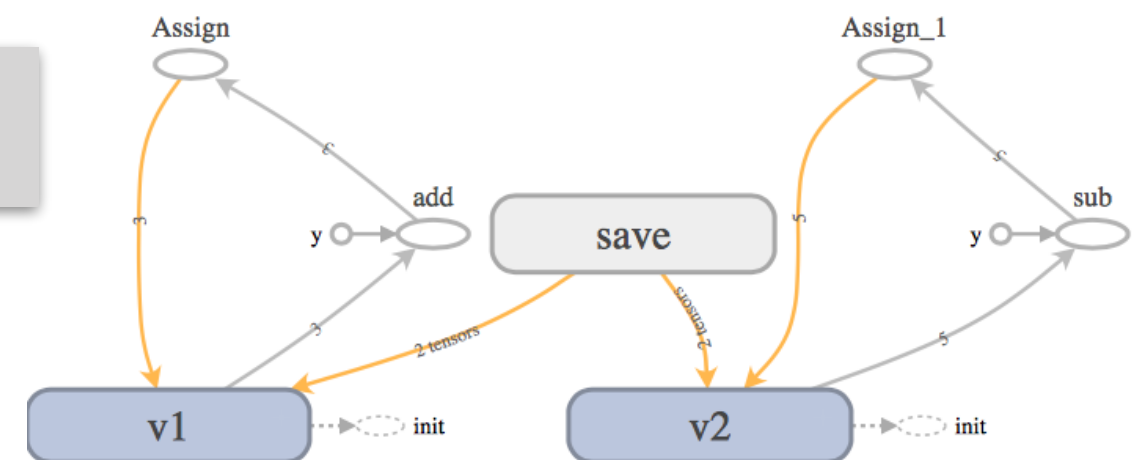
```
# Create some variables.
v1 = tf.get_variable("v1", shape=[3], initializer = tf.zeros_initializer)
v2 = tf.get_variable("v2", shape=[5], initializer = tf.zeros_initializer)
```

```
inc_v1 = v1.assign(v1+1)
dec_v2 = v2.assign(v2-1)
```

v1을 1증가하고 v2를 1 감소하는 연산이다.

```
init_op = tf.global_variables_initializer()
```

```
# Add ops to save and restore all the variables.
saver = tf.train.Saver()
```



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

```
    sess.run(init_op)
```











```
    # Do some work with the model.
```

```
    inc_v1.op.run() ← sess.run([inc_v1, dec_v2])와 동일하다.  
    dec_v2.op.run()
```

```
    # Save the variables to disk.
```

```
    save_path = saver.save(sess, "logs/model.ckpt")  
    print("Model saved in file: {}".format(save_path))
```

logs 디렉토리와  
model.ckpt 파일이 생성되었는지  
확인한다.

	checkpoint	
	events.out.tfevents.1516758739.Kwonui-iMac.local	
	model.ckpt.data-00000-of-00001	
	model.ckpt.index	
	model.ckpt.meta	

```
# Create some variables.
```

```
v1 = tf.get_variable("v1", shape=[3])  
v2 = tf.get_variable("v2", shape=[5])
```

← 저장된 변수의 값을 읽어올 것이므로 initilaizer는 필요없지만 있어도 상관없다.(실제로는 default initializer가 있다.)

```
# Add ops to save and restore all the variables.
```

```
saver = tf.train.Saver()
```

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

```
    saver.restore(sess, "logs/model.ckpt")  
    print("Model restored.")
```

← model.ckpt 파일로 부터 저장된 변수의 값을 읽어 온다.

```
# Check the values of the variables
```

```
print("v1 : {}".format(v1.eval()))  
print("v2 : {}".format(v2.eval()))
```

← v1.eval()과 v2.eval() 대신  
v1\_val, v2\_val = sess.run([v1, v2])  
으로 두 변수의 값을 읽어온 후 출력해도 된다.

```
Model restored.  
v1 : [ 1.  1.  1.]  
v2 : [-1. -1. -1. -1. -1.]
```